2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REENGINEERING REAL-TIME SOFTWARE SYSTEMS

by

Randall C Scott

September 1993

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 9 Sept. 1993	3. REPORT TYPE AND DATES COVERED Final Thesis May 92 - 9 Sept. 93

4. TITLE AND SUBTITLE Reengineering Real-Time Software Systems	5. FUNDING NUMBERS
6. AUTHOR(S) Randall C Scott	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Post Graduate School Monterey, California 93943	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The content of this report reflects the views of the author alone and not the Naval Post Graduate School or any other DOD government agency.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Unlimited Distribution	12b. DISTRIBUTION CODE A

13. ABSTRACT *(Maximum 200 words)*

The problem this thesis solves is how to reengineer existing real-time applications implemented without software engineering (SE) attributes; with poor modularity and robustness, and that are difficult to read and maintain. The real-time system chosen for this study was the Model-based Mobile robot Language (MML) used on the Yamabico-11 mobile robot, which was implemented without SE attributes.

The approach taken was reengineering MML with a focus on improving modifiability while preserving functionality. First we developed a systematic plan using manual static analysis, then we incrementally reengineered the application with thorough system-level testing. Code review was used to locate and remove dead code, and synonymous and redundant variables and functions (improving modifiability, readability and robustness). Call-hierarchy tracing was used to gain explicit module restructuring insight for tighter cohesion (improving modifiability, modularity, and readability). Global-variable tracing was used to improve module coupling by localizing and minimizing global variables (improving modularity, readability, and robustness).

The results were as follows: A method for applying SE to existing real-time applications after-the-fact called "Reengineering Real-Time Software Systems" was developed, which improves modifiability, modularity, robustness and readability. MML now has improved modularity and robustness, and is easier to read and maintain.

14. SUBJECT TERMS Reengineering, Real-Time, Software Engineering	15. NUMBER OF PAGES 73
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

REENGINEERING REAL-TIME SOFTWARE SYSTEMS

by
Randall C Scott
Captain, United States Army
BS Computer Engineering, Syracuse University, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Author: [redacted]
Randall C Scott

Approved By: [redacted]
Yutaka Kanayama, Thesis Advisor

[redacted]
Timothy Shimeall, Second Reader

[redacted]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The problem this thesis solves is how to reengineer existing real-time applications implemented without software engineering (SE) attributes; with poor modularity and robustness, and that are difficult to read and maintain. The real-time system chosen for this study was the Model-based Mobile robot Language (MML) used on the Yamabico-11 mobile robot, which was implemented without SE attributes.

The approach taken was reengineering MML with a focus on improving modifiability while preserving functionality. First we developed a systematic plan using manual static analysis, then we incrementally reengineered the application with thorough system-level testing. Code review was used to locate and remove dead code, and synonymous and redundant variables and functions (improving modifiability, readability and robustness). Call-hierarchy tracing was used to gain explicit module restructuring insight for tighter cohesion (improving modifiability, modularity, and readability). Global-variable tracing was used to improve module coupling by localizing and minimizing global variables (improving modularity, readability, and robustness).

The results were as follows: A method for applying SE to existing real-time applications after-the-fact called "Reengineering Real-Time Software Systems" was developed, which improves modifiability, modularity, robustness and readability. MML now has improved modularity and robustness, and is easier to read and maintain.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

The foundation of software engineering (SE) has been its contribution to large-scale software system development. A main goal of software engineering is the ability to write software for large-scale systems predictably by many people in such a way that the system is efficient, correct, and modifiable. Classic software engineering works note that modifiability derives from modularity, robustness, and readability. The Classical definition of modifiability is controlled change, in which some parts are altered while others remain the same in such a way that a desired new result is obtained. Modularity is achieved when the components of the system are relatively small, lightly coupled, and have strong cohesion (Yourdon,1980). Robustness allows the system to operate reliably under both optimized and worst case environmental conditions, preserving the desired behavior of the system. Readability allows for easier coordination, understanding, and maintenance during all phases of software development (Parnas, 1972).

Applied software engineering currently has most of its research efforts focused on new system development; systems that apply SE concepts at the very beginning of the life cycle. Although several examples could be cited, David L. Parnas stated it best in his paper "On the Criteria to be used in Decomposing Systems into Modules": "The modularizations include the design decisions that must be made before the work on the independent modules can begin". However, there is growing interest in reengineering existing large-scale (or real-time) systems; systems designed prior to or during the advent of applied SE (Parnas 1979, Freeman 1980). Is there a way to transform existing real-time systems to systems that achieve modularity, robustness, and readability? Can the transformed systems lend themselves more readily to portability (applications developed for multiple platform flexibility), language conversion (C to C++ and C++ to Ada), and concurrency exploitation (task candidate selection and priority task scheduling)? Considering the abundance of existing real-time systems that were not developed with applied SE concepts, the question that arises is would it be worthwhile to reengineer these systems or start from scratch? (Yourdon, 1989) The answer requires consideration of several factors. Not all real-time systems need reengineering, especially systems that have out-lived their usefulness. However, for some systems it may be determined worthwhile for some or all of the following reasons (Yourdon, 1989):

- The system is no longer reliable
- Maintenance costs are too high
- Time and Staff changes have rendered the system unreadable / unmaintainable
- Every new change introduces bugs
- New hardware technology has rendered the system non-portable
- The operational environment demands greater flexibility

If reengineering is determined worthwhile, the next step is to find a systematic method for the reengineering effort. That is the purpose of this thesis, to provide a systematic method for reengineering real-time systems. The system chosen for this study was the Yamabico-11 mobile robot, which is controlled using the Model-based Mobile robot Language (MML). The Yamabico-11 mobile robot is a real-time system; and the MML software used to control it is implemented using a mixture of assembly and C language modules, all of which were written in an ad-hoc fashion (from an engineering viewpoint) by more than a dozen implementers. Observed modification times and error rates are increasing to an unacceptable level. By improving modifiability, the overall MML software system will benefit by facilitating quicker and easier development, considerably easier maintenance, and possibly improved run-time reliability.

B. RESEARCH QUESTIONS

This thesis answers the following two questions:

1. Given an existing real-time system designed without applied SE, can any significant improvements be achieved by reengineering the system with a focus on modifiability? The context of significant meaning a measurable improvement in modularity, readability, and robustness.

2. Is reengineering feasible for the rapidly-evolving Yamabico control system, considering the systems real-time constraints, the target systems development environment, and the current implementation? Reengineering would be infeasible if the desired changes altered functionality or hinder system performance. Reengineering a real-time system concurrently with a team of several research developers would be infeasible if it caused mass confusion. Reengineering would be infeasible if SE principles can only be applied at the beginning of a development cycle.

C. THESIS ORGANIZATION

The layout of the thesis will follow the outline of the research. Chapter II describes and defines the desired software engineering attributes of the reengineered system. Chapter III covers static analysis performed to identify anomalies and describe resulting required transformations. Chapter IV describes the Yamabico-11 hardware and software system, compares the previous version (non-SE MML-3) with the new version (SE MML-10) of the MML software system, and validation of functional equivalence and ease of modification of the new version. Finally, chapter V provides a summary discussing the reengineering benefits achieved and future research.

II. DESIRED SOFTWARE ENGINEERING ATTRIBUTES

Before reengineering any system, it is important to first establish the desirable software engineering (SE) attributes for the reengineered system that do not exist in the current system. The next step is then to fabricate a systematic method to reengineer the system such that the reengineered system has the desired SE attributes. This chapter discusses modifiability, modularity, robustness, and readability. These attributes were chosen based on their interlocking support, individual merits, derived benefits, and relevance to software engineering and reengineering.

A. MODIFIABILITY

Although correctness tends to be the overall measure of success of any software system, particularly large-scale systems, modifiability makes its contribution through ease of both development and maintenance. By improving ease of development, the design and implementation phase of the software life-cycle is shorter and the time spent debugging is shorter. By improving ease of maintenance, functionality enhancements are easier to support for the finished product. In addition, improving ease of both development and maintenance benefits both the developer and the maintainer who are part of a several-member team. Consider the following example portrayed by Figure 1. A person needs to modify module 1 in subsystem X. Module 1 has an interface to module 2 in subsystem X and an interface to subsystem Y. With the given scenario, when module 1 can be modified without worry of side effects in subsystem X, subsystem Y, or the overall system, then the system is said to be truly modifiable.

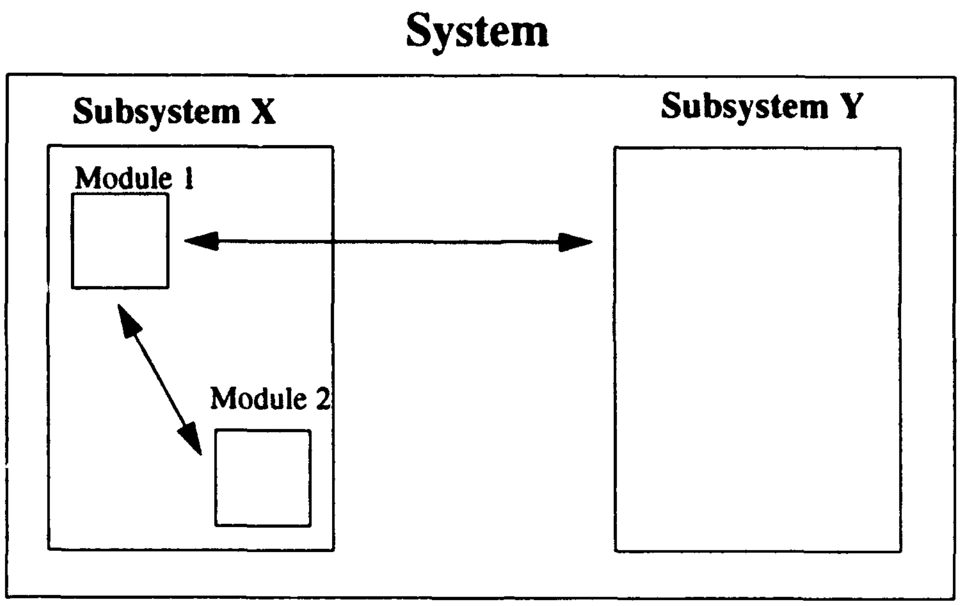


Figure 1: Modifiability

Modifiability of a system is acquired through a thorough understanding and comprehensive implementation supporting conformance to modularity, robustness, and readability. (Parnas, 1979) With modifiability as the primary focus, ease of maintenance is addressed at the front end, where it should be, rather than as an after thought as it usually is.(Yourdon, 1980) Modifiability eases system development, through modularizations that model object-oriented design (OOD) and object-oriented programming (OOP), via data encapsulation and abstraction. (Berzins & Luqi 1991, Stubbs & Webre, 1993). In addition, systems written with modifiability as the primary focus lend themselves more easily to language transformation, such as converting software written in C to C++ or C++ to Ada. Modifiability gives systems the quality and flexibility needed to accommodate within shrinking budgets the high demand for utilization using many different hardware architectures, and systems using system-subset functionality.(Yourdon 1980)

B. MODULARITY

Modularity is characterized by segmentation of the system during implementation and maintenance. Modularity is the way that the developers compartmentalize a system's functionality, and is measured through coupling and cohesion.(Parnas, 1972) Coupling is the degree to which modules share information, and cohesion is the degree to which operations are divided across modules. (Yourdon, 1989) With modifiability as the primary focus, modularity is accompiished by localization of the system's functionality; decomposing the system into subsystems, modules, and levels of abstraction (Parnas, 1979). Although the terminology associated with modularity has been heavily debated and is somewhat subjective (Parnas, 1979, Berzins & Luqi 1991), particularly in languages (such as C) that support no encapsulation of functions, modules are easier to understand when considering both logical relationships and physical storage properties. Whatever physical storage is used, the important issue is the logical decomposition that relates module functionality and the creation of modules that lend themselves to function encapsulation. Once the logical relationships are determined, the physical storage may then be considered. However, it is better to keep the physical modules as small as possible (such as 1 module per file) and consistent with the logical decomposition. Size is important because it affects readability, which in turn affects maintainability and modifiability; the larger the file, the harder it is to read and understand. A common measure of source size is one page of paper per module. (Yourdon, 1980) Also, large physical modules tend to be highly susceptible to dead code (code written but never exercised), synonymous functions and variables (functions and variables defined with different names but perform similar operations or carry the same values), and redundant functions and variables (functions and variables repeatedly defined in different modules). Dead code degrades readability and maintainability by needlessly increasing module size and forcing the developer or maintainer to read code that is never used. Synonymous functions and variables similarly degrades readability and maintainability by needlessly increasing system size, in addition to degrading reliability through confusion of which functions or variables are suppose to be used. Redundant functions and variables cause problems similar to synonymous functions and variables, however reliability problems are more related to functionality, locality, and scope. Compilers are helpful in locating redundant variables when scope can not be resolved. But when scope is resolved and redundancy still exists, then reliability is subject to degradation. An example of a reliability problem associated with function redundancy is two functions in different modules that carry the same name and scope, but are not functionally equivalent. An example of a reliability problem associated with redundant variables is assignments made to a variable with the same name but in the wrong module.

Localization is the means by which all required resources are made available to the module. The amount of localization is the measure of coupling and cohesion. The goal is to minimize coupling (well-defined module interface design and minimal use of global variables) and to maximize cohesion (localized call hierarchy and minimal use of global variables). It is important to note that global variables affect both coupling and cohesion. Figure 2, demonstrates the poor coupling and cohesion characteristics of MML-3's instruction queue functions. The variables used for the instruction queue are defined in mml.h and the functions that perform the queue operations are spread across three other files: loco.c, track.c, and main.c. This design strongly couples the four files together because modifications to one function in one file may require modifications to all four files. Also, due to the division of locality of functions and variables, the design demonstrates weak cohesion. Figure 3 demonstrates how to improve the coupling and cohesion characteristics of MML-3's instruction queue functions. By localizing the instruction queue functions and variables to a single module (file), both coupling and cohesion are improved thereby enhancing modifiability. Cohesion is strengthened by localizing the queue variables and operations to the same module. Coupling is minimized by reducing required modifications to a single module.

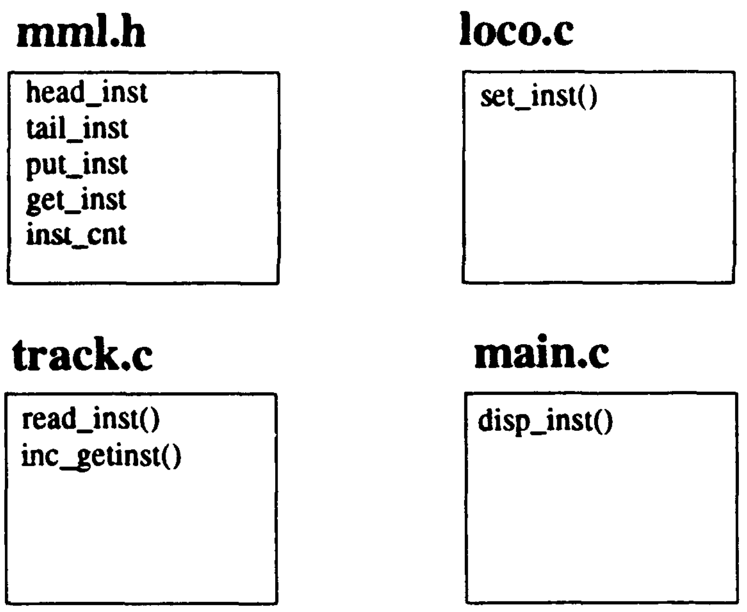


Figure 2: Strong Coupling / Weak Cohesion - Instruction Queue Functions and Variables

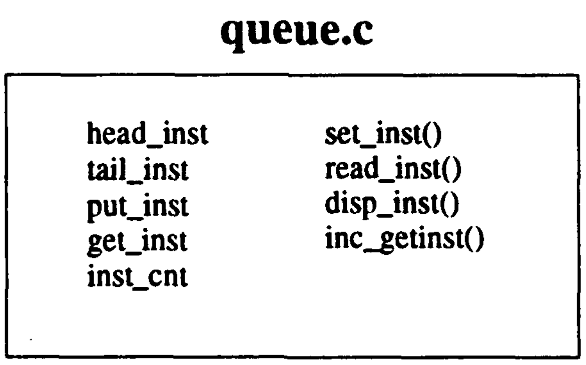


Figure 3: Weak Coupling / Strong Cohesion - Instruction Queue Functions and Variables

C. ROBUSTNESS

Robustness is defined as the degree to which the modules of a system provide an operational structure that prevents or inhibits unintended reactions to change.

