NISTIR 4936

A Real-Time Control System Methodology for Developing Intelligent Control Systems

Richard Quintero
Group Leader
Unmanned Systems Group

A.J. Barbera
Advanced Technology and Research Corporation
Laurel Technology Center
14900 Sweitzer Lane
Laurel, MD 20707

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Robot Systems Division
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

NIST

NISTIR 4936

A Real-Time Control System Methodology for Developing Intelligent Control Systems

Richard Quintero
Group Leader
Unmanned Systems Group

A.J. Barbera
Advanced Technology and Research Corporation
Laurel Technology Center
14900 Sweitzer Lane
Laurel, MD 20707

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Robot Systems Division
Bldg. 220 Rm. B124
Gaithersburg, MD 20899

October 1992

U.S. DEPARTMENT OF COMMERCE
Barbara Hackman Franklin, Secretary

TECHNOLOGY ADMINISTRATION
Robert M. White, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

A Real-Time Control System Methodology for Developing Intelligent Control Systems

Richard Quintero
Robot Systems Division
National Institute of Standards and Technology

A. J. Barbera
Advanced Technology and Research Corporation

Abstract

This paper presents an approach to a ***Real-Time Control System (RCS)*** systems engineering methodology which complies with the RCS Reference Model Architecture developed by Robot Systems Division researchers at the National Institute of Standards and Technology (NIST). We also offer software implementation examples within the context of this RCS Methodology approach. NIST has been conducting research in the area of hierarchical real-time control systems for automation and robotics, for more than a decade. NIST researchers, working in the Automated Manufacturing Research Facility (AMRF) and on a number of other agency projects, have defined a theoretical reference model architecture as a first step in establishing a framework for standards development. This paper represents a second step toward that goal.

Key Words:

automation, control systems, hierarchical control, RCS, real-time systems, robotics, software analysis, software design, software methods, task oriented method

Table of Contents

List of Figures and Tables

A Real-Time Control System Methodology for Developing Intelligent Control Systems

Richard Quintero
Robot Systems Division
National Institute of Standards and Technology

A. J. Barbera
Advanced Technology and Research Corporation

SECTION 1. INTRODUCTION

This paper presents a systems engineering methodology for developing complex, integrated, intelligent machine control systems. We call it a ***Real-Time Control System (RCS) Methodology.*** The particular systems engineering approach presented here was originated by Barbera and others while he was with the Robot Systems Division at the National Institute of Standards and Technology (NIST) in the late 1970's and early 1980's. The methodology complies with the RCS Reference Model Architecture developed by Albus, Barbera and others at NIST [Al 89a, Al 89b]. We also offer "C" code examples of a software implementation developed using this RCS Methodology. These examples are in Appendices A, B, and C. Researchers at NIST are exploring several RCS implementation approaches in addition to this one. Each of these approaches generally reflects the application used to demonstrate the researchers work and they are typically optimized for a particular subclass of RCS applications.

Real-time intelligent machine control systems applications cover a very broad spectrum. For example: 1) Controls engineers often deal with robotic or machine tool applications with requirements for high-speed servo control of machines with multiple joints and/or several axis of motion. These problems become even more interesting and demanding when real-time closed-loop compensation is introduced to achieve very high accuracies or compliant motion. 2) Systems engineers are interested in coordinating the control of several machines or in the case of large vehicles (e.g., ships, submarines, aircraft) several major subsystems in order to accomplish a set of desired system goals. Such systems usually require some degree of human interaction. Closed-loop control is introduced in these applications in order to deal with uncertain and noisy input data and in order to be able to function in unstructured environments. 3) At a higher level of abstraction, systems engineers become concerned with the enterprise model in manufacturing, traffic resource management in vehicle problems, or battle coordination in military applications. Communications networks, human interface, and knowledge management receive increased attention in these applications. Sensory-feedback is used in these applications for the same reasons discussed above but at a higher level of abstraction.

In this paper we will be presenting a systems engineering methodology which can be used across this spectrum of applications, however our examples and detailed discussion will emphasize the middle level (type 2 above) and to a lesser extent the high level intelligent control systems problems (type 3 above). There are many alternative implementation methodologies possible

which could comply with the NIST RCS Reference Model. The kinds of applications which are particularly well suited to the Barbera approach are typically rule driven (i.e., they employ strategies, tactics, and process knowledge) and they are characterized by a need to monitor sensory input to detect events in the intelligent system's environment which are used to trigger the system's activities and its reaction to exception conditions. We will discuss, but not emphasize, control systems problems dealing with path planning, trajectory generation and control law algorithms (type 1 above).

1.1. Product Endorsement Disclaimer

References to specific brands, equipment, or trade names in this document are made to facilitate understanding and do not imply endorsement by the National Institute of Standards and Technology.

