Approved for public release; distribution is unlimited

IMPROVING SOFTWARE CHARACTERISTICS OF A REAL-TIME SYSTEM USING REENGINEERING TECHNIQUES

by

Scott Allan Book
Lieutenant, United States Navy
B.A., Mount Vernon Nazarene College, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1994

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1994	Master's Thesis

4. TITLE AND SUBTITLE
Improving Software Characteristics of a Real-Time System Using Reengineering Techniques (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Book, Scott Allan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is how to improve an existing real-time software system's readability, maintainability, stability and portability using reengineering techniques. A fundamental portion of the Model-based Mobile robot Language (MML) was the real-time system chosen as the basis for this study.

The approach taken was to create a new system design. The new design was based on system specifications obtained by conducting static and dynamic analysis on the existing system.

The results are that a new core system was implemented using a design that focused on creating independent software sub-systems while encapsulating data. Hardware dependencies were localized and assembly code minimized. The new system is easier to understand and modify and is portable to other hardware platforms.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Autonomous vehicle, robot, software engineering, real-time system	118
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

ABSTRACT

The major problem addressed by this research is how to improve an existing real-time software system's readability, maintainability, stability and portability using reengineering techniques. A fundamental portion of the Model-based Mobile robot Language (MML) was the real-time system chosen as the basis for this study.

The approach taken was to create a new system design. The new design was based on system specifications obtained by conducting static and dynamic analysis on the existing system.

The results are that a new core system was implemented using a design that focused on creating independent software sub-systems while encapsulating data. Hardware dependencies were localized and assembly code minimized. The new system is easier to understand and modify and is portable to other hardware platforms.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. PROBLEM STATEMENT 1
B. BACKGROUND 1
C. OVERVIEW 2
II. YAMABICO SOFTWARE SYSTEM ARCHITECTURE 3
A. STATIC ANALYSIS OF MML-10 4
1. User vs. Kernel 4
2. Kernel System 5
3. Motion Control Process 7
4. I/O Process 8
5. Sonar Process 9
B. DYNAMIC ANALYSIS OF MML-10 9
C. SYSTEM CHARACTERISTICS 9
1. Coupling 10
2. Cohesion 10
3. Modifiability 11
4. Modularity 11
5. Readability 11
6. Robustness 11
D. CHARACTERISTICS OF MML-10 12
III. YAMABICO HARDWARE ARCHITECTURE 14
A. THE CPU SYSTEM 16
B. THE SERIAL SYSTEM 17
C. THE WHEELS SYSTEM 17
D. THE SONAR SYSTEM 18
E. THE UNIX HOST SYSTEM 18
F. THE CONSOLE SYSTEM 18
IV. SYSTEM DESIGN 19
A. DESIGN GOALS 19
B. MODELING NOTATION 19
C. SYSTEM DESIGN 20
1. System Overview 20
2. CPU System 20
3. Serial System 22
4. Motion Control System 24
5. Wheel System 25
6. Sonar System 25
7. Tracing Systems 26
8. Terminal System 27
V. YAMABICO SOFTWARE DEVELOPMENT ENVIRONMENT 29
A. C COMPILERS 29

1. Compiler Drivers 29
2. Code Optimization 30
3. Standard Libraries 31
4. CC vs. GCC 31
5. Using GCC 31
B. LINKING 32
1. The Kernel Module 34
2. The User Module 35
C. LOADING 35
D. DEBUGGING TOOLS 36
1. The Onboard Debugger 36
2. The Unix nm Command 37
VI. SYSTEM IMPLEMENTATION AND TESTING 38
A. IMPLEMENTATION TECHNIQUES 38
1. Stability 38
2. Portability 39
3. Readability 40
4. Backward Compatibility 41
B. SYSTEM IMPLEMENTATION 41
1. The CPU System 42
2. The Terminal System 43
3. The Motion Control System 43
4. The User Program 43
C. SYSTEM TESTING 43
1. Using an External Power Supply 44
2. Output from Interrupt Handlers 44
3. Measurements 44
VII. CONCLUSIONS 46
A. RESULTS 46
B. RECOMMENDATIONS 46
APPENDIX A 47
A. DEFINITIONS.H 47
B. MAIN.C 48
C. SYSTEM.H 49
D. SYSTEM.C 50
E. MOTOROLA.H 53
F. MOTOROLA.ASM.S 54
G. SERIAL.H 58
H. SERIAL.C 60
APPENDIX B 69
A. IOSYS.H 69
B. IOSYS.C 70
APPENDIX C 76

A. MOTION.H 76
B. MOTION.C 77
C. MOTIONTRACE.H 85
D. MOTIONTRACE.C 87
E. WHEELS.H 91
F. WHEELS.C 93
APPENDIX D 101
A. USER.H 101
B. USER.C 101
C. COMPATABILITY.H 104
LIST OF REFERENCES 107
INITIAL DISTRIBUTION LIST 109

LIST OF FIGURES

Figure 1: MML System Overview 3
Figure 2: MML System 4
Figure 3: Kernel System 6
Figure 4: Motion Control Process 7
Figure 5: Introduction of a Unique Variable 13
Figure 6: Yamabico-11 Computer Architecture 14
Figure 7: Yamabico-11 Mobile Robot 15
Figure 8: Serial Board Conceptual Diagram 17
Figure 9: Design Notation 20
Figure 10: System Network 21
Figure 11: CPU System 22
Figure 12: Serial System 23
Figure 13: Motion Control System 24
Figure 14: Wheel System 25
Figure 15: Sonar System 26
Figure 16: Tracing Systems 27
Figure 17: Terminal System 28
Figure 18: Compiler Driver Process 30
Figure 19: Example of Composite Object File 33
Figure 20: Effects of Compilation on Strings 34
Figure 21: Yamabico-11 Memory Map 36
Figure 22: Velocity Control Results 45

I. INTRODUCTION

A. PROBLEM STATEMENT

The problem this thesis solves is how to reengineer MML, a real-time control system for the Yamabico-11 autonomous robot, while improving system stability, maintainability and portability.

B. BACKGROUND

MML is a real-time system under development for the Yamabico-11 robot. The goal of the project is to create a robot-independent, high-level language for mobile robot control. The language contains sets of library functions to handle geometry, motion, sonar, and I/O. These routines could then be used by developers to program a robot's movement without the requirements of low-level motion control understanding.

In the class of motion control functionality, MML originally used a sequence of configurations to describe a vehicles desired path. A configuration represents the robot's current position and orientation in a 2D world. Current research involves expanding the language to describe motion control using directed path segments as well. Each directed path is defined by a point that lies on the segment, the orientation of the segment in the 2D world and the segment's curvature.

The status of the MML system typifies that of many software systems in existence today. Roughly 50 to 80 percent of a software organization's resources are spent maintaining existing systems. Many of these systems were developed without complete specifications or documentation by analysts and programmers who are no longer with the organization. Analysts and programmers with incomplete system knowledge are then required to perform the necessary maintenance [Yourdon 93].

As a research system, MML evolved through modifications made from several graduate students. Many of these changes were unstructured and made use of global

variables. At times, particular functions or sections of code were altered simultaneously by different developers. The result is an unstable system that is extremely difficult to maintain.

Some organizations are investing resources into software reengineer techniques, hoping to reduce maintenance efforts and costs dedicated to these existing systems. There are three fundamental approaches to reengineering a system. The first technique, called restructuring, reorganizes unstructured source code into a modular form. The new form remains functionally equivalent to the older version. The second method is to reengineer the system. The goal of this plan is to replace existing code with newer versions, possibly written in a higher level language. This is done gradually by the maintenance programmers each time maintenance is required. Reverse engineering is the third technique. The objective of this procedure is to reconstruct the design, and/or the specifications from the existing source code [Yourdon 93].

C. OVERVIEW

This thesis shows a procedure for reengineering an existing real-time system, MML-10. The first step in this process is to determine the functionality of the existing system. Chapter two analyzes MML-10's software system, while chapter three covers the hardware components of the Yamabico-11 robot. Generating a new design is the second step and is covered in chapter four. The third step is to implement and test the new design. Chapter five details Yamabico-11's software development environment. The design implementation and results are discussed in chapter six. Chapter seven presents the conclusions and future recommendations.

II. YAMABICO SOFTWARE SYSTEM ARCHITECTURE

Prior to reengineering an existing software system, the maintenance programmer must have a thorough understanding of that system's logical design. Unless the programmer was involved with the system's original implementation, this is only achieved by reviewing the software's specifications and/or documentation. For systems lacking complete specifications and /or documentation, such as MML, it may be necessary to perform reverse engineering techniques to recover this information. Since current CASE technology focuses on developing new systems, these tools can provide limited support, but not at a high level [Yourdon 93].

Two of the most effective methods for reconstructing system specifications and/or documentation are static and dynamic analysis. The results of these two techniques form a logical description of the system. This description can then be translated into physical depictions, such as data flow diagrams or state transition diagrams [Yourdon 89]. The context diagram of the current MML system is shown in Figure 1.

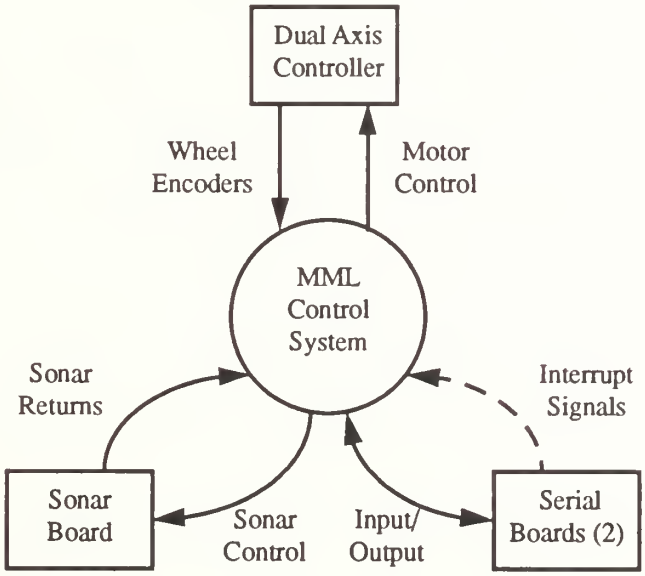

Figure 1: MML System Overview

A. STATIC ANALYSIS OF MML-10

Static analysis is the process of gaining an understanding of a software system through source code examination. The goal of this phase is to recover enough information to create a logical picture of the existing system. One important technique involves the tracing of function and procedure calls. When performing this procedure, it is not desirable to produce a complete or detailed trace as this only wastes time and resources.

Another important technique involves variable tracing. The emphasis during this operation should be on tracing parameters passed to functions and procedures, any values they return and references to global variables.

1. User vs. Kernel

As shown in Figure 2, the MML system is composed of two distinct parts: the kernel module and a user program. The kernel contains the low level routines required to

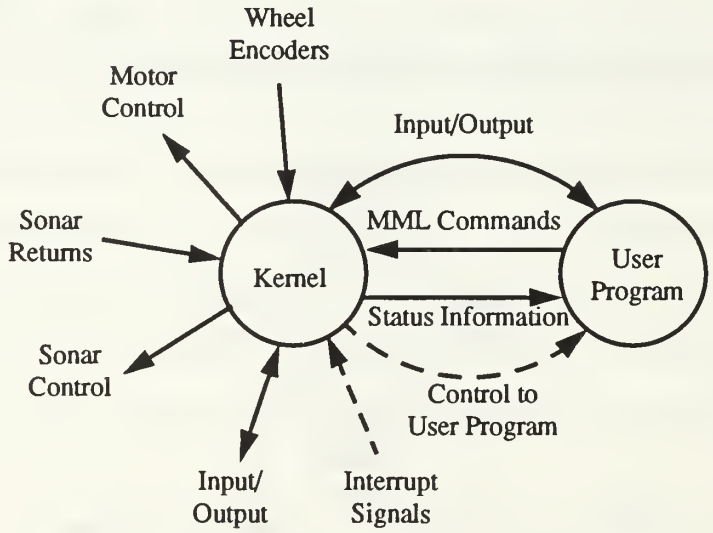

Figure 2: MML System

control movement and to perform input/output. The low level details are hidden from developers through the MML command set. A user program consists of MML command calls and C constructs to describe one or more motion behaviors. Each command either requests status information from the kernel or commands a desired behavior [Scott 93].

Each module is downloaded to the robot separately. Robot operation is only possible after both modules are loaded. Program execution starts with the kernel. The kernel initializes the sub-systems and then passes control to the user program. The user program maintains primary control until termination. However, the kernel's control processes will interrupt execution for short durations. This design allows a user to quickly alter the behavior of the robot by simply changing the user program used by the kernel.

2. Kernel System

Figure 3 is a modified data flow diagram showing the kernel as a collection of control type processes. Each control process, indicated by a dashed circle, is associated with a distinct hardware generated interrupt. The interrupts are prioritized through the hardware's configuration. The current CPU provides eight priority levels. When an interrupt is generated, the associated control process assumes control of the CPU, provided it is not interrupting a higher level process. Otherwise, it will wait until all higher priority interrupts are serviced before taking control. The interrupt level of each process is indicated by the value within parentheses.

Since motion control is responsible for maneuvering the robot, receives the highest priority given to the control processes, interrupt level four. It controls movement by first estimating the robot's current odometry configuration through dead reckoning. This estimate is then used to calculate the necessary linear and rotational velocities. These calculations are based on the motion required to follow the current path segment. Finally, these velocities are translated into pulse width modulation commands and sent to the motors controlling the robot's wheels. This process, called the motion control cycle, repeats every 10 milliseconds and requires approximately 2.5 milliseconds to execute [MacPherson 93].

The input/output process receives the next highest priority at interrupt level three. It uses interrupt driven input and output to effect information transfers between the

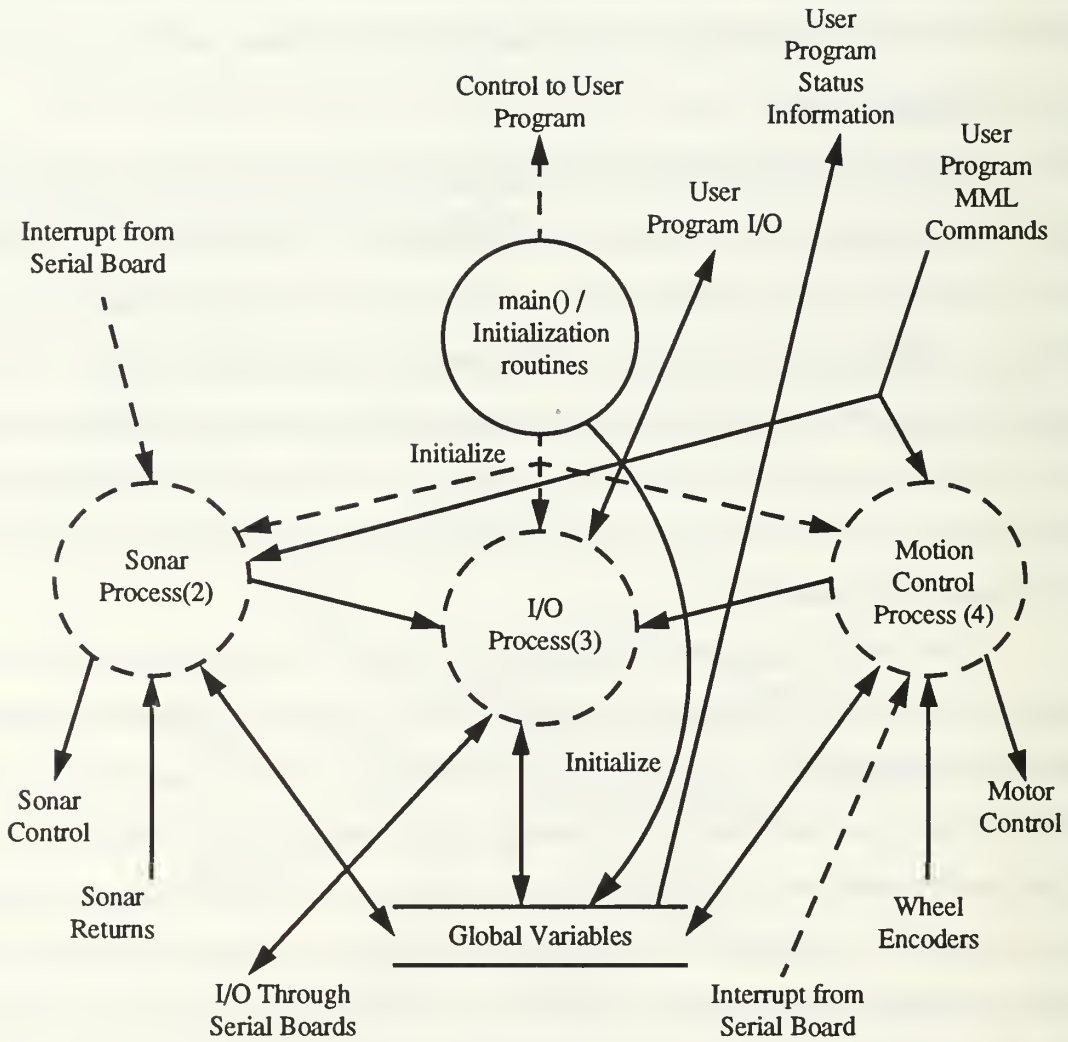

Figure 3: Kernel System

robot and the onboard console device. It also controls the transfer of data between the robot and a Unix workstation [MacPherson 93].

Interrupt priority level two signals the robot's sonar process. This process collects sonar range information used for obstacle avoidance. The process requires 240 microseconds to complete and is repeated every 24 milliseconds [DeClue 93].

3. Motion Control Process

As depicted in Figure 4, the major responsibilities of motion control are processing MML commands and pathtracking. Each MML command is characterized as

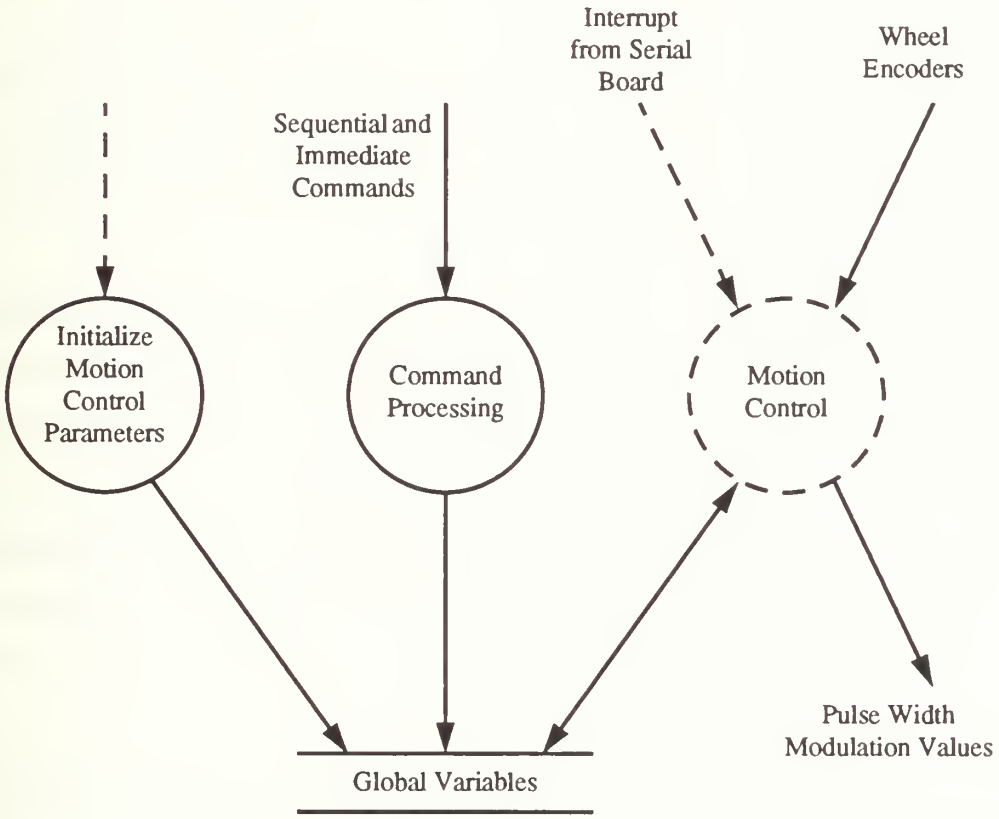

Figure 4: Motion Control Process

either an immediate or sequential command. Both types induce changes to one or more control parameters. Immediate commands induce the changes at the instant they are called

from within the user program. However, when a sequential command is called from the user program, it is added to an instruction queue. Each sequential command is then executed only after the previous sequential command is finished. Immediate command execution and storage of sequential commands into the instruction queue is independent from the motion control cycle.

For path tracking, a three step process is executed every motion control cycle. First, Yamabico updates its current odometry configuration, position and orientation, by reading hardware registers associated with wheel movement. Next, this information is used to calculate the robot's next intended movement using control rules. Early motion description methods in MML compared the current configuration against a reference configuration [Kanayama 91]. However, recent research has proven that smoother motion control is achieved by comparing the current configuration against a reference path segment [MacPherson 93]. Finally, the intended movement is translated into pulse width modulation values and sent to the motor control board.

A path segment is either a straight line, circular arc, parabola or cubic spiral. Yamabico will track a sequence of these path segments added to the instruction queue by calculating a transition point from one segment to the next. Once the transition point is reached, the robot will begin following the next path segment.

A secondary task of motion control is data logging. This purpose of this function is to record control data every motion control cycle. The collected data is used by the system programmers for debugging.