Robustness is mostly reflected by reliable operation in all possible environments. The two usual means of measuring robustness are fault tolerance and breadth of processing range. Fault tolerance is "the degree to which software corrects erroneous processing." (Anderson / Lee 1989) Breadth of processing range is the fraction of possible values processed properly. In other words, a system is robust if it can detect errors and still preform satisfactorily (fault tolerance); and, provides execution paths for all required conditional flows and exception handling for unacceptable variable value ranges (breadth of processing range).

Robustness is a desirable property of behavior for real-time systems that have unpredictable, or not completely reliable, input systems; such as the sensors on an aircraft or missile. If a sensor should give incorrect information or completely fail, a robust system would not allow the aircraft to crash or the missile to hit the wrong target. However, robustness is also a function of modifiability. How reliable will a system be when changes are made? By removing synonymous and redundant functions and variables and minimizing global variables, potential real-time reliability problems caused by modifications are reduced. By encapsulating the functions that drive hardware, real-time reliability problems are easier to trace. Also as hardware technology evolves, enhancing behavior such as speed or scope of processing are much easier to deal with. Thus, by localizing hardware driving functionality in an appropriate module hierarchy, modifiability is enhanced and the overall system is more robust. Although robustness is a property of behavior and modularity is a property of the source code, both benefit substantially from modifiability. Hence, with modifiability as the primary focus, careful employment of modularity can help improve robustness.

D. READABILITY

Readability is defined as the degree to which the system behavior in operation is mirrored by the text of the system's source code.

Original authors are not always available to explain or discuss their code, the number of modules for the system and subsystems tend to be very large, and the interfacing of modules can be very complex. (Yourdon, 1980) Therefore it is important for modules to possess simplicity, locality, uniqueness, and consistency. The best modules are small, simple, and easy to read. The behavior of the code is independent of the operating context. The behavior is well defined and clearly documented such that any professional could independently read the documentation and understand the desired function of the system.

E. REENGINEERING OF THE YAMABICO-11 SYSTEM

The Yamabico-11 has a Model-based Mobile robot Language (MML) system that is a real-time system, but lacks the SE attributes described in this chapter: modifiability, modularity, robustness, and readability. MML modules do not exhibit strong cohesion or minimum coupling. A modification to one module often leads to extensive required changes to other modules. MML modules have degraded readability and maintainability because they are large in size, have inconsistent naming conventions and dead code, and contain functions that are not logically related. In addition, MML modules contain needless global variables, synonymous functions and variables, and redundant functions and variables, which not only degrade readability and maintainability, but also contribute to reliability problems.

The chapters that follow describe in detail the process used to reengineer MML for Yamabico-11 such that MML possesses the desired SE attributes of improved modifiability, modularity, robustness, and readability. Chapter III discusses static analysis methods used to measure the relative strengths of the SE attributes in the current version of MML. The static analysis methods used for Yamabico-11 were manual review of every file and function, global-variable tracing, and call-hierarchy tracing. Manual review of file sizes, lines of code per module, and the number of functions per module helps to measure the degree of readability. Manual review of the code to identify and remove dead code and needless variables helps to improve readability. Call-hierarchy tracing helps to identify poor modular design and to gain explicit insight to improved modular design. Global-variable tracing helps to measure the degree of module cohesion and coupling. In addition, call-hierarchy and global-variable tracing help to improve robustness by identifying reliability problems associated with synonymous variables and functions, and redundant variables and functions. Chapter IV then describes the details of reengineering of MML for Yamabico-11. Reengineering was performed using the results of the static analysis to restructure MML such that the modifications made provide thc desired SE attributes. In addition, chapter IV discusses the process used to transition to the reengineered version, and concludes by discussing the validation process used to ensure the reengineered version has acceptable control behavior, is functionally equivalent to the previous version, and is easy to modify.

III. STATIC ANALYSIS

In the process of analyzing a current real-time system, two static tracing techniques can be used in conjunction with each other as tools to gain insight to structure and design deficiencies. These two static analysis tracing techniques are call-hierarchy tracing and global-variable tracing.

This chapter discusses the use of call-hierarchy tracing, global-variable tracing, and the derived benefits of using these tracing techniques. In addition, this chapter shows how call-hierarchy tracing and global-variable tracing aid in formulating a module decomposition transformation and guidance for reengineering.

A. CALL-HIERARCHY TRACING

Call-hierarchy tracing is a technique that helps to resolve placement of functions and modules in addition to isolating local vs. global function call scope. More importantly, call-hierarchy tracing provides a physical picture, which is an extremely valuable tool for system design or reengineering. Once the trace is constructed, the next step is to analyze function category (local, module/subsystem local, or global). General-purpose functions are likely candidates for the global function category and should be targeted for placement in a library, such as is used for general math and input/output functions in C or packages in Ada. Global functions are found by the number of other functions that call them and the dispersion of the calling functions (i.e., the calling functions are located in different modules and their operations are not necessarily related to each other). On the other hand, if a function is only called by a single other function, then it should be local to a module that contains both. Finally, some functions may be called by a few other functions in such a way that the trace will show these functions to be local to a module or sub-system (i.e., there exists a relationship among the calling functions that allow them to coexist in a reasonable module or sub-system). Figures 4 through 6 demonstrate how a call-hierarchy trace is used to categorize functions as local, module/subsystem local, or global.

Figure 4 shows the local category, which is the easiest to identify. The local category occurs when a called function has only one calling function (function B is only called by function A). When there is only one calling function, both the calling and called functions can be placed in the same module. Figure 5 shows the module/subsystem local category. The module/subsystem local category occurs when a called function has several calling functions that have operations related to each other (function D is called by functions A and B). When the calling functions have operations related to each other, then both the calling functions and the called function can be placed in the same module or subsystem (Module C). Figure 6 shows the global category. The global category occurs when a called function has several calling function, such that none of the calling functions have operations related to each other (function H is called by functions A through G). Global category functions are best suited for libraries such as math, input/output/ and memory management. Appendix A shows a call hierarchy trace used to analyze the locomotion module for Yamabico-11.

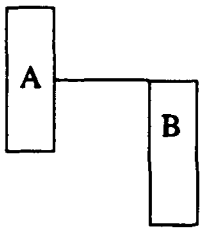


Figure 4: Local - function A calls B

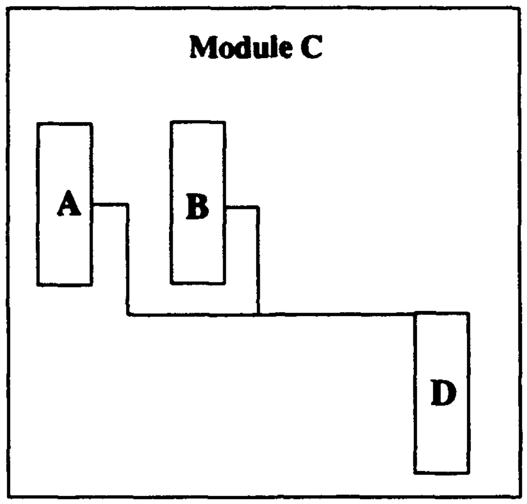


Figure 5: Module/Subsystem Local - functions A & B call function D

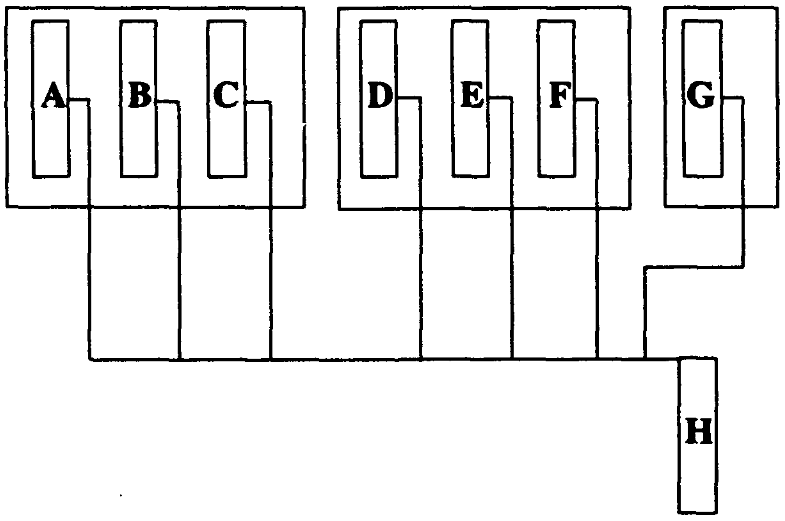


Figure 6: Global - functions A through G call function H

The output of any cross reference tool, such as *cxref* (UNIX, p. 1-124, 1990) for C, can be helpful in constructing a call-hierarchy trace. Considering that function/procedure-call cross-reference listings are usually a documentation requirement to begin with, a more-than-ample starting point exists for anyone to construct a call-hierarchy trace.

The most important direct benefit of call-hierarchy tracing is modularity design vision provided by an explicit picture of the system and how it is tied together. Another benefit is the ability to examine, to a limited extent, the current systems degree of coupling and cohesion at least at the module or file level. As a reengineering feasibility study, call-hierarchy trace is very useful in visualizing the current design, obtaining ideas fc. reengineering, and determining the relative amount of work required. Finally, if kept up-to-date a call hierarchy trace can be an invaluable tool for system maintenance or for enhancement by individuals who did not author the original code.

B. GLOBAL-VARIABLE TRACING

Global-variable tracing is probably the most difficult task of reengineering any software system of significant size. This process requires the ability to locate all variables that are not local to any given function or procedure. Although tools exist and are helpful, they are seldom a complete solution. A good example is *cxref* (UNIX, p. 1-124, 1990) for C, which is a UNIX tool that cross references variables and functions for programs written in C. The shortcoming of *cxref* is its inability to locate negated variables. Another handy but incomplete tool example is the UNIX *grep* command (UNIX, p. 1-217, 1990). *Grep* is great for locating variables (actually matching strings and string patterns). The most effective way to do a global-variable trace is to review each and every function one at a time manually, using a diagram similar to the call-hierarchy trace. Variables that are not passed as parameters or are not declared locally for the function, are annotated on the diagram function. This is a very tedious and error-prone process, which may need more than one review to ensure all global variables are located. However, this process is not required at the frontend of a reengineering effort, nor does it need to be accomplished in a sequential fashion that would inhibiting work in other areas. Figure 7 provides an example of an annotated call-hierarchy trace for functions A and B. Appendix A shows a call-hierarchy trace with annotated global variables used to analyze the locomotion module for Yamabico-11 and MML.

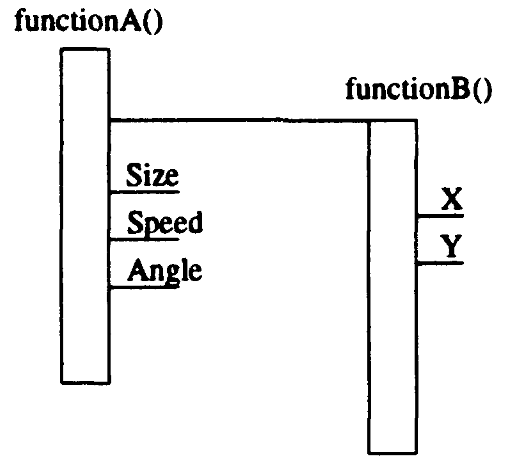


Figure 7: Global Variable Trace Annotations

Once the global variables are located the next step is similar to the call-hierarchy tracing, in that they must then be analyzed, categorized, and resolved. The goal here is to either complete elimination or minimize the number of all global variables. Using a database or data dictionary can be helpful by storing the names of all of the functions and their global variables, then sorting by global variable names to locate commonality or redundancy of use. Although automated tools would be a desirable substitute, there is no getting around the need to manually review each and every module and function to make effective design decisions regarding variables. Global variables need to be reviewed to decide whether they should be global, local to a subsystem, module, or function, or passed as a function parameter. Variables need to be reviewed to locate and remove synonyms, redundancy, and unwanted overloading (variables that have the same name but are encapsulated differently). Also, variables that impact real-time timing constraints need to be thoroughly reviewed to ensure the current implementation provides optimum performance and are clearly documented. Besides global variable analysis, variables used as function parameters also need to be reviewed to determine whether they should be passed by value or by reference. Finally, the process of locating and analyzing global variables also requires a thorough understanding of the functions and modules that use the variables in order to gain better insight to the desired design decisions.

The largest direct benefit of elimination or minimization of global variables is enhanced reliability and improved readability. Reliability is enhanced by eliminating the failures caused by erroneous use of duplicate, synonymous, and overloaded variables. Reliability is also enhanced by reducing failures associated with poorly defined module, function, and process interface communication. Readability is improved by removing or minimizing the strain of locating variables when reviewing or preforming maintenance, in addition to establishing the absolute minimum variables required for the job.

C. DERIVED BENEFITS

The derived benefits of optimizing function placement through call-hierarchy tracing and global-variable minimization through global variable tracing are improved modifiability, testability, reliability, and readability. Modifiability is enhanced from better modularity with light coupling and strong cohesion. Testability is improved as a result of improved modularity clarifying unit and module level testing, in addition to range scope reduction as a result global variable elimination. Confidence in reliability is improved through improvements in testability and better data flow communication. Finally, readability enhancements are derived from locating useless functions and variables, removal of needless global variables, identification of undesirable overloading and synonymous variables, and better overall modularity.

D. MODULE-DECOMPOSITION TRANSFORMATION

Another major job involved with reengineering any system of significant size is to preform a module-decomposition transformation. Although this sounds like a complicated process it is one of the easiest. As described briefly before, this is the process used to improve modularity with the help of the call-hierarchy trace, global-variable trace, and knowledge of the system. In a way, this portion of the reengineering effort is similar to reverse engineering in that the original design is studied to enable an improved or reengineered design, while functionality remains the same. What makes the module-decomposition transformation process somewhat easier is knowing that systems that were either poorly designed or designed without applied software engineering, are usually designed with some form of structure. Most systems embody some form of modularity at least at the file level, and with experience human habits that generate modularity anomalies (such as weak cohesion and strong coupling caused by a bias toward building modularity based on physical storage rather than logical relationships or encapsulation of functions) become more easy to detect.