1.2. Background

The National Institute of Standards and Technology has been conducting research in the area of hierarchical real-time control systems for automation and robotics for more than a decade. This paper offers an initial attempt at formalizing a systems engineering methodology to compliment the results achieved by NIST researchers in the Automated Manufacturing Research Facility (AMRF) [Si 83] and on a number of other agency projects undertaken by the Robot Systems Division (RSD).

Early work by Albus [Al 81] and Barbera [Ba 84], in the AMRF, gave rise to the first definition of a Real-Time Control System (RCS) systems engineering approach focusing primarily on software design. This approach was derived from a control systems engineering perspective rather than a data processing perspective. The Robot Systems Division has refined and evolved these techniques by applying the RCS approach to a number of robotic problems in manufacturing as well as robotic applications intended for unstructured environments including the Army Field Material-Handling Robot (FMR) project [Mc 86], the Defense Advanced Research Projects Agency (DARPA) Multiple Autonomous Undersea Vehicles (MAUV) project [Al 88], the Army Tech-based Enhancement for Autonomous Machines (TEAM) project [Sz 88], the U.S. Bureau of Mines Coal Mining Automation Project [Al 89b], [Hu 92], and [Hu 91] as well as others. Advanced Technology & Research Corporation (ATR) has used the RCS Methodology described here in the implementation of control systems for a number of U.S. Postal Service materials handling systems, employing monorail carriers, distributed staging towers, wire-guided vehicles, conveyor sorting loops, and carousels, as well as other projects such as autonomous vehicles and security systems.

One of the more well known NIST architecture definition efforts was the development of the NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM) [Al 89a]. NASREM was adopted by NASA to provide a software control system architecture guideline for use by development contractors charged with building the Flight Telerobot Servicer (FTS) control system as part of the Freedom Space Station project.

A number of versions of the NIST control system have been implemented (i.e. RCS-1, RCS-2 and RCS-3) over the last ten years, with each version utilizing the most advanced "off-the-shelf" hardware and software technologies available. RCS versions have been implemented using the FORTH, C, and ADA languages and running on Motorola 68010/20/30 processors as well as on Intel 286/386 machines and on Multibus and VME backplanes. Applications have been built using real-time operating systems such as: GRAMPS, pSOS, and VxWorks. RCS applications have also been hosted within the DOS operating system on personal computers (PCs).

The methodology presented in this paper is based on our most recent work in applying RCS techniques to the automation of submarine maneuvering control, under DARPA sponsorship, and in demonstrating the automation of a continuous coal mining machine, under the sponsorship of the U. S. Bureau of Mines. This work is being carried out by a team of researchers from NIST and ATR.

Even though we have achieved some measure of success in applying these techniques in a research environment, acceptance of the RCS approach by industry has been slow to materialize. Two things that can help to accelerate RCS technology transfer are the development of a formal RCS systems engineering methodology and a comprehensive set of Computer-Aided Software Engineering (CASE) tools based on the methodology. This paper addresses the methodology issue.

1.3. Some Preliminary Definitions

This paper presents one possible software implementation approach which complies with the RCS Reference Model Architecture (or referred to as simply the RCS Architecture). The implementation approach, in turn, allows us to begin to define a complementary RCS Methodology by offering implementation specific details necessary for understanding the engineering trade-offs that must be considered as well as a basis for selecting certain software engineering methods over others.

We will use the terms RCS, RCS Methodology and RCS Architecture interchangeably throughout the remainder of this document for convenience. We will also generally make use of the words "method" and "methodology" as synonyms even though a more rigorous definition of the word "methodology" is a collection of methods.

The Random House College Dictionary, [Ra 82], defines ***architecture*** as "the character or style of building; the structure of anything". The RCS Architecture is a style of building real-time intelligent control systems. These systems generally include software, hardware, machines, people, communications, information repositories, information/knowledge models and real-time execution models. The RCS Architecture defines a highly structured, modular organization of these control system components which can serve as a standard reference model for an open-system architecture.

In describing RCS systems integration rules, we will use the term ***function*** to mean the conceptual role, or activity prescribed for an abstract process. A function may or may not have a

one-to-one mapping to implementation code. Functions may be decomposed into subfunctions, and some functions will be distributed across the RCS Architecture.

When we use the term ***module,*** we will be describing a software component capable of performing one or more functions. A module has clearly defined input and output specifications which are in compliance with the integration rules we will be enumerating. A module will also have a one-to-one mapping to implementation code. Modules can be thought of as software subprograms which can be assigned to and executed on a host Central Processing Unit (CPU). Modules contain mechanisms for communicating with other modules, and they encapsulate component specific knowledge and processes. Modules, therefore, are the building blocks with which an integrated system can be built.

