To The Graduate Council:

I am submitting herewith a thesis written by Anjanjyoti Das entitled "Evolution of Structured and Robust Code for Telerobotic Control of a Seven-degree-of-freedom Manipulator." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mechanical Engineering.

Rajiv V. Dubey, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council

Associate Vice Chancellor
and Dean of the Graduate School

EVOLUTION OF STRUCTURED AND ROBUST CODE FOR TELEROBOTIC CONTROL OF A SEVEN-DEGREE-OF-FREEDOM MANIPULATOR

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Anjanjyoti Das

August 1995

ACKNOWLEDGMENTS

I would like to thank Dr. Rajiv V. Dubey who is the major professor of this thesis. Without his constant guidance it would have been impossible to pursue this research. Most of the ideas presented in this thesis came out of discussions I had with him. I would also like to thank Dr. William Hamel and Dr. Micah Beck who agreed to be members in my committee. My sincere thanks are due to Mr. Tan Fung Chan for his helpful suggestions throughout the entire research work.

ABSTRACT

An improved, structured, and modular software for simulation and real-time control of a seven-degree-of-freedom telerobotic manipulator is presented. The software is reorganized by reducing the high fan-out at the top level. The parts connected with real-time control of the manipulator are carefully structured so that the execution time of these portions of the software is unaffected. The portions that are not directly connected with real-time control are thoroughly restructured. Information hiding is built up throughout the modules.

Inadvertent errors on the part of the programmer may cause operational problems in real-time control of the manipulator. The module and function interfaces are therefore made more robust to avoid problems of this type. The evolved structure is simpler to understand, program, and maintain. It is also easy to test and port to a different environment.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE **PAGE**

CHAPTER 1

INTRODUCTION

The major effort of this project is directed towards the development of maintainable and efficient software for the simulation and real-time control of the Robotic Research Corporation's (RRC) k2107 manipulator. The concepts discussed here can be applied in general to any robot control software being developed.

To meet the desired goals in a real-time application, the most important aspect of the software is its quality. Quality is linked to meeting user needs consistently under different conditions. These needs motivate specific quality attributes for any software product. It turns out that most software products share a core set of quality attributes that includes the following: correctness, efficiency, maintainability, portability, readability, reliability, reusability, robustness, and easy testability.

The above quality attributes have complex inter-relationships. For example, highly optimized code is often concise but not very readable. Less readable code is usually less maintainable. Conversely, optimized code is usually more efficient. Programmers often are able to increase certain quality attributes of their programs,

though usually at the expense of others. Thus it has to be decided which quality attributes are most important for a particular product.

1.1 Thesis Objectives

The objective of this thesis is to develop structured and modular design of the code for simulation and real-time control of the seven degree-of freedom Robotic Research Corporation's (RRC) manipulator without compromising the speed of the software. The requirement is to exercise extreme caution in selecting parts of the software where structure can be imposed without compromising the speed in real-time control of the manipulator. Also, care should be taken so that the overall appearance of the software still looks familiar to the old programmers who are used to the older version of the software. At the same time, the program has to be made more readable to new programmers who can start concentrating on certain parts of the software without the need to try to understand the entire software. Due to the real-time nature of the project, evolved code should be free from error-prone constructs and thus should promote robustness. The interfaces should be consistent so that correct and consistent data is passed between them. Due to the big size of the software mentioned above, avenues should be explored to reduce the effective size of the software by creation of a library of routines from where programmers can call the subroutines without having to worry about the

implementation details. In the final analysis, the software should become better structured, and easy to use by new programmers for increased maintainability and improvement. It needs to be free from errors that might occur due to the inadvertent mistakes of the programmer, thus resulting in undesired and potentially dangerous trajectories during real-time control of the manipulator.

1.2 Thesis Organization

The thesis is organized in the following structure:

- Chapter 2 discusses a few of the software design principles that should guide the process and development of any software product.
- Chapter 3 discusses the steps towards modularizing of the software. The chapter makes a study of the older version of the software and discusses the avenues where it can be made more structured without sacrificing the speed of the software in real-time control of the manipulator.
- Chapter 4 summarizes the corrective measures taken to develop information hiding in different modules of the software.
- Chapter 5 discusses how the software can be made more robust to eliminate the chances of inadvertent errors on the part of the programmer. It emphasizes the need to maintain correct interfaces to promote robustness.

- Chapter 6 looks at the new software from the angle of ease of testing and porting.
- Finally, conclusions are drawn in chapter 7.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

The software for the simulation and real-time control of the seven degree of freedom Robotic Research corporation's manipulator is written in C language to run on a Silicon Graphics workstation on IRIX 5.2 operating system. The main objectives of the software are to :

1. Display the manipulator on the Silicon Graphics workstation.
2. Provide commands to the manipulator in real-time control through the Bit3 adapter.
3. Allow study of different performance criteria in manipulation of the arm.
4. Allow use of the Kraft hand controller to give commands to the slave.
5. Allow use of different types of sensors like force, ultra-sonic and proximity.

The telerobotic manipulator shown in Figure 2.1 is Robotic Research Corporation (RRC) K2107 seven-degree-of-freedom redundant arm. This is a dexterous manipulator capable of running both in robotic and telerobotic mode. The use of the sensors like the ultra-sonic sensor, the proximity sensor, and the force sensors, make the arm much more versatile in exploiting its kinematic redundancy.

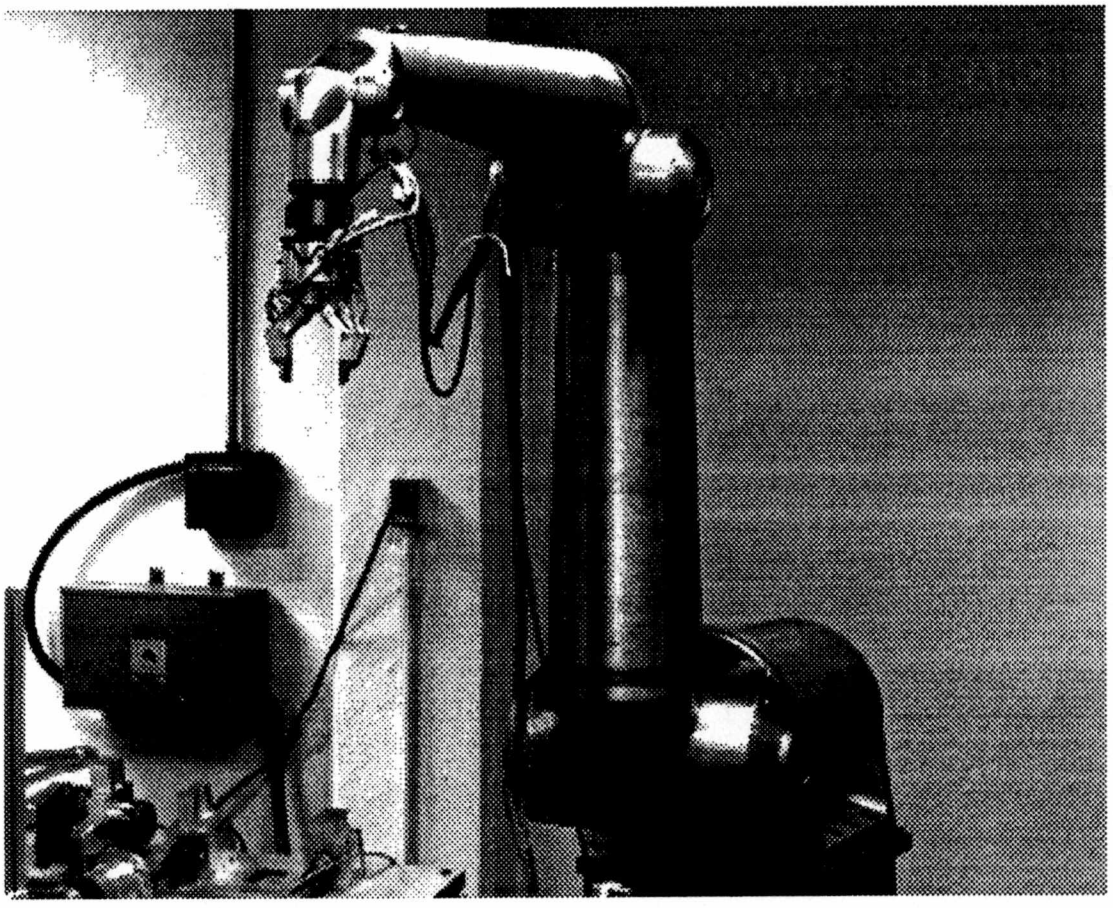


Figure 2.1 Seven dof RRC K2107 Manipulator

The communication between the software running on the Silicon Graphics workstation and the RRC controller is through a bus-to-bus Bit3 corporation adapter using memory mapping. The communication between the master and the slave programs is through system V shared memory model. The Kraft hand controller, shown in figure 2.2, provides the input to the master program which passes on to the slave program through the use of shared memory. The slave program outputs the required joint commands, and these are passed on through memory mapping through the Bit3 adapter to the RRC controller. Finally, these commands are translated to the motion of the telerobotic arm mentioned above.

The software, written in C, is an efficient one for use in real-time control. Despite its efficiency, the software has a few areas in which it can be improved to make it more robust. There is a high fan-out at the top level of the software and this has affected the development of abstraction in it. The lack of information hiding has made the software difficult to extend and maintain. There are a large number of global variables which are currently accessible from all the modules. All the functions are accessible from all the modules in the software, and this has made the public interface of the software quite large. This public interface can be reduced considerably by making many global variables local or semi-global, and many functions static. Upon study it was found that some of the interfaces were not consistent, there was immense scope for inadvertent errors that might go

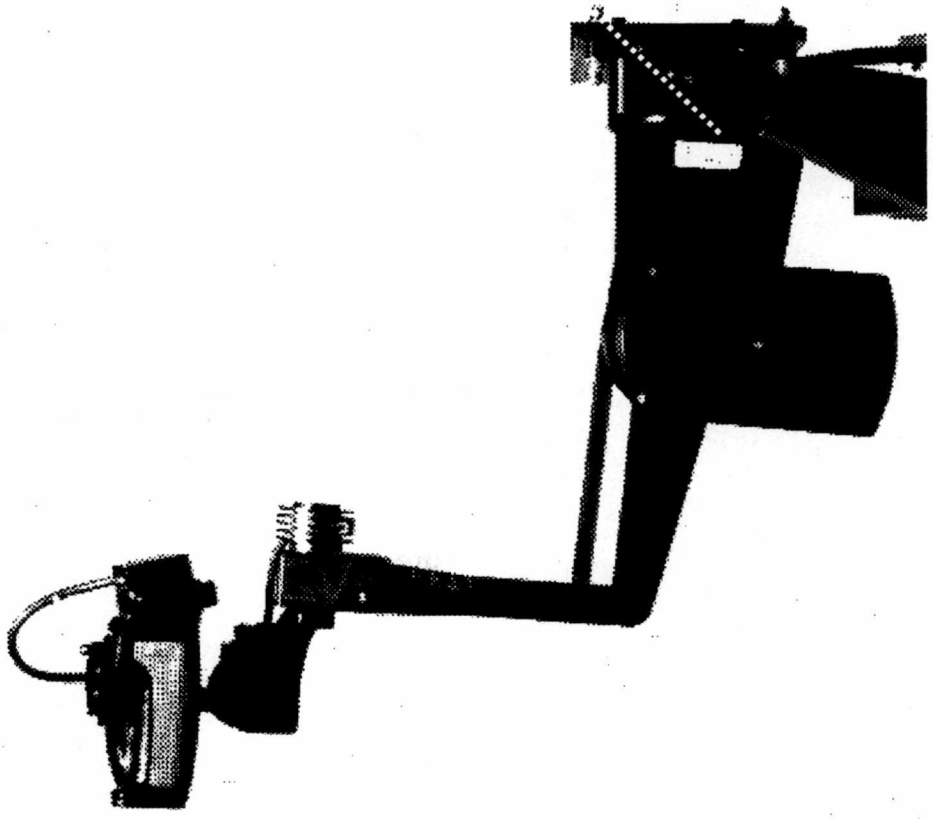


Figure 2.2 The Kraft Hand Controller

unnoticed. Because of lack of development of abstraction, the software was not easy to test and port.

In this chapter, we look at a few of the software design principles that should guide our design process for restructuring the software.

2.1 Software Design Principles

The design phase of a software focuses on how the system requirements are met. The structure and processing details of the software are conceived, documented, and reviewed during the design phase. Below we discuss a few important concepts and principles such as abstraction, modularization, cohesion, coupling, and information hiding that guide the design process.

2.1.1 Abstraction

When some properties of objects, events, or situations are suppressed in favor of others, the phenomenon is referred to as abstraction. The more the properties are suppressed, the higher the level of abstraction. In software engineering, abstraction usually involves the suppression of details in favor of general properties. This simplifies analysis by ignoring irrelevant details.

For the purpose of analysis and design, abstraction stands out as a powerful technique. It is imperative that levels of abstraction are carefully and clearly distinguished. All details not appropriate to a particular level of abstraction need to be strictly suppressed[1].

In the software at hand, the levels of abstraction were not well defined. The main() function handled too many details that were not appropriate for that level. As a result the software was becoming less maintainable and the readability was low. It was found that there was immense scope to build up levels of abstraction by which the software can be made easier. A great deal of irrelevant details at higher level could be brought down without affecting the speed of the software in real-time manipulation of the manipulator.

2.1.2 Modularity

For the intellectual and technical manageability of the resulting parts as single units, subsystems should be repeatedly decomposed. A module is any part or subsystem of a large system.

A modular system is one which is composed of well-defined, conceptually simple, and independent parts interacting through well-defined interfaces. The desirability of modular systems is enhanced because of the following reasons[3]:

- Modular systems can be understood and explained easily because they can be approached a piece at a time, and because of their well-defined inter-relationships.
- They are easier to understand and explain, hence easier to document.
- Programming a modular system is much easier because different groups can work on separate modules with little communication.
- They are easier to test because they can be tested separately, and integrated and tested together, one module at a time.
- Truly independent modules are easier to maintain. Changes can be made to a few modules without disturbing the rest of the system.

Good modularity is based on the principle of decomposing systems into parts. These principles, called modularization criteria, are as follows[1].

- Modules should hide implementation decisions that may be subject to change. This is the principle of information hiding.
- Different modules should implement single logically independent tasks.
- Coupling between the modules should be minimized and module cohesion should be maximized.
- Modules should reflect the levels of abstraction in the system design.

The first few of these principles are the most important. The cardinal rule of modularization is "It is far more important to decompose the system so that it is

simple, understandable, and has independent parts, than it is to decompose it so that it is efficient or easy to code."[3]

It was found that while building the levels of abstraction, the software can be decomposed into subsystems to make them more manageable. The software was not very easy to be programmed by a number of people working at the same time. It was found that the system can be decomposed into modules and sub modules.

2.1.3 Cohesion

This refers to the internal glue with which a module is constructed. The more cohesive a module, the more related are the internal parts of the module to each other and to the functionality of the module. Levels of cohesion[1][3] ranked from highest (best) to lowest (worst), as depicted in Figure 2.3 are as follows:

- Data-type cohesion: Modules that implement abstract data types have data-type cohesion.
- Functional cohesion: If a module contains sub programs and data structures that collectively implement a single function, feature, or operation, it is said to have functional cohesion.
- Logical cohesion: A module containing elements that form a set of logically related tasks is said to have logical cohesion.

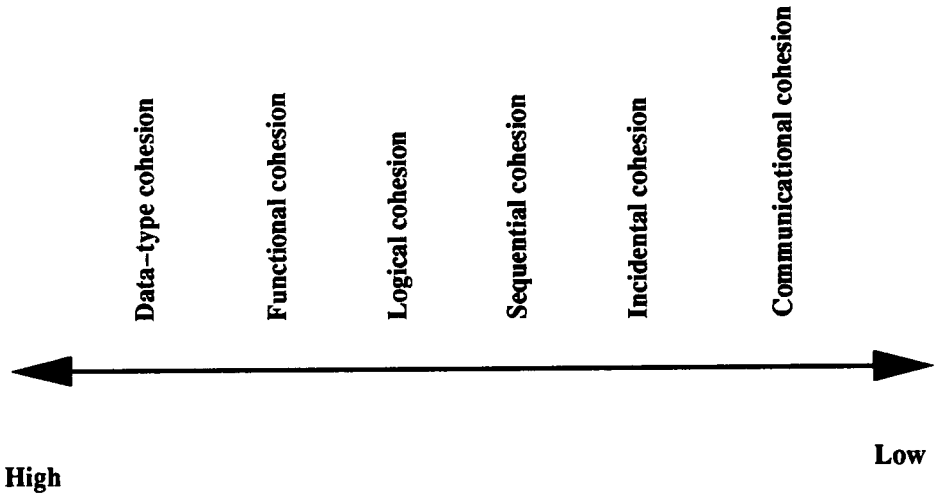


Figure 2.3 Levels of Cohesion

- Sequential cohesion: A module containing elements that are executed in sequence has sequence cohesion.
- Incidental cohesion: The absence of any significant relationship between the elements of a module leads to incidental cohesion.
- Communicational cohesion: We can associate certain functions because they operate on the same data set. Modules constructed in this way are communicationally cohesive. However communicational cohesion often destroys the modularity and functional independence of the design.

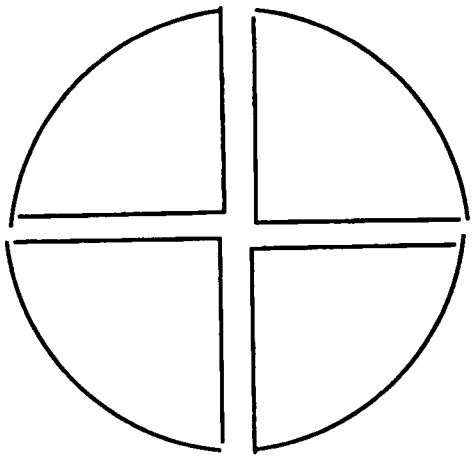
The notion of cohesion can be easily achieved in an object oriented design. This is because, the compositional process forces the actions to be placed with the objects they affect. In such a design all the cohesive data and members are put inside the class of the object. This makes the class a cohesive unit which can be maintained and reused easily. In a programming environment like C that does not support object oriented paradigm, programmer has to be extra careful in building up the desired level of cohesion.

The software for the simulation and real-time control of the RRC manipulator was studied to identify the levels of cohesion. It was found that there was scope to improve the cohesion of the software. There was scope to remove incidental cohesion. Functional and logical cohesion could be built up to make the software more maintainable.

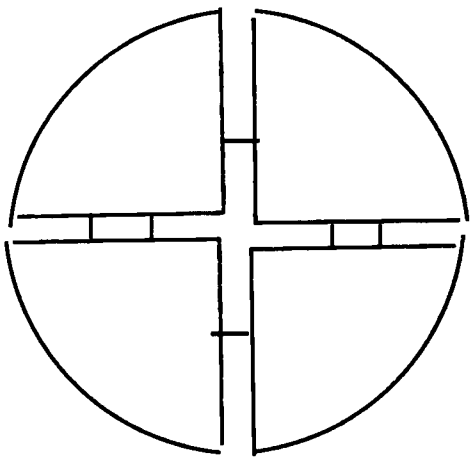
2.1.4 Coupling

Coupling is a measure of how much modules depend on each other. Figure 2.4 shows levels of dependency between modules. Two modules are highly coupled if there is a great deal of dependence between them. Loosely coupled modules have some dependence, but the interconnections among modules are weak. Uncoupled modules have no interconnections at all. In general, lower-level coupling is better. Loosely coupled modules are easily understood, easily documented, coded, tested, and maintained. Five levels of coupling, listed subsequently from lowest (best) to highest (worst) are as shown in Figure 2.5:

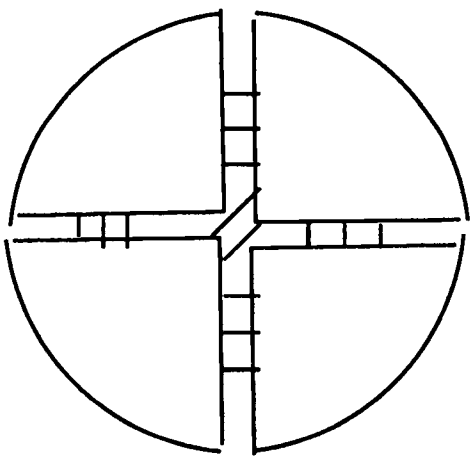
- Data definition coupling: This arises from the manipulation of data of the same type, by the modules.
- Data element coupling: In this case, modules pass data elements through a disciplined interface, like a parameter list. Modules coupled only through data elements may be changed independently as long as their interface is unchanged.
- Control coupling: It is possible that a module has the ability to influence the flow of control within another module by passing control information as flags. This form of coupling, referred to as control coupling disperses decision-making code between modules. What results is the interdependency of the modules.



Uncoupled
No dependencies



Loosely Coupled
Some dependencies but not too many



Highly Coupled
Many dependencies

Figure 2.4 : Modules – levels of dependency

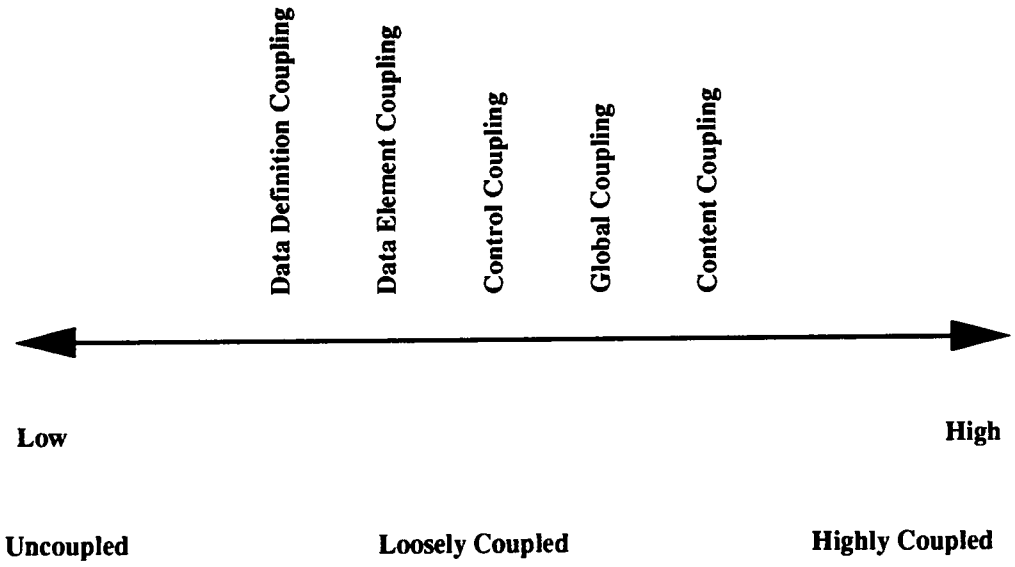


Figure 2.5 : Types of Coupling

- Common coupling: Common coupling is displayed when modules reference common global variables. The interface between modules that share global data may be undisciplined, and this brand of coupling has the tendency to increase program complexity enormously. Figure 2.6 explains common or global coupling. The figure shows the global data members at the top and the modules that access them at the bottom. As can be clearly seen, the global data can be easily manipulated by any module.
- Content coupling: A module may alter data or control local to another module, leading to the occurrence of content coupling. Content coupling occurs in C, and this happens when the supposedly private (i.e., static) data, local to another module are accessed through the medium of pointers.

The occurrence of content coupling should be prevented while global coupling should be minimized and controlled as much as possible through the judicious use of scope rules. The level of coupling that can usually be avoided is control coupling. The inter module communication mechanisms that are generally preferred are data definition and data element coupling.

The modules of the software had a very high level of coupling. Thus there was a great deal of dependence between the modules. There were great amount of

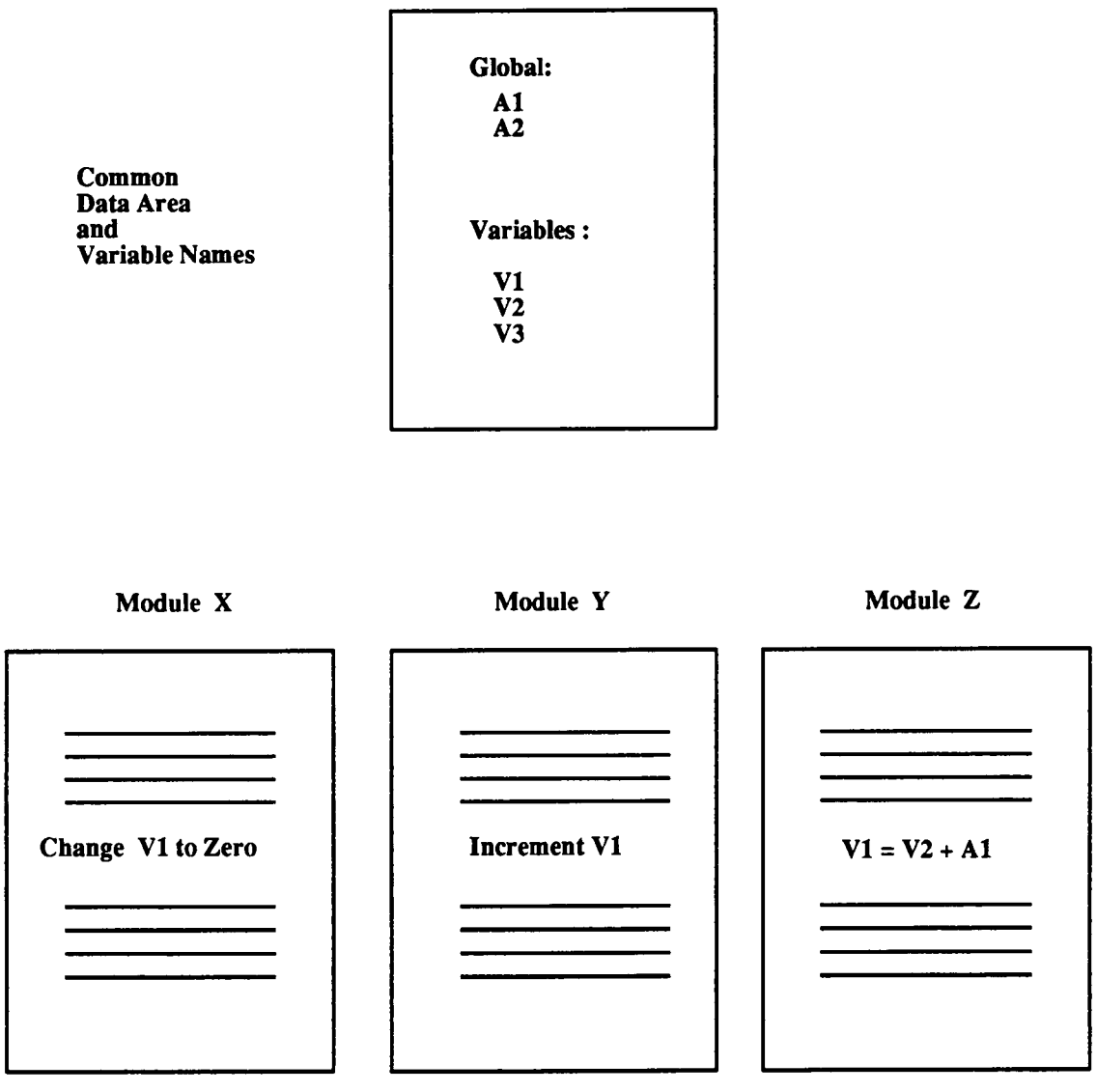


Figure 2.6 : Common Coupling

common or global coupling. It was found that these can be greatly reduced to make the modules less dependent on each other.

