A Proposal for the Concurrent Hierarchical Control of Robot Systems

John Gruber Waller

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Computer Science at Concordia University
Montréal, Québec, Canada

September 1985

ISBN 0-315-30625-4

ABSTRACT

A Proposal for the Concurrent Hierarchical Control of Robot Systems

John Gruber Waller

A generalized robot control system referred to as the cross-coupled processing hierarchy has been proposed by the National Bureau of Standards, Washington DC. The NBS proposal is reviewed in this thesis and an enhanced proposal referred to as Concurrent Hierarchical Control is presented. Concurrent Hierarchical Control is based on design principles which are common to real-time computer operating systems.

Table of Contents

1. Introduction

The structuring of robotics software is becoming increasingly important, as both control and sensory functions in contemporary robots grow in complexity. While early robot software was essentially derived from numerical control techniques used in open-loop precision machine tools, the programming of complex task sequences, the handling of sensory information, and the need for exception processing has placed greater demands on robot control system design

This increased need for flexibility requires a hierarchical structure analogous to the layered structures of contemporary operating systems such as IBM-MVS or Unix. There are however, significant differences between the performance goals of a general purpose operating system and robot control software Robot software does not require many of the resource management functions of an operating system, but, conversely imposes constraints defined in real-time by its functions and its coordination with external events. In particular, the management of sensory feedback and the execution of emergency procedures that may arise suggest the need for a communications function that is significantly different from operating system requirements. In this way, it resembles process control coupled with sophisticated programming techniques.

A hierarchically organized computer or process-control system usually suggests that the hierarchy is based on several levels of abstraction. The lower levels take care of simple routines, such as device drivers and buffering, and the upper levels take care of higher functions, such as resource management policies and communications protocols. Levels of abstraction are closely linked, the very term suggests that higher levels are merely distilled versions of lower ~~levels~~ levels.

However, another approach to robot control systems is the concept of levels of understanding. This suggests that lower levels may be equipped with their own intelligence to make decisions independently of the higher levels. It is often helpful to take a human situation in order to exemplify the point being made. Let us say that a person has just picked up a cup which is too hot to the touch. When the cup has been grasped, the person quickly releases the cup and withdraws his hand. It is only after the event that the person in fact understands what has taken place, the action itself has been a reflex. Nonetheless, a high priority signal has been sent to the brain, i.e., a pain signal. The pain signal allows the brain to analyze the events after the fact and choose subsequent actions, e.g., treat the burn and clean up the cup's spilt contents.

In a system organized by levels of abstraction, we may consider all decision-making authority originates at the top. By contrast, in our example the higher centres of the brain have not been involved in the decision to withdraw the hand from the cup. In a system organized by levels of understanding, we may consider such authority to be much more distributed throughout the system. Nevertheless, the higher centres may have an override ability. Our human subject may be able to override the reflex to release the hot cup if he strongly does not wish to spill its contents. However, he must deliberately choose to override the reflex action which has been communicated to him by the pain signal. This would be a learned response. The pain signal provides a key to understanding the system, i.e., communication between levels of understanding is accomplished through messages, which implies that message-based systems may provide significant results for robot control requirments.

For several years, the National Bureau of Standards under the leadership of J.S. Albus and A.J. Barbera has been doing research in the area of generic robot control systems which address the issues typical of the example outlined above. The result of their work has been the development of the 'cross-coupled processing hierarchy' system which is the first real attempt at developing what might be called a general-purpose robot operating system. In this thesis an analysis of this work is presented as well as a proposal referred to as Concurrent Hierarchical Control. This seeks to enhance the processing throughput of the cross-coupled processing hierarchy by applying well-accepted principles of structure and development found in modern operating system design. In so doing, it was found that processing throughput could be substantially improved by approximately one order of magnitude, with the added benefit of providing a more user-friendly environment and therefore greater programmer productivity. A work bench approach has been adopted which allows application flexibility and dynamic reconfiguration as opposed to a single monotholic application.

A brief description of the organization of this thesis follows. Chapter two provides several examples of computer-controlled robot systems which are more sophisticated than the simple open-loop playback robots which are now common in industry Chapter three provides background to the National Bureau of Standards project in robot control systems which is used as a starting point for the study outlined in this thesis. Chapter four discusses the specific issues of software design and implementation implied by the NBS project. Chapter five investigates how the NBS proposal has addressed the issues outlined in chapter four by defining an arc-welding robot application as a specific example. Possible alternatives are also discussed The proposal for Concurrent Hierarchical Control and its supporting operating system is presented in chapter six, including discussion of the operating system architecture, programmer development, downloading of the system into satellite processors, and performance evaluations. Conclusions are stated in chapter seven.

There are three appendices in this thesis appendix one gives the example application introduced in chapter five as it might appear using the concept of a cross-coupled hierarchy as outlined by the National Bureau of Standards. Appendix two gives the specifications of the processes which are discussed in the example in chapter five. In appendix three is provided a stub list of the subroutines written for the simulated operating system described in chapter six.

2. Background

The management of sensory feedback and its interaction with a robot control structure can be a highly complex operation. While open-loop control functions are easily expressed in a strict hierarchy or in a flat structure, the integration of sensory feedback can involve not only predictable state information such as position sensing, heat levels etc., but also exception information which must modify the whole operational sequence, sometimes radically in case of emergency. Such emergencies include both conditions which are dangerous to equipment as well as, more importantly, dangerous to operators [1]. Sensory feedback of exception conditions should provoke an immediate response on all appropriate levels, from a reflex reaction on the 'motor' level of automatic sequencing, to input to decision making software on a higher 'cognitive' level of operation.

One of the first applications of robots was in the exploration of the Moon and the planets. Well known examples are NASA's Viking Landers which were launched during the mid-1970s to explore the surface of the planet Mars. The Viking could be considered to be a robot in the classic sense since it was equipped with a shovel-like end-effector used to take soil samples of the Martian surface. One of Viking's big drawbacks was the long communication turn-around time with its terrestrial control station; because it had little on-board control, it took as long as several hours for it to execute a simple command such as reaching out to take a soil sample [2].

The desire to build a robot with greater autonomy prompted NASA's Jet Propulsion Laboratory to develop its 'roving robot'. Autonomy implies greater on-board computer control of such functions as navigation, sensory processing and error recovery A.M.Thompson describes the control system for the roving robot as consisting of three distinct concurrent processes: vision, manipulator control and guidance [3,4]. The three processes are coordinated by an executive in a loosely defined hierarchy. However, control does not take place in real-time. The JPL rover was capable of visually mapping out a terrain, storing maps for future reference and then referring to its internally stored maps in order to navigate When the rover started out on its path, it was blind and did not have the capability of real-time guidance with the use of visual landmarks.

An area of robotics in which much research has been conducted is compliant control. It may be defined as that control which performs an action which is made to comply with sensory feedback. For example, for a robot to grasp an egg requires not only a command to do so, but also continuous sensory feedback to ensure that the egg is being grasped with just the right amount of force so that it neither cracks in the robot's hand nor falls from its grasp. One of the first experiments in robotics to test the use of compliant control was undertaken at Edinburgh University by R. Popplestone [5]. The robot was able to recognize objects, although not in real-time. However once objects had been recognized and positioned, it was able to do an assembly using compliant control based only on force sensing.

The Westinghouse Research and Development Centre has recently been involved in an experiment using real-time visual input for the assembly of small electric motors [6]. The significance of the experiment lies in the fact that Westinghouse has attempted to build a cost-effective assembly system using robot vision in real-time, and that the system is designed for small batch rather than mass production. It is in the area of batch production, i.e., the manufacture of articles in small quantities, where the industrial robot is expected to find its greatest usefulness.

The interaction of sensory processing with control functions has been the subject of much research. Lozano-Perez [7] suggests four general categories of such functions:

* initiation and termination of functions.
* selection of alternative actions
* identification of objects
* compliant control

Nitzan [8] bases justification of increased sensory interaction on four current inadequacies of industrial robots:

* insufficient flexibility
* open-loop control
* inability to process errors
* high cost of accurate open-loop positioning

Laugier [9] describes the open-loop control of the trajectory of a robot arm whose very complexity underlines the gains that could be made through sensory feedback. In Laugier's example, the programming is complex, the computation of coordinate transforms expensive, and the arm itself must be engineered to a very small tolerance of error.

Hierarchical control is an approach frequently cited in the literature to address the complexity of robot software, both with and without sensory processing. The term is used in two broad senses: computer architecture, and software. In the area of architectural hierarchy, C.S.G.Lee describes a two-level micro-processor system which is used to control a Unimate PUMA robot [10]. Mercer and Vincent describe 'Function to Function Architecture' which is based on a set of single card micro-computers on a common bus [11].

In the area of software hierarchy, Friedman [12], Saridis [13], and Jappinen [14] have proposed various approaches for structuring the interaction of sensory processing and command processing. All use some version of a 'world model' to direct control functions, while sensory functions may modify this model to adjust to varying conditions in the environment of the robot. The concept of world model will be elaborated in section 3.1.6.

Friedman has identified the value of defining two complementary processing systems operating in real-time, one for sensory processing and the other for command processing Sensory processing may typically be used by the command processing system for feedback and for learning, and command processing may similarly be used by sensory processing to provide contextual filtering of incoming sensory data

Saridis has proposed a hierarchical control system which consists of three layers organization, coordination and hardware Cutting across the three levels are three hierarchies: command, vision and sensory The hardware level represents the control hardware for sensors and actuators The coordination level represents the drivers and filters for sensors and actuators. The organization level represents higher-level control and interpretation. Saridis' approach is based on the assumption that increasing intelligence implies decreased precision, i e , as sensory data is increasingly refined the details are more generalized

Jappinen has proposed the concept of an acquired skill as a basis for a robot control system. Higher skills may be defined by combining lower skills together leading to a list-like structure similar to a LISP program

Since the first space robots, there have been a number of examples of computer applications with robots. However, few commercially available robots exploit computer technology on anything but a relatively primitive level. Also, most applications of computer technology to robots in research have been characterized by an ad-hoc approach to the problem The most significant exception to this trend has been the work conducted by J.S.Albus et al at the National Bureau of Standards in Washington DC.

3. Derivation of the Cross-coupled Processing Hierarchy

In 1977, R.G. Abraham et al undertook a general survey of research in robotics covering such diverse areas as robot vision, coordinate transformations, servo-motor control, navigation, assembly and so forth [15] Their survey uncovered no work which sought to integrate the results of this diverse research into a generalized robot control system. Recent years have seen the beginnings of this type of integration.

A proposal for a generalized approach approach has been presented by Albus et al [16-22] at the National Bureau of Standards(NBS). They argue that the complexity of real-time control requires some sort of modularization, which in turn may imply a hierarchy of organization. Additionally, they maintain that sensory processing, in particular visual processing, is inherently hierarchical. The result of their work is a cross-coupled processing hierarchy consisting of 'n' levels, with each level containing a command processing module and a sensory processing module. Each level is also equipped with a world model which includes the point of interaction from control processing to sensory processing, as well as sensory processing which interacts directly with control processing (figure 3.8).

In this chapter we describe how the NBS cross-coupled processing hierarchy is built from a chain of feedback control mechanisms. The feedback control mechanism is derived using vector notation. The goal of this chapter is to familiarize the reader with the NBS proposal which is the basis and starting point for the work presented in this thesis. A preliminary discussion is provided in section 3.1 to 3.7 of the vector notation used by Albus to introduce the concept of the cross-coupled processing hierarchy. The reader may wish to continue directly to section 3.8 where the cross-coupled processing hierarchy is defined and described.

The simplicity of the derivation in light of the significance of the results is a great strength of the NBS proposal. In this chapter we start by discussing the details of that derivation.

3.1. Vectors

The NBS uses vector notation in order to derive the cross-coupled processing hierarchy. Almost anything can be represented by a vector in multi-dimensional space. In engineering and computer science applications, the state of a physical or logical process is often represented by a vector of parameters An object may even be defined with an abstract component part such as time For example, if we presume that the variable W represents the state of the weather, then we may define a vector W = (w1,w2,w3,w4) which represents four component scalars giving specific information about the weather (figure 3.1)

These scalars may be:

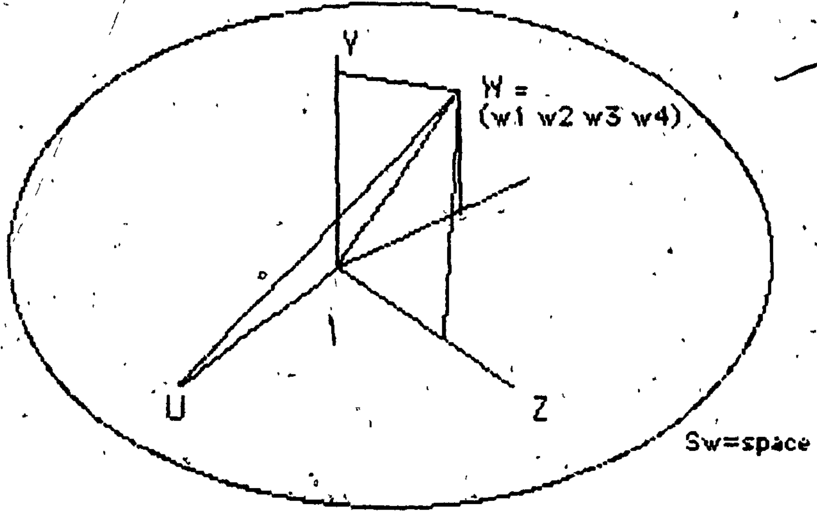

Figure 3.1

w1 = temperature
w2 = humidity
w3 = wind velocity
w4 = rate of precipitation

Therefore, W is a vector defined in a four-dimensional space. The space Sw is defined by all the possible values of

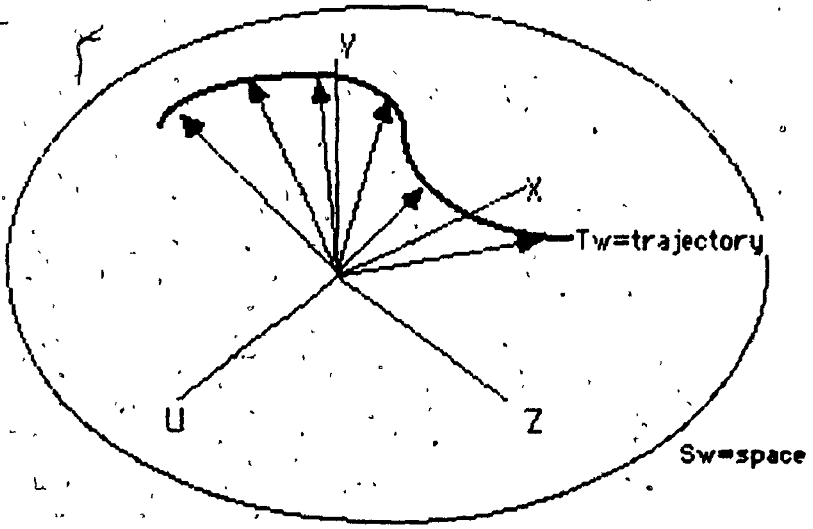

Figure 3.2

W. The trajectory Tw represents the path traced by the tip of the vector W as it changes through time (figure 3.2). We can represent the value of the trajectory Tw at any one point in time, i.e., Wt = (w1, w2, w3, w4, t) where t would be represented by a fifth dimension orthogonal to the first four.

3.2. Functions

A function defines a relationship between vectors. Let us presume that the vector S represents some set of input variables and that the vector P represents the corresponding set of output variables. Assuming a one to one correspondence from each input variable in S to a single output variable in P, we can say that there is some function H which

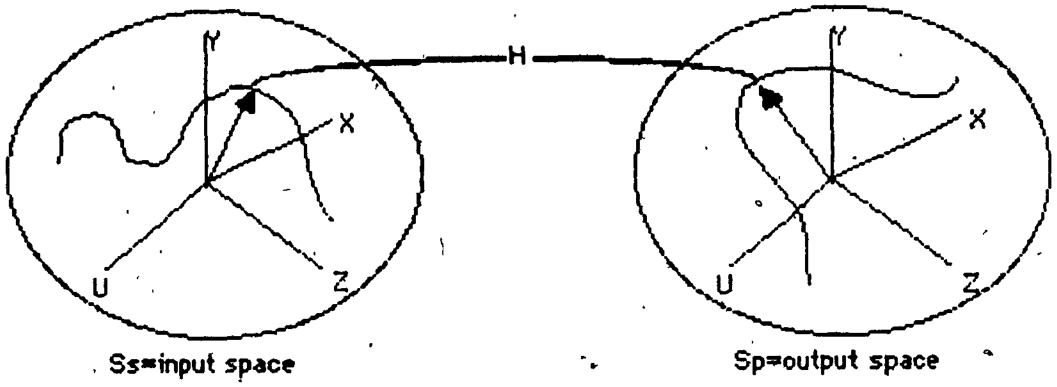

Figure 3.3

describes this relationship. This is shown by: P = H(S). Correspondingly, if S traces out a trajectory as it changes through time, then P may be presumed to trace out some sort of trajectory also (figure 3.3). The function H therefore maps the input trajectory Ts onto the output trajectory Tp.

A vector may be defined by component vectors For example, let us say that the vector S is comprised of two vectors C and F, where C = (c1, c2, c3) and F = (f1, f2, f3)

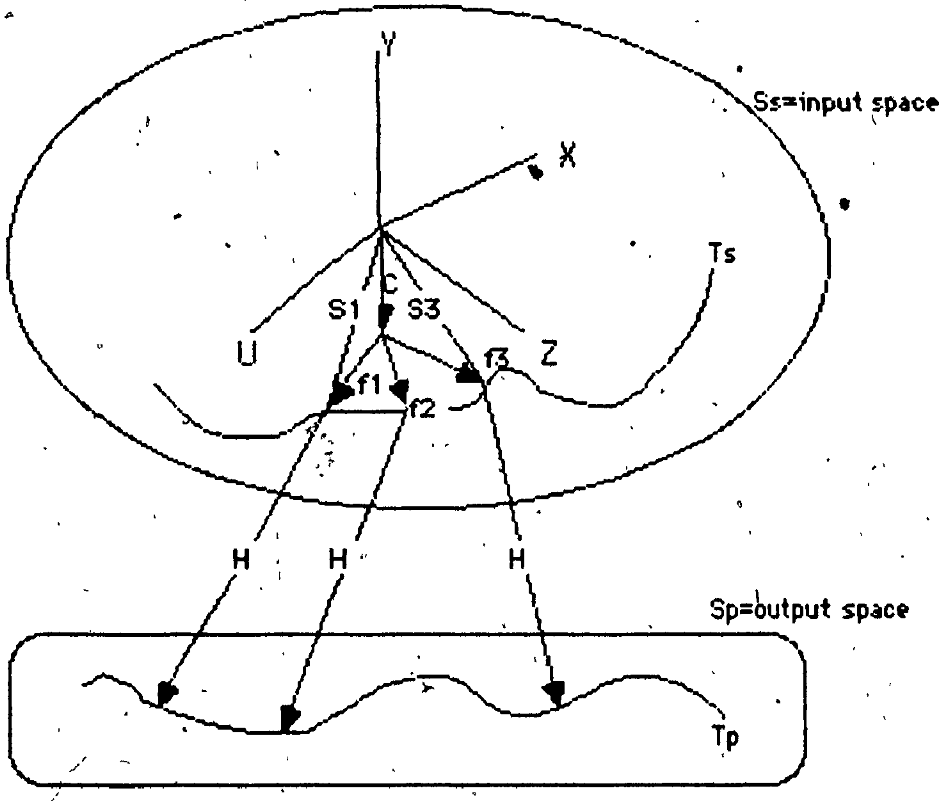

Figure 3.4

Therefore, S = (c1, c2, c3, f1, f2, f3) or S = C+F If C remains constant while F changes through time, then C defines a set-point. If C represents a vector of command instructions and F represents sensory feedback, then it may be said that a single C vector can produce a variety of output P vectors. In the example shown in figure 3 4, the C vector produces three output vectors using the three different values of the F vector.