4. I/O Process

As previously mentioned, the input/output process is responsible for transferring information between the robot and either the Unix workstation or the console. Communications between the robot and the Unix workstation are accomplished by polling a serial port mapped to a specific memory location.

Transferring information with the console is also memory mapped, however this process is interrupt driven. During output, data is stored in a 1024 character circular buffer. The interrupt sequence is then initiated by sending a null character directly to the console. After the character is received by the console, an interrupt causes the next character in the buffer to be sent. After this character is written to the console, another interrupt is generated. This process continues until the buffer is empty and a command to terminate the interrupt cycle is sent to the port. Input is handled in a similar fashion.

5. Sonar Process

As mentioned earlier, the sonar process is responsible for recording sonar range returns. This information is made available to user programs for obstacle avoidance. The returns can also be used for automated cartography by building line segments using a least squares linear fitting algorithm [DeClue 93][MacPherson 93].

B. DYNAMIC ANALYSIS OF MML-10

When static analysis provides insufficient insight to a system's behavior, dynamic analysis should be used. Dynamic analysis is the process of tracing a system during execution. Using dynamic analysis, a maintenance programmer can follow the path of execution, monitor access to a particular memory location or alter the storage values. This is particularly useful when trying to locate the cause of a system crash [Yourdon 93].

Tracing MML-10 has proven to be very difficult. Two reasons are the system's low readability and high degree of coupling. This is the result of using inappropriate modularity techniques. These and other system characteristics are described below. Another reason is the difficulty associated with using a debugger due to the timing constraints of a real time system.

C. SYSTEM CHARACTERISTICS

A by-product of conducting static and dynamic analysis is knowledge of a system's software characteristics. Software characteristics are those attributes used to describe the

quality of a system's design and implementation. The following list of attributes is used to describe the internal makeup of a system and is directly related to the effort required to perform system maintenance.

1. Coupling

When one module references a symbolic address defined outside of that module, a connection (or interdependency) is created between the module with the reference and the module with the definition. Coupling describes the types and strength of these connections between modules. References to internal data elements or data structures is known as common coupling since code segments are referencing a common data area. Another form of coupling is control coupling. This exists when control switches such as flags are used between modules. The purpose of these flags is to change or modify the behavior or actions of a routine. Low coupling exists when references between modules are limited to procedure and/or function names.

As coupling increases, a system is more difficult to understand and maintain. Therefore, it is desirable to reduce coupling by reducing the references to another module's internal elements. One method for reducing common coupling is to bring the externally referenced elements inside the module. However, this only works if the elements are not referenced by other sections of code. Control coupling can be reduced by splitting the routines effected by the flags into seperate procedures or functions. Then the calling routine would make seperate calls to the new routines [Stevens 74].

2. Cohesion

When more than one code segment references the same element, these segments are related. Cohesion measures the strength of relationships between code segments within the same module. It is desirable that modules exhibit strong cohesion. For that reason, related segments should be collected in the same module that contains the referenced element [Stevens 74].

3. Modifiability

A system is modifiable if changes can be made to one segment of code without generating adverse side effects in another segment. Another name for this attribute is stability. The degree that a system is modifiable is directly related to the system's measure of coupling and cohesiveness. A modifiable system is produced through the application of sound implementation techniques to a solid design [Yourdon 80]. Once produced, a modifiable system is easily changed and maintained.

4. Modularity

Modularity is defined as the partitioning of the system into small segments. Creating a modular system also begins during the design phase. A major goal of this process is to design each segment around a particular logical function performed by the system [Parnas 79]. This produces a system exhibiting strong cohesion. Another goal is to minimize the amount of coupling. This is done by using a clean and concise interface with data encapsulation; the hiding of data elements. The success of this process is measured through the ease of implementation and maintenance.

5. Readability

Occasionally, original developers are no longer available after a system's completion [Yourdon 80]. For this reason, systems need to exhibit the same behavior during operation as expressed in the source code. This characteristic is termed readability. Small modules with independent, well defined and clearly documented behavior are the most readable. Therefore it is important for modules to exhibit simplicity and consistency [Scott 93].

6. Robustness

Systems that can detect errors (or exceptions) and recover are considered robust or fault tolerant. This requires the addition of exception handling functions and procedures

to the system. These code segments allow the system to process exceptional conditions such as division by zero or value out of limits [Scott 93].

D. CHARACTERISTICS OF MML-10

MML-10 was an attempt to restructure the system by collecting related functions into seperate modules. Although this strengthened cohesion, improvements were limited due to the usage of global variables [Scott 93]. Almost all of the global variables are declared and initialized in the module *main.c*. However, most global variables are actually used or referenced in one or more different modules. There are also several instances in MML-10 where global flags set in one module effect the behavior of a function in another. Collecting the data elements into one common area also eliminated the requirements for clean interfaces between modules. This has resulted in a tightly coupled system.

To reduce system coupling, the required global variables should be re-located to the modules that reference them. These variables should then be encapsulated with the development of module interfaces.

The MML-10 source code is difficult to read. One reason is poor documentation. MML-10's comments are minimal and often give incomplete descriptions of the code. Also, the short motion control theory nomenclature is used for the system's global variable names. Source code documentation can be improved using a combination of well placed, informative comments and descriptive symbolic names.

Improper use of pointers is another reason MML-10 tends to be confusing and difficult to understand. In many places, global pointers reference global variables while some assignments are improperly type casted. In other places, pointers are used in a cryptic manner. For example, instead of using standard indexing to sequence through an array, pointers are used. These practices should be avoided.

One of the goals of the MML project was to modify the system, making it easily ported to other hardware platforms. This involved converting the current system to a portable

language, such as ANSI C or C++, while minimizing the amount of required assembly code. It also implies that hardware dependencies should be local to a few modules.

However, attempts to reengineer MML-10 by replacing existing non-ANSI C or assembly code with ANSI C failed. For example, Figure 5 shows the addition of two

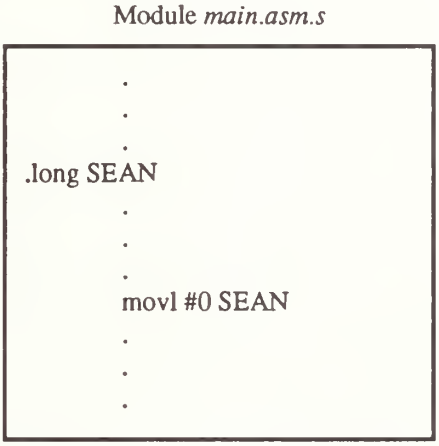

Figure 5: Introduction of a Unique Variable

statements to an assembly language module belonging to the system. The first statement, *.long SEAN*, simply introduces a unique variable, while the second statement, *movl #0,SEAN*, initializes it. Prior to the change, the system executed properly. After adding the two program statements, the resulting system would no longer function. No other changes were made. Similar results occurred when adding or deleting output statements. To create a stable system, MML should be re-implemented using solid engineering techniques.

III. YAMABICO HARDWARE ARCHITECTURE

The Yamabico-11 is a collection of hardware sub-systems assembled on an aluminum frame. As shown in Figure 6, these hardware sub-systems include: a CPU system, a wheels

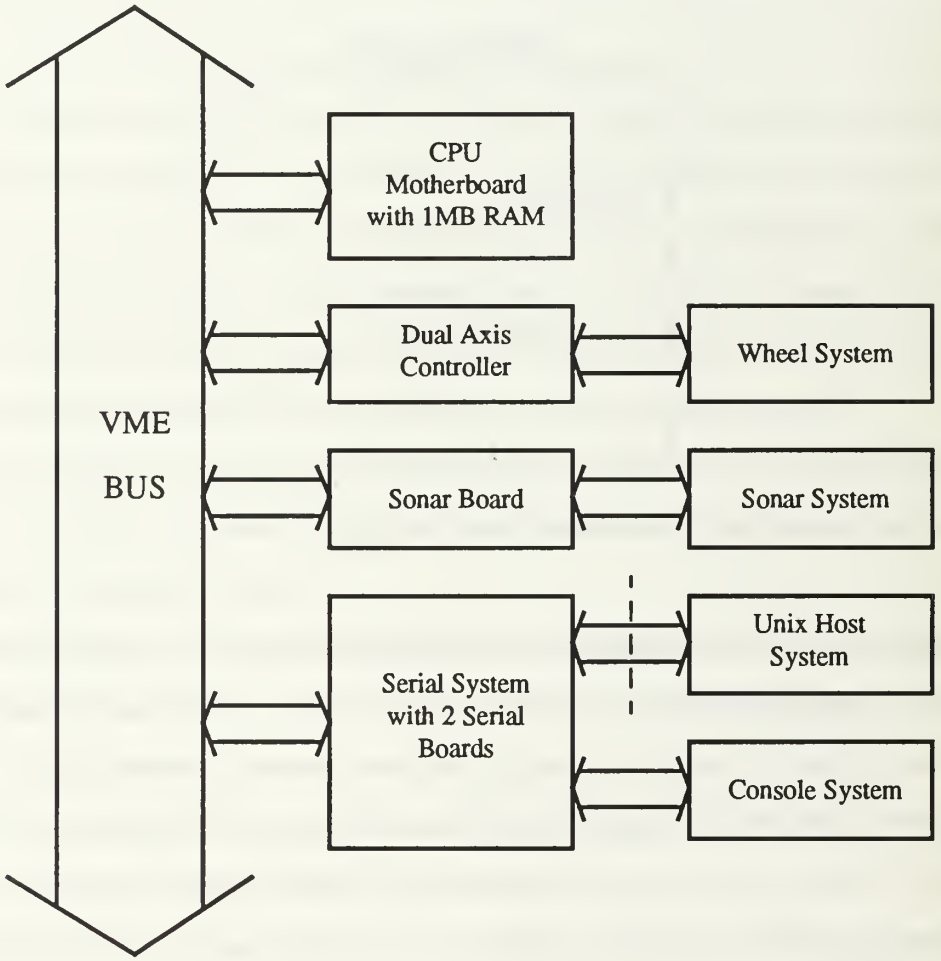

Figure 6: Yamabico-11 Computer Architecture

system, a sonar system, a serial system, a Unix host system and a console system used as the communications interface.

The chassis houses two 12-volt rechargeable batteries. These batteries power all sub-systems except for the console. The chassis rests on four spring-loaded castering wheels for balance while two wheels attached to the chassis in a differential arrangement control robot movement. The actual robot is shown in Figure 7.

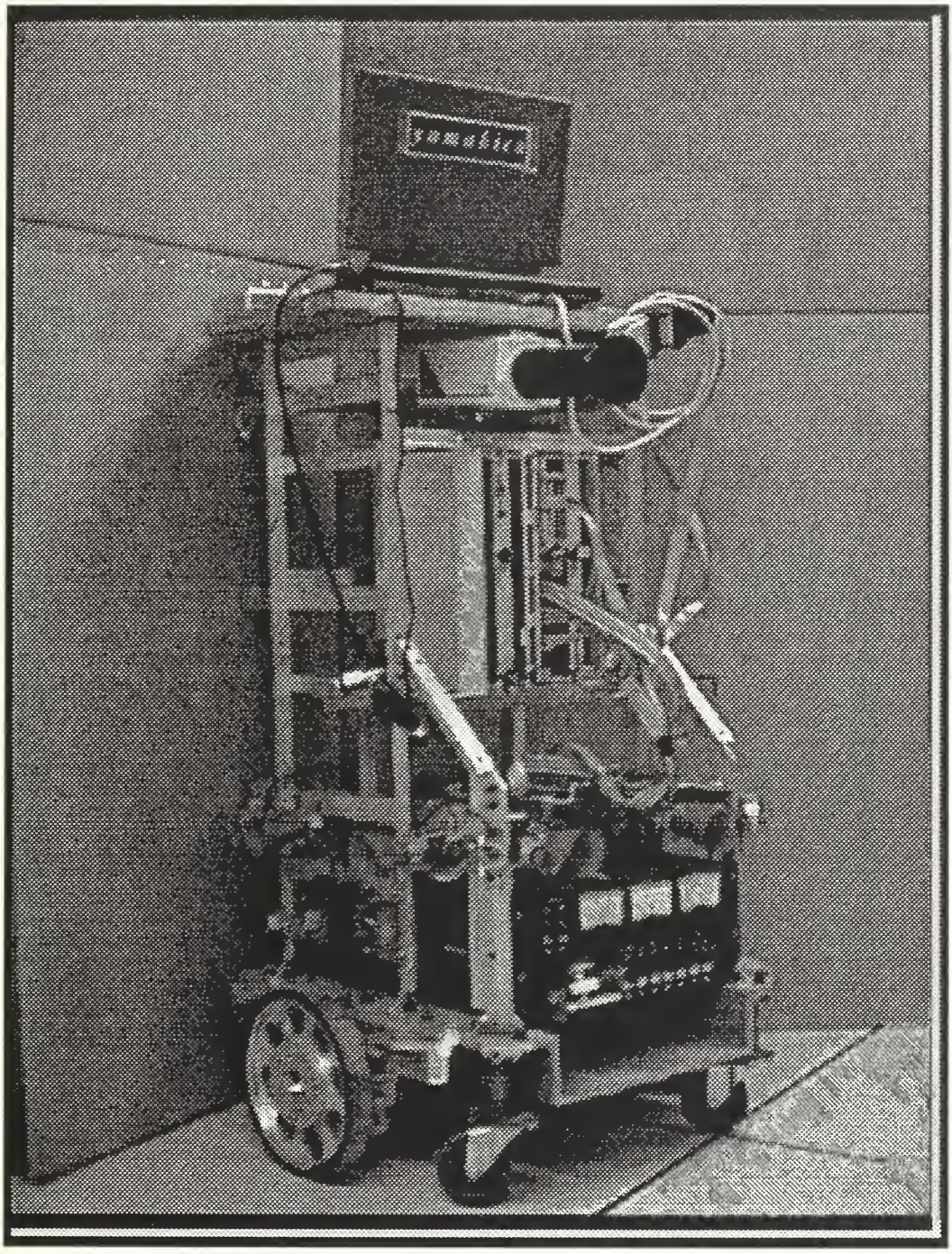

Figure 7: Yamabico-11 Mobile Robot

A. THE CPU SYSTEM

The current CPU sub-system uses a VME7120 32-bit motherboard. The board contains a Motorola based MC68020 CPU operating at 16MHz and a MC68881 co-processor for floating point arithmetic [VCM 86]. The system also includes one megabyte of dynamic memory and a ROM based VME7920 Debugging Package. The debugger is used as the monitor program when the system is powered on [VDP 86]. An upgrade of the sub-system to a SPARC-4 processor board containing 16 megabytes of dynamic memory is planned.

One of the main functions of the CPU is to manage the interrupt driven requests from the other subsystems. As mentioned earlier, the MC68020 CPU provides eight levels of interrupts. The assignment of each interrupt level is summarized in Table 1 [MacPherson 93]. To handle the interrupts, the MC68020 uses a special purpose register to hold the

TABLE 1: MML SYSTEM TASK PRIORITY

Interrupt Level	Interrupt Source	Function	Interrupt Type	Vector	Duration (μs)
7	Stop Button	Reset	Asynchronous	-	-
6	-	Not Used	-	-	-
5	-	Not Used	-	-	-
4	Serial Board 1	Motion	Synchronous	64	2500
3	Serial Board 0	Console	Asynchronous	65	variable
2	Sonar Board	Sonar	Synchronous	66	240
1	Serial Board 0	Debugger	Synchronous	67	-
0	-	User Program	None	-	-

address of a vector table. This table is essentially an array of 256 elements, where each element can store the address of an exception/interrupt handler. When an exception/ interrupt is signaled, the CPU saves the current status word and the status word is modified for interrupt processing. Next, an index value into the vector table is obtained from the interrupting device. Then the current context is saved on a supervisor stack. Finally,

execution resumes at the address located in the vector table cell specified by the index value. When the interrupt handler is finished, the *rte* assembly instruction restores the processor to the state prior to the interrupt [MC68020 85].

B. THE SERIAL SYSTEM

The serial sub-system is composed of two VME8300 Quad Serial Port Boards. Each board contains a VME bus interface and two serial communications controllers. Each controller manages two serial ports. Associated with each port is a timer/counter that may be used for baud rate generation or asynchronous/synchronous interrupt control. There also exists a fifth timer/counter that is primarily used to generate synchronous interrupts. However, each board contains only two latches for a maximum configuration of two levels of interrupts per board [VQS 86]. Figure 8 shows the conceptual layout of a serial board.

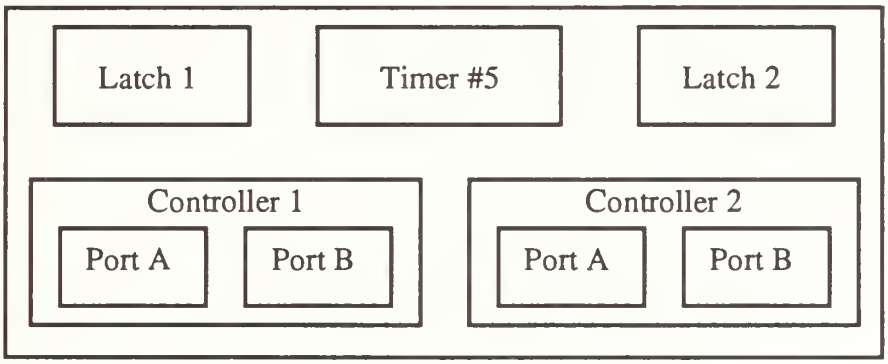

Figure 8: Serial Board Conceptual Diagram

The ports and timers are configured by reading from and writing to registers located on the serial boards. These registers are memory mapped and are accessed through absolute addressing.

C. THE WHEELS SYSTEM

The wheel sub-system consists of two independent DC motors that drive the two wheels in either the forward or reverse direction, controlling the robot's movement.

Braking can also be applied to the motors. Each motor has an associated shaft encoder that is used to determine distance traveled, speed, and to make odometry correction.

A dual axis controller board serves as the interface between the wheel motors, the shaft encoders and the CPU. It contains memory mapped registers that are accessed through absolute addressing. These registers are used to enable/disable the motors, read the shaft encoders and to send pulse width modulation values. A pulse width modulation value is an eight-bit integer that determines the strength of a short electrical pulse sent to a motor to create movement. Since each pulse is short, constant pulse generations are required to produce smooth, continuous motion.

D. THE SONAR SYSTEM

The sonar sub-system contains twelve 40kHz ultrasonic sensors. These sensors are used to gather sonar return information from the forward/rear, lateral or diagonal directions. Three control boards are used to control the sensors and collect return information. A VME bus card is used as the interface between the control cards and the CPU [DeClue 93].

E. THE UNIX HOST SYSTEM

The Unix host system is a Sun-3 workstation using SunOS 4.1.1. It is connected to the robot through a serial port on the first serial board. The host is used to develop software that will be transferred to the robot. It also accepts collected data from the robot. The interface is easily disconnected during robot operation, allowing full motion freedom.

F. THE CONSOLE SYSTEM

A Macintosh Power Book is the main component of the console sub-system and is the only input/output device when Yamabico operates as a self-contained robot. The Power Book is connected to the robot through a serial port on the second serial board and provides the interface between the user and the robot's debug monitor through a software communications package. It also has its own rechargeable power supply.

IV. SYSTEM DESIGN

A. DESIGN GOALS

Initially, the goal of the reengineering process was to add structure to MML-10, replacing non-ANSI C code with ANSI C. However, it was soon realized that a monumental effort would be required. Therefore, a plan to re-design the core system was initiated.

Since MML is used in conducting research by many people, the new design was based on three requirements. First, the system must be easy to read and easy to maintain. Therefore, modularity is a primary goal of the new design. As discussed earlier, a modular system exhibits strong cohesion and loose coupling.

MML will see many modifications and changes as the system continues to evolve and expand. Therefore, the second requirement is system stability. The creation of a stable system begins by applying solid implementation techniques to a modular design. Some important techniques that promote stability are discussed in Chapter VI.

The third requirement is portability. When the Yamabico's processor is upgraded, the new software should only require minor changes. This implies that the assembly code should be minimized and the hardware dependencies localized. A portable system also promotes software re-use for future platforms.

A secondary objective is to create an object oriented type design. This will encourage the smooth transition to an object oriented language such as C++.

B. MODELING NOTATION

When modeling software, the goal of the notation is to produce a picture of the software system that is clear and easy to interpret. It must simplify the system by highlighting the important features, while hiding the details [Constantine 94].

The notation chosen to model the new design is shown in Figure 9. A software system or hardware component is represented by a solid box. Interrupt handlers are shown as

dashed circles while solid circles indicate system processes. In some systems, the interface is not yet finalized. For this reason a single solid circle may represent more than one process. Solid arrows depict messages between systems and/or processes, while dashed arrows display the signals used to activate interrupt handlers and system control processes. Finally, data stores are indicated by two solid parallel lines.

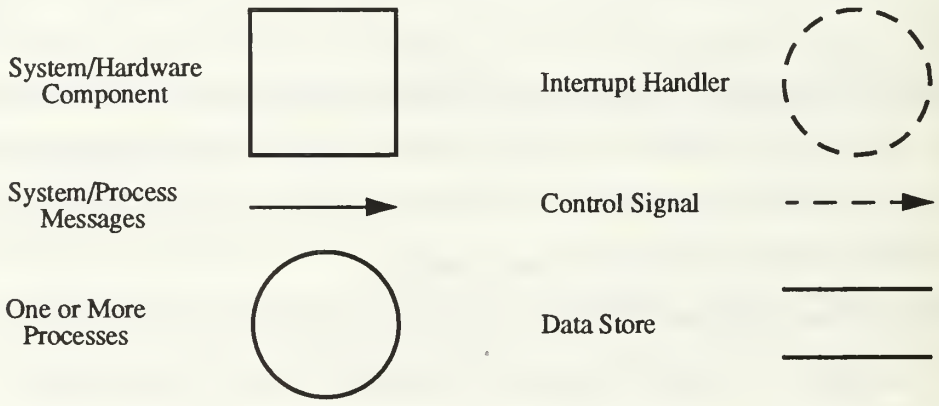

Figure 9: Design Notation

C. SYSTEM DESIGN

1. System Overview

A model of MML is illustrated in Figure 10. This diagram depicts MML as a network of five sub-systems: a CPU system, a motion control system, a terminal system, a sonar system and a user program. This overview is slightly more complex than the views depicted in Figure 1 and Figure 2. However, this approach helps produce the desired object oriented design where each sub-system is treated as an object. This approach results in a simpler system design overall.