2.1.5 Information Hiding

Information hiding stands out as a powerful principle encouraging other good modularization practices. In this decomposition strategy, each module shields the internal details of its processing from all other modules.[2][5] Well-defined interfaces are required if communication is to take place between modules. Module information can be categorized as private and public information. Private information, also referred to as a module's "secrets" includes the details of a module's internal data representation, algorithmic operations, internal organization, and the data objects and operations it requires from other modules. Specified in the module's public interface is the information necessary to make use of a module's services. The purposes of interfaces are to specify the names, properties, and parameters of the data objects and operations provided by the module, their behavior, and their error conditions.

Information hiding is not restricted to just one particular level in a software system, but takes place at several levels. The principle of information hiding applied at the block level, (e.g., in C all code between curly brackets) and at the compile module level, emphasizes the maximum possible locality of objects. This

principle is manifested at the compile module level[4] when private (in C static) functions and data objects are packaged in a compile module.

The advantages of information hiding need to be outlined. The fact that it is a species of abstraction is one of its merits. A module's secrets are details suppressed in the specification of a module's interface. Information hiding therefore has the potential to reduce complexity. Second, high cohesiveness and loose coupling are the desirable qualities that emerge from modules that thoroughly hide their internal details. Finally, independence from other modules is obtained when design and implementation details are hidden inside modules. This feature is of prime importance when implementation details must be changed. Easy and reliable changing of secrets is facilitated because the effects of such changes are restricted to the modules where they are hidden.

These advantages of information hiding lend currency to the belief of experts that information hiding is the fundamental principle of modularization[5].

2.1.6 Control Issues

The issue of control of one module by another is an important one for functionally decomposed designs. Once we have a design of functionally cohesive modules with a lower degree of coupling, the next requirement to examine the quality of design are the control issues.

Fan-in/Fan-out: Fan-in is the number of modules controlling a particular module, and Fan-out is the number of modules controlled by a module. The better design is one in which we try to minimize the number of modules with high fan-out. A module controlling many other modules usually indicates that the controlling module is doing too much; the controlling module is probably performing more than one function. On the other hand, as we increase the number of levels in the design, we sometimes want to use a particular module more than once. This type of modules are called utility routines. A typical utility module has high fan-in because it is invoked by many other modules. A good design is one which has high fan-in and low fan-out.

There was a great scope to improve the older version of the software by building up levels judiciously by reducing the fan out at the top and other levels.

Scope of Control and Effect: This aspect of a good design is also very important for correct development and maintenance of a correct system design. It is important to ensure that the modules in the design do not effect other modules over which they have no control.. For a given module, the set of modules to which arrows are drawn from it is called the scope of control of the module. The modules controlled by the given module are collectively referred to as the scope of effect. No modules should be in the scope of effect if it is not in the scope of control. If the scope of the effect of a module is wider than the scope of its control, it is

almost impossible to guarantee that a change to the module will not destroy the entire design.

From the above discussion the characteristics of a good system design that evolves are[1]:

- Low coupling of modules.
- High cohesiveness of modules.
- Minimal number of modules with high fan-out.
- Scope of effect of a module limited to its scope of control.

It is useful to know these characteristics before one starts the design of a system so that these can be built into the system design.

CHAPTER 3

STEPS TOWARDS A MODULAR SOFTWARE

The software that we have at hand is of considerably big size. Therefore there is a necessity to develop it in a modular fashion. Programming in a modular fashion not only makes the software more readable and easy to work with, but also promotes error free programming and makes it easier to test and port to a different hardware. Thus development of a modular program is of utmost importance. Here we are working with an existing software. Hence, we have to maintain the continuity while making the changes.

3.1 Scheme for Numbering the Subroutines

A consistent scheme for numbering the different subroutines was developed to explain the software. The numbering scheme is started from A which is the main() function. The other subroutines are numbered depending on how the subroutines are called. A subroutine can only call another routine which is below it. For example, a subroutine D07 can call lower level routines like E02, F01, H03, I07, L29 etc., but it cannot call C05 or B02. The entire software was marked according to this scheme and we use this scheme to explain different portions of

the software. The names of the subroutines along with their respective schematic representations are presented in Appendix E.

3.2 Restructuring the Program to Make it Modular

In this section we explore the possibilities of restructuring the software to make it more modular. In doing so we make a thorough study of the software at hand and search for design heuristics to guide our restructuring process. Different people have different views about the most desirable structure to be developed in a software. Even after taking consideration of these opposing views, a consensus still emerges as an outline of a desirable structure. We base our restructuring on such a design heuristics that exhibits the characteristics explained below.

3.2.1 Characteristics of a Desirable Structure

The main() function of the original software was a function with a very high fan-out. The main() function handled too many details and as a result the software had become less readable. This could create a maintenance problem at a later stage. In the older version, all the lower level routines in the software were being called from the main() function with a view to increase the speed of the software in real-time control of the RRC manipulator. Also the main() function handled many

detailed portions of the code which could have been easily brought down to a lower level to make it more structured. From the point of view of controlling the manipulator, this served the purpose as there was not much overhead involved in calling different subroutines from the topmost level, i.e., the main() function. But this was against the spirit of modular programming. This had highly affected the readability of the software. The chances of committing mistake was also high in this model of structure. To design a good software, attempts should be made to minimize structures with high fan-out[5]. Figure 3.1 shows two different types of structures. The first structure, Figure 3.1(a) should be avoided whereas one should strive for the second structure, Figure 3.1(b)[5]. The first structure does not make effective use of factoring. A more reasonable structure is shown in the second figure. This structure takes an oval shape, indicating a number of layers of control with the highly utilitarian modules at the lower levels[5].

3.2.2 Classification of the Aspects Controlled by main()

With a view to modularize the software, a thorough reading of the software was made. Here the consideration was to evolve a structure that would promote modularity without sacrificing the speed which is extremely important in real-time software like this. In the original software it was found that the main() function was basically controlling the following seven aspects of the software :

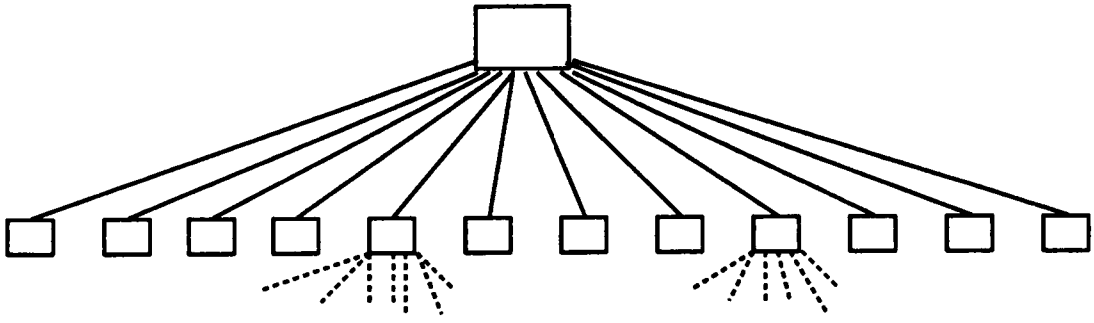


Figure 3.1(a) : Avoidable structure

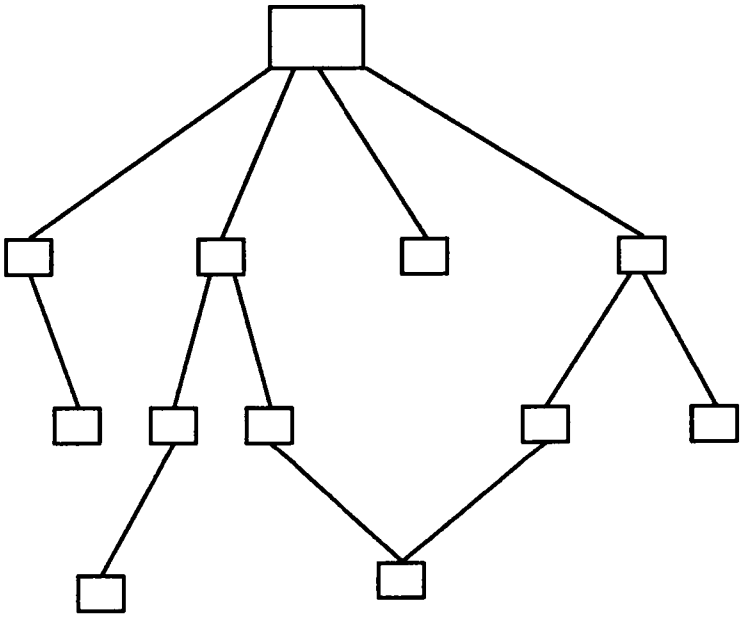


Figure 3.1(b) : Structure to be strived for

1. Moving the RRC manipulator through the use of inverse and forward kinematics.
2. Moving the manipulator between pre defined points through trajectory planning. This also includes recording the position and orientation of the manipulator at different points.
3. Allowing use of the master hand controller to give commands to the slave through the slave program using shared memory.
4. Allowing manipulation of different parameters to test different performance criteria.
5. Changing the view and angle of the camera in the graphics mode to view the manipulator from different angles.
6. Writing different experimental information to the files for latter use and
7. Displaying the manipulator on the SiliconGraphics workstation.

On close examination it was found that all the above segments were not connected with speed of manipulation in the real-time control of the robot, thus structure can be imposed by reducing the fan-out in these areas. The areas which are not connected with speed of manipulation in real-time control are 4, 5, 6. It was found that even the speed intensive components can be made more structured without sacrificing any speed. The skeleton of the evolved new structure is shown in Figure 3.2. A close look at this figure will show that imposing structure to the

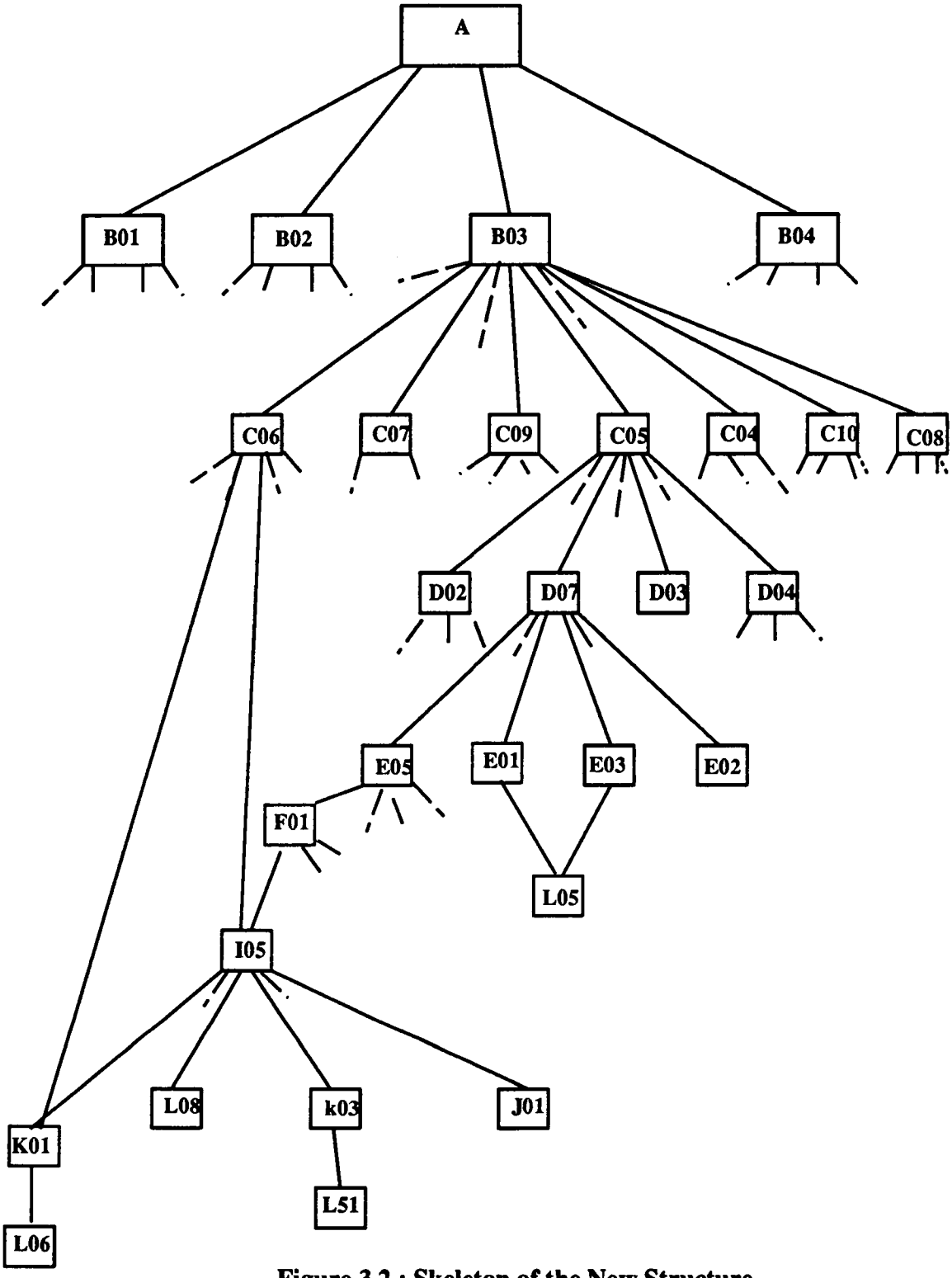


Figure 3.2 : Skeleton of the New Structure

software in this manner does not sacrifice the speed affecting real-time control. The number of function calls does not increase any further though fan-out is reduced at the upper layer by increasing the depth of the software. It is clearly visible that the effective depth of the sub trees of the speed intensive portions of the software is still unchanged.

3.3 Examination of the New Structure vis-a-vis the Old One and Study of Relative Efficiency

Upon close examination of the software it was found that there was sufficient scope to build up levels of abstraction without sacrificing any speed. It was also found that different parts of the software can be made more structured by reducing the fan-out and increasing layers of depth.

3.3.1 Development of Levels of Abstraction

In the older version of the software the main() function is a big routine which takes care of varied tasks to perform different sub tasks directly. The motivation behind keeping all the tasks to the main() itself is the execution speed. But a close examination reveals that this can be easily modified to make it more structured. The main() function actually does three distinct tasks at the highest level of abstraction:

1. initializes the various parameters connected with the manipulator,
2. initializes the shared memory to be used for real-time control of the robot.
3. waits for user input to perform different tasks, depending on the type of input.

The program in fact spends the entire time in the third phase except for a small initialization phase which encompasses phase 1 and 2. The third phase is an infinite *while* loop which is terminated when the user clicks at the END button. Thus the large fan-out of the software at the topmost level can be drastically reduced without sacrificing any speed. With this point in view, the main() function was rewritten in the new version, by incorporating four subroutines viz., init_robot(), init_bit3(), check_and_process_input() and end_program(). Since the program spends majority of its time in the check_and_process_input() subroutine, the effective depth of the sub trees to evaluate function overhead cost should be calculated from this subroutine only instead of main() as in the older version of the software. The new structure defined above, lays the groundwork for the structuring of the software on defined design heuristics[5]. It should be noted that though the restructuring appears to be based on the graphic display and the teach-pendant buttons, a close analysis will reveal that making the software structured

makes it suitable and more amenable to be driven by any form of input- mouse, keyboard, joystick or any other sensory input.

3.3.2 Study of Effective Depth of Different Sub Trees to Determine Function Overheads

It can be seen that in the old version of the software, the program goes into an infinite *while* loop after drawing the manipulator on the graphic monitor. It is this loop from where all the different lower level routines are called. This *while* loop also had lots of details which did not involve speed. All the lower level subroutines are called from this loop so as to minimize the overhead of function calls while calling these routines. Below we discuss the effective depth of a few sub trees in the two versions of the software to study their relative efficiencies.

3.3.2.1 teach_robot() Sub Tree

A close look at the software will however reveal that only the components connected with movement of the manipulator through inverse and forward kinematics of the manipulator are important here. These routines need to call the lower level routines again and again after taking the user input in order to move the manipulator through desired locations. Thus these routines need to be called

from the topmost level to minimize the overhead of repeated calling of lower level subroutines.

If we look at the figures 3.3 (a) and (b) it will be evident that the effective height of this component of the software remains unchanged. But at the same time taking the other components to lower levels make the functionality of those components more clear. This makes the components clearly identifiable and easy to manipulate rather than trying to locate those in the midst of other subroutines in the older version. Figure 3.3 (a) shows how the teach_robot() function was called from the main(), (A in the figure) in the old version of the software. Figure 3.3 (b) shows that the same part of the software was brought to the lower level by making it to be called from the check_and_process_input() function. The check_and_process_input() function is a *while* loop in which the program keeps looping after it goes through init_robot() and init_bit3() functions. Any user input the said portion of the software takes from the user, is obtained in this subroutine only. Thus it can be clearly inferred that calling the teach_robot() function from the lower level at check_and_process_input() function does not hamper the speed of the software at all. But the evolved result is a more structured design. This makes the software easier to understand and maintain. Any interested person can identify the teach_robot() component of the software very easily.

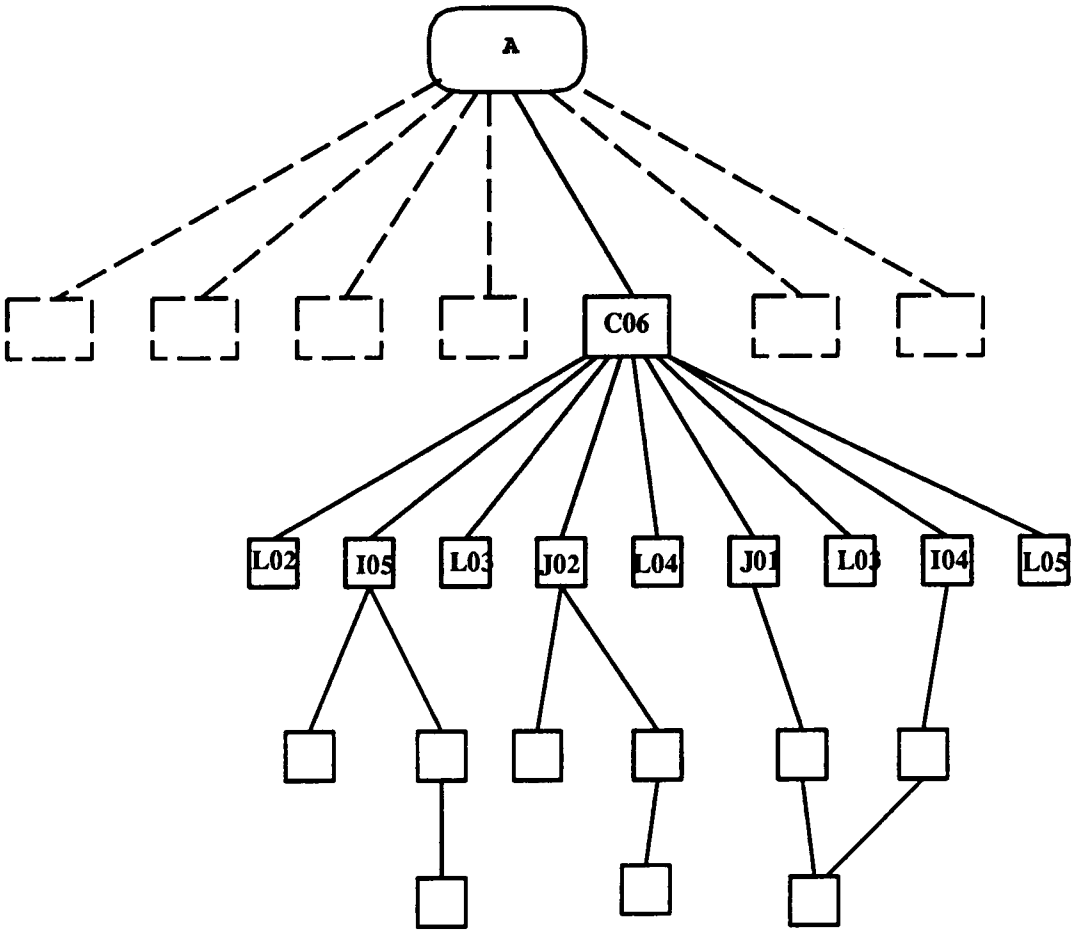


Figure 3.3(a) : Effective depth of teach_robot() in old structure

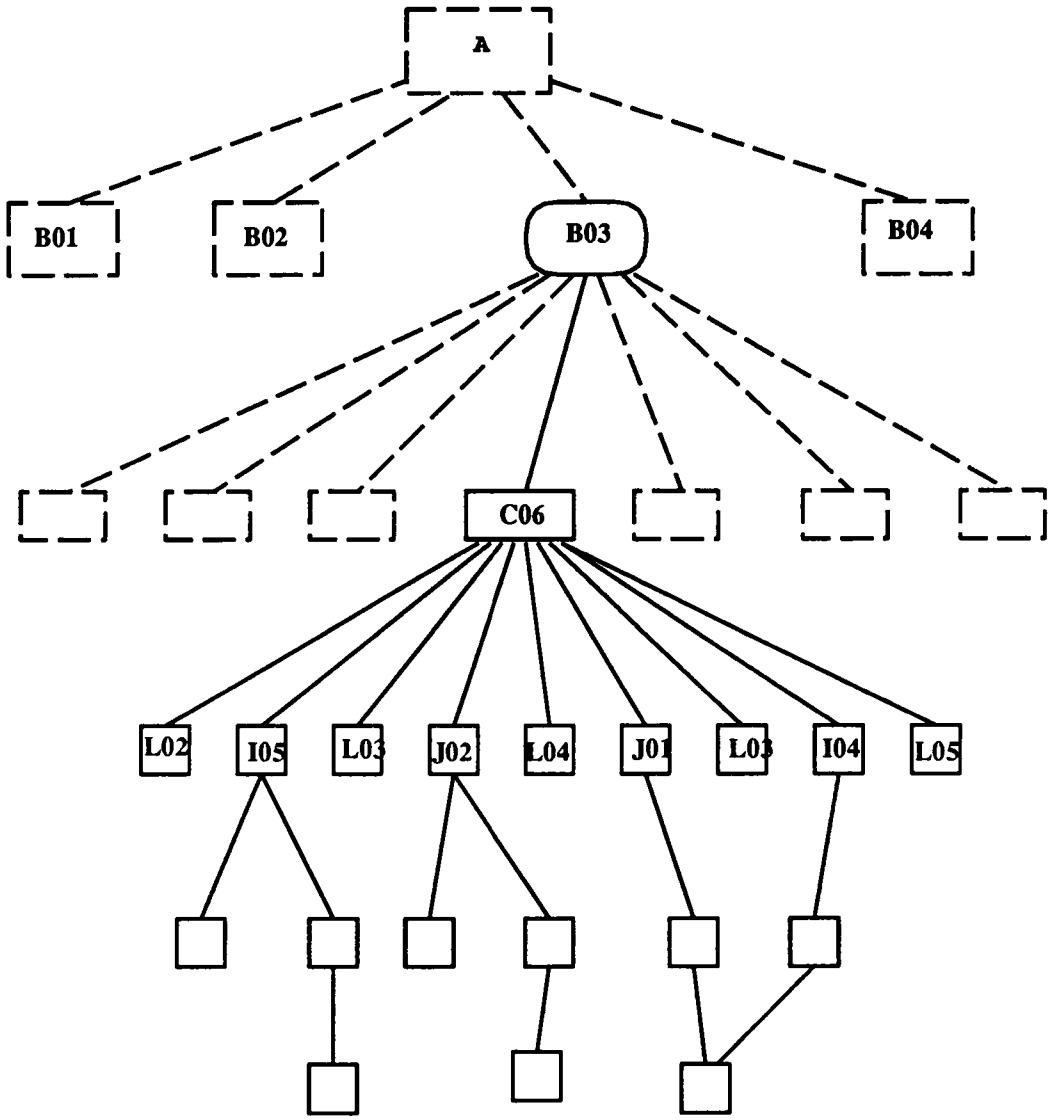


Figure 3.3(b) : Effective depth of teach_robot_cartesian() in new structure

3.3.2.2 Trajectory Planning Sub Tree

The next component which is connected with speed is the trajectory planning part. To activate this part the user has to click at the RUN button in the graphic teach-pendant. But before this, the user has to select the points between which the trajectory planning is to be done. To select the positions and orientations of the points, the manipulator has to be first moved to that location and then the position and orientation can be recorded using either the REC_CT or REC_ST buttons. To move the manipulator to the desired locations, the same part of the software as explained in section 3.3.2.1 is used. Thus speed is not affected here. Then to record the position and orientation of the manipulator, the REC_CT or REC_ST button is pressed. At this point the values connected with the position and orientation are recorded into the locations of the global variables to store these data. Thus it is evident that these components of the software viz, the parts connected with REC_CT or REC_ST buttons, are not connected with speed of real-time control and can be safely taken to a lower level to improve structure by reducing the fan-out at the top level.