An ***atomic algorithm*** is defined here as a complete and self-contained algorithm which does not easily decompose any further. Such an algorithm, when implemented in software, accepts input parameters and generates output parameters in a single pass. A coded algorithm may execute cyclically, and it may accumulate or otherwise retain the results of previous execution passes in order to refine its outputs. A coded atomic algorithm does not contain any external branching, its input/output (I/O) functions are always non-blocking, and it does not contain any indefinite internal computing loops. An atomic algorithm is designed to be computed as sequential code on a single CPU with a deterministic execution time. An atomic algorithm may be executed concurrently with other atomic algorithms in a multiprocessor system. A module may encapsulate any number of atomic algorithms.

Fiala [Fi 90a] defines an ***atomic unit*** as a complete and self-contained concurrent function that communicates with other atomic units only by way of global data system communication primitives (read and write). An atomic unit may also directly access sensors and actuators. A module may contain any number of atomic units, but, by definition, an atomic unit cannot be decomposed further into concurrent architectural elements. Fiala's definition of an atomic unit would meet our definition of a controller module (defined below) which encapsulates the software for an atomic algorithm.

We call your attention to the fact that we are intentionally using the term function to describe processes which are called modules in earlier NIST RCS publications. Here we are reserving the term module to describe a single RCS building block, a ***controller module***. This is one of the characteristics that distinguish the Barbera approach from other RCS Reference Model compliant methodologies. A controller module does not contain any submodules, but it may encapsulate any number of functions, subfunctions, processes, and atomic algorithms. The controller module can be viewed as a systems integration "wrapper" which is implemented as a template. We encapsulate software within this wrapper to ensure compliance with our integration rules. Every module built using the controller module template will inherit a software execution model, a communications mechanism, performance measurement capabilities, and debug mechanisms. All of these properties will be discussed in detail in subsequent sections of this paper.

Appendices A, B, and C offer "C" code examples illustrating how the RCS Methodology presented here is actually implemented in software.

1.4. Preview

In Section 2 we discuss the intelligent machine problem domain, our objectives in defining an RCS Methodology, and we describe the reference model architecture which is used as a foundation for the RCS Methodology presented here. Section 3 defines a set of systems integration and modularity rules we call the RCS tenets. These RCS tenets provide guidelines for a systems engineer who wishes to use the Barbera approach to implement an intelligent control system application in an manner consistent with the RCS Reference Model Architecture discussed in Section 2. Section 4 defines the plan representation models we emphasize as part of this RCS Methodology. Section 5 describes the RCS Controller Module Template and the RCS Main Program, which constitute the basic building blocks for this RCS Methodology. Section 5 also discusses the software services and hardware configuration required of a host computing environment for implementing RCS. Section 6 presents a summary of the rapid prototyping system development steps we use in developing intelligent control systems. Section 7 relates RCS to some popular software engineering methods and presents our concluding remarks.

1.5. Acknowledgements

We would like to offer our thanks to our RCS development team colleagues, whose software development work on the DARPA Submarine Automation Project and the Bureau of Mines Mining Machine Automation Project, laid the foundation for this paper. Special thanks to NIST development team members: Hui-Min Huang, Ron Hira, John Horst, Will Shackleford, Ross Tabachow; and ATR team members: M.L. Fitzgerald, Phil Feldman, Clyde Findley, Nat Frampton, and Mark Routson. In addition we gratefully acknowledge the many technical insights offered by Jim Albus, John Michaloski, and John Fiala.

SECTION 2. THE RCS PROBLEM DOMAIN

RCS specifically addresses intelligent machine control systems problems. We define ***intelligent machines*** to be machines designed to perform useful physical work while employing in situ knowledge (sensory input data), and a priori knowledge, tactics and strategy. Intelligent machines are further defined as utilizing feedback from the physical environment to manifest "intelligent behavior" in real-time via computerized real-time control of the machine's electro-mechanical actuators and sensors. Such systems are termed ***closed-loop control systems*** and are distinguished from ***open-loop control systems*** in that open-loop systems do not have the capacity to alter their behavior in real-time based on sensory feedback from the environment.

The definition given above for intelligent machines is intended to include: ***automation systems*** such as those found in manufacturing, materials processing, mining and construction; ***embedded systems*** such as military cruise missiles, torpedoes and other semi-autonomous devices; and ***robotic systems*** ranging from factory floor robots to space vehicles and planetary exploration robots.

In general, practical intelligent machines almost always require some level of human interaction. Such interaction can range from intensive man-in-the-loop systems to human supervised semi-autonomous systems. In some cases these systems can be fully autonomous after initialization and employment or launch, such as in the case of some cruise missiles, torpedoes and space vehicles.

