A Wireless Link to a Six Degree of Freedom Inertial Tracker

by

John J. Rodkin

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

January 26, 1996

Author ______________________________

Department of Electrical Engineering and Computer Science
January 26, 1996

Certified by [illegible]

Nathaniel Durlach
Thesis Supervisor

Accepted by [illegible]

F.R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Eng.

A Wireless Link to a Six Degree of Freedom Inertial Tracker

by

John J. Rodkin

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.
January 26, 1996

Abstract

The purpose of this thesis is to begin extending the MIT inertial tracker for wireless operation. Many applications exist for a wireless, sourceless tracker, and as head mounted displays and other synthetic environment systems go wireless, more applications for wireless trackers will arise. The importance of the work is discussed in detail, and the design and implementation of two possible serial links to the MIT inertial tracker is described. These links were implemented and analyzed, and the test results are discussed. These results focus on bandwidth concerns because the serial link will soon utilize a standard wireless serial transceiver pair to make MIT's inertial tracker an excellent choice for applications requiring a high-performance, wireless, sourceless tracker in a small, lightweight package.

Thesis Supervisor: Nathaniel Durlach
Title: Senior Research Scientist

Acknowledgments

I never could have accomplished this work or my thesis on my own, so credit should be given where it is due.

I would like to thank my apartment-mate for creating an atmosphere that encouraged me to work late and enabled me to enjoy my time in the lab much more than I should have. My neighbors Jodi Goldstein and Carolyn Waldman helped me maintain sanity throughout. The brothers at Sigma Chi gave me a place to work, a bed in which to rest, and lifelong friendships for which I am truly grateful. Nat Durlach, my thesis supervisor, allowed and encouraged my independence throughout the project.

I would like to thank Eric Foxlin, my mentor, coworker, and friend, most of all. He was always there to guide me, both technically and personally, and most of this work should bear his name instead of mine.

Contents

Chapter 1

Introduction

1.1 An Introduction to Virtual Reality

The Committee on Virtual Reality Research and Development was established in 1992 by the Federal government to "recommend a national research and development agenda in the area of virtual reality." At that time, media and scientific speculation on the future of virtual reality systems was rampant and that speculation has continued: "between 1992 and 1994, roughly 12 new books have been published, 4 new journals or magazines have been started, and 100 new articles have been published on the topic of virtual reality" (Durlach and Mavor, p. 14). It is clear that virtual reality systems have captured the interest of scientists and the imagination of the media worldwide.

Why has virtual reality become such a trend? To answer this question, we must strain to see the scientific foundation that supports all of the "hype" and misinformation that daily circulates through the press. We must cast aside artists' sketches of the "shopping mall of the 21st century" and writers' accounts of adventures that we can take from the comfort of our own living rooms. It is wonderful to dream, to imagine interactive "virtumalls" and to contemplate "traveling" to faraway planets, but such fantasies do not fuel the research and development of new technologies so distant. Incremental progress can and is being made toward these futuristic dreams, and these advances have helped make many virtual reality systems useful today.

Before mentioning these useful applications, some terminology should be clarified. The media generally uses "virtual reality" to describe the set of systems in which a "human operator is transported into a new interactive environment by means of devices that display signals to the operator's sense organs and devices that sense various actions of the operator" (Durlach and Mavor, p. 13). The Committee on Virtual Reality Research and Development uses a slightly different terminology, however, and this document will hereafter conform to the Committee vocabulary. "Synthetic environments" is the inclusive term synonymous with the media's use of "virtual reality". Synthetic environments include two types of systems: virtual reality or virtual environment systems that allow a human user to interact with a computer-generated virtual world, and teleoperator systems in which a human user's interaction with the physical world is mediated by some set of electromechanical sensors and actuators (Durlach and Mavor). A virtual reality system is commonly associated with video games and entertainment while a teleoperator system is more useful as a means of performing hazardous tasks or of remote exploration. Hybrids of these two synthetic environment types exist and are referred to commonly as "augmented reality" systems.

These hybrids and both types of pure synthetic environments find many applications today with many more uses visible on the horizon. Applications in design, manufacturing, marketing, medicine, training, handling of hazardous materials, education, telecommunications, and entertainment are discussed in Chapter 12 of Durlach and

Mavor, and the reader is referred there for a detailed examination of synthetic environment applications that are driving today's research and development efforts. It is likely that the most prevalent short-term commercial efforts in virtual environments will be made for entertainment purposes with commercial applications in other virtual environment systems realizable only in the long-term time frame (Durlach and Mavor, p. 381). Teleoperation, however, is already applied widely, but its lack of demonstrated commercial value has limited its funding for further research and development (Durlach and Mavor, p. 381). Many scientific and economic hurdles stand between the applications of synthetic environments in the near future and the "virtumalls" and "teletravel centers" of the media, but these obstacles can be overcome and synthetic environments can eventually have a significant impact on our daily lives.

1.2 An Introduction To Tracking Technologies

It is clear at this point that all synthetic environments require a set of sensors to report action in a synthetic environment. These sensors enable a computer-generated virtual environment to respond to the actions of a human user and allow a teleoperator system to correctly report information about the telerobot's interaction with the real environment. In either case, the sensors play an important role in the system, and the accuracy and resolution of those sensors could be a significant limiting factor in performance.

Consider a virtual environment video game that is unable to accurately sense a player's motion on the battlefield; such a deficiency could defeat the purpose of a virtual reality game altogether. Likewise, consider a system that can accurately sense the player's motion only in sizable discrete steps. The poor resolution of this system could cause motion sickness and confusion quite quickly. Now imagine a teleoperator system that fails to report movement accurately or that lacks sufficient resolution. Such a system may be useful for remote exploration or for lifting heavy objects, but it would certainly fail during surgery.

It is important to mention that the accuracy and resolution of a sensor are not the only parameters in the accuracy and resolution of an entire tracking system. A sensor simply relays raw information to some processor that then calculates movement from that data. Small inaccuracies and low resolution in the raw data can be improved by data conditioning - that is, the data can be averaged or operated on in some more clever way to achieve tracking system performance better than the performance of an individual sensor. It is clear, though, that sensor performance can be a limiting factor in overall tracker performance.

Sensor performance, then, can be a tradeoff parameter in a tracking system. It may be desirable to use a lower grade sensor with very complex data conditioning algorithms to achieve the desired system performance, or it may be preferable to use an excellent sensor with little or no operation on the raw data. Whatever the tradeoff, performance of a tracker as a whole significantly impacts the performance of the entire synthetic environment system.

Sensor performance is closely linked to the tracker methodology. Five methodologies - mechanical, magnetic, acoustic, optical, and inertial - are commonly used today, and each is discussed below.

1.2.1 Mechanical Tracking

Mechanical tracking is the most straightforward tracking method, involving a direct physical connection between the tracked point and a reference point. The displacement between the two points can then be calculated using the displacement of joints within the physical connection. Mechanical tracking is fast, accurate, and cheap, but the direct physical connection imposes many limitations. The most notable limitations include severely limited range and a cumbersome mechanical setup that can interfere with action in a synthetic environment. For applications that require cheap, accurate tracking in a small volume, mechanical tracking is an acceptable choice.

1.2.2 Magnetic Tracking

Magnetic trackers are composed of a sensor containing three orthogonal coils and an emitter that generates three orthogonal magnetic fields. Position and orientation can then be calculated using the nine induced currents (a current is induced in each coil by each field). For a discussion of the mathematics see Raab, Blood, Steiner, and Jones, 1979.

Magnetic trackers have several advantages. These include a small, lightweight sensor, good accuracy and resolution when the sensor is near the emitter, and relatively low cost for the performance achieved. The magnetic method, though, also contains inherent limitations. Nearby ferromagnetic objects distort the orthogonal magnetic fields and degrade performance. Performance also degrades as the sensor is moved farther from the magnetic field emitter. Despite these limitations, magnetic trackers continue to be the most common trackers used in today's virtual reality applications.

1.2.3 Acoustic Tracking

Acoustic tracking systems utilize acoustic emitters and sensors to determine either the time of flight or the phase shift of an acoustic wave traveling from a fixed reference to an unknown point. From these data, an acoustic system calculates distances and uses triangulation to compute position. An acoustic system can not measure orientation directly. Instead, the position of two or more unknown points (held fixed in relation to each other) must be tracked so that orientations can be determined using position measurements.

Acoustic trackers overcome some of the inherent limitations of magnetic trackers, but impose restrictions of their own. Obviously, ferromagnetic objects do not interfere with acoustic trackers, but any object placed in the line of sight between emitters and receivers will adversely affect system performance. Distance does not degrade accuracy or resolution for an acoustic tracker (provided the distances remain within the range of the system), but increased distance does slow the data rate for time of flight systems. Acoustic phase systems track only relative change in position and are, therefore, hampered by errors that accumulate between zeroings. In addition, sources that produce acoustic waves in the frequency used by an acoustic tracker will cause the tracker to confuse its acoustic emissions and the offending source, and gross position errors will result.

1.2.4 Optical Tracking

Optical tracking encompasses a wide range of methods that use light to track objects. Optical trackers can use light emitters and sensors that measure angles for triangulation, or an optical tracker can involve pattern recognition, rangefinding with lasers, or analysis of digitized video and still photography. In general, optical trackers are very accurate and fast, but also expensive. Because of the expense, optical tracking technology has been less widely used than any of the previously mentioned methods. These optical trackers may develop into cost-effective high performance systems as imaging systems continue to become cheaper and more powerful.

1.3 Inertial Tracking

Inertial tracking requires sensing acceleration and angular rate and then integrating that data to derive position and orientation. This method has been used effectively in aircraft navigation systems for quite some time, but the size and cost of the required sensors have made inertial tracking unattractive for synthetic environment applications. However, "the use of accelerometers or of angular rate sensors for motion tracking is becoming increasingly attractive because of advances in sensor design" (Durlach and Mavor, p. 200). As inertial sensors decrease in size and cost and increase in accuracy and resolution, they become more useful for synthetic environment systems.

A major advantage of inertial tracking systems is their inherent sourceless operation. All of the previously mentioned tracking methods (except mechanical tracking) require both a source (to produce magnetic fields, acoustic waves, or light) and a sensor. Systems that require a source suffer from inherent range limitations and often from line of sight limitations as well. Although magnetic trackers overcome the line of sight difficulty, they do have difficulty with a propinquity of ferromagnetic objects. A sourceless inertial tracker overcomes all of these problems -- no line of sight restrictions, no inherent range limitations, and no problems with nearby objects of any material.

1.3.1 The MIT Inertial Tracker

Unfortunately, stand-alone inertial tracking systems drift, unable to accurately derive absolute position and orientation. To compensate for this drift, "either an inertial package must periodically be returned to some home position for offset correction, or it must be used in conjunction with some other (possibly coarse) position sensor and an appropriate method of data fusion" (Durlach and Mavor, pp. 200-201).

Using this alternate sensor method, "an inertial orientation tracker has been built at MIT using triaxial angular rate sensors with gravimetric tilt sensors and a fluxgate compass for drift compensation (Foxlin and Durlach, 1994)" (Durlach and Mavor, p. 201). This tracker has recently been hybridized with an acoustic tracker to track translation as well as orientation. With three degrees of freedom for orientation and three more degrees of freedom for translation, the tracker has achieved six degree of freedom tracking using a sensor block whose volume is approximately one cubic inch.

This sensor block is connected via cable to a processor that integrates the data received by the sensor and that sends necessary commands to the sensor. Unfortunately, the cabling imposes some limitations on the system. The obvious limitation concerns the range of the tracker, with the range limited to the length of the cable. Tangling and knotting of the cable also slightly restrict the motion of the user. Eliminating these frustrating limitations imposed by the cable would make an already attractive six degree of freedom tracker even more desirable.

The following block diagram illustrates the MIT hybrid system. The dashed box includes the acoustic tracker that is included for six degree of freedom operation -- for applications that track orientation but not translation, the dashed box is excluded. The bold lines indicate the limiting cables in the system; because the acoustic system tracks a wireless emitter with cabled receivers, the wired connections are fixed and do not impose limitations. In addition, the host computer is not part of the hybrid system. Instead, the host computer is usually a large virtual reality engine that controls the entire system.