Coming to the RUN button, as soon as the user clicks at this button, the control of the program passes to a *while* loop which moves the manipulator through the trajectory by calling the subroutines like goal_gp(), circular_path(), goal_jnt(), goal_obt(), goal_lin(), difference(), k_inverse(), linear_cont(),

goal_rpy() etc. These subroutines are called again and again as the manipulator moves from one point to the next infinitesimally till the interrupt button PAUSE is pressed. The speed is important in this part of the software. But at the same time one has to notice that all these routines are called from inside the *while* loop mentioned above. That means the effective depth of the tree that has to be considered for speed consideration, starts only from the point the user clicks at the RUN button, and after this the program does not take any more user input till the PAUSE button is pressed. Thus it is evident that the part of the software that lies beyond the point when the user clicks at the RUN button can safely be moved to a lower level to improve the structure of the software. Figures 3.4 (a) and (b) show how the effective depth of this portion of the software remains unchanged even after imposing structure to the software. Figure 3.4 (a) shows how this trajectory planning part of the software was called from the main() function itself (A in the figure). To facilitate moving of this portion of the software to a lower level, a new function run_robot() (D07) was written and was called from a lower level as shown in figure 3.4 (b). After initializations are performed, the software keeps looping in the check_and_process_input() function (B03 in figure). As soon as the user clicks at the RUN button, the control of the software passes to the point D07 through C05. D07 is the run_robot() function and this incorporates a *while* loop which simply calls the other routines mentioned above without any farther

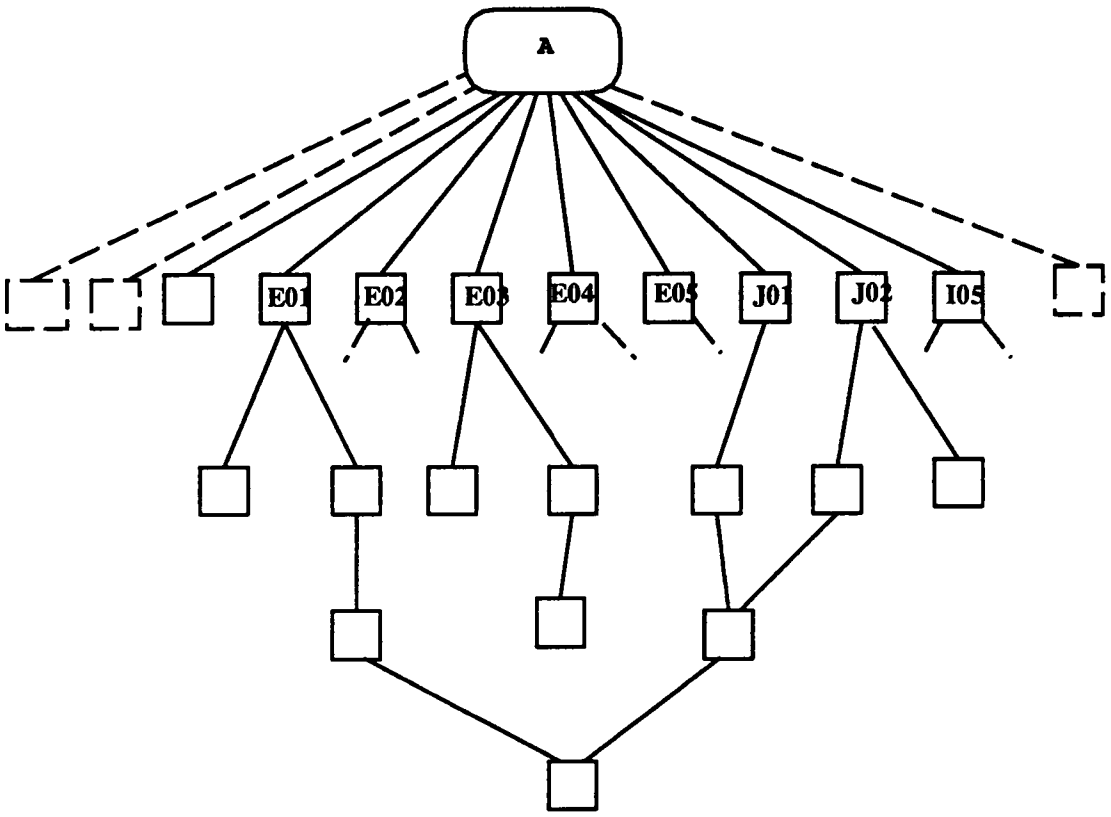


Figure 3.4(a) : Effective depth of trajectory planning unit in old structure

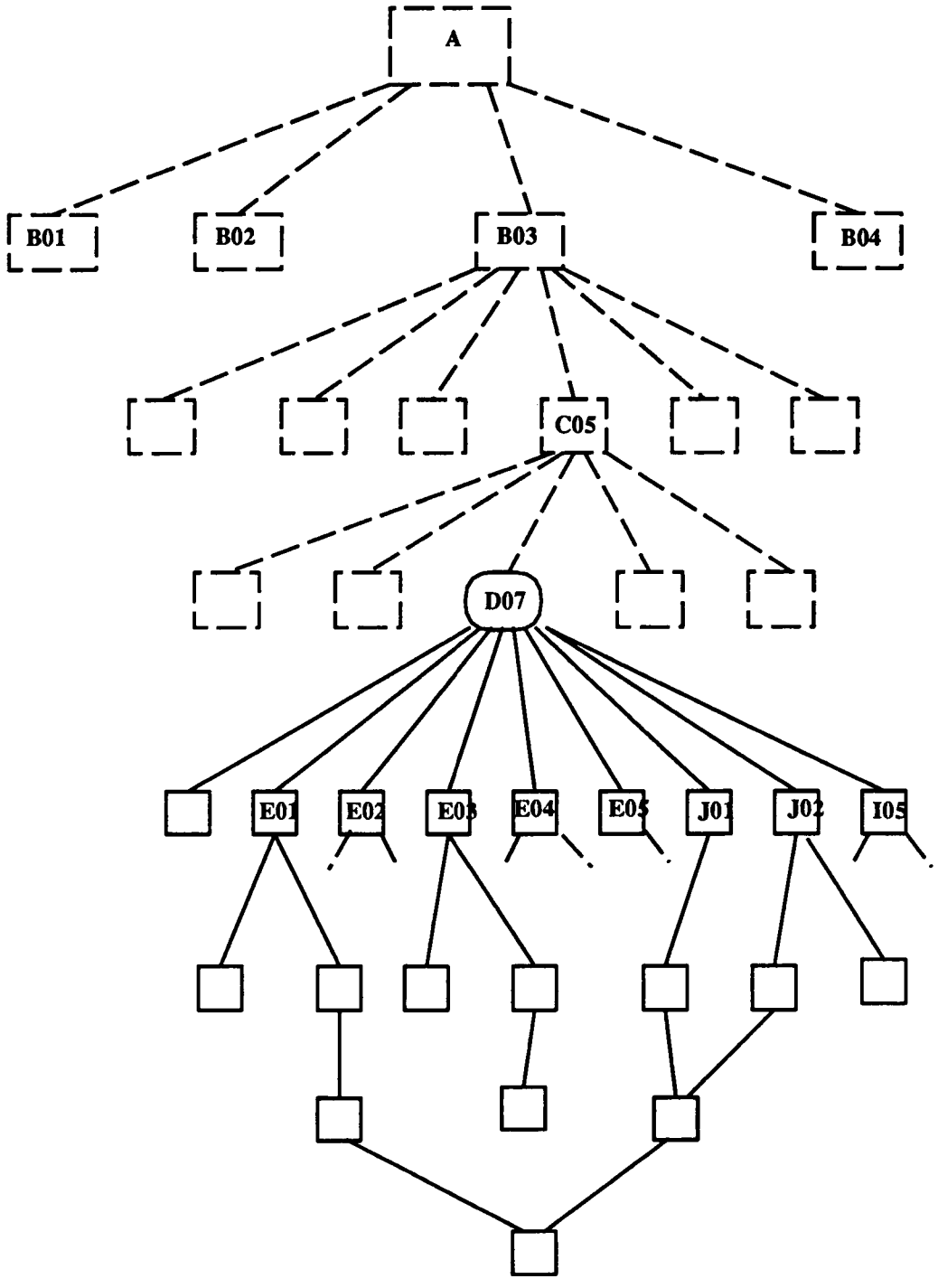


Figure 3.4(b) : Effective depth of trajectory planning unit in new structure

user input. The effective depth of the sub tree to calculate the function overhead should be calculated from this level i.e., D07. Therefore, it is clearly evident that the effective depth of the trajectory planning sub tree remains same in both cases, figures 3.4 (a) and (b). Thus the run time of the trajectory planning part of the software remains unchanged in the new version of the software. But at the same time, making it structured and making it clearly identified makes it easier to test and maintain.

The newly created teach_robot_cartesian() and teach_robot_joint() functions are called directly from the check_and_process_input() function to reduce the function overheads, and rest of the routines are divided into subgroups by grouping them within newly created subroutines poll_left_column(), poll_lower_middle_column(), poll_right_column() and poll_left_top_corner(). By looking at the teach pendant of the graphic display (figure 3.5) alone, an individual can easily identify the associated portions of the code in the software. This has been made easier by incorporation of the pre-processor macros in the subroutines check_and_process_input(), poll_left_column(), poll_lower_middle_column(), poll_right_column() etc. The naming convention adopted here facilitates the identification of the related code in the software. For example LEFT_COLUMN_TP refers to (1020<= mx) && (mx <= 1090)

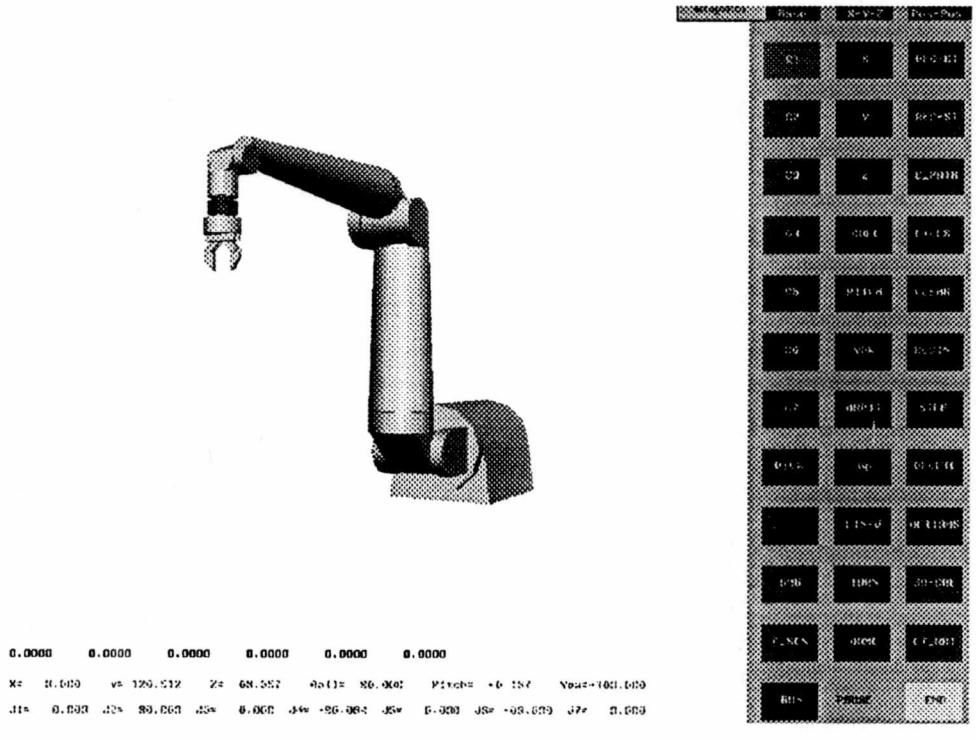


Figure 3.5 : Graphic display of the manipulator

3.3.3 Imposition of Structure on Functions

Below we discuss how structure was imposed on different sub trees to make these parts of the software more structured.

3.3.3.1 Functions Connected with Left Column of the Teach-pendant

All the buttons in the left column of the teach pendant are not connected with speed in the real-time control of the manipulator, except the RUN button. The RUN button has already been explained in section 3.3.2.2. The following newly created functions are called from inside poll_left_column() subroutine as shown in Figure 3.6. Each of these newly created functions performs one distinct task and thus promote simplicity and easy readability. The change_local_coord() function changes the coordinate system to be used by manipulation of the associated variables. The change_viewpoint() function changes the view of the camera by calling the appropriate routines. None of these routines are connected with speed of manipulation in the real-time control. The only subroutine which is connected with speed here is the run_robot() function. This is entered when the user clicks at the RUN button. But this function is itself an infinite *while* loop, and as explained in section 3.3.2.2 all the effective depth of the sub trees should be calculated from this point below. Thus it is immaterial whether this function is

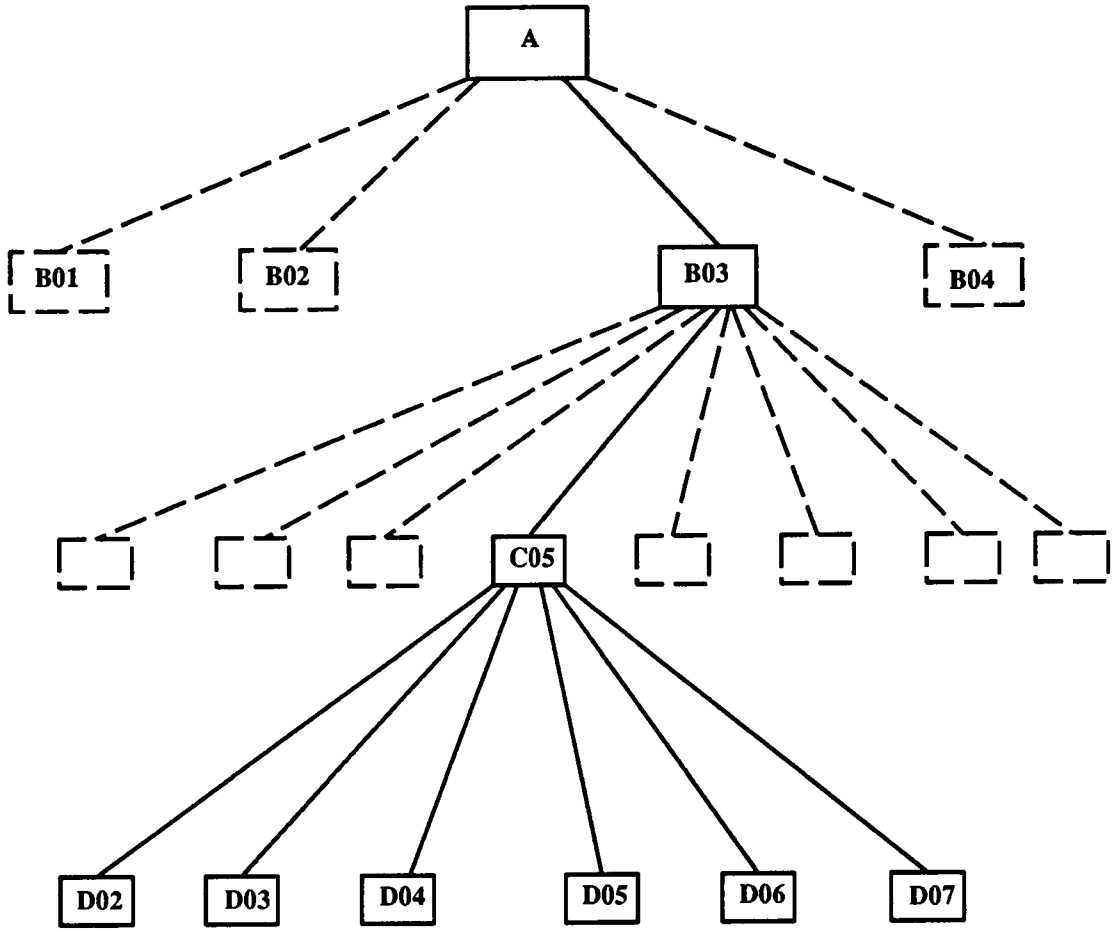


Figure 3.6 : Structure of poll_left_column()

called from the topmost level or from inside poll_left_column() subroutine. But calling it from inside poll_left_column() makes the software more structured and easily maintainable.

3.3.3.2 Functions Connected with Middle Column of the Teach-pendant

The newly created subroutines being called from poll_lower_middle_column() are choose_joint_or_cart(), set_velocity_limit(), turn(), bring_robot_home() as shown in figure 3.7. All these routines manipulate some or the other global variables to set different options. None of these functions are dependent on time and the difference in time spent is negligible in comparison with the total run time of the program. Thus from the point of view of time of execution, it is immaterial whether they are called from the topmost level or from inside poll_lower_middle_column() subroutine. Calling them from the later however greatly improves the structure of the program.

3.3.3.3 Functions Connected with Right Column of the Teach-pendant

Most of the newly created functions being called from poll_right_column() do not involve appreciable time factor and hence justifies their place at a lower layer in the poll_right_column() subroutine (refer to figure 3.8). The ask_option()

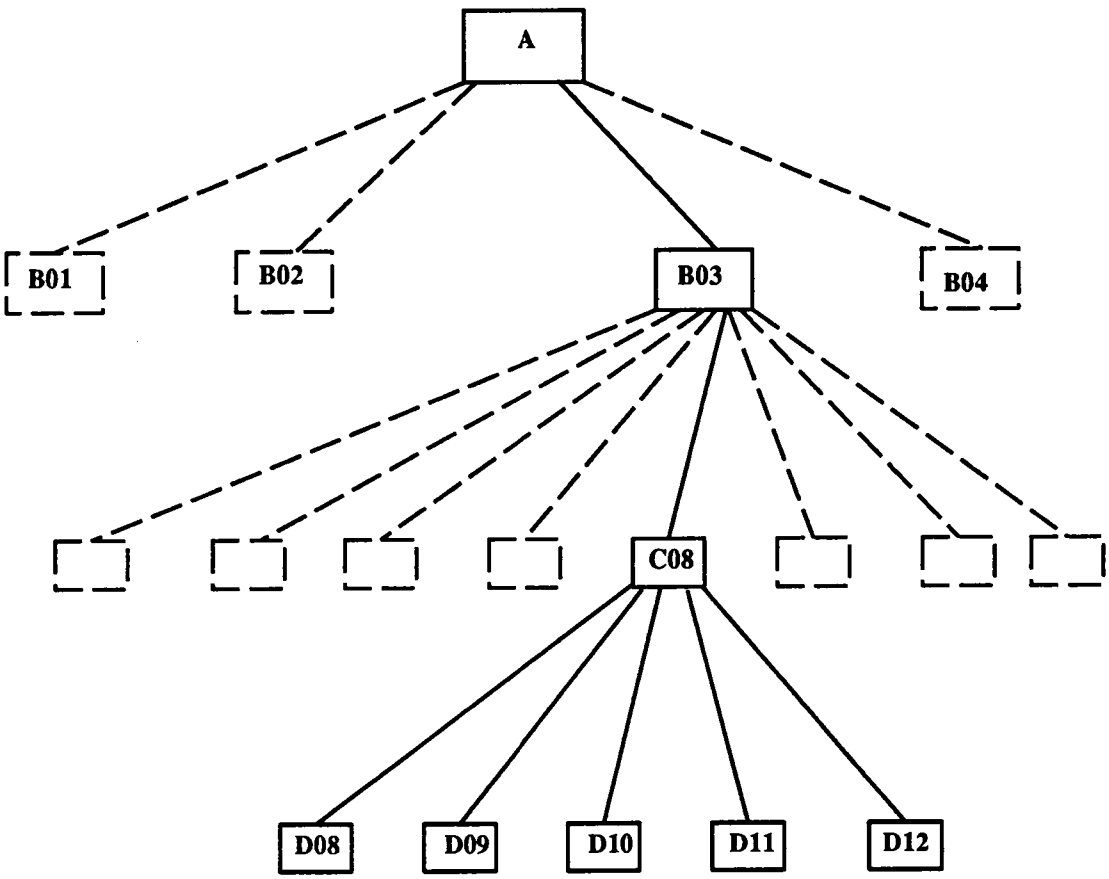


Figure 3.7 : Structure of poll_lower_middle_column()

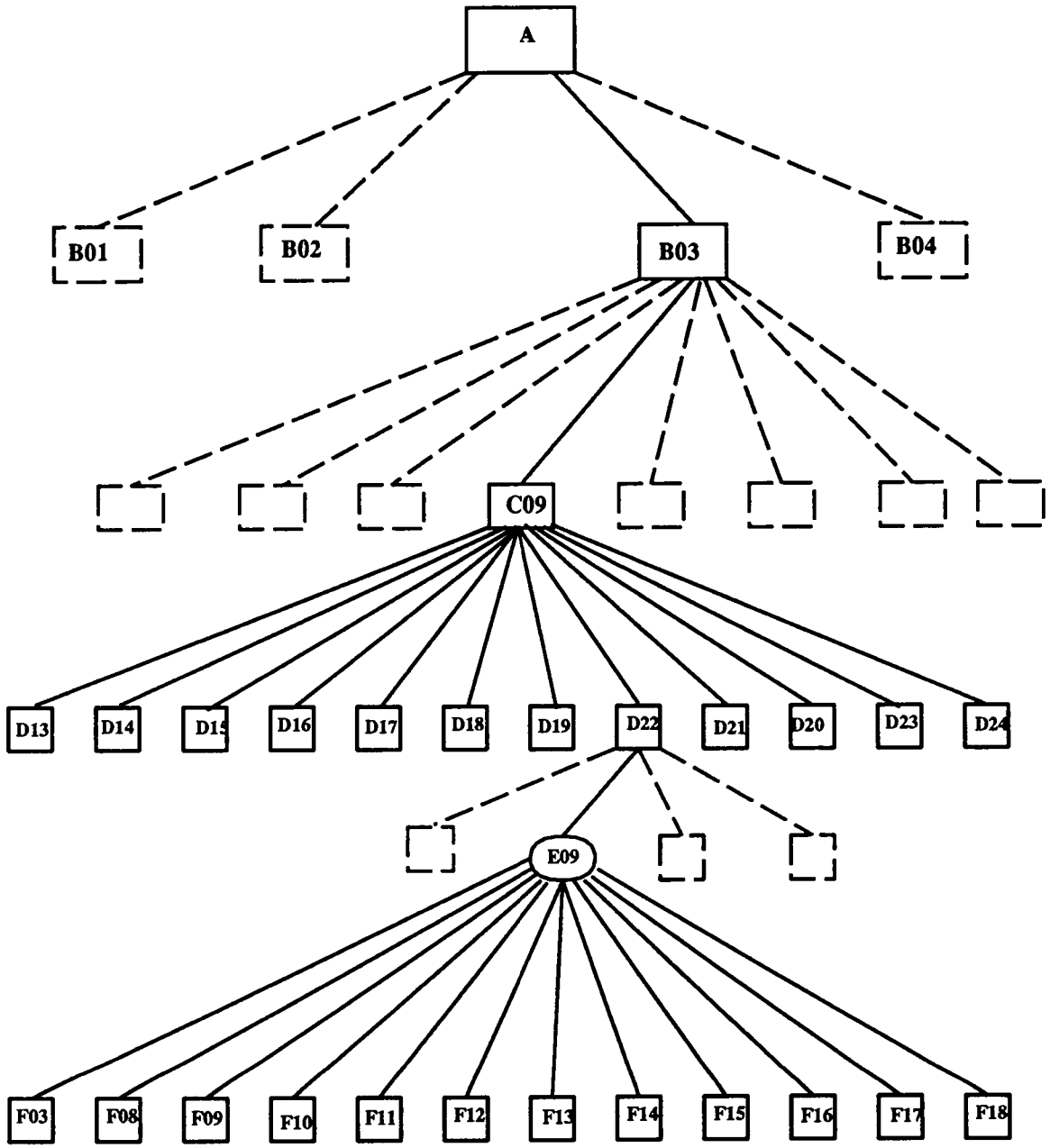


Figure 3.8 : Structure of poll_right_column() and available_options()

subroutine being called from inside poll_right_column(), leads the program to another *while* loop which allows the user to manipulate different global variables associated with different options. The *while* loop being mentioned is inside the available_options() subroutine. This subroutine was newly created as a driver to call the different options to manipulate different global variables. The newly created subroutines group the related global variables together and thus promote easy maintenance. A few of these routines are: change_gain_factor(), change_radius(), change_parameters(), select_control_scheme(), show_end_effector(), change_color(), repeat_trajectory(), select_manipulator() etc.

3.3.4 Separation of Cartesian and Joint Motion

At this point it was also observed that the logically different joint and Cartesian motion of the manipulator was implemented inside the same subroutine teach_robot(). Inside this subroutine, the program used to take different paths depending on the value of the variable *t_p_index*. The value of the variable was checked at a lower level inside the teach_robot() subroutine and accordingly motion was decided to be joint or Cartesian. Now to reduce complexity of the software, these logically different sub parts of joint and Cartesian motion have been written into two separate subroutines teach_robot_cartesian() and

teach_robot_joint(). This makes the parts connected with joint or Cartesian motion clearly identifiable. These two subroutines are very important parts of the software and keeping them as two different clearly identifiable entities make the software simple, easily readable and more manageable.

3.3.5 Restructuring the Master-controller Part of the Software

Another important part of the software is the real-time control of the manipulator using the Kraft hand controller. In the older version of the software, the branching for this portion of the software was from the main() function itself. In the newer version, the test whether the Kraft hand controller is on or not, is done from a subroutine check_switches() which is called from the looping subroutine check_and_process_input(). Figure 3.9 shows the location of the master hand controller software inside the general structure of the software. This makes the portion connected with the master hand controller clearly identifiable in the software. Whenever the master hand controller is off, this portion of the software does not perform any task.

A close look clearly reveals that making the software layered into multiple layers and keeping different portions of the software distinct, greatly facilitates the testing and integration process. Moreover it also does not slow down the software

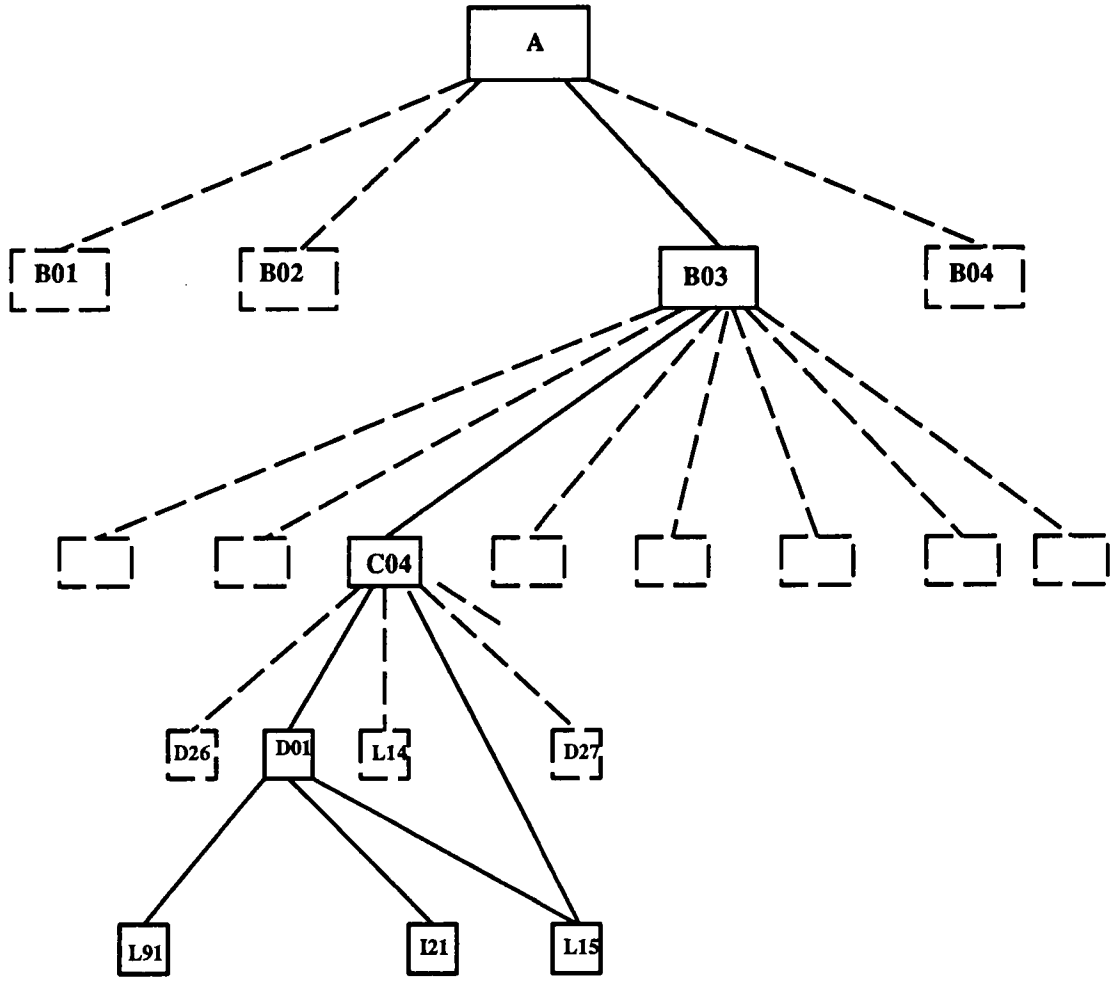


Figure 3.9 : Structure of the master controller part of the software

where speed is important. This facilitates the porting of the software to a different hardware easier. This part will be explained in chapter 6.

CHAPTER 4

INFORMATION HIDING IN THE MODULES

The principle of information hiding suggests that modules be characterized by design decisions that (each) hide from all others[5]. In other words, the modules should be designed in such a way that information, procedure, and data, contained in one module should be inaccessible from other modules that have no need for such information. The use of information hiding in module design is most beneficial in later phases of software life cycle, viz., testing and maintenance. Since most of the data and procedures are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations. This makes debugging and maintenance of the software much easier. This also makes the porting of the software to a different environment much easier.

4.1 Implementation of Information Hiding in Different Modules

In the older version of the software there was no information hiding in any of the modules. To improve the intra module cohesion and decrease inter module

coupling, movement of code was done between modules in addition to building of information hiding in the modules. Below we explain how these things were implemented in each of the modules.