2.1. RCS Methodology Objectives

In priority order, our objectives in developing RCS are to:

1) Improve human understanding of the design result. As computers and software tools evolve and become more sophisticated, it becomes increasingly important to ensure that we are able to teach people to develop, extend and maintain real-time control systems using standard techniques. This is particularly apparent when undertaking very large development efforts requiring teams of engineers over development life cycles of several to many years. Our designs must not only communicate with the original developers, they must also be understood by those charged with enhancing and maintaining these systems over their full life cycle. In this context, human understanding of the design overrides the importance of developing software that executes efficiently on any given computer platform using any particular set of software languages, operating systems or tools.

2) Manage software complexity. The problem of managing software complexity is directly related to improving human understanding of the design. In fact, managing complexity is a technique for achieving improved understandability. Complexity management generally implies organizing information in a modular form easily understood by people. When managing the complexity of large systems, we need to establish systems integration standards as a useful technique in organizing the contributions of teams of developers. Creating well understood primitive (reusable) modules which can be pieced together using the established integration rules to form macro structures or subsystems is another powerful approach. An

extension of this technique is to generalize repeating patterns of primitive building blocks and use these patterns as templates to create new instances of application-specific system modules. Often complexity management involves a trade-off between imposing some level of overhead structure and achieving a very streamlined (efficient) design for a specific application (also known as a point solution).

3) Provide for robust, verifiable, efficient, coordinated, real-time performance. We would like our designs to result in control systems that are: a) ***robust*** - systems that can function in the presence of noise, systems that can tolerate minor errors in knowledge modeling, and fault tolerant systems that exhibit graceful degradation; b) ***verifiable*** - deterministic systems whose intelligent behavior can be completely analyzed and verified, and whose execution time performance can be reliably predicted by calculation or direct measurement; c) ***efficient*** - designs that provide reasonable techniques for conserving resources; d) ***coordinated*** - designs that specifically address and model concurrent execution and coordinated multi-actuator machine behavior; e) ***real-time performance*** - designs that specifically address the need to compute solutions in time to be effective given the physical demands of the application.

4) Provide for extensibility, portability, and software reuse. a) ***Extensibility*** - We are interested in producing designs that can be easily extended. It is important to allow large systems to be developed using an evolutionary approach, adding functionality to address new requirements not envisioned in the early life cycle phases. b) ***Portability*** - As hardware technology evolves we want to take advantage of more powerful and efficient platforms by porting software to new computers with minimal recoding. c) ***Software Reuse*** - Lastly we would like to collect libraries of software modules and software subsystems which can be reused in similar applications. Some examples of such reusable software are software templates, algorithms which compute common mathematical functions, algorithms for computing different types of trajectories, algorithms for interpreting and updating computer files using standard computer information models (e.g., geometric models, maps, A* search, quadtrees, etc.), and software device drivers which link specific hardware devices to many different software applications.

A Robot Systems Division position paper entitled, "A Reference Model Architecture for ARTICS", [Al 91a], provides an in-depth discussion of the issues raised above.

2.2. RCS Methodology Approach

Developing an RCS Methodology involves: 1) establishing a comprehensive set of integration rules, 2) identifying information models and real-time software execution models which explicitly highlight critical components of the RCS problem domain (intelligent machine control systems), and 3) selecting software engineering implementation techniques which are compatible with the integration rules, the models and the RCS Methodology objectives presented above. Figure 1 provides a pictorial summary of these ideas. In Figure 1, there are two major divisions of the architecture models - software and hardware. The software models include information models and execution models to be described in more detail later in this document. The hardware

architecture includes all of the people, machines, sensors, actuators, and computing hardware as well as a communications network. The critical design components to be explicitly addressed include:

- real-time task behavior and software execution models
- interfaces and communications methods between software modules, hardware resources and human operators
- information models and knowledge base management
- allocation of resources (assignment of software modules to computing hardware)
- rules for decomposing the design spatially and temporally