The most important preparation to performing a module decomposition transformation is to have a thorough understanding of the overall system. In addition, this process tends to be most effective when working both top-down and bottom-up. Working top-down the system should first be broken up into logical families of sub-systems, then use a bottom-up approach to populate the sub-systems with reasonably-sized modules with modularity that is influenced by the call-hierarchy trace and global-variable trace.

E. GUIDANCE FOR REENGINEERING

The reengineering process and techniques described so far, provide the concepts and methods that allow dynamic flexibility with a systematic approach. The important issues remaining are the critical relationships and coordination timing constraints. For large systems and real-time systems the call hierarchy trace and the global variable trace need to be done before the module decomposition transformation. The order of performing the call hierarchy trace vs. global variable trace is not that important; in addition the order will be dependent on the tools being used and the number of people involved. I chose to do the call-hierarchy trace first so that I could use the diagram for the global-variable trace. The most critical coordination timing constraint is the module-decomposition transformation effort, which is best performed by freezing the current version of the software to inhibit functionality changes while structure is changed. Since module-decomposition transformation only affects structure, the time involved is relatively short. Also, if developers are allowed to modify functionality of previous versions, migration of modifications can be difficult to resolve due to structural displacement, not to mention any global variable eliminations.

For real-time systems, and in particular for Yamabico-11, phasing of global to local as a result of module-call-hierarchy tracing or global-variable tracing, needs to be done very carefully to preserve functionality. One difficult part is to validate changes with system-level testing as compared to unit-level testing. This is not to say that unit-level testing is not important, it just emphasizes the fact that reengineering an existing real-time system places a higher demand on ensuring that reengineering changes do not contribute to system failure. This in itself is a very time-consuming process because each change must be validated at the system level. My recommendation is to identify and document global variables at the beginning of the reengineering effort, but delay global-variable-minimization to the end. The reason is that the biggest dividend from reengineering is structural change that improves modifiability while preserving functionality. Improving modifiability acts as a multiplier not only for better modularity, readability, and maintenance; but also for further modification enhancements or for portability. As far as conducting global-variable-minimization, the reengineered system is still functional and working fine. Therefore although global-variable-minimization will improve reliability and robustness, this part of the process is not as time critical as compared to structure change, and can be done incrementally as time permits.

This chapter has discussed the value and benefits of static analysis using call-hierarchy tracing and global-variable tracing. In addition, this chapter has shown how the results of these static analysis techniques can provide a valuable road map for the reengineering effort. The next chapter will describe the real-time system used for this thesis and the reengineering effort employed.

IV. YAMABICO-11 SYSTEM DESCRIPTION

As discussed in the previous chapter, quality static analysis can provide an excellent road map for reengineering a real-time software system. However, before static analysis can be conducted on the Model-based Mobile robot Language (MML), a thorough understanding of the Yamabico-11 hardware system and the MML software system must be acquired.

This chapter describes the Yamabico-11 hardware system, the MML software system, and the reengineering of MML. Discussion includes the previous version, the reengineered version, comparisons of the previous vs. the reengineered versions, the reengineered changes, transition to the reengineered version, and validation of functional equivalence and ease of modification of the reengineered version.

A. HARDWARE

The Yamabico-11 is a mobile robot that translates in 2 dimensional space. The software that controls Yamabico is a Model-based Mobile robot Language and its hardware consists of the following sub-systems: locomotion, sonar, vision, power supply, CPU, and Input/Output.

The locomotion system consists of two DC motors, shaft encoders, a motor control circuit card, and a VME bus based interface card. The motors can drive each wheel in either the forward or reverse direction, can be set to a variety of speeds, and includes braking. The shaft encoders are used as feed back to determine distance traveled, speed, and to make odometry correction. The interface card allows the user to read information and send commands. The motor control circuit card does the actual manipulation of the motors and brakes based on information communicated through the interface card.

The sonar system consists of three groups of transmit and receive sonar cones, a sonar control circuit card, and an interface card. The sonar cones are placed in such a way as to provide the ability to capture forward/rear, lateral, or diagonal sonar data. There are four sonar cone sets for each direction and are mounted at waist high elevation. The sonar control circuit cards provide the necessary circuitry to transmit and receive sonar signals for each group. The interface card, similar to the locomotion interface card, allows the user to read information and send commands.

The vision system currently consists of a camera connected to a radio transceiver. The camera is mounted for forward looking vision at waist high elevation. The software that is used to process image information, is currently being developed and executed using an IRIS workstation coupled through a radio transceiver to the camera. A future goal is to parallel process the vision system with the sonar and locomotion systems, with all processors co-resident on the robot for full autonomy.

The CPU consists of a Motorola 68020 based motherboard, which is scheduled for an upgrade to a SPARC architecture. Power is supplied using two 12 volt motorcycle batteries. Finally, two means of communication are provided. The robot has a 9600 baud port connected to a Sun3 workstation for compiling and downloading the robot system software (MML); also a 9600 baud port connected to a laptop MacIntosh Power Book for direct communication with the robot. (Mizar, 1986) Figure 8 provides a pictorial representation of Yamabico-11.

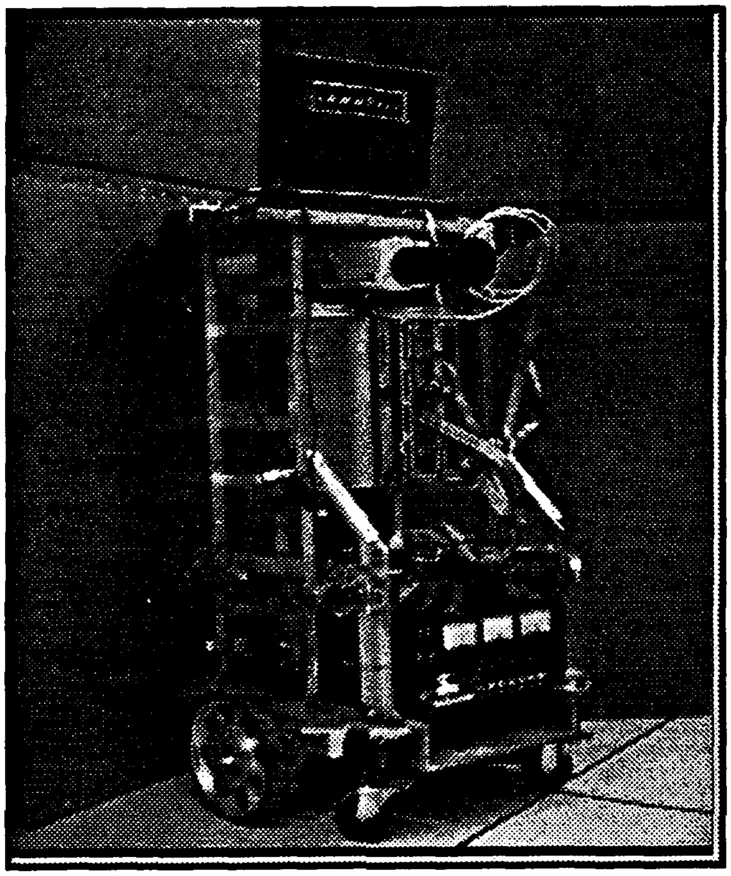


Figure 8: Yamabico-11

B. SOFTWARE (MML)

MML is a high level Model-based Mobile robot Language software system that controls the Yamabico-11 robot. The original design concept for MML was to create a general-purpose control language for autonomous mobile robots independent of the physical attributes of the robot, such as number of wheels, degrees of freedom, and drive motors. The modules for the original design were predominately composed of geometric, locomotion, sonar, and input/output functions. (Kanayama 1989/91/93, Abresch 1992) The real-time features of MML lie in the hardware interrupts used to gain background pseudo-concurrent processing. Pseudo-concurrent in that the code for any process is never interleaved with another, because processes are not allowed to share the same priority. Processes can be interleaved, but not their code. Also, since an operating system such as UNIX is not hosted on the robot, common features like spawning or forking processes dynamically are not available. This method of producing process concurrency is therefore limited to the available interrupts supplied by the CPU architecture.

There are two main processes that operate in background; one to perform the odometry corrections and one to operate the sonar. These processes are driven by hardware timers which provide highly accurate clocking. The timer for the odometry correction routines generates an interrupt every 10 msec. The odometry correction routines are the most time critical set of real-time routines for Yamabico because they must be able to start and finish within the 10 msec time interval between interrupts. The timer for the sonar timing is dependent on user supplied sonar cone configurations: any combination of group 1, 2, and/or 3. (Williams, 1992) The abort interrupt always carries the highest priority and can be generated through a physical switch on the robot. The user program interrupt is not a real interrupt, it is the absence of any other interrupts; however, Motorola documentation treats (0) as an interrupt level strap setting for the CPU motherboard. Finally, interrupt calls are layered in that masking and interrupting are only one-way. Figure 9 provides a graphic example of the interrupt levels used for Yamabico-11. The numbers in parenthesis are the physical hardware interrupt assignments, those not shown are used by the hardware system and are not available to the user. The innermost interrupt has the highest priority and can interrupt any outer interrupt if currently in operation. The rest of this section provides an overview of both the previous (MML-3) and reengineered (MML-10) versions of MML.

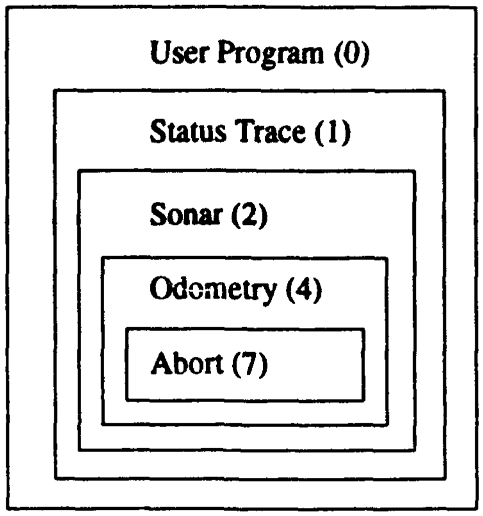


Figure 9: Interrupt Hierarchy

1. Previous MML

The previous version of MML used as the base line of study for this thesis and for the reengineered version, is called MML-3. MML-3 consists of 21 files: 3 header files, 13 C source files, and 5 assembly language files. Statistical data and graphs for MML-3 are provided in Appendix B.

Users write motion and path commands in the form of a C programs in the user.c file. (MacPherson 1993) The bulk of the functions that provide motion and path commands are predominately located in loco.c, track.c, and geom.c. These files contain the bulk of the Yamabico user command set and command queuing functions. The system kernel uses the rest of the files for initialization, math coprocessor, input/output, and control for the motors and sonar. Since MML is ported to a system which does not host an operating system such as UNIX, library functions for input/output, memory management, and math must be hand coded.

Static analysis of these files revealed that MML-3 has poor functional modularity and is difficult to read. The files were organized and named from an engineering conceptual point of view based on path planning technologies, instead of from a functional point of view that maximizes sound SE principles. Many of the files contain functions and variables that were haphazardly placed. Functions that present modular encapsulation properties, such as for queue operations, were dispersed through several files. Some variable names had synonym names elsewhere. Redundant constants, variables, and functions exist. The mml.h file was used as one large file to force global visibility for macros, functions, structures, and variables. And, with minimum use of function parameters and extensive use of global variables, MML-3 was weakly cohesive and tightly coupled. A high-level modular design based on functionality relevant to the hardware system and sub-systems was never performed.

In a real-time system, use of global variables can be a necessity to avoid context switching overhead associated with function calls, and the delays associated with passing data by value or reference through a function call-stack frame. However, global variables should be limited to those that are directly manipulated by the time sensitive routines or those that must share data among time critical processes, which for the Yamabico robot are those that perform the odometry correction and sonar processing. As the single largest contributor to unreliability, making a variable global should be a last resort to maintain strict timing constraints. In addition, periodic fine tuning for global variable reduction should be performed every time the system is enhanced by improved technology, such as a faster system clock speed, improved CPU architecture, or better optimized compilers.

Due to the size and scope of the MML system, the reengineering effort focused on modifiability, by establishing a high level modular design that allows structural change while preserving functionality. Global-variable resolution was limited to identification land isolation for future fine tuning.

2. Reengineered MML

The reengineered version of MML is now called MML-10. MML-10 consists of approximately 115 files: 34 header files, 33 C source files, 8 assembly language files, and 40 log files. The log files do not contribute to the MML system operational code, but rather towards documentation and historical information. Statistical data and graphs for MML-10 are provided in Appendix C.

Table 1 provides statistical data collected from Appendices B and C to compare total system size, average file size, and the file size standard deviation of MML-10 vs. MML-3. As can be seen from the file size data, the MML-10 version of the code has been spread out in a more even fashion. All functions and macros were placed in modules in a functional manner, and associated with a particular sub-system. The largest header files are the system header files for structures, constants, and variables. The two large module header files are OutputIO.h, that contains extensive macros; and spatial.h, which is currently just a research module and not used in the robot. Also, by reviewing the contents of the C source files, the largest files are those that have either extensive mathematical functions or functions that are very lengthy.

Table 1: File Size Comparisons

	MML-3	MML-10
Total System Size	252216	265631
Average File Size	12010	3541
File Size Standard Deviation	9897	4352

Figure 10 on the next page shows the high level modular design outline used to decompose and restructure MML-3. The files were created during reengineering.

I. System

A. Header files: mml.h - structures.h - constants.h - variables.h

II. Sub-Systems

A. Wheel System

WheelSys.h	WheelSys.c	WheelSys.asm.s	WheelSys.log
WheelMeters.h	WheelMeters.c		WheelMeters.log
	WheelLog.c		WheelLog.log

B. Sonar System

SonarSys.h	SonarSys.c	SonarSys.asm.s	SonarSys.log
SonarAvoidance.h	SonarAvoidance.c		SonarAvoidance.log
SonarCard.h	SonarCard.c		SonarCard.log
SonarIO.h	SonarIO.c		SonarIO.log
SonarLog.h	SonarLog.c		SonarLog.log
SonarMath.h	SonarMath.c		SonarMath.log

C. Computer System

ComputerSys.h			ComputerSys.log
Math68881.h		Math68881.asm.s	Math68881.log
		IOSys.asm.s	IOSys.log
		ProcArch.asm.s	ProcArch.log
InputIO.h	InputIO.c		InputIO.log
OutputIO.h	OutputIO.c	OutputIO.asm.s	OutputIO.log
MemSys.h	MemSys.c		MemSys.log
AuxIO.h	AuxIO.c		AuxIO.log

D. Motion System

MotionSys.h			MotionSys.log
PathPlan.h	PathPlan.c		PathPlan.log
PathMath.h	PathMath.c		PathMath.log
	TransitionMatrix.c		TransitionMatrix.log
Cubic.h	Cubic.c		Cubic.log
ImmCmd.h	ImmCmd.c		ImmCmd.log
	Queue.c		Queue.log
	SeqCmd.c		
Geom.h	Geom.c		Geom.log

E. Utility System

UtilitySys.h			UtilitySys.log
ConvertUtil.h	ConvertUtil.c	ConvertUtil.asm.s	ConvertUtil.log
Error.h	Error.c		Error.log
Status.h	Status.c		Status.log
StringUtil.h	StringUtil.c		StringUtil.log
	TimeSys.c	TimeSys.asm.s	TimeSys.log
AuxMath.h	AuxMath.c		AuxMath.log
Main.h	Main.c	Main.asm.s	Main.log

F. Misc. (Unrefined Research and/or Simulator)

spatial.h	spatial.c		spatial.log
	tangent.c		tangent.log
	world.c		world.log

Figure 10: Module Decomposition

MML-10 consists of 271 functions and macros, all of which were displaced based on the reengineered structure. A complete listing of the reengineered structure with functions and macros is provided at Appendix D.