4.1.1 main_pro.c

The main() function of the older version of the software has been totally rewritten. Previously the main() was around 1800 lines of code and it handled almost all the details by itself. It was however found out that the main() function basically does three broad different tasks as shown in Figure 4.1:

- Initializes the parameters connected with the manipulator.
- Initializes the shared memory to be used during real-time control of the manipulator.
- Waits for user input to perform different tasks.

It was found that there was lot of scope to build levels of abstraction into the software to make the code simpler. The new main() function now contains only four function calls: init_robot(), init_bit3(), check_and_process_input() and end_program(). Upon close observation, it was observed that most of the newly written functions could be written as *static* functions. All these routines are accessed from inside main_pro.c alone. Hence they were written as static local functions. A listing of these static functions can be found in Appendix B. The

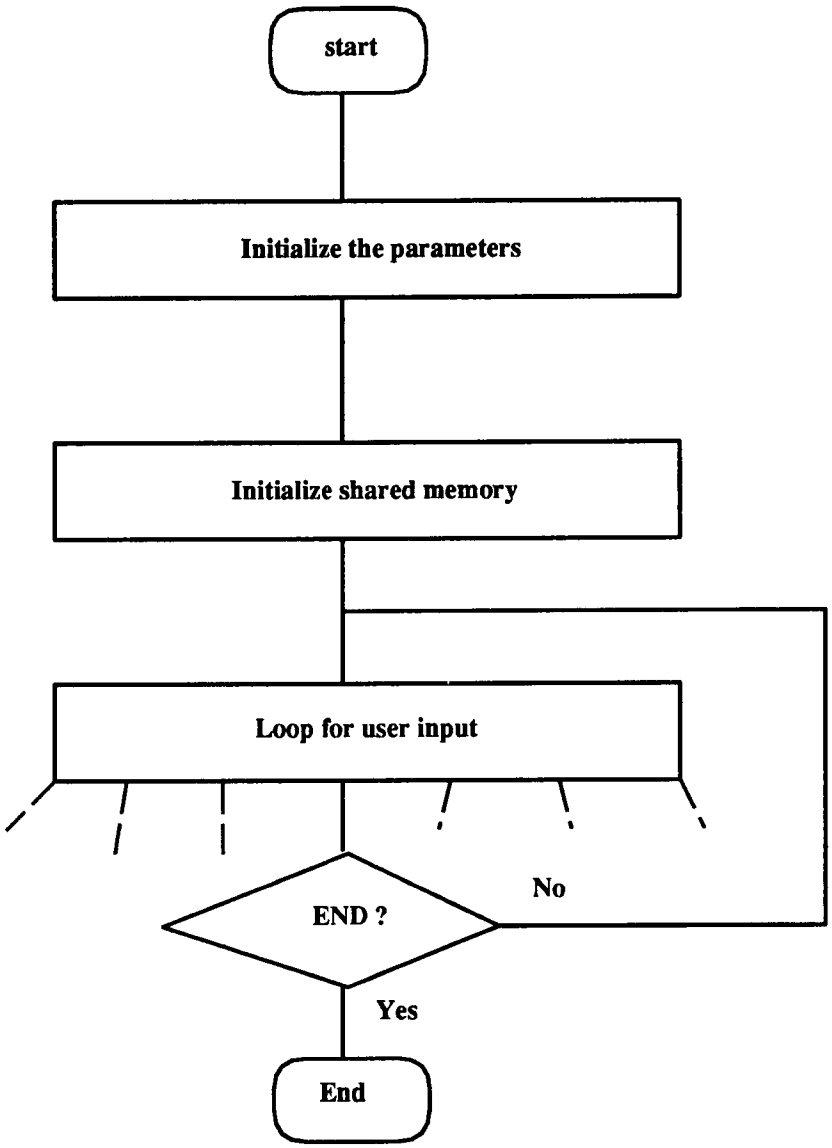
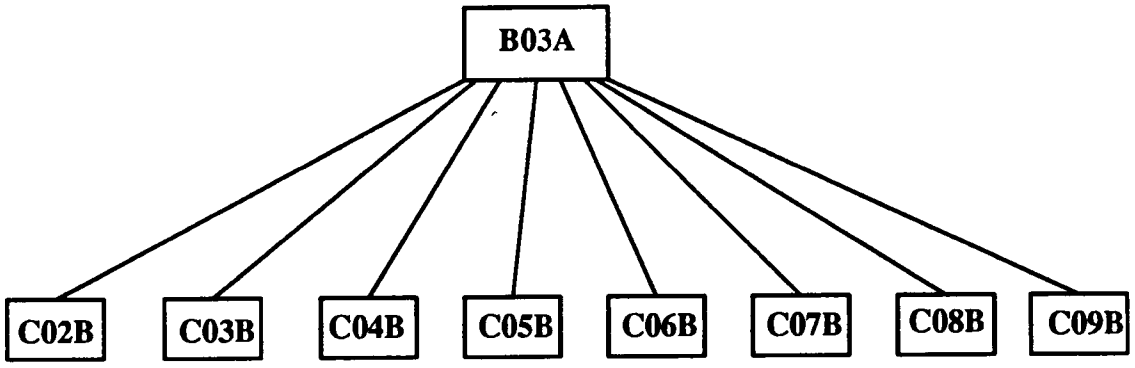


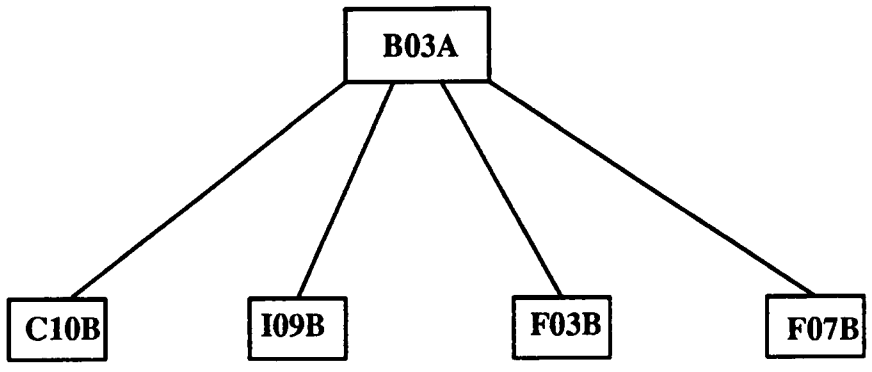
Figure 4.1 : Levels of abstraction in main() of new structure

advantage of making these functions static is that now they cannot be accessed from outside this module. Thus any inadvertent error cannot propagate from these functions to other modules. Thus debugging becomes much easier. Also making these functions small and specific task oriented, makes them easier to be tested. The function where the program will spend most of its time is the check_and_process_input() function. The different segments of the software are clearly identifiable from this subroutine itself. The master hand controller function flows from check_switches() function. The Cartesian motion of the manipulator flows out from teach_robot_cartesian() and the joint motion flows out from teach_robot_joint(). poll_lower_middle_column(), poll_left_column(), and poll_right_column() and poll_left_top_corner() provides more specialized functions that become evident once these functions are entered. Refer to Figure 4.2 for a graphical description. The different portions of the teach pendant are clearly identifiable in the software through the use of pre-processor macros that are defined in the robot.h header file.

4.1.2 draw_LTM.c

Upon close examination, the following global data members were found to be suitable candidates for semi global status: int frame_num; float frame_vect[100][8]; float cyn1[], cyn2[], cyn3[], cyn4[], cyn5[], cyn6[], cyn7[],





B03A check_and_process_input()
C02B process_keybrd_entry()
C03B read_mouse()
C04B check_switches()
C05B poll_left_column()
C06B teach_robot_cartesian()
C07B teach_robot_joint()
C08B poll_lower_middle_column()
C09B poll_right_column()
C10B poll_left_top_corner()
I09B draw_robot()
F03B draw_curv()
F07B final_check()

Fig 4.2 check_and_process_input()

cyn8[], cyn9[], cyn10[], cyn11[], cyn12[], cyz1[], cyz2[]; float nxy1[], nxy2[], nxy3[], nxy4[],nyz1[], nyz2[], nyz3[], nyz4[], nxz1[], nxz2[], nxz3[], nxz4[], nxzp1[], nxzp2[], nxzn1[], nxzn2[], nzxp1[], nzxp2[], nzxn1[], nzxn2[], cxy_n1[], cxy_n2[], cxy_n3[], cxy_n4[], cxy_n5[], cxy_n6[], cxy_n7[], cxy_n8[], cxy_n9[], cxy_n10[], cxy_n11[], cxy_n12[], cxz_n1[], cxz_n2[], cxz_n3[], cxz_n4[], cxz_n5[], cxz_n6[], cxz_n7[], cxz_n8[], cxz_n9[], cxz_n10[], cxz_n11[], cxz_n12[], cyz_n1[], cyz_n2[], cyz_n3[], cyz_n4[], cyz_n5[], cyz_n6[], cyz_n7[], cyz_n8[], cyz_n9[], cyz_n10[], cyz_n11[], cyz_n12[], LTM_fg_n[], cyl1[], cyl2[], cyl3[], cyl4[], cyl5[], cyl6[], cyl7[], cyl8[], cyl9[], cyl10[], cyl11[], cyl12[], cyl13[], cyl14[], cyl15[], cyl16[], cyl17[], cyl18[], cyl19[], cyl20[], cyl21[], cyl22[], cyl23[], cyl24[], cylct1[], cylct2[]; float LTM_low_lim[], LTM_up_lim[], LTM_jv_lim[], LTM_home[]; int LTM_init_obt_index, LTM_init_obt_sign; float LTM_theta[MAT_DIM], LTM_obt_vel; int LTM_obt_sign, LTM_obt_index; float LTM_view_mat[MAT_DIM][4][4], LTM_init_view_mat[MAT_DIM][4][4].

It was observed that these globals are used only in draw_LTM.c and thus should be private to this module. They were declared as static global variables so that they cannot be accessed from other modules by mistake. Thus this module remains intact even if the programmer redefines any of these semi global variables in another module.

The following global data members were found to be suitable candidates for local variables inside different subroutines. The subroutines and the incorporated global variables are as follows:

void dr_box(): float bx1[], bx[], bx2[], bx3[], bx4[], bx5[], bx6[], bx7[], bx8[].

static void cylinder1(): This has been made into a static function with the incorporation of following global variables as locals. float cy_fs_n1[], cy_fs_n2[], cy_fs_n3[], cy_fs_n4[], cy_fs_n5[], cy_fs_n6[], cy_fs_n7[], cy_fs_n8[], cy_fs_n9[], cy_fs_n10[], cy_fs_n11[], cy_fs_n12[].

static void draw_LTM_axis(): Similar to *cylinder1()* function this has been written as a local function. float LTM_axis1[], LTM_axis2[], LTM_axis3[], LTM_axis4[], LTM_axis5[], LTM_axis6[], LTM_axis7[], LTM_axis8[], LTM_axis9[], LTM_axis10[], LTM_axis11[], LTM_axis12[], LTM_axis13[], LTM_axis14[], LTM_axis15[], LTM_axis16[], LTM_axis17[], LTM_axis18[], LTM_axis19[], LTM_axis20[], LTM_axis21[], LTM_axis22[], LTM_axis23[], LTM_axis24[], LTM_axis25[], LTM_axis26[], LTM_axis27[], LTM_axis28[], LTM_axis29[], LTM_axis30[], LTM_axis31[], LTM_axis32[], LTM_axis33[], LTM_axis34[], LTM_axis35[], LTM_axis36[], LTM_axis37[], LTM_axis38[], LTM_axis39[], LTM_axis40[], LTM_axis41[], LTM_axis42[], LTM_axis43[], LTM_axis44[], LTM_axis45[], LTM_axis46[], LTM_axis47[], LTM_axis48[], LTM_axis49[], LTM_axis50[], LTM_axis51[], LTM_axis52[], LTM_axis53[],

LTM_axis54[], LTM_axis55[], LTM_axis56[], LTM_axis57[], LTM_axis58[], LTM_axis59[], LTM_axis60[], LTM_axis61[], LTM_axis62[], LTM_axis63[], LTM_axis64[], axis_n1[], axis_n2[], axis_n3[], axis_n4[], axis_n5[], axis_n6[], axis_n7[], axis_n8[], axis_n17[], axis_n18[], axis_n19[], axis_n20[], axis_n21[], axis_n22[], axis_n23[], axis_n24[], axis_n33[], axis_n34[], axis_n35[], axis_n36[], axis_n37[], axis_n38[], axis_n39[], axis_n40[], axis_n49[], axis_n50[], axis_n51[], axis_n52[], axis_n53[], axis_n54[], axis_n55[], axis_n56.

void draw_LTM_stand(): float LTM_stand1[], LTM_stand2[], LTM_stand3[], LTM_stand4[], LTM_stand5[], LTM_stand6[], LTM_stand7[], LTM_stand8[], LTM_stand9[], LTM_stand10[], LTM_stand11[], LTM_stand12[], LTM_stand13[], LTM_stand14[], LTM_stand15[], LTM_stand16[], LTM_stand17[], LTM_stand18[], LTM_stand19[], LTM_stand20[], LTM_stand21[], LTM_stand22[], LTM_stand23[], LTM_stand24[].

static void draw_LTM_part(): float LTM_bas1[], LTM_bas2[], LTM_bas3[], LTM_bas4[], LTM_bas5[], LTM_bas6[], LTM_bas7[], LTM_bas8[], LTM_bas9[], LTM_bas10[], LTM_bas11[], LTM_bas12[], LTM_bas13[], LTM_bas14[], LTM_bas15[], LTM_bas16[], LTM_bas17[], LTM_bas18[], LTM_bas19[], LTM_bas20[], LTM_bas21[], LTM_bas22[], LTM_bas23[], LTM_bas24[], LTM_bas25[], LTM_bas26[], LTM_bas27[], LTM_bas28[];

static void draw_LTM_uparm(): float LTM_uparm1[], LTM_uparm2[], LTM_uparm3[], LTM_uparm4[], LTM_uparm5[], LTM_uparm6[], LTM_uparm7[],LTM_uparm8[], LTM_uparm9[], LTM_uparm10[], LTM_uparm11[], LTM_uparm12[], LTM_uparm13[], LTM_uparm14[], LTM_uparm15[], LTM_uparm16[], LTM_uparm17[], LTM_uparm18[], LTM_uparm19[], LTM_uparm20[], LTM_uparm21[], LTM_uparm22[], LTM_uparm23[], LTM_uparm24[], LTM_uparm25[], LTM_uparm26[], LTM_uparm27[], LTM_uparm28[], LTM_uparm29[], LTM_uparm30[], LTM_uparm31[], LTM_uparm32[], LTM_uparm_n1[], LTM_uparm_n2[], LTM_uparm_n3[], LTM_uparm_n4[], LTM_uparm_n5[], LTM_uparm_n6[], LTM_uparm_n7[], LTM_uparm_n8[], LTM_uparm_n9[], LTM_uparm_n10[], LTM_uparm_n11[], LTM_uparm_n12[], LTM_uparm_n13[], LTM_uparm_n14[], LTM_uparm_n15[], LTM_uparm_n16[], LTM_uparm_n17[], LTM_uparm_n18[], LTM_uparm_n19[], LTM_uparm_n20[], LTM_uparm_n21[], LTM_uparm_n22[], LTM_uparm_n23[], LTM_uparm_n24[], LTM_uparm_n25[], LTM_uparm_n26[], LTM_uparm_n27[], LTM_uparm_n28[], LTM_uparm_n29[], LTM_uparm_n30[].

static void draw_LTM_lowarm(): float LTM_lowarm1[], LTM_lowarm2[], LTM_lowarm3[], LTM_lowarm4[], LTM_lowarm5[], LTM_lowarm6[], LTM_lowarm7[], LTM_lowarm8[], LTM_lowarm9[], LTM_lowarm10[],

LTM_lowarm11[], LTM_lowarm12[], LTM_lowarm13[], LTM_lowarm14[], LTM_lowarm15[], LTM_lowarm16[], LTM_lowarm17[], LTM_lowarm18[], LTM_lowarm19[], LTM_lowarm20[], LTM_lowarm21[], LTM_lowarm22[], LTM_lowarm23[], LTM_lowarm24[], LTM_lowarm25[], LTM_lowarm26[], LTM_lowarm27[], LTM_lowarm28[], LTM_lowarm29[], LTM_lowarm30[], LTM_lowarm31[], LTM_lowarm32[], LTM_lowarm33[], LTM_lowarm34[], LTM_lowarm35[], LTM_lowarm36[], LTM_lowarm37[], LTM_lowarm38[], LTM_lowarm39[], LTM_lowarm40[], LTM_lowarm41[], LTM_lowarm42[], LTM_lowarm43[], LTM_lowarm44[], LTM_lowarm45[], LTM_lowarm46[], LTM_lowarm47[], LTM_lowarm48[], LTM_lowarm49[], LTM_lowarm50[], LTM_lowarm51[], LTM_lowarm52[], LTM_lown1[], LTM_lown4[], LTM_lown5[], LTM_lown6[], LTM_lown9[], LTM_lown19[], LTM_lown20[], LTM_lown21[], LTM_lown22[], LTM_lown23[], LTM_lown25[].

static void draw_LTM_rst(): float LTM_rst1[]; LTM_rst2[]; LTM_rst3[]; LTM_rst4[]; LTM_rst5[]; LTM_rst6[]; LTM_rst7[]; LTM_rst8[]; LTM_rst9[]; LTM_rst10[]; LTM_rst11[]; LTM_rst12[]; LTM_rst13[]; LTM_rst14[]; LTM_rst15[]; LTM_rst16[]; LTM_rst17[]; LTM_rst18[]; LTM_rst19[]; LTM_rst20[]; LTM_rst21[]; LTM_rst22[]; LTM_rst23[]; LTM_rst24[]; LTM_rst25[]; LTM_rst26[]; LTM_rst27[]; LTM_rst28[]; LTM_rst29[];

LTM_rst30[]; LTM_rst31[]; LTM_rst32[]; LTM_rst33[]; LTM_rst34[]; LTM_rst35[]; LTM_rst36[]; LTM_rst37[]; LTM_rst38[].

static void draw_LTM_grip(): float LTM_gp1[], LTM_gp2[], LTM_gp3[], LTM_gp4[], LTM_gp5[], LTM_gp6[], LTM_gp7[], LTM_gp8[], LTM_gp9[], LTM_gp10[], LTM_gp11[], LTM_gp12[], LTM_gp13[], LTM_gp14[], LTM_gp15[], LTM_gp16[], LTM_gp17[], LTM_gp18[], LTM_gp19[], LTM_gp20[].

staic void draw_LTM_fg(): float LTM_fg1[], LTM_fg2[], LTM_fg3[], LTM_fg4[], LTM_fg5[], LTM_fg6[], LTM_fg7[], LTM_fg8[], LTM_fg9[], LTM_fg10[], LTM_fg11[], LTM_fg12[].

static void draw_wall(): float swa1[], swa2[], swa3[], swa4[], swa5[], swa6[], swa7[], swa8[], swa9[], swa10[], swa11[], swa12[], swb1[], swb2[], swb3[], swb4[], swc1[], swc2[], swc3[], swc4[], swd1[], swd2[], swd3[], swd4[], swd5[], swd6[], swd7[], swd8[].

static void draw_stand1(): float sda1[], sda2[], sda3[], sda4[], sda5[], sda6[], sda7[], sda8[].

static void draw_stand2(): float sdb1[], sdb2[], sdb3[], sdb4[], sdb5[], sdb6[], sdb7[], sdb8[].

static void draw_plane(): float plane1[], plane2[], plane3[], plane4[].

4.1.3 goal_p.c

The goal_p.c contains utility routines used in trajectory and path planning. Upon close examination it was found that the following functions are essentially local to the module and hence qualifies to be static functions.

int circle(), int self_motion(), int linear_cont_grad(), int grad_lin(), int grad_circle(), void get_grad_dat(), void gcost_data(), void gdet_data(), void geig_data(), void gdet6_data(), void get_jlim_data(), void get_jlim_data2(), void get_max_jv(), int pause_key().

4.1.4 draw_k2107.c

A close examination of the global variables show that the following global variables can be made into semi global variables:

float nx1[], nx2[], ny1[], ny2[], nz1[], nz2[], v0[], v1[], v2[], v3[], v4[], v5[], v6[], v7[], v8[], v9[], v10[], v11[], v12[], v13[], v14[], v15[], v16[], v17[], n1[], n2[], n3[], n4[], n5[], n6[], n7[], av0[], av1[], av2[], av3[], av4[], av5[], av6[], av7[], av8[], av9[], av10[], av11[], av12[], av13[], av14[], av15[], av16[], av17[], av18[], av19[], av20[], av21[], av22[], av23[], av24[], av25[], av26[], av27[], av28[], av29[], av30[], av31[], av32[], av33[], av34[], av35[], av36[], av37[], av38[], av39[], av40[], av41[], av42[], av43[], av44[], av45[], av46[], av47[], av48[],

av49[], av50[], av51[], av52[], av53[], av54[], av55[], av56[], av57[], av58[], av59[], an1[], an2[], an4[], an5[], an7[], an8[], an10[], an11[], an26[], an27[], an29[], an30[], bv0[], bv1[], bv2[], bv3[], bv4[], bv5[], bv6[], bv7[], bv8[], bv9[], bv10[], bv11[], bv12[], bv13[], bv14[], bv15[], bv16[], bv17[], bv18[], bv19[], bv20[], bv21[], bv22[], bv23[], bv24[], bv25[], bv26[], bv27[], bv28[], bv29[], bv30[], bv31[], bv32[], bv33[], bv34[], bv35[], bv36[], bv37[], bv38[], bv39[], bv40[], bv41[], bv42[], bv43[], bn2[], bn3[], bn5[], bn6[], bn8[], bn9[], bn21[], bn22[], bn22[], bn24[], bn25[], bn27[], bn28[], bn30[], bn31[], cv0[], cv1[], cv2[], cv3[], cv4[], cv5[], cv6[], cv7[], cv8[], cv9[], cv10[], cv11[], cv12[], cv13[], cv14[], cv15[], cv16[], cv17[], cv18[], cv19[], cv20[], cv21[], cv22[], cv23[], cv24[], cv25[], cv26[], cv27[], cv28[], cv29[], cv30[], cv31[], cv32[], cv33[], cv34[], cv35[], cv36[], cv37[], cv38[], cv39[], cv40[], cv41[], cv42[], cv43[], cv44[], cv45[], cv46[], cv47[], cv48[], cv49[], cv50[], cv51[], cv52[], cv53[], cv54[], cv55[], cv56[], cv57[], cv58[], cv59[], cv60[], cv61[], cn0[], cn1[], cn2[], cn3[], cn4[], cn5[], cn6[], cn7[], cn8[], cn9[], cn10[], cn11[], cn16[], cn20[], cn25[], cn26[], cn28[], cn29[], cn31[], cn32[], dv0[], dv1[], dv2[], dv3[], dv4[], dv5[], dv6[], dv7[], dv8[], dv9[], dv10[], dv11[], dv12[], dv13[], dv14[], dv15[], dv16[], dv17[], dv18[], dv19[], dv20[], dv21[], dv22[], dv23[], dv24[], dv25[], dv26[], dv27[], dv28[], dv29[], dv30[], dv31[], dv32[], dv33[], dv34[], dv35[], dv36[], dv37[], dv38[], dv39[], dv40[], dv41[], dv42[], dv43[], dn2[], dn3[], dn5[],

dn6[], dn8[], dn9[], dn21[], dn22[], dn24[], dn25[], dn27[], dn28[], dn30[], dn31[], ev0[], ev1[], ev2[], ev3[], ev4[], ev5[], ev6[], ev7[], ev8[], ev9[], ev10[], ev11[], ev12[], ev13[], ev14[], ev15[], ev16[], ev17[], ev18[], ev19[], ev20[], ev21[], ev22[], ev23[], ev24[], ev25[], ev26[], ev27[], ev28[], ev29[], ev30[], ev31[], ev32[], ev33[], ev34[], ev35[], ev36[], ev37[], ev38[], ev39[], ev40[], ev41[], ev42[], ev43[], ev44[], ev45[], ev46[], ev47[], ev48[], ev49[], ev50[], ev51[], ev52[], ev53[], ev54[], ev55[], ev56[], ev57[], ev58[], ev59[], ev60[], ev61[], en0[], en1[], en2[], en3[], en4[], en5[], en6[], en7[], en8[], en9[], en10[], en11[], en16[], en20[], en25[], en26[], en28[], en29[], en31[], en32[], fv0[], fv1[], fv2[], fv3[], fv4[], fv5[], fv6[], fv7[], fv8[], fv9[], fv10[], fv11[], fv12[], fv13[], fv14[], fv15[], fv16[], fv17[], fv18[], fv19[], fv20[], fv21[], fv22[], fv23[], fv24[], fv25[], fv26[], fv27[], fv28[], fv29[], fv30[], fv31[], fv32[], fv33[], fv34[], fv35[], fv36[], fv37[], fv38[], fv39[], fv40[], fv41[], fv42[], fv43[], fn1[], fn2[], fn4[], fn5[], fn7[], fn8[], fn21[], fn22[], fn24[], fn25[], fn27[], fn30[], fn31[], gv0[], gv1[], gv2[], gv3[], gv4[], gv5[], gv6[], gv7[], gv8[], gv9[], gv10[], gv11[], gv12[], gv13[], gv14[], gv15[], gv16[], gv17[], gv18[], gv19[], gv20[], gv21[], gv22[], gv23[], gv24[], gv25[], gv26[], gv27[], gv28[], gv29[], gv30[], gv31[], gv32[], gv33[], gv34[], gv35[], hv0[], hv1[], hv2[], hv3[], hv4[], hv5[], hv6[], hv7[], hv8[], hv9[], hv10[], hv11[], hv12[], hv13[], hv14[], hv15[], hv16[], hv17[], hv18[], hv19[], hv20[], hv21[], hv22[], hv23[], hv24[], hv25[], hv26[], hv27[], hv28[], hv29[],

hv30[], hv31[], hv32[], hv33[], hv34[], hv35[], hv36[], hv37[], hv38[], hv39[], hv40[], hv41[], hv42[], hv43[], hv44[], hv45[], hv46[], hv47[], gv56[], gv57[], gv58[], gv59[], gv60[], gv61[], gv62[], gv63[], gv64[], gv65[], gv66[], gv67[], gv68[], gv69[], gv70[], gv71[], gv72[], gv73[], gv74[], gv75[], gv76[], gv77[], gv78[], gv79[], gv80[], gv81[], gv40[], gv41[], gv42[], gv43[], gv44[], gv45[], gv46[], gv47[], gv48[], gv49[], gv50[], gv51[], gv52[], gv53[], gv54[], gv55[], gn1[], gn2[], gn4[], gn5[], gn7[], gn8[], gn10[], gn11[],gv0[], ggv1[], ggv2[], ggv3[], ggv4[], ggv5[], ggv6[], ggv7[], ggv8[], ggv9[], ggv10[], ggv11[], ggv12[], ggv13[], ggv14[], ggv15[], ggv16[], ggv17[], ggv18[], ggv19[], ggv20[], ggv21[], ggv22[], ggv23[], ggv24[], ggv25[], ggv26[], ggv27[], ggv28[], ggv29[], ggv30[], ggv31[], ggv32[], ggv33[], ggv34[], ggv35[], ggn[], ggn2[],

4.1.5 init_graph.c

The following global data members were found to be suitable candidates for semi global status: long *font_big;int show_path; Angle cam_angle[MAT_DIM]; float cam_fovy[MAT_DIM]; Coord cam_near[MAT_DIM], cam_far[MAT_DIM]; long color_back, color_data, color_button, color_title, color_bt_char, color_back2, color_frame; int changing_view; float norm_light[TSTEPS][TPATHS][3], view_point2[], g_mat1[4][4], g_mat2[4][4], id_mat[4][4]; long *font_icon; float view_btn[][3]; int graph_state, prn_color; float norm_x1[], norm_x2[];

The following global data members were made local to make the associated functions self contained.

void def_light(): float left_right[], right_light[], back_light[], shiny_material[], sensor_material[], sensor_material2[], new_material[], shiny_material2[], shiny_material3[], shiny_material4[], normal_material1[], wall_material[], ceiling_material[], floor_material[].

void def_light2(): float light1[], light2[], light3[], light4[], normal_material[].

void drawaxis(): float axis_x1[], axis_x2[], axis_x21[], axis_y1[], axis_y2[], axis_y21[], axis_z1[], axis_z2[], arrox1[], arrox2[], arroy1[], arroy2[], arroz1[], arroz2[], arrox11[], arrox21[], arroy11[], arroy21[], arroz11[], arroz21[].

void fourth_ellipse(): float ellipse0[], ellipse1[], ellipse2[], ellipse3[], ellipse4[], ellipse5[], ellipse6[], ellipse7[], ellipse8[], ellipse9[], ellipse10[], ellipse11[], ellipse12[], ellipse13[], ellipse14[], ellipse15[], ellipse16[], ellipse17[], ellipse18[], ellipse19[], ellipse20[], ellipse21[], ellipse22[], ellipse23[], ellipse24[], ellipse25[], ellipse26[], ellipse27[], ellipse28[], ellipse29[], ellipse30[], ellipse31[], ellipse32[], ellipse33[], ellipse34[], ellipse35[], ellipse36[], ellipse37[], ellipse38[], ellipse39[], ellipse40[], ellipse41[], ellipse42[], ellipse43[], ellipse44[], ellipse45[], ellipse46[], ellipse47[], ellipse48[], ellipse49[], ellipse50[], ellipse51[], ellipse52[], ellipse53[], ellipse54[], ellipse55[], ellipse56[], ellipse57[], ellipse58[], ellipse59[], ellipse60[], ellipse61[], ellipse62[], ellipse63[], ellipse64[], ellipse65[], ellipse66[], ellipse67[],

ellipse68[], ellipse69[], ellipse70[], ellipse71[], ellipse72[], ellipse73[], ellipse74[], ellipse75[], ellipse76[], ellipse77[], ellipse78[].