2. CPU System

The CPU system, shown in Figure 11, is the primary system. One objective of this system is to initialize the other sub-systems. If the system is implemented with an

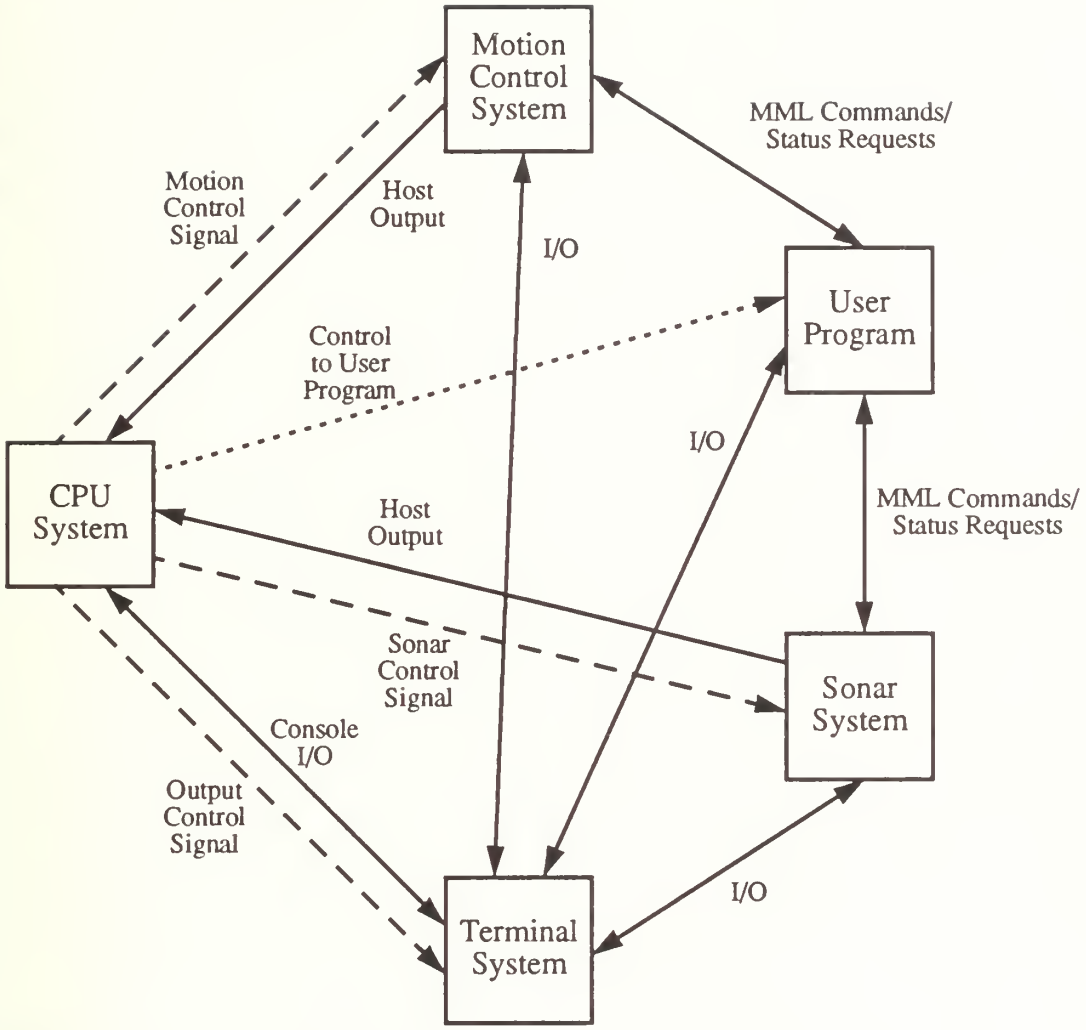

Figure 10: System Network

object oriented language, C++ for example, this can be accomplished when the object is instantiated. However, if a non-object oriented language is used, such as ANSI C, then each sub-system is required to have an initialization process. In the later case, the CPU system must specifically call these initialization routines for each sub-system.

Since the interrupt control mechanism is CPU dependent, the interrupt handling routines and the mechanism setup is part of the CPU system. The objective of this operation

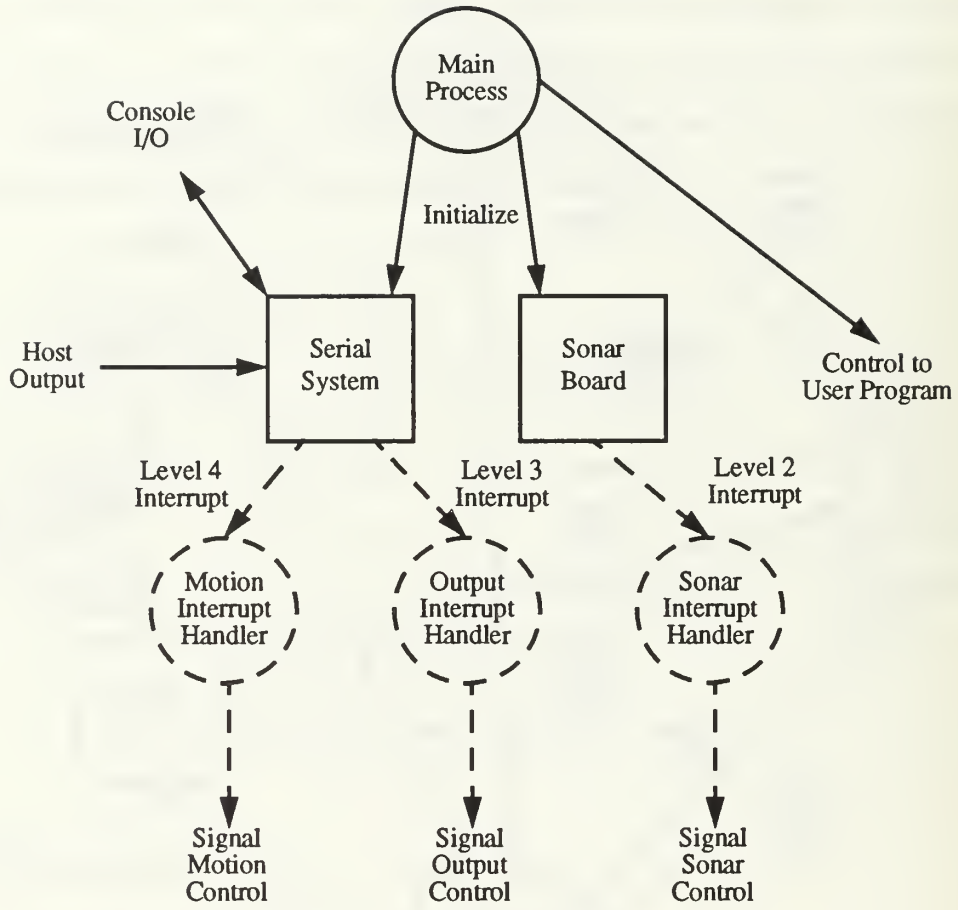

Figure 11: CPU System

is to match the interrupt handling functions to the appropriate interrupt signals. However, the interrupt priority levels are still based on the hardware configuration.

The third function of the CPU system is to pass control to the user program. This is done after the sub-systems have been initialized and the interrupt control process has been properly established.

3. Serial System

Illustrated in Figure 12, the serial system shields the hardware details of the VME8300 serial boards from the other sub-systems in MML. Each serial port or timer must be properly initialized prior to its first use. This is accomplished through a pair of

initialization processes. A set of interface routines are required by each port to read and write character data. These interface routines are modeled after examples found in the user's manual [VQS 86]. Since this system directly accesses the hardware, it must NOT be optimized. If optimization techniques are used, the initialization routines will improperly setup the ports.

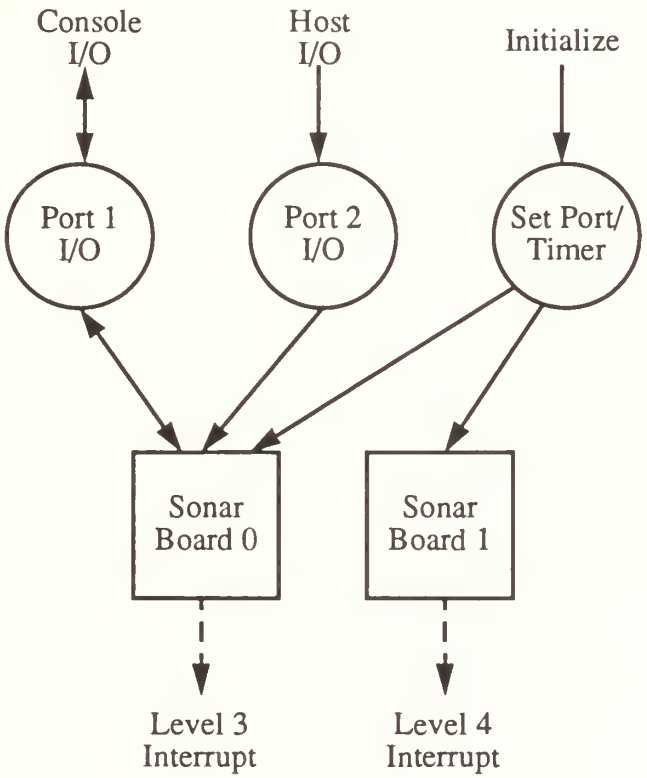

Figure 12: Serial System

The VME8300 boards ability to generate synchronous interrupts is a major benefit. They are required to generate control signals to activate other sub-systems. For this reason the serial system is considered to be a component of the CPU system rather than an MML sub-system (see Figure 11).

4. Motion Control System

The motion control system depicted in Figure 13 serves two primary purposes. First, it provides the interface for MML's immediate and sequential commands. Immediate commands cause instantaneous changes to control variables and are implemented using a single process. A sequential command is only executed after the robot has completed the previous sequential command. Therefore, each sequential command requires a process pair. One process, called from the user program, stores the command in the instruction buffer. The other process executes the command and is called from the motion system control process.

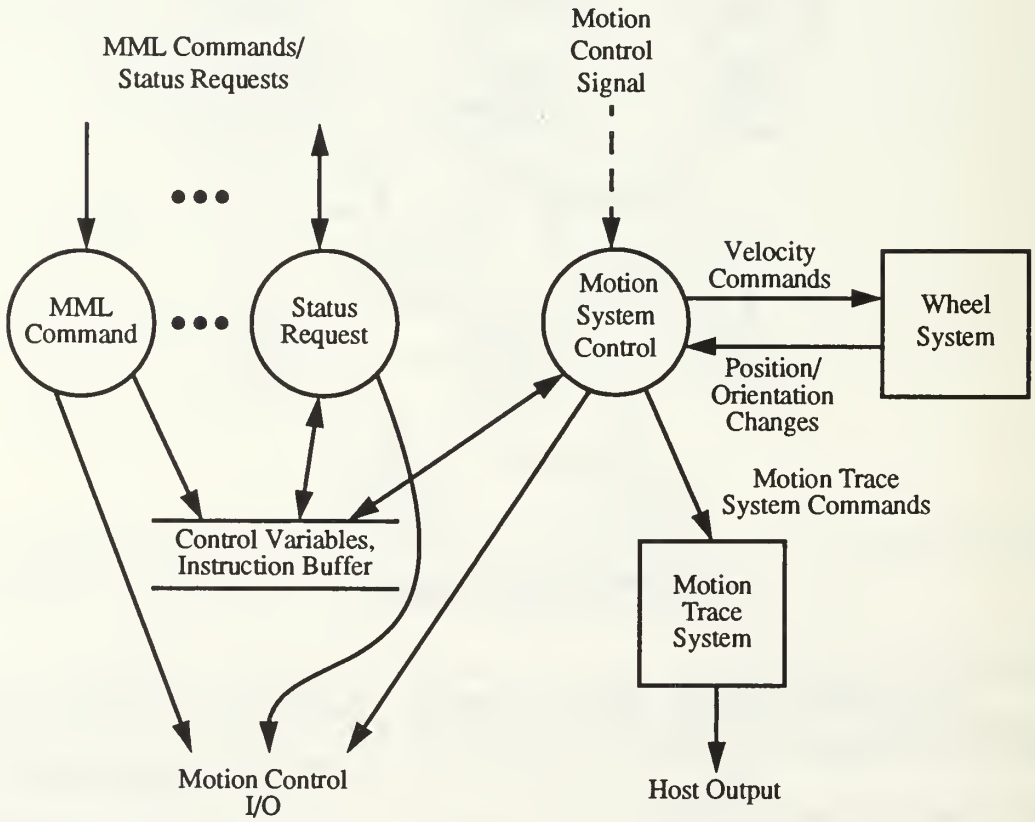

Figure 13: Motion Control System

The system's second purpose is to control the robot's movement. This is done by executing the motion system control process every 10 milliseconds. Each motion control

cycle begins by updating the current configuration using information obtained from the wheel system. Using motion control parameters and the new configuration, it then calculates the desired linear and rotational velocities needed to follow the current path element. These desired velocities are then sent back to the wheel system for execution.

5. Wheel System

As indicated by Figure 13, the wheel system is a component of the motion control system. Presented in Figure 14, the system's primary function is to provide an interface between the motion control system and the dual axis controller. This interface was designed to eliminate knowledge about the robot's architecture from the motion control system. Therefore, the wheel system can easily be replaced by another type of locomotion system.

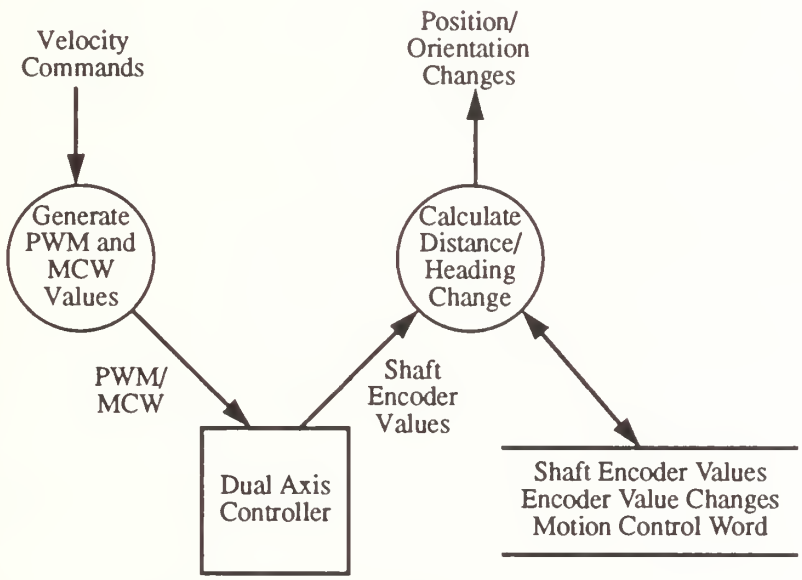

Figure 14: Wheel System

6. Sonar System

As seen in Figure 15, the sonar system's design is similar to the motion control system. It provides the MML sonar command interface between the user program and the

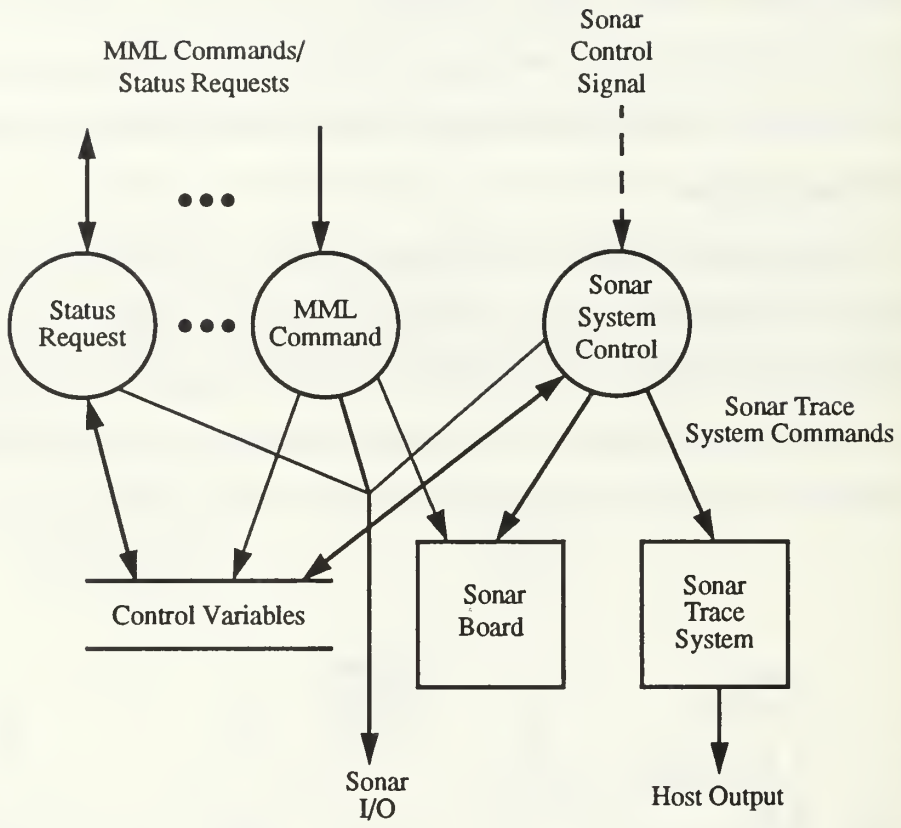

Figure 15: Sonar System

sonar board. Some commands control the boards operation by setting control variables, while others return sonar information to the user program. It also contains the sonar system control process. This process saves the current sonar returns that may be requested by the user program. It can also record the sonar returns by sending the data to the sonar trace system.

7. Tracing Systems

A trace system is a small sub-system used to record selected information. Shown in Figure 16, its interface consists of three parts. First, it contains control routines to initialize and enable/disable the system. The initialization routine sets the size of the logging buffer, while the enable routine sets the frequency that the data is logged. The

second part of the interface consists of the logging routines. These routines save the selected data after ensuring that the buffer is not full. The third component downloads the recorded information to the host Unix system. Only character data can be sent to the host system. Therefore, the download process must convert the data if necessary.

There are two tracing system components in the new design. One is located in the motion control system and is used for debugging. The other is located in the sonar system. This component is used to log sonar return information.

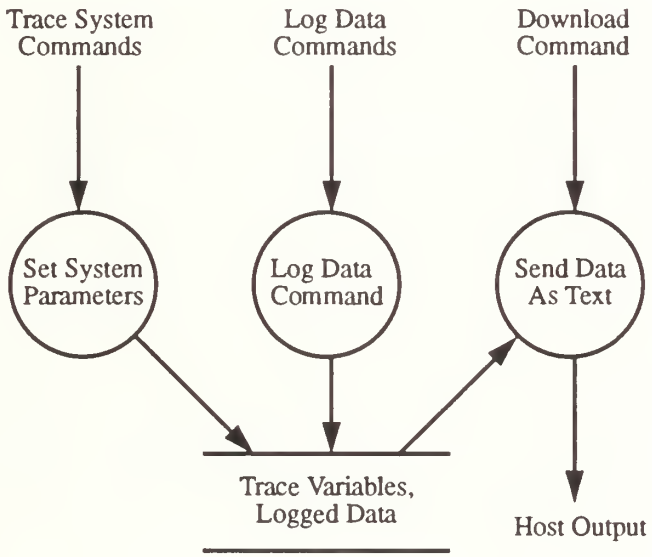

Figure 16: Tracing Systems

8. Terminal System

The transfer of information between a processor and a device, such as a terminal, is time consuming. In real-time systems, it is unacceptable for a high priority system to wait for I/O completion, monopolizing the CPU. However, the ability to display information from these high priority systems is still desirable.

The purpose of the terminal system is to provide any sub-system the capability to display information. As presented in Figure 17, the terminal system accomplishes this

objective using asynchronous interrupt driven output along with a set of I/O routines. This allows a high priority sub-system to continue executing by sending output information to the terminal system for processing. Input is not interrupt driven and should only be used by low priority routines.