To preserve functionality during the restructuring, a header-file include hierarchy was created as shown by Figure 11. This include hierarchy is part of the incremental modification method used to reengineer the MML-3 real-time system. Reengineering of mml.h was accomplished as follows:

First, based on the extern definitions in mml.h, new header files were created to be consistent with the reengineered module hierarchy structure. Then all extern definitions were migrated out of mml.h and into the new module header files. Separate header files were then created to isolate global constants, variables, and structures. Then, subsystem header files were created. What remained was to modify mml.h to include the global constants, variables, and structures header files; followed by includes of the subsystem header files. The subsystem header files then include the header files of the modules that make-up the subsystems. The new structure and nesting of the include files allowed segmentation of the previous mml.h such that the new mml.h no longer required modification, encapsulation of include file information was now modular, more visible, readable and consistent, and functionally the same as the previous mml.h after compilation. It is important to point out that this include hierarchy substantially eased the reengineering effort and improved modifiability; however, the final phase requires the xxx.c files be modified to include only those module header files required.

Header File Include Hierarchy

mml.h includes:
- structures.h
- constants.h
- variables.h
- WheelSys.h
- SonarSys.h
- ComputerSys.h
- MotionSys.h
- UtilitySys.h

WheelSys.h includes:
- WheelMeters.h
- WheelLog.h (not yet)

SonarSys.h includes:
- SonarAvoidance.h
- SonarIO.h
- SonarCard.h
- SonarMath.h
- SonarLog.h

MotionSys.h includes:
- PathPlan.h
- PathMath.h
- Cubic.h
- ImmCmd.h
- Geom.h

ComputerSys.h includes:
- ComputerSys.h
- Math68881.h
- InputIO.h
- OutputIO.h
- MemSys.h

UtilitySys.h includes:
- ConvertUtil.h
- Error.h
- Status.h
- StringUtil.h
- AuxMath.h
- Main.h

All xxx.c files include mml.h

Figure 11: Header Include Hierarchy

3. Comparison of Previous vs. Reengineered versions

Comparing the previous version with the reengineered version reveals the answer to the first research question of this thesis: can significant measurable improvements be made through reengineering with a focus on modifiability. It is important to point out here that this thesis was started using MML-3 as the base line of study for an evolving developmental system. Therefore, the comparisons are made using MML-3 and MML-10. However, MML-10 reflects new functionality (cubic spiral path tracking functions) and minor modifications added to other versions between MML-3 and MML-10. The structural changes reflected in MML-10 encapsulate the functionality of the MML-8 and MML-9 versions.

The measurable improvements can be found by comparing the statistical data provided in Appendices B and C. The data shows that MML-10 has 54 more files than MML-3 with an increase of 13415 bytes in total system file size. The increase in file count was expected in order to achieve better modularity and improved readability. However, several reductions were accomplished consisting of: a reduction of 281 total lines of code, a reduction in average file size of 8469 bytes, a reduction in the average number of functions per file by 9, and a reduction in average lines of code per file by 148. The reductions achieved provide better modifiability by improving modularity and readability. Also, by locating and removing redundant code, dead code, and synonymous variables, MML-10 is inherently more robust than MML-3. The increase in total file size was due to the offset of the development team adding functionality from MML-3 to MML-10. This increase in functionality somewhat countered the effects of redundant and dead code elimination, and slightly distorts the accuracy of the statistical data provided. However, considering the size of MML the distortion presented by the added functionality is insignificant relative to the reductions in average file sizes, average number of lines of code, and average number of functions per module. Therefore, if functionality would have remained the same, the only increase would have been the total file count.

The method used to count lines of code sometimes raises credibility issues as to what is a line of code. Does the count include blank lines and comments in addition to a line of code. If one line of code is spread across several lines, does it count as one or all the lines it occupies? The method used to count the lines of code reflected in Appendices B and C, was through the use of the UNIX tools *nl* and *grep*. (UNIX, p. 1-340 and 1-217, 1990) The command *grep -c ';' *.h *.c*, was used to count lines of code in both the header and C source files, assuming a semicolon provides a fairly accurate count of a line of code for C source code. Therefore if one long line of code is spread across several lines, it gets a count of one. To count the lines of code in the assembly language files, the difference was calculated between the lines reported from the *nl *.s* command and the *grep -c '\<[#]' *.s* command. The *nl* command produces a line count of an ASCII file that does not count blank lines; the complicated *grep* command provides a count of assembly language comment lines. The total non-blank lines in the file minus the comment lines equals the total lines of code for assembly language source code.

C. TRANSISTION TO REENGINEERED SYSTEM

Although not an easy task, transition to the reengineered version of MML seemed easy because of the effort to work with developers as modifications were made, and by preserving system functionality during reengineering. One hardship incurred was that developers who used the previous version (MML-3) must reorientation themselves by learning the reengineered version: structure, modularity, function mapping, etc. To ease this transition, a database was used to map the functions from their old locations to their new locations. The database can be used to aid a developer in tracing function movements and understanding the new systems modularity. A module decomposition and system level hierarchical design was created, analyzed, and coordinated with the group before structural changes were made, and updated as structural refinements were conducted. As incremental structural changes were made, all members were notified of the changes. Finally, a comprehensive system manual was established and is being written for reference by all developers, which will include the module decomposition, hierarchical design, and function map. The system manual will document the Yamabico-11 hardware and software, and provide user, operator, implementer, . .ıd tester manuals in addition to research theory for path planning technologies and SE paradigms.

D. TRANSFORMATION VALIDATION

Validating a reengineered transformation requires significant testing at all levels: unit-level, integration-level, and system-level testing. However, unit and integration-level testing are less importance with a reengineered system as compared to new system. For a reengineered real-time system, unit-level and integration-level testing are best during the end of the reengineering effort, and as an aid to solving reliability anomalies (such as changing variable scope or structure that now causes system failure) that did not surface until reengineering started. Also, if the reengineered system represents significant changes in structure, most existing unit and integration-level test harnesses will be useless or in need of reconstruction (i.e., the reengineered structural change no longer allows useful testing of existing test harnesses because of function and/or variable displacement, however functionality of the previous test harnesses may be used to reconstruct new ones). Finally, unit-level and integration-level test harness construction requires a significant amount of time that cuts sharply into the precious time spent acquiring a functional reengineered system. Eventually, unit and integration-level test harnesses need to be constructed for all modules, but can be built as needed and the end of the reengineering effort for full SE compliance (modifiable, modular, robust, and readable).

System-level testing is the most important validation testing for a reengineered real-time system. The reason system-level testing is the most important is because functionality preservation always carries the highest priority during the reengineering effort. With preservation of functionality at the highest of priorities, testing at the system level needs to be exercised often as the system undergoes incremental change during the reengineering effort. In addition to validating functional equivalence, validation of ease of modification is also important for the reengineered system. Validating ease of modification helps to measure the quality of modularity and structure design. The next sections discuss how MML was validated for functional equivalence and ease of modifiability.

1. Validate MML Functional Equivalence

Validation of functional equivalence for MML was somewhat complicated. It was infeasible to conduct unit-level testing because the time needed to construct test harnesses that could exhaustively exercise 271 functions exceeded the research time available. In addition, the reengineering effort did not alter any functionality of the existing functions; reengineering changes were limited to structure changes with redundant and dead code removal. Therefore, it was assumed that all functions were previously validated. However, unit-level testing still needs to be developed for each of the modules created. It is noteworthy to point out that previous unit-level test harnesses do not exist and their absence can partially be blamed on the construction difficulty presented by the previous poor modular design, which lacked functional modularity and exhibited strong coupling with weak cohesion. Also, the reengineered version now lends itself more easily for unit-level test harness construction. Integration-level test harnesses were also non-existent. The only testing facilities available to validate functional equivalence were user programs written to exercise motion control and demonstrate path movement.

Therefore, testing MML-10 was validated for functional equivalence at the system level using the user programs that exercised motion control and path movement. This a very tedious process involving making incremental structural changes, then compiling, down loading, and run-time testing the changes made. Having 271 functions available for execution, this kind of testing will not tax them all, nor will it adequately cover the scope for input/output data set verification. However, this method of testing did help to ensur structural changes did not impair important expected behaviors of the robot and was useful to isolate structural changes that did. Based on the testing facilities available and the fact that the reengineered version did not alter previous function functionality, the reengineered version (MML-10) is functionally equivalent to the previous version (MML-3).

2. Validate MML Ease Of Modifiability

Validation of ease of modification lies mostly with the structural changes that reflect good modular design and improved readability in the reengineered version (MML-10). By having a high-level design based on the hardware system and its sub-systems, with reasonably-sized modules that have consistent and meaningful names, code is significantly easier to locate when changes are desired or when bugs need to be traced. By possessing the features of strong cohesion and loose coupling, modules can be modified more easily by many developers of a team with less fear of side effects and with a higher confidence in run-time reliability, making the system inherently more robust. The header-file include hierarchy allows module headers to be more easily constructed and modified, then added or subtracted from the overall system with minimum changes in the makefile (a C compiler file used to define compilation instructions). By careful placement of code and stabilization of header file definitions, the compile-download-execute time cycle has been substantially improved because the whole system no longer requires compilation every time a change is made.

This chapter described the Yamabico-11 hardware system, the MML software system, and the reengineering effort. In addition, this chapter presented how MML was reengineered, provided comparisons between the previous and reengineered versions, discussed transition steps taken with respect to the reengineered version, and the process used to validate functional equivalence and ease of modification of the reengineered version.

V. SUMMARY AND CONCLUSIONS

A. REENGINEERED CHANGES

The highlight of reengineering for MML is the nice structure formed and the modular design. All modules now exhibit loose coupling and tight cohesion. All modules are functionally complete. A consistent naming convention was established. All global constants, variables, and structures were identified and localized at the system level. An include hierarchy was created through nested calls to improve modifiability and readability, which both contribute to ease of maintenance and reliability. The size of the new modules and organization lends the MML system substantially easier to read, modify, and maintain. Also the functional completeness of modules allow for easier and quicker co-development by a many member team with minimum interference or side effects. The new structure has significantly reduced the code and compile cycle time.

In addition, journal files were created for each new module with a filename the same as the module and a filename extension called *log*. The journal files provide a handy way to document changes made and provide a historical reference for each module. A bug journal (called Bugs.log) was also created as a means to coordinate with the group as a whole concerning problems and potential problems for the entire system.

Finally, the reengineered MML is now in a form that lends itself more easily for transformation into C++ or Ada. For C++, the current module header files would need to be converted to prototypes and encapsulated classes. The C++ transformation would allow an object oriented approach that is consistent with the modular design. For Ada, the header files could be used to generate the specifications and the C source files could then be used to generate the procedure/function bodies. The modules could then be packaged as needed. An Ada transformation would allow exploitation of total concurrency through tasking (which can also be strapped to hardware interrupts through representation clauses), and easier exception handling. Either transformations would provide enhancements through the special features the of the languages used.

B. BENEFITS OF REENGINEERING

The most important benefits attained by reengineering is transforming a system into one that conforms to sound SE principles. This is a long-term payoff that maps a worthwhile system into one that is competitive, portable, flexible, and easy to maintain and enhance. A system whose maintainability and ease of enhancement are not dependent on the institutional knowledge of the development team or the individual programmer. A system whose evolution dependencies are primarily oriented toward hardware technology and compiler efficiency. The resultant system helps reduce the cost and time involved with maintenance. All of these positive assets can be achieved through reengineering with a focus on modifiability.

This thesis has therefore answered the two research questions presented.

1. Given an existing real-time system designed without applied SE, significant improvements can be achieved by reengineering with a focus on modifiability. Significant improvements reflected by the improved structure and modular design, the degree of improved coupling and cohesion, and the higher confidence in run-time reliability. Also, improved modifiability and readability reflected by reductions in the average module size, average lines of code per module, and average number of functions per module. By making modifiability the focus of reengineering a real-time system, optimization of structure and modularity becomes more intuitive, identification of poor coupling and cohesion becomes more obvious, and improved readability becomes almost a complete derivation.

2. Reengineering is feasible for an evolving developmental real-time system, considering the systems real-time constraints, the target systems development environment, and the current implementation. By reengineering structural changes so that modules encapsulate hardware driving functionality, real-time constraints are easier to maintain and fine-tune. The development environment needs to be carefully coordinated so that all developers are working with the same version, do not interfere with each other, and do not experience wasted time. As long as the hardware used is stable, and all developers exercise cooperative coordination, reengineering the current implementation is very feasible. By limiting reengineering changes to structural changes with redundant and dead code removal, reengineering will not cause undesirable changes in functionality or hinder system performance; instead, it will help locate sections of code that contribute to unreliability and improve readability. With effective coordination, it is feasible to reengineer a real-time system concurrently with a team of several research developers. Finally, SE can be applied either at the very beginning of the development cycle, or after fielding of a system designed and implemented without SE.

C. FUTURE RESEARCH

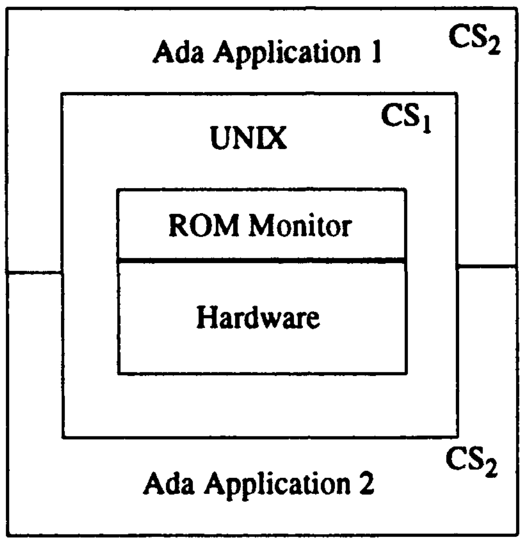
This has been a very rewarding research project that has provided tangible results (success is visible on the Yamabico-11 robot). If more time were available, global variable reduction or elimination could be conducted for all of the MML system, unit and integration-level test harnesses could be built, the MML code could be made ANSI-C-compliant, assembly language modules could be minimized or eliminated through transformations to C, and the MML system could be converted to run on the SPARC architecture (a near future upgrade currently scheduled to replace a Motorola 68020 architecture).

As for future research, one area that would have been valuable for Yamabico and real-time system research is the benefit of development from the perspective of a development environment. An environment that provides cross-compiler technology and comes with standard libraries that support ROMable code, transparent initialization, memory management, input/

output, and math functions; eliminating the need to hand code these in C or at the assembly language level. An environment that supports both C and C++ compiling with meaningful reported errors and warnings performed both on syntax and semantics. An environment that does not force the developer to become dependent on cryptic makefiles for application building. An environment that provides remote symbolic debugging to speed the trace of real-time timing constraint violations and general debugging. The current disadvantage presented to the Yamabico team is that the desired research is more oriented towards path-planning technologies; however, considerable research time gets spent due to the weakness of the development environment.

Another area of res^ rch that would be valuable to both Yamabico and real-time research is tasking through Ada, which is currently infeasible due to resource constraints. Two alternatives exist to execute Ada code on Yamabico, either host UNIX on the robot or use the cross-compiler technology already mentioned.