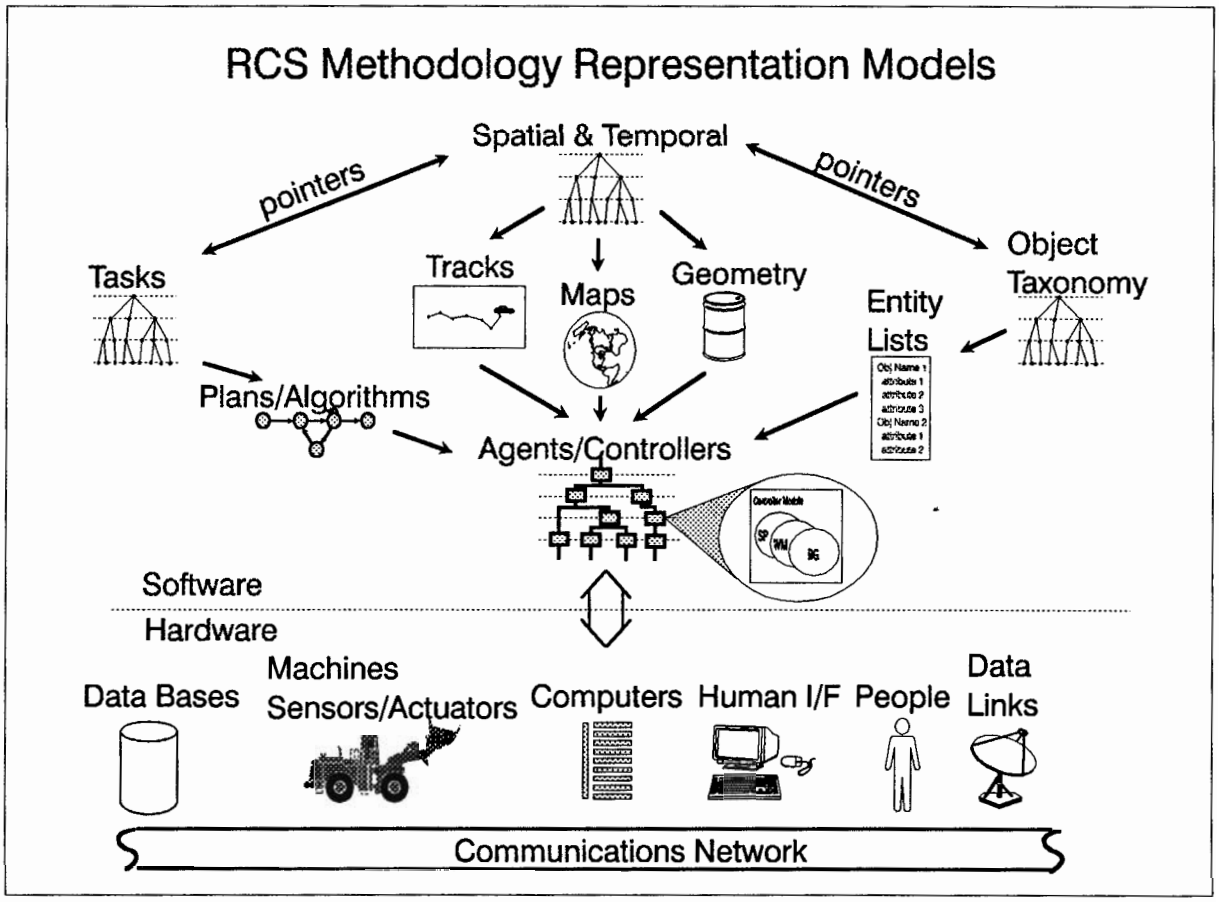


Figure 1.

2.3. The RCS Architecture Reference Model

The RCS Methodology described in this paper complements and is based upon the theoretical RCS Architecture principles developed by Albus, Barbera and other NIST researchers, since the mid 1970s. A comprehensive treatment of the RCS Architecture is contained in the following publications: [Al 91b], [Al 90a], [Al 90b], [Al 89a], [Al 81], [Ba 84]. We will present a brief summary of those concepts here.