3.3. Feedback Control Mechanisms

Vector notation can be used in order to represent a simple feedback control loop used for servomechanisms. A servomechanism is used to control the angular position of a robot joint. It is equipped with an actuator to move the joint in a positive or negative direction, and with a sensor which measures the angle of the joint at any point in time. The difference between the present angle and the desired angle is known as the error. The servomechanism activates the actuator until the error becomes zero. In figure 3.5, the input vector S is divided into two subvectors C and F, which represent command and feedback respectively. The command vector provides the desired position of the joint, the feedback vector the actual value of the angle of the joint. If the error is zero, then the value of P will indicate that the actuator is to be turned off. If the error is positive or negative, then the value of P will indicate that the actuator must be activated in a positive or negative direction. The H function is shown by a box and indicates that it maps input to output variables, i.e., it is a processing module.

If the feedback vector F requires some processing, such as filtering or conversion from analogue to digital, then a new function G may be required for the processing of the sensory signal (figure 3.6). The function G is defined as $Q = G(E+R)$. The E vector is the sensory signal. The R vector

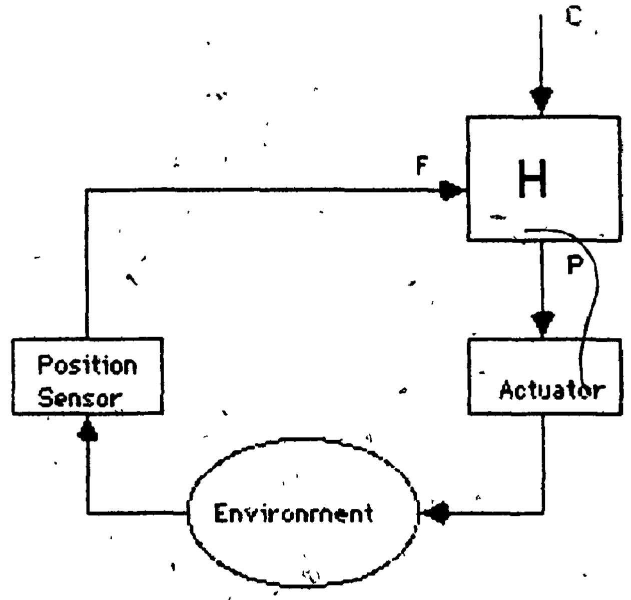

Figure 3.5

is information provided by the H module to help in interpreting the sensory signal. The Q vector is an output having two subvector components: the F vector (sensory feedback), and another vector En whose purpose will remain undefined at this point. Essentially, the interpretative function of the H function has been relocated in the G function,

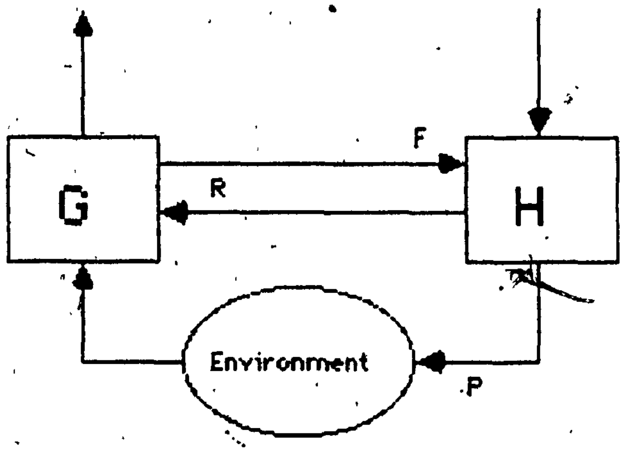

Figure 3.6

and a two-way communication has been defined between them.

Taking our servomechanism example again, this would imply that the R vector provides the positional goal, G calculates the error, and the feedback vector F tells the H function whether it should advance, reverse, or stop the motion of the actuator. We have at this point two distinct functions: an H function which alters the state of the environment, and a G function which interprets the state of the environment.

3.4. Linked functions

It is possible to conceive of a linkage of H functions, such that the output of one is used as the input of another, i.e., Pn+1 = Cn (figure 3.7) If there is a difference in the rate of change of the F vectors, such that F1 changes frequently and F2 and F3 change infrequently, then the vectors P1, P2 and P3 will change accordingly. This is what would be expected if F1 represents low-level sensory feedback which changes freqently, and F2 and F3 represent more highly processed sensory information which changes less frequently. For example, let us say that F1 represents positional feedback indicating the present angle of a robot joint. While the joint is moving, then F1 is continuously changing. Let us say that F2 represents a boolean value indicating whether the desired position indicated by C1 has been reached. All the time that F1 is continuously changing in value, F2 is false. Once the joint has reached the desired position, F2's value becomes true. The vector F2 thus changes its value infrequently with respect to F1. In

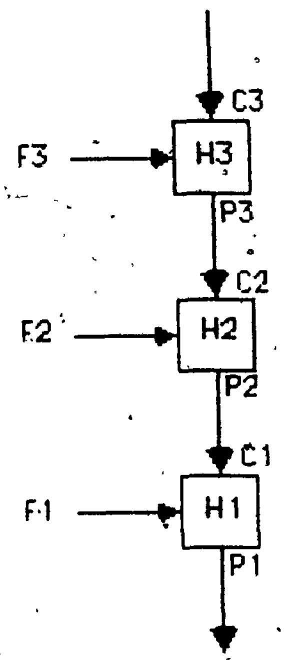

Figure 3.7

sensory processing, the more highly processed the information, the less frequently it tends to change. It is on this basis that NBS justifies the concept of hierarchical control.

Feedback control loops, such as the one shown in figure 3.6, may also be linked in a similar way to the H modules in figure 3.7. In this case, there is an added output vector E calculated by the G functions. The E vector outputted by a G function provides processed sensory information to another G function. If we take our servomechanism example once again, then the G1 function would also provide present positional information to the H1 module, and would provide processed sensory information to a G2 module indicating whether the move has been completed or not (figure 3.8).

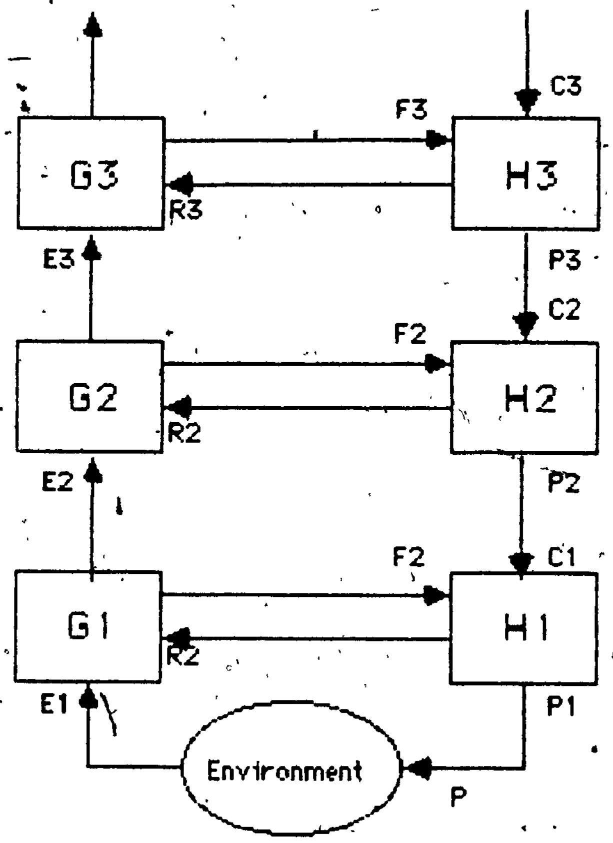

Figure 3.8

3.5. Error Correction

This vector notation can also be used to illustrate how errors can be corrected. The existence of an error implies the existence of some goal or ideal against which present reality is being compared. In figure 3.9, this is shown by the trajectories Ts1, Ts2 and Ts3. These trajectories represent the ideal trajectories of the S input vectors to the H modules. An S vector is simply the sum of a 'C' command vector and an 'F' feedback vector, therefore a single S vector such as S2.1 traces a path Ts2 through space according to the changes in the feedback vector F2.1 whereas the command vector C2.1 remains constant. When a new command is invoked, such as C2 2, then a new S vector S2 2 is defined and all feedback is subsequently defined as F2.2.

In the first diagram, there is a slight deviation of Ts1 from the ideal trajectory caused by unexpected feedback Therefore, the S1 vector has a slightly different value and the H module is able to calculate a P1 command vector which will seek to correct the error We may take an example of an arc welding robot where the temperature of the arc is adjusted constantly in order to take care of minor variations. What is significant is that the error detected by the sensors does not cause any change to the sensory processed information being sent to the next-highest level.

In the second diagram, the feedback error indicated by the deviation from the Ts1 trajectory has been large enough to affect the sensory feedback information at the S2 level. Therefore, the S2 vector has a value which deviates from the ideal and thus the H2 module can output another P2 command vector in an effort to bring the situation back to normal. For example, if we continue to take the example of an arc-welding robot, let's say that the arc temperature drops drastically to room temperature indicating that the arc has failed or that the tip has been frozen to the surface of the metal. Now, the sensory information being transmitted to the next highest level indicates an error, and the path traced out by the S2 vectors has been perturbed.

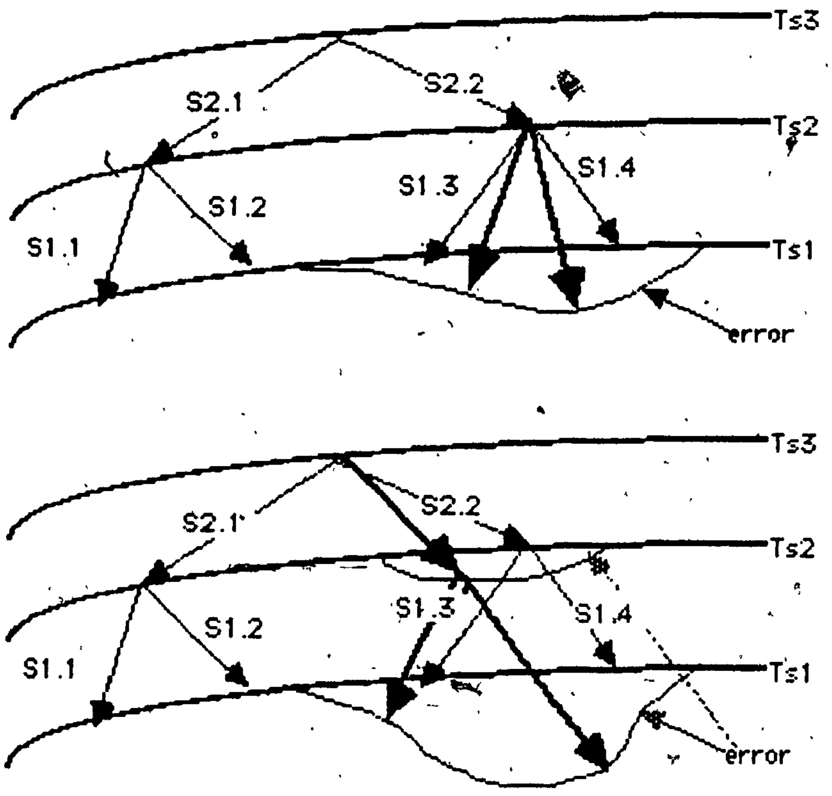

Figure 3 9

3.6. Context Information

The context information provided by the R vector to the G module can be illustrated using vectors. In figure 3.10, two separate sensory information vectors, Ea and Eb, are very similar in value. This means that the G module may have some difficulty in distinguishing between the two values. However, if another vector R is added to the E vectors such that Q = R+E, then Qa and Qb may be much more easy to differentiate. This is the effect that the R vector input has on the G module, i.e., it provides a context within which the sensory information vector E can be interpreted. This context is provided by the command which is presently selected in the H module The M module is defined as the

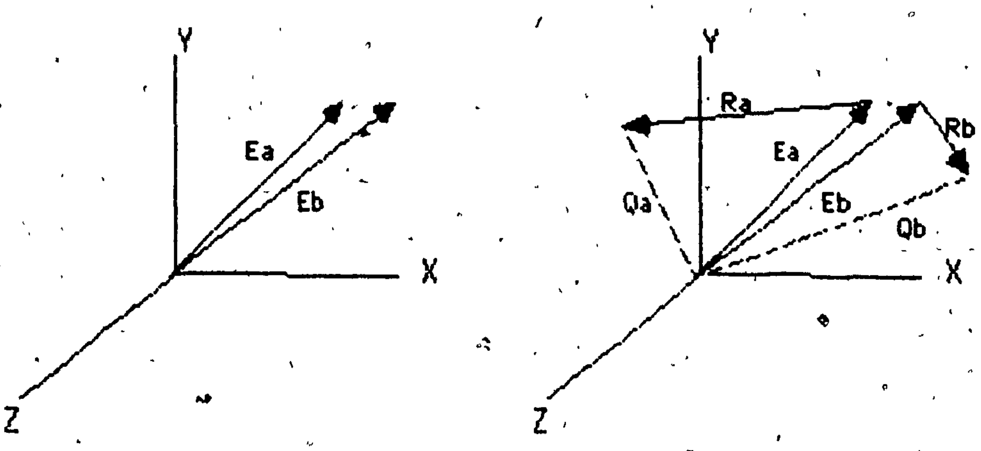

Figure 3.10

function which is responsible for context information It has two vector inputs, the P vector which is the output of the H module, and an X vector which draws from various other sources of information to help in the interpretation of sensory information, although these sources are not specified. Therefore, we have the relation R = M(P+X) (figure 3.11).

3.7. Prediction Information

As well as providing context to the G modules to help in the interpretation of sensory data, the M modules also provide a prediction of what to expect, i e., a goal or an ideal against which present sensory information represented by the E vector must be compared in order to calculate an error. The result of the comparison is then fed directly back to the H module

The M module can be considered to be hard-coded when it is normally executing. However, it may also be placed in a self-modifying or learning mode. This is accomplished by

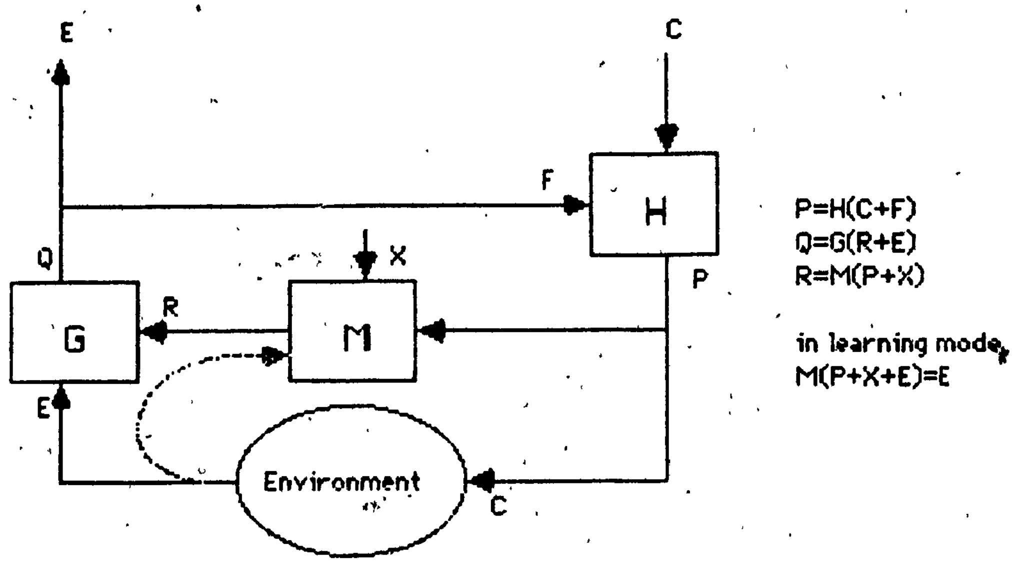

Figure 3.11.

inputting the E vector directly into the M module. This is indicated in figure 3.11 by the vector shown as a dashed line. This has the effect of making the vector R equivalent to the vector E so the G module produces a zero error.

3.8. Cross-coupled Processing Hierarchy

In its final form, the NBS proposal consists of 'n' feedback control loops built on top of one another in a ladder-like structure whereby each module is implemented as a finite state machine (figure 3.12). The entire system executes once every time interval t=28ms. At each time interval, every module samples its inputs and determines which entry in its table corresponds to the input vector. The entry to which the vector points may either contain an output value or a pointer to a procedure which calculates an

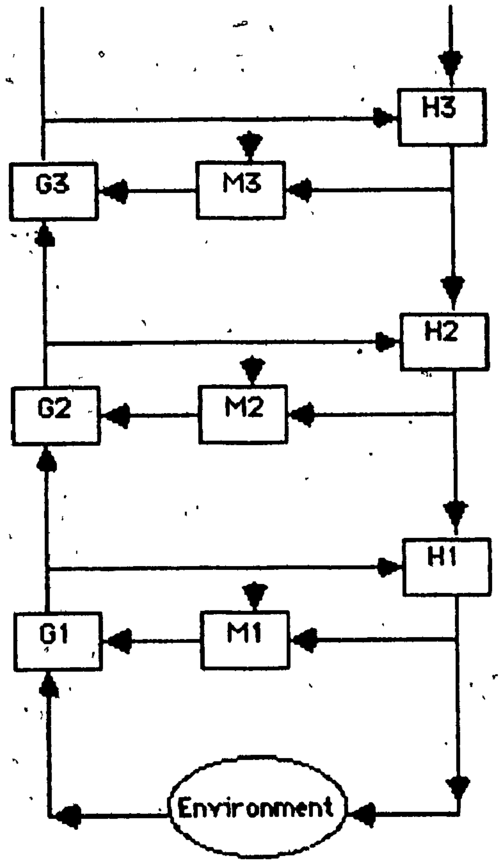

Figure 3.12

output value.

The National Bureau of Standards has thus defined a generic control system for robots based on the feedback control loop mechanism. The conceptual simplicity of the cross-coupled processing hierarchy belies its effectiveness and its significance. It is one of the first serious attempts at defining a generic approach to robotics software by capitalizing on the feedback control loop mechanism well known in process control applications

4. Issues Raised by the NBS Proposal

4.1. Principles of Hierarchy

Hierarchy is a much used and perhaps even abused concept in the field of computer science It is useful therefore to consider how hierarchy is understood in general terms, and then how it may be specifically applied to the problem of robot control systems.