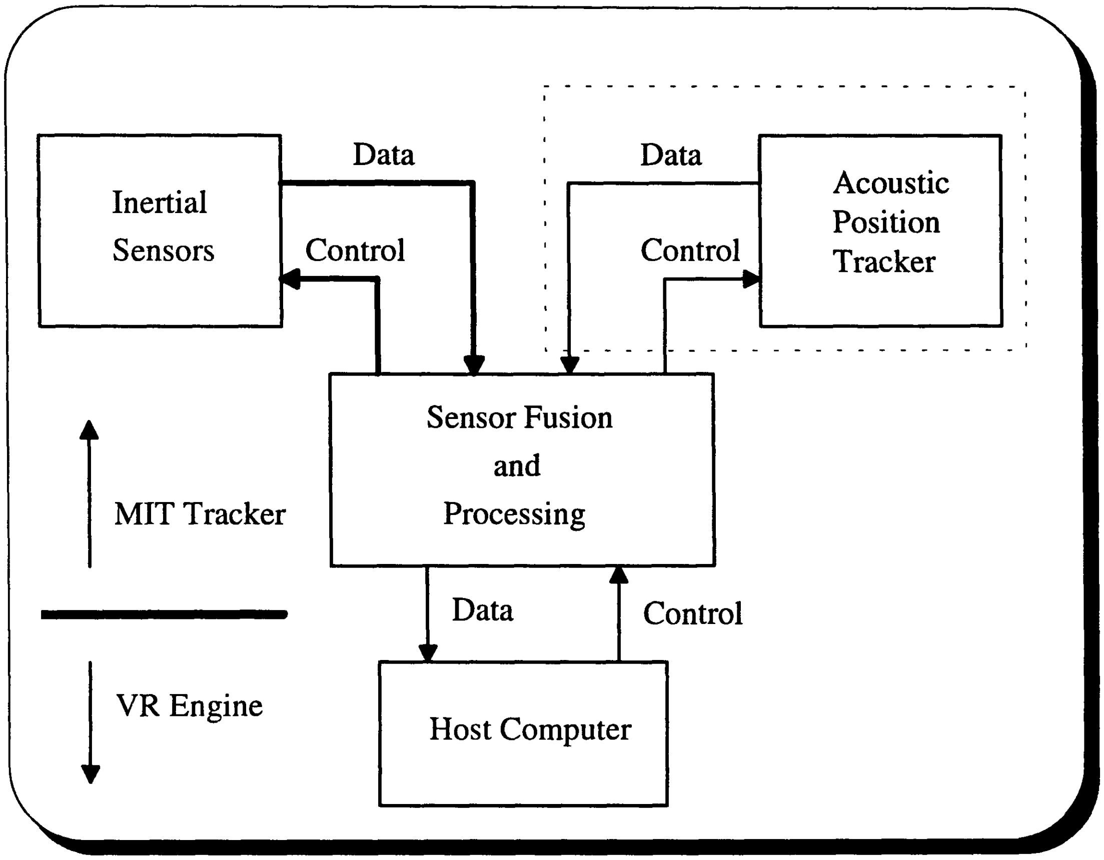


Figure 1-1: A block diagram of the MIT inertial tracker.

1.4 Improving the MIT Inertial Tracker: Adding a Wireless Link

It is clear that a wireless link between the sensor block and the processing block would make the aforementioned six degree of freedom tracker more desirable. Increased range would make the system more attractive to teleoperator systems and to augmented reality systems (less so for pure virtual reality systems) and the elimination of logistic problems created by the cable would increase tracker desirability in all synthetic environment systems.

This link could be established using either infrared or radio frequency (RF) technology. The method used is less important than the design parameters imposed by a potentially commercial system. That is, the system must be small, lightweight, safe, and inexpensive to compete in the marketplace. The end user desires a high level of performance and a transparent link - the proposed wireless link should solve the problems created by the cable in the current system without creating any new difficulties in the system.

1.4.1 Thesis Scope

The goal of the present project is simple: design the cable out of the MIT inertial tracking system. Designing a custom set of transceivers is one possible method of reaching this goal. Such a method may not be the most far-sighted, however, because the system continues to evolve and evolving the transceivers in parallel could be an arduous task. Another avenue from which to approach the goal is to reengineer the existing interfaces to the system so that any of several standard transceivers can solve the wireless link problem. Such reengineering is the focus of this thesis.

Additions for the existing system were designed so that they would not only allow standard transceivers to make the wireless link but also seamlessly overlay the existing high-performance inertial tracker. Chapter 3 discusses the design and implementation of the new interfaces, and Chapter 4 provides analysis of test results and suggests the next steps that should be taken. Before undertaking the project, though, it is necessary to understand some of the benefits of a successful design, so Chapter 2 discusses some important potential applications of the wireless, sourceless MIT inertial tracker.

Chapter 2

Potential Applications

2.1 Potential Applications for the MIT Inertial Tracker

The following potential applications are outlined to show the usefulness of the sourceless and wireless MIT inertial tracker. The list of applications is by no means exhaustive and is not meant to guide further development efforts. Instead, the list of potential applications should serve as a highlight of the tracker's capabilities and the broad range of industries in which those capabilities could be utilized.

2.1 Applications for a Sourceless Tracker

The MIT sourceless inertial tracker does not suffer from line of sight limitations and has no range restrictions when sensing orientation only (the acoustic-inertial hybrid that senses orientation and translation has moderate line of sight and range restrictions imposed by the acoustic portion of the system). Remarkably, the inertial system overcomes the limitations of other systems without a degradation in performance. So, applications that require high performance, long range, and possibly occluded operation are ideal for the sourceless MIT inertial tracker.

2.1.1 Head Tracking in Virtual Environments

Today's primary application for the MIT inertial tracker is head tracking for virtual reality systems. For applications that require orientation but not translation (i.e. most video games and simulators) the MIT inertial tracker offers high performance, virtually unlimited range, and no degradation near noise sources or ferromagnetic objects. When translational data is required, the MIT hybrid system offers the same high performance and immunity to ferromagnetic objects, but is limited to a range of approximately five meters (as opposed to 1-2 meters for popular magnetic trackers), is degraded by noise sources in the acoustic frequencies used by the tracker, and suffers from the line of sight difficulties of the acoustic tracker. Despite these limitations, the MIT acoustic-inertial hybrid can be effectively applied to head-tracking applications.

2.1.2 Telerobots

Telerobot applications require the tracking system to report the orientation and position of some remote robot to a human controller. The benefits of the MIT inertial tracker and the MIT acoustic-inertial hybrid mentioned in the previous section apply to telerobot applications as well. The long range of the MIT system should be emphasized because it is likely that the telerobot will operate in a sizable environment. In addition, immunity to ferromagnetic objects is obviously important for telerobot applications that use metallic materials to construct the telerobots.

2.2 Applications for a Wireless, Sourceless Tracker

Two specific potential applications for a wireless, sourceless tracker are discussed in this section, but certainly many more exist. Any application that requires high performance orientation tracking over extremely long ranges can utilize the MIT inertial tracker. Applications that need this performance from a small, inexpensive tracker are ideal for the MIT system.

2.2.1 Remotely Piloted Vehicles

Remotely piloted vehicles can be grouped with telerobots. The distinction made here uses remotely piloted vehicles to imply much longer range than a telerobot.

One can imagine the need for monitoring the orientation of the turret of a remotely piloted tank during combat. In addition, a remotely piloted aircraft would also need orientation monitoring. A Global Positioning System (GPS) receiver could be integrated with the inertial tracking system to provide a fairly accurate inertial navigation system (INS). Because multiple inertial trackers can be operated in a closed space without interfering with each other (all other tracking methods do self-interfere), it can be imagined that a remotely piloted helicopter could utilize an integrated GPS INS and then multiple inertial trackers to aim weapons. It should be emphasized that the MIT inertial tracker is useful for this application when a small size and low cost are desirable; bulkier and much more expensive inertial navigation systems are currently available that offer superior performance.

Remotely piloted commercial vehicles can benefit from the MIT inertial tracker as well. One can imagine using an inertial orientation sensor coupled with a camera or set of cameras on a remotely piloted vehicle to develop panoramic views of terrain. These panoramas could be used for search and rescue operations, exploration, and entertainment. Certainly many more applications exist; any task that requires human operation of a remote machine is a prime candidate for a sourceless, wireless tracker.

2.2.3 Surveying

Surveying is another potential application of a sourceless, wireless tracker. Consider a system composed of a laser rangefinder on a tripod with a reflector placed some distance away. A survey team outfitted with the MIT inertial tracker could quickly setup their laser and reflector and take several measurements, rather than leveling the tripod and measuring the angles between the laser and reflector precisely for each measurement. All of the angle measurement would be performed by the tracker, and compensation could be made for an unlevel tripod. In some cases, the laser rangefinder could be aimed at a known point (i.e. a nearby building) and the inertial tracker could instantly measure all of the angles necessary to fix the point of the tripod. A surveyor equipped with a wireless, sourceless tracker could quickly move his tripod from point to point and aim his rangefinder at a fixed point while the tracker measured all angles necessary and downloaded them to the surveyor's nearby van.

Chapter 3

The Design and Implementation of a Wireless Tracker

3.1 The System

A wireless tracker must contain only a wireless data and control link and a sensor or block of sensors that produce measurements from which position and orientation can be computed. The sensor fusion and processing can be performed on a distinct processor after the raw data have been received from the sensors (figure 3-1), or the orientation and position can be computed between the sensor block and data transmitter (figure 3.2). This tradeoff is discussed in more detail in Section 4.4. For now it is sufficient to recognize that a wireless tracking system can be represented by three functions in a block diagram - data collection (the sensor block), data transmission (the wireless link), and computation.

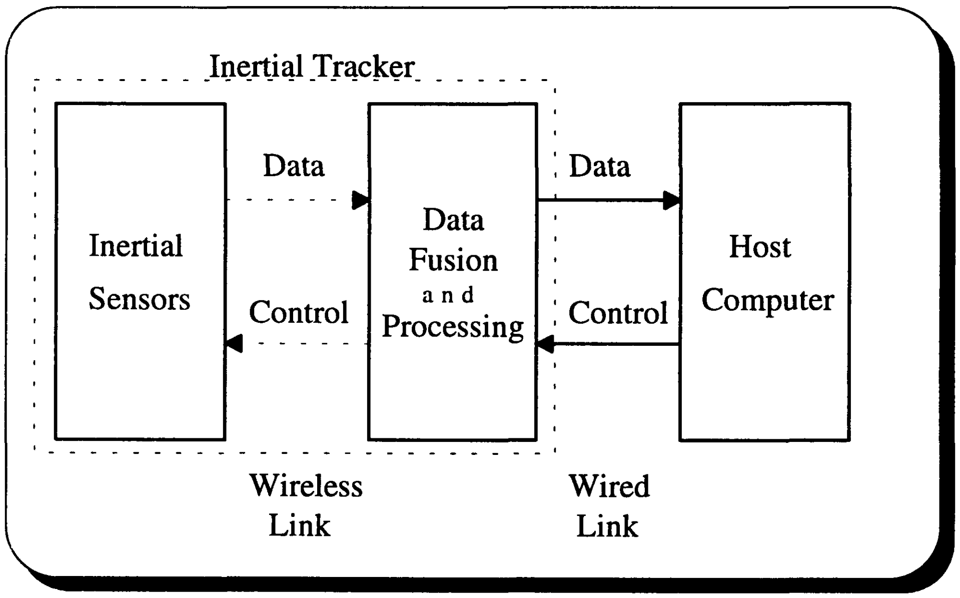


Figure 3-1: The wireless link before computation.

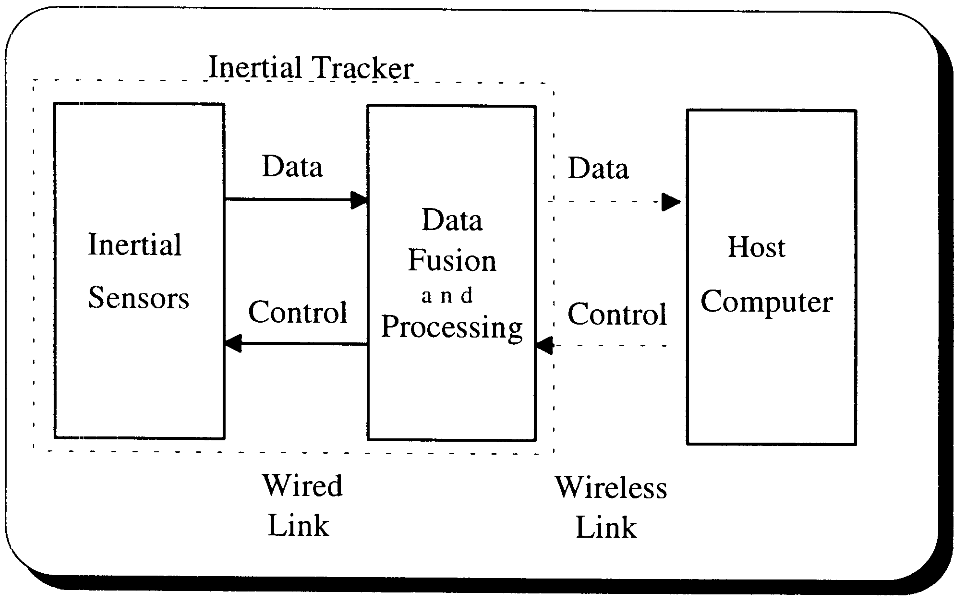


Figure 3-2: The wireless link after computation.