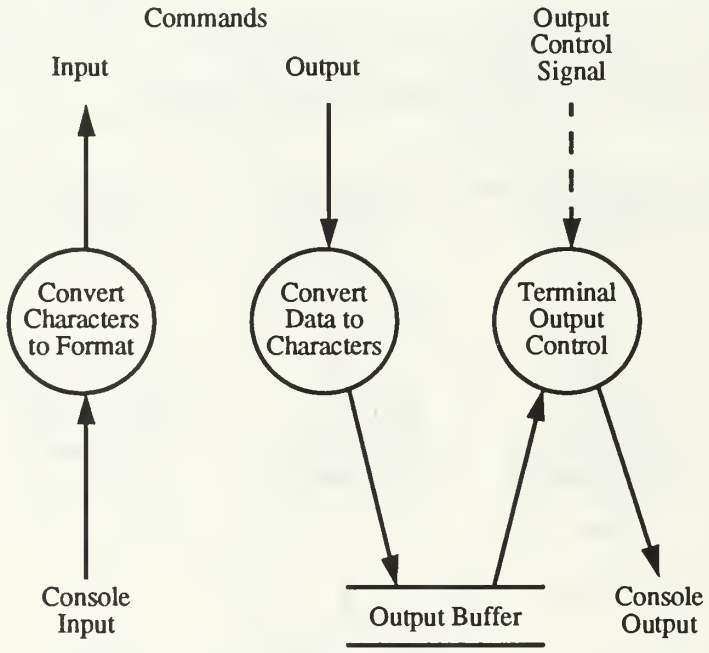

Figure 17: Terminal System

When an output routine is called, the routine converts the data into a character representation if necessary. It then stores the string of characters into an output buffer that uses a queue data structure. The routine then sends a null character directly to the terminal using console output. After a character is sent to the terminal, an interrupt is generated by the hardware requesting another character. This interrupt activates the terminal output control process. The process de-queues the first character from the buffer and sends it to the terminal, which then generates another interrupt. This operation continues until the buffer is empty and the control process terminates the interrupt cycle.

V. YAMABICO SOFTWARE DEVELOPMENT ENVIRONMENT

The current software development environment for the robot is restricted to a single Sun-3 workstation using SunOS 4.1.1. The reason is that this workstation shares the same processor architecture as the robot. The development cycle will improve when the robot's CPU system is upgraded to a SPARC-4 processor due to more advanced development tools, FTP transfer capability and support for system I/O by the resident debugger.

The Yamabico-11 does not use a commercial operating system. This means the user must perform some of the basic operating system functions through the resident debugger. First, the user must load the program into memory. Then control of the CPU must be transferred from the debugger program to the user's program by placing the address of the first instruction to be executed into the program counter (pc). The user must also ensure the program returns control to the debugger when it has terminated. Preparations for these functions begin with the compilation phase.

A. C COMPILERS

The only compilers available for the current development environment are the Unix C Compiler and the GNU Project C Compiler. Both compilers are invoked through a compiler driver.

1. Compiler Drivers

As shown in Figure 18, a compiler driver is a program that creates executable code by sequentially calling the pre-processor, compiler, assembler and linker with the appropriately supplied parameters and files. The driver for the Unix C compiler is *cc*, while the driver for the GNU C compiler is *gcc*. When invoked, the driver passes the C source file to the pre-processor. Output from the pre-processor is then passed to the compiler. Each compiler is essentially a translator, translating the C source file into equivalent assembly code. The compiler driver then passes this assembly code to the assembler to create object code. Next the driver passes the assembler output to the linker to build executable code. The

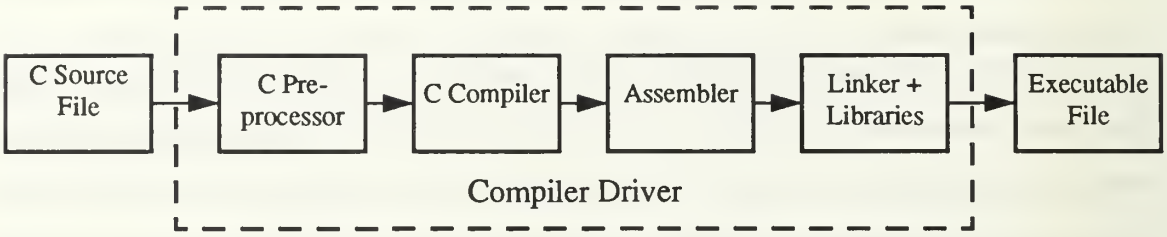

Figure 18: Compiler Driver Process

output from the linker is then stored in a file in the developer's directory. This process can be verified by using compiler flags to display the compiler driver commands. These flags are *-dryrun* for *cc* and *-v* for *gcc* [CPG 88][Stallman 89].

2. Code Optimization

Producing optimized object code is one of the goals of program development. Both compilers will optimize source code if the *-O* flag is used. However, this is not always desirable. There are instances where several values must be assigned to a variable or address in succession, as in initializing serial ports for example. If this optimization function is enabled, the compiler would eliminate all but the last assignment, resulting in an improperly initialized port. This can be verified by inspecting the assembly code translation produced when the flag *-S* is used with *-O*. The *-S* flag instructs the compiler driver to stop after translating the C source file into assembly code, storing the translation in a file. This file will be given the same name as the source code with the exception of a *.s* extension. Inspection of the assembly code will reveal that only the last assignment is retained. Therefore the developer must be careful in deciding which modules can be safely optimized by the compiler [CPG 88][Stallman 89].

Another way of optimizing code is by using '#define' statements to declare constants instead of 'const.' Constants declared using 'const' are stored on the stack during run time. References are implicit, using the stack pointer and an offset. Using the '#define' method causes the compiler's pre-processor to substitute the actual value into the source

code prior to compilation. This is the equivalent to hard coding the values except that it is cleaner and easier to maintain/modify.

3. Standard Libraries

It is assumed that the generated executable code will be run on the platform used for development. Therefore each compiler uses and depends on its own set of standard libraries. These libraries are usually linked with the object code automatically by the compiler driver when creating the executable. However, some of the routines in these libraries make calls to the operating system. Since the robot does not have an operating system, these libraries should not be used. Standard C functions for input/output or file operations such as *printf()* should also not be used. Therefore, the compiler driver should be instructed to stop before the linking phase by using the *-c* flag. It will then save an object file with a *.o* extension for each C source file passed to the compiler driver. These files can then be used in an explicit call to linker without the libraries [CPG 88][Stallman 89].

4. *CC* vs. *GCC*

There is a major difference between the two compilers. The Unix compiler is for source code developed using the K&R standard, a style of developing C source code designed by Brian Kernighan and Dennis Ritchie. This standard was developed prior to the ANSI standard adopted in 1989. The GNU compiler was designed for source code written in ANSI C. The use of function prototypes for parameter checking is one of the major advantages the ANSI standard has over the K&R standard.

5. Using *GCC*

The GNU Project C Compiler does have some peculiarities that must be taken into consideration when developing programs for the robot. First, the compiler generates a call to *__main()* as the first statement in *main()*. Since *gcc*'s libraries are not linked with the object code, the result will be an undefined symbol error during linking. Therefore, the

developer must explicitly create a function called *__main()* that returns void. The function definition should only include a simple return statement.

The compiler may also insert calls to certain standard library routines, such as *memcpy()* and *memset()*. Since the libraries are not included during the linking phase, again the result will be an undefined symbol error. Therefore the user must either develop these functions or build a new library for the robot. The Unix *ar* command can be used to extract copies of these functions from the standard libraries and to build the new library [SRM 88].

Another precaution is associated with how the compiler generates storage for locally declared strings. Storage for strings declared within a function body is allocated immediately prior to the address of the function itself. Currently, this only effects the functions *main()* and *user()* since they must start at an absolute address. This is discussed later in this chapter. Therefore, the developer should not declare local strings in either function.

Two flags that are helpful to use are *-Wall* and *-Wpointer-arith*. The *-Wall* flag instructs the compiler to issue common warning messages. For example messages will be displayed when variables are declared but not used, when functions are implicitly declared as returning an integer or when functions declared to return a non-void value but do not have a return statement in the function definintion. To receive warnings concerning the use of pointer arithmatic, use the *-Wpointer-arith* flag. This allows the developer to find accidental uses of pointer arithmatic [Stallman 89].

B. LINKING

Each object file passed to the Unix linker (*ld*) consists of a text segment and a symbol table. The text segment contains the executable instructions while the symbol table contains the symbols (function names and global variables) that can be accessed by other modules. When multiple object files are used, the linker starts by appending the text segment of the second file to the first's. The text segment of each additional object file is then appended in the order given to create one text segment. All symbols are also collected into one

symbol table [SRM 88]. Figure 19 shows an example object module created from the following command: *ld main.o iosys.o serial.o.*

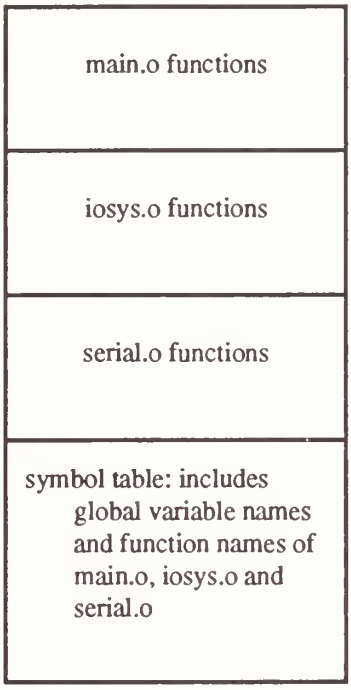

Figure 19: Example of Composite Object File

After all object files have been processed, the linker must calculate the absolute address of each symbol in the symbol table. Addresses for functions are determined using the offset relative to the beginning of the program and the program's starting location. Absolute address calculation for global variables is based on an offset from the next page boundary following the text segment. It then resolves all references to these symbols.

After all references are resolved, the linker creates a load module. This module is a file of records used by the loader. These records contain either instructions or initialized global data along with the address in memory where they are to be located by the loader. All of the records containing text (instructions) are collected and placed prior to the records containing the initialized global data.

1. The *Kernel* Module

Currently, all programs are divided into two load modules. One module is the *kernel* while the other contains the motion commands. The *kernel* contains *main()* and most of the code. As mentioned earlier, the user must know where the program execution begins. It is standard practice to load the *kernel* at hexadecimal address 0x304000. Therefore, the first instruction in the function *main()* must be located at this address. To correctly resolve the symbolic references, the *kernel*'s starting location must be passed to the linker by using the *-T 304000* flag [SRM 88].

Since the creation of the text segment is based on the order of object files passed to the linker, the object file containing the definition of *main()* must be the first file in that list. The order of the remaining files is unimportant. Additionally, since *cc* and *gcc* generate object code as defined in the C source file, *main()* must be the first function defined in its source file. Finally, if local strings are defined in *main()*, *gcc* will reserve memory storage prior to *main()*. Consequently, 0x304000 would be the address of the string instead of the first executable instruction. These last two points can be verified by using the *-S* flag and inspecting the resulting assembly code. For an example, see Figure 20.

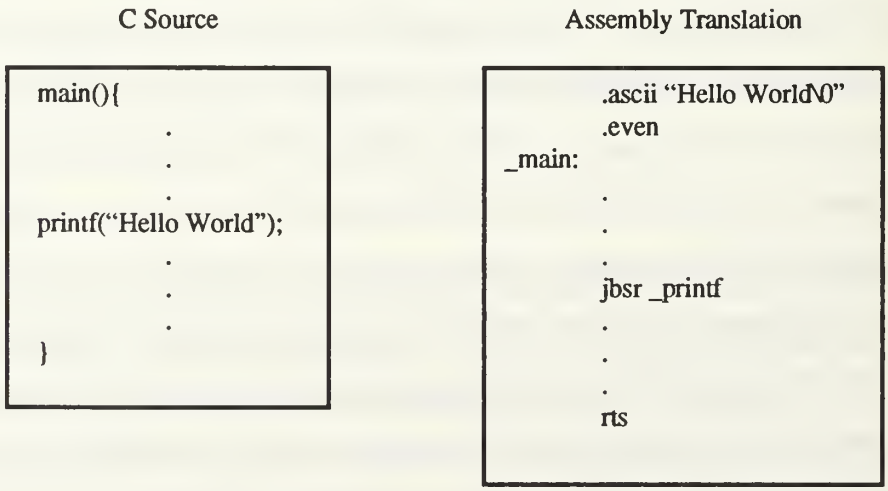

Figure 20: Effects of Compilation on Strings

2. The *User* Module

The *user* module contains MML library function calls. Its construction is similar to the *kernel* with a few exceptions. First, the function *User()* is the primary function instead of *main()*. Second, the module is loaded at hexadecimal address 0x334000 vice 0x304000.

Since *User()* is called by the *kernel*, the *kernel* must know *User()*'s location. However, *User()* must also know the location of the functions in the *kernel* too. Making *User()* accessible to the *kernel* is accomplished by declaring pointer to a function returning void within the *kernel*, and initializing it to 0x334000. To provide access to the *kernel*'s functions, use the *-A kernel* flag to link the *user* module with *kernel*'s symbol table. One note of caution, the *user* module is now dependent on the *kernel*. Therefore any change to the *kernel* requires the *user* module to be re-linked.

C. LOADING

Before a program can be executed, it must be placed into random access memory by a loader. Each record in the load module contains instructions or initialized data information along with an address. The loader then loads the information from each record at the address specified.

However, the loader for the robot is a simple program that essentially dumps the information sequentially from all records beginning at either 0x304000 or 0x334000. This does not interfere with the placement of the program instructions. But the global variable initialization data is erroneously placed at the end of the text segment instead of in the global variables. Therefore, global variables initialized at compile time actually contain unpredictable data when the program is loaded. The solution to this problem is to initialize all data at run-time, prior to executing the program. Figure 21 shows the robot's memory framework after program loading.

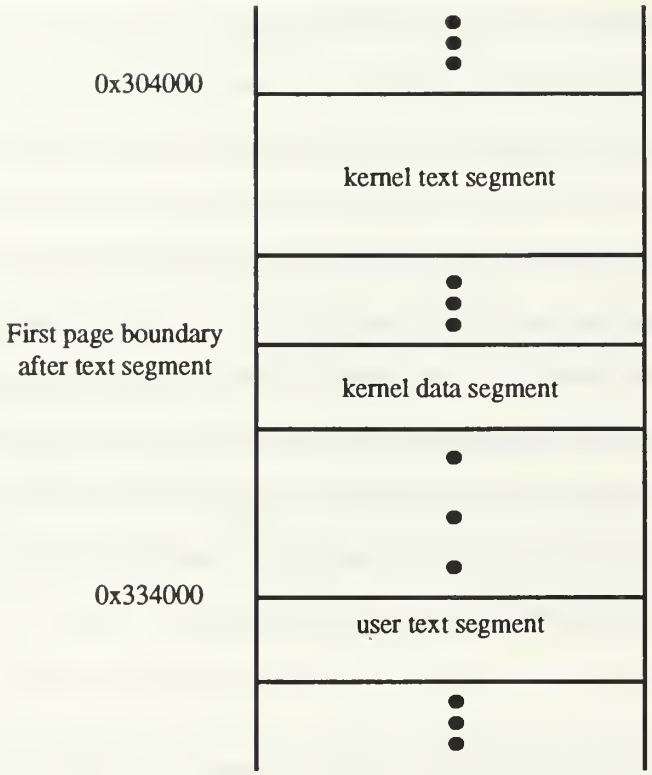

Figure 21: Yamabico-11 Memory Map

D. DEBUGGING TOOLS

1. The Onboard Debugger

As mentioned earlier, the CPU system includes the VME7920 Debugging Package. This monitor program is primarily used to load and execute programs on the robot. However, it can be used to debug programs. The developer can set breakpoints at a particular instruction, halting program execution. The value of variables or registers can be examined and modified. The debugger will even aid the developer by displaying machine code in assembly instruction format. The only requirement is knowledge of the instruction's or variable's memory location [VDP 86].

2. The Unix *nm* Command

To obtain the address of variables or functions from an executable file, the Unix command can be used. The *nm* command displays a files symbol table. For example, entering nm *-n user* will generate a listing containing each symbol (global variable or function name) in *user*'s symbol table along with its location in memory. The *-n* flag instructs the command to sort the list by memory location. This listing gives the developer the memory address locations needed to debug programs with the VME7920 Debugging Package [SRM 88].

VI. SYSTEM IMPLEMENTATION AND TESTING

ANSI C was the language chosen to implement the new design instead of the older K&R version used in MML-10 since it provides stronger type checking through prototypes. Since the two versions are similar, some of MML-10's source code could be ported to the new design with minor modifications. ANSI C also enables a smooth transition of MML to an object oriented paradigm using C++. However, the only ANSI C compiler locally available for the robot's software development environment is GNU's *gcc*.

A. IMPLEMENTATION TECHNIQUES

1. Stability

Producing a stable system is one of the primary objectives during the implementation phase. One method for producing a stable system is to use encapsulation. In C, this is accomplished by declaring global variables and local functions with the *static* specifier. These identifiers can be referenced from sources within the file but prevents external references.

Reducing unnecessary pointers and pointer arithmetic is another way of producing a stable system. Generally, there are only two reasons to call a function. The first reason is to produce some action, such as displaying a value. In this case, actual values are passed as parameters and the function does not return a value. The second reason to call a function is to process information, producing one or more results. In this case, it is common to let the function assign the values directly by passing the addresses of these variables as parameters. However, this can lead to many programming errors. A better method for writing these functions is to use a *return* statement. To return more than a single value, a structure of values should be used.

2. Portability

a. Required Assembly Code

Each assembly instruction corresponds to a single machine instruction. It also produces hardware dependencies since each processor has its own assembly language. Therefore programming in assembly language is very difficult and should be avoided.

One reason to use assembly language programming is to increase the speed of program execution [Schildt 90]. Although MML is a real-time system with timing constraints, it is still a research project. Using assembly language to increase program execution simply adds complexity.

The primary reason to use assembly language is to access specific hardware components or instructions that can not be accessed in any other manner [Schildt 90]. The status register and *rte* instruction are examples of such requirements when using the Motorola 68020 processor. Modification of the status register is handled by instructions used for special registers. The *rte* is used to return from an interrupt handling routine instead of the normal *rts* instruction.

b. Handling Interrupts

Setting up an interrupt mechanism requires two steps. First, an interrupt handler must be created. This routine is a basic shell that must be written in assembly language. The routine must begin by saving all of the CPU's registers onto the stack. If a coprocessor is used, those registers must be saved as well. Then the routine must make a call to a C function that will control the real processing during the interrupt cycle. Once control is returned from the C function, the interrupt handler must restore each register to the value it contained prior to the interrupt. It is important to note that the registers must be restored in the reverse order as they were saved. Finally, the *rte* instruction is used to return from the interrupt.

The second step in establishing an interrupt mechanism is to setup the CPU and interrupting hardware component. This is a two step process. First, the address of the

interrupt handler must be placed in the CPU's vector table. Next, the index value of the table entry is passed to the interrupting device. Once these two steps are complete, the hardware device can be enabled.

c. Absolute Addressing in C

In C, each variable declared as a pointer stores a 32-bit memory address and can be assigned constant values between zero and 2^{32}-1. The amount of memory that a pointer references depends on the pointer's declaration. For example, if a variable is declared as a character pointer, only a single byte, located at pointer's value, will be referenced. But, if the variable is declared as an integer pointer, then four bytes will be referenced by the pointer, starting at the value of the pointer.

The amount of memory that a pointer references can be temporarily changed. Using the cast operator, a pointer is changed to another pointer type. However, the change only lasts for that operation. Afterwards the pointer reverts back to a pointer of its declared form.

3. Readability

One technique for making a system more readable is to use descriptive names for functions, variables and constants. Abbreviations should never be used. When declaring a symbol, two conventions are generally used: separate multiple words with underscores or capitalize the first letter of each word in the symbol. The later convention is used throughout this implementation. Another naming convention utilized in this implementation is to declare or define constant symbols with all capital letters.

Header files are used to declare the prototypes for each function in a module that can be accessed using external calls. This presents the file's interface and should be well documented. The comments should describe what parameters are required and the expected results. A description of the algorithm is not needed. However, constants needed by a C file that may also be referenced by other files are included in its header file also.

Another technique to making a system more readable is to limit the size of the functions to one page. Techniques such as page breaks between functions and good descriptive comments should also be used.

4. Backward Compatibility

To maintain backward compatibility, the new system generates user and kernel object files. The kernel is still loaded at 0x304000 and user is loaded at 0x334000. The absolute address of user() is declared in motorola.asm.s as _user. To prevent the linking error of duplicate symbols, the actual name of the user function in *user.c* should be User() instead of user(). However, to transfer control to the user program, main() is still required to call user() instead of User().

To maintain compatibility with MML-10's immediate command names, a translation header file was created. This is a file containing macros, used by a pre-processor, that replace the old command syntax with the new command syntax. A similar one should be created for other MML-10 commands so that previously created user programs can operate with the new system.

B. SYSTEM IMPLEMENTATION

Due to the scope of the design, it was decided that only a core portion of the new design would be implemented, starting with the CPU system. The system described in this section consists of files with *.c*, *.h*, and *.s* extensions. Those files ending with a *.c* are C source files. These files contain function definitions and encapsulated data structures. The *.h* files are the header files, while the files ending with a *.s* are written in assembly language. With the exception of *main.c*, each C and assembly file has an associated header file. All files are presented in the Appendices and were developed as part of this study except for a few routines found *wheels.c* and *motorola.asm.s*. These routines were ported from MML-10. However, the system does make use of some utility functions, such as memory management and math routines that were developed in earlier projects.

1. The CPU System

The CPU system was implemented first since it is required to initialize all other systems. Appendix A contains the source code for this system. It contains the files: *definitions.h, main.c, system.h, system.c, motorola.h, motorola.asm.s, serial.h,* and *serial.c.*

The *definitions.h* file contains the system's standard type declarations. Some of the new data types refer to constructs used by the motion system. Other declarations define standard assembly language data types, such as BYTE and WORD. These definitions make the code easier to follow.

The file *main.c* only contains main() to prevent the linking problem discussed earlier. This is the function that gets loaded at 0x304000. It initializes the sub-systems and then calls user(), the user program. After the user program is finished, rexit() is called to return control back to the resident debugger.

The Initialization routines called by main() are located in *system.c.* They are used to setup the interrupt mechanisms described earlier. Also included in this file are the some system functions for enabling and disabling interrupts and the __main() function required by the *gcc* compiler.