4.1.6 robot_math.c

This module has been created newly and contains the utility routines which are being called by different modules. All the functions in this module are public and qualify themselves to be included in the library of routines, as they have become pretty standard by now. The function in this modules are: void end_vels(), void determ(), int equit(), void cal_jjt(), void svd(), void new_theta(), void rpy_angles(), void get_position(), void difference(), void minimum_rot(), void pop_cursor(). Making of this utility module greately reduces the compilation time as they have been placed in the library librobot.a along with the other graphics related routines.

CHAPTER 5

MAINTENANCE OF CORRECT INTERFACES FOR ERROR FREE PROGRAMMING

This chapter discusses the importance of maintaining correct interfaces between functions, and discusses the implementation of correct remedies. First of all we discuss the maintenance of the interfaces in general, with emphasis on C programming. At the later section, we explore a few techniques to remove chances of inadvertent errors on the part of the programmer. In this we discuss how the power of UNIX and C can be exploited to develop correct, error free programs, for a real-time application like this.

5.1 Importance of Correct Interfaces

Development and maintenance of correct interfaces between modules and functions is extremely important for development of correct and robust programs. Whenever we talk of function interface we basically mean how data is passed from one function to another. There can be two ways in which data can be passed from one function to another or between functions:

- through global or common data and

- through parameter passing and return values.

The first form of passing data leads to high coupling between modules. Presence of too many global variables in a program is undesirable, because beside increasing coupling, they also have side effects. This aspect has been discussed thoroughly in chapter 4. The second form of data passing is through function arguments and their return values. It is extremely important that function interfaces are consistent for correct development of the program. As a matter of fact, C is not a very typed language as its object oriented offspring C++. This is a fault with C. Many programs that may compile without any warning when compiled using a particular C compiler, may not compile under a C++ compiler. This is because C++ is a strongly typed language. This means any C code passes through a strong type checking when it is compiled with a C++ compiler. It may not be obvious to many experienced programmers of C that any program that compiles correctly may not necessarily produce correct or desired code. This is due to the lack of type checking by the C compiler. C provides the onus of this on the programmer. Fortunately, C and UNIX provides some tools through which the programmer can assure development of correct code in C. These tools are the header files in C and the Make utility in UNIX. Any object which is referenced in more than one source files should be defined in a header file which should be #included in the two

source files. In the makefile, header file based dependency is built up so that whenever there is a change in the interface, the associated files are recompiled.

5.2 Study of the Interfaces and Remedies

In this section we study the software for possible errors in maintenance of interfaces and then we talk corrective measures so that the burden of checking for the maintenance of correct interfaces gets passed on from the programmer to the compiler.

5.2.1 Possible Trouble Due to Inadvertent Error on the Part of the Programmer

The older version of the software was checked for these kind of problems and it was found that sufficient error checking capability was not built into it to take care of inadvertent errors made by the programmer. As a result, there was scope that even though the program compiled without any warning, the code might have been producing a wrong result. Two scenarios were taken to illustrate a problem of this type. In one, the function interfaces were maintained correctly, that is the functions were called using the same arguments as its prototype definition. In the second case, a particular function was called with wrong arguments. As there is no type checking mechanism built into the program, both

the times the program compiled successfully without producing any error. But the program when executed separately in each case, showed simulated displays that were drastically different. Figure 5.1 shows the display of the RRC manipulator when correct interfaces were used, whereas Figure 5.2 shows the simulated display of the manipulator when one function select_robot(int robot_index) was called without any arguments. The compiler failed to catch the error the programmer had made, thus the programmer had no clue of his inadvertent error. So he/she will assume that everything is correct and will execute the program to run the robot. Figure 5.2 shows the configuration of the robot which should have been exactly like the display in Figure 5.1. One can clearly imagine what can happen to the RRC manipulator when such a piece of code is encountered in the real-time control of the manipulator.

5.2.2 Corrective Measures to Remove the Chances of Inadvertent Errors

This problem can be removed by incorporating header files and correcting the inconsistencies in the makefile being used in the older version of the software. As there were no user defined header files in the entire software and no sufficient dependency built into the makefile, the compiler did not get enough information to check for this kind of error. To remove these dangerous errors, header files were

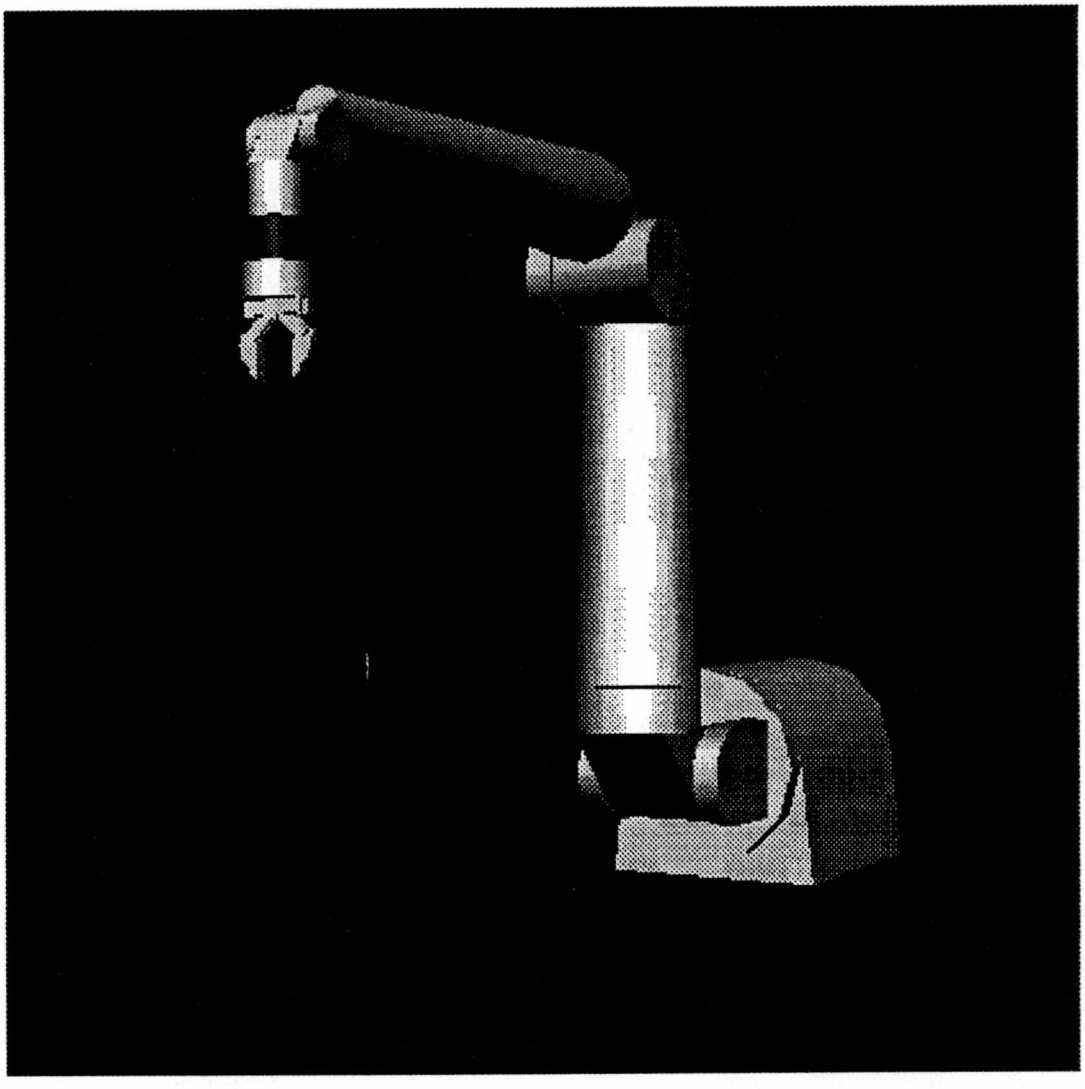


Figure 5.1 Correct configuration of the Robot

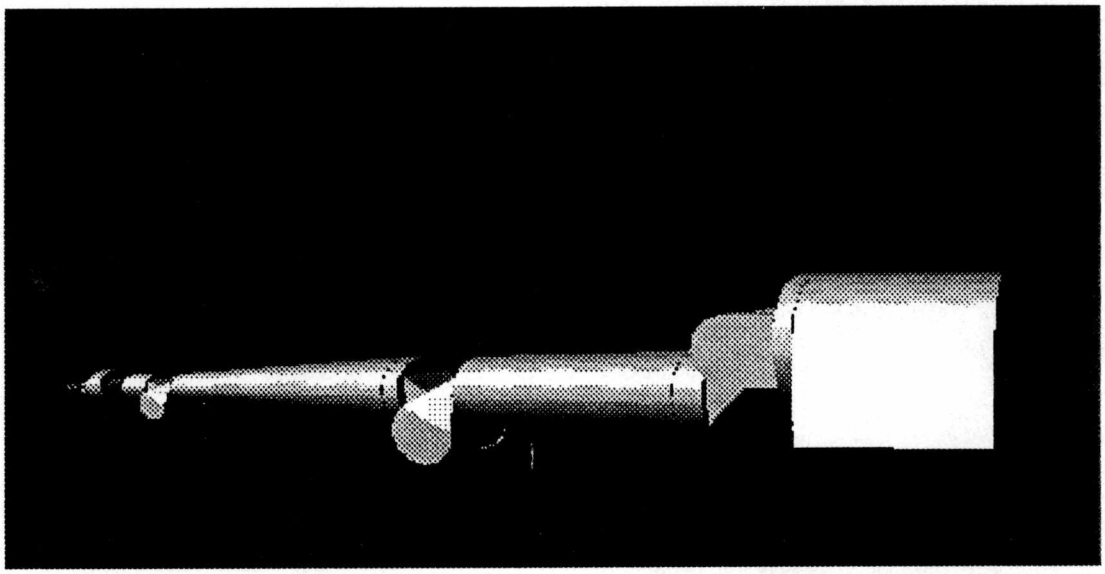


Figure 5.2 : Faulty configuration of the robot due to faulty interface (This configuration should have been just like Figure 5.1)

created to contain the information common to two or more source files. These header files were carefully incorporated into different modules using #include directive of the preprocessor. The makefile was completely rewritten with the associated header file dependencies. The software was treated for the above kind of errors. This it was found that when a function was called wrongly, the program will not even compile and the compiler/linker immediately informs the programmer of the location of the error in advance. Thus the chances of encountering these kind of inadvertent errors were eliminated due to incorporation of the header files and correction of the makefile. The associated header files and the makefile are shown in Appendix C.

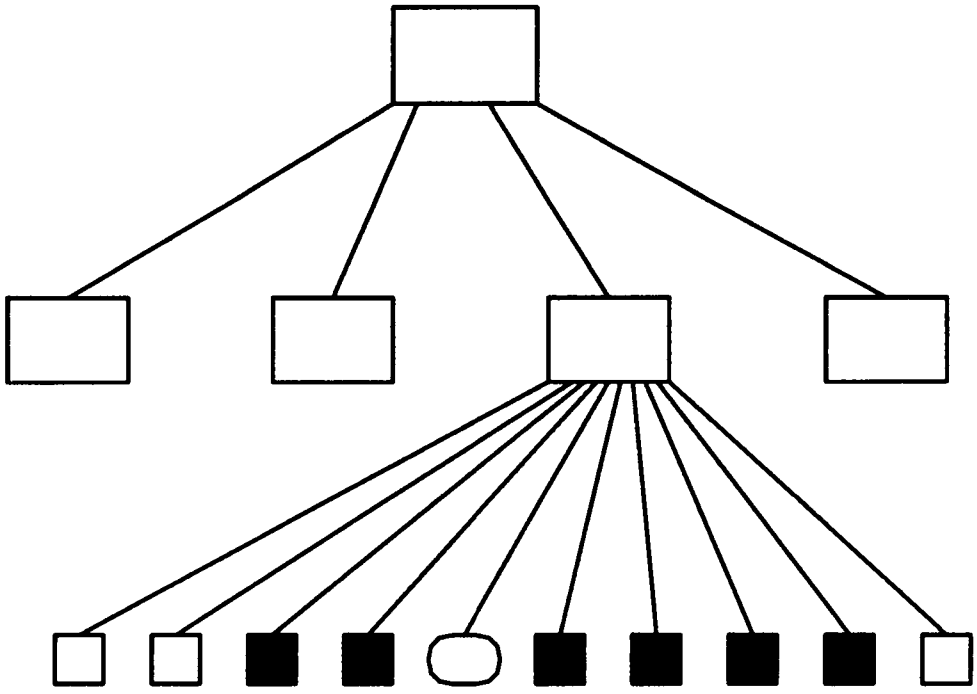
CHAPTER 6

THE NEW STRUCTURE : EASE OF TESTING AND PORTING

In this chapter we explain how the new structure is easier to test and also easier to port to a different system. The information hiding built into the software facilitates the unit and integration testing of the software. Since the software has been built with a structure with low fan-out and increased number of layers, it can be easily tested unit by unit with the help of appropriate drivers and stubs.

6.1 A Study of a Few Test Cases

We consider the scenario of testing the Cartesian motion of the manipulator on the IRIX workstation with the graphical user interface in place. Figure 6.1 shows a diagrammatic view of testing the Cartesian motion by placing stubs at other places below check_and_process_input() subroutine. Stubs can be placed at check_switches(), poll_left column(), teach_robot_joint(), poll_lower_middle_column(), poll_right_column(), poll_left_top_corner(). Other portions of the software need not be touched at all. This was not so easy in the older version as different portions of the software were not well defined. As a



Stubs are at:

- **check_switches()**
- **poll_left_column()**
- **teach_robot_joint()**
- **poll_lower_middle_column()**
- **poll_right_column()**
- **poll_left_top_corner()**

Unit to be tested

- **teach_robot_cartesian()**

Figure 6.1 Testing of Cartesian Motion

result, changes had to be made at different places. Similarly if we want to test the changing of view of the camera, we can place stubs on the check_switches(), teach_robot_joint(), teach_robot_cartesian(), poll_lower_middle_column(), poll_right_column(), poll_left_top_corner(), change_local_coord(), poll_new_menu(), dm6_buton(), force_sensor_option(), run_robot(). Other portions of the software need not be touched at all. Figure 6.2 shows this pictorially.

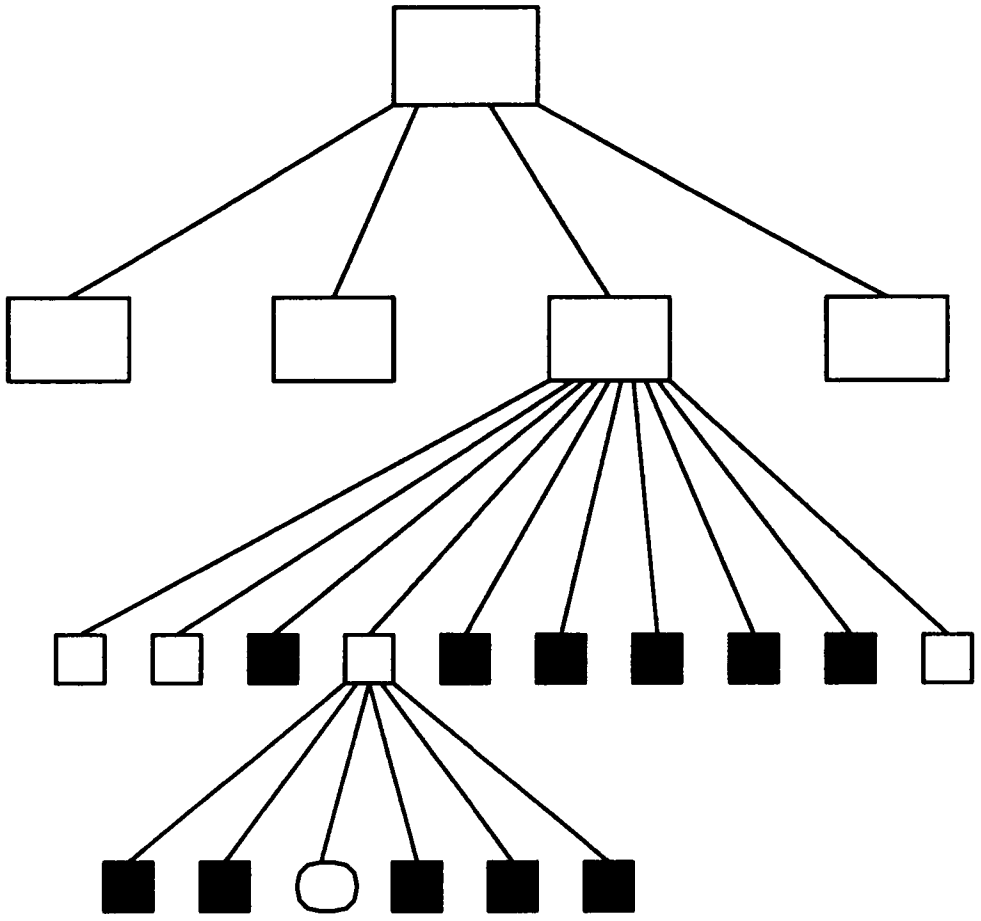
6.2 Ease of Porting to a Different Environment

As mentioned earlier, though this structure seems to be developed on the basis of the present graphical user interface, it can be easily modified to take input from any other input sources like keyboard, joy stick or even network connection. Only thing that needs to be done is the changing of the macros on the robot.h header file. For example, the macro LEFT_COLUMN_TP on the check_and_process_input() function now refers to the location (1020<=mx) && (mx<=1090) on the graphic display through this line in the robot.h file

#define LEFT_COLUMN_TP (1020<=mx) && (mx<=1090)

Now if we want this routine to be keyboard driven, we will simply replace the above macro with something like

#define LEFT_COLUMN_TP (getbutton(UPARROWKEY))



Stubs are at:

- check_switches()
- teach_robot_cartesian()
- teach_robot_joint()
- poll_lower_middle_column()
- poll_right_column()
- poll_left_top_corner()
- change_local_coord()
- change_view_point()
- dm6_button()
- force_sensor_option()
- run_robot()

Unit to be tested

- poll_new_menu()

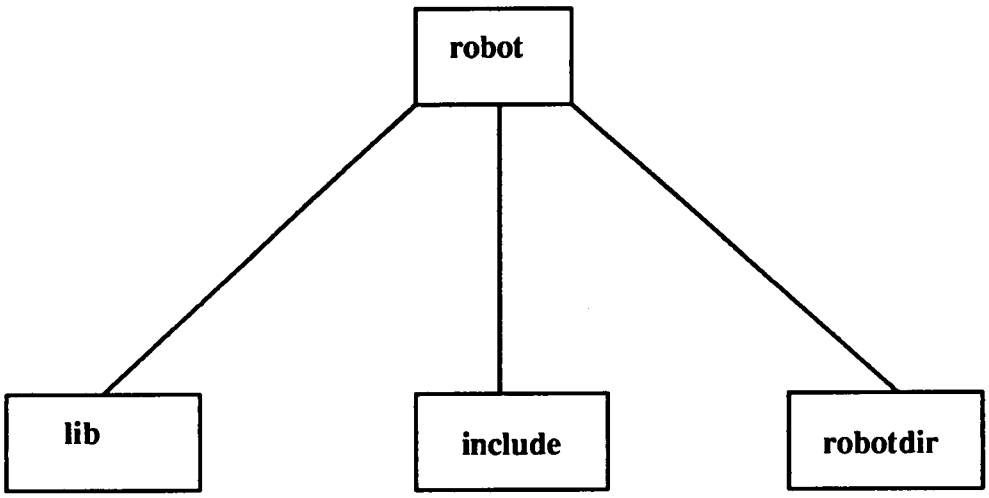
Figure 6.2 Testing of poll_new_menu()

Now the same action as before will occur but the input is from the keyboard instead of mouse. There is no need to change any other portion of the software. Similarly, if we need to port the software to a hardware that does not support graphics, we only need to undefine IRIX through the header file or the makefile. Any changes in the draw_robot() function can be made if required, and we can simply put stubs in place of the functions connected with graphics like changing camera angle etc. Even without putting the stubs it should work. The logic behind this is that portions within #ifdef IRIX and #elif, are compiled only when the variable IRIX is defined. If the software is ported to a different environment supporting a different graphics environment, the environment dependent graphics can similarly be incorporated with preprocessor directive #elif to conditionally compile those routines by defining a variable other than IRIX. An example of this is shown in Appendix D.

6.3 Reduction of the Effective Size of the Software

The software under study has a considerable volume and this necessitates the need for reducing the effective size of the software so that the program development time is reduced. Information hiding acts as a powerful tool for reducing the effective size of the software. If information hiding is properly built up, a programmer who starts working on the project need not bother about the

private components of the software unless he wants to change them. He/She only has to bother about the inter module interfaces. Thus making many of the global variables and functions static reduces the effective size of the software. Also many parts of the software have been reasonably standardized by now and are not likely to change in the future. All these can be made into a library of routines. The graphics have been almost standardized and are not likely to undergo further changes. In fact, they may be phased out in the future. The need to understand these parts of the software may well be eliminated. In order to reduce the development time of the software four modules namely draw_K2107.c, draw_LTM.c, init_graph.c and the newly created robot_math.c have been archived into a library of routines. They have been placed into a directory named "lib". All the header files have been placed in directory "include" and rest of the modules are placed in the directory "robotdir"(Figure 6.3).



Contents of the directories

lib : library of routines and makefile

include : Header files

robotdir : source .c files and makefile

robot : robot executable and makefile

Figure 6.3 Directory structure of the software

CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

Conclusions

A thorough study of the entire software - for simulation and real-time control of the seven degree of freedom RRC manipulator - was made to identify different portions where structure could be imposed. Care was taken so that the speed of the software in real-time control of the manipulator was not affected due to structuring.

Levels of abstraction were built up in the software by reducing fan-out at upper levels. A structured design in the style of desirable design heuristics was built up by judicious layering of the software. Portions of the software that were connected with speed in real-time control of the manipulator were carefully restructured to maintain the function overheads. These include the Cartesian and Joint motion, trajectory planning of the manipulator between specified points, and real-time control of the manipulator using the Kraft hand controller. The other portions of the software were thoroughly restructured to build up the levels of abstraction. The respective parts of the code can now be identified by looking at the graphic display itself. Preprocessor macros have been used to make the software easily readable. This restructuring along with these macros have also

made the software easy to change by taking input from a source other than the mouse. This has resulted in simplicity and easy readability of the software. Different parts of the software have become easily identifiable. This should have the effect of reducing the development and maintenance time of the software. Certain logically dissimilar parts were segregated to improve intra-module cohesion and to reduce inter module coupling. The joint and Cartesian motions have been separated into two distinct sub modules.

Information hiding was built up throughout the modules by suppressing details. Many of the global variables have been made local or semi global. Number of functions have been made private so that they cannot be accessed from outside the module where they are defined. Many functions have been made self contained to improve code reusability.

Problems with interface design were identified. Instances of possible trouble in real-time control of the manipulator were discovered. Remedial steps were taken to eliminate the chances of these inadvertent errors on the part of the programmer. Software was made more robust by giving more information to the compiler. Header files were incorporated and dependencies based on these were developed in the makefile.

The improved structure has resulted in the ease of testing of the software. Porting of the software has been made easier by incorporation of C preprocessor

directives in hardware dependent code sections. Effective size of the software has been reduced by 40-45% by creation of a library of routines from where the programmer can directly call the routine he/she may need without bothering about implementation details. To create these library routines, parts of the software that had become relatively standardized were identified and moved to a different utility module. The entire graphics code has been made self contained and moved into the library. The programmer no longer has the source code of the graphics part of the software before him/her. Now the programmer simply needs to call the required subroutine from the library. This should also make the planned phasing out of the graphics part easier.

The software was tested successfully in the simulation mode using the Kraft hand controller as the master and the 3-D graphics model on Silicon Graphics as slave.

Recommendations

To eliminate the chances of the programmer making inadvertent errors, it is imperative to come up with a good interface design. Areas of public and private information should be identified. Efforts should be made to minimize the public interface. To promote consistent interfaces, function prototypes should be declared, and the necessary information should be shared between the modules by

the use of header files. *Lint*, a static analysis tool available in the UNIX environment, should be used to check error in the interfaces. The use of lint will assist in reporting the following[3]:

- The syntax errors the compiler would have found.
- Functions devoid of return value checks.
- Functions declarations and calls with incorrect numbers or types of arguments.
- Declared but unused variables.
- Program statements that are unreachable.
- Inadvisable automatic type conversions.

Since the software still has a considerable number of global variables, it was decided that these global variables would be retained so that the function overheads are minimized by reducing the number of arguments passed. However, many of these global variables can be reduced by putting the related variables in suitable *structures*. For example, variables connected with kinematics can be put in one structure, variables connected with trajectory planning can be put in another structure. Instead of passing too many variables through parameter passing, a pointer to the relevant *structures* can be passed.

The problem of global variables can be greatly eliminated by moving towards an object oriented design. By writing the software in C++, these global variables can be made into *private* data members which can be manipulated only

by the associated *class* member functions. For example, if we have a kinematics *class* all the global variables connected with it can be made *private* data members of the *class*. To improve data encapsulation, member functions like jacobian(), transf(), can be made into *private* member functions. This is because they need to be accessed from inside the class only. By carefully designing a *class* structure, the software can be made easily portable and reusable. The reduction of the *public* interface will greatly reduce the programming time in future. This way any programmer using a particular class will need to concentrate only on the public interface. He/She need not bother anything about the private components. Also there will be no chanches of clashing variable names.