The sensor block in a wireless tracker should be similar to the sensor block of a wired tracker using the same tracking technology. It is the data from this block that is used to calculate position and orientation, and it is assumed that the performance of a wireless tracker should approach the performance of a wired tracker as closely as possible. Therefore, the technology versus cost trades that are made for a wired tracker should have the same outcomes for a wireless version. Perhaps we are willing to accept higher cost or lower performance in a wireless tracker, but we would certainly prefer to maintain the standards we hold for wired trackers. For the proposed design, the sensor block in the wireless version is identical to the wired sensor block.

It is assumed that the data transmission block is a likely bottleneck for a wireless tracking system. Certainly if cost is a factor in the system, the wireless link must be the bottleneck. Otherwise, we can purchase an extremely low cost, low baud rate wireless link to use for the system. In the limit, we have no wireless transmission link and no cost, but we are then forced to collocate the host computer, data fusion and processing functionality, and the sensor block. In another limit, we can buy an extremely expensive wireless link that achieves baud rates as high as we like over arbitrary transmission distances with low bit error rates. This system is very expensive, and it is uninteresting. In the proposed system, it is assumed that both low cost and high bandwidth are similarly desirable, so the wireless transmission link is an important and constraining block.

Computation in a wireless tracker should be similar to the computation in a wired tracker, assuming similar precision and accuracy versus speed tradeoffs. Because the computation block is basically the same in both wired and wireless trackers, its contents are less interesting for the current project than its placement. If most computation can be performed before the wireless transmission link, then we can reduce bandwidth and cost. However, we assume that fast computation power is more available and much cheaper at the host computer than it is at the sensor side of the transmission link. Otherwise, we would move the entire host computer and data fusion and processing functionality to the sensor block and we would eliminate the need for a wireless link at all. The placement of this computation block is discussed in more detail in Chapter 4.

The overall sourceless, wireless system is based on the MIT inertial tracker developed by Eric Foxlin. Though the remainder of the discussion focuses specifically on this tracking system, the concepts can be generalized to include other inertial tracking systems and other tracking methodologies as well.

3.1.1 System Design Goals

As mentioned in Section 3.1, the primary goal of a wireless tracking system is to approach the same performance and functionality of a wired tracking system using similar tracking technology. In addition, the cost of the system should not be prohibitive, and the size should be manageable. It is also hoped that the transmission link will not introduce a bottleneck that severely hampers the performance of the MIT inertial tracker. Moreover, the whole system should operate on a battery for an acceptable length of time. Please see Table 3-1 for a complete list of design goals.

Characteristic	Current System	Proposed Goal for Wireless System
Range	length of cable for orientation only; 15 ft. for orientation plus position	30 feet without line of sight.
Battery Life	Not Applicable	2 hours
Update Rate	500 Hz / Total number of sensor blocks	100 Hz / total number of sensor blocks
Price	$9,200	$10,000
Size of wearable components	1.25 in. x 1 in. x .91 in.	2.5 in x 2.5 in x 1 in.

Table 3-1: Design goals.

3.1.2 Design Constraints

Constraints for this design come largely from the need to integrate the new system with the existing MIT inertial tracking system. The proposed hardware and software must overlay the existing hardware and software platforms seamlessly. In addition, the proposed system must operate through an available RS-232 serial port, allowing a maximum of 115 Kbaud transmission rate.

This choice to use an existing serial port was made to reduce cost and complexity. Higher bandwidths (and therefore, higher performance) could be obtained by combining a high bandwidth transmission link with a custom card that plugs into the bus of the data fusion and processing unit. This method increases complexity and cost and was not investigated.

3.2 The Hardware

3.2.1 An Overview

The following Figure 3-3 illustrates a block diagram of the hardware for a wireless link to the MIT inertial tracker. The blocks are discussed in the remainder of Chapter 3.
A variety of digital hardware was utilized to overlay the proposed wireless tracker onto the existing system. This hardware included an RS-232 interface between the data processing unit and the microprocessor that controls the sensor block and an interface between the microprocessor and

the sensor block. In addition, further interface circuitry may be necessary to utilize an off-the-shelf wireless serial link. This circuitry should not involve much more than a wired connector and socket because most wireless serial links are designed to plug easily into existing wired serial connections.

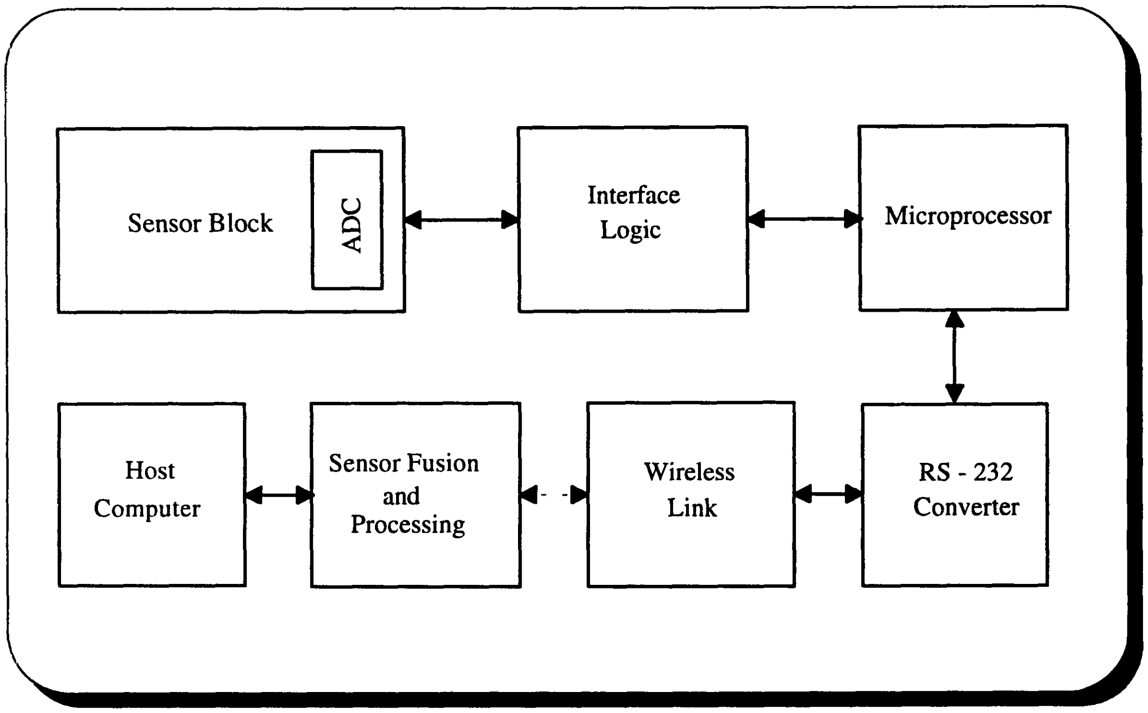


Figure 3-3: A block diagram of the serial link.

3.2.2 Hardware in the Sensor Block

For the present project, the only relevant hardware in the sensor block of MIT's inertial tracker is an analog to digital converter (ADC). The converter reads an eight bit serial control word that specifies channel and output format during the first eight clock cycles of its twelve cycle output. The ADC has a 10 microsecond conversion time and runs on a 4 MHz clock, so the total cycle time is 13 microseconds, allowing a maximum conversion rate of approximately 77,000 channels per second (the current MIT system uses the converter to convert approximately 4000 analog values per second). In the proposed system, the converter is selected and driven directly by the microprocessor and interface hardware.

3.2.3 The Microprocessor and its Interface

The proposed system utilizes a Zilog Z180 microprocessor running at 18.432 MHz on a Zworld SmartCore board. The SmartCore provides 512K of read only memory (ROM), a watchdog timer to recover from system lock-ups, and a 40 pin interface to help reduce the layout complexity of nearby interface circuitry. See Appendix A for the schematics of the Zworld SmartCore.

Zworld's suggested interface to an analog to digital converter (ADC) is shown in Figure 3-4. This interface uses an 8-bit addressable latch (74HC259) to decode the least significant three address bits into separate data, chip select, and clock lines. In addition, an octal buffer /

line driver (74LS244) is used to gate the ADC data output stream onto the three-state data bus. See Appendix B.1 for the microprocessor software used to drive the interface in Figure 3-4.

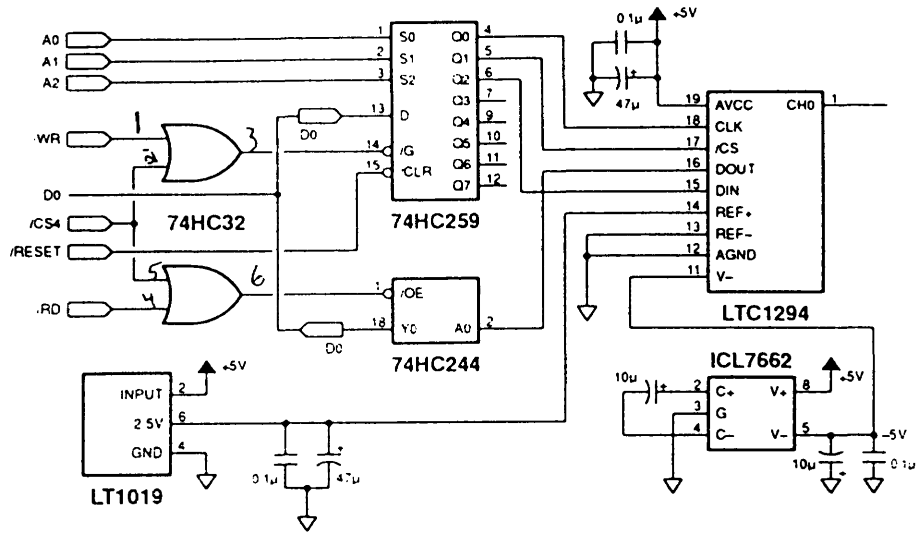


Figure 3-4: Zworld's suggested interface.

The suggested interface was constructed and evaluated, and it was found to have severe limitations. Because the microprocessor directly drives the clock for the ADC, each bit of output data requires at least two output commands from the microprocessor to toggle the clock. For a twelve bit output word, at least 24 microprocessor output commands are required. In addition, the sample interface only uses the least significant bit of the data bus, thereby requiring the microprocessor to internally shift the received data to make room for incoming bits. These limitations greatly impede the cycle time of the ADC - allowing approximately 400 conversions per second with a 115 Kbaud serial connection. (It takes three serial words of 10 bits each to transmit two 12 bit conversion results with overhead. A 400 channel per second conversion rate implies 6000 bits per second transmitted across the serial line, for a utilization of 5%. A 115 Kbaud channel perfectly utilized can theoretically support 7680 conversions per second with no data compression.)

An interface circuit that overcomes these limitations was designed and constructed and is shown in Figure 3-5. The 74LS259 now has its data input wired to a logic high, so any line of the multiplexer that is selected by the three least significant address bits is asserted to a logic high. Logic lows are set on all lines of the multiplexer by forcing the third chip select line on the microprocessor to a logic low. The 74HC259 is used to select modes for the 8-bit shift / storage registers (74LS299) and to provide the clock pulse necessary to load the resgisters with the 8 bit value present on the data bus.

A 4 MHz crystal oscillator is used to eliminate the need of the microprocessor to directly drive the ADC clock. Because we need bursts of twelve clock pulses separated by enough time to read the data from the ADC and set up for the next conversion, a presettable 4-bit binary up / down counter (74LS193) is used with additional logic gates to provide a 12 count framing pulse that is used to regulate the clock input to the ADC. In addition, the 74LS299's enable the microprocessor to read 8-bits of each conversion result with one read operation, reducing the internal shifting necessary to form a 12 bit result. Please see Appendix B.2 for the driver software of the circuit in Figure 3-5.