One of the purposes of computer hierarchy is to simplify a problem by dividing it up into tasks and the tasks into subtasks Top-down and bottom-up design in structured programming are well understood concepts.

A hierarchy may be defined using the principles of graph theory Specifically, some types of hierarchy may be represented by a tree Any hierarchy consists of two types of objects: nodes and edges. Examples of nodes could be procedures, data structures, processes or computers Examples of edges could be commands, data, messages or communication lines. Nodes are connected to each other through edges in a specific way. A tree is defined as a graph of nodes and edges in which there is one, and only one path between every pair of nodes. A graph with more than one path between any pair of nodes is defined as a network and would be characterized by the existence of at least one cycle or loop in its structure. In an oriented tree [23], one node is designated as the root, and the edges have a direction with respect to the root. The paths start at the root and trace themselves out toward the terminal nodes or leaves (figure 4 1). However, it is conceivable that in some circumstances the reverse might be true, i.e., that all paths lead to the root (figure 4 2).

A structured program may be characterized as an oriented tree. The nodes and the leaves beneath the root represent procedures, and the edges represent procedure calls. The structure of a program in a language such as C could be represented by a two-level tree with the root node being the procedure 'main()' and the leaves connected to the root being all other procedures. This is because C is not a block structured language where procedures may be defined within other procedures. However, the execution of a C program is not necessarily represented by a two-level tree, since procedures may call other procedures. By contrast, the structure of a Pascal program would be represented by an n-level tree, since procedures may be defined within procedures to any level of nesting Because programs written in C and Pascal execute serially, we can say that there is never more than one node which is executing at any time at any level.

However, there is no reason why some of the nodes of an oriented tree may not be considered to operate in parallel, particularly those which are found on the same level. A program written in a language such as Concurrent Pascal or

Euclid [24] may define processes which are allowed to execute concurrently. Similarly, the nodes (i.e. CPUs) of a multi-computer system operate concurrently.

In the NBS project, three such oriented-tree like hierarchies are defined: the H, M and G hierarchies which handle the command, world model and sensory processing functions respectively. Although each level contains only one node implemented as a finite state machine, the execution sequence is represented by an oriented tree with each node executing serially (figure 4.3). The H hierarchy is the one which is the most clearly articulated in the literature published by NBS. It is a simple hierarchy with one node at each level, each node being a finite state machine or FSM and each edge being a command pointing downward from one machine to the next lower machine. The nodes in the H hierarchy execute in parallel, each one sampling its input and calculating its output during a 28 millisecond time slice. However, there is only one node on each level and no means by which commands on any one level can operate concurrently.

In the G or sensory processing hierarchy, each node may also be an FSM according to the authors. The edges connecting the G nodes are directed upwards and represent information processed by one G module sent to the next higher G module. If we consider the edges between G modules to be references rather than information flow, then we can say that the edges are pointing downward The G hierarchy also contains only one module at each level and there is no concurrency implied at any single level.

The world model, although conceived as a hierarchy, has not been represented hierarchically in the literature published by NBS. Instead, M modules appear to be autonomous on each level with no edges explicitly defined from any M module to any other M module. The X vectors, which are intended to carry information from elsewhere in the system to the M modules, have not been defined by the authors

The cross-coupled processing hierarchy as put forward by NBS consists of these three tree-like hierarchies, H, M and G, connected together to form a single system. However, because the links across the three hierarchies operate in two directions, from the H to the G hierarchy and vice versa, the cross-coupled processing hierarchy actually forms a network since a loop is defined at each level Another way to describe the cross-coupled processing hierarchy is as 3 hierarchies cross-coupled into a network, or if an entire level is considered to be a single node, as a hierarchy of feedback control loops. It may be more accurate to qualify the cross-coupled processing hierarchy as three hierarchies cross-coupled into a network.

It is the arrangement of edges and nodes which determines whether a particular structure or graph is a hierarchy and what type of hierarchy it is There is another way to understand hierarchy which is less rigid than that suggested by oriented trees It is referred to as hierarchy by level by Brown and Denning [25]. In this approach a new object is defined, the level, by its distance from the root. A level hierarchy is exactly like an oriented tree with the exception that an edge may exist between a node in one level and a node in any lower level (figure 4.4). This implies that there may be more than one path from the root to a node, but

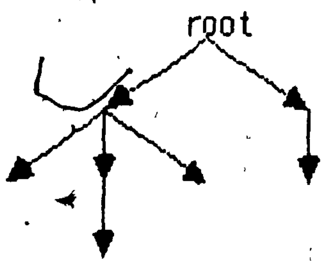

Figure 4.1

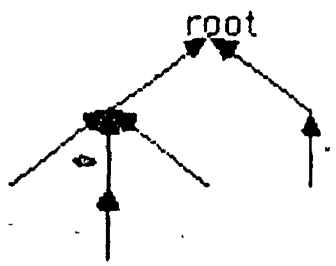

Figure 4.2

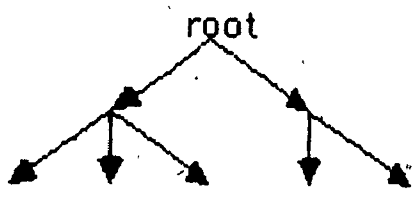

Figure 4.3

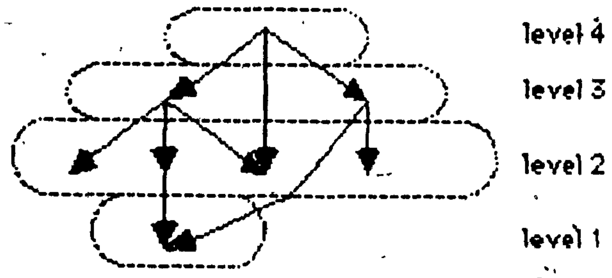

Figure 4.4

there still are no cycles. There is only one direction defined from a higher level to a lower level. Xinu is an example of an operating system written using this approach [26]. It exhibits the advantage of being able to access lower-level routines from high levels without the necessity of passing routine calls through the intervening layers.

The nature of the nodes in a tree or level hierarchy is usually straight-forward. In the NBS model, the nodes are FSM modules. However, the exact nature of the edges also merits discussion. The nodes of a hierarchy may be connected by control flow, information flow, physical connections, parent/child relationships etc. In the NBS model, the edges of the H hierarchy are command messages (i.e. invocations) sent from a higher module to the next lower module. Later versions of the model also include a 'report' from each lower H module to the next higher module The purpose of the report is to return the status of an executing procedure. This is similar to what happens when a C program makes a procedure call. When the procedure terminates its execution, it always returns a value to the calling procedure which may be used to indicate status. Therefore, the invoke-command/return-report function in the NBS model may be considered to be analogous to a C procedure call

However, edges do not necessarily need to represent procedure calls. In the G hierarchy of the NBS proposal, the edges are directed from the bottom to the top, and represent the flow of processed sensory information There is no suggestion that there is any information returned in the form of a report indicating whether sensory information has been received. If one were to consider an edge in the G hierarchy to be a reference to a lower module by the next higher one, then the edges could be considered to be directed downward from the top, and this would imply a two-way communication similar to that defined for the H_ hierarchy. In the NBS concept, edges are used to represent information flow between nodes, whether the information is command information or sensory processed information.

In the context of a robot control system however, edges may represent other more specific relationships between nodes as well. They may represent messages sent from one node to the next. A message may contain either control or data information, depending on how it is interpreted, but the implication is that the communication is only one way. There is no status report returned to the calling module unless it is implemented by using a second message.

Edges may also represent events. In such a hierarchy, there may be a central controlling program which interprets events and readies processes according to various combinations of events. In short, a structure which is stated to be hierarchical may not actually be so from the point of view of Standish's oriented trees or Brown and Denning's hierarchy by level. Hierarchy is used in many different senses, but may be generalized to refer to any system in which more than one level has been defined and where the levels are related by degrees of difference of a set of attributes.

In summary, there are two justifications for the use of hierarchy in a robot control system. First, hierarchy provides an approach by which the enormous complexity of an application problem may be subdivided into smaller more manageable subproblems. Second, hierarchy may provide a method by which a task may execute subtasks concurrently

4.2. Robot Control System Concepts

4.2.1. Application of Hierarchy

In the context of robotics, hierarchical control may be characterized metaphorically as a kind of structured programming applied to mechanical control. In a structured program, the operands may be records, files or other data items. The operators are arithmetic or logical In robotic control, the operands are objects and the operators are actions which are done to physically alter the position or composition of the objects

Complex systems tend to be best managed through some sort of hierarchical approach. Early in the history of computer programming, an executing program would need to take care of all computer I/O and other system operations. A program would even need to load its own loader before it could start execution. It soon became clear that if another program were to run at a level higher than the application program, then such functions as loading and I/O could be offloaded from the application. This meant that the application programmer could concentrate more on the actual nature of the application, and less on loading and I/O. A good indication of the sophistication to which operating systems have evolved is provided by the work of Brown and Denning [25] outlining their multi-layer operating system. The principles they articulate may be applied to robot control systems with some differences related to the nature of the control problem.

The development of structured programming was similar. Until the advent of such programming languages as C and Pascal, applications were written in a single-level fashion, i.e., every statement of the program was accessible from every other statement, and there could be any number of exit points. As applications became more complex, it became more difficult to write good programs. The advent of structured programming introduced hierarchy into programming where previously none existed. It allowed the programmer to do his job in a more efficient manner by simplifying the programming process, i.e., by breaking it up into small manageable sub-tasks. It also led to the development of higher performance computers with stacks, interrupts, memory tagging etc.

In robotics, the number and sophistication of I/O devices is growing. A robot's I/O devices are used to measure or alter the environment, as opposed to a computer's I/O devices whose purpose is to read or write data. A robot's I/O consists of such devices as touch, force, position and optical sensors along with actuators and end-effectors.

The only input devices installed on a typical industrial robot today are joint position sensors which allow the robot to be servo-controlled, and a command console. A program for such a robot is usually a simple linear sequence of joint positions with no conditional branching or subroutines. The position of the object to be manipulated must be precise, for the industrial robot usually has no means of detecting the object's presence or absence. This of course may lead to some ridiculous situations where, for example, an automotive arc-welding robot may start welding thin air if the assembly line associated with it has broken down. Certainly it can be argued that an arc-welding robot might be more cost-effective if it were able to do a weld according to the actual position of the seam to be welded as well as to identify the car model and adapt its programming accordingly.

The growing range and complexity of sensory input heightens the necessity of looking at more sophisticated robot control systems. This is not to say that an existing operating system such as Unix may be applied to robot control; the constraints of the problem are quite different and the I/O devices of a robot demand different treatment.

4.2.2. Differences Between Robot Control and Timeshared Systems

If we presume that a process in a robot control system represents a skill of some type, then processes do not need to be created or destroyed. They may be started up at load-time and allowed to exist indefinitely. In a timeshared computer operating system such as Unix, processes may be created or destroyed according to the often unpredictable needs of users. This affects such things as memory allocation, virtual memory, process management etc.

In a robotic environment, the operator of the robot may be considered to be the only user. There is therefore no user contention for resources in the same way as there is in a timeshared environment, i e , a robot control system has different constraints for resource management than a computer operating system. Processes cooperate rather than compete for resources. The interaction of such processes is similar to that of the 'task force' concept of Jones and Ousterhaut [27,28]. They define a task force as as a group of independent concurrent processes which cooperate in order to accomplish a single predefined task. In a task force, the tasks of a robot control system rely more on one another than is the case with processes in a typical multiprogramming system. In contrast to a task force however, there is a clear hierarchical structure to the set of tasks Interprocess communication therefore takes on special importance. For the purpose of clarity, this thesis will refer to a process as an independent executable unit commonly found in timeshared systems where they compete for resources and have no functional relationship to each other. A task will be referred to as an independent executable unit which is generally more function specific (i.e. smaller) than a process and also competes for resources but which has a clearly defined and inter-dependent relationship with other tasks.

A robot control system may contain within it some procedure for modifying its own structure, i.e. it may have the ability to learn or to optimize its performance. This is not typical of a computer operating system.

The purpose of a robot control system is to effect some change on the environment, such as painting or welding Therefore, the robot control system will usually have a goal or an ideal with which it can compare the present situation A computer operating system does not require any such goal or ideal with which to compare its present activity, apart from low-level traps indicating potentially disastrous error conditions.

Real-time constraints in a robot control system are demanding. Sensory data must be processed and decisions made with split second timing. This is true particularly in the case of exceptional or emergency conditions, such as a workman stumbling into a robot's work area If a robot control system is developed entirely ad-hoc, then an off-the-shelf operating system may be used as a basis for execution. However, an off-the-shelf operating system such as Unix or IRMX [29] may contain too many features that are not required. As well, it may not contain features which are required, such as support for high-priority exception conditions. Certainly a standard multiprogramming operating system such as Unix would be inappropriate for run-time use, although it may be suitable for development purposes. An operating system such as IRMX or some other system intended for process control may be more appropriate for run-time execution IRMX for example, may be configured in such a way that it does not contain higher-level file and process management features of a more sophisticated operating system. Somehow, a robot control system needs to combine the real-time responsiveness of process control with the sophistication of multi-user timeshared operating systems.

4.3. Real-time Operating Systems

4.3.1. Timing Issues

A robot control system be considered to be a very sophisticated real-time control system with strong similarities to process control. A real-time operating system has stringent requirements on performance. It must be able to receive and process asynchronous events within strict time response constraints. Real-time systems for process control have often been designed ad-hoc since time response constraints have been so demanding, however there are now available off-the-shelf operating systems such as IRMX which are designed to provide a workbench for the design of process control applications on a single computer.

The NBS approach to real-time constraints has been to design the cross-coupled hierarchy such [illegible] each finite state machine at each level executes at eve[illegible]me interval t=28ms This is acceptable if the procedure pointed to and executed by an FSM module at any time does not require more time. The 28ms constraint must accomodate the slowest procedure in the system, if higher levels are added to the cross-coupled processing hierarchy which exhibit greater sophistication, the 28ms constraint may need to be relaxed in order to accomodate the longer processing time required Albus states that the constraint that a procedure must have finished its execution within 28ms may be relaxed so that execution may cross a time interval boundary If the time interval boundary is relaxed, then the response time of the procedure is some multiple of 28ms and the response time of the entire system has to be altered accordingly. The crossing of a time interval boundary also requires a certain amount of processing overhead to manage completed and incompleted procedure states. By comparison, in a timeshared operating system where time intervals are not used to determine process scheduling, procedures or processes may be scheduled to execute, be interrupted, or terminate at completely unpredictable times. Additionally, the fact the each finite state machine module is required to sample its inputs and produce an output at each time interval may mean that some modules will find that their inputs do not change from one time interval to the next, or change only slowly There would therefore be no need for such a module to execute at all if it could know whether its inputs have been altered or not. In operating system design, such a scenario where inputs are checked on a regular basis is referred to as polling or busywaiting The alternative approach is for a change in input to cause a interrupt which informs the system that the new input is available for tha module

The cross-coupled processing hierarchy may be ported to faster hardware in order to overcome timing constraints as the finite state machine modules grow An alternative might be to restructure the software of the control system in such a way that would minimize the processing requirements of the finite state module and its procedure entries

Although the three hierarchies, sensory processing, world model and command decomposition may theoretically operate concurrently, the actions defined by any one module cannot. A significant example is the lowest-level command decomposition module, where many actions could be considered to be taking place simultaneously if allowed to The fact that each module is defined as a finite state machine means that only one output is permitted at a time. The action or output from this module is a single linear command stream initiating and terminating actions

4.3.2. Multi-computer Issues

The NBS project includes a proposal for a hardware implementation of several Intel 8086 micro-processors and a PDP-11 mini-computer. Inter-processor communication is to be implemented using a 'mail-drop' system whereby each computer writes to and reads from a common memory at every time interval The application is partitioned across the multi-computer network by assigning the lower-level FSM modules each to a 8086 micro-computer and placing the remaining higher-level modules in the mini-computer. It is possible that a particular module does not fully exploit the resources provided by a micro-computer and spends some time waiting for the end of the time interval Also, the inflexible partitioning of the application may cause difficulties if the number of processors is variable due to the fact that the smallest object which can be moved from one processor to another is the FSM module If there were a means to partition a module across more than one processor, then a more efficient means of distributing processing requirements might be devised.

Another issue is the use of common memory for message transfer. Common memory is a relatively fast and easy way to implement inter-processor communication, but common memory requires that the processors be bus-compatible and that the physical distance between the processor be within a narrowly defined limit. If a new processor were to be added to the computer network, the bus-compatibility problems might be considerable

If common memory were not used between processors, then an alternative means of inter-processor communication would need to be implemented Processors may communicate using a parallel communication port, i.e , a bus, or a serial interface such as Ethernet New problems then arise, such as how and whether or not the processors are to be synchronized with one another and what is to be the communications protocol. The use of a multi-processor operating system may also be implied in order to render the hardware transparent to the application.

4.4. System Development

Determinism of execution refers to the fact that every conceivable set of input conditions has a pre-defined output associated with it. The cross-coupled processing hierarchy is a deterministic system. Such a system optimizes processing speed but at the cost of flexibility and a consequently reduced capacity to process exception conditions. The NBS work suggests that there has been considerable grappling with this issue, particularly in the the work of Shneier and Kent [30,31], whose development of the sensory processing hierarchy for the cross-coupled processing hierarchy has produced a software architecture which is very different from the command decomposition hierarchy with which it is intended to interact.

The NBS approach to system development is to design H, M and G modules by defining each as a finite state machine with state-table entries pointing to output vectors or to procedures which calculate the value of the output vectors. This approach produces what is called by Holt a monolithic monitor [32], which includes all hardware and software support. An alternative would be to separate out of the application those facilities which remain constant from one application to the next; in short, to develop an operating system as a workbench

In the workbench approach to system development, either a two-tiered module is produced composed of the application and operating system linked into a single load module, or the application is linked and then loaded onto a computer which already contains the run-time operating system The lower level is the operating system, which may provide various system facilities to the application such as scheduling, memory management, file management, interrupt handling etc. Although the operating system may itself be designed as a hierarchical system with information hiding between layers as described by Brown and Denning [25], to the application it appears as a single layer and that is how it will be considered here.

The development system need not be the same as the run-time operating system. For example, the application may be written on Unix in C using a special library of robot procedure calls, and compiled into a load module.

Any application is often structured using a particular object, whether procedures as in Pascal and C, tasks as in Concurrent Pascal [33] or modules as in Modula [34]. The NBS project uses FSM modules as the main object to structure the application and it shows both strengths and weaknesses. If tasks or processes are chosen as the main object for execution, then there is the possibility of concurrent execution of the commands contained within a single FSM and flexible partitioning over hardware boundaries

4.5. The World Model

A robot control system should have a means of representing the robot's world at any given point in time. Separating a world model from other control system components is one of the first steps in building a system unique to a robotics application. The NBS world model is designed to provide command contextual information to the sensory processing hierarchy to help in the interpretation of sensory input data. This concept may be expanded to include memory of previous events, heuristic processing and learned behaviour.