The *motorola.asm.s* module contains all of MML's required assembly code. The interrupt handling shells described above are located in this module. Also defined here is a routine to change the CPU's interrupt priority level. The rexit() routine is located in this module since it uses a sequence of instructions that are difficult to duplicate using a high level language. The address of user() is also set in this file.

The *serial.c* module contains the serial system. As mentioned earlier, this is a component of the CPU system. It serves as an interface between the serial boards and the other sub-systems. The previously discussed technique for referencing absolute memory locations is evident in this module.

2. The Terminal System

The terminal system was chosen as the next system to implement as this would aid in debugging other systems. The files *iosys.h* and *iosys.c* contain the system's source code and are located in Appendix B. The functions defined in *iosys.c* provide input and output for string, integer and real number data types.

Some of the functions depend on conversion routines. Some routines convert strings to integer or real values. Others convert integer and real values to strings. Notice how the global data is encapsulated by using the *static* specifier. This prevents routines outside this module from accessing these variables.

3. The Motion Control System

Appendix C contains the files: *motion.h, motion.c, motiontrace.h motiontrace.c, wheels.h,* and *wheels.c*. These files define the motion control system. *motion.c* defines the functions that support odometry and velocity control. The wheel system component is defined in the *wheels.c* module while *motiontrace.c* defines motion's tracing system component. The tracing system depends on memory management routines. The motion control rules required for path following are not implemented in this core system.

4. The User Program

A sample *user.c* is included in Appendix D along with its *user.h*. This module shows how to use the core system. Another file associated with the user system is *compatability.h*. This file provides the translation layer for older user programs developed for MML-10.

C. SYSTEM TESTING

Since the system was incrementally developed, testing was conducted at each system modification. This section describes the testing techniques that proved most helpful.

1. Using an External Power Supply

The robot's battery power was often drained due to software testing by several people. System testing was aided by an AC power supply. To test the motion control system, the robot was placed on wooden blocks to prevent movement. The power supply provided sufficient power when testing the CPU and terminal systems. However, it was discovered that the generator did not produce enough power to drive the wheels above moderate speeds. Fast wheel movements caused the CPU to shut down due to insufficient power. So stationary testing was conducted at slow speeds.

2. Output from Interrupt Handlers

Displaying information to the terminal from the motion control system required special handling. The problem was that the information would be sent to the terminal system every 10 milliseconds, causing the output buffer to overflow. The solution was to send the messages at a lower frequency. This was accomplished by using the LoopTest variable in *motion.c* as a counter. Every motion control cycle incremented the LoopTest. When the counter reached the frequency value of 100, it would be reset to zero. All output messages were then placed in a conditional block that checked the value of LoopTest. Only when it was zero would the message be sent to the terminal system. The result is that the message would be sent to the terminal every second instead of every 10 milliseconds.

3. Measurements

The first part of the motion control system to be implemented was the wheel system component. Manual calculations were required to verify the operation of this system. To verify the change in distance and orientation information, the robot was manually pushed along a path with known distance. Several trials were performed using paths of different lengths. When the robot's results were compared to the path lengths measured with a measuring tape, the differences were consistently within one centimeter.

To verify the velocity inputs to the wheel system, the tracing component is used. Figure 22 is an sample of results obtained by changing the robot's commanded velocity

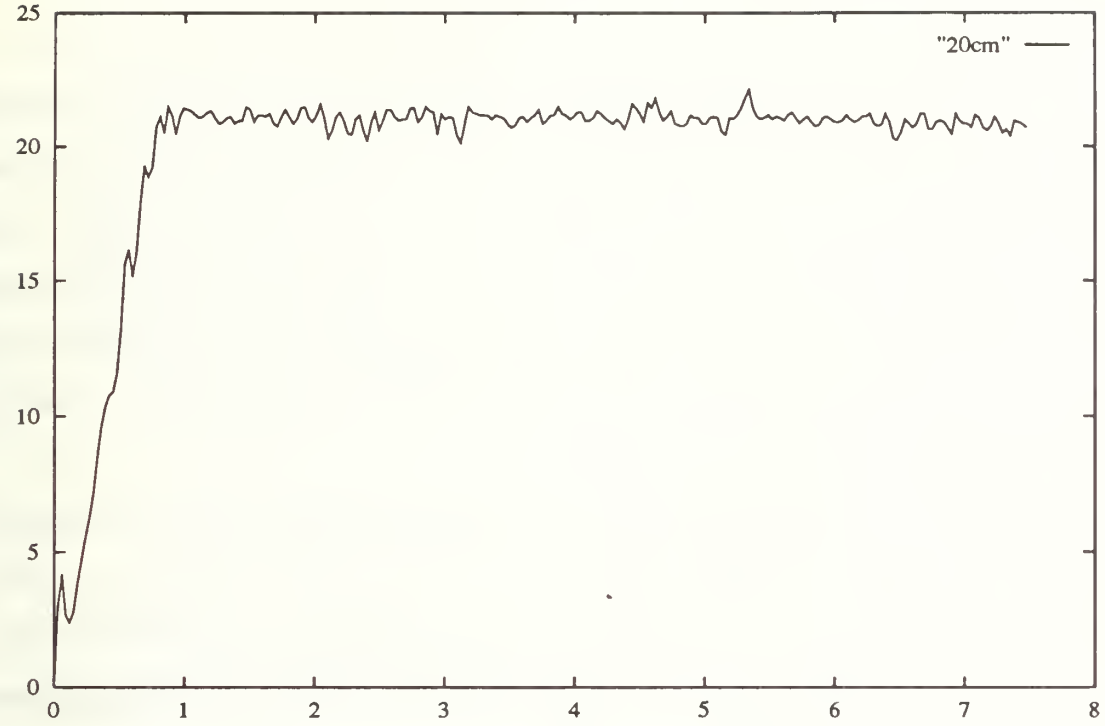

Figure 22: Velocity Control Results

from zero to 20 cm/s. The results show that the actual velocity is one to two centimeters per second faster than the commanded velocity. This difference was found at all speeds tested and is the result of a lookup table used to calculate pwm values. This table is tailored for this specific system and requires further testing to correct the discrepancies.

VII. CONCLUSIONS

A. RESULTS

The end product is a solid core system, capable of serving as a foundation for further MML research. The system is implemented using a standardized language. Hardware dependencies are localized and the required assembly code is reduced to a single module. This results in a system that is very portable. Each module is provided with a clean interface making the system very modular. With limited pointer references and data encapsulation the system is also very stable. The result is a system that is very readable, easy to maintain and easy to modify.

One of the underlying themes presented in this study is pointer reduction. This is not to imply an unfamiliarity with the C language or the use of pointers. In fact the author is aware of the value of using pointers and has been programming with pointers in the C language since 1985. However, many of the researchers on the MML project do not possess the same experience level. This has resulted in improper pointer usage, thereby increasing program complexity and decreasing stability. This is the primary reason for recommending the reduction pointer usage.

B. RECOMMENDATIONS

The next step for this new system is to complete the motion control system. This requires the implementing of the motion control rules to provide path following. It also requires the addition of the sequential commands and the instruction queue.

Following the completion of the motion control system, the sonar system should be implemented. This will involve writing the interrupt handler shell and control process. The assembly code should also be converted to ensure portability.

Another potential project is to develop a simulator based on the new system. Due to the system's modularity, this should only require the simulation of the interrupt mechanisms and the hardware dependent code.

APPENDIX A

A. DEFINITIONS.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 4, 1994
  File Name:  definitions.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains standard definitions and data type
          declarations used throughout the reset of the MML system.
*****************************************************************************/

#ifndef __DEFINITIONS_H

#define __DEFINITIONS_H


#define MOTION_CONTROL_CYCLE 0.01

#define MAX_REAL_PRECISION 15
#define MAX_INTEGER_DIGITS 19

typedef enum {FALSE = 0, TRUE} BOOLEAN;

typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long LONG;
typedef unsigned long* ADDRESS;


typedef struct{
  double Linear;
  double Rotational;
} VELOCITY;


typedef struct{
  double XPosition;
  double YPosition;
} POINT;


typedef struct{
  POINT Position;
  double Orientation;
  double Kappa;
```

```
} CONFIGURATION;


#endif
```

B. MAIN.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 4, 1994
  File Name:  main.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains main(). Its purpose is to initialize all
          sub-systems and then pass control to user(). Once user() is
          complete, the routine returns control to the resident debugger.
*****************************************************************************/

#include "definitions.h"
#include "motorola.h"
#include "system.h"
#include "memsys.h"
#include "iosys.h"
#include "serial.h"
#include "motion.h"
#include "user.h"

void user();

void main(){

  CpuSysInitialize();

  ResetSerialBoards();

  SetInterruptPriority(7);

  InitializeMemSys();

  InitializeConsole();
  InitializeHost();

  IoSysInitialize();
  MotionSysInitialize();

  SetInterruptPriority(0);

  user();
```

```
    rexit();
}
```

C. SYSTEM.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 4, 1994
  File Name:  system.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the prototypes for routines that are hardware
          dependent.
*****************************************************************************/

#ifndef __SYSTEM_H

#define __SYSTEM_H


/*****************************************************************************
  Function __main() is an empty function. It is required if gcc is used since
  gcc will insert a call to __main() within main().
*****************************************************************************/
void __main();

/*****************************************************************************
 Function CpuSysInitialize() is used to set up any requirements that are
 specific to the Motorola 68020 CPU.
*****************************************************************************/
void CpuSysInitialize();

/*****************************************************************************
  Function EnableInterrupts() turns on interrupt servicing.
*****************************************************************************/
void EnableInterrupts();

/*****************************************************************************
  Function DisableInterrupts() turns off interrupt servicing.
*****************************************************************************/
void DisableInterrupts();

/*****************************************************************************
  Function SetConsoleIntMechanism() establishes the console's interrupt
  driven output mechanism.
*****************************************************************************/
void SetConsoleIntMechanism();
```

```
/*****************************************************************************
  Function SetMotionIntMechanism() establishes the synchronous interrupt
  mechanism required for motion control.
*****************************************************************************/
void SetMotionIntMechanism();

#endif
```

D. SYSTEM.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 4, 1994
  File Name:  system.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the required cpu specific routines. The file
          is seperated into two sections: private and public. The private
          section contains the encapsulated data while the public section
          contains the routines available to other systems/modules.
*****************************************************************************/

#include "definitions.h"
#include "motorola.h"
#include "serial.h"
#include "system.h"


static int SavedInterruptMask;
static BOOLEAN InterruptsOn;


/*****************************************************************************
 Routine ___main is required when using the 'gcc' compiler. This is because the
 compiler inserts a call to this routine at the beginning of the main function
 defined for the program. This is normally taken care of by linking in the
 bootstrap object modules, however these are not added to a program that
 operates without an operating system such as the mml program. Therefore, since
 this routine is called, the only requirement is for this routine to simply
 return back to the main program.
*****************************************************************************/

void __main(){
  return;
}
```

```
/*****************************************************************************
  Function CpuSysInitialize() is used to set up any requirements that are
  specific to the Motorola 68020 CPU.
*****************************************************************************/

void CpuSysInitialize(){
   ADDRESS VectorTableBaseAddress = GetVectorBase();

   /*************************************************************************
   * Use a blank interrupt handler to recover from unexpected exceptions
   * such as when the status word is set to a specific value in function
   * SetInterruptPriority. So enter the address of that interrupt handling
   * routine into the 255th entry of the exception/interrupt vector table.
   *************************************************************************/

   *(VectorTableBaseAddress + 255) = (LONG)Unexpected;

   InterruptsOn = TRUE;
   SavedInterruptMask = 0;
}



/*****************************************************************************
   Function EnableInterrupts() allows other systems to reset the interrupt mask
   to the previously stored value.
*****************************************************************************/

void EnableInterrupts(){
   if(InterruptsOn == FALSE){
      InterruptsOn = TRUE;
      SetInterruptPriority(SavedInterruptMask);
   }
}



/*****************************************************************************
   Function DisableInterrupts() allows other systems disable all interrupt
   servicing. The current interrupt mask is saved so that it can be restored
   later.
*****************************************************************************/

void DisableInterrupts(){
   if(InterruptsOn == TRUE){
      SavedInterruptMask = SetInterruptPriority(7);
      InterruptsOn = FALSE;
   }
}
```

```
/*****************************************************************************
  Function SetConsoleIntMechanism() establishes the console's interrupt
  driven output mechanism.
*****************************************************************************/

void SetConsoleIntMechanism(){
  ADDRESS VectorTableBaseAddress = GetVectorBase();

  DisableInterrupts();

  /* Enter the address of the interrupt handling routine into the 65th */
  /* entry of the exception/interrupt vector table of the CPU.       */
  *(VectorTableBaseAddress + 65) = (LONG)IoSysHandler;

  SetLatch(0xffff0031,65);

  EnableInterrupts();
}



/*****************************************************************************
  Function SetMotionIntMechanism() establishes the synchronous interrupt
  mechanism required for motion control.
*****************************************************************************/

void SetMotionIntMechanism(){
  ADDRESS VectorTableBaseAddress = GetVectorBase();

  /* Formula:  counter_value = (4e06MH / 16 frquency divider) * Interrupt_Interval */
  /*     ex:  (4e06/16) * 0.01 seconds = 2500                                    */
  SetTimer(TIMERADDRESS_1,5,0x1c61,2500,2500);

  DisableInterrupts();

  /* Enter the address of the interrupt handling routine into the 64th */
  /* entry of the exception/interrupt vector table of the CPU.       */
  *(VectorTableBaseAddress + 64) = (LONG)MotionSysHandler;

  SetLatch(0xffff0141,64);

  EnableInterrupts();
}
```

E. MOTOROLA.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 3, 1994
  File Name:  motorola.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the prototypes to the required assembly
          language routines.
*****************************************************************************/

#ifndef __MOTOROLA_H

#define __MOTOROLA_H

#include "definitions.h"

/*****************************************************************************
  Function rexit() is used to safely return to the debugging routine used on the
  Yamabico's processor. It is a required call in order to run the kernel again
  without having to reload it.
*****************************************************************************/
void rexit();


/*****************************************************************************
  Function GetVectorBase() returns the starting address of the interrupt vectors
  to the calling routine. This is needed in order to calculate the positions to
  place the interrupt handler addresses.
*****************************************************************************/
ADDRESS GetVectorBase();


/*****************************************************************************
  Function SetInterruptPriority() sets the interrupt priority level to the value
  passed in as a parameter, while returning the old priority level to the
  calling function. The return value can then be used to reset the priority
  level at a later call. The parameter must be a 4-byte integer, and the value
  must be in the range 0-7. If it is out of range, a -1 is returned to indicate
  an error.
*****************************************************************************/
int SetInterruptPriority(int);


/*****************************************************************************
  Interrupt handler _Unexpected is used to handle the unexpected exceptions
  raised during execution of the program. Explicitly changing the status word
  is an example of such an exception.
*****************************************************************************/
```

```
void Unexpected();

/*****************************************************************************
  Interrupt handler _IoSysHandler is used to handle the interrupts from the
  serial port that is set up to handle the console I/O. It is really a shell
  routine needed to call the C function that does the real work.
*****************************************************************************/
void IoSysHandler();

/*****************************************************************************
  Interrupt handler _MotionSysHandler is used to handle the interrupts from a
  timer on the serial board set for synchronous interrupts.
*****************************************************************************/
void MotionSysHandler();

#endif
```

F. MOTOROLA.ASM.S

```
##############################################################################
# Author(s):  Scott Book
# Project:    Yamabico Robot Control System
# Date:       December 8, 1993
# Revised:    March 3, 1994
# File Name:  motorola.asm.s
# Environment: Sun-3 assembler for the motorola 68020 processor
# Description: This file contains the only required assembly lanaguage for the
#           MML system. The main purpose of this file is to define the
#           system routines that can not be defined using a higher level
#           language.
##############################################################################


# The following declaration is necessary when using cc with the -f68881 argument
# since the corresponding library is not linked in. It can be eliminated if gcc
# is used.
    .comm    f68881_used, 4

# Defines the address of user().
    .globl      _user
_user= 0x334000

    .globl      _rexit
    .globl      _GetVectorBase
    .globl      _SetInterruptPriority
    .globl      _Unexpected
```

```
    .globl      _IoSysHandler
    .globl      _MotionSysHandler
    .data
    .text

#############################################################################
# Routine _rexit is used to safely return to the debugging routine used on the
# Yamabico's processor.
#############################################################################
    .even
_rexit:
    trap        #15
    .word       0x0063
    rts


#############################################################################
# Routine _GetVectorBase returns the starting address of the interrupt vectors
# to the calling routine. This is only needed by higher level languages that
# can't read the vector base register (vbr) directly.
#############################################################################
    .even
_GetVectorBase:
    link        a6,#0               | When entering an assembly subroutine, use the
    moveml      #0,sp@              | link command to preserve the previous address
                                    | in the stack pointer. This makes parameter
                                    | passing and clean up simpler. The previous
                                    | contents of a6 are then pushed onto the stack.

    clrl        d0                  | Return the starting address of the vector
    movc        vbr,d0              | interrupt table to the calling routine.

    unlk        a6                  | If the link command was used, ensure that the
    rts                             | unlk command is also used to restore the
                                    | previous contents of both the stack pointer
                                    | and a6 when the subroutine was entered.


#############################################################################
# Routine _SetInterruptPriority sets the interrupt priority level to the value
# passed in as a parameter, while returning the old priority level to the
# calling function. The return value can then be used to reset the priority
# level at a later call. The parameter must be a 4-byte integer, and the value
# must be in the range 0-7. If it is out of range, a -1 is returned to indicate
# an error.
#############################################################################
    .even
_SetInterruptPriority:
    link        a6,#0               | When entering an assembly subroutine, use the
    moveml      d1-d2,sp@-          | link command to preserve the previous address
                                    | in the stack pointer. This makes parameter
```

```
                                | passing and clean up simpler. The previous
    clrl    d0                  | contents of a6 are then pushed onto the stack.
    clrl    d1                  | Also save and clear any registers that might
    clrl    d2                  | be used in the routine.

    movl    a6@(8),d2           | Get the first parameter.

    cmpl    #7,d2               | Ensure that the parameter is between 0 and 7.
    bgt     L10                 | If it's not, then branch to the error area. If
    cmpl    #0,d2               | it is, then place the value in the correspond-
    blt     L10                 | ing interrupt priority area of the status
    lsll    #8,d2               | register

    movw    sr,d0               | Get the current status word and use it to
    movw    d0,d1               | construct the new status word with the
    andw    #0xF0FF,d1          | interrupt priority greater than or equal to
    orw     d2,d1               | the previous interrupt priority. All other
    movw    d1,sr               | bits remain unchanged.

    andw    #0x0700,d0          | Finally, get the value of the previous
    lsrl    #8,d0               | interrupt priority by selecting the priority
    bra     L20                 | bits and then return the value in register d0.

L10:movl    #-1,d0              | An error has occured, so return a negative
                                | value.

L20:moveml  sp@+,d1-d2          | Restore the registers that were saved on
    unlk    a6                  | entry. If the link command was used, ensure
    rts                         | that the unlk command is also used to restore
                                | the previous contents of both the stack
                                | pointer and a6 when the subroutine was
                                | entered.