With this configuration, a single microprocessor and interface hardware circuit can drive up to four sensor blocks (this limitation is imposed by the number of available chip select lines on the microprocessor) with a maximum three quarters degradation in performance. The performance is degraded most when the microprocessor independently controls each sensor block. The degradation can be minimized with additional shift registers -- if each sensor block has its own set of shift registers then the analog to digital conversions can occur in parallel. In addition, if each sensor block operates on the same channel at any given time, the control words can be written simultaneously to reduce the number of required commands on the microprocessor.

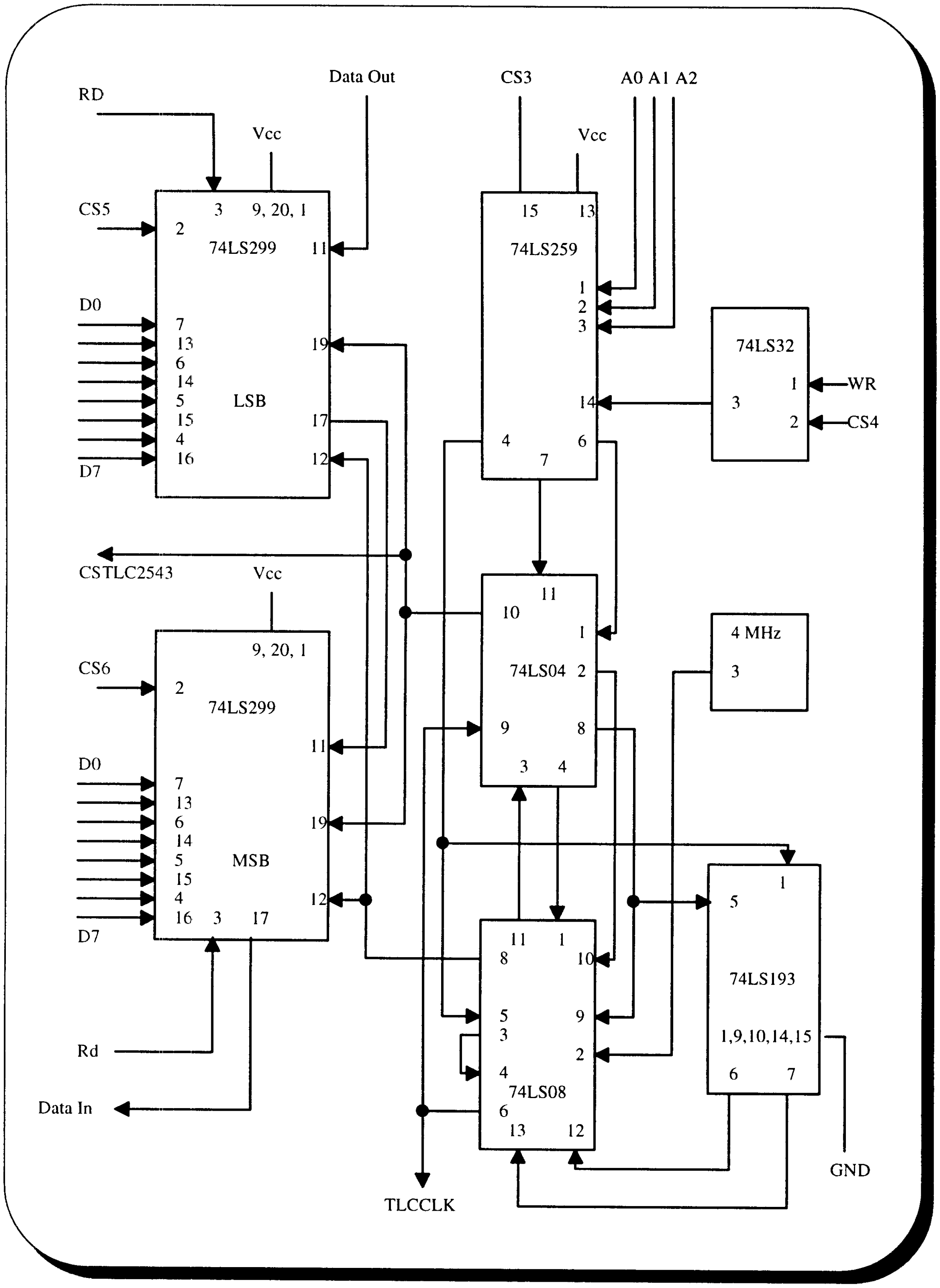


Figure 3-5: An improved interface circuit.

As we can see, many of the limitations of the suggested circuit are eliminated. The microprocessor now requires six output commands and two input commands to program and read each conversion instead of the 51 commands required previously to toggle the clock and read bits individually. The ADC is now operated at its maximum allowable clock frequency of 4 MHz. The system can now perform approximately 2800 analog to digital conversions per second, a sevenfold improvement over the first revision. Including the overhead start and stop bits, the 115 Kbaud serial connection is now 36% utilized.

Why is the serial connection only 36% utilized? It is certainly preferable to utilize all of the available bandwidth. In the present setup, though, the microprocessor must read the data bus twice for each conversion before loading the bus with the next channel to be converted. During these read operations and the subsequent write to prepare the system for the next conversion, the serial connection is idle. In addition, when the microprocessor scans a block of channels, it stores all of the conversion results until it is ready to send all of the data back to the host computer and the serial link is idle during this pause. This storage allows the microprocessor to format the data efficiently and reduce the overall bandwidth requirements for a given number of conversions.

3.2.4 Hardware Interface to the Data Fusion and Processing Unit

The data fusion and processing unit must have an available serial port (when computation is performed before transmission, the host computer must have an available port), but no additional hardware is required. The current MIT inertial tracker uses a custom card that connects the sensor block to the data processor's 8-bit Industry Standard Architecture (ISA) data bus. This custom card allows extremely fast operation of the ADC within the sensor block, but the speed is unnecessary when limited by a 115 Kbaud serial connection.

The current MIT system already uses both serial ports available on the existing controller card. The first port is used to connect to the VSCOPE ultrasonic positioning system, and the second is a data connection to a host computer. A second controller card must be added and configured to allow the operation of three serial ports.

3.3 The Software

Software for the proposed system has two distinct parts - the code burned on the EPROM that controls the microprocessor, and the code inserted into the existing data fusion and processing unit. The implementation of both of these parts is discussed below, and the complete listing of the code is given in Appendices B and C.

3.3.1 Microprocessor Software

The code on the microprocessor receives a single control byte from the data fusion and processing unit. At the time of this writing, only two control commands have been implemented -- a channel scan command and a single conversion command. To scan a block of channels, the microprocessor must receive a byte that contains four leading zeros and then the channel up to which to scan. A single conversion command byte contains four leading ones and then the channel to be converted.

Please see Appendix B for the complete code implemented on the microprocessor. Each function is described in detail in the appendix.

Some additions should be made to the microprocessor code to allow better control of the system. A function that remaps the channels should be implemented, as should another function that changes the data rate of the serial port on the SmartCore. In addition, functions must be implemented to control lines for the sensor block that are not tied to the ADC.

3.3.2 Other Software

In addition to the software implemented on the microprocessor, the software running on the data fusion and processing unit was modified. Please see Appendix C for the modifications.

It is clear that some software is needed to send commands to the microprocessor and receive data from the microprocessor. The software in Appendix C serves this purpose, and it accomplishes other tasks for the inertial tracker that are unrelated to the present discussion.

Chapter 4

Data Analysis and Conclusions

4.1 Results

It is important to recall that the present serial system has been designed for use with existing wireless serial connections. Most of the wireless serial links presently available for a reasonable cost operate at baud rates of 38,400 or less. Performance requiring more than the easily available bandwidth would be degraded when operated with a less expensive wireless link. It is important, then, to improve the cycle time of conversions without increasing the bandwidth required to transmit the results to the host computer. It may also be desirable to accept a small performance degradation if the bandwidth requirements are greatly reduced by the change.

The following Table 4-1 and Figures 4-1 and 4-2 summarize useful data that help suggest an optimal bandwidth for the present system. For 8000 conversions, the total data sent by the ADC is 120,000 bits with 10,000 bits required to control the microprocessor.

Baud Rate	Seconds per 8000 conversions (microprocessor code C.2)	Percentage of utilized bandwidth
2,400	54.8	98.8
9,600	14.8	91.2
19,200	8	84.6
38,400	4.6	73.6
57,600	3.5	64.4
76,800	2.7	62.7
96,000	2.6	52.1
115,200	2.6	43.4

Table 4-1: Bandwidth utilization.

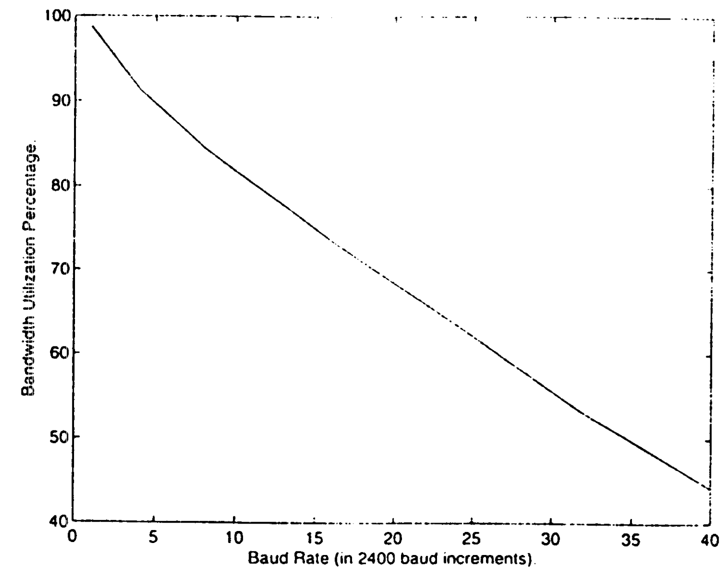


Figure 4-1: Diminishing returns for increased bandwidth.

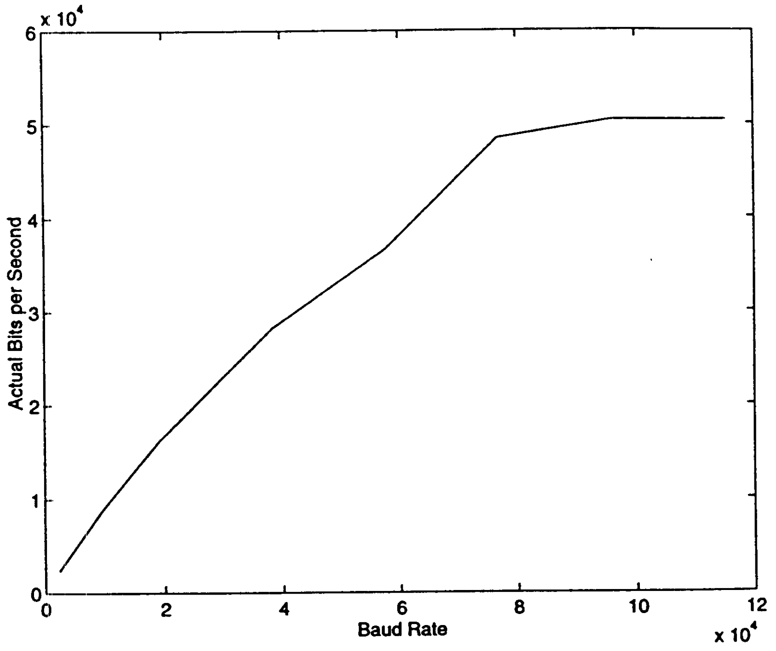


Figure 4-2: Bits per second through serial link.

4.2 Analysis

It is clear from Table 4-1 and Figure 4-1 that there is a diminishing return on bandwidth in the serial connection. Why are there diminishing returns? Recall the discussion in Section 3.2.3. While the microprocessor is reading and formatting data, the data fusion and processing unit is idle and no data flows through the serial connection.

As the baud rate is increased, the idle time remains fixed, but transmission time decreases, so the percentage of idle time rises (the percentage of idle time is inversely proportional to the bandwidth utilization). Once the system reaches a 76 Kbaud transmission rate, the transmission time is negligible compared to the processor time on the microprocessor, and the system becomes limited by the speed of the microprocessor instead of by the serial bandwidth.