In its present form, the NBS system has the capability to learn incrementally if a given set of sensory inputs is not recognized There are two ways of accomplishing this: first, when the system detects a set of input conditions for which it has no table entry, it halts and waits for the operator to insert a new table entry to handle the present set of conditions. Second, the sensory information vector E can be inputted into an M module which resets its output vector R to correspond to the same value as the E vector.

The world model is described by NBS as a hierarchy of M modules, with an X vector representing contextual information derived from elsewhere in the system, providing the hierarchical structure. Unfortunately, the nature of the X vector input is not specified, so it has not been possible to examine it in greater detail

4 6. Exception Processing

Exception processing is required in the case of some unexpected event, such as an unidentified object inside the robot's workplace, or a potentially destructive reading detected on a sensor's input Exception processing may be characterized by two types self-protective reflex action, and reasoned response to a change in environment An exception may provoke either or both of these reactions depending on its nature A reflex action is essentially a lower-level response. An unexpected object in the path of a robot arm must provoke an immediate stop or an avoidance manoeuvre This response would be implemented in the lower levels of the processing hierarchy Subsequently, the robot arm may have to call for instructions or notify a higher manufacturing level of the event This response would be the result of a message sent to the highest command level, which is the decision level indicating what move to take next A robot control system must take into account the need for high-priority processing to respond to real or potential emergencies.

5. The Design of a Robot Operating System

The problem of a robot operating system is illustrated below with the specific example of an arc-welding robot. The detailed definition of an example was of great usefulness in investigating the issues associated with a robot operating system and in demonstrating the viability of the proposal made in this thesis.

5.1. An Example of a Robot Application

The example defined was that of a hypothetical arc-welding manipulator with five degrees of freedom and its implementation is described in terms of cross-coupled processing hierarchy and Concurrent Hierarchical Control. Arc welding was chosen as an application because several of the issues introduced in chapter four may be addressed by discussing and conceptualizing an arc-welding system. The variety of sensors and actuators required for intelligent arc welding is rich enough to demonstrate a reasonably complex application. The application lends itself to some sort of structured programming which can illustrate the hierarchical and multi-processor issues implied by a sophisticated robot control system.

The operation of a sensory controlled arc-welding manipulator requires real-time processing and can thus illustrate the problems encountered when a sophisticated hierarchically-organized system is required to operate under rigid time restraints. One measure of an arc-welding robot used in this thesis is the number of sensor and actuator messages it can support per second (i e how quickly it can respond to its environment)

The concept of exception processing can be well illustrated with the application example since arc welding is by nature a hazardous job, and an intelligent robot must be equipped to react instantly to potentially dangerous situations. The example also provides an opportunity to address issues related to the concept of the world model.

In designing the example of an arc-welding robot manipulator, J. Engelberger proved to be a valuable source of information [35]. In arc welding, an electric arc is generated between the welding electrode and the metal surface to be welded The arc melts the electrode as well as the metal surfaces along the joint to be welded. A constant supply of electrode is moved forward to replace the electrode which has been melted and deposited in the weld. When a joint is to be welded, the initial step is to turn on a continuous flood of an inert gas such as helium or argon. This prevents the molten metal from oxidizing. Then a voltage potential is generated between the electrode and the surface and an arc is created. The position of the electrode needs to be at an optimum distance from the joint in order to start and maintain the arc. If the electrode is too far then there will not be enough heat generated to form a pool of molten metal. If it is too close, ~~then~~ the pool of molten metal will be too hot and too much of the metal surface will melt away If the electrode is touching the metal surface, then the voltage across the arc drops to zero and the electrode will freeze to the metal surface A good weld will be one in which there is no metal build-up or blow holes

Our theoretical manipulator is presumed to be equipped with devices to measure the following environmental constraints:

* position of joints 1 to 5
* temperature in the vicinity of the electrode
* constant supply of inert gas
* constant supply of electrode material
* flow of inert gas, on/off
* velocity of electrode material
* start-of-gap/end-of-gap indicators
* range to gap to be joined, 0-10 cm

Our theoretical manipulator also has actuators capable of the following primitive actions:

* move joint 1 to 5 to a given angle
* increase/decrease voltage to electrode
* turn on/off inert gas
* adjust velocity of electrode material
* recognize start-of-gap/end-of-gap

Severe environmental constraints, such as heat, smoke and sparks are not considered as part of the problem definition, although they are nonetheless important. However, the goal of this proposal is to investigate an alternative approach to high-level control systems.

An example of an H hierarchy with three finite state machine modules has been derived and is provided in appendix one. It is a system comprised of three levels of the command decomposition hierarchy. The first step in designing the arc-welding example in the cross-coupled processing hierarchy was to define the sensors and actuators and to define the H1 module which supports them. Each table entry provides three fields for input and two for output the three input fields form a vector comprised of the input command, sensory feedback information and the report from the lower H module, the two output fields comprise the command output to the lower H module and a report returned to the upper H module. Level H2 was defined using the table entries of H1, similarly H3 was defined using the entries of H2. Unfortunately, insufficient information is provided as to the exact nature of the G and M hierarchies so there is no means to effectively derive an example of them based on the example arc-welding manipulator

Several processes within the same FSM module were identified that could conceivably take place in parallel. In module H1 for example, such functions as monitoring and adjusting the arc voltage, checking the temperature, moving the actuators could all be implemented as tasks executing continuously rather than functions which are simply turned on or off as in the cross-coupled processing hierarchy. In appendix two the arc-welding example is altered by first identifying in the H1, H2 and H3 modules elemental feedback control loops and implementing them as discrete concurrent tasks. By so doing, each task may be dedicated to a single function, and is free to execute or not execute according to demand, and the functions may be continuously monitored rather than simply turned on or off. In the Concurrent Hierarchical Control example in appendix two, a more detailed representation of the interaction of sensory processing with command decomposition is provided than that which could be represented for the cross-coupled processing hierarchy. The world model has been distributed among all the tasks so that each maintains an image of its little bit of the world; it is conceivable that a task may be prompted by another task to return some information it has stored in its world model, although this has not been specifically addressed in the example.

5.2. Concurrency

There have been numerous proposals over the years for programming languages which support concurrent or parallel processing. Brinch Hansen [36] has developed a set of basic principles that may be applied to languages which support concurrent processing. Among these are Concurrent Pascal [33] and Modula [34].

In the cross-coupled processing hierarchy, parallel execution exists because of the multi-computer hardware architecture, with lower-level modules being assigned one each to an Intel 8086-based micro-computer and the remainder being assigned to a PDP-11. It does not appear that any subtasks within a module operate in any way but serially, although it is suggested by Albus that the FSM modules may execute concurrently on a single mini-computer if the need existed. An FSM module is a very large object to manipulate and distribute across a multi-computer architecture without there being inefficiencies in the processing resources allocated. A more flexible approach to concurrency may be applied to the monolithic levels of the cross-coupled processing hierarchy The simplest way of doing this would be to redefine the procedures pointed to by an FSM module's state table as concurrent tasks The module in this way may still be defined as a finite state machine, but the procedures may execute concurrently as tasks, and the smallest object of independent and therefore concurrent execution has been reduced in size from an FSM module to a procedure. Such tasks may be divided into two types sensory tasks and command decomposition tasks.

If each procedure is defined as a task, then each task in the application must have associated with it an execution priority. Sensory tasks should be given high priority for the same reason that interrupts are given high priority in a data communications network, because the interrupt may contain some information required by a high-priority waiting task. Command decomposition tasks may therefore be given lesser priority. In this way, some of the inflexibility of the cross-coupled hierarchy can be dealt with, and resources more flexibly allocated to where they are needed. For example, a continuously executing sensory task which checks for potentially dangerous conditions can be given a slightly higher priority and exclusive access to a very high priority command task designed to deal with emergencies. The command task would remain suspended from using system real-time resources until an emergency condition was detected and it was scheduled to execute.

Another slightly different approach may be to define more than one finite state machine at any level in the G, M or H hierarchy In other words a broadened adaptation of the NBS system with concurrency both between levels and on levels The finite state machines may then execute concurrently, and the application may be more closely characterized by a Petri-net [37] Petri-nets are a modification of finite state machines; instead of requiring that a machine move only from one state to the next, a machine can move from one state to several others and then back to one state or on to more For example, moving from one state to several others may be implemented by one finite state machine calling several others to execute concurrently. The call would be completed when all the machines had terminated execution

5.3. Communication and Synchronization

In the cross-coupled processing hierarchy, communication is implemented in a simple and straight-forward way through common memory. Synchronization does not depend on communication but on the time interval of 28ms

A broadened approach to a robot control problem which exploits modern operating system design principles would require more complex communication and synchronization techniques. Presumably, the greater communication overhead implied is made worthwhile by the more effective use of system resources and greater flexibility. Also, in a system which does not execute in lock-step at every time interval, communication needs to be synchronized where necessary such as is now the case in an operating system like Unix where signal() and wait() system calls are defined for process synchronization. The communication lines defined between processes may be characterized by the edges between the nodes of the control hierarchy

Presuming that concurrent tasks are the primary programming tool, then there are a number of ways in which tasks may communicate They may communicate by pipes such as processes do under Unix A pipe is an unnamed file of any size under 4K bytes which acts as a FIFO or first-in/first-out queue. In Unix, producer and consumer processes execute concurrently on a pipe, i e , the consumer does not need to wait for the producer to complete execution. Therefore, a whole chain of concurrently executing processes can be defined using pipes, and the chain acts as a single program As soon as information has entered the pipe, the consumer process can start execution, and the operating system ensures proper information transfer. If the producer or consumer process fails in its execution or Unix is unable to effect the transfer, then Unix will inform the user that the pipe has been broken, and the program terminated.

Another means of communication is through messages A message sends information packets only one at a time, there is therefore no FIFO queueing as with pipes A message may be used to send information, to indicate that an event has taken place or to signal a process In a message transfer, the sending process may or may not be informed whether the transfer has been successful or not If any additional information if required, such as the status of an invoked process returned to a calling process, then this will usually require a separate message Message handling requires some sort of operating system facility to effect the transfer. It may be that messages are handled by a single system message handler, in which case messages could be prioritized and queued by the system.

One of the implications of message transfer is that messages do not necessarily need to be processed FIFO. In a real-time system, if a new message has been received by a process before an old message has been processed, it might be preferable to process the new message first and discard or save the old message, if the new message has more relevent information or a higher priority. In a robot control system for example, if a sensory process is monitoring the environment and sending messages to a receiver process, any previous message sent to the receiver process will no longer be valid since it no longer represents the present state of the environment, much like yesterday's newspaper does not give today's news It may be preferable that the receiver process its messages LIFO, or last-in/first-out, or that it only process the most recent or highest priority message

Processes may also communicate through procedure calls This is an approach used in Modula as described by N.Wirth [34]. Processes may be defined in a Modula program so that global procedures are available in each process which are available to every other process. The advantage of using a procedure call is that there is always a return from the call which indicates to the calling process whether the procedure executed properly, and by implication, whether the information was properly received by the receiving process. Also, a procedure call may be invoked directly by the calling process without the need of invoking any kernel facilties although this is because the entire application has been compiled and linked into a single load module In a multi-processor system, a remote procedure call may be used to implement a program distributed across several processors. Schravastava and Panzieri [38] describe remote procedure calls in terms of client and server, which correspond to the invoking and the invoked procedures respectively.

When a client wishes to invoke a server (remote procedure), it must send a message over the inter-processor link The client will then wait for the server to return a reply message when it has completed execution Although the program syntax is in the form of a procedure call, the mechanism is actually an invoke message followed at some point by a reply message.

5.4. The World Model

An example of a world model function may be given by the task C1 (figure A2 1) for checking whether the welding tip has been inadvertantly frozen to the metal surface to be welded If the task is active, then it knows that a very high temperature and voltage is to be expected. Under some circumstances a high temperature may be considered dangerous For example, high temperatures are not expected when the task D1 is executing and the gap is merely being inspected to ensure that its dimensions are within accepted tolerances But if C1 is executing, then we know that D2 has invoked it and the context indicates that high temperatures are expected. If the temperature should dip and the voltage drop to zero, then the task knows that the tip has been frozen and an evasive action should be initiated This is the same as the context information function provided in the NBS world model. There may also be a task defined to implement learned behaviour. Taking the system illustrated in figure A2.1, we may define an exception processing task

D3 into which most or all sensory processed information is sent, and whose sole task is to check for previously recognized patterns in order to take evasive action. The patterns to be recognized may be hard coded, or the task may be equipped with a heuristic that allows it to correlate sensory patterns with task results.

The world model may be implemented as an abstract data type [37], i e , an object comprised of data structures plus commands which operate exclusively on those data structures This gives advantage to the programmer through information hiding The compiler or development system takes care of the details of structures and memory blocks and access to them is only through a clearly defined set of procedures Since the world model is intended to represent the robot's internal and external world, there would then be two data structures: a sensory data structure and a command data structure. A sensory data structure would store all processed sensory information provided by a single task and a command data structure would store the currently executing commands in a task.

In the NBS proposal, communication of world models is defined in one direction, from H module to G module: It provides context and prediction to the G modules. However, the world model might also be defined to communicate in the reverse direction, from G module to H module, implying that its function include learned response.

If the world model's sensory data structures contain an up-to-date representation of all sensory information, then it would represent everything the robot knows about its external world. This would allow the world model to recognize patterns which may elicit a preprogrammed or learned response different from what is presently executing in the H module. For example, if the robot were trying to achieve a certain goal but met with no success, then it might consult the world model to see if another sequence of commands is permissible given the present state of the world as indicated by the sensory data structure. This approach to the world model thus seeks to enhance the NBS concept of world model to provide for the capacity of learned behaviour. This capacity might be presumed to exist already within the cross-coupled processing hierarchy, but it would have to constitute part of an H module

Application development may use the world model as a structuring tool; for example, if the robot application were to be written using Modula, then each level might be written as a module containing one world model and a number of tasks. Tasks would communicate using procedure calls, and the world model might be defined as a monitor thereby ensuring mutual exclusion Building up the system may involve writing one module per level each containing a world model and several sensory and command tasks associated with it. The modules would be defined from level one to level n and be linked together into a load module Externally defined system calls would allow inter-process communication between the modules. Or, the world model may be distributed throughout the system, with each pair of sensory/command tasks having access to their own private world model Access to the world model of another pair in the system would require a command to its associated command task

The world model may also be associated with procedures to implement adaptive learning and heuristic behaviour. A world model which is capable of self modification in real-time would present problems if there were no reference point against which it could compare itself The NBS proposal does provide for a means of statically modifying or 'pre-setting' the world model where the programmer provides the real world reference point.

Exception Processing

Implementation of exception processing requires the inclusion of task pairs at a layer where the exception can be identified and appropriate action taken A reflex-type action will necessarily be on a lower layer, e g., in the case where an evasive manoeuvre is executed when visual sensors detect a sudden massive change in the perceived pattern; in such a case, the pattern has not even been recognized yet by the higher centres We may refer to the task pairs as a sensory exception task and an assassin respectively. The sensory exception task should be able to receive messages from all lower-level sensory tasks, and the assassin should be able to execute or to suspend any of the lower-level command tasks. For example, as soon as the sensory exception task detects an unidentified object in the robot's workspace, a message is sent to the appropriate world model to check whether the command context changes the way in which the sensory information should be interpreted. If the sensory information indicates a definite exceptional condition, a message indicating this fact is sent to the assassin which can then take evasive action. Sensory exception tasks and assassins are assigned a high execution priority but would be expected to execute infrequently.

6. An Approach to Development Concurrent Hierarchical Control

A distinction can be made between a robot control system and the operating system which must be designed to support it. This is the workbench approach to application design, in contrast to the monolithic approach of building a dedicated application as a single program The system remains constant and provides facilities to the application. This allows the programmer to design and load different tasks onto the existing system as opposed to redesigning and relinking the entire system each time, therefore improving programmer productivity There is much which is provided in a multi-programming operating system which is not required in a real-time system. Particularly such facilities as process creation and termination, memory block management, virtual memory and file management may either not exist or exist in a very different form in a real-time system. By contrast, other facilities such as interrupt handling and message switching may be optimized

This chapter describes a proposal for the design and implementation of an operating system which supports Concurrent Hierarchical Control based on the principles outlined in the previous chapter, and which will be referred to as Concurrent Hierarchical Control Operating System, or CHC-OS. CHC-OS is an attempt to apply operating system design principles to the problem of a robot control system while maintaining the integrity of the NBS work Much of CHC-OS has already been simulated and proven to be viable, in particular the task manager and message manager.

A significant feature of the proposed operating system is that it is a message-based system, implying that the sending of messages between processes or devices and processes is optimized. To ensure that the receiving process maintains an up to date view of the world, exclusive priority attention may be given to the most recent message sent. Previous messages may be considered out of date and no longer valid. This is in contrast to a timeshared operating system where all messages are queued and processed FIFO and no messages are superseded. Another significant feature is that the operating system is intended to support a multiprocessor hardware configuration and make it appear as a single system to the programmer. It is because of this that the system modules controlling messages and communications have been given their prominent positions in the OS hierarchy so that a single set of message sources and destinations may be defined across the system. Since the task force configuration may only be added to and not taken away from, i.e., new tasks may be installed but no task may be terminated, then it is possible to keep the same message table in each processor and to make updates of the installation of a new task in a single processor by broadcasting to all processors.

The layered structure of CHC-OS was modelled after the design of the Xinu operating system which is a scaled-down version of Unix [26], and after Brown and Denning's work on multi-layered operating systems [25].

Appendix three provides a list of procedure calls for CHC-OS.

6.1. Architecture

6.1.1. Feedback Control Loop

The feedback control loop as described by Albus et al is a fundamental component of any process control application and has been in wide use for a long period of time. The principle of feedback control is easily understood and examples abound of its use in everyday life, such as thermostats in a home or motorcar. CHC-OS uses feedback control loops as the atomic building block, unlike the cross-coupled processing hierarchy in which each feedback control loop is made up of three finite state machines which make up a single level. The reason for making the feedback control loop the fundamental building block is to render the control problem more manageable by defining feedback control loops for specific functions rather than for every function which must exist at a certain level of abstraction. It is conceivable therefore that a single level in the cross-coupled processing hierarchy may require several feedback control loops if implemented under CHC-OS.