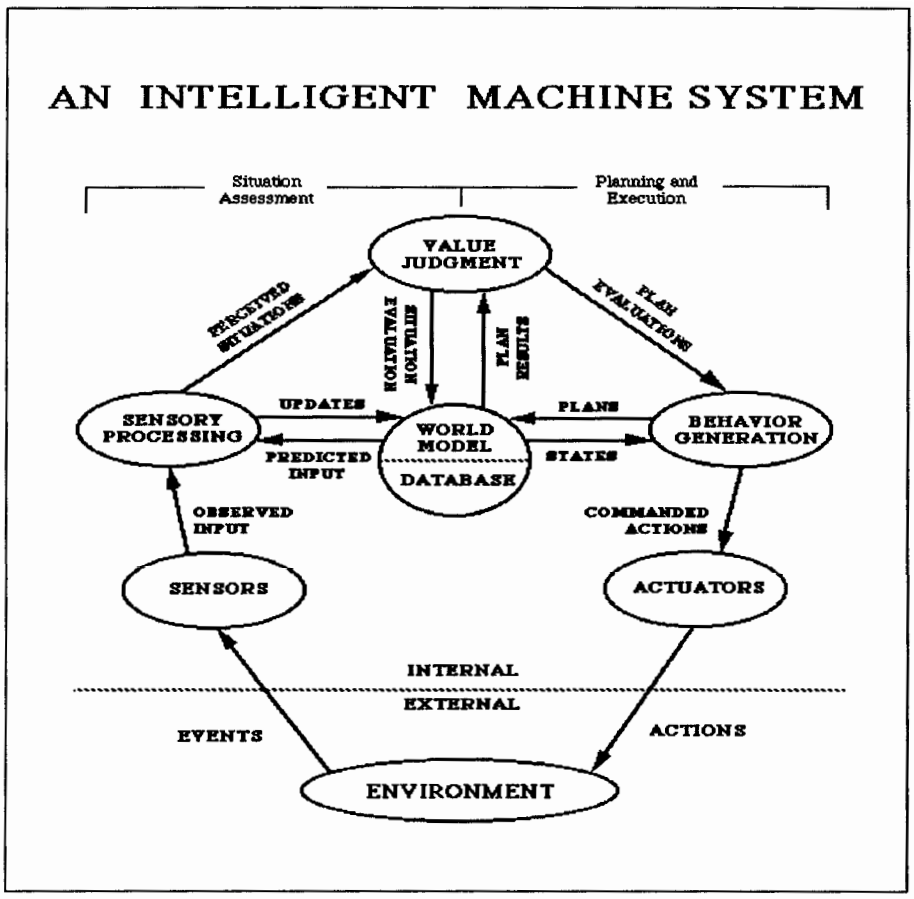


Figure 2.

2.3.1. Closed-Loop Control in an Intelligent Machine

Figure 2 presents a conceptual view of an intelligent machine. Such a machine employs actuators with which it can manipulate objects or materials in the environment. The machine also possesses sensors to monitor the current status of the environment, the effect it is having on the environment as a result of its actions, and the internal state of the machine itself as it performs its functions. An intelligent machine must possess a model of the world it lives in, in order to interpret the information it senses and the effect of its real-time behavior on things in the environment. A closed-loop control system is formed in the machine by inputting sensory data to ***Sensory Processing (SP)***, passing the processed information off to the ***World Modeling (WM)*** function, which maintains the machine's best estimate of the state of its world, and finally closing the loop through ***Behavior Generation (BG)*** which plans and executes actions to be performed through the machine's actuators.

In earlier NIST architecture reference model documents (NASREM, etc.) the Behavior Generation function was called the ***Task Decomposition (TD)*** function. In this document we group the subfunctions comprising (TD) and label them ***Behavior Generation (BG)***. The term task decomposition is used here to refer to the design process used in developing an RCS application.

An intelligent machine must also utilize a value system in order to judge the "goodness" of the results of its actions in the context of the tasks it is expected to perform. The value system, or the ***Value Judgment (VJ)*** function, works in conjunction with goal selection to direct Behavior Generation in selecting alternative plans and actions. The value system and goal selection functions are generally thought of as existing within the basic SP, WM, and BG functions.

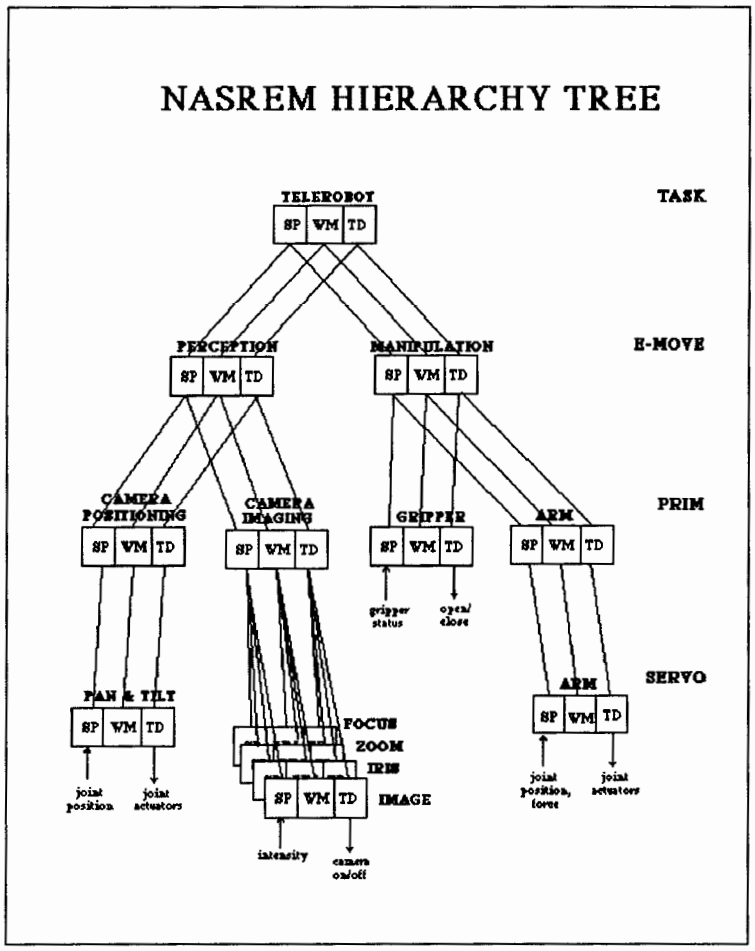


Figure 3.