Hosting UNIX on the robot is not the best approach for several reasons. First, it would require mounting a mass storage device such as a hard drive on the robot. Mounting a mass storage device on the robot increases the overall weight of the robot and the demand for power, which is already in short supply. In addition, hosting UNIX to access Ada tasking will not allow Ada to directly access to the hardware. Ada representation clauses used to map hardware interrupts or system memory are implemented through system addresses. Ada programs gain access only by permission and after review by UNIX. UNIX is a multi-user operating system written to execute several user programs. UNIX is also a protected system written to limit access to critical resources. Therefore, UNIX is a barrier between the application wanting direct access and control of the hardware, and the actual hardware itself. For hardware control applications that have lax timing constraints or do not have time-sensitive operations, this may be acceptable. However, for hardware control applications that have strict real-time constraints, like Yamabico-11, this is unacceptable. Finally, besides not having direct access to the hardware, hosting UNIX forces two levels of context switching: UNIX system and applications context switching and Ada task context switching. Two levels of context switching distorts concurrency analysis of a real-time system that would be preferred to be relative to the Ada application alone. Figure 12 shows the barrier hosting UNIX presents, in that the users program can only access the hardware indirectly through the operating system, and concurrency analysis for the real-time system is distorted due to the two levels of context switching.



CS_i: Context Switching level i

Figure 12: UNIX Barrier

Embedded system development is the best approach, but requires the purchase of cross-compiler technology. Embedded system development allows an application to be truly plugged into the system hardware, freeing the developer to control all aspects of the hardware using a high-level language suited for the job such as Ada. Ada representation clauses can be assigned to actual memory addresses and physical hardware interrupts. Context switching is reduced to a single level thereby removing distortion in concurrency analysis associated with the UNIX operating system. In addition, concurrency for non-hardware interrupt driven tasks could be written, which for the current Yamabico-11 and MML-10 do not exist. An Ada embedded-system development environment would encourage experimentation and analysis of task-scheduling algorithms and mixed-mode preemptive vs. non-preemptive tasking. And, an Ada embedded-system development environment would not limit the number of concurrent tasks to the number of system hardware interrupts available, which limits the current implementation of MML for Yamabico-11. Figure 13 demonstrates the beauty of barrier-free Ada application that has direct access to the hardware and single level context switching. A barrier-free Ada application can only be accomplished through the use of an Ada embedded system development cross-compiler.

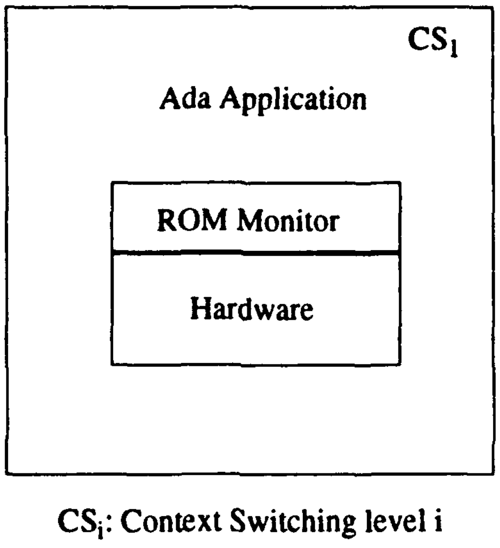
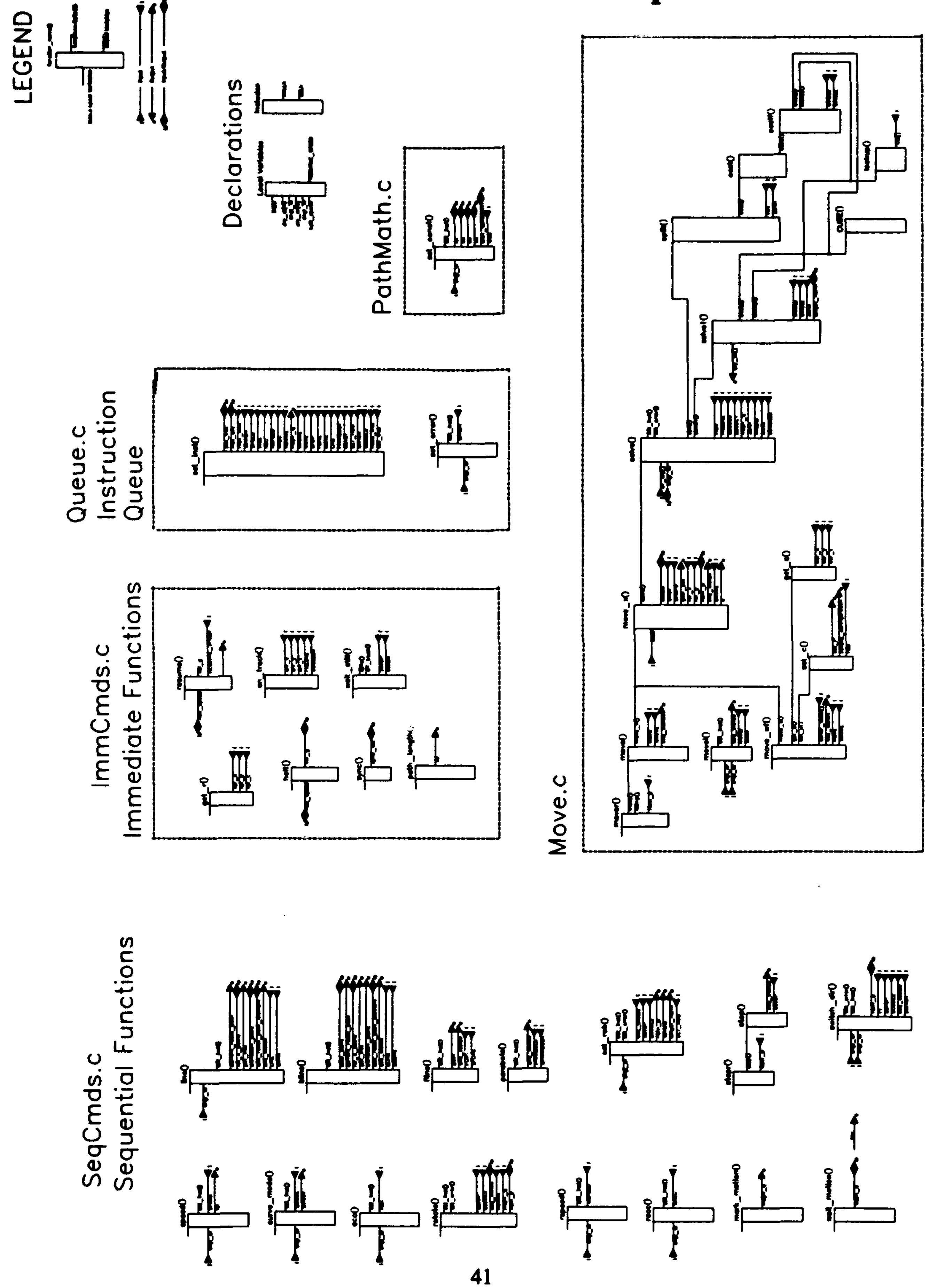


Figure 13: Barrier Free

APPENDIX A - Loco.c Map



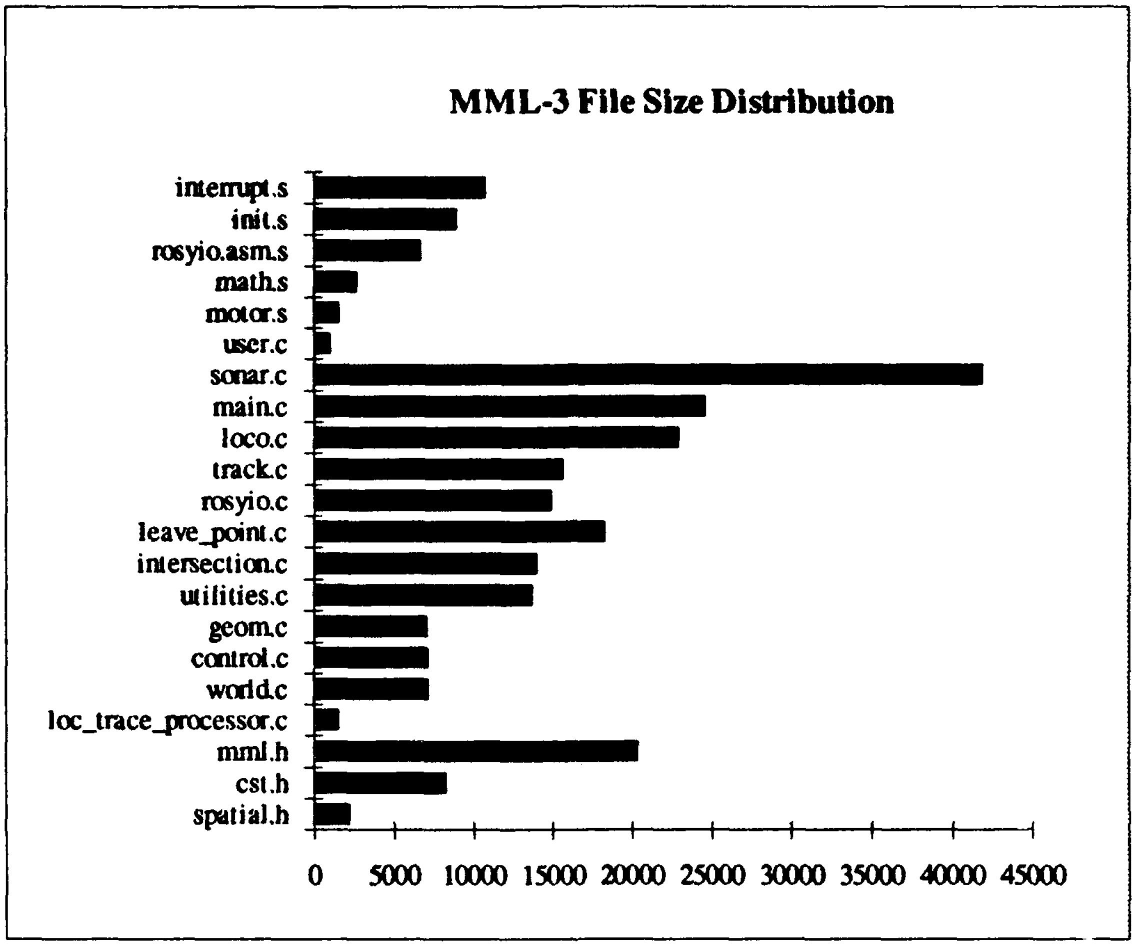
APPENDIX B - MML 3 Statistics

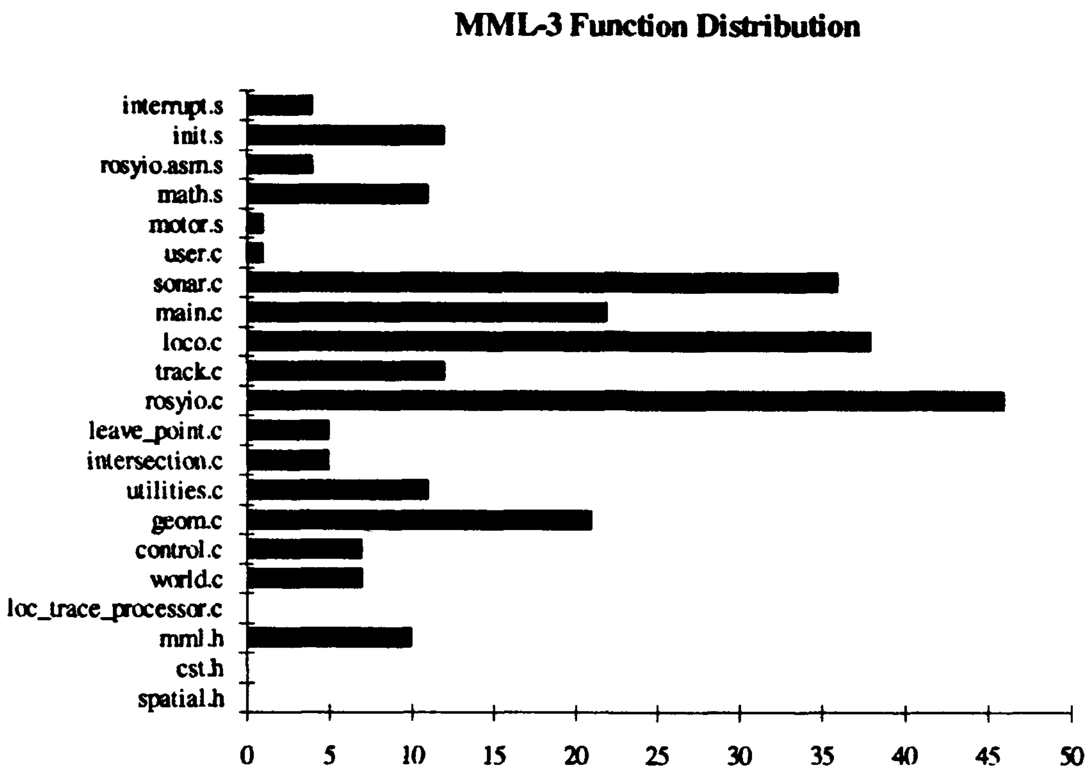
MML-3 Files	File Size	Functions	Lines of Code
spatial.h	2251	0	34
cst.h	8323	0	1
mml.h	20384	10	230
loc_trace_processor.c	1547	0	39
world.c	7196	7	65
control.c	7156	7	96
geom.c	7139	21	113
utilities.c	13759	11	89
intersection.c	14019	5	164
leave_point.c	18347	5	251
rosyio.c	14945	46	243
track.c	15694	12	277
loco.c	23003	38	351
main.c	24672	22	552
sonar.c	41937	36	529
user.c	1012	1	30
motor.s	1600	1	72
math.s	2705	11	162
rosyio.asm.s	6735	4	261
init.s	9004	12	299
interrupt.s	10788	4	343