The feedback control loop (figure 6 1) contains a sensory processing procedures, command processing procedures, and world model procedures and each loop contains the functionality exhibited by a level in the cross-coupled process-

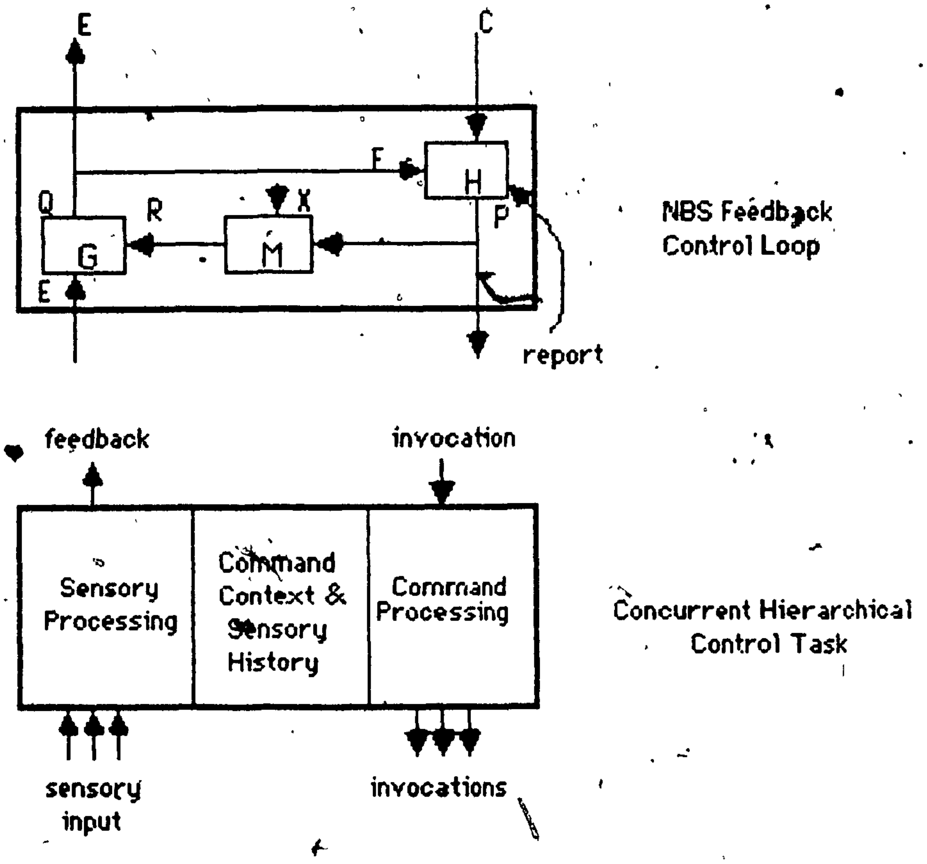

Figure 6.1

ing hierarchy. It is intended that each loop be written in C and the nature of the program written is left to the programmer; a loop may be written as a finite state machine or it may be written using heuristic algorithms. The only constraints imposed on the programmer are the input/output conventions to be followed in order to integrate the loop into the system. The key of CHC-OS is that each loop is a single task and may execute independently of other tasks according to priority.

6.1.2. OS Structure

The structure of the operating system designed to support Concurrent Hierarchical Control is described in this section and is represented diagrammatically in figures 6 2

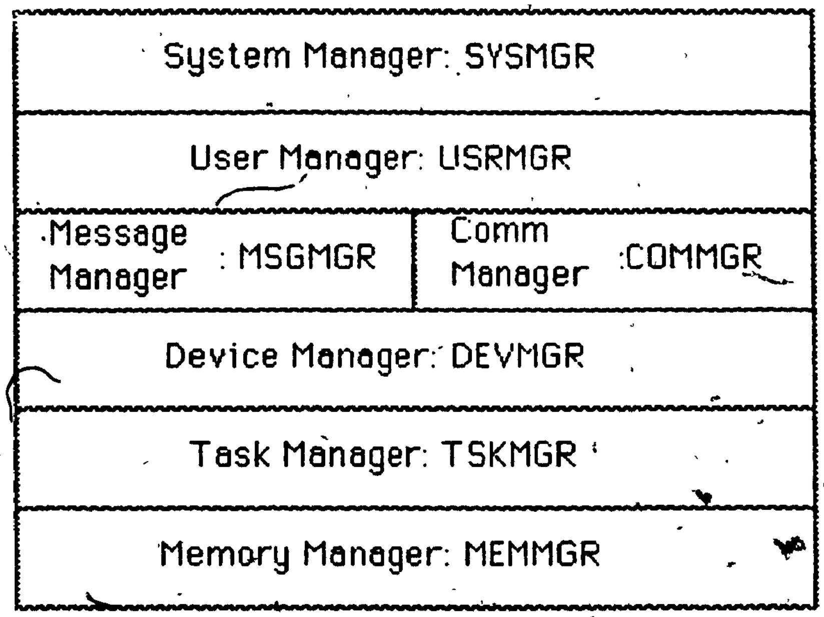

Figure 6.2

and 6 3 A detailed description of the procedure calls which have been defined at each level is provided in Appendix three. In the remainder of this section each level is described and illustrated with example procedures written in C. CHC-OS is not synonomous with Concurrent Hierarchical Control since the latter includes an approach to program development and user environment; nonetheless, it is an essential component of the proposal.

A scaled-down simulated version of CHC-OS was written and executed in order to test some of the concepts on which it is based. The hardware consisted of two Cadmus 9000 computers connected by means of Ethernet; the Cadmus 9000 is based on the Motorola 68000 micro-processor and supports Unix. Each level of the OS was coded in C, although not all features described were supported. It was demonstrated that inter-processor message handling, the scheduling of tasks according to message priority, the definition of Unix-like system calls to support low-level robot tasks and the flexible partitioning of robot applications over several processors were all viable concepts worthy of further research.

The Kernel

The kernel supports a small number of low-level utilities which includes context switching, device IO and the communications link. The kernel is the only part of the system which needs to be written in assembly language in order to interface with the hardware. All other modules are written in C.

The Memory Manager· Memmgr

Dynamic allocation of memory segments is not required since in a relatively static system, tasks can be loaded using absolute memory addressing. However, stack and queue management procedures are required to support procedure calls and task management. Stack space may be allocated

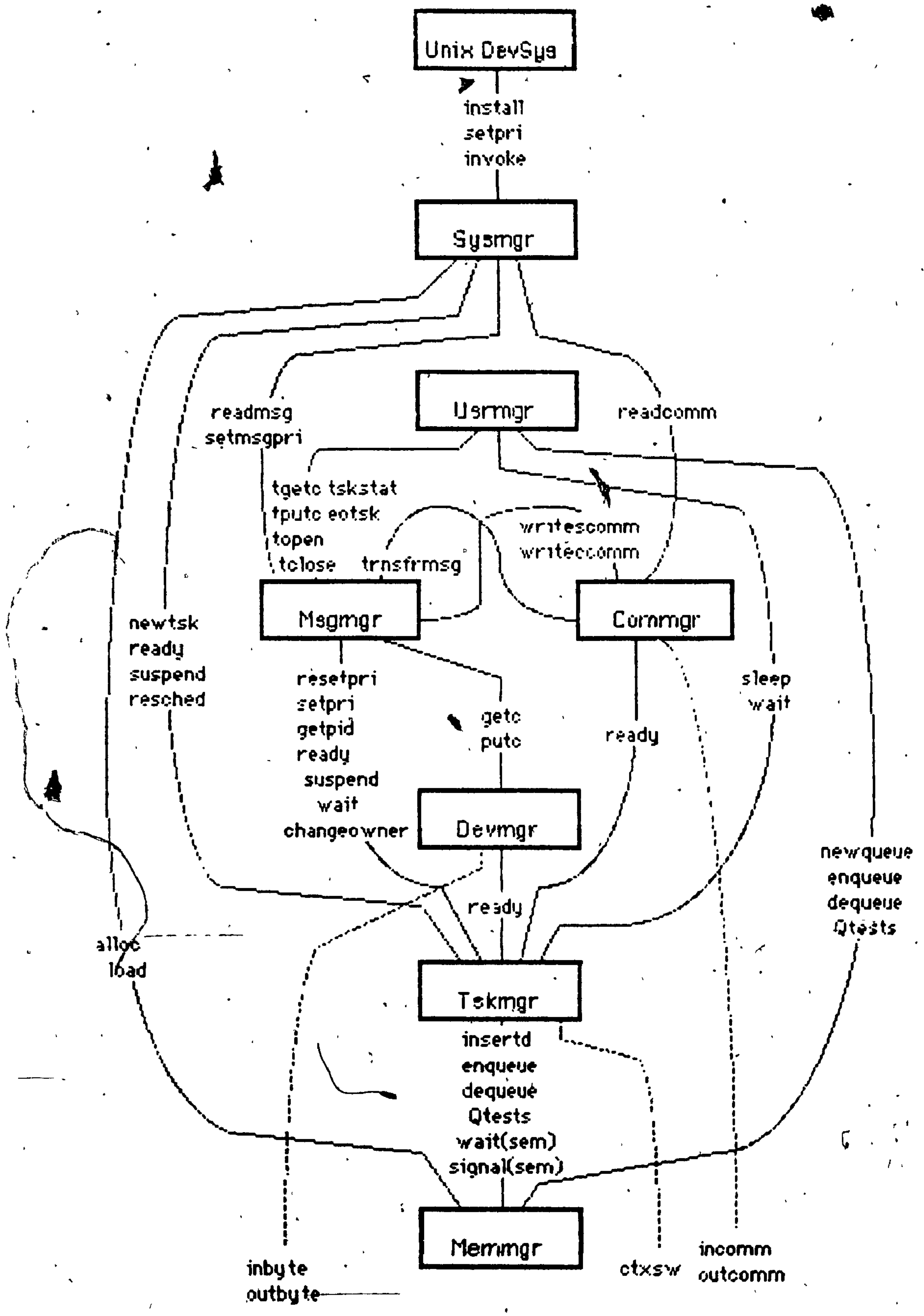
Unix DevSys
install
setpri
invoke
Sysmgr
readmsg
setmsgpri
Usrmgr
readcomm
tgetc tskstat
tputc eotsk
topen
tclose
trnsfrmsg
writescomm
writeccomm
newtsk
ready
suspend
resched
Msgmgr
Commgr
resetpri
setpri
getpid
ready
suspend
wait
changeowner
getc
putc
ready
sleep
wait
Devmgr
newqueue
enqueue
dequeue
Qtests
ready
alloc
load
Tskmgr
insertd
enqueue
dequeue
Qtests
wait(sem)
signal(sem)
Memmgr
inbyte
outbyte
ctxsw
incomm
outcomm

Figure 6.3

dynamically and is manipulated using either stack or queue procedures. The following procedure implements the procedure 'dequeue' which is used to remove an item from a queue whereever it is located on the queue:

```
struct tsktabentry rdyQ[20];
dequeue(item)
int item;
{
          struct tsktabentry *mptr;
          mptr = &rdyQ[item];
          rdyQ[mptr->Qprev].Qnext = mptr->qnext;
          rdyQ[mptr->Qnext].Qprev = mptr->Qprev;
          return(item);
}
```

<u>The Task Manager</u>· <u>Tskmgr</u>

The task manager is responsible for scheduling tasks to run on a processor. It keeps tasks in various queues, indicating their present state, such as ready, suspended, sleeping etc. The task manager may be called to reschedule the processor, usually after an interrupt has been received, where it will choose the task on the ready list with the highest priority and give control of the processor to it. Since there is no requirement for virtual memory and page swapping, the executing task environment can be saved and the new task environment restored with great efficiency.

The task manager also provides procedures which can alter the state of tasks and manage semaphores. Also, each task has defined to it a set of input and output ports which are used for message passing. An example of a task manager procedure is provided below:

```
resched()
{
        struct tentry *optr,      /* old regs */
        struct tentry *nptr,      /* new regs */
        optr = tsktab[current];
        if((optr->state==CURRENT) &&
                (lastkey(rdytail)<optr->prio))
                return(OK);
        if(optr->state==CURRENT) {
                if((gettime()<LIMIT) &&
                     (lastkey(rdytail)==optr->prio))
                     return(OK),
                }
        nptr = tsktab[(currtid=getlast(rdytail))];
        nptr->state = CURRENT,
        enqueue(currtid,currtail);
        ressetime(),
        ctxsw(&(optr->ps),&(nptr->ps)), /* switch */
}
```

The Device Manager Devmgr

The device manager handles interrupt drivers for all physically connected devices, including sensors, actuators and any other device required to operate the system such as discs, consoles or communication lines The purpose of the device manager is simply to effect the transfer of an input byte to a buffer or an output byte to a port . The interrupt drivers are written in C When a processor has been interrupted because of an incoming byte, then the device manager interrupt routine selected by the interrupt vector executes, and reads the byte off the device port The task manager then selects a user-defined device driver and places the byte in the task's appropriate input buffer and schedules the task for execution. While the interrupt routine is executing, the interrupted task is still the owner of the processor, even though the processing it is doing is likely of no interest to it At the end of the interrupt routine, typically the processor will be rescheduled and control may pass to another task; this is same way interrupt routines are handled in Unix. If another byte is read before the device driver task has been able to use the previous byte, then the previous byte is overwritten in the interests of prioritizing the most recent information

All the device interrupt routines for all processors are included in the device manager, however only a subset of the total number of devices will be attached to any one processor. There are two ways of dealing with this situation: first, a standard set of interrupt vectors may be loaded into each processor as long as no two devices anywhere in the system are associated with the same interrupt number; second, the loader determines what interrupt routines to associate with a vector according to a processor/device/address table at load time

An example of a device manager procedure which reads devices is provided below:

```
readdev(dev)
int dev,
{
        if(devtab[dev].type !='r')
                return(SYSERR);
        switch(devtab[dev].state) {
                case DEVREADY:
                        devtab[dev].flag = RECEIVED;
                        return(read(devtab[dev].fp));
                        break;
                case DEVBUSY:
                        wait(dev),
                        break,
                }
}
```

<u>The Communications Manager: Commgr</u>

The Communications Manager supports inter-processor communication. A task may write a sensory or command message directly to a destination in any other processor since it can initiate a message to another machine, however it cannot read from another machine since it cannot know what is the intended destination of any incoming message. Any incoming message causes an interrupt scheduling the system task (TSKO) for execution which then calls the Commgr to read the incoming message and determine its type and destination. The following procedure is an example taken from Commgr.

```
readcomm(bus,srctype,srcid,srcport,msg)
char *bus;
int *srctype, *srcid, *srcport, *msg;
{
        FILE *fp;
        fp = opendev(bus,"r"),
        if(fscanf((fp,"%d %d %d %d",&srctype,&srcid,
                &srcport,&msg)==EOF)) {
                closedev(fp);
                return(EOF);
                }
        else {
                closedev(fp);
                initdev(bus);
                }
}
```

<u>The Message Manager: Msgmgr</u>

The message manager is the level at which all message passing between sources and destinations in the system is controlled. A source or a destination may be a task, a device or a Unix file. Whenever a message is passed to a task, the message manager will call the task manager to ready the receiving task and to reschedule the processor on the principle that the sending of a message may make some waiting or suspended task ready for execution It is the message manager which is primarily responsible for task scheduling in the processor. The following procedure is used to write a sensory message from a lower to a higher-level task:

```
tputc()
int dst;
char msg;
{
        struct msgtabentry *msgid,
        int src
        src = gettid(),
        msgid = msgtab[src,dst]
        if(msgid==NULL)
                return(SYSERR);
        switch(msgid->status) {
                case OK:
                        usrtab[dst] inport[msgid->dst] = msg;
                        enqueue(Qprio,tsktab[dst].prio);
                        tsktab[dst].prio = msgid->prio;
                        break;
                case SUSPENDED:
                        return(SUSPENDED);
                        break;
                default:
                        return(SYSERR);
                }
}
```

The User Manager: Usrmgr

The user has has no routines uniquely associated with it but nonetheless is an important part of the system Only one other level exists above the user manager and it does not require any Usrmgr routine in order to manage tasks since this may be done through the task manager.

Usrmgr acts as the interface between the user programmer and the operating system and has access to a set of GHC-OS system routines which are exported through a header called 'Usrmgr h'; this ensures that the programmer has access only to those system routines which are of interest to him. User tasks have access to a subset of subroutines in Msgmgr for effecting message transfer, in Memmgr for user list management, and in Tskmgr to allow the task to put itself to sleep or optionally to wait after invoking a lower-level task. When invoking a lower-level task, a user task may wait until the invoked task has reached completion or has been pre-empted, or it may continue executing concurrently. If topen() is called to invoke the task and the invoked task already belongs to the invoking task, then there is no change.

Device drivers are also defined as user tasks and are under user control. CHC-OS is intended merely to effect byte transfer to or from the device drivers and does not imply any interpretation of the byte stream.

The System Manager: Sysmgr

The system manager is uniquely associated with the task TSKO since this the only task which has access to Sysmgr routines. TSKO is scheduled by the communications manager for execution when a message is being received on an incoming inter-processor communications link, although the same is not true for outgoing inter-processor communications since that may be done by any user task by using the message manager. The system manager is also scheduled for execution by the clock interrupt routine in the device manager and by other hardware interrupts such as the communications ~~port~~. It must therefore be able to distinguish interrupt types and to do the housekeeping associated with each type. The interrupt routines handle the system clock and emergency interrupts such as power failure or segmentation fault.

The system manager also acts as the interface between the Unix system and each satellite processor and is defined with a set of procedures to permit that interface The procedure presented below is used to implement a simple invocation from the Unix Development System to any task in a satellite processor.

```
invoke(tsk),
int tsk;
{
        ready(tsk),
        resched();
}
```

6.1.3. Operating System Tables

As important as the definition of system subroutines for an operating system are the system tables. Most timeshared computer operating systems have a standard complement of tables which are used to keep track of system resources. Such standard tables include free memory lists and maps to virtual memory, process tables and device tables. Since CHC-OS does not need to handle a dynamic timeshared environment, the table structure is simplified. The task table does not need to contain quite as much information as a process table; the free memory list only needs to bother about allocating memory but not garbage collection. Virtual memory is not required. The primary difference is the emphasis placed on messages in CHC-OS, exemplified by the fact that a system message table has been defined for the use of the message manager in order to coordinate all inter-task communication in the system. Also, in a multi-computer environment, the tables do not need to be updated from one system to the next, since every processor has an up-to-date view of what every other processor contains. A description of each table is given below and the table structures are provided in Appendix three.

The memory table 'Memtab' keeps track of remaining unallocated memory as a single block. Memory is allocated upon request as the application system is being built up until there is not enough room left in the memory block.

The task table 'Tsktab' provides the environment for context switching. An entry in this table provides the task's name, status, etc. Although within a feedback control loop many of the individual functions have been referred to as tasks for the sake of discussion, the only object which is understood by Tsktab is one entire feedback control loop which is scheduled for execution as an entity.

The device table 'Devtab' contains entries for devices of two types, sensors and actuators, therefore devices are either input or output and not both.

The message table 'Msgtab' contains information relating to each inter-task link and defines message as a system object in its own right. A Msgtab entry includes information regarding the priority of a message, source and destination, and present status such as response pending.

With each task is also associated a user structure referred to as 'Usrtab' which contains a place to which the user task has access and where system and user procedures may interact. Specifically, Usrtab contains eight input and eight output message ports which are assigned by the install program when the task is installed.