Inheritance can be built up to extend the software to manipulators of different degrees of freedom. This can be done by the creation of abstract base classes and through the use of *virtual* functions. For example, an abstract base *class* robot can be created with *pure virtual* functions jacobian(), transf() etc. as *protected* member functions. These *pure virtual* functions can then be redefined in the derived classes related to different degrees of freedom. The non *virtual* or other *virtual* portions of the code in the *super class* can be used in the *derived* classes as the need arises. Readability and ease of programming can be vastly improved by judicious use of function and operator overloading. Overheads in function invocation can be greatly reduced by taking advantage of *inline* functions.

Functions like new_theta(), rpy_angles(), get_position() etc. can easily be made into *inline* functions to reduce the function overheads in their calls. Taking the advantage of inline functions will also improve the readability of the software.

LIST OF REFERENCES

REFERENCES

1. Pfleeger, S. L - Software Engineering: the production of Quality Software. - Macmillan Publishing Company.

2. Kenneth D. Shere - Software Engineering and Management - Prentice Hall, Englewood Cliffs, New Jersey 07632

3. Frakes. W. B, Fox. C. J, Nejmeh. B. A, - Software Engineering in the UNIX/C Environment. Prentice Hall Software Series.

4. Kernighan , B.W & Ritchie, D.M -The C Programming Language - Englewood Cliffs, N.J: Prentice Hall, 1988

5. Pressman Roger S.- Software Engineering: A practitioner's Approach. - 3rd Edition 1992, McGraw-Hill, Inc.

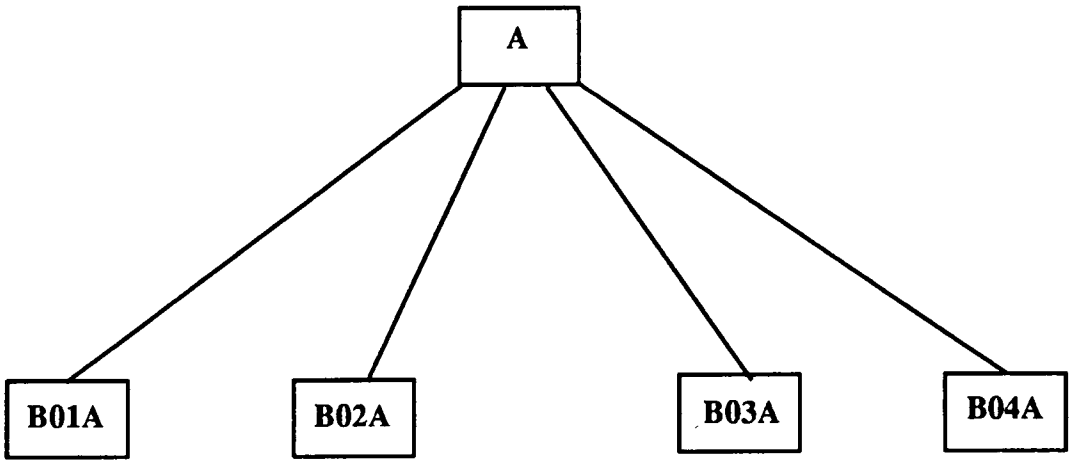
6. Chan T. F. - Simulation and Real-Time control of a seven-degree of freedom Manipulator.

7. Stroustrup. B. -The C++ Programming Language - Addison-Wesley Series in Computer Science

APPENDICES

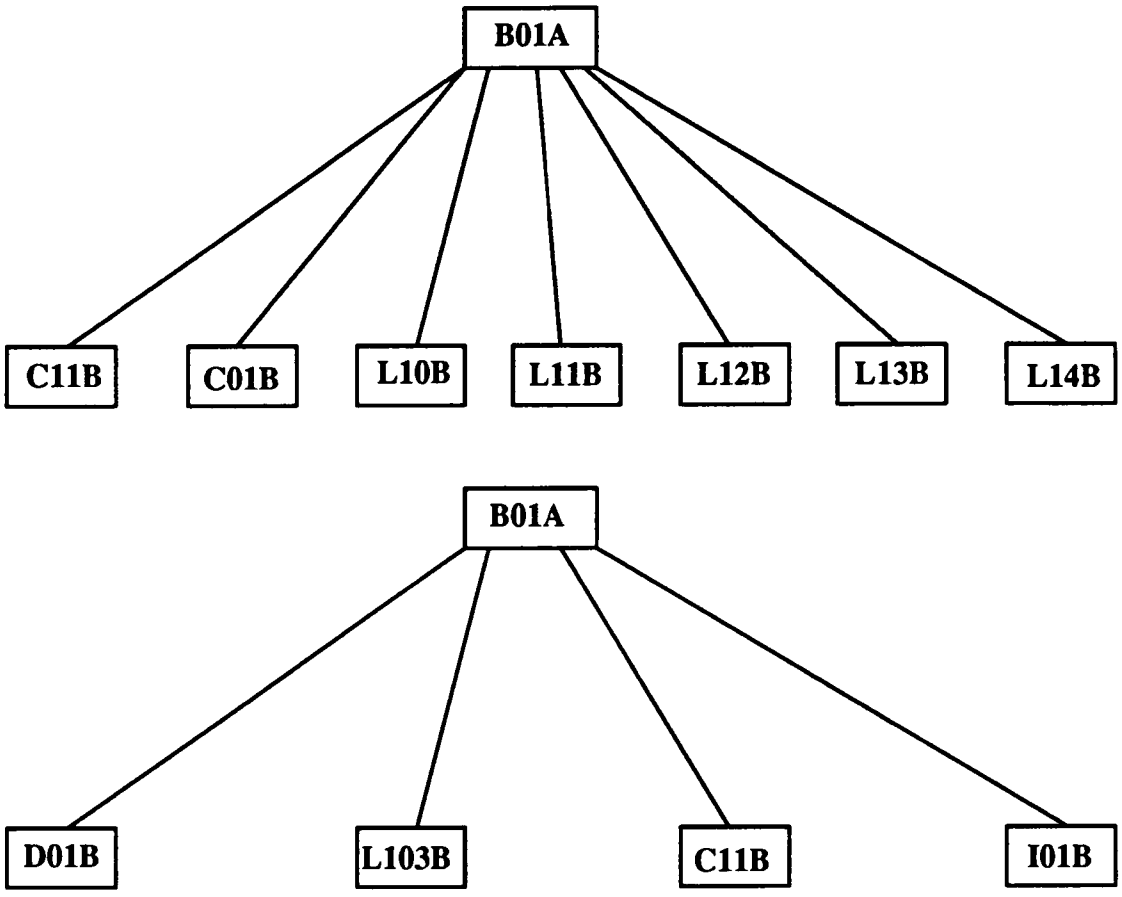
APPENDIX A

FUNCTION DOCUMENTATION



A main()
B01A init_robot()
B02A init_bit3()
B03A check_and_process_input()
B04A end_program()

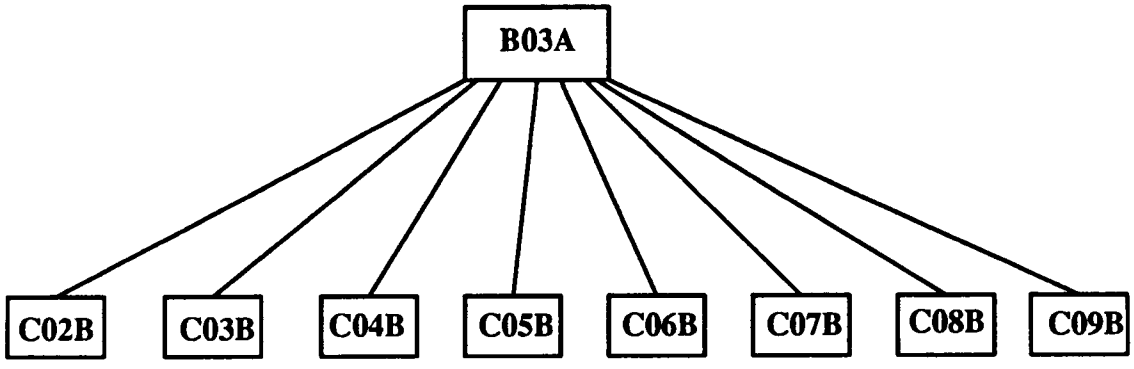
Fig A-1 main()

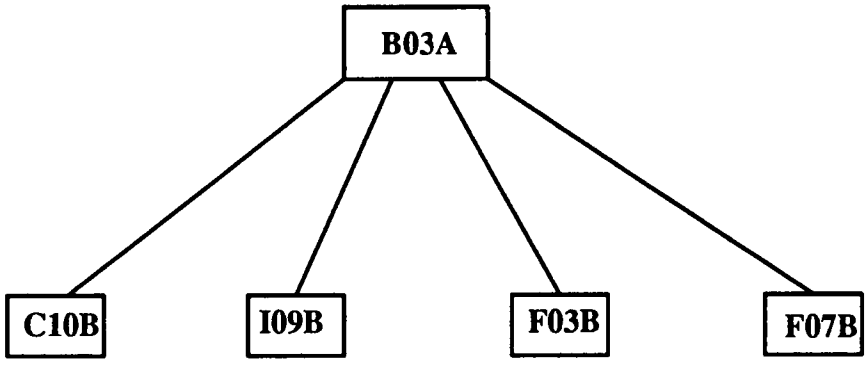


B01A init_robot()
C11B init_parameters()
C01B select_robot()
L10B init_com3()
L11B init_com_mast()
L12B init_comm5()

L13B init_comm6()
L14B init_comm7()
D01B poll_manager()
L103B init_curv_win()
C11B init_rob_win()
I01B get_position()

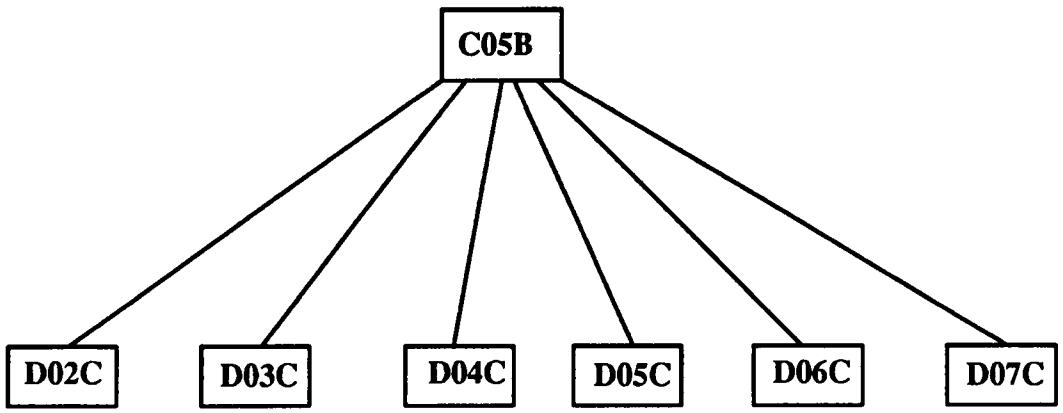
Fig A-2 init_robot()





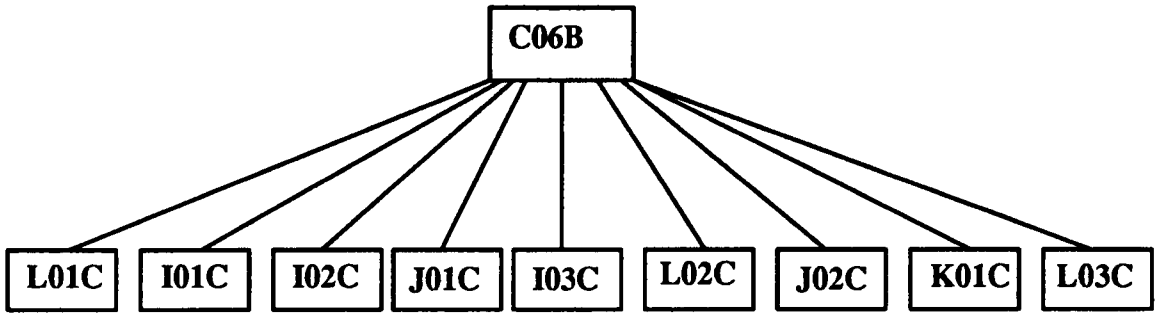
B03A check_and_process_input()
C02B process_keybrd_entry()
C03B read_mouse()
C04B check_switches()
C05B poll_left_column()
C06B teach_robot_cartesian()
C07B teach_robot_joint()
C08B poll_lower_middle_column()
C09B poll_right_column()
C10B poll_left_top_corner()
I09B draw_robot()
F03B draw_curv()
F07B final_check()

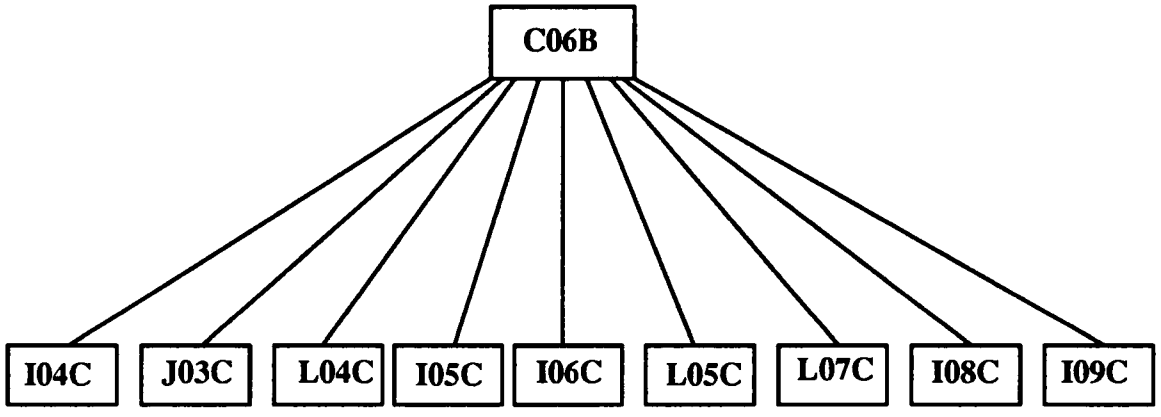
Fig A-3 check_and_process_input()



C05B poll_left_column()
D02C change_local_coord()
D03C change_view_point()
D04C poll_new_menu()
D05C dm6_button()
D06C force_sensor_option()
D07C run_robot()

Fig A-4 poll_left_column()

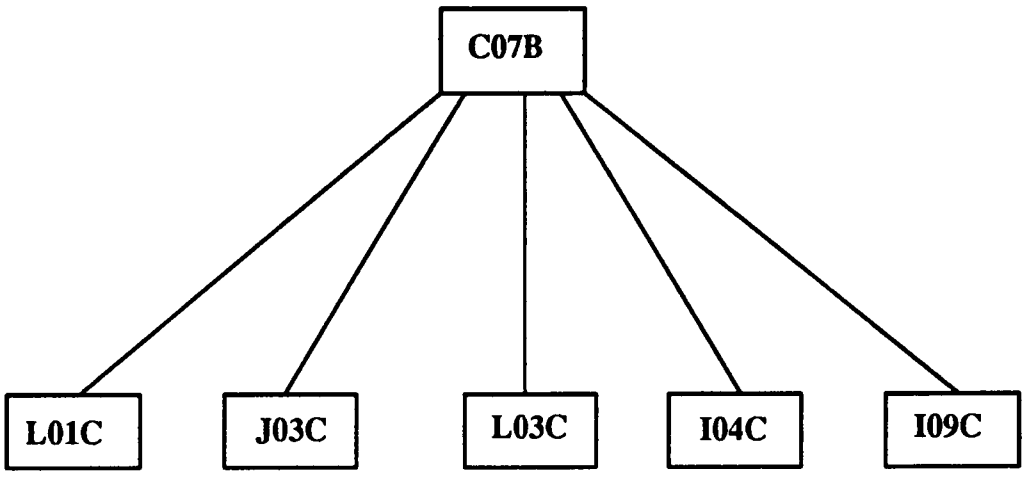




C06B teach_robot_cartesian()
L01C h_b_transf()
I01C get_position()
I02C elbow_direction()
J01C jacobian()
I03C o_motion()
L02C new_theta()
J02C difference()
K01C least_norm()
L03C check_joint_limit()

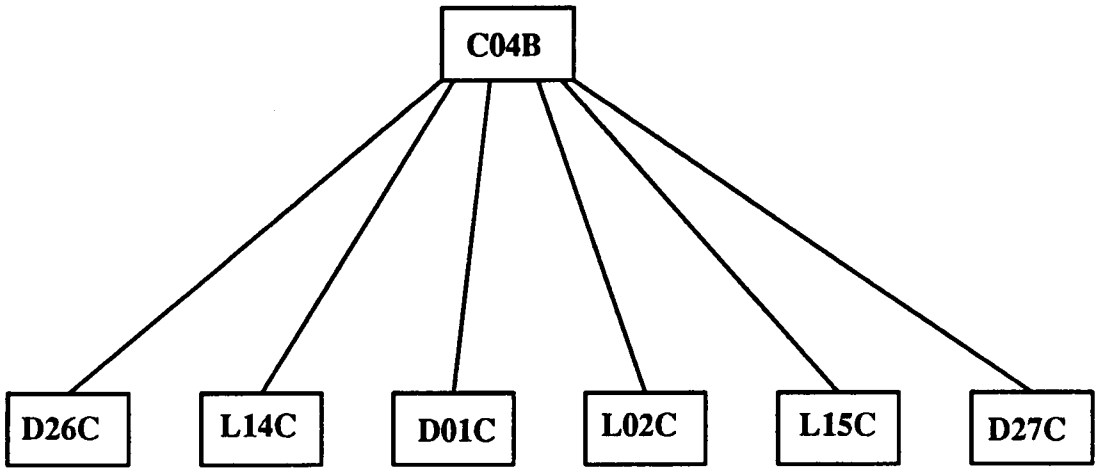
I04C check_collision()
J03C update_bit3_ctl_variable()
L04C imp_control()
I05C k_inverse()
I06C jv_max_ratio()
L05C obj_no_change()
I07C init_offset()
I08C bit3_end_vel()
I09C draw_robot()

Fig A-5 teach_robot_cartesian()



C07B teach_robot_joint()
L01C h_b_transf()
J03C update_bit3_ctl_variable()
L03C check_joint_limit()
I04C check_collision()
I09C draw_robot()

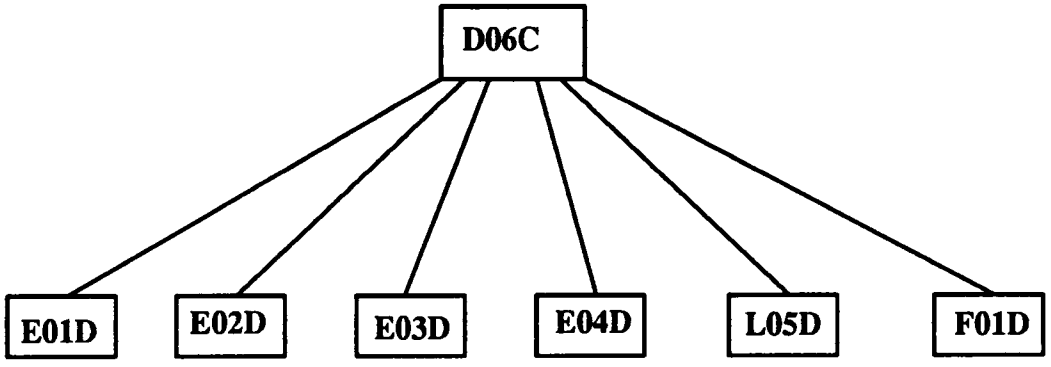
Fig A-6 teach_robot_joint()

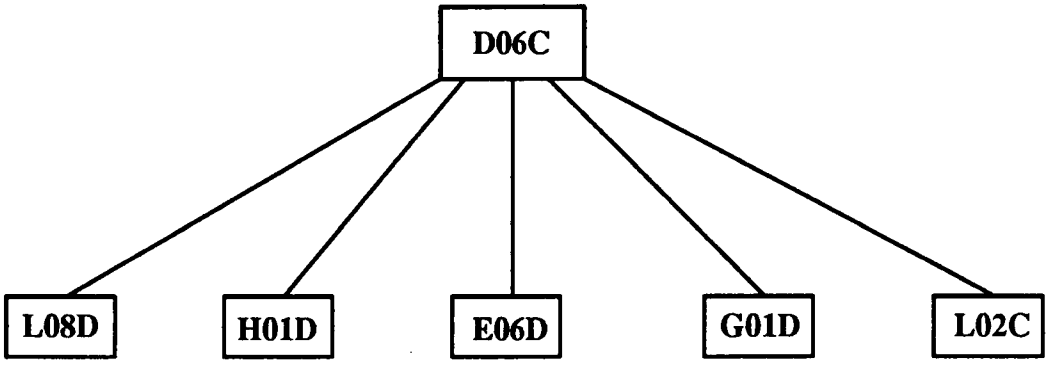


C04B check_switches()
D26C bit3_read_force()
L14C force_sensor()
D01C poll_manager()

L02C new_theta()
L15C command_master()
D27C ult_sensor()

Fig A-7 check_switches()

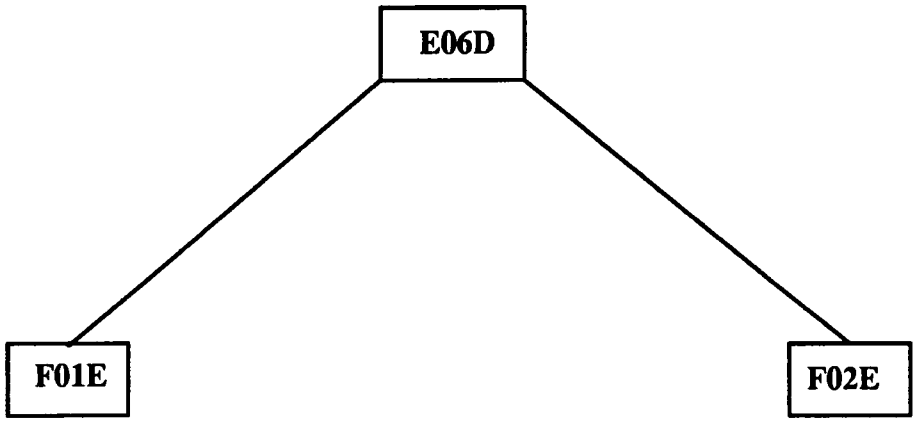




D06C run_robot()
E01D goal_gp()
E02D circular_path()
E03D goal_jnt()
E04D goal_obt()
L05D jacobian()

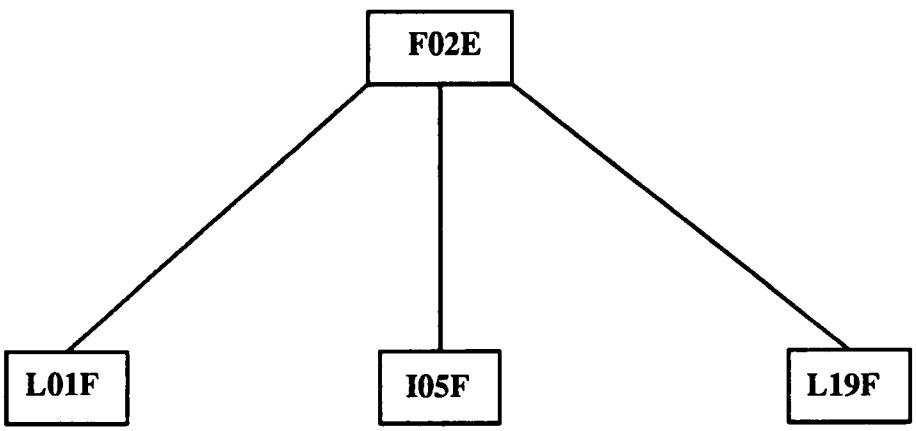
F01D goal_lin()
L08D difference()
H01D k_inverse()
E06D linear_cont()
G01D goal_rpy()
L02C draw_robot()

Fig A-8 run_robot()



E06D linear_cont()
F01E goal_lin()
F02E circle()

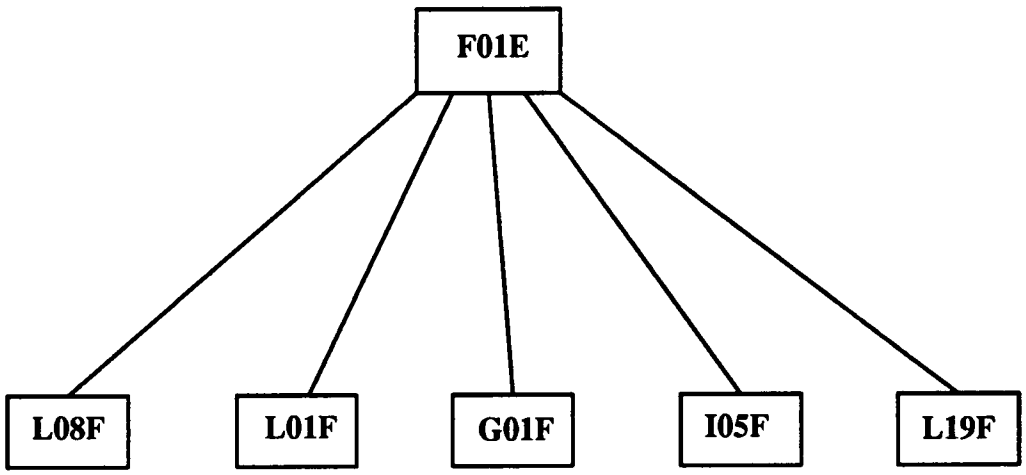
Fig A-9 linear_cont()



F02E circle()
L01F get_position()

I05F k_inverse()
L19F pause_key()

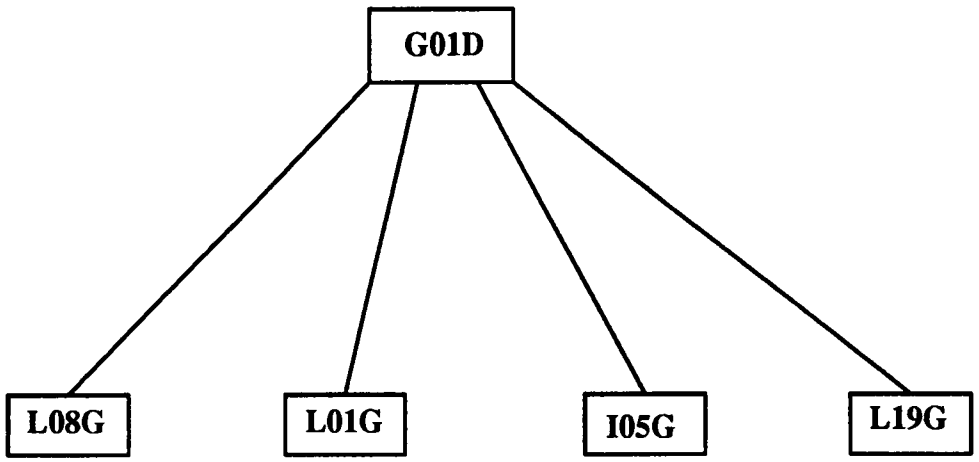
Fig A- 10 circle()



F01E goal_lin()
L08F difference()
L01F get_position()