How can these bottlenecks be overcome? It must be emphasized that the bottleneck at low baud rates is the serial bandwidth, while at high baud rates the limiting factor is microprocessor speed. This distinction implies that the same method will not overcome both problems. At low baud rates, the processing time on the microprocessor (idle time in the system) is less significant than transmission time. Data compression, then, seems to be a workable solution to the problem. A complex compression algorithm (especially one that involves floating point numbers) could substantially increase processor time and degrade system performance. Instead, a simple compression algorithm should be used. One possible simple compression scheme could send a single bit of overhead that would signify if the forthcoming conversion result contained 10 or 12 bits. Ten bit data would be sent when the conversion result was within 1024 counts of a fixed value (that would be added to the 10 bit result after data transmission), and 12 bit data would be sent otherwise. This compression scheme would probably not achieve more than 10% compression, but it would be easy to implement, and it could avoid slowing down the microprocessor too much.

For high baud rates, compression would actually degrade overall system performance. Instead, the data fusion and processing unit should be active while the microprocessor is working. To accomplish this, we could change the current scheme of requesting ADC channel scans at the beginning of each computation loop. Instead, we could request scans during the loop. Then, the microprocessor would be preparing another set of data while the central processor continued to merge the sensor data from the previous request. The system idle time would be greatly reduced with the new method, and performance would improve at all baud rates, but higher baud rates (that are not themselves bottlenecks) would see more improvement than low baud rates.

How fast can the system get? The maximum conversion rate is approximately 5,800 channels per second (when the microprocessor is put in continuous mode), and the system currently converts about 3,100 channels per second with a 115 Kbaud serial connection. The current system speed is only 53% of the empirical limit.

4.3 Summary

A serial connection to the MIT inertial tracker has been designed and implemented. The intention is to make this connection wireless using a standard transceiver pair, so system performance was tested using different baud rates. Higher baud rates are obviously preferred, but the returns do diminish as baud rate is increased. Cost and desired performance must be factored into any transceiver choice -- if allowed cost and desired performance are both low, then a low baud rate link will meet the requirements. If high performance is preferred and cost is less of an issue, a high baud rate link is desired. Recall, however, that no gain is made in the present system for baud rates above 76 Kbaud.

4.4 Suggestions for Further Work

The software modifications mentioned in Section 4.2 are a good beginning to further work on this project. If we refuse to increase hardware complexity further, the largest performance gains must be made with software enhancements.

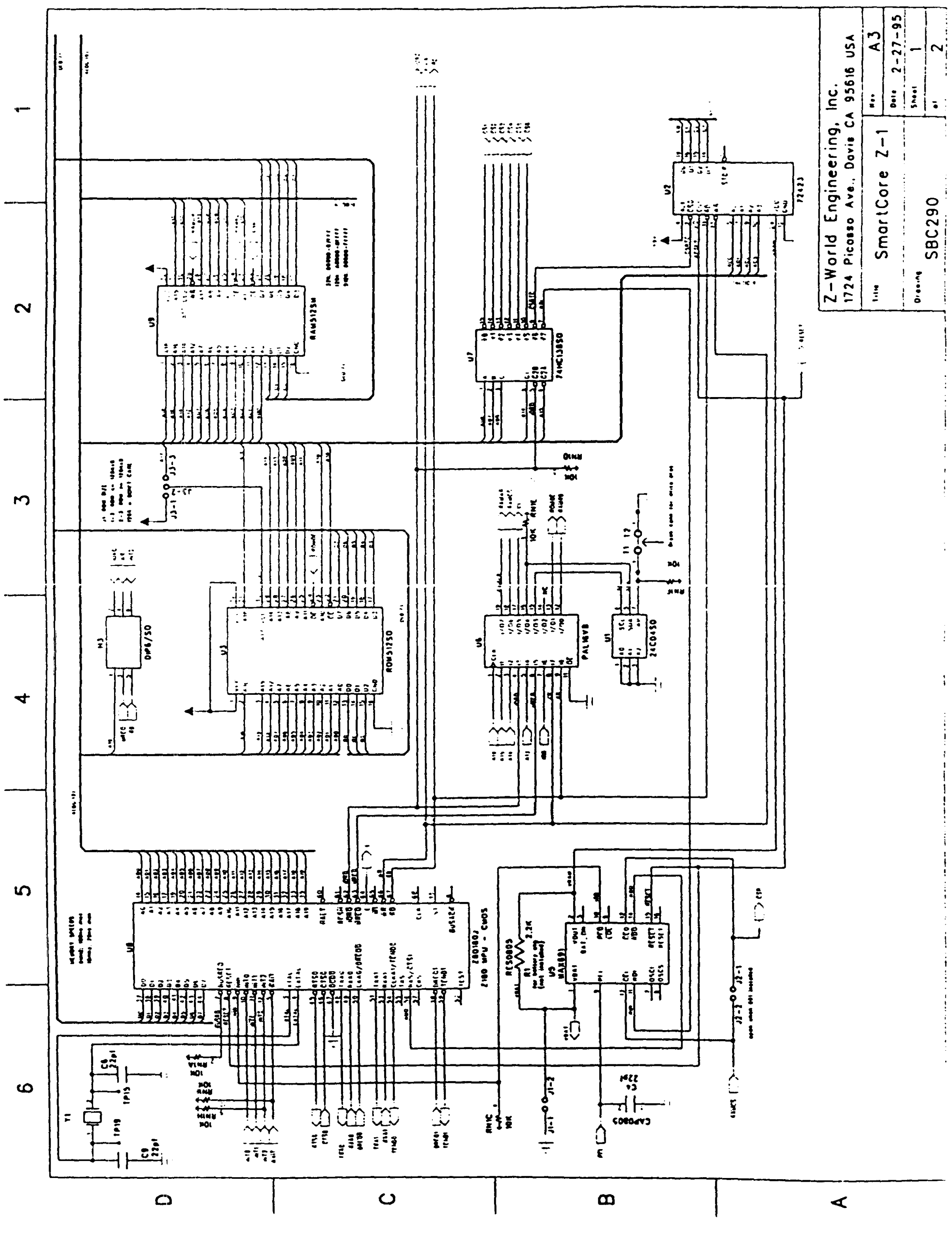
Another interesting area involves changing the data representation in the system. Currently, the ADC reports 12 bit values to the microprocessor that then sends the entire 12 bit result to the central processor. The microprocessor could instead send the differential between results (rather than the entire result each time) to save bandwidth. Differential results cause errors to accumulate in the filtering algorithm, but the filter eventually overcomes the errors. To a head mounted display user, bit errors under the differential scheme would cause their view to drift, and then ease back to where it should be. With bit errors in the current system, the field of view would shift more dramatically, but recovery would take only a single update (assuming the next conversion result is received without errors). It is unclear which system is preferable to the user. It is also unclear if these preferences are dependent upon the tracker's application.

Another interesting topic concerns the placement of the data fusion and processing block within the system (as mentioned in Section 3.1). If the block is placed before the transmitter, the required bandwidth is greatly reduced. To collocate the sensor fusion with the microprocessor, however, implies a reduction in computing power and, therefore, a degradation in the system's update rate. The system would also be much cheaper, though. It is unclear if the cost reduction would make the performance degradation worthwhile.

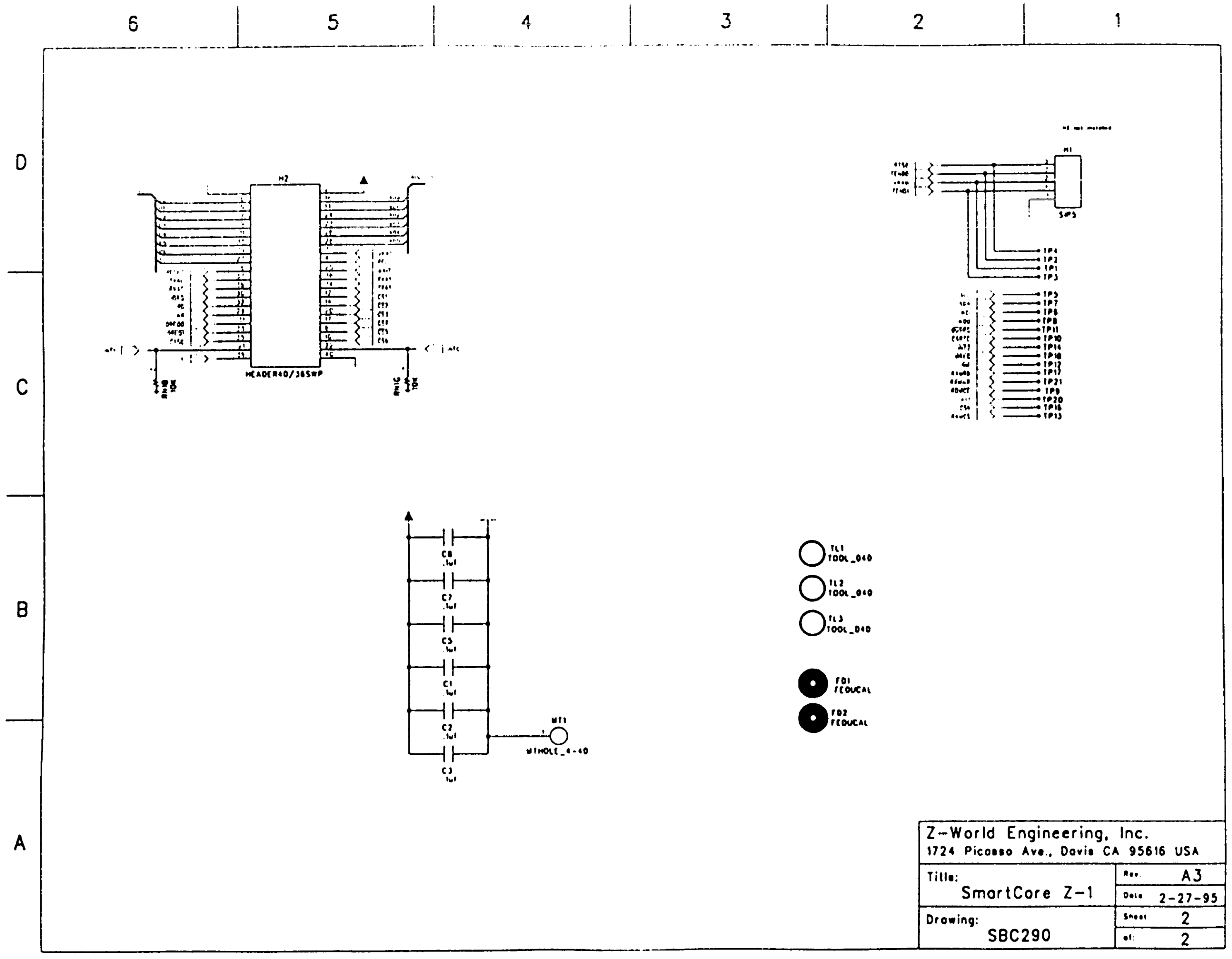
The first step toward a wireless, sourceless tracker has been made. Many applications already exist for MIT's wireless inertial tracker, and as wireless HMD's come on line, the MIT inertial tracker will become a critical component in the emergent virtual reality industry.