#############################################################################
# Interrupt handler _Unexpected is used to handle the unexpected exceptions
# raised during execution of the program. Explicitly changing the status word
# is an example of such an exception.
#############################################################################
    .even
_Unexpected:
    rte


#############################################################################
# Interrupt handler _IoSysHandler is used to handle the interrupts from the
# serial port that is set up to handle the console I/O. It is really a shell
# routine needed to call the C function that does the real work. All of the
# system registers must be pushed onto the stack prior to calling the C function
# since the compiler may or may not save the contents of the registers prior to
# use, thus possibly corrupting any data in the functions that were interrupted.
```

```
# The restoration of the register must be done in reverse order!!
#############################################################################
    .even
_IoSysHandler:
    link        a6,#-184            | When entering an assembly subroutine, use the
    fsave       a6@(-184)           | link command to preserve the previous address
    fmovemx     fp0-fp7,sp@-        | in the stack pointer. This makes parameter
    fmovel      fpcr,sp@-           | passing and clean up simpler. The previous
    fmovel      fpsr,sp@-           | contents of a6 are then pushed onto the stack.
    fmovel      fpiar,sp@-          | Then save all of the system registers.
    moveml      d0-d7/a0-a5,sp@-

    movel       #-65529,a0          | Clear the B control/status register of
    moveb       a0@,d0              | port 2 for console output (as per example
                                    | in serial pot manual).

    jsr         _IoSysControl       | Call the C function that is the real work
                                    | horse of the interrupt handler

    moveml      sp@+,d0-d7/a0-a5    | Restore system registers in reverse order.
    fmovel      sp@+,fpiar          | If the link command was used, ensure that the
    fmovel      sp@+,fpsr           | unlk command is also used to restore the
    fmovel      sp@+,fpcr           | previous contents of both the stack pointer
    fmovemx     sp@+,fp0-fp7        | and a6 when the subroutine was entered.
    frestore    a6@(-184)
    unlk        a6
    rte


#############################################################################
# Interrupt handler _MotionSysHandler is used to handle the interrupts from the
# serial board timer that is set up to generate synchronous interrupts for motion
# control. It is really a shell routine needed to call the C function that does
# the real work. All of the system registers must be pushed onto the stack prior
# to calling the C function since the compiler may or may not save the contents
# of the registers prior to use, thus possibly corrupting any data in the
# functions that were interrupted. The restoration of the register must be done
# in reverse order!!
#############################################################################
    .even
_MotionSysHandler:
    link        a6,#-184            | When entering an assembly subroutine, use the
    fsave       a6@(-184)           | link command to preserve the previous address
    fmovemx     fp0-fp7,sp@-        | in the stack pointer. This makes parameter
    fmovel      fpcr,sp@-           | passing and clean up simpler. The previous
    fmovel      fpsr,sp@-           | contents of a6 are then pushed onto the stack.
    fmovel      fpiar,sp@-          | Then save all of the system registers.
    moveml      d0-d7/a0-a5,sp@-

    jsr         _MotionSysControl   | Call the C function that is the real work
                                    | horse of the interrupt handler
```

```
    moveml   sp@+,d0-d7/a0-a5    | Restore system registers in reverse order.
    fmovel   sp@+,fpiar          | If the link command was used, ensure that the
    fmovel   sp@+,fpsr           | unlk command is also used to restore the
    fmovel   sp@+,fpcr           | previous contents of both the stack pointer
    fmovemx  sp@+,fp0-fp7        | and a6 when the subroutine was entered.
    frestore a6@(-184)
    unlk     a6
    rte
```

G. SERIAL.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 2, 1994
  File Name:  serial.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the prototypes/interface for the available
          serial system for the VME8300 Quad Serial Port
          Board.
*****************************************************************************/

#ifndef __SERIAL_H

#define __SERIAL_H

#include "definitions.h"

#define TIMERADDRESS_0 0xffff0011
#define TIMERADDRESS_1 0xffff0111

#define CONSOLE 0xffff0001
#define HOST 0xffff0005
#define YSB1 0xffff0121
#define YSB2 0xffff0125


/*****************************************************************************
  Function ResetSerialBoards() resets both VME8300 Quad Serial Port Boards.
*****************************************************************************/
void ResetSerialBoards();


/*****************************************************************************
  Function InitializeConsole() prepares the port that connects the onboard
  console with the robot for I/O.
*****************************************************************************/
```

```
void InitializeConsole();

/*****************************************************************************
  Function InitializeHost() sets up the port that connects the Unix work-
  station with the robot to be used for object code and data transfers.
*****************************************************************************/
void InitializeHost();

/*****************************************************************************
  Function InitializePort() establishes the communications of the given Port
  to the parameters given. Baud can be assigned standard communication rates
  300, 1200, 2400, 4800, 9600, or 19200. HandShaking, TxInterrupt, and
  RxInterrupt are used as ON/OFF flags. An assigned value of zero will disable
  these options, while any non-zero value will enable them.
*****************************************************************************/
void InitializePort(LONG Port,
                    LONG Baud,
                    LONG HandShaking,
                    LONG TxInterrupt,
                    LONG RxInterrupt);

/*****************************************************************************
  Function SetTimer sets and starts a particular timer to generate interrupts
*****************************************************************************/
void SetTimer(LONG TimerAddress,
              LONG CounterNumber,
              LONG ModeCmd,
              LONG LoadCmd,
              LONG HoldCmd);

/*****************************************************************************
  Function SetLatch() passes to the specified latch the index value of the
  vector table entry  containing the interrupt handlers address to the serial
  board.
*****************************************************************************/
void SetLatch(LONG LatchAddress, BYTE VectorNumber);

/*****************************************************************************
  Function PutConsole() initiates the printing of the single character Source
  to the console.
*****************************************************************************/
void PutConsole(char Source);

/*****************************************************************************
  Function PutHost() sends a single character to the port connected to the host
```

```
  workstation.
*****************************************************************************/
void PutHost(char Source);

/*****************************************************************************
  Function GetConsole() returns a single character typed from the console.
*****************************************************************************/
char GetConsole();

/*****************************************************************************
  Function TerminateConsoleOutput() signals the console port that the interrupt
  generated by the port has been serviced.
*****************************************************************************/
void TerminateConsoleOutput();

/*****************************************************************************
  Function TerminateConsoleIntCycle() tells the console port to terminate all
  unserviced interrupts.
*****************************************************************************/
void TerminateConsoleIntCycle();


#endif
```

H. SERIAL.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 2, 1994
  File Name:  serial.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the I/O routines necessary to interface
          between the VME8300 Quad Serial Port Board and the functions of
          the I/O System. The details of hardware setup and initialization
          followed the examples in the VME8300 Quad Serial Port Board
          Users Manual. The file is seperated into two section. The first
          is the private section containing the encapsulated data and
          functions. The second section is the public section. This
          section defines the interface routines.
  WARNING!!!This system must not be optimized when it is compiled. There are
          places where it is necessary to write several values to the same
          address in succession. If an optimization technique is used, the
          compiler could discard all but the last write. This would cause
          the serial board to be improperly setup.
```

```
*****************************************************************************/

#include "definitions.h"
#include "system.h"
#include "motorola.h"
#include "iosys.h"
#include "serial.h"



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

  PRIVATE SECTION

  The following section defines the encapsulated definitions, data structures
  and prototypes used in the system.

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

#define ARM 0x60
#define DISARM 0xc0
#define LOAD 0x40

#define CONTROLREGISTER_0 0
#define CONTROLREGISTER_1 1
#define CONTROLREGISTER_2 2
#define CONTROLREGISTER_3 3
#define CONTROLREGISTER_4 4
#define CONTROLREGISTER_5 5
#define CONTROLREGISTER_6 6
#define CONTROLREGISTER_7 7


/*****************************************************************************
  The following static function declarations are the prototypes for the
  encapsulated functions.
*****************************************************************************/
static void SetDataTransfer(LONG, LONG, LONG, LONG);
static void PutB(char Data, int MemoryAddress);



/*****************************************************************************
  Function SetDataTransfer() establishes the communications for a given Port
  by setting the appropriate bits in each of the seven control registers for
  that Port. Details on the values sent to the command and data registers can
  be found in the VME8300 Quad Serial Board Users Manual.
*****************************************************************************/
```

```
static void SetDataTransfer(LONG Port,
                            LONG HandShaking,
                            LONG TxInterrupt,
                            LONG RxInterrupt)
{
  BYTE* CmdRegister = (BYTE*)(Port + 2);

  *CmdRegister = CONTROLREGISTER_0;
  *CmdRegister = 0x18;

  *CmdRegister = CONTROLREGISTER_2;
  *CmdRegister = 0x00;

  *CmdRegister = CONTROLREGISTER_4;
  *CmdRegister = 0x44;

  *CmdRegister = CONTROLREGISTER_1;
  *CmdRegister = 0x04 | TxInterrupt | RxInterrupt;

  *CmdRegister = CONTROLREGISTER_3;
  *CmdRegister = 0xc1 | HandShaking;

  *CmdRegister = CONTROLREGISTER_5;
  *CmdRegister = 0xe8;

  *CmdRegister = CONTROLREGISTER_6;
  *CmdRegister = 0x00;

  *CmdRegister = CONTROLREGISTER_7;
  *CmdRegister = 0x00;
}



/*****************************************************************************
  Function PutB() displays the character in Data on the console. It does
  this by polling the command/status register to ensure that the transmitter
  buffer is empty and ready to receive the character. The delays (the for loops)
  are required to keep the data from overwhelming the port and becoming garbage.
NOTE:
  The function required the 'for' loop as a timing delay. Without this delay,
  some of the characters sent to the console were lost. The reason behind this
  data loss was not determined.
*****************************************************************************/

static void PutB(char Data, int MemoryAddress){
  BYTE* OutputData = (BYTE*)MemoryAddress;
  BYTE* OutputCmd = (BYTE*)(MemoryAddress + 2);
  const int DelayTime = 0x300;
  int Wait;
```

```
    *OutputCmd = 0;
    for(Wait=0; Wait < DelayTime; Wait++)
      ;
    while((*OutputCmd & 0x04) == 0)
      ;

    *OutputData = Data;

    if(MemoryAddress == CONSOLE && Data == '\n'){
      for(Wait=0; Wait < DelayTime; Wait++)
        ;
      while((*OutputCmd & 0x04) == 0)
        ;

      *OutputData = '\r';
    }
}



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

    PUBLIC SECTION

    The following section defines the functions that provide access to the
    serial sub-system.

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

/*****************************************************************************
  Function ResetSerialBoards() resets both VME8300 Quad Serial Port Boards.
*****************************************************************************/

void ResetSerialBoards(){
    BYTE* Timer;
    int Wait;

    Timer = (BYTE*)(TIMERADDRESS_0 + 2);
    *Timer = 0xff;

    Timer = (BYTE*)(TIMERADDRESS_1 + 2);
    *Timer = 0xff;

    for(Wait=0; Wait<0x100; Wait++)
      ;
}
```

```
/*****************************************************************************
  Function InitializeConsole() prepares the port that connects the onboard
  console with the robot for I/O. It also establishes the mechanism for
  interrupt driven output.
*****************************************************************************/

void InitializeConsole(){
  InitializePort(CONSOLE,9600,0,1,0);
  SetConsoleIntMechanism();
}



/*****************************************************************************
  Function InitializeHost() sets up the port that connects the Unix work-
  station with the robot to be used for object code and data transfers.
*****************************************************************************/

void InitializeHost(){
  InitializePort(HOST,19200,0,0,0);
}



/*****************************************************************************
  Function InitializePort() establishes the communications of the given Port
  to the parameters given. Baud can be assigned standard communication rates
  300, 1200, 2400, 4800, 9600, or 19200. HandShaking, TxInterrupt, and
  RxInterrupt are used as ON/OFF flags. An assigned value of zero will disable
  these options, while any non-zero value will enable them.
*****************************************************************************/

void InitializePort(LONG Port,
                    LONG Baud,
                    LONG HandShaking,
                    LONG TxInterrupt,
                    LONG RxInterrupt)
{
  LONG taddr;
  LONG cnum;
  LONG lcount;
  LONG hcount;

  switch (Port){
    case CONSOLE:
      taddr = TIMERADDRESS_0;
      cnum = 1;
      break;
```

```
    case HOST:
      taddr = TIMERADDRESS_0;
      cnum = 2;
      break;
    case YSB1:
      taddr = TIMERADDRESS_1;
      cnum = 3;
      break;
    case YSB2:
      taddr = TIMERADDRESS_1;
      cnum = 4;
      break;
    default:
      PutStr("i_Port: illegal port");
      rexit();
    }

switch (Baud){
    case 300:
      lcount = 0x01a0;
      hcount = 0x01a1;
      break;
    case 1200:
      lcount = 0x0068;
      hcount = 0x0068;
      break;
    case 2400:
      lcount = 0x0034;
      hcount = 0x0034;
      break;
    case 4800:
      lcount = 0x001a;
      hcount = 0x001a;
      break;
    case 9600:
      lcount = 0x000d;
      hcount = 0x000d;
      break;
    case 19200:
      lcount = 0x0006;
      hcount = 0x0007;
      break;
    default:
      PutStr("i_port: bad baud rate");
      rexit();
}

SetTimer(taddr, cnum, 0x0b62, lcount, hcount);

SetDataTransfer(Port,
            (HandShaking ? 0x20 : 0),
```

```
            (TxInterrupt ? 0x02 : 0),
            (RxInterrupt ? 0x10 : 0));
}


/*****************************************************************************
  Function SetTimer() initializes one of the five counters located on the
  serial board (with the corresponding TimerAddress) by assigning values to
  command and data registers for that timer. Details on the values sent to the
  command and data registers can be found in the VME8300 Quad Serial Board
  Users Manual.
*****************************************************************************/

void SetTimer(LONG TimerAddress,
          LONG CounterNumber,
          LONG ModeCmd,
          LONG LoadCmd,
          LONG HoldCmd)
{
  BYTE* CmdRegister = (BYTE*)(TimerAddress + 2);
  BYTE* DataRegister = (BYTE*)(TimerAddress);
  BYTE CounterSelectBit;
  BYTE TimerCommand;

  CounterSelectBit = 1;
  CounterSelectBit = CounterSelectBit << (BYTE)(CounterNumber - 1);

  TimerCommand = DISARM;
  TimerCommand = TimerCommand | CounterSelectBit;
  *CmdRegister = TimerCommand;

  TimerCommand = LOAD;
  TimerCommand = TimerCommand | CounterSelectBit;
  *CmdRegister = TimerCommand;

  *CmdRegister = (BYTE)CounterNumber;

  *DataRegister = (BYTE)ModeCmd;
  *DataRegister = (BYTE)(ModeCmd >> 8);
  *DataRegister = (BYTE)LoadCmd;
  *DataRegister = (BYTE)(LoadCmd >> 8);
  *DataRegister = (BYTE)HoldCmd;
  *DataRegister = (BYTE)(HoldCmd >> 8);

  TimerCommand = ARM;
  TimerCommand = TimerCommand | CounterSelectBit;
  *CmdRegister = TimerCommand;
}
```

```
/*****************************************************************************
  Function SetLatch() passes to the specified latch the index value of the
  vector table entry  containing the interrupt handlers address to the serial
  board. When an interrupt associated with the given latch is generated, the
  VectorNumber is placed onto the address bus, indicating to the cpu the vector
  table entry that contains the address of the interrupt handler.
*****************************************************************************/

void SetLatch(LONG LatchAddress, BYTE VectorNumber){
  *(BYTE*)LatchAddress = VectorNumber;
}



/*****************************************************************************
  Function PutConsole() initiates the printing of the single character Source
  to the console.
*****************************************************************************/

void PutConsole(char Data){
  PutB(Data,CONSOLE);
}



/*****************************************************************************
  Function PutConsole() initiates the printing of the single character Source
  to the host system.
*****************************************************************************/

void PutHost(char Data){
  PutB(Data,HOST);
}



/*****************************************************************************
  Function GetConsole() polls the command/status register, waiting until a
  character from the console is placed in the receive buffer. It then returns
  that character to the calling function.
*****************************************************************************/

char GetConsole(){
  BYTE* InputAddress = (BYTE*)CONSOLE;
  BYTE* InputCmd = (BYTE*)(CONSOLE + 2);

  *InputCmd = 0;
```

```
  while((*InputCmd & 0x01) == 0)
    ;
  return *(BYTE*)InputAddress;
}



/*****************************************************************************
  Function TerminateConsoleOutput() signals the console port that the interrupt
  generated by the port has been serviced.
*****************************************************************************/

void TerminateConsoleOutput(){
  *(BYTE*)(CONSOLE + 2) = 0x28;
}



/*****************************************************************************
  Function TerminateConsoleIntCycle() tells the console port to terminate all
  unserviced interrupts.
*****************************************************************************/

void TerminateConsoleIntCycle(){
  *(BYTE*)(CONSOLE + 2) = 0x38;
}
```

APPENDIX B

A. IOSYS.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 2, 1994
  File Name:  iosys.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the prototypes/interface for the functions
          available in the I/O System module.
*****************************************************************************/

#ifndef __IOSYS_H

#define __IOSYS_H

#include "definitions.h"


/*****************************************************************************
  Function IoSysInitialize() initializes the ports used for serial communications.
*****************************************************************************/
void IoSysInitialize();


/*****************************************************************************
  Function IoSysControl() is the work-horse routine for the output interrupt
  handler.
*****************************************************************************/
void IoSysControl();


/*****************************************************************************
  Function PutStr() initiates the printing of the characters in the Source
  string to the console.
*****************************************************************************/
void PutStr(char* Source);


/*****************************************************************************
  Function PutInt sends the ascii representation of the parameter Number to
  the console.
*****************************************************************************/
void PutInt(int Number);
```

```
/*****************************************************************************
  Function PutReal sends the ascii representation of the parameter Number to
  the console. The output is in exponential notation with a total of Places
  digits after the decimal (up to a maximum of the constant MAX_REAL_PRECISION
  found in the source file).
*****************************************************************************/
void PutReal(double Number, int Places);


/*****************************************************************************
  Function GetStr() copies the string of characters typed at the console (up to
  Length characters) into the string pointed at by Source.
*****************************************************************************/
void GetStr(char* Source, int Length);


/*****************************************************************************
  Function GetInt() reads a string from the console (up to MAX_INTEGER_DIGITS)
  and converts the ascii representation into its integer value. All leading
  white space (spaces and tabs) are discarded. The conversion stops at the
  first character that can not be part of a legel integer value. All remaining
  characters in the string are discarded. Strings where the first non white-
  space character is not a digit will default to 0. Empty strings also default
  to 0.
*****************************************************************************/
int GetInt();


/*****************************************************************************
  Function GetReal() reads a string from the console (up to MAX_REAL_PRECISION)
  and converts the ascii representation into its double value. All leading
  white space (spaces and tabs) are discarded. The conversion stops at the
  first character that can not be part of a legel double value. All remaining
  characters in the string are discarded. Strings where the first non white-
  space character is not a digit will default to 0.0. Empty strings also
  default to 0.
*****************************************************************************/
double GetReal();


#endif
```

B. IOSYS.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
```

```
   Revised:    March 2, 1994
   File Name:  iosys.c
   Environment: GCC ANSI C compiler for the motorola 68020 processor
   Description: This file contains the routines and data structures needed to
                provide I/O capabilities to the rest of the subsystems in the
                Yamabico project. This file contains two marked sections. The
                first is the private section containing the encapsulated data
                and functions. The second section is the public section. This
                section defines the interface routines.
*****************************************************************************/

#include "definitions.h"
#include "convertutil.h"
#include "serial.h"
#include "iosys.h"



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

   PRIVATE SECTION

   The following section defines the encapsulated definitions and data
   structures used in the system.

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

#define BUFSIZE 1024

/*****************************************************************************
   Structure and declaration of the output buffer. It is declared static to
   prevent access from routines external to this module.
*****************************************************************************/
typedef struct{
   int Head;
   int Tail;
   int Count;
   char Buffer[BUFSIZE];
} IOBUFFER;


static IOBUFFER OutputBuf;



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

   PUBLIC SECTION
```

```
  The following section defines the functions that provide access to the
  terminal system.
*****************************************************************************/

/*****************************************************************************
  Function IoSysInitialize() initializes the ports used for serial communications.
  The CONSOLE port is set for interrupt driven output, but direct memory (polled)
  input. The output buffer is also initialized.
*****************************************************************************/

void IoSysInitialize(){
  OutputBuf.Head = 0;
  OutputBuf.Tail = 0;
  OutputBuf.Count = 0;
}



/*****************************************************************************
  Function IoSysControl() controls the printing of characters in the output
  buffer to the screen. It is called from the interrupt handling routine. It
  operates by printing one character to the screen and then terminating the
  interrupt. The act of printing a character to the screen generates another
  interrupt to print the next character in the buffer. If the buffer is empty,
  it sends a command to the port to stop the interrupt chain.
*****************************************************************************/

void IoSysControl(){

  if(OutputBuf.Count > 0){
    PutConsole(OutputBuf.Buffer[OutputBuf.Tail]);
    OutputBuf.Count--;
    if(OutputBuf.Tail == BUFSIZE-1){
      OutputBuf.Tail = 0;
    }else
      OutputBuf.Tail++;
  }else
    TerminateConsoleOutput();
  TerminateConsoleIntCycle();
}



/*****************************************************************************
  Function PutStr copies the string pointed to by Source into the output buffer.
  A critical region exists where the number of characters in the output buffer
  is incremented. Therefore, the priority mask is set to prevent any interrupts
```

```
  from taking control of the CPU during its execution.
*****************************************************************************/

void PutStr(char* Source){
  while (*Source != '\0'){
    OutputBuf.Count++;

    OutputBuf.Buffer[OutputBuf.Head] = *Source;
    Source++;
    if(OutputBuf.Head == BUFSIZE-1){
      OutputBuf.Head = 0;
    }else
      OutputBuf.Head++;
  }

  /* Initiate an output interrupt by sending a null character to the console. */
  PutConsole('\0');
}



/*****************************************************************************
  Function PutInt converts the parameter Number to its ascii representation and
  sends the resulting string to the console by calling to PutStr.
*****************************************************************************/

void PutInt(int Number){
   char NumStr[20];

   ItoA(Number,NumStr);
   PutStr(NumStr);
}



/*****************************************************************************
  Function PutReal converts the exponential notation of the parameter Number
  into its ascii representation. It then sends the resulting string to the
  console by calling PutStr.
*****************************************************************************/

void PutReal(double Number, int Places){
   char NumStr[MAX_REAL_PRECISION+7];

   if(Places<MAX_REAL_PRECISION){
     RtoAE(Number,NumStr,Places+1);
   }else{
     RtoAE(Number,NumStr,MAX_REAL_PRECISION+1);
   }
```

```
    PutStr(NumStr);
}



/*****************************************************************************
  Function GetStr() copies the string of characters typed at the console (up to
  Length characters) into the string pointed at by Source.
*****************************************************************************/

void GetStr(char* Source, int Length){
  BYTE KeyStroke;
  int I = -1;
  int Size = Length - 1;

  do{
    KeyStroke = GetConsole();

    switch(KeyStroke){
      case '\b':
        if(I >= 0){
          I--;
          PutConsole(KeyStroke);
        }
        break;
      default:
        Source[++I] = KeyStroke;
        PutConsole(KeyStroke);
    }
  }while((Source[I] != '\r') && (I < Size));

  Source[I] = '\0';
}



/*****************************************************************************
  Function GetInt() reads a string from the console (up to MAX_INTEGER_DIGITS)
  and converts the ascii representation into its integer value. All leading
  white space (spaces and tabs) are discarded. The conversion stops at the
  first character that can not be part of a legel integer value. All remaining
  characters in the string are discarded. Strings where the first non white-
  space character is not a digit will default to 0. Empty strings also default
  to 0.
*****************************************************************************/

int GetInt(){
  char NumStr[MAX_INTEGER_DIGITS+1];
  char* Temp = NumStr;
```

```
  GetStr(NumStr,MAX_INTEGER_DIGITS+1);

  while((*Temp=='\t') || (*Temp==' '))
    Temp++;

  retum Atol(Temp);
}



/*****************************************************************************
  Function GetReal() reads a string from the console (up to MAX_REAL_PRECISION)
  and converts the ascii representation into its double value. All leading
  white space (spaces and tabs) are discarded. The conversion stops at the
  first character that can not be part of a legel double value. All remaining
  characters in the string are discarded. Strings where the first non white-
  space character is not a digit will default to 0.0. Empty strings also
  default to 0.
*****************************************************************************/

double GetReal(){
  char NumStr[MAX_REAL_PRECISION+1];
  char* Temp = NumStr;

  GetStr(NumStr,MAX_REAL_PRECISION+1);

  while((*Temp=='\t') || (*Temp==' '))
    Temp++;

  retum AtoR(Temp);
}
```

APPENDIX C

A. MOTION.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 16, 1993
  Revised:    March 2, 1994
  File Name:  motion.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the prototypes/interface for the functions
               available in the Motion System module. The first two routines
               are for system setup and control. The rest are MML's immediate
               commands defined by the language.
*****************************************************************************/

#ifndef __MOTION_H

#define __MOTION_H

#include "definitions.h"


/*****************************************************************************
  Function MotionSysInitialize() initializes the motion subsystem by assigning
  default values to the local variables and establishing the interrupt handling
  mechanism.
*****************************************************************************/
void MotionSysInitialize();


/*****************************************************************************
  Function MotionSysControl() is the interrupt handler workhorse and is called
  from the assembly interrupt handler shell.
*****************************************************************************/
void MotionSysControl();


/*****************************************************************************
  The following declarations are prototypes for MML's immediate commands.
  The command descriptions can be found in the MML Bible.
*****************************************************************************/
void SetRobotConfiguration(CONFIGURATION NewConfiguration);

void GetRobotConfiguration(CONFIGURATION* CurrentConfiguration);

void Stop();
```

```
void SetLinearVelocity(double LinearVelocity);

void SetRotationalVelocity(double RotationalVelocity);

void SetLinearAcceleration(double LinearAcceleration);

void SetRotationalAcceleration(double RotationalAcceleration);

void SetSizeConstant(double SizeConstant);

double GetTotalDistance();

void SkipPathElement();

void HaltMotion();

void ResumeMotion();

CONFIGURATION SetInitialPosition();

void ReportRobotConfiguration(CONFIGURATION CurrentConfiguration);

void MotionOn();

void MotionOff();

#endif
```

B. MOTION.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 16, 1993
  Revised:    March 2, 1994
  File Name:  motion.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file provides the routines and data structures needed to
          provide the motion control capability for the robot. The file
          is divided into three sections. The first is the private section
          containing the encapsulated data and functions. The second
          section is the control section. This section defines the
          routines required for motion control. The third section is the
          Immediate command section. This section defines MML's immediate
          commands. The routines in these last two sections can be
          accessed publicly.
*****************************************************************************/

#include "definitions.h"
```

```
#include "system.h"
#include "iosys.h"
#include "wheels.h"
#include "math.h"
#include "math68881.h"
#include "motiontrace.h"
#include "motion.h"



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

  PRIVATE SECTION

  The following section defines the encapsulated definitions, data structures
  and prototypes used in the system.