G01F goal_rpy()
I05F k_inverse()
L19F pause_key()

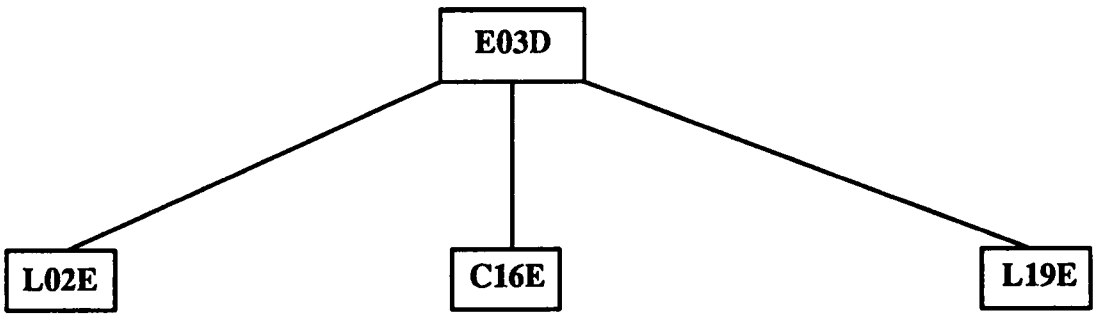
Fig A-11 goal_lin()



G01D goal_rpy()
L08G difference()
L01G get_position()

I05G k_inverse()
L19G pause_key()

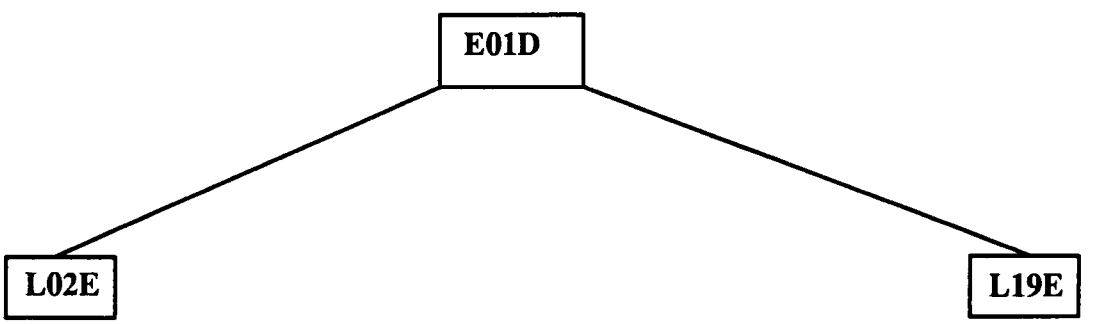
Fig A-12 goal_rpy()



E03D goal_jnt()
L02E draw_robot()

C16E draw_curv()
L19E pause_key()

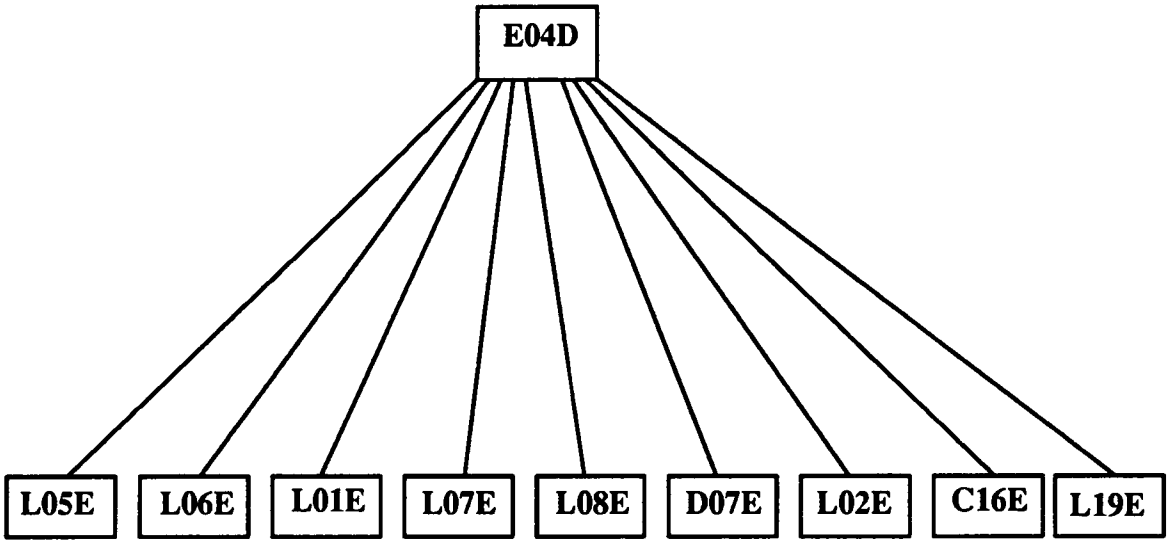
Fig A– 13 goal_jnt()



E01D goal_gp()
L02E draw_robot()

L19E pause_key()

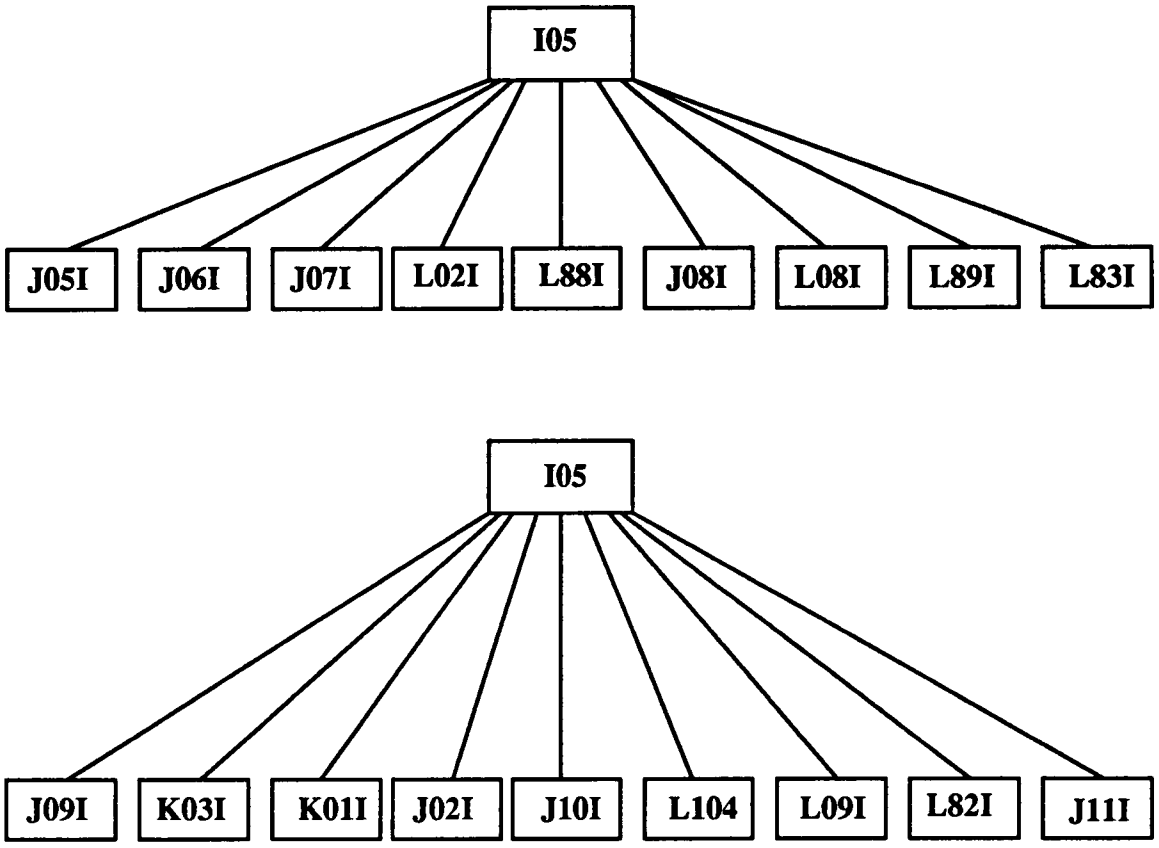
Fig A– 14 goal_gp()



E04D goal_obt()
L05E jacobian()
L06E o_motion()
L01E get_position()
L07E new_theta()

L08E difference()
D07E least_norm()
L02E draw_robot()
C16E draw_curv()
L19E pause_key()

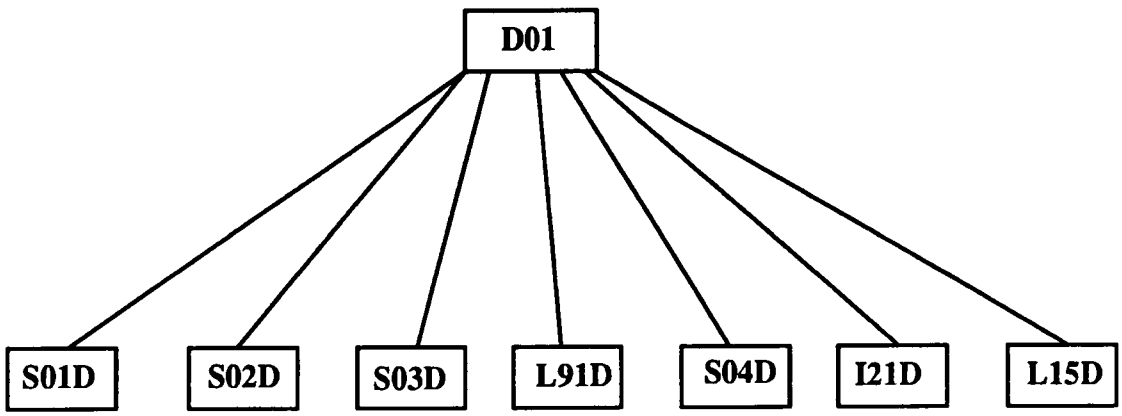
Fig A-15 goal_obt()



I05 k_inverse()
J05I bit3_end_vel()
J06I draw_curv()
J07I jlimit_weights()
L02I new_theta()
L88I get_j_lim(
J08I o_motion()
L08I determ()
L89I get_j_lim2()
L83I dist_perform()

J09I draw_robot()
K03I jacobian()
K01I least_norm()
J02I difference()
J10I get_position()
L104I cal_jjt()
L09I svd()
L82I link_vector()
J11I j_v_limit()

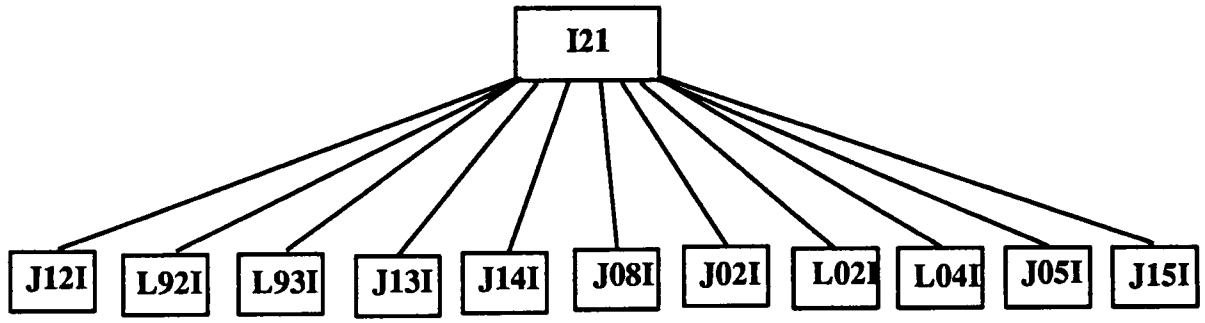
Fig A-16 k_inverse()

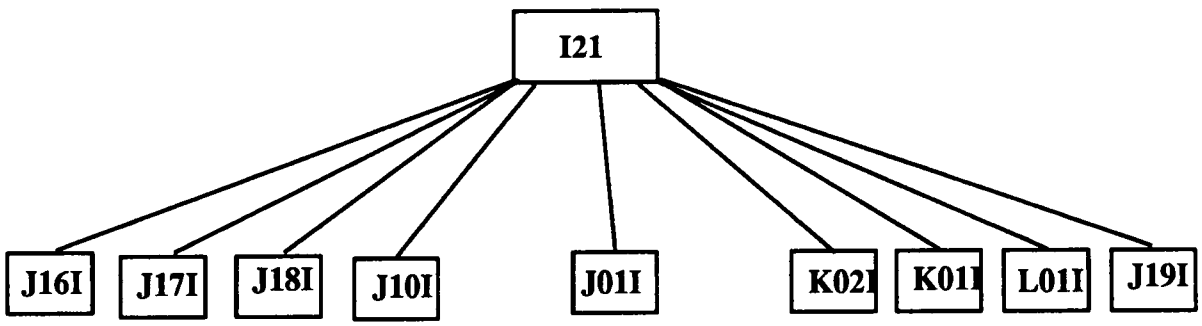


D01 poll_manager()
S01D poll()
S02D write()
S03D ioctl()
L91D command_gripper()

S04D read()
I21D master_control()
L15D command_master()

Fig A–17 poll_manager()

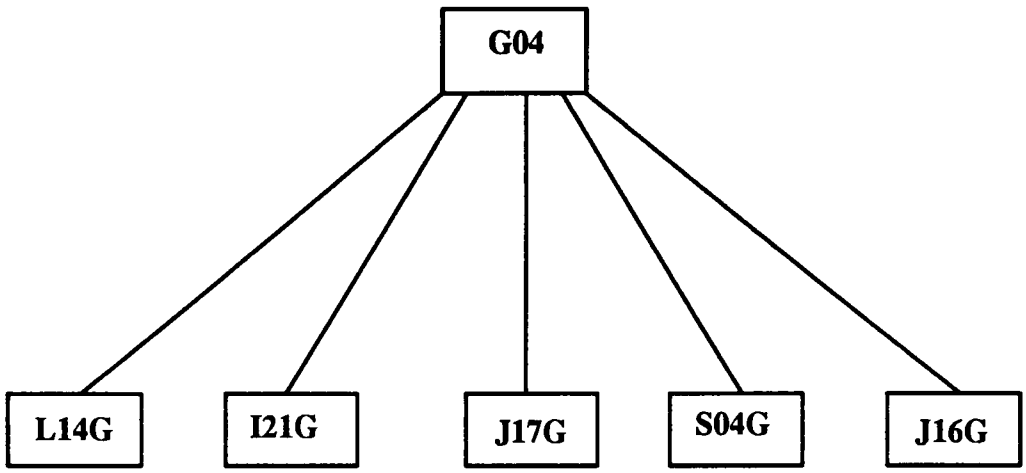




I21 master_control()
J12I init_offset()
L92I gripper_control()
L93I minimum_rot()
J13I master_force_reflection()
J14I elbow_direction()
J08I o_motion()
J02I difference()
L02I new_theta()
L04I imp_control()
J05I bit3_end_vel()
J15I use_max_jv()

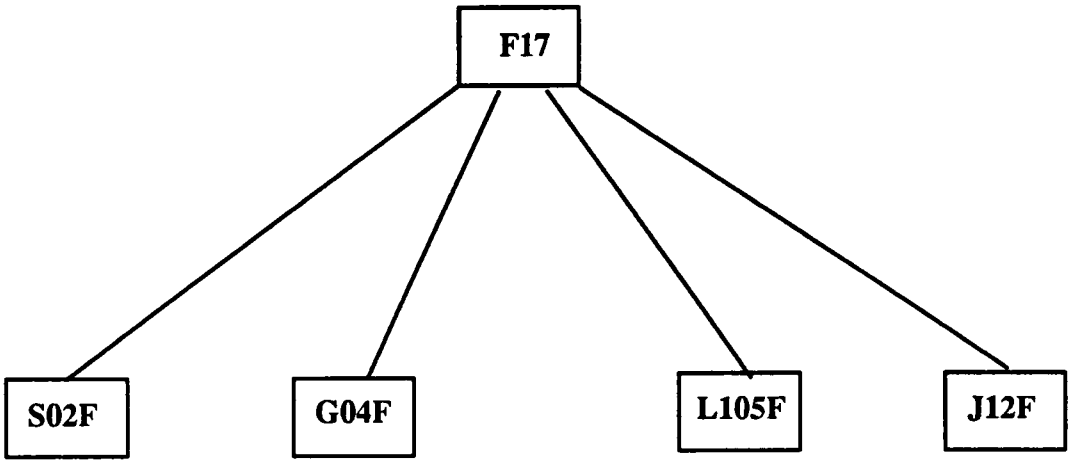
J16I msvo_to_rad()
J17I master_jacobian()
J18I m_s_error()
J10I get_position()
J01I jacobian()
K02I transf()
K01I least_norm()
L01I h_b_transf()
J19I jv_max_ratio()

Fig A-18 master_control()



G04 master_check
L14G command_master()
I21G master_control()
J17G master_jacobian()
S04G read()
J16G msvo_to_rad()

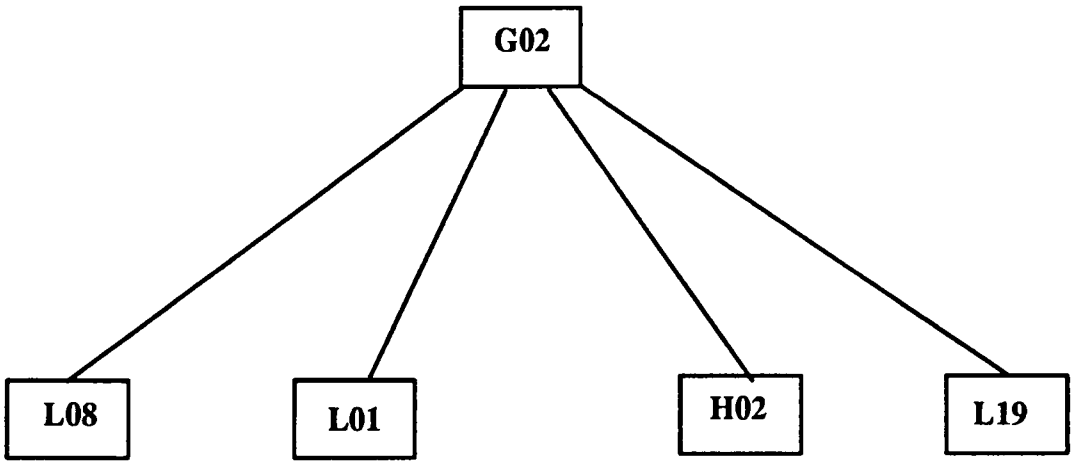
Fig A-19 master_check()



F17 talk_to_master()
S02F write()
G04F master_check()

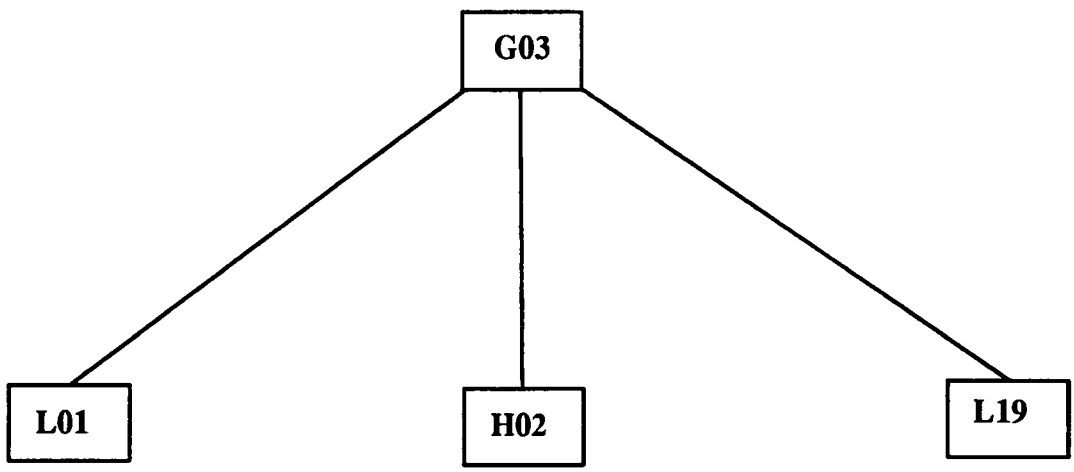
L105F change_baud()
J12F init_offset()

Fig A-20 talk_to_master()



G02 grad_lin()
L08 difference()
L01 get_position()
H02 get_grad_dat()
L19 pause_key()

Fig A-21 grad_lin()



G03 grad_circle()
L01 get_position()
H02 get_grad_dat()
L19 pause_key()

Fig A-22 grad_circle()

APPENDIX B

NEW STATIC FUNCTIONS IN main_pro.c

```
static void init_robot(void);

static void check_and_process_input(void);

static void poll_mouse(void);

static void end_program(void);

static void process_keybrd_entry(void);

static void read_mouse(void);

static void check_switches(void);

static void poll_left_column(void);

static void poll_lower_middle_column(void);

static void poll_right_column(void);

static void poll_left_top_corner(void);

static void change_local_coord(void);

static void force_sensor_option(void);

static void dm6_button(void);

static void run_robot(void);

static void choose_joint_or_cart(void);

static void teach_robot_cartesian(int m_button, float *theta_rd);

static void teach_robot_joint(int m_button,float *theta_rd);

static void control_gripper(void);

static void set_velocity_limit(void);
```

```
static void turn(void);

static void bring_robot_home(void);

static void gripper_closing_option(void);

static void record_ct_path(void);

static void record_st_path(void);

static void circle_path(void);

static void disk_files(void);

static void clear_record(void);

static void begin_record(void);

static void move_step(void);

static void delete(void);

static void ask_option(void);

static void calculate_3D(void);

static void ct_robot(void);

static void talk_to_ult_sensor(void);

static void select_manipulator(void);

static void change_swap_grafcs_time(void);

static void set_joint_vel_limit(void);

static void RT_control_gripper(void);

static void gripper_force_feedback(void);
```

```
static void talk_to_d6(void);

static void talk_to_force_sensor(void);

static void add_force_offset(void);

static int stand_motion(int m_button, float *tp_theta);

static int check_bit3(float *theta_rd);

static void available_options(void);

static void print_menu(void);

static void change_gain_factor(void);

static void select_control_scheme(void);

static void change_radius(void);

static void change_parameter(void);

static void repeat_trajectory(void);
```