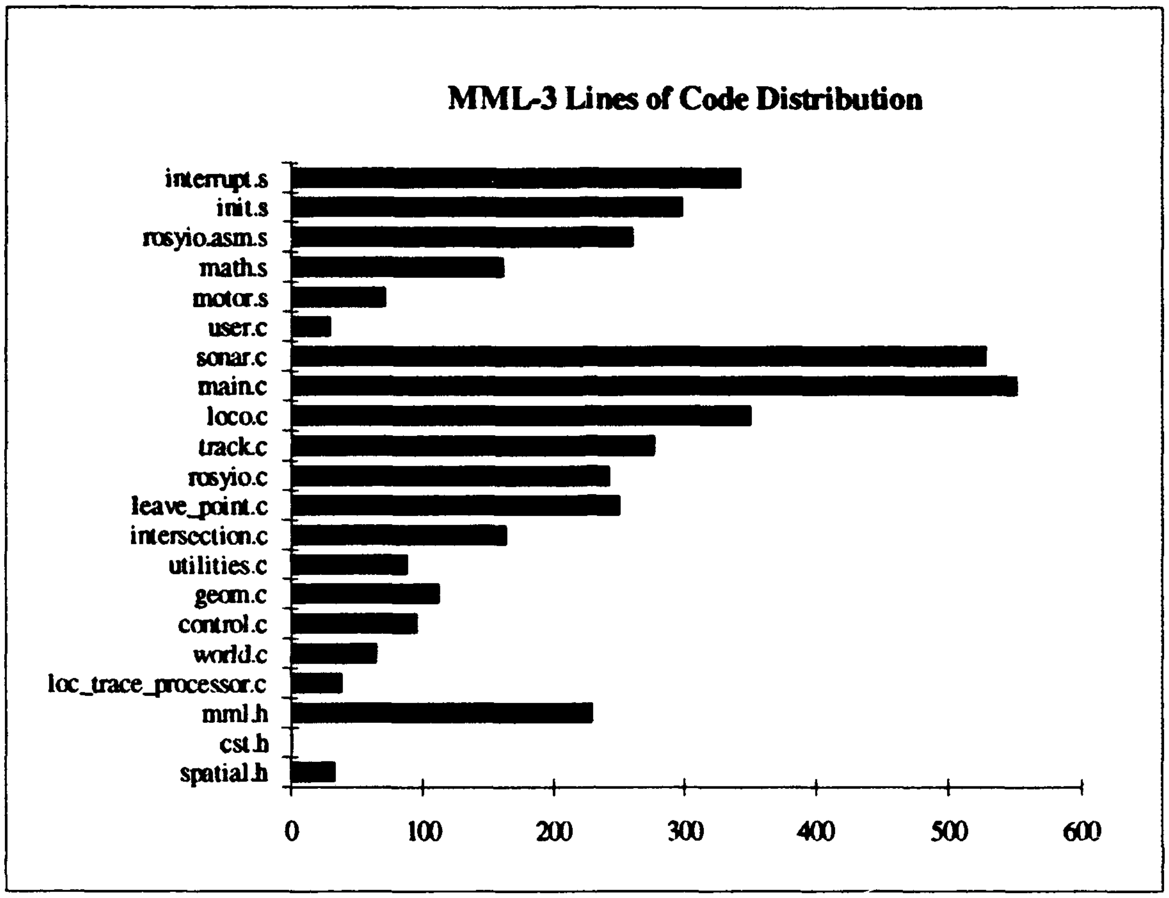
MML-3 File Sizes	
Mean	12010.28571
Standard Error	2159.911415
Median	9004
Standard Deviation	9897.957553
Variance	97969563.71
Kurtosis	2.873049978
Skewness	1.435257463
Range	40925
Minimum	1012
Maximum	41937
Sum	252216
Count	21

MML-3 Functions	
Mean	12.04761905
Standard Error	2.905230251
Median	7
Standard Deviation	13.31343754
Variance	177.247619
Kurtosis	1.297452399
Skewness	1.454556648
Range	46
Minimum	0
Maximum	46
Sum	253
Count	21

MML-3 Lines of Code	
Mean	200.047619
Standard Error	34.16448143
Median	164
Standard Deviation	156.5613222
Variance	24511.44762
Kurtosis	0.191734028
Skewness	0.836164898
Range	551
Minimum	1
Maximum	552
Sum	4201
Count	21


MML-3 File Size Distribution
interrupt.s
init.s
rosyio.asm.s
math.s
motor.s
user.c
sonar.c
main.c
loco.c
track.c
rosyio.c
leave_point.c
intersection.c
utilities.c
geom.c
control.c
world.c
loc_trace_processor.c
mml.h
cst.h
spatial.h
0
5000
10000
15000
20000
25000
30000
35000
40000
45000


MML-3 Function Distribution
interrupt.s
init.s
rosyio.asm.s
math.s
motor.s
user.c
sonar.c
main.c
loco.c
track.c
rosyio.c
leave_point.c
intersection.c
utilities.c
geom.c
control.c
world.c
loc_trace_processor.c
mml.h
cst.h
spatial.h
0
5
10
15
20
25
30
35
40
45
50



MML-3 Lines of Code Distribution
interrupt.s
init.s
rosyio.asm.s
math.s
motor.s
user.c
sonar.c
main.c
loco.c
track.c
rosyio.c
leave_point.c
intersection.c
utilities.c
geom.c
control.c
world.c
loc_trace_processor.c
mml.h
cst.h
spatial.h
0
100
200
300
400
500
600

APPENDIX C - MML 10 Statistics

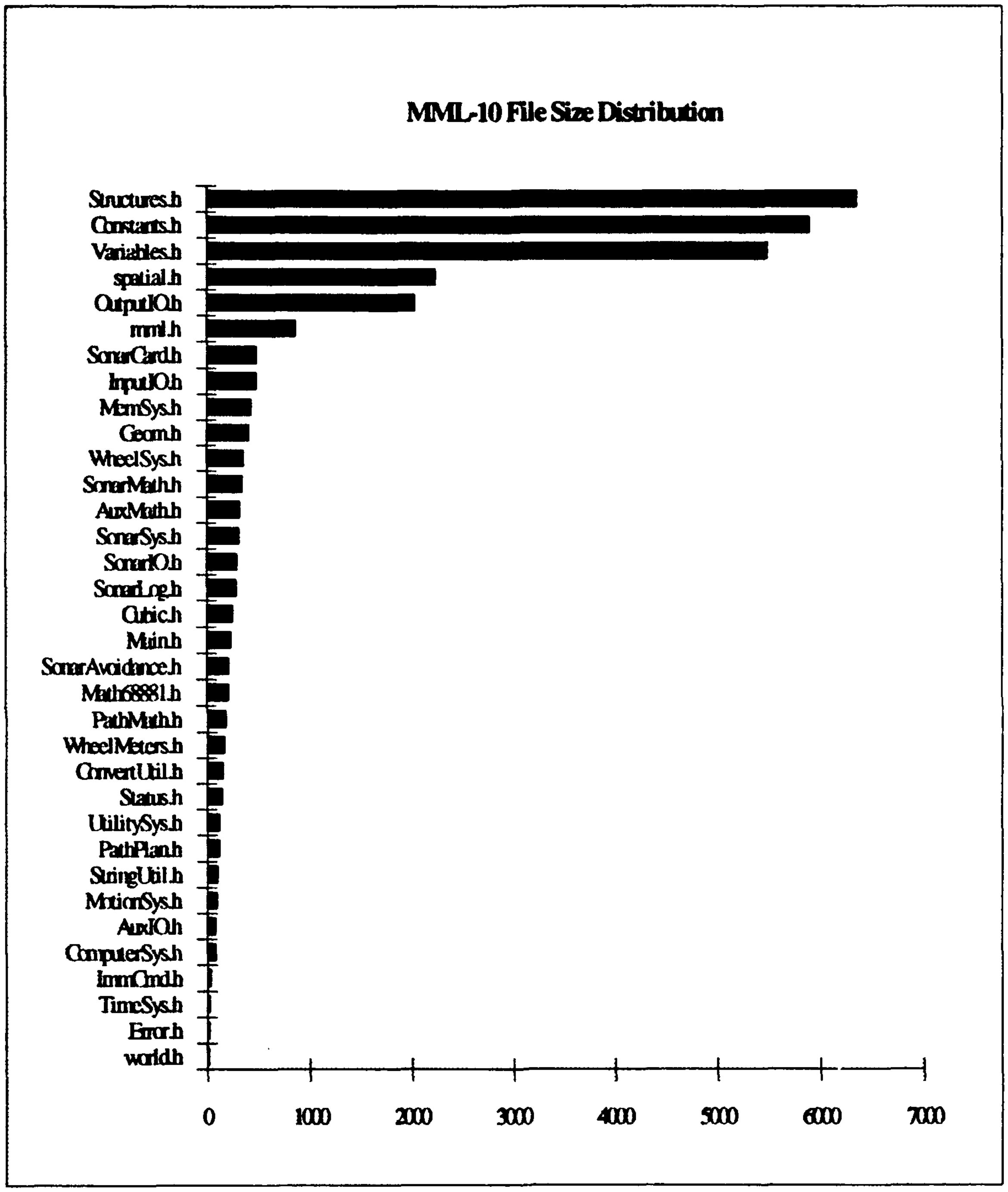
MML-10 Files	File Size	Functions	Lines of code
world.h	20	0	0
Error.h	27	0	1
TimeSys.h	27	0	1
ImmCmd.h	38	0	1
ComputerSys.h	86	0	0
AuxIO.h	89	0	2
MotionSys.h	102	0	0
StringUtil.h	107	2	0
PathPlan.h	125	0	3
UtilitySys.h	128	0	0
Status.h	153	0	3
ConvertUtil.h	159	0	2
WheelMeters.h	174	0	3
PathMath.h	190	0	5
Math68881.h	218	0	3
SonarAvoidance.h	219	0	6
Main.h	241	2	0
Cubic.h	259	0	8
SonarLog.h	297	0	4
SonarIO.h	300	0	4
SonarSys.h	320	0	3
AuxMath.h	327	9	0
SonarMath.h	350	0	10
WheelSys.h	365	0	7
Geom.h	417	3	6
MemSys.h	440	3	7
InputIO.h	486	3	3
SonarCard.h	486	0	9
mml.h	873	0	7
OutputIO.h	2041	19	5
spatial.h	2240	0	34
Variables.h	5493	0	113
Constants.h	5900	0	0
Structures.h	6360	0	77

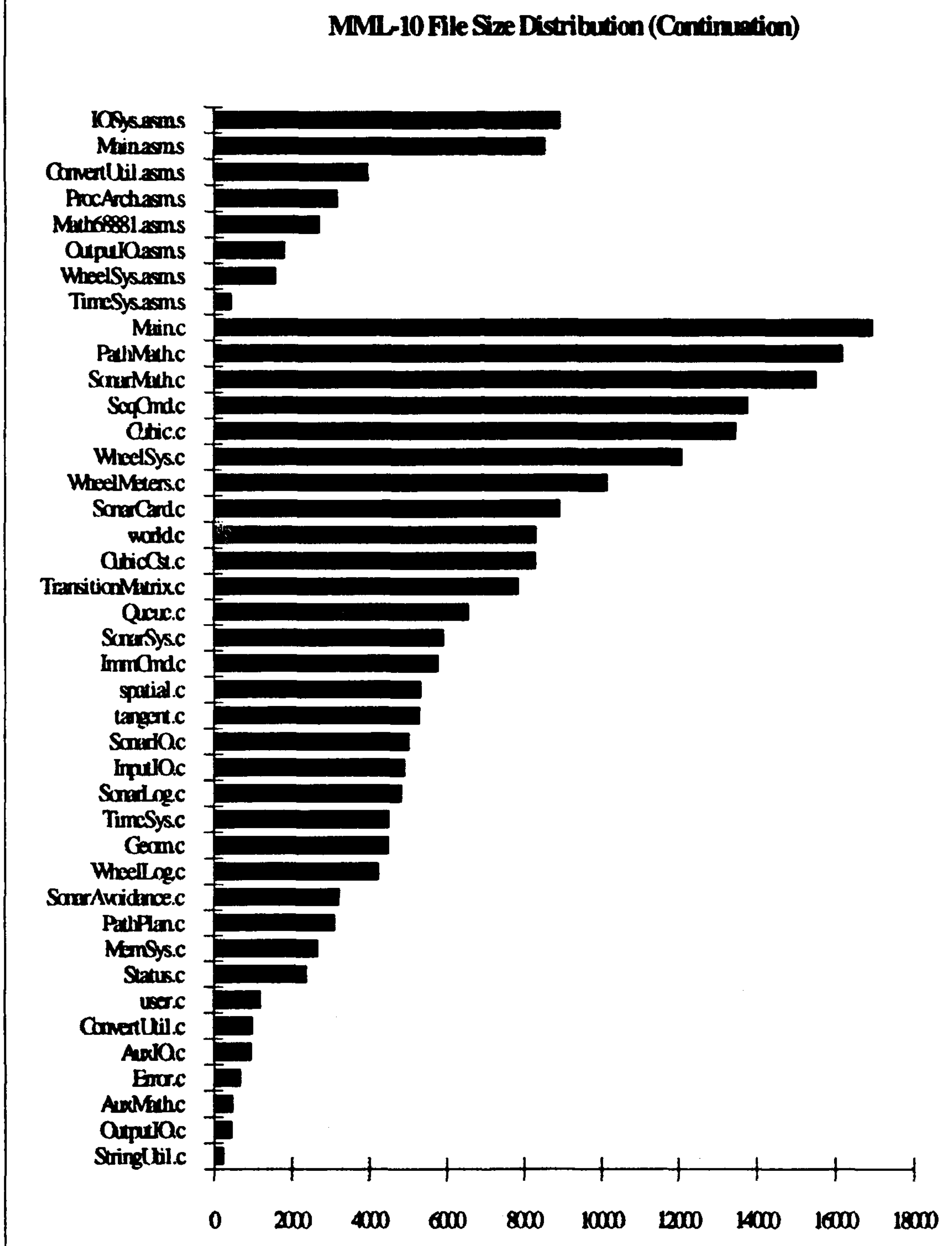
MML-10 Files	File Size	Functions	Lines of code
StringUtil.c	255	2	6
OutputIO.c	465	1	5
AuxMath.c	495	1	7
Error.c	683	1	8
AuxIO.c	981	1	4
ConvertUtil.c	1001	1	21
user.c	1220	1	18
Status.c	2393	3	37
MemSys.c	2678	8	74
PathPlan.c	3121	5	38
SonarAvoidance.c	3240	6	37
WheelLog.c	4250	5	58
Geom.c	4502	11	56
TimeSys.c	4516	7	77
SonarLog.c	4849	4	41
InputIO.c	4930	4	77
SonarIO.c	5043	4	81
tangent.c	5306	3	42
spatial.c	5337	5	33
ImmCmd.c	5781	12	55
SonarSys.c	5932	4	76
Queue.c	6581	3	134
TransitionMatrix.c	7881	4	109
CubicCst.c	8323	0	1
world.c	8336	8	83
SonarCard.c	8942	9	91
WheelMeters.c	10173	8	149
WheelSys.c	12100	11	155
Cubic.c	13489	8	144
SeqCmd.c	13782	18	156
SonarMath.c	15536	10	212
PathMath.c	16208	15	207
Main.c	16964	15	411
TimeSys.asm.s	460	1	14
WheelSys.asm.s	1600	1	57
OutputIO.asm.s	1827	1	56
Math68881.asm.s	2705	11	155
ProcArch.asm.s	3172	5	73
ConvertUtil.asm.s	3989	2	96
Main.asm.s	8567	5	189
IOSys.asm.s	8961	6	250