Appendix A

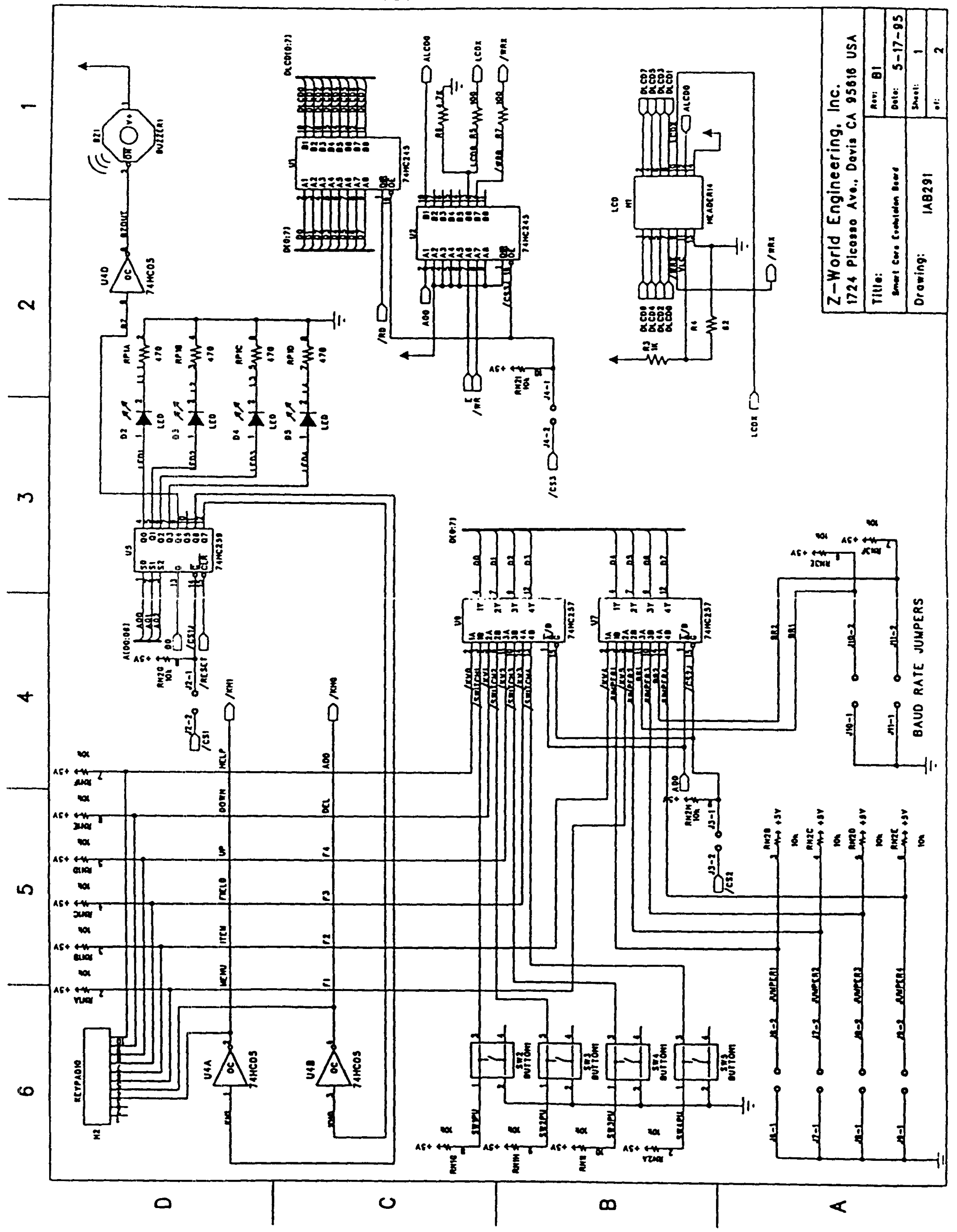
SmartCore Schematics



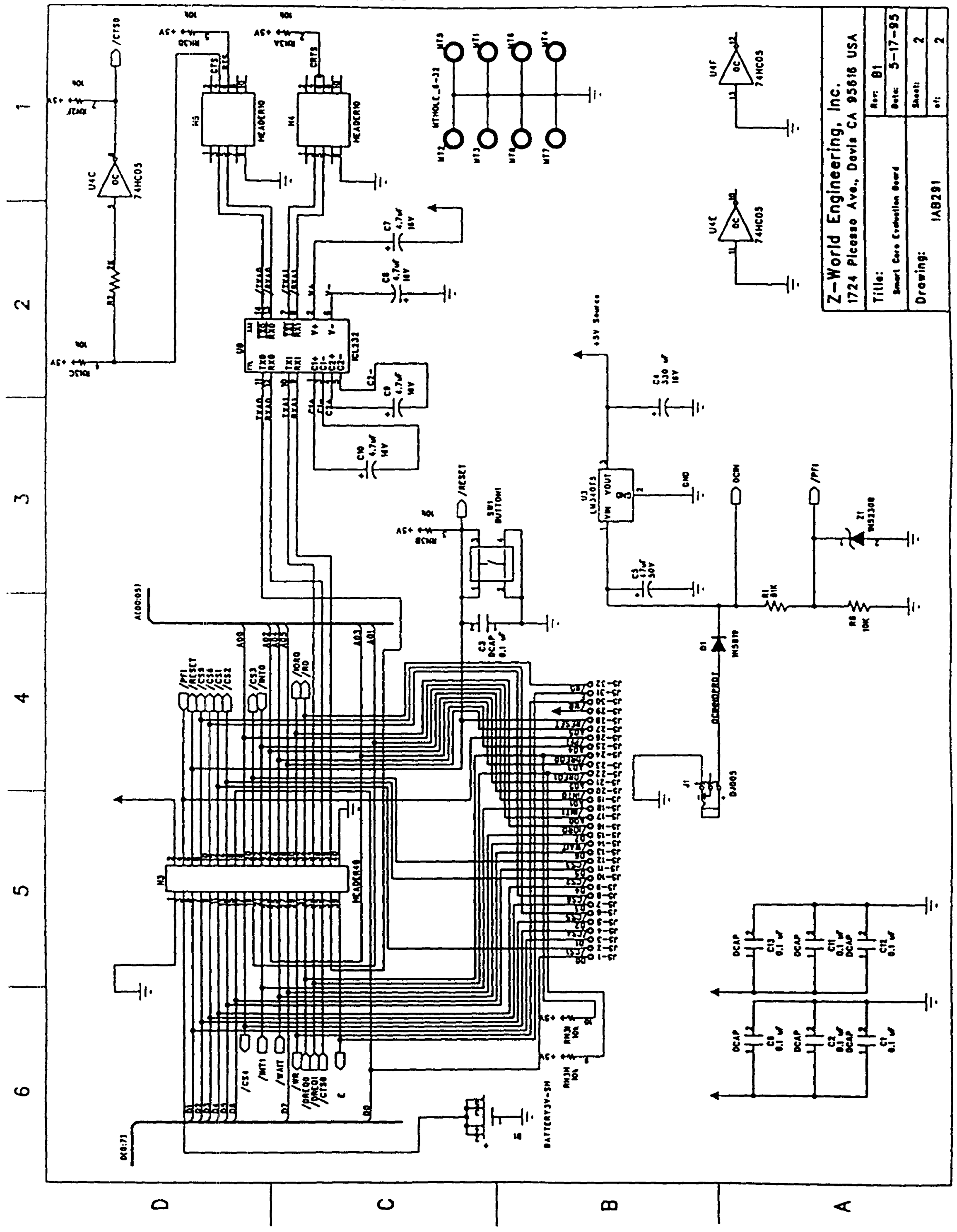
Z-World Engineering, Inc.
1724 Picasso Ave., Davis CA 95616 USA
Title SmartCore Z-1
Drawing SBC290
Rev A3
Date 2-27-95
Sheet 1
of 2
U8 Z8018Q / Z180 MPU - CMOS
U9 RAM512SM
U3 ROM512SO
U7 74HC138SO
U6 PAL16V8
U1 24C04SO
U2 72423
U5 MAX691
M3 DIP6/SO
R1 2.2K
RES0805
CAP0805
1 2 3 4 5 6
A B C D


6
5
4
3
2
1
D
C
B
A
H2
HEADER40/36SWP
H1
SIP5
TP4
TP2
TP1
TP3
TP5
TP7
TP6
TP8
TP11
TP10
TP14
TP18
TP12
TP17
TP21
TP9
TP20
TP16
TP13
C8 .1uf
C7 .1uf
C5 .1uf
C1 .1uf
C2 .1uf
C3 .1uf
MT1
MTHOLE_4-40
TL1 TOOL_040
TL2 TOOL_040
TL3 TOOL_040
FD1 FEDUCAL
FD2 FEDUCAL
Z-World Engineering, Inc.
1724 Picasso Ave., Davis CA 95616 USA
Title:
SmartCore Z-1
Drawing:
SBC290
Rev. A3
Date 2-27-95
Sheet 2
of: 2

iab291b2.sch-1 - Tue Jul 11 14:08:40 1995



iab291b2.sch-2 - Tue Jul 11 14:08:42 1995



Appendix B

Microprocessor Code

B.1 Control Code for Figure 3-4

```
/**************************************************
12-bit ADC sample code for the Smart Core and the TLC2543
**************************************************/

#define TBUFSIZE 384      // size of transmit buffer
#define RBUFSIZE 384      // size of receive buffer
#define CS4 0x40C0
#define REPEATS 1         // # of scan repetitions

unsigned int read_2543(int channel);
void channel_scan (int n, unsigned int * values, int * map);
void send_data(unsigned int * values);
unsigned int single_channel(int channel);
void initialize_map (int * map);

/**************  set_clock **********************
set_clock sets the clock on the TLC2543.  (Actually this
sets Q0 on the 74LS259, but that line is connected to CLK on
the TLC2543 (see hardware schematics for more info).

     Protocol:
     void set_clock(int state)

     Inputs:
     (int state) is the value to which the clock is being
     set.
*****************************************************/
void set_clock(int state)
{
     outport(CS4,state);
}

/*********************  read_2543  ********************
read_2543 reads a specified channel on the ADC.  This
function should be called from within channel_scan or
single_channel.  (Channel_scan and single_channel order the
data array properly.)

     Protocol:
     unsigned int read_2543 (int channel)

     Inputs:
     (int channel) is the channel that will be converted.
     [It's serial data will be output the next time a
     channel is converted.

     Outputs:
     The unsigned int that is returned is the result of the
     digital conversion of the analog channel requested by
     the PREVIOUS call to read_2543 (The PREVIOUS result is
     shifted out because data is stored by the TLC2543 until
     the next time a channel is shifted into its register).

     Changes to arrays:
     None.
******************************************************/

unsigned int read_2543(int channel)
{
int j,control;
unsigned int value;
control = channel<<8;     // The channel is shifted by 8
                          // bits instead of the required
                          // 4 bits because the bottom four
                          // bits of the control word are
                          // dummy bits.
value=0; outport(CS4,0); // Set the clock low before
                         // asserting CS.

outport(CS4+1,0);        // assert the CS on the 2543

     for(j=11;j>=0;j--)
     {
          hitwd();
          outport(CS4,0);// The first time through
                         // this loop, this is NOT
                         // a falling edge of I/O Clock
                         // because I/O clock is set low
                         // before entering this loop.

          value=(value<<1)+(inport(CS4)&1);

          outport(CS4+2,(control>>j)&1);
          outport(CS4,1);          // Set the clock high
          }
outport(CS4+1,1);            // clear the CS for the 2543
return value;
}
/********************* channel_scan *********************
channel_scan reads a block of the first n channels from the
channel map and puts their returned values into an array.

     Protocol:
     void channel_scan(int n, unsigned int * values)

     Inputs:
     (int n) is the number of channels to scan.
     (unsigned int * values) is a pointer to the data array.
     (int * map) is a pointer to the map array.  This array
     should contain the integer values of the channels in
     the order in which a scan is desired.  That is, if we'd
     like to scan channels 3, 4, 2 in that order, then map
     should be an array that begins with {3,4,2} and ends
     with the remaining channels in the desired order.  This
     order should remain the same (we don't want to change
     map every time we do a channel scan);

     Outputs:
     None.

     Changes to Arrays:
     The values array is loaded with the conversion results.
     In addition, values[11] is loaded with the number of
     valid results in the values array.
```

```
*******************************************************/
void channel_scan(int n,unsigned int * values, int * map)
{
int i;
     for (i=0;i<n;i++)
     {
     values[i] = read_2543(map[i]);
     }
values[11]=i;       //  This stores the number of valid
                        //  entries in the data array.
return;
}
/**************  send_data  ***************************
send_data sends data serially to the master computer.
12-bit results are combined to form 3 8-bit ASCII characters
that are sent through an RS232 connection.  The most
significant 8 bits of the earlier result (the result that
has a lower array index value) specify the first character
sent.  The least significant 4 bits of the earlier result
specify the most significant 4 bits of the next character.
The most significant 4 bits of the later result specify the
least significant 4 bits of the second character, and the
least significant 8 bits of the later result specify the
last of the three characters for each pair of results.
(Obviously these characters must be decoded at the master
computer).

     Protocol:
     void send_data(unsigned int * values)

     Inputs:
     (unsigned int * values) is a pointer to the data array.

     Outputs:
     None.

     Changes to arrays:
     values[11] is reduced to zero in this function.
*******************************************************/
void send_data(unsigned int * values)
{
int i;
long combined;

i=0;
     for(values[11];values[11]>=2;values[11]-=2)
          // This loop is active
          // when two or more
          // valid results are
          // left in the data array.
          // (So this loop sends
          // out 3 bytes).
     {
     combined=((long)values[i]<<12)+(long)values[i+1];

     Dwrite_z01ch((combined>>16)&255);
     Dwrite_z01ch((combined>>8)&255);
     Dwrite_z01ch((combined&255));

     i+=2;
     }

                    //  This part of the function is
     if (values[11])//  active when only one valid
     {              //  result remains in the data
                    //  array.  Only 2 bytes are
                    //  sent.  The master computer
                    //  needs to keep track of this.
     combined=((long)values[i]<<12);

     Dwrite_z01ch((combined>>16)&255);
     Dwrite_z01ch((combined>>8)&255);
     }
}


/*********************  initialize_map **************
initialize_map sets the mapped channel values.  That is, it
assigns the proper channel number to each analog input.

Protocol:
     void initialize_map(void)

     Inputs:
     None.

     Outputs:
     None.

     Changes to Arrays:

     The map array is loaded with channel mappings.
*********************************************************/
void initialize_map (int * map)
{
map[0]=8; // x gyro
map[1]=7; // y gyro
map[2]=6; // z gyro
```

```
map[3]=4; // x accel
map[4]=5; // y accel
map[5]=3; // z accel
map[6]=2; // x compass
map[7]=1; // y compass
map[8]=9; // temp sensor
map[9]=10;// unused
map[10]=11;// unused
}

/******************** single_channel  *****************
single_channel loads the specified channel into the control
register of the TLC2543 and returns the result of the
PREVIOUS conversion.  (Therefore, to get an arbitrary
channel converted on demand, this function must be called
twice.)

     Protocol:
     unsigned int single_channel (int channel)

     Inputs:
     (int channel) is the channel that will be converted.
     [It's serial data will be output the next time a
     channel is converted.

     Outputs:
     The unsigned int that is returned is the result of the
     digital conversion of the analog channel requested by
     the PREVIOUS call to read_2543 (The PREVIOUS result is
     shifted out because data is stored by the TLC2543 until
     the next time a channel is shifted into its register).

     Changes to Arrays:
     None.

*********************************************************/

unsigned int single_channel(int channel)
{
return(read_2543(channel));
}

/********************************************************/
main(){
int i,k;
unsigned int values[12];
int map[11];
char data;
int   cursor, value;
int   ledonoff;
int   beeponoff;
int   changebaud;
char tbuf[TBUFSIZE];      // transmit buffer
char rbuf[RBUFSIZE];      // receive buffer
char buf[RBUFSIZE + 1];   // dummy buffer for receiving a
                          // complete command

//  use reload_vec( ..) instead of #INT_VEC SER0_VEC if
//  port0 is also the Dynamic C programming Port

#if ROM==0
reload_vec(14, Dz0_circ_int);
#endif
Dinit_z0(rbuf, tbuf, RBUFSIZE, TBUFSIZE, 20, 38400 / 1200,
0, 0);
z0binaryset(); // interrupt routine will receive all
               // characters
hitwd();
cursor = 1;
ledonoff = 1;
beeponoff = 0;
changebaud = 0;

initialize_map(map);

     while(!Dwrite_z01ch('^'));
     for (;;)  // endless loop constantly monitoring for
               // new character
     {

     hitwd();

     if( Dread_z01ch(&data))
     {
          switch(data&240)
          {
          case 128:
               if(ledonoff) ledonoff = 0;
               else ledonoff = 1;
               outport(CS1, ledonoff);
               outport(CS1+1, ledonoff);
               outport(CS1+2, ledonoff);
               outport(CS1+3, ledonoff);
               break;
          case 64:
               up_beep(1000);    // beep for 1 second
               break;
          case 0:      // Run the channel_scan routine
```

```
            channel_scan((data&15),values,map);
            hitwd();
            send_data(values);
            break;

            default:
            up_beep(250);   // beep for 250 ms if
                            // non functional key is pressed.
            while(!Dwrite_z01ch('^'));
            break;
            }
      }
      }
}
```