APPENDIX C

HEADER FILES AND MAKEFILE

HEADER FILES

```
#ifndef BIT3PROG_MAIN_H
#define BIT3PROG_MAIN_H


void init_bit3(void);
void reset_time_delay(void);
void real_time_switch(int i);
int robot_homing(int joint);
void update_bit3_ctl_variable(void);
void bit3_end_effector_pos(float *origin);
void update_bit3_ef_old(float *origin);
void use_f_sensor(void);
void stop_f_sensor(void);
void update_f_gain(float *gain);
void update_f_offset(float *offset);
int bit3_read_force(signed short *force_buffer,float *force_ave);

#endif

#ifndef BIT3PROG_H
#define BIT3PROG_H

void bit3_end_vel(float *vels, float *slave_pos_err);

#endif

#ifndef BIT3PROG_INIT_H
#define BIT3PROG_INIT_H


void bit3_interface(float *theta_rd);


#endif

#ifndef BIT3PROG_LOCAL_H
#define BIT3PROG_LOCAL_H


#include <sys/ipc.h>
```

```
#include <sys/shm.h>
#include <sys/prctl.h>


/************** SERVO JOINT COUNTERS PER RADIAN
******************/
#define J1_CNT 365063.24
#define J2_CNT 342804.82
#define J3_CNT 208607.57
#define J4_CNT 216513.79
#define J5_CNT 104303.78
#define J6_CNT 59640.77
#define J7_CNT 62582.27


struct bit3 {
   char servo_stat;
   char homing_robot;     /* Moving to home command  */
   char homing_flag;    /* Bit set if it near a home position */
   int control_on;
   int control_mode;   /* Control mode in servo level        */
                /*   0: joint position control mode    */
                /*   1: end-effecter pos. contr. mode  */

   int change_scheme;  /* If scheme parameter is changed: 1    */
   int ask_pos;        /* If main program ask robot position: 1 */
   int command_flag;   /* Chang data from main program side    */
   int feed_back_flag; /* Chang data from sub. program side    */
   int vels_error_fb_flag; /* velocity errors feed back     */
   int rbt_vel[MAT_DIM];
   int s_index;          /*    Contr. scheme index number  */
   float s_gain;         /*   Self motion gain factor  */
   float j_weight[MAT_DIM];      /*  Joint weights           */
   float end_pos_com[5][5];    /*  command end-effector position     */
   float end_pos_fb[5][5];    /*  Real end-effector position     */
   float end_pos_old[5][5];    /*  Old end-effector position     */
   float end_vels_com[10];      /* End effector velocity commamd */
   float end_pos_err[10];      /* End effector position error */
   float dtime;
   float rbt_pos[MAT_DIM];
```

```
    float com_pos[MAT_DIM];
    float theta_new[MAT_DIM];

    int f_sen_flag;
    int f_sen_switch;          /* When using force_sensor: 1 */
    short f_sen_data[20];
    float force_gain[10];      /* Gain factor for force sensor */
    float force_offset[10];    /* offset for force sensor */
    float force_data_ave[10];  /* force average data */
    float vels_error[10];      /* velocity error feed back */
    float last_joint_vel;      /* Last joint rotation velocity */
    };

void rpy_to_h(float *origin,float h[5][5]);

#endif

#ifndef BIT3PROG_MAIN_INIT_H
#define BIT3PROG_MAIN_INIT_H


int read_rbt_position(float *theta_rd, float *vels_error_fb);



#endif



#ifndef DRAW_LTM_H
#define DRAW_LTM_H

/*
 * This file contains the function prototypes for the "draw_LTM.c"
 * file. This file is #included in "the robot.h" file.
 */

void dr_box(float,float,float,float,float,float);
void cylinder(float *);
```

```
void calc_base_vect(void);
void h_b_transf(float *theta_rd, float h[][5], float h_b[][5]);
void LTM_elbow_pos(float *, float *, float *);

#endif

#ifndef DRAW_LTM_GOAL_H
#define DRAW_LTM_GOAL_H


void link_vector(float *, float *, float *);
void dist_perform(float *, float *, float frame_vect[100][8]);

#endif

#ifndef DRAW_LTM_INIT_H
#define DRAW_LTM_INIT_H

void draw_LTM_robot(float *);
void draw_LTM_robot2(float *);
void enveron(float ,float ,float ,float );

#endif

#ifndef DRAW_LTM_LOCAL_H
#define DRAW_LTM_LOCAL_H


static void draw_LTM_axis(void);
static void draw_LTM_stand(void);
static void draw_LTM_part(void);
static void draw_LTM_uparm(void);
static void draw_LTM_lowarm(void);
static void draw_LTM_rst(void);
static void draw_LTM_grip(void);
static void draw_LTM_fg(void);
static void cylinder1(float *);
static void get_LTM_parameters2(void);
static void draw_wall(void);
static void draw_stand1(void);
```

```
static void draw_stand2(void);
static void draw_plane(void);
static void min_distance(float *, float *, float *, float *);

#endif

#ifndef DRAW_LTM_OBJECTS_H
#define DRAW_LTM_OBJECTS_H

void dr_box(float,float,float,float,float,float);
void cylinder(float *);

#endif

#ifndef DRAW_LTM_TOOLS_H
#define DRAW_LTM_TOOLS_H

void init_LTM_parameters(void);
void get_LTM_parameters(void);
void save_LTM_parameters(void);
void init_LTM_obt_struct(void);
void LTM_jacobian(float *,float t[5][5],float jc[][MAT_DIM]);
void LTM_transf(float *theta,float t[5][5]);
void init_LTM_parameters2(void);
void save_LTM_parameters2(void);
void init_LTM_obt_struct2(void);
void LTM_jacobian2(float *theta,float t[][5],float jc[][MAT_DIM]);
void LTM_transf2(float *theta,float t[][5]);

#endif

#ifndef DRAW_K2107_H
#define DRAW_K2107_H


void drawnextfig(void);
void drawsensor(void);


#endif
```

```
#ifndef DRAW_K2107_INIT_H
#define DRAW_K2107_INIT_H

void drawlink0(void);
void drawlink1(void);
void drawlink2(void);
void drawlink3(void);
void drawlink4(void);
void drawlink5(void);
void drawlink6(void);
void drawgripper(void);
void drawfig(void);
void draw_k2107_robot(float *);

#endif

#ifndef DRAW_K2107_TOOLS_H
#define DRAW_K2107_TOOLS_H

void init_k2107_parameters(void);
void get_k2107_parameters(void);
void save_k2107_parameters(void);
void init_k2107_obt_struct(void);
void k2107_jacobian(float *,float t[5][5],float jc[MAT_DIM][MAT_DIM]);
void k2107_transf(float *, float t[5][5]);
void k2107_elbow_pos(float *theta,float *elbow_dir,float *elbow_pos);

#endif

#ifndef  GOALP_H
#define  GOALP_H


void k_inverse(int draw,float scheme_gain, float *theta_rd, float *vels, float
*origin);

#endif

#ifndef GOALP_LOCAL_H
```

```
#define GOALP_LOCAL_H

static int circle(int connect,float scheme_gain,float r, float *p1, float *center, float
*omega,float turn);
static int self_motion(int ref_joint,float self_inc,float goal_rd, float *origin, float
*theta_rd);
static int linear_cont_grad(int ref_joint,int path,float r, float *x2, float *x3,int
stop);
static int grad_lin(int ref_joint,int path,float *dst);
static int grad_circle(int ref_joint,int path,float r, float *p1, float *center, float
*omega,float turn);
static void get_grad_dat(int ref_joint,int path,int steps, float *vels, float *origin);
static void gcost_data(int ref_joint,int path,int steps, float *vels, float *origin);
static void gdet_data(int ref_joint,int path,int steps, float *vels, float *origin);
static void geig_data(int ref_joint, int path,int steps,float *vels,float *origin);
static void gdet6_data(int ref_joint,int path,int steps, float *vels, float *origin);
static void get_jlim_data(int ref_joint,int path,int steps, float *vels, float *origin);
static void get_jlim_data2(int ref_joint,int path,int steps, float *vels, float *origin);
static void get_max_jv(int ref_joint,int path,int steps, float *vels, float *origin);
static int pause_key(void);

#endif

#ifndef GOALP_MAIN_H
#define GOALP_MAIN_H

int goal_lin(float scheme_gain,float *dst);
int goal_rpy(float scheme_gain,float *dst);
int goal_jnt(int n,float *dst);
int goal_gp(float *theta_rd,float gp);
int goal_obt(int n, float *dst);
int linear_cont(float scheme_gain,float r, float *x2, float *x3,int stop);
int circular_path(float scheme_gain, float *x2, float *x3,char offset,char cycle);
int grad_data(int ref_joint,float self_inc,int total_path,float r,int s_count,int
goal_steps);
void home_robot(void);  /* goalp.c , main_pro.c */


#endif
```

```
#ifndef GRIPPER_H
#define GRIPPER_H

#include <ctype.h>

void gripper_control(float *theta_rd);
void init_gripper(void);

#endif

#ifndef GRIPPER_TOOLS_H
#define GRIPPER_TOOLS_H

void command_gripper(void);

#endif

#ifndef INIT_GRAPH_H
#define INIT_GRAPH_H




void fm_print(void);
void draw_curv(void);
void draw_robot(float *);
void draw_all(void);

#endif

#ifndef INIT_GRAPH_LOCAL_H

#define INIT_GRAPH_LOCAL_H

void change_view2(float *view_point2);
void def_light(void);
void use_light(void);
void def_light2(void);
```

```
void use_light2(void);
void view_butn(void);
void change_view(void);
void c_view(float *);
void drawaxis(void);
void draw_grad(int, int);
void grad_surf(int,int,int,int);
void least_curv(int, int);
void c_path(int ,float [TSTEPS][3]);
void show_paths(int,int);
void show_steps(int , int );
void show_data(int,int);
void fourth_ellipse(void);
void ellipse(float,float,float);
void use_camera(int,float *,float,float,float,float [][5],float [][5]);

#endif

#ifndef INIT_MAIN_H
#define INIT_MAIN_H

void poll_new_menu(void);
void show_end_effectr(void);
void change_viewpoint(void);
void change_color(void);
void init_rob_win(void);
void draw_teach_pendant(int,int, int);
void home_teach_pendant(char);
void init_curv_win(void);
void grad_plot(int, int);
void n_cal(int,int, int);
void final_check(float *);
void c_view_cont(float *);
int select_camera(void);


#endif

#ifndef MAIN_PRO_LOCAL_H
#define MAIN_PRO_LOCAL_H
```

```
static void init_robot(void);
static void check_and_process_input(void);
static void poll_mouse(void);
static void end_program(void);
static void process_keybrd_entry(void);
static void read_mouse(void);
static void check_switches(void);
static void poll_left_column(void);
static void poll_lower_middle_column(void);
static void poll_right_column(void);
static void poll_left_top_corner(void);
static void change_local_coord(void);
static void force_sensor_option(void);
static void dm6_button(void);
static void run_robot(void);
static void choose_joint_or_cart(void);
static void teach_robot_cartesian(int m_button, float *theta_rd);
static void teach_robot_joint(int m_button,float *theta_rd);
static void control_gripper(void);
static void set_velocity_limit(void);
static void turn(void);
static void bring_robot_home(void);
static void gripper_closing_option(void);
static void record_ct_path(void);
static void record_st_path(void);
static void circle_path(void);
static void disk_files(void);
static void clear_record(void);
static void begin_record(void);
static void move_step(void);
static void delete(void);
static void ask_option(void);
static void calculate_3D(void);
static void ct_robot(void);
static void talk_to_ult_sensor(void);
static void select_manipulator(void);
static void change_swap_grafcs_time(void);
static void set_joint_vel_limit(void);
static void RT_control_gripper(void);
```

```
static void gripper_force_feedback(void);
static void talk_to_d6(void);
static void talk_to_force_sensor(void);
static void add_force_offset(void);
static int stand_motion(int m_button, float *tp_theta);
static int check_bit3(float *theta_rd);
static void available_options(void);
static void print_menu(void);
static void change_gain_factor(void);
static void select_control_scheme(void);
static void change_radius(void);
static void change_parameter(void);
static void repeat_trajectory(void);

#endif

#ifndef MAIN_PRO_TOOLS_H
#define MAIN_PRO_TOOLS_H

void rec_position(int index,int state,int steps,float *theta_rd, float h_transf[5][5]);
void select_robot(int robot_index);
void save_parameters(int robot_index);
void init_parameters(void);
void send_command(int ser_port,char *command);
void init_comm3(void);
void init_comm5(void);
void init_comm6(void);
void init_comm7(void);
void ult_sensor(void);
void force_sensor(signed short *force_buffer);
void pop_cursor(void);
int check_joint_limit(float *theta_rd);

#endif

#ifndef  MASTER_CTL_H
#define  MASTER_CTL_H


void command_master(void);
```

```
#endif

#ifndef MASTER_CTL_LOCAL_H
#define MASTER_CTL_LOCAL_H

int master_check(float a[5][5]);
void msvo_to_rad(short *joint, char *buffer, float *theta);
void m_s_error(float *err);
void mult_3x3(int flag,float matrix1[5][5],float matrix2[5][5],float matrix3[5][5]);
void master_cursor(char *buffer);
void change_baud(int ser_port_master,int baud);
void master_jacobian(float *theta,float a[5][5],float jc[MAT_DIM][MAT_DIM]);
void master_force_reflection(float jc[MAT_DIM][MAT_DIM]);


#endif

#ifndef MASTER_CTL_MAIN_H
#define MASTER_CTL_MAIN_H

int talk_to_master(void);
void init_offset(void);
void init_comm_mast(void);


#endif

#ifndef MASTER_CTL_TOOLS_H
#define MASTER_CTL_TOOLS_H


void master_control(char *buffer);

#endif

#ifndef OBJECTS_H
#define OBJECTS_H
```

```
int check_pick_up(float *theta_rd);
void imp_control(int pick_up,float h[][5],float h_b[][5], float *vels,float *theta_rd,
float *obj_inc);
int check_collision(float *theta_rd);

#endif

#ifndef OBJECTS_INIT_H
#define OBJECTS_INIT_H

void draw_object1(void);
void draw_object2(void);
void draw_object3(void);
void draw_object4(void);
void draw_object5(void);

#endif

#ifndef OBJECTS_LOCAL_H
#define OBJECTS_LOCAL_H


struct obj_struct{
        int   obj_constrain[7];
        float  obj_position[7];
        float  home[7];
};

int check_pick_object1(float *theta_rd,float h_b[][5]);
int check_pick_object2(float *theta_rd,float h_b[][5],float h_transf[][5],float
*obj_position,float *handle);
int check_pick_object(float *theta_rd,float h_b[][5],float h_transf[][5],float
*obj_position,float *handle);

#endif

#ifndef OBJECTS_MAIN_H
#define OBJECTS_MAIN_H
```

```
void get_object1(void);
void get_object2(void);
void get_object3(void);
void get_object4(void);
void get_object5(void);
void obj_no_change(int pick_up, float *obj_inc);



#endif

#ifndef _ROBOT_H

#define _ROBOT_H

#include <sys/types.h>
#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gl/gl.h>
#include <gl/device.h>
#include <termio.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <poll.h>
#include <stropts.h>
#include <fmclient.h>
#include <sys/time.h>
#include <device.h>
#include <gl.h>

#define TPATHS 100              /* Total paths for 3-D Map plotting      */
#define TSTEPS 2000             /* The max. steps of trajectory          */
#define MAT_DIM 12              /* The dimension of arrays or matrices   */
#define T_JOINT 7
```

```
/* Total joints of the manipulator        */

#include "init_graph.h"
#include "draw_LTM.h"
#include "draw_k2107.h"
#include "tools.h"
#include "objects.h"
#include "goalp.h"
#include "master_ctl.h"
#include "bit3prog.h"
#include "robot_math.h"
#define IRIX

#define SIN(X) sin(X)
#define COS(X) cos(X)

#define LEFT_COLUMN_TP      (1020<=mx) && (mx<=1090)
#define RIGHT_COLUMN_TP     (1200<=mx) && (mx<=1270)
#define MIDDLE_COLUMN_TP    (1100<=mx) && (mx<=1180)
#define TOP_LEFT_CORNER     (mx>=10) && (mx<=sub_winx) &&
(my>=sub_winy) && (my<=1000)
#define FROM_C1_TO_C7_BUTTON  (420<=my) && (my<=950)
#define VIEW_BUTTON         (my>=340) && (my<=390)
#define DM6_BUTTON          (180<=my) && (my<=230)
#define F_SEN_BUTTON        (100<=my) && (my<=150)
#define RUN_BUTTON          (20<=my) && (my<=70)
#define TOP_LEFT_BUTTON     (my>=980) && (my<=1000)
#define TOP_MIDDLE_BUTTON   (980<=my) && (my<=1000)
#define FROM_X_TO_ORBIT_OR_J1_TO_J7_BUTTON  (my>=420) &&
(my<=950)
#define GP_BUTTON           (340<=my) && (my<=390)
#define LIN_V_OR_J_V_BUTTON  (my>=260) && (my<=310)
#define TURN_BUTTON         (180<=my) && (my<=230)
#define HOME_BUTTON         (my>=100) && (my<=150)
#define OPEN_CLS_OR_POS_POS_BUTTON  (980<=my) && (my<=1000)
#define REC_CT_BUTTON       (900<=my) && (my<=950)
#define REC_ST_BUTTON       (820<=my) && (my<=870)
#define C_PATH_BUTTON       (740<=my) && (my<=790)
#define FILES_BUTTON        (660<=my) && (my<=710)
#define CLEAR_BUTTON        (580<=my) && (my<=630)
```

```
#define BEGIN_BUTTON        (500<=my) && (my<=550)
#define STEP_BUTTON         (420<=my) && (my<=470)
#define DELETE_BUTTON       (340<=my) && (my<=390)
#define OPTION_BUTTON       (260<=my) && (my<=310)
#define BUTTON_3D_CAL       (180<=my) && (my<=230)
#define CT_RBT_BUTTON       (my>=100) && (my<=150)
#define END_BUTTON          (20<=my) && (my<=70)
#define ORBIT_BUTTON        (my>=420) && (my<=470)
#define X_BUTTON           (my>=900) && (my<=950)
#define Y_BUTTON           (my>=820) && (my<=870)
#define Z_BUTTON           (my>=740) && (my<=790)
#define ROLL_BUTTON         (my>=660) && (my<=710)
#define PITCH_BUTTON        (my>=580) && (my<=630)
#define YAW_BUTTON          (my>=500) && (my<=550)

struct obt                  /* Self motion control structure          */
   {int index;              /* The  max. joint velocity joint number  */

    int sign;               /* Self motion signal                     */
                            /*  1 --- forward motion                  */
                            /* -1 --- back motion                     */

    float obt_v;            /* The max. joint velocity for self motion */
};


struct gt                   /* The structure for recording the        */
                            /* trajectory data and motion type        */
   {char st_con[3];         /* Motion type recorder                   */
                            /* st_con[0]: motion type                 */
                            /* st_con[1]: continue or stop signal     */
                            /*    0: Do not use circular path planning */
                            /*    1: Use circular path planning for   */
                            /*       smooth transation between two    */
                            /*       linear paths                     */
                            /* st_con[2]: cycles for a circular motion */
                            /*    0: Less than 360 degree             */
                            /*  >=1: cycles for circular motion       */

   float goal_position[MAT_DIM];   /* Trajectory data of joint revolutions or */
```

```
                    /* the end-effector position          */
};
#endif

#ifndef ROBOT_MATH_H
#define ROBOT_MATH_H


void end_vels(float theta_vel[],float jc[MAT_DIM][MAT_DIM],float vels[]);
void determ(int n,float d[MAT_DIM][MAT_DIM]);
int equit(int n ,float d[MAT_DIM][MAT_DIM] ,float *x);
void cal_jjt(int m,int n,float jaco[MAT_DIM][MAT_DIM],float
ans[MAT_DIM][MAT_DIM]);
void svd(int n, float a[MAT_DIM][MAT_DIM],float *eigenvalues);
void new_theta(float *theta,float *dtheta);
void rpy_angles(float h_transf[5][5]);
void get_position(float *origin,float h_transf[5][5]);
void difference(float *theta_rd,float h_transf[5][5],float *goal, float *diff);
void minimum_rot(float h_transf[][5],float h[][5],float *diff);
void pop_cursor(void);

#endif

#ifndef TOOLS_H
#define TOOLS_H

void end_vels(float [],float jc[MAT_DIM][MAT_DIM],float vels[]);
void cal_jjt(int m,int n,float jaco[MAT_DIM][MAT_DIM],float
ans[MAT_DIM][MAT_DIM]);
int least_norm(float jacob[][MAT_DIM], float *dx,float  *dtheta);
int o_motion(float jacob[][MAT_DIM],float *dtheta);
void elbow_direction(float *theta_rd);
void svd(int n, float a[MAT_DIM][MAT_DIM],float *eigenvalues);
void new_theta(float *theta,float *dtheta);
void rpy_angles(float h_transf[5][5]);
void get_position(float *origin,float h_transf[5][5]);
void jlimit_weights(float *theta_rd);
void jv_max_ratio(float *dtheta,float *max_ratio);
void use_max_jv(float *dtheta,float ratio);
void difference(float *theta_rd,float h_transf[5][5],float *goal, float *diff);
```

```
void minimum_rot(float h_transf[][5],float h[][5],float *diff);
void jacobian(float *theta,float h_transf[5][5],float
jaco[MAT_DIM][MAT_DIM]);
void transf(float *theta,float h_transf[5][5]);
int feed_back(long time_delay,char *buffer);
void poll_manager(int control,int *read_index);


#endif

#ifndef TOOLS_GOALP_H
#define TOOLS_GOALP_H

void determ(int n,float d[MAT_DIM][MAT_DIM]);
int equit(int n ,float d[MAT_DIM][MAT_DIM] ,float *x);
int j_v_limit(float *dtheta_m, float *dtheta_h,float gain, float *dtheta);
void init_obt_struct(int robot_index);
void get_j_lim(float *j_lim_h, float *theta_rd);
void get_j_lim2(float *j_lim_h, float *theta_rd);


#endif
```

MAKEFILE

```
CC = cc
INCLUDE_DIR = ../include
ROBOTLIB_DIR = ../lib
CFLAGS = $(DEBUG) -I$(INCLUDE_DIR)
DEBUG =
LIBS =  -lrobot -lfm_s -lgl_s -lm


include make_include

all::robot

robot:$(OBJS) ../lib/librobot.a
        @echo "linking..."
        $(CC) $(CFLAGS) -o $@ -L$(ROBOTLIB_DIR) $(OBJS) $(LIBS)
        cp robot ../robot

tools.o: $(TOOLS_HDRS)
goalp.o: $(GOALP_HDRS)
bit3prog.o: $(BIT3PROG_HDRS)
main_pro.o: $(MAIN_PRO_HDRS)
gripper.o: $(GRIPPER_HDRS)
master_ctl.o: $(MASTER_CTL_HDRS)
objects.o: $(OBJECTS_HDRS)
        $(CC) $(CFLAGS) $(DEBUG)  -c $*.c

$(INCLUDE_DIR)/robot.h:$(HDRS)
        touch $(INCLUDE_DIR)/robot.h

spbt3:
        @echo making "spbt3"
        $(CC) -o $@ spbt3.c -lm
        cp spbt3 ../.
clean:
        rm -f $(OBJS) robot spbt3
```

make_include

```
OBJS = tools.o goalp.o\
    bit3prog.o main_pro.o gripper.o master_ctl.o objects.o


HDRS = $(INCLUDE_DIR)/draw_LTM.h $(INCLUDE_DIR)/draw_k2107.h
$(INCLUDE_DIR)/init_graph.h $(INCLUDE_DIR)/tools.h $(INCLUDE_DIR)/goalp.h\
    $(INCLUDE_DIR)/bit3prog.h $(INCLUDE_DIR)/master_ctl.h
$(INCLUDE_DIR)/objects.h


TOOLS_HDRS = $(INCLUDE_DIR)/robot.h  $(INCLUDE_DIR)/draw_LTM_tools.h
$(INCLUDE_DIR)/draw_k2107_tools.h \
    $(INCLUDE_DIR)/gripper_tools.h $(INCLUDE_DIR)/master_ctl_tools.h
$(INCLUDE_DIR)/main_pro_tools.h \
    $(INCLUDE_DIR)/tools_goalp.h

GOALP_HDRS = $(INCLUDE_DIR)/robot.h $(INCLUDE_DIR)/goalp_local.h
$(INCLUDE_DIR)/goalp_main.h $(INCLUDE_DIR)/draw_LTM_goalp.h\
      $(INCLUDE_DIR)/tools_goalp.h

BIT3PROG_HDRS = $(INCLUDE_DIR)/robot.h $(INCLUDE_DIR)/bi3prog_main.h
$(INCLUDE_DIR)/bit3prog_init.h $(INCLUDE_DIR)/bit3prog_local.h \
      $(INCLUDE_DIR)/bit3prog_main_init.h

MAIN_PRO_HDRS = $(INCLUDE_DIR)/robot.h $(INCLUDE_DIR)/main_pro_local.h
$(INCLUDE_DIR)/init_main.h\
      $(INCLUDE_DIR)/master_ctl_main.h $(INCLUDE_DIR)/goalp_main.h
$(INCLUDE_DIR)/bit3prog_main_init.h\
      $(INCLUDE_DIR)/bi3prog_main.h $(INCLUDE_DIR)/gripper_main.h
$(INCLUDE_DIR)/objects_main.h $(INCLUDE_DIR)/main_pro_tools.h

GRIPPER_HDRS = $(INCLUDE_DIR)/robot.h $(INCLUDE_DIR)/gripper_main.h
$(INCLUDE_DIR)/gripper_tools.h

MASTER_CTL_HDRS = $(INCLUDE_DIR)/robot.h
$(INCLUDE_DIR)/master_ctl_local.h $(INCLUDE_DIR)/master_ctl_main.h \
       $(INCLUDE_DIR)/master_ctl_tools.h

OBJECTS_HDRS = $(INCLUDE_DIR)/robot.h $(INCLUDE_DIR)/objects_init.h
$(INCLUDE_DIR)/objects_local.h $(INCLUDE_DIR)/objects_main.h\
    $(INCLUDE_DIR)/draw_LTM_objects.h
```

APPENDIX D

EXAMPLE OF PORTING

```
#define IRIX

/* put platform independent header files here */

#if  defined   IRIX

 /* put header files available on IRIX ( Silicon Graphics)  */

#elif defined SUN

 /* put header files available on Sun  */

#endif

void draw_k2107_robot(float *theta_rd)
{

/* put general information here */

 #if  defined   IRIX

   /* put SiliconGraphics related code here */

#elif defined SUN

  /* put Sun related code here */

#endif

}
```

APPENDIX E

FUNCTION LIST

A main()

B01 init_robot()

B02 init_bit3()

B03 check_and_process_input()

B04 end_program()

C01 select_robot()

C02 process_keybrd_entry()

C03 read_mouse()

C04 check_switches()

C05 poll_left_column()

C06 teach_robot_cartesian()

C07 teach_robot_joint()

C08 poll_lower_middle_column()

C09 poll_right_column()

C10 poll_left_top_corner()

C11 init_rob_win()

D01 poll_manager()

D02 change_local_coord()

D03 change_viewpoint()

D04 poll_new_menu()

D05 dm6_button()

D06 force_sensor_option()

D07 run_robot()

D08 choose_joint_or_cart()

D09 control_gripper()

D10 set_velocity_limit()

D11 turn()

D12 bring_robot_home()

D13 gripper_closing_option()

D14 record_ct_path()

D15 record_st_path()

D16 circle_path()

D17 disk_files()

D18 clear_record()

D19 begin_record()

D20 move_step()

D21 delete()

D22 ask_option()

D23 calculate_3D()

D24 ct_robot()

D25 grad_plot()

D26 bit3_read_force()

D27 ult_sensor()

E01 goal_gp()

E02 circular_path()

E03 goal_jnt()

E04 goal_obt()

E05 linear_cont()

E06 grad_data()

E07 home_robot()

E08 bit3_end_effector_pos()

E09 available_options()

E10 change_view()

E11 change_view2()

E12 draw_grad()

F01 goal_lin()

F02 circle()

F03 change_gain_factor()

F04 self_motion()

F05 linear_cont_grad()

F06 init_obt_struct()

F07 final_check()

F08 select_control_scheme()

F09 change_radius()

F10 change_parameter()

F11 show_end_effector()

F12 repeat_trajectory()

F13 change_color()

F14 select_manipulator()

F15 change_swap_graphcs_limit()

F16 set_joint_vel_limit()

F17 talk_to_master()

F18 RT_control_gripper()

F19 gripper_force_feedback()

G01 goal_rpy()

G02 grad_lin()

G03 grad_circle()

G04 master_check()

G05 view_butn()

H01 get_grad_dat()

H02 get_grad_dat()

I04 check_collision()

I05 k_inverse()

I10 gcost_data()

I11 gdet_data()

I12 geig_data()

I13 gdet6_data()

I14 get_jlim_data()

I15 get_jlim_data2()

I16 get_max_jv()

I20 reset_time_delay()

I21 master_control()

J01 jacobian()

J02 J14()

J03 update_bit3_ctl_variable()

J04 draw_robot()

J05 bit3_end_vel()

J06 draw_curv()

J07 jlimit_weights()

J08 o_motion()

J09 draw_robot()

J10 get_position()

J11 j_v_limit()

J12 init_offset()

J13 master_force_reflection()

J14 elbow_direction()

J15 use_max_jv()

J16 msvo_to_rad()

J17 master_jacobian()

J18 m_s_error

J19 jv_max_ratio()

K01 least_norm()

K02 transf()

K03 jacobian()

K04 draw_k2107_robot()

K05 draw_LTM_robot()

K06 draw_LTM_robot2()

L01 h_b_transf()

L02 new_theta()

L03 check_joint_limit()

L04 imp_control()

L05 obj_no_change()

L06 equit()

L07 rpy_angles()

L08 determ()

L09 svd()

L10 init_comm3()

L11 init_comm_mast()

L12 init_comm5()

L13 init_comm6()

L14 init_comm7()

L15 command_master()

L16 def_light()

L17 use_light()

L18 def_light2()

L19 use_light2()

L20 show_end-effector()

L21 draw_teach_pendant()

L22 home_teach_pendant()

L23 fm_print()

L24 view_button()

L25 drawaxis()

L26 graw_grad()

L27 grad_surf()

L28 n_cal()

L29 least_curv()

L30 c_path()

L31 show_paths()

L32 show_steps()

L33 show_data()

L34 fourth_ellipse()

L35 use_camera()

L36 draw_link0()

L37 draw_link1()

L38 draw_link2()

L39 drawlink3()

L40 drawlink4()

L41 drawlink5()

L42 drawlink6()

L43 drawsensor()

L44 drawgripper()

L45 drawfig()

L46 drawnextfig()

L47 init_K2107_parameters()

L48 get_k2107_parameters()

L49 save_k2107_parameters()

L50 init_k2107_obt_struct()

L51 k2107_jacobian()

L52 k2107_transf()

L53 k2107_elbow_pos()

L54 dr_box()

L55 cylinder1()

L56 cylinder()

L57 draw_LTM_axis()

L58 draw_LTM_stand()

L59 draw_LTM_part()

L60 draw_LTM_uparm()

L61 draw_LTM_lowarm()

L62 draw_LTM_rst()

L63 draw_LTM_grip()

L64 draw_LTM_fg()

L65 init_LTM_parameters()

L66 get_LTM_parameters()

L67 save_LTM_parameters()

L68 init_LTM_obt_struct()

L69 LTM_jacobian()

L70 LTM_transf()

L71 init_LTM_parameters2()

L72 get_LTM_parameters2()

L73 save_LTM_parameters2()

L74 init_LTM_obt_struct2()

L75 LTM_jacobian2()

L76 LTM_transf2()

L77 draw_wall()

L78 draw_stand1()

L79 draw_stand2()

L80 draw_plane()

L81 draw_frame()

L82 link_vector()

L83 dist_perform()

L84 min_distance()

L85 calc_base_vect()

L86 drawlink1()

L87 LTM_elbow_pos()

L88 get_j_lim()

L89 get_j_lim2()

L90 real_time_switch()

L91 command_gripper()

L92 gripper_control()

L93 minimum_rot()

L94 drawlink2()

L95 draw_object1()

L96 draw_object2()

L97 draw_object3()

L98 draw_object4()

L99 draw_object5()

L100 end_vels()

L101 bit3_interface()

L102 read_rbt_position()

L103 init_curv_win()

L104 cal_jjt()

L105 change_baud()

L105 change_baud()

S01 poll

S02 write

S03 ioctl

S04 read

VITA

Anjanjyoti Das was born in India in 1963. He obtained his bachelor's degree in Mechanical Engineering from S. V. Regional College of Engineering and Technology (Surat), India.

Mr. Das joined the University of Tennessee, Knoxville in 1992. Now after his graduation, he intends to pursue a career in Software Engineering.