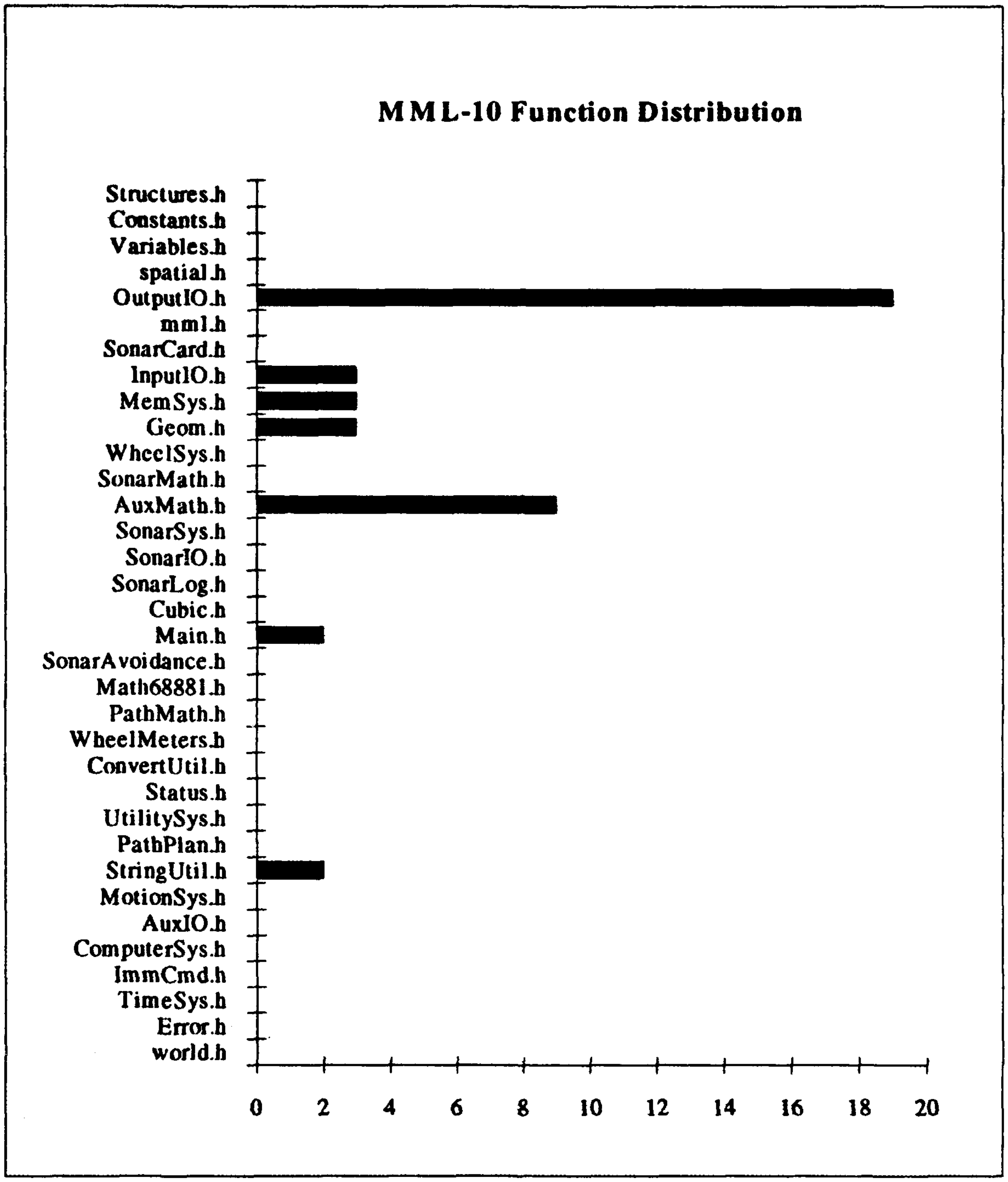
MML-10 File Sizes	
Mean	3541.746667
Standard Error	502.6322959
Median	1600
Mode	27
Standard Deviation	4352.92337
Variance	18947941.87
Kurtosis	1.573204792
Skewness	1.47317113
Range	16944
Minimum	20
Maximum	16964
Sum	265631
Count	75

MML-10 Functions	
Mean	3.613333333
Standard Error	0.534534683
Median	2
Mode	0
Standard Deviation	4.629206147
Variance	21.42954955
Kurtosis	1.873659105
Skewness	1.520305396
Range	19
Minimum	0
Maximum	19
Sum	271
Count	75

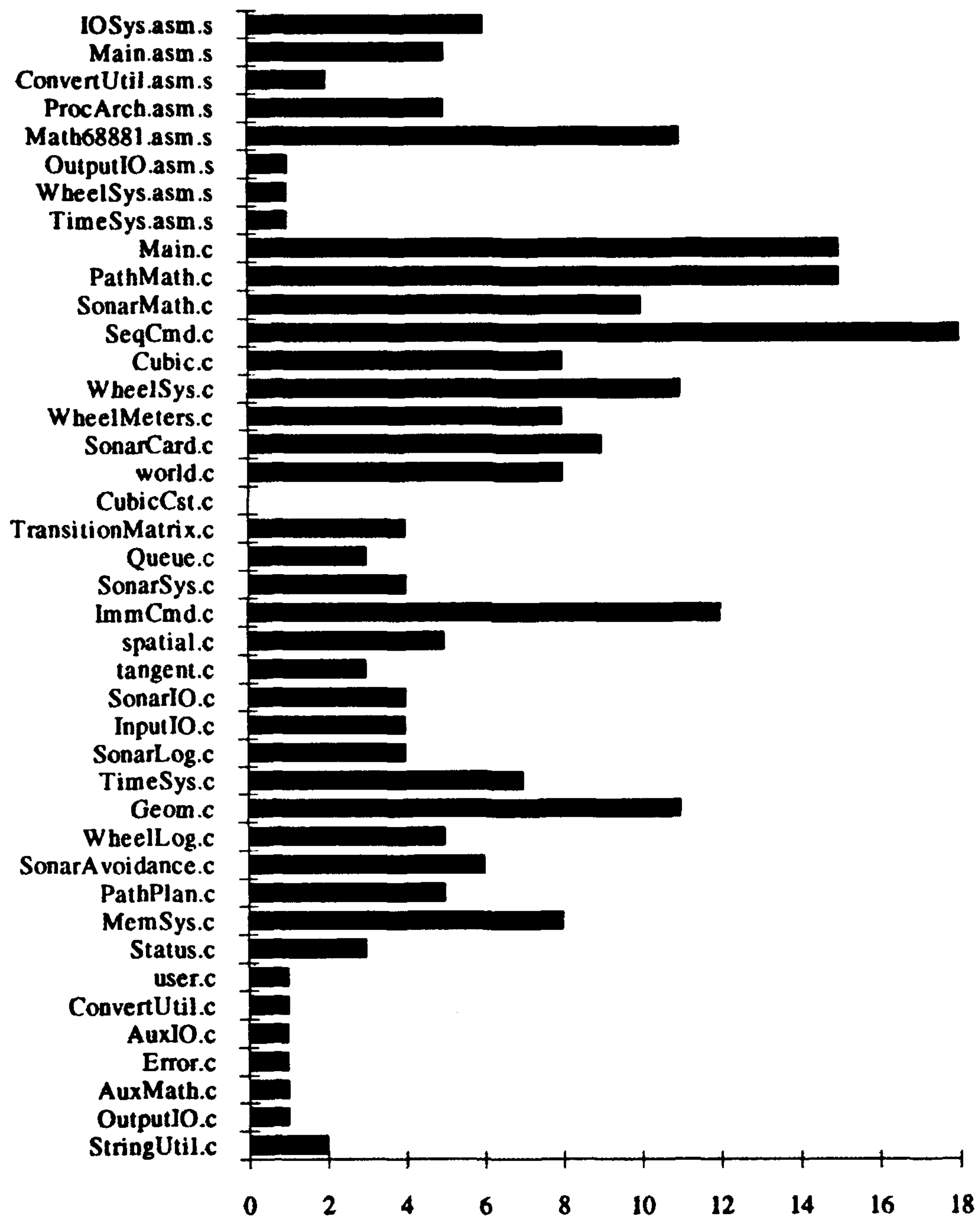
MML-10 Lines of Code	
Mean	52.26666667
Standard Error	8.498422394
Median	14
Mode	0
Standard Deviation	73.59849685
Variance	5416.738739
Kurtosis	7.091089635
Skewness	2.297196259
Range	411
Minimum	0
Maximum	411
Sum	3920
Count	75

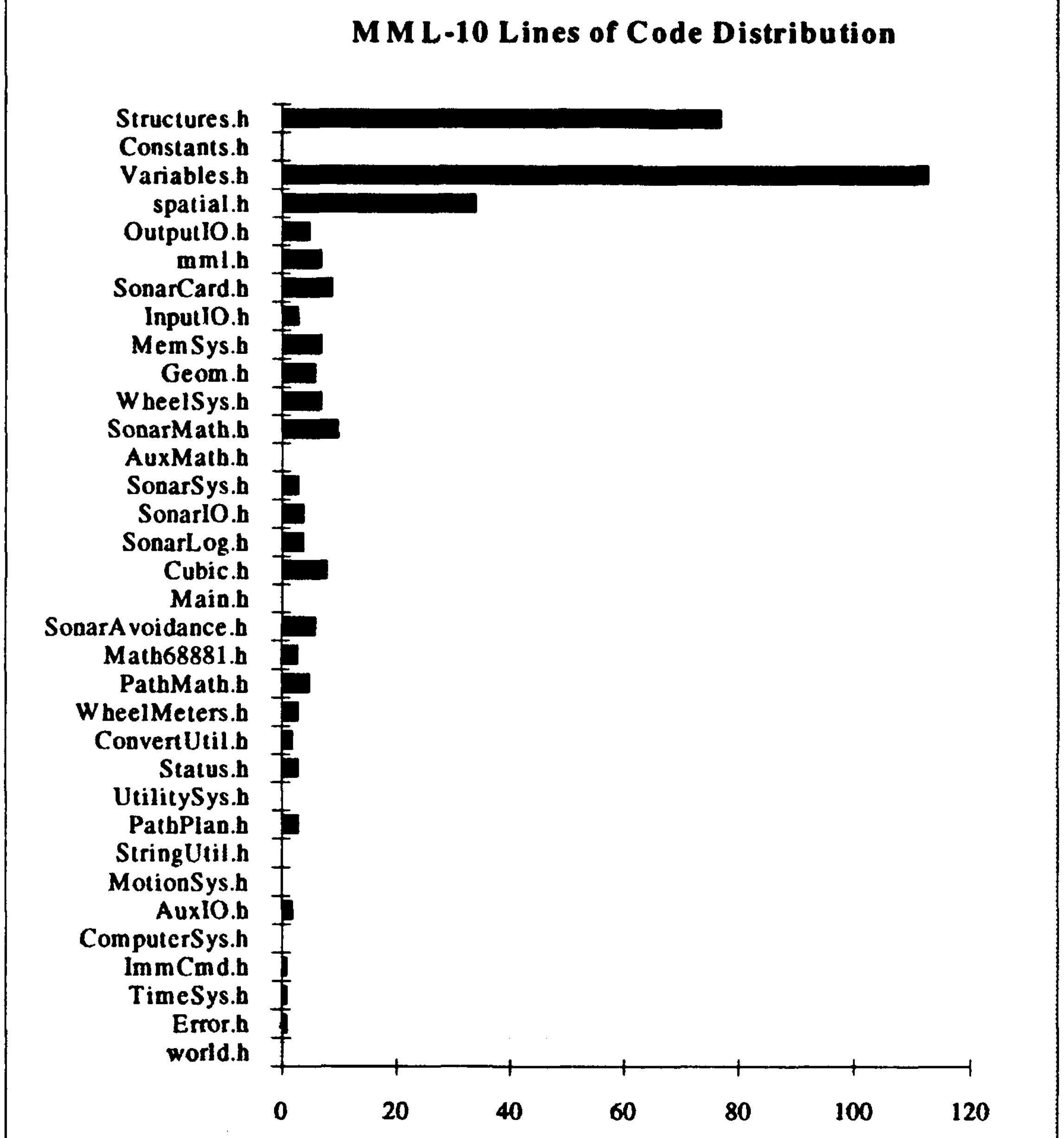

MML-10 File Size Distribution
Structures.h
Constants.h
Variables.h
spatial.h
OutputIO.h
mml.h
SonarCard.h
InputIO.h
MemSys.h
Geom.h
WheelSys.h
SonarMath.h
AuxMath.h
SonarSys.h
SonarIO.h
SonarLog.h
Cubic.h
Main.h
SonarAvoidance.h
Math68881.h
PathMath.h
WheelMeters.h
ConvertUtil.h
Status.h
UtilitySys.h
PathPlan.h
StringUtil.h
MotionSys.h
AuxIO.h
ComputerSys.h
ImmCmd.h
TimeSys.h
Error.h
world.h
0
1000
2000
3000
4000
5000
6000
7000

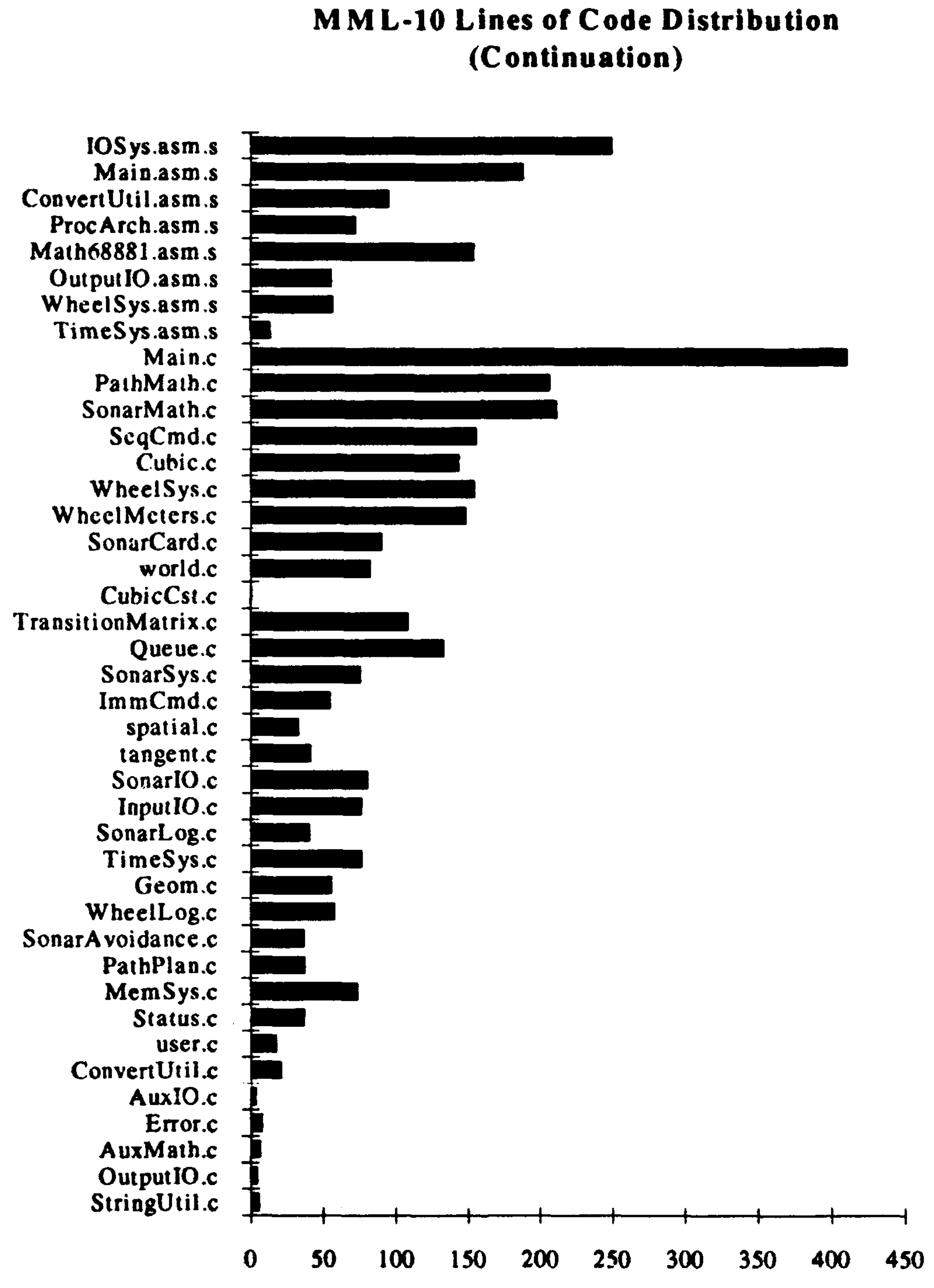

MML-10 File Size Distribution (Continuation)
IOSys.asms
Main.asms
ConvertUtil.asms
ProcArch.asms
Math68881.asms
OutputIO.asms
WheelSys.asms
TimeSys.asms
Main.c
PathMath.c
SonarMath.c
SeqCmd.c
Cubic.c
WheelSys.c
WheelMeters.c
SonarCard.c
world.c
CubicCst.c
TransitionMatrix.c
Queue.c
SonarSys.c
ImmCmd.c
spatial.c
tangent.c
SonarIO.c
InputIO.c
SonarLog.c
TimeSys.c
Geom.c
WheelLog.c
SonarAvoidance.c
PathPlan.c
MemSys.c
Status.c
user.c
ConvertUtil.c
AuxIO.c
Error.c
AuxMath.c
OutputIO.c
StringUtil.c
0
2000
4000
6000
8000
10000
12000
14000
16000
18000