B.2 Control Code for Figure 3-5

```
/**************************************************
12-bit ADC sample code for the Smart Core and the TLC2543
**************************************************/
#include <stdlib.h>
#define TBUFSIZE 384            // size of transmit buffer
#define RBUFSIZE 384            // size of receive buffer
#define CS3 0x4080
#define CS4 0x40C0
#define CS5 0x4100
#define CS6 0x4140

void read_2543(int channel, char * bytes, int index);
void channel_scan(int n,char * bytes, int * map);
void send_data(char * bytes);
unsigned int single_channel(int channel);
void initialize_map (int * map);



/*********************  read_2543************************
read_2543 reads a specified channel on the ADC.  This
function should be called from within channel_scan or
single_channel.  (Channel_scan and single_channel order the
data array properly.)

     Protocol:
     void read_2543 (int channel, char * bytes, int index)

     Inputs:
```

```
     (int channel) is the channel that will be converted.
     [It's serial data will be output the next time a
     channel is converted.]
     (char * bytes) is a pointer to the transmitted bytes
     array.
     (int * map) is a pointer to the array of channel
     mappings.

     Outputs:
     None.

     Changes to Arrays:

     The bytes array is loaded with 2*n bytes of data from
     the converter.  These bytes are ordered and sent in
     send_data.  Bytes[22] stores the number of valid
     entries in the array.

     No change is made to the map array.
***********************************************************/
void read_2543(int channel, char * bytes, int index)
{
int j,control;

control = channel<<4;// The channel is shifted by 4
                    // bits and written into the shift
                    // registers.  When the data is
                    // shifted into the A/D, only
                    // the first 8 input bits are
                    // used by the TLC2543.

outport(CS3,0);     // Clear the addressable latch
                    // and put the registers in parallel
                    // load mode.

outport(CS4+2,control);  // Clock Parallel Load
outport(CS3,control);

outport(CS4+3,control);  // Put registers into shift mode
                         // and assert the CS on TLC2543

outport(CS4,control);    // Trigger the conversion cycle

hitwd();                 // Make sure that this hitwd()
                         // call stays in this position.
                         // That way we can't try to read
                         // conversion results too quickly.

bytes[index]=inport(CS6);
bytes[index+1]=inport(CS5);

return;
}
/********************  channel_scan  ******************
channel_scan reads a block of the first n channels from the
channel map and puts their returned values into an array.

     Protocol:
     void channel_scan(int n, char * bytes, int map)

     Inputs:
     (int n) is the number of channels to scan.
     (char * bytes) is a pointer to the transmitted bytes
     array.
     (int * map) is a pointer to the map array.  This array
     should contain the integer values of the channels in
     the order in which a scan is desired.  That is, if we'd
     like to scan channels 3, 4, 2 in that order, then map
     should be an array that begins with {3,4,2} and ends
     with the remaining channels in the desired order.  This
     order should remain the same (we don't want to change
     map every time we do a channel scan);

     Outputs:
     None.

     Changes to arrays:
     The bytes array is loaded with 2*n bytes of data from
     the converter.  These bytes are ordered and sent in
     send_data.  Bytes[22] stores the number of valid
     entries in the array.

     No change is made to the map array.
*******************************************************/
void channel_scan(int n,char * bytes, int * map)
{
int i;

     for (i=0;i<n;i++)
     {
     read_2543(map[i],bytes,i*2);
     }
bytes[22]=i*2;           //  This stores the number of valid
                         //  entries in the data array.
return;
}
```

```
/*********************  send_data **********************
send_data sends data serially to the master computer.
12-bit results are combined to form 3 8-bit ASCII characters
that are sent through an RS232 connection.  The most
significant 8 bits of the earlier result (the result that
has a lower array index value) specify the second character
sent.  The least significant 4 bits of the earlier result
specify the most significant 4 bits of the first character.
The most significant 4 bits of the later result specify the
least significant 4 bits of the first character, and the
least significant 8 bits of the later result specify the
last of the three characters for each pair of results.  When
only one valid result is left in the array, the least
significant four bits of the first character sent are the
four most significant bits of the result, and the second
character sent is the least significant eight bits of the
result.   (Obviously these characters must be decoded at the
master computer).

     Protocol:
     void send_data(char * bytes)

     Inputs:
     (unsigned int * values) is a pointer to the data array.

     Outputs:
     None.

     Changes to arrays:
     Bytes[22] (the valid bytes index) is reduced to zero
     within this function.
*******************************************************/


void send_data(char * bytes)
{
int i;
unsigned int values[2];

i=0;
     for(bytes[22];bytes[22]>=4;bytes[22]-=4)
               // This loop is active
               // when four or more
               // valid bytes are
               // left in the bytes array.
               // (4 valid bytes implies
               // two valid conversion
               // results -> 3 bytes
               // of transmitted data).
     {


     Dwrite_z01ch((((long)bytes[i]&15)<<4)+(bytes[i+2]&15));
     Dwrite_z01ch(bytes[i+1]);
     Dwrite_z01ch(bytes[i+3]);

     i+=4;
     }
                         //  This part of the function is
     if (bytes[22])      //  active when only two valid
     {                   //  bytes remains in the byte
                         //  array.  Only 2 bytes are
                         //  sent.  The master computer
                         //  needs to keep track of this.
     Dwrite_z01ch(bytes[i]&15);
     Dwrite_z01ch(bytes[i+1]);
     }
}

/*********************  initialize_map **************
initialize_map sets the mapped channel values.  That is, it
assigns the proper channel number to each analog input.

Protocol:
     void initialize_map(void)

     Inputs:
     None.

     Outputs:
     None.

     Changes to Arrays:

     The map array is loaded with channel mappings.
*********************************************************/
void initialize_map (int * map)
{
map[0]=8; // x gyro
map[1]=7; // y gyro
map[2]=6; // z gyro
map[3]=4; // x accel
map[4]=5; // y accel
map[5]=3; // z accel
map[6]=2; // x compass
map[7]=1; // y compass
map[8]=9; // temp sensor
map[9]=10;// unused
```

```
map[10]=11;// unused
}
/*******************************************************/

 main()
{
int i,k;
unsigned int values[12];
int map[11];
char data;
char bytes[23];
int   cursor, value;
int   changebaud;
char tbuf[TBUFSIZE];      // transmit buffer
char rbuf[RBUFSIZE];      // receive buffer
char buf[RBUFSIZE + 1];   // dummy buffer for receiving a
                          // complete command
//  use reload_vec( ..) instead of #INT_VEC SER0_VEC if
//  port0 is also the Dynamic C programming Port

#if ROM==0
reload_vec(14, Dz0_circ_int);
#endif
Dinit_z0(rbuf, tbuf, RBUFSIZE, TBUFSIZE, 20, 115200 / 1200,
0, 0);
z0binaryset(); // interrupt routine will receive all
               // characters

hitwd();
changebaud = 0;

initialize_map(map);


     while(!Dwrite_z01ch('^'));
     for (;;)  // endless loop constantly monitoring for
               // new character
     {
          runwatch();
          hitwd();

          if( Dread_z01ch(&data))
          {
               switch(data&240)
                    {
                    case 128:  // Convert Single Channel

                    read_2543((data&15),1, bytes, map);
                    send_data(bytes);
```

```
                        break;

                        case 0:     // Scan Channels
                        read_2543(0,(data&15),bytes,map);
                        send_data(bytes);
                        break;

                        default:
                        while(!Dwrite_z01ch('^'));
                        break;
                        }
                }
            }
}
```