6.1.4. Hardware

Concurrent Hierarchical Control is intended to run on a set of Motorola 68000 based processors (figure 6.4). The arrangement of processors is hierarchical in two levels; one mother processor runs Unix and is required for development and downloading, and one or more satellite processors are loaded with CHC-OS from Unix. The Unix processor supports terminals, discs etc., while the satellite processors support only sensors and actuators and require no other peri-

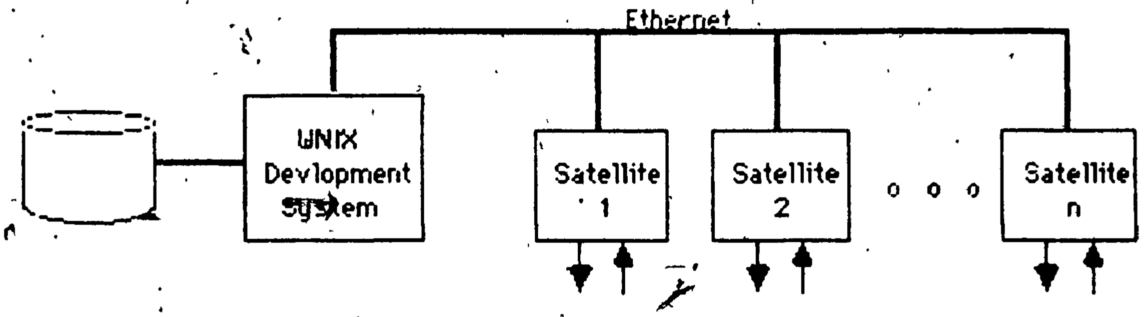

Figure 6.4

pherals. Sufficient memory is installed in each satellite processor to abrogate the need for virtual memory. Communications takes place through an Ethernet serial link.

A simulation of several modules of CHC-OS has already been written using a two processor configuration Each processor was a MC68000-based Unix system and communicated with the other by means of Ethernet The modules which were simulated in order to demonstrate the viability of the architecture included the memory manager, task manager, and message and communications managers

6.1.5. A System Example

In figure 6 5 is provided an example of a system which indicates how various capabilities of Concurrent Hierarchical Control and CHC-OS may be exploited In the example, a set of sensors, S1, S2 and S3 have been defined along with a set of actuators M1 and M2; the sensors and acuators do not necessarily reside on the same processor. Another set of input/output objects have been defined in order to help in system development, control and monitoring, these are a Unix input file, 'testfile', a Unix output file, 'log' and a Unix terminal, 'tty0'. The tasks A1 and A2 are user-defined drivers which read inputs and calculate the appropriate outputs as feedba[illegible] control loops The tasks A3 and A4 are drivers provided by [illegible]e system in order to interface to the Unix file system.

Taking the sensor S1 as an example, if an interrupt is received by task A1 from S1 and A1 has been opened for execution by B1, then it will execute at the priority level assigned to it by the B1/A1 command message and check its

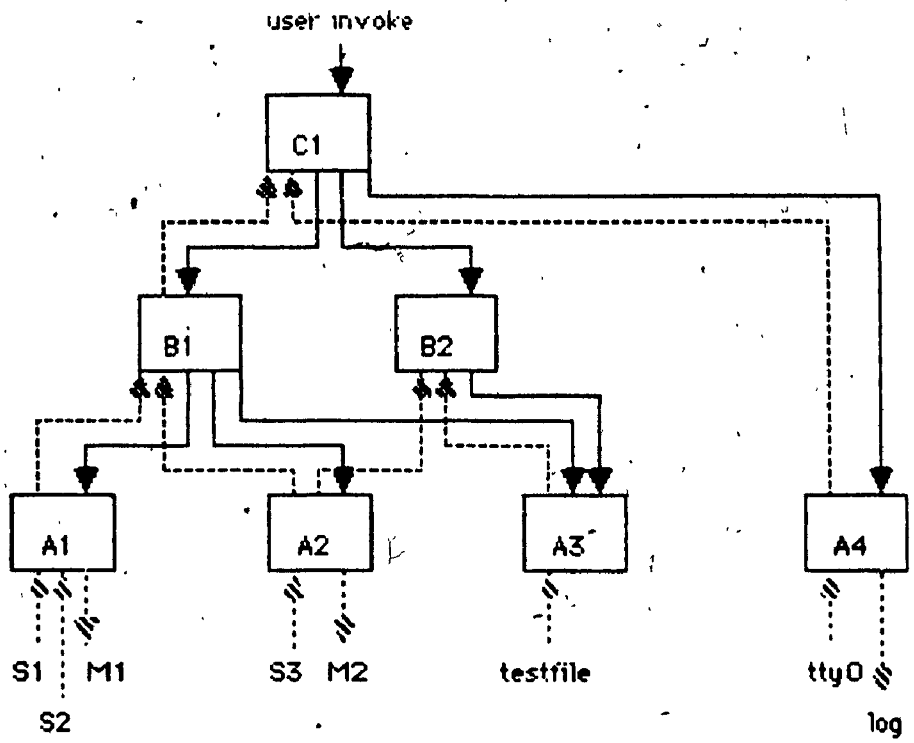

Figure 6.5

inputs A1 will find that a message is waiting to be picked up from S1 and may also check to see if a message is waiting at S2 as well If there is no message waiting at S2, then the read may be blocked or non-blocked depending on the desire of the programmer. A1 will then execute and produce two outputs, one for the actuator M1 and another to send to the task B1. Also, if A1 completes its task, it will return an eotsk, or end of task, message to B1 which terminates the B1/A1 command message

Task A3 is a driver provided by the system which interfaces to a Unix file by sampling from it at a certain rate which is determined by the person downloading the system from Unix. This provides a means by which a programmer can enter test or other data into the system Task A4 is a system provided driver which receives input from a Unix file defined as a TTY device and another file defined as an output log to record pertinent system events . The interfaces with tasks A3 and A4, i.e., the input/output ports, must be defined to Unix through its file system thereby allowing dynamic user access to the Concurrent Hierarchical Control system by means of a file interface

The system example also illustrates exception processing. The command message C1/B1 may indicate that an exceptional situation has arisen and that B1 should therefore take control of all A-level tasks and perhaps shut them down. The message has a high priority which would cause the task B1 to execute at a high priority. Message priorities are set from Unix and may be done dynamically allowing the programmer to fine tune the system.

6 2. Application Development

All user tasks are written in C and compiled in the Unix mother processor The programmer must include the file 'sysfile.h' in order to gain access to system procedures, interrupt routines, and buffer addresses 'Sysfile.h' may also be edited by the programmer in order to insert sensors and actuators. Each sensor or actuator needs to be identified by name, home machine and hardware interrupt number. The programmer will then compile 'sysfile.h' with 'sysfile.c' and other files through a Unix 'make' command in order to create 'sysfile' which will be the new system object module by which CHC-OS is loaded into each satellite processor. Discrepancies from satellite to satellite are resolved when sysfile is downloaded

The procedures which are available to the user programmer in sysfile.h are quite few but may be increased by the user since most of CHC-OS is written in C. The procedures which are available are shown below with their accompanying parameters.

```
sensory input:    tgetc(port[8],&userbuff)

sensory output:   tputc(userbuff)

command output(invoke,complete):
                  topen(task,parms)
                  tclose(task)
                  teotsk()

task management:  sleep(self)
                  wait(self)

memory management:
                  newqueue(list)
                  enqueue(list)
                  dequeue(list)
                  Qtests(list)
```

Once a program is written in C, it is then compiled into an object file like any other program in C; for example, if the user were to write the driver for A1, then his source file would be named A1.c which would include file sysfile.h. The object file produced by the compiler would normally be A1.o, indicating that it has been compiled without linking it to run under Unix; however for the sake of simplicity, the object files have been presented in the next section without the suffix ' o'

6.3. Downloading

Downloading refers to the way in which the application is set up in the satellite processors from the Unix mother processor. There are two aspects to downloading: one is from the point of view of Unix in the source machine and the other is from the point of view of CHC-OS in the destination satellite machine. A new command 'install' is defined in Unix in order to allow the user to initiate the downloading, and another command 'install' resides in the system manager of each satellite machine and is prompted by an inter-processor invocation to TSKO. The procedure 'install' is analogous to the Unix 'urun' command which allows the invocation of a process on a foreign processor and may actually be implemented with 'urun' in a Unix shell-script. When the system is first being brought up, the first file to be installed is 'sysfile' which loads CHC-OS into each satellite. This requires that the loader on each satellite monitor the Ethernet to determine when it is about to receive an object file. There is only one copy of 'sysfile' with interrupt routines to handle all devices on the system; however, since the file sysfile.h has been included in sysfile, there is enough information to tell the loader which devices are on which satellite processor. The loader must therefore determine through sysfile which devices are attached at which interrupts and must then load the interrupt vector table with the addresses of only those interrupt routines it requires Once the loading is completed, control is given over to the system manager, i e , TSKO

The Concurrent Control Hierarchy is built up incrementally. Any message path implies an upward or downward direction depending on whether it is a sensory or a command message. No sensory messages are ever exchanged from one task to another which is at the same or a lower level, similarly no command messages are ever exchanged from one task to another which is at the same or a higher level TSKO must monitor the Ethernet for any other installs at any time. The Unix 'install' will broadcast to all satellite processors when an 'install' is to take place. Even if a satellite is not selected to load a new task, it must nonetheless update its own system tables to reflect such a system-wide change. It is through the thoughtful use of 'install' that these rules are maintained, although 'install' must also be able to detect when the rules are being broken and the nature of the hierarchy is threatened There is also a place for writing a program in Unix which will read'sysfile map' and output a real-time graphic representation of the hierarchy of tasks which is being built up under CHC-OS

The command syntax for 'install' follows the Unix convention of command followed by options followed by file names. Therfore, the command to load a satellite would have the following syntax where the option '-m2' refers to satellite processor number two

```
install -m2 sysfile
```

In Unix, system information is maintained in a file called 'sysfile map' which is updated with each 'install'. It contains information regarding the tasks which have been loaded into all satellite processors When a new 'install' is to be executed from Unix, 'install' will first consult sysfile.map in order to ensure that the tasks with which the new task will communicate already exist in the system. An 'install' of a sysfile implies that the satellite is to be reloaded from scratch and that discrepancies may be created in sysfile.map

The full format of the 'install' command is as follows:

```
install [-m#i#o#]
        [tsk]
        [infile1. infile#]
        [outfile1..outfile#]
```

where 'm#', 'i#', and 'o#' in the option field refer to machine number, number of input files and number of output files respectively, 'tsk' refers to the object file to be loaded as a task, 'infile' refers to sensors, Unix files or other tasks to be used as sensory input; and 'outfile' refers to actuators, Unix files or other tasks to be used for command output, i.e., other task to be invoked The pound sign '#' refers to any integer. Another command, 'setpri', may be used from Unix in order to set the priorities of the message links, and by implication, the execution priority of the tasks which are the destination of those links. The fact that this command may be executed dynamically allows the programmer the chance to fine tune the message priorities of the system The format of the setpri command is as follows:

```
setpri [-#] [srctsk dsttsk]
```

where '-#' refers to the priority given the message, for example 0..99, and 'srctsk' and 'dsttsk' refers to the source task and the destination task of the message The source, destination and implied upward/downward direction will identify the message by type and number to the message manager

We can take the example system illustrated in figure 6.5 and show how part of it might be loaded from Unix onto a set of three satellite processors This may also be used to illustrate how the CHC-OS system tables will be altered by a single 'install' command. Let us take the example of installations in machine 0 and show in a simplified way the entries in CHC-OS's system tables just after it has been loaded with sysfile

Tsktab0.	TSK0
Devtab0:	S1 S2 M1
Msgtab0:	no entries

Then the following installs are executed from Unix:

```
        install -m0i2o1 A1 S1 S2 M1
        install -m0i2o3 B1 A1 A2 A1 A2 A3
```

Then all tables in each machine will have been changed to reflect the installation.

```
        Tsktab0:        TSK0
                        A1
                        B1

        Devtab0:        S1
                        S2
                        M1

        Msgtab0:        S1/A1   sensory
                        S2/A1   sensory
                        A1/M1   command
                        A1/B1   sensory
                        B1/A1   command
```

All the 'install' and 'setpri' commands can be included into a single Unix shellscript or command file allowing the user to 'install' all tasks with a single command. This is done once an application was prepared for a commercial environment. Following is a shellscript which would be used to load the system illustrated in figure 6.5 onto a system of three satellite computers:

```
        install -m0     sysfile
        install -m1     sysfile
        install -m2     sysfile
        install -m0i2o1 A1 S1 S2 M1
        install -m1i1o1 A2 S3 M2
        install -m2i1   A3 /usr/testfile
        install -m2i1o1 A4 /dev/tty0 /usr/log
        install -m0i2o3 B1 A1 A2 A1 A2 A3
        install -m1i2o1 B2 A2 A3 A3
        install -m2i1o3 C1 A4 B1 B2 A4
        setpri  -9      A4 C1
        setpri  -9      C1 B1
```

6.4. System Performance

There are many reasons for believing that the enhancements which CHC-OS provides to the cross-coupled processing hierarchy will result in a substantial increase in system performance. There are two ways in which system performance has been estimated. the first way is to make a direct comparison between Concurrent Hierarchical Control and the cross-coupled processing hierarchy based on the example illustrated in figure 6 5. This provides a very broad estimate of performance based on the processing requiring to handle sensory input messages; the second way is to make an approximation of the number of software and hardware interrupts which can be supported by Concurrent Hierachical Control, i e., how quickly it can interact with its environment in the form of its sensors and actuators This provides a more quantified estimate of performance.

For the first case, let us take a subset of the tasks indicated in figure 6.5, specifically removing only task A4 which provides the user interface to Unix This provides us with the system shown in figure 6.6 The functions supported by the tasks in figure 6.7 must be partitioned somewhat differently in order to conform to the cross-coupled processing hierarchy; specifically, all the concurrent tasks on a given level must be combined into a single feedback control loop. It is assumed that the execution of a task under Concurrent Hierchical Control is roughly similar to the execution of an

FSM (finite state machine) under the cross-coupled processing hierarchy. The partitioning has been done in the following way:

```
A1 + A2 + A3     ==>     H1 + M1 + G1
B1 + B2          ==>     H2 + M2 + G2
C1               ==>     H3 + M3 + G3
```

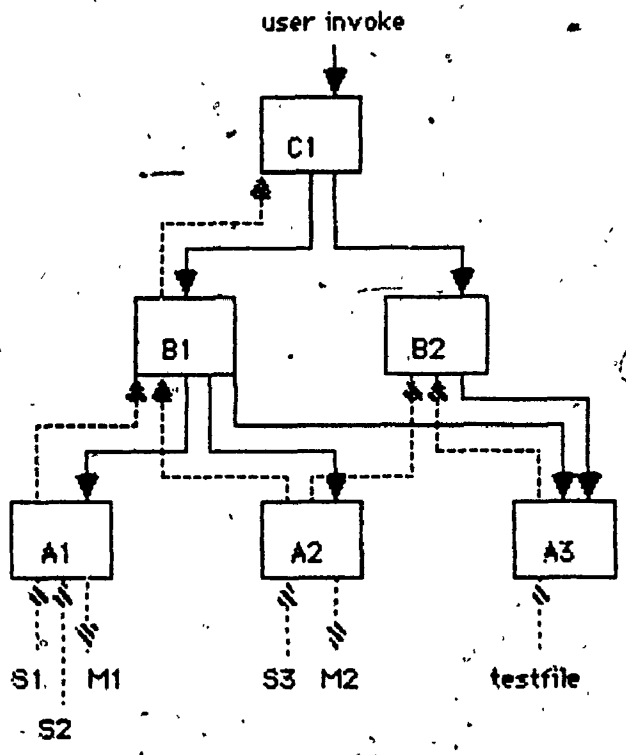

Figure 6.6

The resulting structure is shown in figure 6.7.

We now assume that each sensory input to the Concurrent Hierarchical Control system, i.e., S1, S2, S3 and testfile input a message once every 100 milliseconds. Since the messages are received by interrupt, there is then a total of 40 messages received each second each one causing a driver task to execute. Let us assume also that five inputs to a task

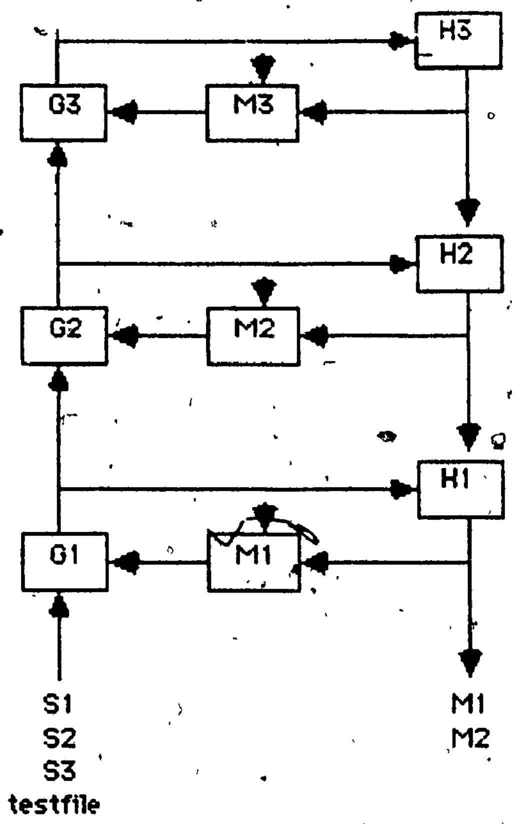

Figure 6.7

at level n will cause a sensory output to a task at level n+1; therefore in our example, there will be a total of 51 task executions/sec or one execution/20ms. If we apply the same principle to the cross-coupled processing hierarchy, we assume that the G1 module must be able to handle at least 40 messages per second. Since the cross-coupled processing hierarchy is not interrupt driven, it must poll the sensors at such a rate that ensures that no messages be lost. Typically, a rule of thumb is to double the average rate in order to derive the polling rate, which in this case would be 80 times per second. Since there are nine FSMs which must execute 80 times per second, there are therefore 720 executions per second In other words, the cross-coupled processing hierarchy requires 720 FSM execution per second and Concurrent Hierarchical Control requires 51 task executions in order to process the same incoming sensory data. If the cross-coupled processing hierarchy were to execute round-robin once every 28ms as is intended by its authors, then there would be 3200 FSM executions every second. The difference in performance may be primarily attributed to the elimination of busywaiting, although in Concurrent Hierarchical Control there is an overhead implied by its greater sophistication which does not appear in this comparison.

The second means of approximating performance has been to estimate the processing required by hardware and software interrupts, which are analogous to sensor and actuator message transfers respectively. An estimate was made of the number of procedure calls each interrupt would generate and of the length of code each procedure would execute. A Motorola MC68000L8 was used as a basis for the performance estimate with a clock period of 8 Megahertz [39].

The objective of the estimate was to determine how many sensor and actuator interrupts per second could be handled by a system with one, two or three MC68000L8 satellite processors. An example system routine was chosen as a basis for making an estimate as to the size of an average system routine. The procedure 'resched' was considered to be of sufficient complexity. 'Resched' was taken from the task manager which was written as an early simulation for CHC-OS It is similar to the procedure by the same name written by Comer [26]. The procedure was compiled and an assembly language listing was produced which was then quantified in order to get the total execution time in machine cycles It was found that the procedure 'resched' would execute in a maximum of 300 cycles with a set-up time of 50 cycles for a total of 350 cycles per execution which was used as an average for the execution of all system procedure calls

Each interrupt is supported by a driver routine which is usually written by the user A driver routine needs to do more than just transfer data, some interpretation is also required. It was estimated that each driver routine when called would execute for about 1050 cycles or the equivalent of three procedure calls. The driver routine schedules a user task Given that each user task is probably a very tightly programmed feedback control loop, it was estimated that a message arriving as the result of a hardware interrupt or a software interrupt prompted by a message would require about 2000 cycles worth of processing within a user task. Each user task on interrupt or completion also reschedules the processor which takes two procedure calls at 700 cycles. This is summarized in figure 6.7.