2.3.2. The RCS Hierarchy

Figure 3 extends the notion of an intelligent machine design containing the basic SP, WM and BG functions by creating a hierarchy. It shows an example NASA FTS robot structure arranged in hierarchical levels. In order to enhance human understanding and to manage software complexity, the basic SP, WM, and TD (or BG) functions are grouped as ***controller nodes*** and distributed in a hierarchically organized, integrated set of nodes. In an RCS implementation a node is a collection of one or more software modules. Each controller node is assigned a set of tasks at an appropriate level of abstraction and each has a limited range of authority and responsibility within the chain-of-command formed by the RCS hierarchy (much like a human military command structure would be organized). Later in this paper we will discuss how to create templates for

controller modules (controller nodes implemented as modules or subprograms) to use as a basic building block for RCS designs.

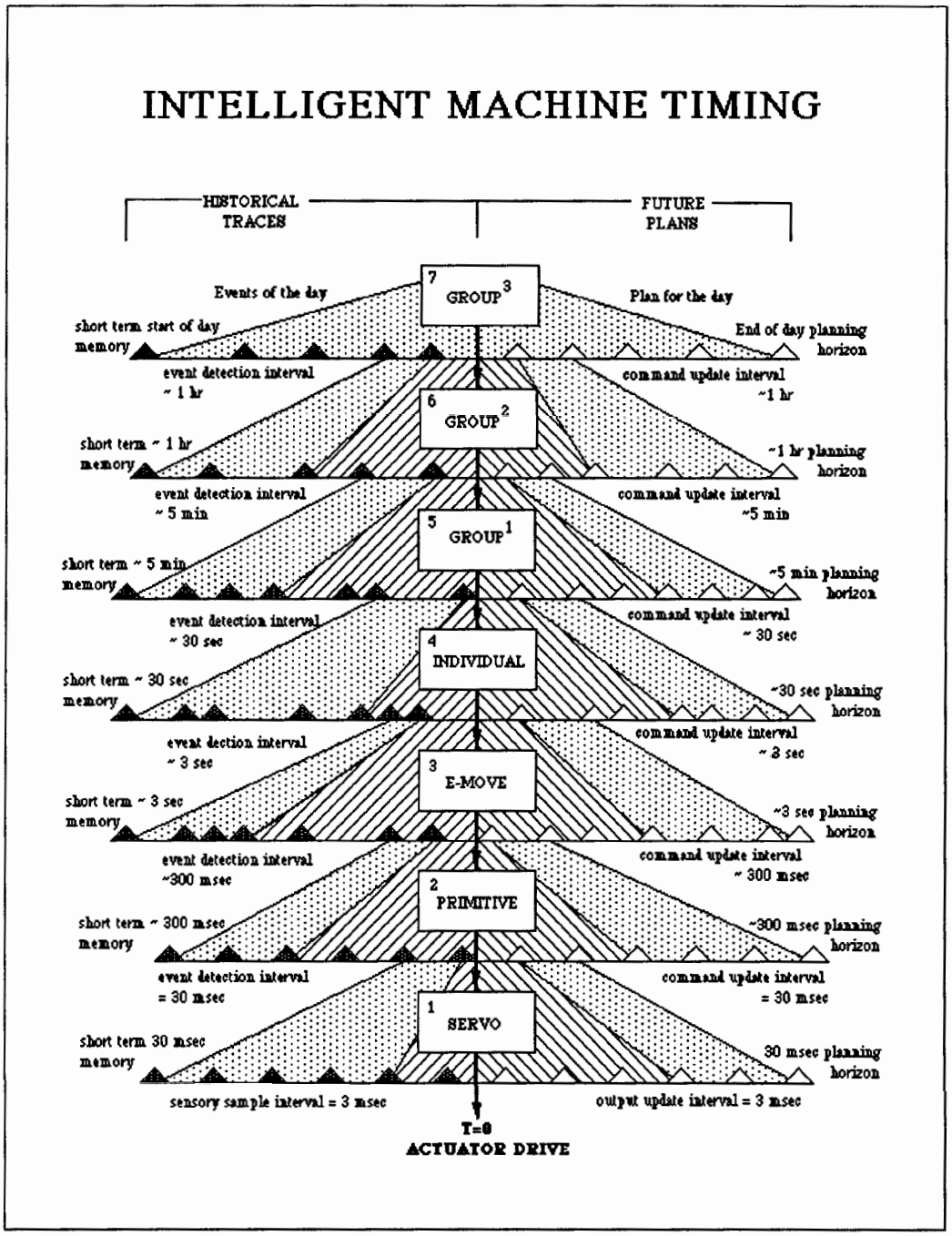


Figure 4.