B.3 Improved Control Code for Figure 3-5

This code is faster than the code in B.2, but the speed is achieved by sacrificing modularity. With an overhead of approximately 30 instructions per function call, the program speed was significantly increased by reducing function calls.

```
/***************************************************
12-bit ADC sample code for the Smart Core and the TLC2543
***************************************************/
#include <stdlib.h>
#define TBUFSIZE 384         // size of transmit buffer
#define RBUFSIZE 384         // size of receive buffer
#define CS3 0x4080
#define CS4 0x40C0
#define CS5 0x4100
#define CS6 0x4140

void read_2543(int channel, int n, char * bytes, int * map);
void send_data(char * bytes);
void initialize_map (int * map);




/*********************  read_2543 ***************
read_2543 reads a specified channel on the ADC.  This
function replaces channel_scan and single_channel from
earlier versions.
Protocol:
     void read_2543 (int channel, int n, char * bytes, int *
                         map)
     Inputs:
```

```
     (int channel) is first mapped channel to be converted.
     [It's serial data will be output the next time a
     channel is converted.]
     (int n) is the number of mapped channels to scan.
     (char * bytes) is a pointer to the transmitted bytes
     array.
     (int * map) is a pointer to the array of channel
     mappings.

     Outputs:
     None.

     Changes to Arrays:

     The bytes array is loaded with 2*n bytes of data from
     the converter.  These bytes are ordered and sent in
     send_data.  Bytes[22] stores the number of valid
     entries in the array.

     No change is made to the map array.
**********************************************************/

void read_2543(int channel, int n, char * bytes, int * map)
{
int control,i,index;

     for (i=channel;i<n;i++)
     {
     control = map[i]<<4;// The channel is shifted by 4
                         // bits and written into the shift
                         // registers.  When the data is
                         // shifted into the A/D, only
                         // the first 8 input bits are
                         // used by the TLC2543.

     outport(CS3,0);     // Clear the addressable latch
                         // and put the registers in
                         // parallel load mode.

     outport(CS4+2,control);  // Clock Parallel Load
     outport(CS3,control);


     outport(CS4+3,control);  // Put registers into shift
                              // mode and assert the CS on
                              // TLC2543

     outport(CS4,control);// Trigger the conversion cycle
     hitwd();              // Make sure that this hitwd()
                           // call stays in this position.
                           // That way we can't try to read
                           // conversion results too quickly.
     index = i*2;

     bytes[index]=inport(CS6);
     bytes[index+1]=inport(CS5);
     }

bytes[22]=i*2;             //  This stores the number of valid
                           //  entries in the data array.
return;
}

/********************* send_data **********************
send_data sends data serially to the master computer.
12-bit results are combined to form 3 8-bit ASCII characters
that are sent through an RS232 connection.  The most
significant 8 bits of the earlier result (the result that
has a lower array index value) specify the second character
sent.  The least significant 4 bits of the earlier result
specify the most significant 4 bits of the first character.
The most significant 4 bits of the later result specify the
least significant 4 bits of the first character, and the
least significant 8 bits of the later result specify the
last of the three characters for each pair of results.  When
only one valid result is left in the array, the least
significant four bits of the first character sent are the
four most significant bits of the result, and the second
character sent is the least significant eight bits of the
result.   (Obviously these characters must be decoded at the
master computer).

     Protocol:
     void send_data(char * bytes)

     Inputs:
     (unsigned int * values) is a pointer to the data array.

     Outputs:
     None.

     Changes to arrays:
     Bytes[22] (the valid bytes index) is reduced to zero
     within this function.
*****************************************************/
```

```
void send_data(char * bytes)
{
int i;
unsigned int values[2];

i=0;
     for(bytes[22];bytes[22]>=4;bytes[22]-=4)
               // This loop is active
               // when four or more
               // valid bytes are
               // left in the bytes array.
               // (4 valid bytes implies
               // two valid conversion
               // results -> 3 bytes
               // of transmitted data).
     {


     Dwrite_z01ch((((long)bytes[i]&15)<<4)+(bytes[i+2]&15));
     Dwrite_z01ch(bytes[i+1]);
     Dwrite_z01ch(bytes[i+3]);

     i+=4;
     }
                         //  This part of the function is
     if (bytes[22])      //  active when only two valid
     {                   //  bytes remains in the byte
                         //  array.  Only 2 bytes are
                         //  sent.  The master computer
                         //  needs to keep track of this.
     Dwrite_z01ch(bytes[i]&15);
     Dwrite_z01ch(bytes[i+1]);
     }
}

/*********************  initialize_map ***************
initialize_map sets the mapped channel values.  That is, it
assigns the proper channel number to each analog input.

Protocol:
     void initialize_map(void)

     Inputs:
     None.

     Outputs:
     None.

     Changes to Arrays:
```

```
     The map array is loaded with channel mappings.
*********************************************************/
void initialize_map (int * map)
{
map[0]=8; // x gyro
map[1]=7; // y gyro
map[2]=6; // z gyro
map[3]=4; // x accel
map[4]=5; // y accel
map[5]=3; // z accel
map[6]=2; // x compass
map[7]=1; // y compass
map[8]=9; // temp sensor
map[9]=10;// unused
map[10]=11;// unused
}
/********************************************************/

 main()
{
int i,k;
unsigned int values[12];
int map[11];
char data;
char bytes[23];
int   cursor, value;
int   changebaud;
char tbuf[TBUFSIZE];      // transmit buffer
char rbuf[RBUFSIZE];      // receive buffer
char buf[RBUFSIZE + 1];   // dummy buffer for receiving a
                          // complete command
//  use reload_vec( ..) instead of #INT_VEC SER0_VEC if
//  port0 is also the Dynamic C programming Port

#if ROM==0
reload_vec(14, Dz0_circ_int);
#endif
Dinit_z0(rbuf, tbuf, RBUFSIZE, TBUFSIZE, 20, 115200 / 1200,
0, 0);
z0binaryset(); // interrupt routine will receive all
               // characters

hitwd();
changebaud = 0;

initialize_map(map);
```

```
    while(!Dwrite_z01ch('^'));
    for (;;)  // endless loop constantly monitoring for
              // new character
    {
         runwatch();
         hitwd();

         if( Dread_z01ch(&data))
         {
              switch(data&240)
                   {
                   case 128:  // Convert Single Channel

                   read_2543((data&15),1, bytes, map);
                   send_data(bytes);
                   break;

                   case 0:     // Scan Channels
                   read_2543(0,(data&15),bytes,map);
                   send_data(bytes);
                   break;

                   default:
                   while(!Dwrite_z01ch('^'));
                   break;
                   }
         }
     }
}
```

Appendix C

Driver Code

C.1 Zworld.c

```
/*  ZWORLD.C -- This program communicates with the Zworld
Z180 processor thru a serial port.  The processor then
operates the A/D inside the sensor block, thus making a
serial link from INTRACK to the sensors.  This link can then
be made wireless with an off the shelf wireless serial
connection.  Uses the C Communications Toolkit from Magna
Carta Software to achieve fast RS-232 communications up to
115 Kbaud.

     AUTHOR: John Rodkin
     DATE: January, 1996
     Based on RS232.C by Eric Foxlin, October, 1993*/

#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <dos.h>
#include <graphics.h>
#include <stdio.h>
```

```
#include <math.h>
#include <sys\timeb.h>
#include <comm.h>
#include "globls.h"
#include "rs232.h"
#include "mccmds.h"

#define   ADBASE    0x260 // ISA interface card base address
#define SELECT0 ADBASE
#define SELECT1 ADBASE+4
#define SELECT2 ADBASE+8
#define SELECT3 ADBASE+12
#define SELECT7 ADBASE+28

void goodbye(void);

/* local function prototypes */

void set_channel(int);    /* SET_CHANNEL: Loads the
                          // multiplexer with the specified
                          // channel and updates the variable
                          // muxchan to reflect this. This
                          // call has to actually trigger a
                          // conversion to get the new
                          // channel loaded in, so it takes
                          // about 32 microseconds; use only
                          // when needed!  */

void set_channel(int channel)
{
int error;
unsigned char inputs[3];

zworld_send_char(128+channel);

error = zworld_wait_for_bytes(1);

zworld_get_bytes(inputs,2);
return;
}

/* AD_CONVERT: Convert any particular A/D input channel on
demand */

unsigned int ad_convert(int channel)
{
int tlcchan, result, error;
unsigned char inputs[3];
     switch(channel){
```

```
     case 0:  // x gyro
          tlcchan = 8;
          break;
     case 1:  // y gyro
          tlcchan = 7;
          break;
     case 2: //  z gyro
          tlcchan = 6;
          break;
     case 3: //  x accel
          tlcchan = 4;
          break;
     case 4: //  y accel
          tlcchan = 5;
          break;
     case 5: //  z accel
          tlcchan = 3;
          break;
     case 6: //  x compass
          tlcchan = 2;
          break;
     case 7: //  y compass
          tlcchan = 1;
          break;
     case 8: //  temp sensor
          tlcchan = 9;
          break;
     default:
          tlcchan = 0;
          break;
     }
zworld_send_char(128+tlcchan);
error = zworld_wait_for_bytes(1);

zworld_get_bytes(inputs,2);

return(((inputs[0]&15)<<8)+inputs[1])<<4;
}

/* MEASURE_VALUES: scans first n a/d channels and saves into
values[]*/

void measure_values(int n, unsigned int *vvalues)
{
unsigned int data_high, data_low;
int  channel;
int  i;
unsigned char inputs[3];
int error;
```

```
zworld_send_char(n);
error = zworld_wait_for_bytes(n);

     for(i=0;i<n-1;i+=2)        //  We're in this loop if two
                                //  or more channels remain
                                //  (2 channels -> 3 bytes)
     {
     zworld_get_bytes(inputs,3);

     vvalues[i]=(((inputs[0]&240)<<4)+inputs[1])<<4;
     vvalues[i+1]=(((inputs[0]&15)<<8)+inputs[2])<<4;
     }

     if(i<n)    //  We're in this loop if only one
                //  channel remains (1 channels -> 2 bytes)
     {
     zworld_get_bytes(inputs,2);
     vvalues[i]=(((inputs[0]&15)<<8)+inputs[1])<<4;
     }
}

/* MEASURE_BIASES: Average 100 measurements of each A/D
channel while   the sensor assembly is in reference
position. */
void measure_biases(unsigned int *bbiases)
{
int i,channel;
int numtrials = 1000;
int trial;
unsigned int vvvalues[NUMCHANS];
unsigned long accum[NUMCHANS];
     for(channel=0; channel<NUMCHANS; channel++)
     {
     accum[channel] = 0;
     }
     for(trial=0; trial<numtrials; trial++)
     {
     measure_values(NUMCHANS, vvvalues);
          for(channel=0; channel<NUMCHANS; channel++)
          {
          accum[channel] += vvvalues[channel];
          }
     }
     for(channel=0; channel<NUMCHANS; channel++)
     {
     biases[channel] = (unsigned int) (accum[channel]/
                                        numtrials);
     }
```

```
}



void init8255(void)
{
/*  Setup for 8255.  Write a 145 control word to the control
register (which requires A0 and A1 to be high, so write to
SELECT7+3). */

outp((SELECT7+3),145);
led_both();
}

int read_baud (void)
{
/*  This function reads the first four bits of Port C, where
the baud select lines are connected.  Created 8/6/95. */

return(inp(SELECT7+2));
//  SELECT7+2 denotes Port C of the 8255
}

int read_button (void)
{
if(((read_baud())&0x08)!=0)
     return(0);
else
     return(1);
}

void led_3d(void)
{
int read8255;
// Port B is configured as an output but the latches may be
// read.

read8255 = inp(SELECT7+1);

// turn off the 6d led on PB1:
read8255 = read8255&253;

// turn on the 3d led on PB2:
read8255 = read8255|4;

outp(SELECT7+1, read8255);
}
```

```
void led_6d(void)
{

int read8255;
// Port B is configured as an output but the latches may be
// read.

read8255 = inp(SELECT7+1);

// turn off the 3d led on PB2:
read8255 = read8255&251;

// turn on the 6d led on PB1:
read8255 = read8255|2;
outp(SELECT7+1, read8255);
}

void led_both(void)
{
outp(SELECT7+1, 1);
}

void led_neither(void)
{
outp(SELECT7+1, 7);
}
```

Bibliography

Durlach, N. I., and A. S. Mavor, eds. (1995). *Virtual Reality. Scientific and Technological Changes.* Washington, D. C. : National Academy Press.

Foxlin, E. (1993) *Inertial Head Tracking.* Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Foxlin, E., and N. Durlach. (1994). An inertial head-orientation tracker with automatic drift compensation for use in HMDs. Paper submitted to *VRST '94: The ACM Symposium on Virtual Reality Systems and Technology.*

Foxlin, E., and N. Durlach. (1995). Development of Improved 6-DOF Head-Tracking Technology for HMD Applications Based on Inertial Sensors.

Meyer, K., Applewhite, H., and Biocca, F. (1992). A Survey of Position Trackers. *Presence, Volume 1, Number 2.* 173 - 189.

Raab, F., Blood, E., Steiner, O., and Jones, H. (1979). Magnetic Position and Orientation Tracking System. *IEEE Transactions on Aerospace and Electronic Systems, AES-15 #5,* 709 - 717.

Zworld Engineering. (1995). Smartcore: C Programming Core Module. Technical Reference Manual. Version 1.0.