It was previously assumed that relationship between the number of messages on a higher level with respect to a lower level was one to five, and that assumption remains valid for this estimate

It was also assumed that the system clock interrupt handler would execute once every 16 7ms and that the interrupt handler made 10 procedure calls in order to do such business as delta queue management, checking inter-machine communications input and rescheduling Therefore, the total for clock processing is 210,000 cycles per minute.

It was assumed that each interrupt required the execution of five system routines involving the transfer of the message to the right buffer and the readying of the receiving task, and similarly for all message transfers.

Therefore, a sensor or actuator interrupt requires.

1750	cycles for interrupt handling
1050	cycles for driver handling
2000	cycles for user-written portion of driver filtering, buffering
2100	cycles for message transfer
2000	cycles for consumer/producer task
700	cycles for rescheduling after task interruption or completion

In figure 6.7 is illustrated the estimates for processing time; the right hand side indicates processing required when all messages are sent to tasks within the same processor; the left hand side indicates the overhead incurred by using inter-processor communication to transfer messages between tasks on different machines. By assuming that every five messages at level n is associated with one message at level n+1, this means that each sensor/actuator interrupt is associated with 1.25 task executions, one for the driver and

0.25 for the total of higher level task executions There-fore, by summing the weighted values on the right hand side of figure 6 7, we can get the total processing required for a single interrupt in a one machine configuration.

In order to calculate the improvement which would be offered by a multi-machine configuration and to calculate the extra overhead required for inter-processor communica-tion, two assumptions were made. first that the number of incoming messages would equal the number of outgoing mes-sages, and second, that the ratio of inter-processor mes-sages to local messages in a machine would be proportionate to the ratio of total tasks in the system to local tasks. In reference to figure 6 6, it is illustrated that inter-processor communications produces interrupts also, although for our performance study we are interested only in sensor and actuator interrupts, therfore communications interrupts are simply considered to be part of the overhead. For an incoming message, the initial driver filtering has already been done in the remote processor, so the message is input-ted in the local message manager. The communications inter-rupt, the scheduling of the system manager etc., are part of the overhead. For an outgoing message, the communications manager procedures and software interrupt routine are added to the overhead. If we continue to take 0.25 as the number of higher level tasks executed per interrupt, in a two machine configuration we can say that 0 125 of the tasks are local and 0.125 are remote. Therefore 0.125 tasks per interrupt require inter-processor communications overhead Similarly, for a three machine configuration, the estimate is that for each interrupt, 0 083 tasks are local and 0 166 tasks are remote Having derived the total number of machine cycles required to process one interrupt and all tasks associated with it, we can divide it into 7,790,000 which is the total number of machine cycles remaining after clock processing has been accounted for

With this information the following table can be derived for multi-processor configurations indicating the total number of sensor/actuator interrupts which can be supported:

1 satellite	1150	interrupts/CPU/sec
	1150	total interrupts/sec
2 satellites	1050	interrupts/CPU/sec
	2100	total interrupts/sec
3 satellites	1025	interrupts/CPU/sec
	3075	total interrupts/sec

However, estimating the processing capacity required for a real-time interrupt-driven system is not a simple job. In the Albus proposal, this is greatly simplified by the purely deterministic nature of the finite state machine implementation. The Concurrent Hierarchical approach presupposes that a more flexible system based on asynchronous task execution is more appropriate to the problem of robotic control, although this makes the task of estimating CPU requirements rather difficult since not all possible scenarios can be anticipated. If additional CPU capacity is required, CHC-OS is conceived such that new processors may be easily added to the system and the additional overhead required easily estimated.

7 Conclusions

The main purpose of this thesis has been t[illegible]ine in detail the NBS proposal for a cross-coupled processing hierarchy for robot control and to propose a system which seeks to enhance its strengths The author has gone on to underscore the issues raised by the concept of robot control systems, and to suggest some ways in which the cross-coupled processing hierarchy might be enhanced to better exploit modern operating system concepts The principal result has been to define small feedback control loops as concurrent tasks to control various sensors and actuators, thereby eliminating ad-hoc methods of executing very large procedures (i.e cross time interval boundaries) and busywaiting.

In concurrent hierarchical control, each sensor/actuator group is controlled by its own control task, and each command is identified with a task, as opposed to a procedure pointer in the NBS design Since each command is a task, it may be treated independently All tasks may therefore execute concurrently on the same processor or on separate processors This sort of approach is flexible and allows a concurrent control hierarchy to be installed on a variety of hardware architectures The use of multiple CPUs exploits the concepts of parallel processing in order to assure fast computation in real-time

The world model is conceived to be a repository of all information relating to the state of the robot's internal and external environment. Therefore, the world model receives input from both command and sensory tasks. Concurrent Hierarchical Control distributes the world model among its tasks. This allows the world model to predict sensory information according to the command context, and to interpret and channel sensory feedback to the command task as well as providing context information to the sensory task. The world model also has the capacity for adaptive behaviour by virtue of the fact that it is capable of recognizing any set of sensory conditions which may elicit a previously learned response. This allows for the world model to choose among several plans of action when requested to do so by its command task.

Concurrent Hierarchical Control supports workbench programming because the operating system designed to optimize robot control systems remains a constant. Different sets of tasks can be loaded onto the OS without requiring the relinking of the entire system, which suggests that Concurrent Hierarchical Control is in some ways another robot language. However, what distinguishes Concurrent Hierarchical Control from being simply another robot language is the fact that the operating system conceived to support Concurrent Hierarchical Control been designed with a specific view to addressing the issues raised by real-time sensory-interactive robot control. Using timeshared operating systems as a basis, it adds features to optimize performance and takes away other features which would degrade performance.

8. References

[1] J.G.Fuller, "Death by Robot," OMNI, March 1984, New York NY.

[2] J.A.Miller, "Autonomous guidance and control of a roving robot," 5th International Joint Conference on Artificial Intelligence, 1977, Cambridge MA, pp 759-760

[3] A.M.Thompson, "The navigation system of the JPL robot," 5th International Joint Conference on Artificial Intelligence, 1977, Cambridge MA, pp.749-757.

[4] R.Popplestone, "Automatic assembly with the Edinburgh arm eye system," Industrial Robots, Birkhauser Verlag, Basel Switzerland, 1975

[5] A.G.Makhlin, "Robot control and inspection by multiple camera vision system," 11th Symposium on Industrial Robots, 1981, Tokyo, pp 121-128.

[6] R.Sugarman, "The blue collar robot," IEEE Spectrum, vol.17, no.9, Sept. 1980, pp.52-59.

[7] T.Lozano-Perez, "Robotics," Artificial Intelligence, vol.19, no.2, Oct. 1982, pp.821-840.

[8] D.Nitzan et al, "The use of sensors on robot systems," Proceedings of the International Conference on Advanced Robotics, Tokyo, 1983, pp123-132.

[9] C.Laugier, "A program for automatic grasping of objects

with a robot arm," 11th International Symposium on Industrial Robots, 1981, Tokyo, pp 287-294

[10] C.S.G Lee et al, "Hierarchical control structure using special purpose processors for the control of robot arms," IEEE 1982 PRIP-Pattern Recognition and Image Processing Conference, pp 634-640

[11] A.Mercer and G Vincent, "Controllers built using function of function architecture," The Industrial Robot, vol 9, no.4, Dec. 1982, pp.228-232

[12] L.Friedman, "Robot learning and error correction," 5th International Joint Conference on Artificial Intelligence, 1977, Cambridge MA.

[13] G.N.Saridis, "Intelligent robotic control," IEEE Transactions on Automatic Control, vol AC-28, no 5, May 1983, pp.547-557

[14] H.Jappinen, "Sense controlled flexible robot behavior," International Journal of Computer and Information Sciences, vol.10, no.2, April 1981, pp.105-125

[15] R.G.Abraham et al, Westinghouse Research and Development Center USA, International Fluidics Services, "State of the art in adaptable-programmable assembly systems," Kempston Bedford UK, 1977.

[16] J.S.Albus et al, "Hierarchical control for robots in an automated factory," 13th International Symposium on

Industrial Robots, Chicago, 1983, pp 13.29-13 43.

[17] A.J Barbera et al, "Concepts for real-time sensory-interactive control system architecture," IEEE 14th Annual South-Eastern Symposium on System Theory, 1982, pp.121-126.

[18] J.S Albus et al, "Programming a hierarchical robot control system," 12th International Symposium on Industrial Robots, Paris, 1982, pp.505-517

[19] J.S Albus et al, "Hierarchical control for sensory interactive robots," 11th Symposium on Industrial Robots, 1981, Tokyo, pp497-505.

[20] J.S.Albus, Brains Behavior and Robots, McGraw Hill-Byte, Peterborough NH, 1981.

[21] J.S.Albus et al, "Theory and Practice of Hierarchical Control," COMPCON, 1981, pp.18-39

[22] A J Barbera (Institute for Computer Science and Technology), An Architecture for a Robot Hierarchical Control System, National Bureau of Standards, Washington DC, 1977.

[23] J.A Standish, Data Structure Techniques, Addison-Wesley, Reading MA, 1980.

[24] J.W.Atwood, "Developments in Programming Languages for Distributed Programming," International Conference of Computers Systems & Signal Processing, Bangalore India, Dec.1984.

[25] R.L.Brown & P J.Denning, "Advanced Operating Systems," IEEE Computer, vol 17, no 10, Oct 1984, pp 173-190

[26] D.Comer, Operating System Design -The Xinu Approach, Prentice-Hall Inc , Englewood Cliffs NJ, 07632, 1984

[27] A.Jones et al, "StarOS, a multiprocessor operating system for the support of task forces," Proceedings of the 7th Symposium on Operating Systems Principles, SIGOPS, 1979, pp.117-127.

[28] J.K.Ousterhaut et al, "Medusa, an experiment in distributed operating system structure", Communications of the ACM, vol.23, no 2, Feb 1980, pp.92-105.

[29] ISBC Applications Handbook, Intel Corporation, Santa Clara CA 95051, 1981.

[30] M.Shneier, "3-D Robot Vision," IEEE 1982 Proceedings of the International Conference on Cybernetics and Society, pp.332-336.

[31] M.Shneier et al, "Visual feedback for robot control," IEEE 1982 Industrial Applications of Machine Vision, pp.232-326.

[32] R.C.Holt, Concurrent Euclid -The Unix System and Tunis, Addison-Wesley, Toronto Canada, 1983

[33] P.Brinch Hansen, "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, vol.1,

no 2, June 1975

[34] N Wirth, "Modula a Language for Modular Multiprogramming," Software-Practice and Experience, vol 7, 1977, pp.3-35.

[35] J.Engelberger, Robotics in Practice, AMA-American Management Association, 1980

[36] P Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," Communications of the ACM, vol.21, no.11, Nov.1978, pp934-941.

[37] J-L.Baer, Computer Systems Architecture, Computer Science Press, Rockville MD, 1980.

[38] S.K.Shrivastava & F Panzieri, "The design of a reliable remote procedure call mechanism", IEEE Transactions of Computers, vol.C31, no 7, July 1982, pp 692-697.

[39] Motorola Microprocessors Data Manual, Austin TX, 1981.

Appendix 1. An example of cross-coupled processing

The following is a specification of a task-decomposition hierarchy or H module using the method derived by NBS[18]. It can be compared with examples of H modules provided by NBS[18,19,21] The command processes which were defined for the cross-coupled processing hierarchy example of arc welding are here redefined for the concurrent hierarchical control version Command processes are here restructured as table entries in an H module

It was not possible to similarly derive the sensory processing and world model hierarchies (G and M modules respectively) since little information is provided as to their exact nature.

At each level, all commands are executed using the commands of the next lower level or using procedures for simple calculations.

Command	State	Sensory feedback	Report from lower level	Next state	Command	Report to next level
MODULE H1						
1. MOVE TO P (speed=0..9)	1	–	not arrived	2	GET(theta1-5) TO P	move notdone
	2	–	not arrived	2	MOTORS ON(speed)	move notdone
	2		arrived	0	MOTORS OFF	move done
	–	–	timeout	0	MOTORS OFF	move fail
2. TURN ON ARC	1	temp < threshold	arc off	1	INCR VOLTAGE	arc off
	1	temp >= threshold	arc on	0	KEEP VOLTAGE	arc on
	1	–	–	0	CUT VOLTAGE	arc fail
3. TURN OFF ARC	1	temp >= threshold	arc on	1	CUT VOLTAGE	arc on
	1	temp < threshold	arc off	0	–	arc off
	1	–	–	0	CUT VOLTAGE	arc fail
4. TURN ON INERT GAS	1	gas not flowing	bottle full	1	OPEN VALVE	gas off
	1	–	bottle empty	0	CLOSE VALVE	bottle empty
	1	gas flowing	bottle full	0	–	gas on
	1	gas not flowing	timeout	0	CLOSE VALVE	gas fail
5. TURN OFF INERT GAS	1	gas flowing	–	1	CLOSE VALVE	gas on
	1	gas not flowing	–	0	–	gas off
	1	gas flowing	timeout	0	–	gas fail
6. ADVANCE ELECTRODE SPEED	1	speed = x	–	2	SPEEDUP(x+1)	speedup notdone
	2	speed < (x+1)	–	2	SPEEDUP(x+1)	speedup notdone
	2	speed = (x+1)	–	0	–	speedup done
	2	–	timeout	0	–	speedup fail
7. RETARD ELECTRODE SPEED	1	speed = x	–	2	SLOWDOWN(x-1)	slowdown notdone
	2	speed > (x-1)	–	2	SLOWDOWN(x-1)	slowdown notdone
	2	speed = (x-1)	–	0	–	slowdown done

	2	-	timeout	0	-	slowdown fail
8. MOVE 5mm DIR(d) (SPEED=0..9)	1	-	-	2	GET(theta1-5) TO DIR(d) AT 5mm	move notdone
	2	-	not arrived	2	MOTORS ON(speed)	move notdone
	2	-	arrived	0	MOTORS OFF	move done
	2	-	timeout	0	MOTORS OFF	move fail
9. STOP(position)	1	theta1-5	motors on	1	MOTORS OFF	not stopped
	1	-	motors off	0	MEMORIZE(position)	stopped
10. APPROACH(t)	1	-	not aligned	1	GET(theta1-5) TO P	notat(t)
	1	-	aligned	0	-	at(t)

Command	State	Sensory feedback	Report from lower level	Next state	Command	Report to next level
MODULE H2						
1. TURN ON WELD	1	-	gas off	1	TURN ON GAS	weld off
	1	-	gas on	2	TURN ON ARC	weld off
	2	-	arc off	2	TURN ON ARC	weld off
	2	-	arc on	0	-	weld on
	-	-	-	0	-	weld fail
2. TURN OFF WELD	1	-	arc on	1	TURN OFF GAS	weld on
	1	-	arc off	2	TURN OFF GAS	weld on
	2	-	gas on	2	TURN OFF GAS	weld on
	2	-	gas off	0	-	weld off
	-	-	-	0	-	weld fail
3. GOTO HOME	1	-	move notdone	1	MOVE TO(home)	notat home
	1	-	move done	0	-	at home
4. UNSTICK TIP (position)	1	tip stuck	-	1	STOP(position)	tip stuck
	1	tip stuck	stopped	2	TURN OFF ARC	tip stuck
	2	tip stuck	arc on	2	TURN OFF ARC	tip stuck
	2	tip stuck	arc off	3	TURN OFF GAS	tip stuck
	3	tip stuck	gas on	3	TURN OFF GAS	tip stuck
	3	tip stuck	gas off	4	MOVE IN DIR(-z) AT 1	tip stuck
	4	tip stuck	move notdone	4	MOVE IN DIR(-z)	tip stuck
	4	tip stuck	move done	5	WAIT	tip stuck
	4	tip free	-	6	MOVE TO(position)	tip stuck
	5	tip stuck	wait done	4	MOVE IN DIR(-z) AT 1	tip stuck
	6	tip free	move notdone	6	MOVE TO(position)	tip stuck
	6	tip free	move done	0	-	tip free

5. STOP	1	-	not stopped	1	STOP(position)	not stopped
	1	-	stopped	2	TURN OFF ARC	not stopped
	2	-	arc on	2	TURN OFF ARC	not stopped
	2	-	arc off	3	TURN OFF GAS	not stopped
	3	-	gas flowing	3	TURN OFF GAS	not stopped
	3	-	gas not flowing	0	-	stopped
6. FIND START OF GAP	1	-	move notdone	1	MOVE TO(start)	notat gapstart
	1	not found(target)	move done	2	d=0 on XY, max=2, i=2 MOVE IN DIR(d) AT 1	notat gapstart
	1	found(target)	-	3	APPROACH(target)	notat gapstart
	2	not found(target)	move notdone	2	MOVE IN DIR(d) AT 1	notat gapstart
	2	not found(target)	move done, i<max	2	i=i+2 MOVE IN DIR(d) AT 1	notat gapstart
	2	not found(target)	move done i>=max	2	d=d+90, max=max+1, i=2 MOVE IN DIR(D) AT 1	notat gapstart
	2	found(target)	notat(target)	3	APPROACH(target)	notat gapstart
	3	-	notat(target)	3	APPROACH(target)	notat gapstart
	3	-	at(target)	0	-	at gapstart
7 MOVE 5mm DIR(d)	1	-	move notdone	1	MOVE 5mm DIR(d) (speed=0..9)	move notdone
(speed=0..9)	1	-	move done	0	-	move done
8 ADVANCE ELECTRODE SPEED	1	-	advance notdone	1	ADVANCE ELECTRODE SPEED	advance notdone
	1	-	advance done	0	-	advance done
9. RETARD ELECTRODE SPEED	1	-	retard notdone	-1	RETARD ELECTRODE SPEED	retard notdone
	1	-	retard done	0	-	retard done

Command	State	Sensory feedback	Report from lower level	Next state	Command	Report to next level
MODULE H3						
1. INSPECT GAP	1	-	notat gap-start	1	FIND START OF GAP	inspect notdone
	1	pos(next gapcentre)	at gapstart	2	GET DIR(d) TO pos MOVE IN DIR(d) AT 5	inspect notdone
	2	-	move notdone	2	MOVE IN DIR(d) AT 5	inspect notdone
	2	gapwidth notOK	-	3	STOP(pos)	gap notOK

		gapwidth OK,
		pos(next gapcentre)
		notat(gapend)
			move done	2	GET DIR(d) TO pos	inspect notdone

	2	gapwidth OK at(gapend)	move done	4	GOTO HOME	gap OK
	3	-	not stopped	3	STOP(pos)	inspect notdone
	3	-	stopped	4	GOTO HOME	inspect notdone
	4	-	notat home	4	GOTO HOME	inspect notdone
	4	-	at home	0	-	inspect done
2. WELD GAP	1	-	notat(gap-start)	1	FIND START OF GAP	weld notdone
	1	-	at(gapstart)	2	TURN ON WELD	weld notdone
	2	-	weld off	2	TURN ON WELD	weld notdone
	2	-	weld on	3	ADVANCE ELECTRODE SPEED s=5	weld notdone
	3	pos(gapcentre)	-	4	GET DIR(d) TO pos MOVE IN DIR(d) AT s	weld notdone
	4	-	move notdone	4	MOVE IN DIR(d) AT s	weld notdone
	4	tip stuck	-	5	UNSTICK TIP	weld notdone
	4	temp=threshold, pos(nextcentre)	move done	4	GET DIR(d) TO pos MOVE IN DIR(d) AT s	weld notdone
	4	temp>threshold pos(nextcentre)	move done	4	s=s+1 ADVANCE ELECTRODE SPEED	

					GET DIR(d) TO pos MOVE IN DIR(d) AT s	weld notdone
	4	temp<threshold pos(nextcentre)	move done	4	s=s-1 RETARD ELECTRODE SPEED GET DIR(d) TO pos MOVE IN DIR(d) AT s	weld notdone
	4	at(gapend)	-	6	TURN OFF WELD	weld done
	5	tip stuck		5	UNSTICK TIP	weld notdone
	5	tip stuck	timeout	0	-	weld fail
	5	tip free	-	1	-	weld notdone
	6	-	weld on	6	TURN OFF WELD	weld notdone
	6	-	weld off	7	GOTO HOME	weld done
	7	-	notat home	7	GOTO HOME	weld done
	7	-	at home	0	-	weld done
3. STOP	1	-	notstopped	1	STOP(pos)	not stopped
	1	-	stopped	0	-	stopped

Appendix 2. An Example of Concurrent Hierarchical Control

This appendix provides a detailed description of the functions of the tasks which are shown in the example application of figure A2.1.