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

#define SMALLERROR 0.0001

static int LoopTest;

static BOOLEAN Halted;

static VELOCITY HaltedVelocity;
static VELOCITY DesiredVelocity;
static VELOCITY Commanded;
static VELOCITY DesiredAcceleration;

static double TotalDistance;

static double DesiredSizeConstant;

static CONFIGURATION VehicleConfiguration;


/*****************************************************************************
  The following static function declarations are the prototypes for the
  encapsulated functions.
*****************************************************************************/
static void UpdateConfiguration(double DeltaDistanceChanged,
                                double DeltaOrientation);
static VELOCITY GetCommandedVelocity(VELOCITY Desired,
                                     VELOCITY Actual,
                                     VELOCITY Commanded);
static double GetLinearVelocity(double DesiredVelocity,
                                double ActualVelocity,
                                double LastCommandedVelocity);
```

```
/*****************************************************************************
  Function UpdateConfiguration() calculates the robots current position based
  on DeltaDistanceChanged (ds) and DeltaOrientation (dt).
*****************************************************************************/

static void UpdateConfiguration(double DeltaDistanceChanged,
                                double DeltaOrientation){

  double DistanceIncrement = DeltaDistanceChanged;
  double OrientationIncrement = DeltaOrientation / 2;

  DisableInterrupts();

  if(fabs(DeltaOrientation) > SMALLERROR)
    DistanceIncrement *= sin(OrientationIncrement) / OrientationIncrement;

  VehicleConfiguration.Position.XPosition += DistanceIncrement *
                cos(VehicleConfiguration.Orientation + OrientationIncrement);
  VehicleConfiguration.Position.YPosition += DistanceIncrement *
                sin(VehicleConfiguration.Orientation + OrientationIncrement);
  VehicleConfiguration.Orientation += DeltaOrientation;
  if(fabs(DeltaDistanceChanged) > SMALLERROR){
    VehicleConfiguration.Kappa = DeltaOrientation / DeltaDistanceChanged;
  }else{
    VehicleConfiguration.Kappa = DeltaOrientation / SMALLERROR;
  }

  EnableInterrupts();
}



/*****************************************************************************
  Function GetCommandedVelocity() calculates the commanded velocity based on
  the current velocity, the desired velocity, and the previous commanded
  velocity.
*****************************************************************************/

static VELOCITY GetCommandedVelocity(VELOCITY Desired,
                                     VELOCITY Actual,
                                     VELOCITY Commanded){
  Commanded.Linear =
    GetLinearVelocity(Desired.Linear,Actual.Linear,Commanded.Linear);

/* This statement is used since GetRotationalVelocity() is not */
/* currently defined. Otherwise, a statement similar to above  */
/* would be used.                                              */
```

```
Commanded.Rotational = Desired.Rotational;

  return Commanded;
}



/*****************************************************************************
  Function GetLinearVelocity() calculates the linear component of the commanded
  velocity.
*****************************************************************************/

static double GetLinearVelocity(double DesiredVelocity,
                                double ActualVelocity,
                                double CommandedVelocity){
  double VelocityChange;

  VelocityChange = DesiredAcceleration.Linear * MOTION_CONTROL_CYCLE;

  if(ActualVelocity < DesiredVelocity){
    CommandedVelocity = Min(CommandedVelocity + VelocityChange, DesiredVelocity);
  }else{
    CommandedVelocity = Max(CommandedVelocity - VelocityChange, DesiredVelocity);
  }

  return CommandedVelocity;
}



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

  MOTION CONTROL SECTION

  The following section defines the functions that provide access to the
  motion control system. These routines are public.

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

/*****************************************************************************
  Function MotionSysInitialize() initializes all of the private global variables
  in this module to the desired default values. It then calls SetTimer to
  program the 5th timer on serial board #1 (the second serial board) to generate
  synchronous interrupts every 10ms. After the timer has been set up, the
  interrupt handling routine is made available to the system by the call to
  SetMotionInterruptHandler().
*****************************************************************************/

void MotionSysInitialize(){
```

```
  LoopTest = 0;

  Halted = FALSE;

  TotalDistance = 0.0;

  DesiredVelocity.Linear = 0.0;
  DesiredVelocity.Rotational = 0.0;
  Commanded.Linear = 0.0;
  Commanded.Rotational = 0.0;
  DesiredAcceleration.Linear = 20.0;
  DesiredAcceleration.Rotational = 0.5;
  DesiredSizeConstant = 20.0;

  VehicleConfiguration.Position.XPosition = 0.0;
  VehicleConfiguration.Position.YPosition = 0.0;
  VehicleConfiguration.Orientation = 0.0;
  VehicleConfiguration.Kappa = 0.0;

  /* Initialize sub-systems. */
  InitializeWheels();
TraceMotionSysInitialize(400); /* This sub-system only required if data */
                        /* logging is desired.              */

  SetMotionIntMechanism();
}



/*****************************************************************************
  Function MotionSysControl() is the interrupt handler workhorse. It is called
  from the assembly interrupt handler shell. Its first task is to update the
  change in position and orientation through calls to the module responsible for
  movement. It then uses this information in the motion control laws to derive
  the commanded linear and rotational velocities required for this motion
  control cycle. Finally, it passes these computed velocities back to the move-
  ment module for execution.
*****************************************************************************/

void MotionSysControl(){
  double OrientationChange;
  double DistanceChanged;
  VELOCITY Actual;

  UpdateMovement();
  DistanceChanged = GetDistanceTraveled();
  OrientationChange = GetOrientationChange();

  TotalDistance += DistanceChanged;
```

```
  UpdateConfiguration(DistanceChanged,OrientationChange);

  Actual.Linear = DistanceChanged / MOTION_CONTROL_CYCLE;
  Actual.Rotational = OrientationChange / MOTION_CONTROL_CYCLE;

/* The logging statement can be moved, modified or deleted as desired. */
LogTimedMotion(Actual.Linear);

  Commanded = GetCommandedVelocity(DesiredVelocity,Actual,Commanded);

  SetMovement(Commanded.Linear,Commanded.Rotational);

/* LoopTest used to control output from interrupt driven motion control */
/* system. LoopTest is assigned zero every 100 cycles (1 sec).        */
if(LoopTest++ >= 99)
  LoopTest = 0;
}



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

  IMMEDIATE COMMAND SECTION

  The following section defines the functions that provide access to MML's
  immediate commands. The functionality of these routines can be found in the
  language definition. These routines are also public.

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

void SetRobotConfiguration(CONFIGURATION NewConfiguration){
  DisableInterrupts();

  VehicleConfiguration.Position.XPosition = NewConfiguration.Position.XPosition;
  VehicleConfiguration.Position.YPosition = NewConfiguration.Position.YPosition;
  VehicleConfiguration.Orientation = NewConfiguration.Orientation;
  VehicleConfiguration.Kappa = NewConfiguration.Kappa;

  EnableInterrupts();
}



void GetRobotConfiguration(CONFIGURATION* CurrentConfiguration){
  DisableInterrupts();
  *CurrentConfiguration = VehicleConfiguration;
  EnableInterrupts();
}
```

```
void Stop(){
  WheelsDisable();
  DesiredVelocity.Linear = 0.0;
  DesiredVelocity.Rotational = 0.0;
}



void SetLinearVelocity(double LinearVelocity){
  DesiredVelocity.Linear = LinearVelocity;
}



void SetRotationalVelocity(double RotationalVelocity){
  DesiredVelocity.Rotational = RotationalVelocity;
}



void SetLinearAcceleration(double LinearAcceleration){
  DesiredAcceleration.Linear = LinearAcceleration;
}



void SetRotationalAcceleration(double RotationalAcceleration){
  DesiredAcceleration.Rotational = RotationalAcceleration;
}



void SetSizeConstant(double SizeConstant){
  DesiredSizeConstant = SizeConstant;
}



double GetTotalDistance(){
  return TotalDistance;
}
```

```
void SkipPathElement(){
  PutStr("\nInside skip() stub");
}



void HaltMotion(){
  if(!Halted){
    Halted = TRUE;
    HaltedVelocity.Linear = DesiredVelocity.Linear;
    HaltedVelocity.Rotational = DesiredVelocity.Rotational;
    WheelsDisable();
  }
}



void ResumeMotion(){
  if(Halted){
    Halted = FALSE;
    DesiredVelocity.Linear = HaltedVelocity.Linear;
    DesiredVelocity.Rotational = HaltedVelocity.Rotational;
    WheelsEnable();
  }
}



CONFIGURATION SetInitialPosition(){
  CONFIGURATION NewConfiguration;

  PutStr("\nEnter the Starting X Position: ");
  NewConfiguration.Position.XPosition = GetReal();
  PutStr("Enter the Starting Y Position: ");
  NewConfiguration.Position.YPosition = GetReal();
  PutStr("Enter the Starting Orientation: ");
  NewConfiguration.Orientation = GetReal();
  PutStr("Enter the Starting Kappa Value: ");
  NewConfiguration.Kappa = GetReal();

  return NewConfiguration;
}
```

```
void ReportRobotConfiguration(CONFIGURATION CurrentConfiguration){
  PutStr("\nCurrent Robot Configuration:\n\tX =>\t\t");
  PutReal(CurrentConfiguration.Position.XPosition,4);
  PutStr("\n\tY =>\t\t");
  PutReal(CurrentConfiguration.Position.YPosition,4);
  PutStr("\n\tTheta =>\t");
  PutReal(CurrentConfiguration.Orientation,4);
  PutStr("\n\tKappa =>\t");
  PutReal(CurrentConfiguration.Kappa,4);
}



void MotionOn(){
  WheelsEnable();
}



void MotionOff(){
  WheelsDisable();
}
```

C. MOTIONTRACE.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       January 20, 1994
  Revised:    March 3, 1994
  File Name:  motiontrace.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the prototypes/interface for the functions
          available to log motion control data.
*****************************************************************************/

#ifndef __MOTIONTRACE_H
#define __MOTIONTRACE_H


/*****************************************************************************
  Function TraceMotionSysInitialize() prepares the tracing system to log data.
  It requests a block of dynamic memory to store the number of data points
  requested by NumberOfPoints. No error checking for dynamic memory allocation
  is performed by this function.
```

```
*****************************************************************************/
void TraceMotionSysInitialize(int NumberOfPoints);

/*****************************************************************************
  Function MotionTraceEnable() enables data logging. Frequency is used by the
  system's logging functions to determine the number of motion control cycles
  between logged data. For example, if the frequency is 3, then data would be
  logged on every third call to LogTimedMotion() or LogMotionData().
*****************************************************************************/
void MotionTraceEnable(int Frequency);

/*****************************************************************************
  Function MotionTraceDisable() disables data logging.
*****************************************************************************/
void MotionTraceDisable();

/*****************************************************************************
  Function LogTimedMotion() logs Data against time in seconds, starting when
  data logging is turned on. This is based the assumption that the routine is
  called every motion control cycle. The Frequency value given when logging is
  enabled determines the number of number of LogTimedMotion() calls required
  between recorded data.
*****************************************************************************/
void LogTimedMotion(double Data);

/*****************************************************************************
  Function LogMotionData() logs both parameters after every 'Frequency' calls.
*****************************************************************************/
void LogMotionData(double XPlot, double YPlot);

/*****************************************************************************
  Function DownLoadMotionData() prompts the user for a file name on the host
  system to store the data. The file then opened, deleting any previous
  contents. The routine then transfers character string representations of the
  recorded data to the Unix host.
*****************************************************************************/
void DownLoadMotionData();

#endif
```

D. MOTIONTRACE.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       January 20, 1994
  Revised:    March 3, 1994
  File Name:  motiontrace.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the routines required to for the tracing
               system. The file is seperated into two section. The first is
               the private section containing the encapsulated data and
               functions. The second section is the public section. This
               section defines the interface routines.
*****************************************************************************/

#include "definitions.h"
#include "memsys.h"
#include "convertutil.h"
#include "serial.h"
#include "iosys.h"
#include "motiontrace.h"



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

  PRIVATE SECTION

  The following section defines the encapsulated definitions, data structures
  and prototypes used in the system.

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

typedef struct{
  double XData;
  double YData;
} SAMPLINGPOINT;

static SAMPLINGPOINT *MotionData;

static int MaxDataPoints;
static int NextPlot;
static int LoggingData;
static int CycleCounter;
static int LoggingFrequency;


/*****************************************************************************
  The following static function declarations are the prototypes for the
```

```
  encapsulated functions.
*****************************************************************************/
static void PutStrHost(char* Source);



/*****************************************************************************
  Function PutStrHost() is a simple routine that writes a character string to
  the host environment. Writing non-character information to the host
  environment may produce unexpected results. Other routines may be created to
  write non-character data by converting the data to character strings prior
  to being sent.
*****************************************************************************/

static void PutStrHost(char* Source){
  while (*Source != '\0'){

    PutHost(*Source);
    Source++;
  }
}



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

  PUBLIC SECTION

  The following section defines the functions that provide access to the
  tracing sub-system.

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

/*****************************************************************************
  Function TraceMotionSysInitialize() prepares the tracing system to log data.
  It requests a block of dynamic memory to store the number of data points
  requested by NumberOfPoints. No error checking for dynamic memory allocation
  is performed by this function.
*****************************************************************************/

void TraceMotionSysInitialize(int NumberOfPoints){
  NextPlot = 0;
  LoggingData = FALSE;
  CycleCounter = 0;
  LoggingFrequency = 1;
  if(NumberOfPoints > 0)
    MaxDataPoints = NumberOfPoints;
  else
    MaxDataPoints = 0;
```

```
  free((BYTE*)MotionData);
  MotionData = (SAMPLINGPOINT*)malloc(sizeof(SAMPLINGPOINT)*NumberOfPoints);
}



/*****************************************************************************
  Function MotionTraceEnable() enables data logging. Frequency is used by the
  system's logging functions to determine the number of motion control cycles
  between logged data. For example, if the frequency is 3, then data would be
  logged on every third call to LogTimedMotion() or LogMotionData().
*****************************************************************************/

void MotionTraceEnable(int Frequency){
  if(Frequency > 0)
    LoggingFrequency = Frequency;
  else
    LoggingFrequency = 1;

  LoggingData = TRUE;
}



/*****************************************************************************
  Function MotionTraceDisable() disables data logging.
*****************************************************************************/

void MotionTraceDisable(){
  LoggingData = FALSE;
}



/*****************************************************************************
  Function LogTimedMotion() logs Data against time in seconds, starting when
  data logging is turned on. This is based the assumption that the routine is
  called every motion control cycle. The routine uses the variable CycleCounter
  as a counter. When the value of CycleCounter reaches zero, data is logged and
  the counter is reset to the LoggingFrequency set when logging was enabled. If
  the value is greater than zero, the counter is decremented and logging is NOT
  performed.
*****************************************************************************/

void LogTimedMotion(double Data){
  if(LoggingData){
    if((CycleCounter-- <= 0) && (NextPlot < MaxDataPoints)){
```

```
        MotionData[NextPlot].XData = NextPlot * LoggingFrequency
                                * MOTION_CONTROL_CYCLE;
        MotionData[NextPlot].YData = Data;
        NextPlot++;
        CycleCounter = LoggingFrequency;
      }
    }
}



/*****************************************************************************
  Fucntion LogMotionData() records both parameters when the CycleCounter is
  zero. It then resets the counter to LoggingFrequency, assigned when logging
  is enabled. If the value is greater than zero, the counter is decremented and
  logging is NOT performed.
*****************************************************************************/

void LogMotionData(double XPlot, double YPlot){
  if(LoggingData){
    if((CycleCounter-- <= 0) && (NextPlot < MaxDataPoints)){
      MotionData[NextPlot].XData = XPlot;
      MotionData[NextPlot].YData = YPlot;
      NextPlot++;
      CycleCounter = LoggingFrequency;
    }
  }
}



/*****************************************************************************
  Function DownLoadMotionData() prompts the user for a file name on the host
  system to store the data. The file then opened, deleting any previous
  contents. The routine is dependent on the "ytof" call in the yamabico
  account. The routine then transfers character string representations of the
  recorded data to the Unix host. Afterwards, the function ensures that the
  host file is closed.
*****************************************************************************/

void DownLoadMotionData(){
  char FileName[35];
  char DataString[MAX_REAL_PRECISION];
  int LoopCounter = MaxDataPoints;

  MotionTraceDisable();

  PutStr("\n\n\aReady to down load motion data. Connect the");
  PutStr("\ncable and press any key to continue.");
```

```
    GetConsole();

    PutStr("\a\n\nEnter the name of the Host file used to store the Motion Data");
    PutStr("\n(WARNING: PREVIOUS FILE CONTENTS WILL BE DESTROYED)");
    PutStr("\n\n\tFile: ");
    GetStr(FileName,35);

    PutStrHost("ytof ");
    PutStrHost(FileName);
    PutStrHost(" w \n");

    PutStr("\n\n\aReady to down load data to Host computer. ");
    PutStr("\nPress any key to begin ");
    GetConsole();

    PutStr("\n\nDown loading data...");

    for(LoopCounter=0; LoopCounter<MaxDataPoints; LoopCounter++){
      PutStrHost(RtoAE(MotionData[LoopCounter].XData,DataString,5));
      PutStrHost("\t");
      PutStrHost(RtoAE(MotionData[LoopCounter].YData,DataString,5));
      PutStrHost("\n");
    }

    PutHost('\4');
    PutHost('\4');

    PutStr("\aDown load complete");
}
```

E. WHEELS.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 16, 1993
  Revised:    March 2, 1994
  File Name:  wheels.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains the prototypes to available wheel sub-system
          routines. This set of prototypes defines the wheel sub-system
          interface.
*****************************************************************************/

#ifndef __WHEELS_H

#define __WHEELS_H

#include "definitions.h"
```

```
#define MAXVELOCITY 60

/*****************************************************************************
  Function InitializeWheels() initializes the wheels subsystem to its default
  settings.
*****************************************************************************/
void InitializeWheels();


/*****************************************************************************
  Function WheelsEnable() turns on the motors connected to the wheels.
*****************************************************************************/
void WheelsEnable();


/*****************************************************************************
  Function WheelsDisable() turns off the motors connected to the wheels.
*****************************************************************************/
void WheelsDisable();


/*****************************************************************************
  Function UpdateMovement() updates the distance traveled by both wheels.
*****************************************************************************/
void UpdateMovement();


/*****************************************************************************
  Function GetDistanceTraveled() returns the linear distance the robot has
  traveled between the last two calls to UpdateMovement().
*****************************************************************************/
double GetDistanceTraveled();


/*****************************************************************************
  Function GetOrientationChange() returns the difference between the changes
  in the distance of the left and right wheels between the last two calls to
  UpdateMovement().
*****************************************************************************/
double GetOrientationChange();


/*****************************************************************************
  Function SetMovement() translates the commanded linear and rotational vel-
  ocities into commanded velocities for each wheel.
*****************************************************************************/
void SetMovement(double LinearVelocity, double RotationalVelocity);


#endif
```

F. WHEELS.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 17, 1993
  Revised:    March 2, 1994
  File Name:  wheels.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file defines the wheel sub-system. It contains the hardware
        interface routines required for operation with the Dual Axis
        Controller. The file is seperated into two section. The first
        is the private section containing the encapsulated data and
        functions. The second section is the public section. This
        section defines the interface routines.
*****************************************************************************/

#include "definitions.h"
#include "wheels.h"



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

  PRIVATE SECTION

  The following section defines the encapsulated definitions, data structures
  and prototypes used in the system.