MML-10 Function Distribution
Structures.h
Constants.h
Variables.h
spatial.h
OutputIO.h
mml.h
SonarCard.h
InputIO.h
MemSys.h
Geom.h
WheelSys.h
SonarMath.h
AuxMath.h
SonarSys.h
SonarIO.h
SonarLog.h
Cubic.h
Main.h
SonarAvoidance.h
Math68881.h
PathMath.h
WheelMeters.h
ConvertUtil.h
Status.h
UtilitySys.h
PathPlan.h
StringUtil.h
MotionSys.h
AuxIO.h
ComputerSys.h
ImmCmd.h
TimeSys.h
Error.h
world.h
0
2
4
6
8
10
12
14
16
18
20

M?L-10 Function Distribution (Continuation)




MML-10 Lines of Code Distribution
Structures.h
Constants.h
Variables.h
spatial.h
OutputIO.h
mml.h
SonarCard.h
InputIO.h
MemSys.h
Geom.h
WheelSys.h
SonarMath.h
AuxMath.h
SonarSys.h
SonarIO.h
SonarLog.h
Cubic.h
Main.h
SonarAvoidance.h
Math68881.h
PathMath.h
WheelMeters.h
ConvertUtil.h
Status.h
UtilitySys.h
PathPlan.h
StringUtil.h
MotionSys.h
AuxIO.h
ComputerSys.h
ImmCmd.h
TimeSys.h
Error.h
world.h
0
20
40
60
80
100
120


MML-10 Lines of Code Distribution
(Continuation)
IOSys.asm.s
Main.asm.s
ConvertUtil.asm.s
ProcArch.asm.s
Math68881.asm.s
OutputIO.asm.s
WheelSys.asm.s
TimeSys.asm.s
Main.c
PathMath.c
SonarMath.c
SeqCmd.c
Cubic.c
WheelSys.c
WheelMeters.c
SonarCard.c
world.c
CubicCst.c
TransitionMatrix.c
Queue.c
SonarSys.c
ImmCmd.c
spatial.c
tangent.c
SonarIO.c
InputIO.c
SonarLog.c
TimeSys.c
Geom.c
WheelLog.c
SonarAvoidance.c
PathPlan.c
MemSys.c
Status.c
user.c
ConvertUtil.c
AuxIO.c
Error.c
AuxMath.c
OutputIO.c
StringUtil.c
0
50
100
150
200
250
300
350
400
450

APPENDIX D - MML 10 Function Displacement

Wheel System

WheelLog.c	loc-troff()
WheelLog.c	loc-tr_dump()
WheelLog.c	loc-tr_resume()
WheelLog.c	loc_tron()
WheelLog.c	store_loc_trace_data()
WheelMeters.c	correct_odometry_error()
WheelMeters.c	get_robot_speed()
WheelMeters.c	get_rotational_vel()
WheelMeters.c	get_velocity()
WheelMeters.c	pwm_lookup()
WheelMeters.c	read_left_wheel_encoder()
WheelMeters.c	read_right_wheel_encoder()
WheelMeters.c	read_rotate()
WheelSys.asm.s	motor
WheelSys.c	control()
WheelSys.c	end_of_motion()
WheelSys.c	get_initial_position()
WheelSys.c	initialize_current_conf()
WheelSys.c	limit()
WheelSys.c	report_configuration()
WheelSys.c	rest_of_path()
WheelSys.c	update_delta_d()
WheelSys.c	update_image()
WheelSys.c	update_kappa()
WheelSys.c	update_vel()

Sonar System

SonarAvoidance.c	avoid_obstacle()
SonarAvoidance.c	disable_obstacle_avoidance()
SonarAvoidance.c	disable_wall_following()
SonarAvoidance.c	enable_obstacle_avoidance()
SonarAvoidance.c	enable_wall_following()
SonarAvoidance.c	wall_follow()
SonarCard.c	disable_interrupt_operation()
SonarCard.c	disable_linear_fitting()
SonarCard.c	disable_sonar()
SonarCard.c	enable_interupt_operation()
SonarCard.c	enable_linear_fitting()
SonarCard.c	enable_sonar()
SonarCard.c	reset_accumulators()
SonarCard.c	serve_sonar()
SonarCard.c	wait_sonar()

SonarIO.c	host_xfer()
SonarIO.c	xfer_global_to_host()
SonarIO.c	xfer_raw_to_host()
SonarIO.c	xfer_segment_to_host()
SonarLog.c	disable_data_logging()
SonarLog.c	enable_data_logging()
SonarLog.c	log_data()
SonarLog.c	set_log_interval()
SonarMath.c	add_to_line()
SonarMath.c	calculate_global()
SonarMath.c	end_segment()
SonarMath.c	finish_segments()
SonarMath.c	get_current_segment()
SonarMath.c	get_segment()
SonarMath.c	global()
SonarMath.c	linear_fitting()
SonarMath.c	sonar()
SonarMath.c	start_segment()
SonarSys.c	build_list()
SonarSys.c	get_sonar_config()
SonarSys.c	set_parameters()
SonarSys.c	update_sonar()

Computer System

ProcArch.asm.s	fpc_exception
ProcArch.asm.s	i_ih
ProcArch.asm.s	i_mask
ProcArch.asm.s	i_maskoff
ProcArch.asm.s	i_maskon
MemSys.c	delete()
MemSys.c	Free()
MemSys.c	free()
MemSys.c	invF()
MemSys.c	mallac()
MemSys.c	pop()
MemSys.c	push()
MemSys.c	release()
MemSys.h	Active()
MemSys.h	Lbc()
MemSys.h	Size()
IOSys.asm.s	ih_tty
IOSys.asm.s	i_clock
IOSys.asm.s	I_serial
IOSys.asm.s	i_stopwatch
IOSys.asm.s	i_timer
IOSys.asm.s	reset_timer

InputIO.c	getc()
InputIO.c	getint()
InputIO.c	getreal()
InputIO.c	getstr()
InputIO.h	bufM()
InputIO.h	r_getchar()
InputIO.h	whitespace()
OutputIO.asm.s	putb
OutputIO.c	putstr()
OutputIO.h	nl1()
OutputIO.h	nl2()
OutputIO.h	nl3()
OutputIO.h	nl4()
OutputIO.h	nll()
OutputIO.h	nln()
OutputIO.h	nl_flex()
OutputIO.h	nl_log()
OutputIO.h	print1()
OutputIO.h	print1count()
OutputIO.h	print2()
OutputIO.h	print3()
OutputIO.h	print4()
OutputIO.h	printfi()
OutputIO.h	printfr()
OutputIO.h	printloc()
OutputIO.h	printn()
OutputIO.h	print_flex()
OutputIO.h	print_log()
AuxIO.c	fatal()
Math68881.s	acos
Math68881.s	atan
Math68881.s	atan2
Math68881.s	cos
Math68881.s	cot
Math68881.s	exp
Math68881.s	log
Math68881.s	sin
Math68881.s	sqrt
Math68881.s	tabs
Math68881.s	tan

Utility System

AuxMath.c	ceil()
AuxMath.h	cube()
AuxMath.h	d2r()
AuxMath.h	max2()
AuxMath.h	min()
AuxMath.h	min2()
AuxMath.h	r2d()
AuxMath.h	SQR()
AuxMath.h	sqr()
AuxMath.h	SQRT()
ConvertUtil.asm.s	itoa
ConvertUtil.asm.s	rtoac
ConvertUtil.c	rtoa()
StringUtil.c	strcmp()
StringUtil.c	strcpy()
StringUtil.h	isdigit()
StringUtil.h	point()
Error.c	disp_error()
Status.c	change_status()
Status.c	display_status()
Status.c	enable_display_status()
TimeSys.asm.s	get_time
TimeSys.c	input_time()
TimeSys.c	output_time()
TimeSys.c	reset_clock()
TimeSys.c	time()
TimeSys.c	timer()
TimeSys.c	wait_timer()
TimeSys.c	wait_until()

Main.asm.s	ih_displaystatus
Main.asm.s	ih_loco
Main.asm.s	ih_sonar
Main.asm.s	i_wheel
Main.asm.s	r_exit
Main.c	disp_ref_posture()
Main.c	exe_real()
Main.c	exe_sim()
Main.c	imaskoff()
Main.c	imaskon()
Main.c	i_control()
Main.c	i_globl()
Main.c	i_loco()
Main.c	i_port()
Main.c	i_real()
Main.c	i_sim()
Main.c	i_var()
Main.c	i_var_sim()
Main.c	main()
Main.c	stepper()
Main.h	JUMP_USER()
Main.h	r_exit()[SIM]

Motion System

PathPlan.c	report_path()
PathPlan.c	set_length_stop()
PathPlan.c	wait_point()
PathPlan.c	wait_segment()
PathPlan.c	wait_segment1()
TransitionMatrix.c	circle_and_circle()
TransitionMatrix.c	line_and_circle()
TransitionMatrix.c	line_and_line()
TransitionMatrix.c	line_and_parabola()
Queue.c	inc_getinst()
Queue.c	read_inst()
Queue.c	set_inst()
ImmCmd.c	acc0()
ImmCmd.c	get_line0()
ImmCmd.c	get_rob0()
ImmCmd.c	halt()
ImmCmd.c	path_length()
ImmCmd.c	resume()
ImmCmd.c	r_acc0()
ImmCmd.c	r_speed0()
ImmCmd.c	set_rob0()
ImmCmd.c	size_const0()
ImmCmd.c	speed0()
ImmCmd.c	stop0()

SeqCmd.c	acc()
SeqCmd.c	bline()
SeqCmd.c	config()
SeqCmd.c	fline()
SeqCmd.c	line()
SeqCmd.c	mark_motion()
SeqCmd.c	parabola()
SeqCmd.c	rotate()
SeqCmd.c	r_acc()
SeqCmd.c	r_speed()
SeqCmd.c	set_error()
SeqCmd.c	set_rob()
SeqCmd.c	size_const()
SeqCmd.c	skip()
SeqCmd.c	speed()
SeqCmd.c	switch_dir()
SeqCmd.c	sync()
SeqCmd.c	wait_motion()
Geom.h	DIST()
Geom.h	EU_DIS()
Geom.h	PAR_LN()
Geom.c	area()
Geom.c	def_configuration
Geom.c	def_porabola()
Geom.c	def_sym()
Geom.c	def_sym1()
Geom.c	negate()
Geom.c	nnorm()
Geom.c	norm(a)
Geom.c	normalize()
Geom.c	norm_zero_to_pi()
Geom.c	pnorm()
PathMath.c	cfacc2()
PathMath.c	comp()
PathMath.c	distf()
PathMath.c	get_path_intersection()
PathMath.c	get_s_zero()
PathMath.c	get_transition_point()
PathMath.c	inverse()
PathMath.c	new_delta_d()
PathMath.c	next()
PathMath.c	parallel()
PathMath.c	project_path()
PathMath.c	transition_point_test()
PathMath.c	zera()
PathMath.c	zero()
PathMath.c	zerod()

Cubic.c	advance_cubic_image()
Cubic.c	cost()
Cubic.c	costf()
Cubic.c	lookup()
Cubic.c	solve()
Cubic.c	solve1()
Cubic.c	split()
Cubic.c	update_cubic_image()

User Program

User.c	user()

Miscellaneous non-functional Research Modules

spatial.c	linearize()
spatial.c	order()
spatial.c	orientation()
spatial.c	segment_crossing_test()
spatial.c	switch_mode()
tangent.c	common_tangent()
tangent.c	is_tangent()
tangent.c	tangent()
world.c	add_polygon_to_world()
world.c	add_vertex_to_polygon()
world.c	build_world()
world.c	create_isolated_vertex()
world.c	create_line_segment()
world.c	create_polygon()
world.c	create_vertex_pair()
world.c	create_world()

REFERENCES

Abresch, R., *Path Tracking using Simple Planar Curves*, Master's Thesis, Naval Post-Graduate School, Monterey California, March 1992.

Anderson, T. and Lee, P. A., *Fault Tolerance: Principles and Practice*, Second Edition, Springer-Verlag, New York, 1989.

Berzins V. and Luqi, *Software Engineering with Abstractions*, Addison-Wesley Publishing Company, New York, 1991.

Freeman P. and Wasserman A., *Tutorial on Software Design Techniques*, 3rd Edition, p. 434, 1980.

Kanayama, Y., and Hartman B.I., *Smooth Local Path Planning for Autonomous Vehicles*, Part I: Symmetricity, Proceedings IEEE Journal of Robotics and Automation, p. 1265-1270, 1989.

Kanayama, Y., Onishi, M., *Locomotion Functions in the Mobile Robot Language, MML*, Proceedings of the 1991 IEEE International Conference on Robotics and Automation, p. 1110-1115, 1991.

Kanayama, Y., MacPherson D., Alexander J., *MML Locomotion Functions Using Path Specifications*, Draft MML language design paper, Naval Post-Graduate School, Monterey, California, 1 February 1993

MacPherson, D, *Yamabico User's Manual*, Draft Users manual, Naval Post-Graduate School, Monterey, California, 15 January 1993.

Mizar Inc., *VME Quad Serial Port Board*, User's Manual, Manual Revision D.0, Board Revision D, First Edition, 1986.

Parnas D., *On the Criteria to be used in Decomposing Systems into Modules*, Communications of the ACM, December 1972.

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

Stubbs D. and Webre N., *Data Structures with Abstract Data Types and Ada*, PWS-Kent Publishing Company, Massachusetts, 1993.

UNIX Reference Manual, *Section 1: Commands (cxref)*, p. 1-124, Solbourne Computer Inc., Colorado, 1990.

UNIX Reference Manual, *Section 1: Commands (grep)* , p. 1-217, Solbourne Computer Inc., Colorado, 1990.

UNIX Reference Manual, *Section 1: Commands (nl)*, p. 1-340, Solbourne Computer Inc., Colorado, 1990.

Williams, M., *Documentation for Yamabico Sonar Ranging Board*, Naval Post-Graduate School Draft Sonar harware user manual, 27 February 1992.

Yourdon E., *The Practical Guide to Structured System Design*, Prentice-Hall Inc., New Jersey, 1980.

Yourdon E., *Modern Structured Analysis*, Prentice-Hall Inc., New Jersy, 1989.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Dr. Yataka Kanayama, Code CS/Ka Computer Science Departmant Naval Postgraduate School Monterey, CA 93943	2
Dr Timothy Shimeall, Code CS/Sm Computer Science Departmant Naval Postgraduate School Monterey, CA 93943	2
CPT Randall C Scott 4118 Melrose Drive Martinez, Georgia 30907	5