Level 1 Control Loops

A1.s Temp of arc:
* read value
* filter out spikes in temperature
* read threshold from C module
* compare temperature to threshold
* sensor failure

A2.c Adjust voltage level:
* adjust and maintain voltage at required level

A2.s Voltage of arc:
* read value
* filter spikes
* sensor failure

A3.c Turn inert gas flow on/off
* turn gas flow on or off

A3.s Gas bottle full/empty:
* read pressure
* sensor failure

Gas flow on/off:
* read pressure in pipe
* sensor fail

A4.c Move to position(p) at speed(s)
* read present position theta1-5
* derive T5
* regulate speed of servomotors
* calculate next position to be reached within 100 milliseconds
* send message to task CO.1

Using the matrix T5, this task calculates the theta angles required in order to move the end-effector to position=p.

A4.s Position of T5.
* read position of theta1-5
* calculate coordinate transform to get T5
* servo fail

A5.c Advance/retard electrode speed.
* read speed of electrode being fed into arc
* advance or retard speed by 1cm/s

This task advances or retards the speed at which the electrode is being fed into the arc. In the event that the trajectory speed is increased, then the electrode being fed into the arc would also need to be increased

A5.s Electrode speed
* read speed
* sensor fail

A6.c Stop/memorize position:
* store value of T5 for future reference
* store value of theta1-5 for future reference

This task stores the present position of the end-effector and the theta angle values for each of the five joints.

A7.s Linear sensor:
* determine linear binary image
* determine distance to surface or gap
* sensor fail

The linear sensor is conceived to be a linear sequence of light sensors which is used to determine a marker. The marker indicates the beginning of the gap to be welded and is used by the robot to line itself for welding or inspection of the gap. The linear sensor is also used to track along the gap during welding or inspection

Level 2 Control Loops

B1.c Turn on/off weld
* send message to turn on inert gas
* send message to turn on voltage to start-up value

B2.c Stop/assassin
* send message to stop servomotors 1-5
* send message to turn off voltage
* send message to turn off inert gas

This task effects a shut-down of the system. Its use is intended for exception conditions.

B3.c Goto home
* place tip in home position

B4.c Move 5mm in direction(d) at speed(s)

* read present position theta1-5

* derive T5

* regulate speed of servomotors

* get next T5 from present T5 by translating present T5 by 5mm along a vector defined by d at T-origin

This task calculates the present position of the tip (matrix=T5) of the end-effector using coordinate transform matrices, or a pre-processor may provide the value of T5. It is conceivable that a method other than coordinate transform matrices could be used in order to determine the present position of the end-effector since accuracy is not as crucial in a sensory-feedback control system. The task then calculates the theta angles required in order to move the end-effector in direction=d at intervals of 5mm.

~~B5.c~~ Approach target
* centre tip over target at distance of 5mm

The robot uses the feedback from its linear visual sensor in order to centre itself over a marker indicating the beginning of the gap to be welded

B5.s Target sighted/not sighted
* read linear binary image
* determine if present and previous binary images constitute the pattern of the target
* determine centre of target with respect to centre of linear image

This task indicates whether the gap-start marker has been sighted by the linear sensor

<u>Level 3 Control Loops</u>

C1.c Unstick tip:
* send message to stop motors
* send message to memorize present position
* send message to move tip 90 degrees away from gap

In the event that the tip touches the surface to be welded and becomes frozen to it, this task will attempt to free the tip from the surface.

C1.s Tip stuck/free:
* read temp of arc and change in position over time in order to determine if the tip is stuck or free

C2.c Find start of gap

```
* position tip at probable start of gap
* read linear sensor
* if linear sensor indicates target has been found
      then send message to align on target
      else do spiral search
* if spiral search ends in failure
      then report search fail
```

Before beginning the welding sequence, the robot must find the beginning of the gap to be welded This is accomplished by providing to the robot an approximate position where the beginning of the gap is expected to be. Once the end-effector moves to that position, then the linear sensor is used in a spiral search path in order to find the beginning-gap marker. Once found, the tip is centred on the marker.

```
C2.s    At start or end of gap.
      * using command contextual information, determine
        whether tip is at start/end of gap or inbetween
```

<u>Level 4 Control Loops</u>

```
D1.c    Inspect gap
      * send message to fine start of gap
      * move in predicted direction of gap maintaining
        distance from gap and alignment on gap centre
      * compare gap width with acceptable threshold
      * if gap width too wide
           then inspection fail
        else if gap width too narrow
           then inspection fail
        else if gap width OK
           then inspection OK
```

Before beginning a weld, the gap is first inspected to make certain that the gap width is not too wide or too narrow.

```
D1.s    Target sighted/not sighted
      * read linear binary image
      * determine if present and previous binary images
        constitute the pattern of the target
      * determine centre of target with respect to centre
        of linear sensor
```

This task indicates whether the gap-beginning marker has been sighted by the linear sensor

```
        Predicted next centre of gap
      * read linear sensor
      * determine present centre of gap and present error
      * using present and previous centres of gap, predict
        next centre of gap (5mm in Y direction)
```

This task determines the next centre of the gap This is required for tracking along a curving gap The sensor is assumed to be located about 5mm in front of the arc during welding.

```
        Gap width OK/not OK.
      * read gap width threshold
      * read linear sensor and position
      * determine distance to gap
      * determine whether gap is presently too wide or not
        wide enough

D2.c    Weld gap:
      * send message to find start of gap
      * move in predicted direction of gap maintaining
        distance from gap and alignment on gap centre
      * turn on weld
      * using temperature feedback, maintain temperature
        within given threshold
      * if temp > threshold then send messages to increase
        speed of tip and advance electrode speed
      * if temp < threshold then send messages to decrease
        speed of tip and retard electrode speed
      * if target(end of gap) sighted then stop welding
```

The gap is welded and the temperature is monitored in order to maintain an optimum.

```
D2.s    Temp <=> threshold
      * using command contextual information, determine
        whether temp is above, below or equal to threshold
```

__Level 5 Control Loops__

```
E1.c    Stop/assassin:
      * send message to stop
      * memorize position
```

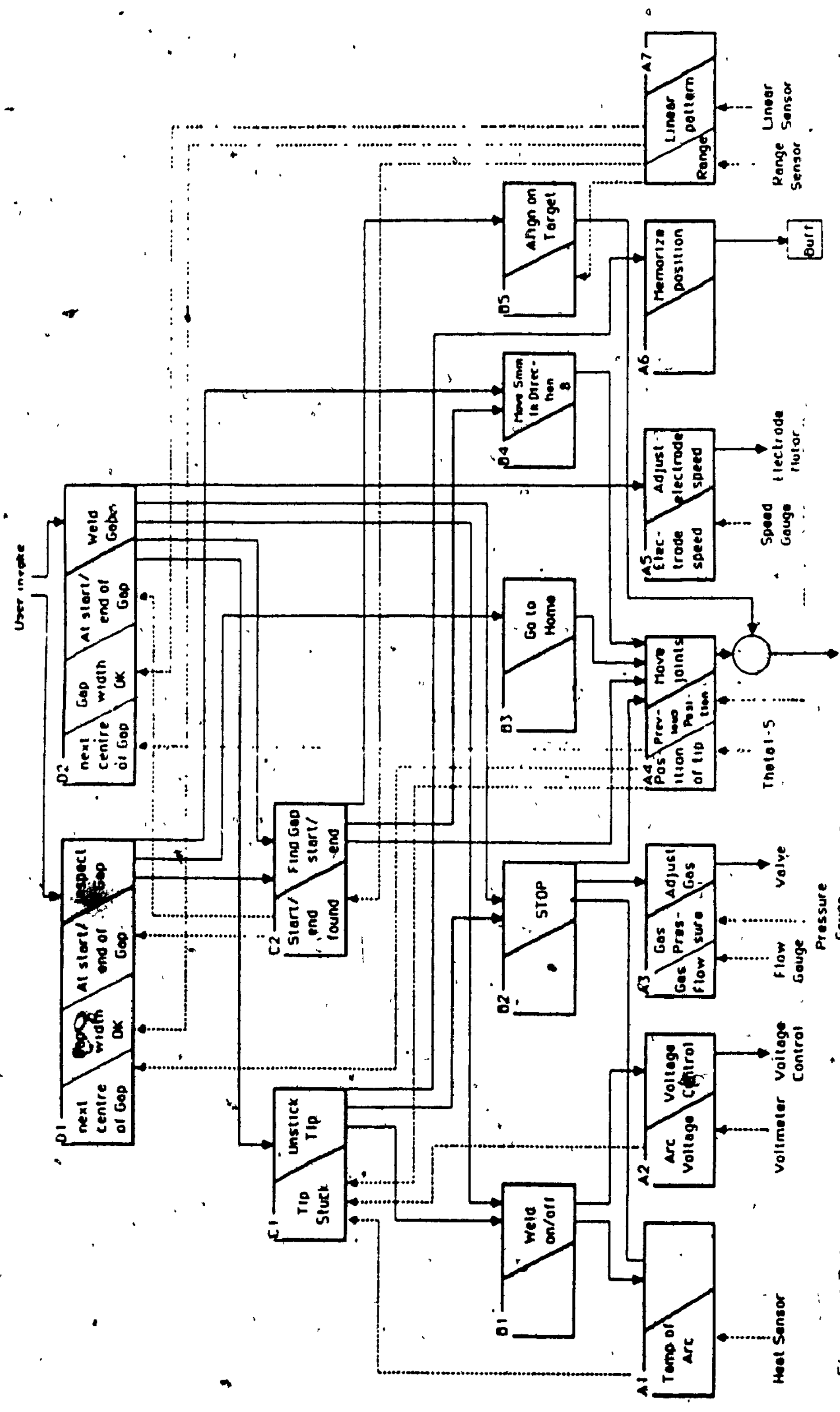
User invoke
D1
next centre of Gap
Gap width OK
At start/ end of Gap
Inspect Gap
D2
next centre of Gap
Gap width OK
At start/ end of Gap
Weld Gap
C1
Tip Stuck
Unstick Tip
C2
Start/ end found
Find Gap start/ end
B1
Weld on/off
B2
STOP
B3
Go to Home
B4
Move 5mm in Direction θ
B5
Align on Target
A1
Temp of Arc
Heat Sensor
A2
Arc Voltage
Voltage Control
Voltmeter
Voltage Control
A3
Gas flow
Gas Pressure
Adjust Gas
Flow Gauge
Pressure Gauge
Valve
A4
Position of tip
Previous Position
Move joints
Theta1-5
Servo 1-5
A5
Electrode speed
Adjust electrode speed
Speed Gauge
Electrode Motor
A6
Memorize position
Buff
A7
Range
Linear pattern
Range Sensor
Linear Sensor

Figure A2.1

1. Appendix 3. CHC-OS: An OS for Concurrent Hierarchical Control

1.1. System Routines

The following is a list of system calls and a brief description of their functions Many of the routine names and their function have been either borrowed or inspired by Comer's Xinu operating system [26]

Kernel Routines

```
ctxsw()
inbyte()
outbyte()
incomm()
outcomm()
```

The routines inbyte() and outbyte() allow the calling task to read and write to a device by means of IO ports defined in the CPU hardware. The routines incomm() and outcomm() are similar to inbyte() and outbyte() except that they are used exclusively for inter-processor communication through a serial link such as Ethernet. The routine ctxsw() performs a CPU context switch which allows the CPU to be shared by a number of tasks concurrently. It is used exclusively by the task manager

Memmgr Routines

```
enqueue()
dequeue()
insertd()
qempty()
qnotemtpy()
qhead()
qtail()
semwait()
semsignal()
alloc()
load()
```

Memmgr provides list management routines for the use of task management as well as general use by a user-defined task. Enqueue() is used to place the pointer to an item onto a pointer list. Each entry on the pointer list has fields for forward and backward pointers Dequeue() is used to remove a pointer from anywhere in a pointer list. It may therefore be used in implement either FIFO (first-in/first-out), LIFO (last-in/first-out) or random list management. Insertd() is used to insert a pointer into its proper location on a delta list which is a list of items waiting for specific time periods. Qempty() and qnotempty() provide boolean values indicating the status of a list, and qhead() and qtail() return list positions indicating which pointers are at the head and the tail of the list.

Semwait() and semsignal are routines used for the management of semaphores.

Alloc() is used when the system manager makes a request for a memory block from 'memtab' when a new task needs to be installed in the system. Load() actually loads an object file from Unix into a satellite processor at the address returned by alloc()

Tskmgr Routines

newtsk()
ready()
suspend()
wait()
sleep()
changowner()
setpri()
resetpri()
resched()
gettid()

The task manager takes care of everything to do with task management through a task table called 'tsktab'. The newtsk() system call is used by the system manager to create a task on the task table during installation of a new task. The system calls ready(), suspend(), wait() and sleep() alter the execution status of a task and sleep() places the selected task on a delta list.

Changeowner() alters the owner field of the tasks tsktab entry to indicate which higher-level task has invoked it. The owner will not be changed if the message priority from a new higher-level task to the task in question is lower than the message priority with the present higher-level task.

The routine setpri() sets the priority of a task associated with a specific command message to the task execution priority associated with the message priority, in other words, the higher the command message priority invoking a task, the higher the execution priority of the task.

Resched() is used by the task manager to reschedule the CPU according to some scheduling policy; resched() is the only routine which has access to the kernel routine ctxsw() which physically effects the context switch. Although resched() is a C procedure like any other, it will in fact never return after it's been called

Gettid() returns the task ID of the presently executing task providing a means by which system routines can identify the task by which they've been called.

<u>Devmgr Routines</u>

```
getc()
putc()
```

The device manager handles hardware and software interrupt routines for sensors and actuators, and readies user-defined device driver tasks for execution Getc() reads a character from a sensor or from the communications port and is called by a hardware interrupt routine. Putc() writes a character to an actutor or to the communications port and is called by a software interrupt routine.

<u>Commgr Routines</u>

```
readcomm()
writescomm()
writeccomm()
```

The communications manager is used to support all inter-processor communication. The routine readcomm() is used by the system manager when it responds to a hardware interrupt indicating that a message is coming in from another machine. It reads the message, determines the message type; whether command or sensory, and then transfers the message to the message manager in order to effect the transfer The procedures writescomm() and writeccomm() write sensory and command messages respectively to another processor which has been determined by the message manager.

Msgmgr Routines

```
newmsg()
setmsgpri()
tgetc()
tputc()
topen()
tclose()
eotsk()
transfermsg()
msgstat()
```

Tgetc() will get a sensory message from a source whether it's in the local CPU or a foreign CPU If the message transfer is local, then the message transfer is effected or the calling task may be suspended if there is as yet no message to be transferred The same thing happens with a foreign processor, except that the task will have to wait on a message addressed specifically for it arriving through the communications port. Tputc() will transfer a sensory message to a destination without blocking to a local

CPU destination or to the communications manager if the destination is in a foreign CPU

Topen() invokes a task by sending a command message to the destination. If the destination task priority is already higher than the message priority trying to gain control of it, then the calling task is placed on a waiting queue associated with the destination task. If the new message priority is higher, then the previous invoking task must be pre-empted and a message to that effect is returned to it, then the owner field of the invoked task is altered to reflect the change Tclose() releases a task from an invocation and either hands over ownership of the task to the next highest priority task on the waiting list or suspends the task

Teotsk() returns a normal task completion report to the invoking task The user task can check the status of any message implying the status of any task, through the procedure msgstat() This will inform the calling task whether the destination task is OK, pre-empted, or if it completed normally. How the pre-empted task handles the situation is up to the user-programmer who may either retry the invocation in which case he is placed on a wait queue, or take evasive action

Newmsg() builds a new message entry into the message entry and is used when the system manager is installing a new task in the system. A parameter must indicate whether the message is for sensory information or command invocation. Setmpri() would be used typically when a new message is defined, although it may conceivably be used dynamically as well in order to test and fine-tune system performance The message priority indicates at what priority the receiving task must execute

Transfermsg() is used by the communications manager to transfer incoming inter-processor messages to the appropriate destination task.

<u>Usrmgr</u>

There are no routines defined within the user manager, all higher level management of user tasks can be done through the task manager

<u>Sysmgr Routines</u>

```
install()
setpri()
invoke()
```

The install() procedure is invoked only from the Unix development system and is used to install a new task in a specified machine. Install() updates the message tables in all machines and in the target machine it requests a memory allocation, loads and suspends the task, and updates the task table.

The setpri() routine allows the user to set message priorities dynamically from Unix by permitting a real-time link between Unix and the message managers in the various processors.

The invoke() procedure provides a real-time link like setpri() which allows the user to manipulate the system by sending a command message from Unix to a task in the same way as a task to a task does Invoke() allows him to invoke or suspend any task in CHC-OS from Unix thereby allowing him to execute only subsets of tasks The entire system would be started up by an invoke() being sent to the highest-level task.

1.2. System Tables

Following is presented the structure of various system tables.

Memory Manager Table: Memtab

* base of segment
* size of segment

Task Manager Table: Tsktab

```
* name
* machine
* status(current
        ready
        suspended
        waiting
        asleep)
* owner(message)
* priority(owner)
* sleep(msecs of sleep remaining)
* environment * pc
              * baseaddr
              * limitaddr
              * stkptr
              * stklimit
```

Device Manager Table: Devtab

```
* name
* status(OK
        suspended)
* buffer
```

Message Manager Table: Msgtab

```
* priority
* status(OK
        suspended
        pre-empted
        normal completion
        abnormal completion)
* type(sensory
      command)
* source(task,sensor)
* destination(task,actuator)
```

Usr Manager Tables: Usrtab

```
* input ports[8]
* output ports[8]
```