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

#define RIGHT_DRIVE_PWM 0xfffff000
#define RIGHT_BRAKE_PWM 0xfffff002
#define LEFT_DRIVE_PWM 0xfffff004
#define LEFT_BRAKE_PWM 0xfffff006

#define MOTIONCONTROLADDRESS 0xfffff008

#define RIGHT_ENCODER_HIGH_WORD 0xfffff010
#define RIGHT_ENCODER_LOW_WORD 0xfffff012
#define LEFT_ENCODER_HIGH_WORD 0xfffff014
#define LEFT_ENCODER_LOW_WORD 0xfffff016

#define TREAD 52.41246478129945756620
#define TREAD_R 53.40092  /* width of the robot (cm) when rotating. */

#define ENC_TO_DIST 0.00105570833333333333
```

```
#define KPWB 10.0

/* Assumed PWM range: 0..127. Set at 90 to allow testing on weak AC generator. */
/* Can be changed to 127 if generator not used or upgraded.                    */
#define PWMLIMIT 90

static int MotorOn;
static BOOLEAN Rotating;

static int RightEncoderValue;
static int LeftEncoderValue;
static int DeltaRightEncoderValue;
static int DeltaLeftEncoderValue;

static double Rpwm;
static double Lpwm;
static WORD MotionControlWord;


/*****************************************************************************
  The following static function declarations are the prototypes for the
  encapsulated functions.
*****************************************************************************/
static int GetWheelEncoder(LONG HighWordAddress, LONG LowWordAddress);
static int EncoderDifference(int NewValue, int OldValue);
static int GetPwm(double CommandedVelocity, double ActualVelocity, double Pwm);
static double PwmLookUp(double Velocity);
static void SetMotorControl(int Lpwm, int Rpwm, int MotionControlWord);



/*****************************************************************************
  Function GetWheelEncoder() appends the contents of the shaft encoder low
  word register to the contents of the high word register, forming a long
  word. It then returns that value.
*****************************************************************************/

static int GetWheelEncoder(LONG HighWordAddress, LONG LowWordAddress){
  LONG Wheel;

  Wheel = *(WORD*)HighWordAddress;
  Wheel = Wheel << 16;
  Wheel += *(WORD*)LowWordAddress;

  return Wheel;
}
```

```
/*****************************************************************************
  Function EncoderDifference() returns the difference between the new shaft
  encoder position and the old shaft encoder position. The shaft encoder values
  contain only 24 bits (0x000000-0xffffff). The routine adjusts for the trans-
  ition from 0xffffff to 0x000000 and vice versa.
*****************************************************************************/

static int EncoderDifference(int NewValue, int OldValue){
  int Difference = NewValue - OldValue;

  if(Difference < -0x800000){
    Difference = Difference & 0x00ffffff;
  }else if(Difference >= 0x800000){
    Difference = Difference | 0xff000000;
  }

  return Difference;
}



/*****************************************************************************
  Function GetPwm() returns a new PWM value based on the desired velocity,
  the actual velocity and the old PWM value.
*****************************************************************************/

static int GetPwm(double CommandedVelocity, double ActualVelocity, double Pwm){
  double a = 0.7;
  int PwmTemp = PwmLookUp(CommandedVelocity) + KPWB *
          (CommandedVelocity - ActualVelocity);

  return (a * PwmTemp + (1.0 - a) * Pwm);
}



/*****************************************************************************
  Function SetMotorControl() sends the PWM values to the motor control board
  if the motors are turned on via a previous call to MotorEnable(). If the
  motors are not turned on, then a command to free the motors is sent.
*****************************************************************************/

static void SetMotorControl(int Lpwm, int Rpwm, int MotionControlWord){
  if(MotorOn){
    *(WORD*)LEFT_DRIVE_PWM = (WORD)Lpwm;
    *(WORD*)RIGHT_DRIVE_PWM = (WORD)Rpwm;
    *(WORD*)MOTIONCONTROLADDRESS = (WORD)MotionControlWord;
```

```
  }else{
    *(WORD*)MOTIONCONTROLADDRESS = 0x0303;
  }
}



/*****************************************************************************
FUNCTION:  pwm_lookup
PARAMETERS: vel (wheel velocity)
PURPOSE:   Determines the estimated pwm ratio given
        the desired wheel velocity as an input. (This table get from 7.9 KHZ
        motor output curve).
RETURNS:   pwm value based upon empirically determined velocity
        vs pwm ratio curve.
CALLED BY: control()
CALLS:     none
COMMENTS:  12 Jan 93 - Dave MacPherson,16 Sep 1993 changed by Ten-Min Lee
TASK: Level 4 interrupt
*****************************************************************************/

static double PwmLookUp(double Velocity){
  double v;
  double pwm_value;

  v = Velocity;
  if (v == 0.0 )
    pwm_value = 0.0;
  else if (v >= 0.0 && v < 25.0)
    pwm_value = (0.96 * v + 49.0);
  else if (v >= 25.0 && v < 53.0)
    pwm_value = (0.82 * (v - 25.0) + 73.0);
  else if (v >= 53.0 && v <= 65.0)
    pwm_value = (2.0 * (v - 53.0) + 96.0);
  else if (v > 65.0)
    pwm_value = 127.0;
  else if (v < 0.0 && v >= - 2.5)
    pwm_value = (1.2 * ( v ) - 54.0);
  else if (v < -2.5 && v >= -13.0)
    pwm_value = (0.76 * (v + 2.5) - 57.0);
  else if (v < -13.0 && v >= -20.0)
    pwm_value = (0.43 * (v + 13.0) - 65.0);
  else if (v < -20.0 && v >= -34.0)
    pwm_value = (1.0 * (v + 20.0) - 68.0);
  else if (v < -34.0 && v >= -41.0)
    pwm_value = (0.7 * (v + 34.0) - 82.0);
  else if (v < -41.0 && v >= -49.0)
    pwm_value = (1.5 * (v + 41.0) - 87.0);
  else if (v < -49.0 && v >= -62.0)
    pwm_value = (1.1 * (v + 49.0) - 99.0);
```

```
    else if (v < -62.0 && v >= -65.0)
      pwm_value = (2.3 * (v + 62.0) - 113.0);
    else
      pwm_value = -127.0;

    return pwm_value;
  } /* end pwm_lookup */



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

    PUBLIC SECTION

    The following section defines the functions that provide access to the
    wheel sub-system.

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

  /*****************************************************************************
    Function InitializeWheels() initializes all of the private global variables
    in this module to the desired default values.
  *****************************************************************************/

  void InitializeWheels(){
    WheelsDisable();
    Rotating = FALSE;

    Rpwm = 0.0;
    Lpwm = 0.0;
    MotionControlWord = 0;

    DeltaRightEncoderValue = 0;
    DeltaLeftEncoderValue = 0;
                                                    RightEncoderValue          =
GetWheelEncoder(RIGHT_ENCODER_HIGH_WORD,RIGHT_ENCODER_LOW_WORD);
                                                     LeftEncoderValue          =
GetWheelEncoder(LEFT_ENCODER_HIGH_WORD,LEFT_ENCODER_LOW_WORD);
  }



  /*****************************************************************************
    Function WheelsEnable() turns on the motors connected to the wheels.
    This is done by setting the MotorOn flag, used by SetMotorControl(), to
    a non-zero value.
  *****************************************************************************/

  void WheelsEnable(){
```

```
  MotorOn = 1;
}



/*****************************************************************************
  Function WheelsDisable() turns off the motors connected to the wheels.
  This is done by setting the MotorOn flag, used by SetMotorControl(), to
  zero.
*****************************************************************************/

void WheelsDisable(){
  MotorOn = 0;
}



/*****************************************************************************
  Function UpdateMovement() updates the distance traveled by both wheels. It
  does this by calculating the difference between the current shaft encoder
  values and the encoder values from the last time they were read. It also
  stores the current encoder values which will be used as the last encoder
  values when this routine is called again.
*****************************************************************************/

void UpdateMovement(){
  int Wheel;

                                                    Wheel                =
GetWheelEncoder(RIGHT_ENCODER_HIGH_WORD,RIGHT_ENCODER_LOW_WORD);
  DeltaRightEncoderValue = EncoderDifference(Wheel,RightEncoderValue);
  RightEncoderValue = Wheel;

  /* The left motor is the mirror image of the right. Therefore the */
  /* amount of change in the left encoder value needs to be negated */
  /* to show the proper direction of rotation.                      */
                                                    Wheel                =
GetWheelEncoder(LEFT_ENCODER_HIGH_WORD,LEFT_ENCODER_LOW_WORD);
  DeltaLeftEncoderValue = - EncoderDifference(Wheel,LeftEncoderValue);
  LeftEncoderValue = Wheel;
}



/*****************************************************************************
  Function GetDistanceTraveled() returns the linear distance the robot has
  traveled between the last two calls to UpdateMovement(). It makes the
  calculations based on the measured center point between the two wheels by
```

```
  taking the average change between them.
*****************************************************************************/

double GetDistanceTraveled(){
  double DistanceRight;
  double DistanceLeft;

  DistanceRight = (double)DeltaRightEncoderValue * ENC_TO_DIST;
  DistanceLeft = (double)DeltaLeftEncoderValue * ENC_TO_DIST;

  return ((DistanceRight + DistanceLeft) / 2.0);
}



/*****************************************************************************
  Function GetOrientationChange() returns the difference between the changes
  in the distance of the left and right wheels between the last two calls to
  UpdateMovement(). It makes the calculations based on the center point between
  the two wheels. MML10 made a distinction based on whether the robot was
  rotating or not when calculating the center point between the wheels. This
  distinction is included in this routine.
*****************************************************************************/

double GetOrientationChange(){
  double DistanceRight;
  double DistanceLeft;
  double OrientationChange;

  DistanceRight = (double)DeltaRightEncoderValue * ENC_TO_DIST;
  DistanceLeft = (double)DeltaLeftEncoderValue * ENC_TO_DIST;

  if(Rotating){
    OrientationChange = (DistanceRight - DistanceLeft) / TREAD_R;
  }else{
    OrientationChange = (DistanceRight - DistanceLeft) / TREAD;
  }

  return OrientationChange;
}



/*****************************************************************************
  Function SetMovement() translates the commanded linear and rotational vel-
  ocities into commanded velocities for each wheel. It then calculates the PWM
  values for each wheel and calls SetMotorControl() to execute PWM commands.
*****************************************************************************/
```

```
    void SetMovement(double LinearVelocity, double RotationalVelocity){
      double RightWheelVelocity;
      double LeftWheelVelocity;
      double CommandRightVelocity;
      double CommandLeftVelocity;
      double Tread2;
      int RpwmTemp;
      int LpwmTemp;

                RightWheelVelocity = DeltaRightEncoderValue * ENC_TO_DIST /
MOTION_CONTROL_CYCLE;
                 LeftWheelVelocity = DeltaLeftEncoderValue * ENC_TO_DIST /
MOTION_CONTROL_CYCLE;

      if(Rotating){
        Tread2 = 0.5 * TREAD_R;
      }else{
        Tread2 = 0.5 * TREAD;
      }

      CommandLeftVelocity = LinearVelocity - (Tread2 * RotationalVelocity);
      CommandRightVelocity = LinearVelocity + (Tread2 * RotationalVelocity);

      LpwmTemp = GetPwm(CommandLeftVelocity, LeftWheelVelocity, Lpwm);
      RpwmTemp = GetPwm(CommandRightVelocity, RightWheelVelocity, Rpwm);

      Lpwm = LpwmTemp;
      Rpwm = RpwmTemp;

      MotionControlWord = (MotionControlWord & 0xf0f0) |
                  ((LpwmTemp > 0) ? 1 : 2) |
                  ((RpwmTemp > 0) ? 0x100 : 0x200);

      if(RpwmTemp > PWMLIMIT){
        RpwmTemp = PWMLIMIT;
      }else if(RpwmTemp < -PWMLIMIT){
        RpwmTemp = -PWMLIMIT;
      }

      if(LpwmTemp > PWMLIMIT){
        LpwmTemp = PWMLIMIT;
      }else if(LpwmTemp < -PWMLIMIT){
        LpwmTemp = -PWMLIMIT;
      }

      SetMotorControl(LpwmTemp, RpwmTemp, MotionControlWord);
    }
```

APPENDIX D

A. USER.H

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 4, 1994
  File Name:  user.h
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains prototype for USER(). This is not required
          by the current implementation of the new MML system since user
          and kernel are two seperate object modules that are loaded at
          seperate locations. However, if the modules are combined in the
          future, main() can call USER() directly instead of user(). The
          prototype would then be required.
*****************************************************************************/

#ifndef __USER_H

#define __USER_H


/*****************************************************************************
  Function USER() is the user system program that receives control from main()
  after the robot's sub-systems have been initialized.
*****************************************************************************/
void USER();


#endif
```

B. USER.C

```
/*****************************************************************************
  Author(s):  Scott Book
  Project:    Yamabico Robot Control System
  Date:       December 8, 1993
  Revised:    March 4, 1994
  File Name:  user.c
  Environment: GCC ANSI C compiler for the motorola 68020 processor
  Description: This file contains a sample user program that can be used with
          the new MML system created using ANSI C. The purpose of this
          program is to provide an interactive menu system to control the
          robot using immediate commands only. It was also used to test
          the implementation of the immediate commands. Since the user
```

```
        module is loaded at 0x334000, USER() must be the first function
        defined.
*****************************************************************************/

#include "definitions.h"
#include "iosys.h"
#include "serial.h"
#include "motion.h"
#include "motiontrace.h"
#include "compatability.h"
#include "user.h"


#define ESC 0x1b

/*****************************************************************************
  The following section presents the prototypes for the encapsulated function(s)
  in this module.
*****************************************************************************/
static void body();



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

  PUBLIC SECTION

  The following section defines the publicly accessible USER() routine that
  is called from main().

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

/*****************************************************************************
  Function USER() is essentially a shell routine that calls body(). The reason
  for this approach is to prevent the compiler from changing the starting
  address of USER() by storing variables and/or strings before the routine.
*****************************************************************************/
void USER(){
  body();
}



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

  PRIVATE SECTION

  The following section defines the encapsulated function definition(s) used
  within this module.
```

```
*****************************************************************************/

/*****************************************************************************
  Function body() is the work horse routine of this user program. It provides
  the menu system to interact with MML's immediate commands. The routine
  purposely uses the older MML-10 commad calls in order to demonstrate use of
  the compatability macros defined in compatability.h
*****************************************************************************/

static void body(){
  int wait;
  char KeyBoardHit;
  CONFIGURATION VehicleConfiguration;

  KeyBoardHit = 0;

  motion_on();

  do{
    PutStr("\nHit: ESC to Terminate Program");
    PutStr("\n     C  to Change Robot's Configuration ");
    PutStr("\n     R  to Report Robot's Current Configuration ");
    PutStr("\n     v  to Change Linear Velocity");
    PutStr("\n     r  to Change Rotational Velocity ");
    PutStr("\n     a  to Change Linear Acceleration");
    PutStr("\n     b  to Change Rotational Acceleration ");
    PutStr("\n     S  to Change the Size Constant ");
    PutStr("\n     s  to Skip the next Sequential Instruction ");
    PutStr("\n     h  to Halt the Robot ");
    PutStr("\n     c  to Resume the Robot ");
    KeyBoardHit = GetConsole();
    PutConsole(KeyBoardHit);

    switch(KeyBoardHit){
      case ESC:
        stop0();
        PutStr("\n\nThe Total Distance Traveled is: ");
        PutReal(path_length(),4);
        break;
      case 'C':
        set_rob0(get_initial_position());
        break;
      case 'R':
        get_rob0(&VehicleConfiguration);
        report_configuration(VehicleConfiguration);
        break;
      case 'v':
        PutStr("\a\n\nEnter Desired Linear Velocity: ");
        speed0(GetReal());
        MotionTraceEnable(3); /* Log the data every 3 Motion Control Cycles. */
```

```
            break;
        case 'r':
            PutStr("\a\n\nEnter Desired Rotational Velocity: ");
            r_speed0(GetReal());
            break;
        case 'a':
            PutStr("\a\n\nEnter Desired Linear Acceleration: ");
            acc0(GetReal());
            break;
        case 'b':
            PutStr("\a\n\nEnter Desired Rotational Acceleration: ");
            r_acc0(GetReal());
            break;
        case 'S':
            PutStr("\a\n\nEnter Desired Size Constant: ");
            size_const(GetReal());
            break;
        case 's':
            skip();
            break;
        case 'h':
            halt();
            break;
        case 'c':
            resume();
            break;
        default:
            break;
      }
    }while(KeyBoardHit != ESC);

    DownLoadMotionData();

    PutStr("\n\nProgram Terminated.\a\n\n");

    motion_off();

    for(wait=0; wait<0x1000; wait++)
      ;
}
```

C. COMPATABILITY.H

```
/*****************************************************************************
  Author(s):  Scott Book
```

```
   Project:    Yamabico Robot Control System
   Date:       January 18, 1994
   Revised:    March 4, 1994
   File Name:  compatability.h
   Environment: GCC ANSI C compiler for the motorola 68020 processor
   Description: This file provides macros to convert old MML-10 immediate
            command calls into the new command calls. This allows user
            programs created for MML-10 to be used on the new system.
*******************************************************************************/

#ifndef __COMPATABILITY_H

#define __COMPATABILITY_H

#include "definitions.h"
#include "motion.h"


#define set_rob0(P) SetRobotConfiguration(P)

#define get_rob0(P) GetRobotConfiguration(P)

#define  stop0() Stop()

#define speed0(P) SetLinearVelocity(P)

#define r_speed0(P) SetRotationalVelocity(P)

#define acc0(P) SetLinearAcceleration(P)

#define r_acc0(P) SetRotationalAcceleration(P)

#define size_const(P) SetSizeConstant(P)

#define path_length() GetTotalDistance()

#define skip() SkipPathElement()

#define halt() HaltMotion()

#define resume() ResumeMotion()

#define get_initial_position() SetInitialPosition()

#define report_configuration(P) ReportRobotConfiguration(P)

#define motion_on() MotionOn()

#define motion_off() MotionOff()
```

```
#endif
```

LIST OF REFERENCES

[ALR 88]*Assembly Language Reference for the Sun-2 and Sun-3*, rev A, Sun Microsystems Inc., May 1988.

[Constantine 94]Constantine, L. L., "Mirror, Mirror" *Software Development*, vol. 2, no. 3, pp. 110-112, March 1994.

[CPG 88]*C Programmers Guide*, rev A, Sun Microsystems Inc., May 1988.

[DeClue 93]DeClue, M. J., *Object Recognition Through Image Understanding for an Autonomous Mobile Robot*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

[Kanayama 91]Kanayama, Y., and Onishi, M., "Locomotion Functions in the Mobile Robot Language, MML," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1110-1115, 1991.

[Kanayama 94]Kanayama, Y., "Two Dimensional Wheeled Vehicle Kinematics," *Proceedings of the IEEE Conference on Robotics and Automation*, 1994, to appear.

[MacPherson 93]MacPherson, D. L., *Object Recognition Through Image Understanding for an Autonomous Mobile Robot*, Ph.D Dissertation, Naval Postgraduate School, Monterey, California, September 1993.

[MC68020 85]*MC68020 32-Bit Microprocessor User's Manual*, 2d ed., Prentice Hall, 1985.

[Parnas 79]Parnas D., "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, March 1979.

[Schildt 90]Schildt, H., *C: The Complete Reference*, 2d ed, Osborne McGraw-Hill, 1990.

[Scott 93]Scott, R. C., *Reengineering Real-Time Software Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

[SRM 88]*SunOS Reference Manual*, rev A, Sun Microsystems Inc., May 1988.

[Stallman 89]Stallman, R. M., *Using and Porting GNU CC*, Free Software Foundation Inc., 1989.

[Stevens 74]Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, 1974.

[VCM 86]*VME7120: VME7120 VME CPU Module User Manual*, rev B.0, Mizar Inc., 1986.

[VDP 86]*VME7920: VME7920 Debugging Package User's Guide*, rev 1.0, Mizar Inc., 1986.

[VQS 86]*VME8300 Quad Serial Port Board User's Manual*, rev D.0, Mizar Inc., 1986.

[Yourdon 80]Yourdon, E., *The Practical Guide to Structured System Design*, Prentice Hall, 1980.

[Yourdon 89]Yourdon, E., *Modern Structured Analysis*, Yourdon Press/Prentice Hall, 1989.

[Yourdon 93]Yourdon, E., *Decline & Fall of the American Programmer*, Yourdon Press/ Prentice Hall, 1993.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
4.	Dr Yutaka Kanayama, Code CS/KA Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
5.	Lt Scott A. Book 4633 Moss Rose Dr. Ft. Worth, TX 76137	1