2.3.3. RCS Levels of Abstraction

Figure 4 depicts the temporal decomposition of an RCS design where in each level deals with planning horizons, memory spans and goal completion rates which are subdivided by roughly an order of magnitude in time between levels. Planning horizons and memory spans increase as we

move to higher levels of the architecture while sub-goal accomplishment rates increase as we traverse the hierarchy towards the lower levels.

In RCS, there is a notion of decomposing the control system design into layers or levels of abstraction (see Figures 3 and 4). The RCS Reference Model Architecture describes the types of tasks carried out at each level of abstraction, starting at the bottom of the hierarchy. These descriptions are useful in conceptualizing the organization of a control system application but should not be interpreted as strict requirement. In a given application some levels may be absent for some branches of a control system hierarchy and others may have additional layering. In addition "black box" subsystems might be substituted for a sub-tree of the hierarchy in some applications. The levels of abstraction are discussed here from the perspective of physical movement tasks (rather than logical operations such as switch closures), as follows:

Level 1. Servo - The Servo Level functions as the interface to the intelligent machine's actuators and sensors. It can be thought of as the device driver level. Task commands from Level 2 are converted into voltages and currents to drive electro-mechanical devices interfaced at this level. The Servo Level generally operates using high bandwidth synchronous closed-loop control and it is the lowest level of abstraction in an RCS hierarchy. Coordinate system transformations and set-point interpolation may be performed at this level. Each actuator may be paired with one or more sensors to effect closed-loop feedback control. The Servo Level periodically samples sensors and sends out drive signals to effect stable control and smooth movements.

Level 2. Primitive - The Primitive or Prim Level accepts commands from Level 3 and decomposes them into regularly spaced "move" commands to the Servo Level. Prim deals with machine dynamics and performs functions like thermal compensation. Prim may also do coordinate system transformations. Prim generally produces set-point output commands to the Servo Level in a synchronous fashion (evenly spaced in time). Sensory data, such as inputs from proximity, force, and torque sensors, are used to produce or modify path trajectories in real-time.

Level 3. Elementary Move - This level is also called the E-Move Level and can be thought of as the subsystem level within a single machine. The E-Move Level accepts subsystem task commands from Level 4 and decomposes them into commands for the Prim Level. "New" task commands from Level 4 generally arrive at irregular intervals, since they are typically driven by events occurring in the machine's environment. The E-Move Level deals with machine kinematics and is typically responsible for generating collision free path commands to be further decomposed by the Prim Level.

Level 4. Individual Machine - This level has many different names depending on the application. In factory automation it has been known as the Equipment Level or Task Level. The Individual Machine Level accepts task commands from Level 5 and converts them into tasks to the machine's subsystems. The Individual Machine Level is responsible for coordinating the actions of the subsystems within a single machine. The control system components are typically hosted on a single multiprocessor backplane at this level

and below. The backplane host is typically connected to a local area network (LAN) for communication with higher RCS levels.

Level 5. Group(1) - Level 5 coordinates the actions of a small group of machines and people (typically one operator). Generally the machine controllers communicate over some type of local area network. In factory applications this level has been called the Workstation Level and it supervises the actions of closely coordinated machines such as a conveyor, a robot and a machine tool. Level 5 controllers are typically hosted on computer workstations in factory applications.

Level 6. Group(2) - Level 6 has been referred to as the Cell Level in factory applications. It is responsible for coordinating the actions of several groups of machines and people or workstations. Level 6 accepts tasks from its supervisor (Level 7) and decomposes them into coordinated tasks for its subordinates at Level 5. Level 6 is typically hosted on a distributed network of multiprocessor computer systems or computer workstations (over a LAN).

Level 7. Group(3) - Level 7 is typically the Shop Level in factory applications. This is the third group supervisory level in an RCS hierarchy. Level 7 coordinates the activities of Level 6 controllers typically over a LAN. If the application calls for higher levels, Level 7 might be connected to a wide area network (WAN) for coordination with other Group(3) controllers via a Group(4) supervisor.

Level 8. and Higher - There is no upper limit on the number of levels in an RCS hierarchy. The number of levels of coordination and abstraction are strictly a function of the demands of the application (i.e., the organization of people, machines, communications links and tasks to be coordinated).

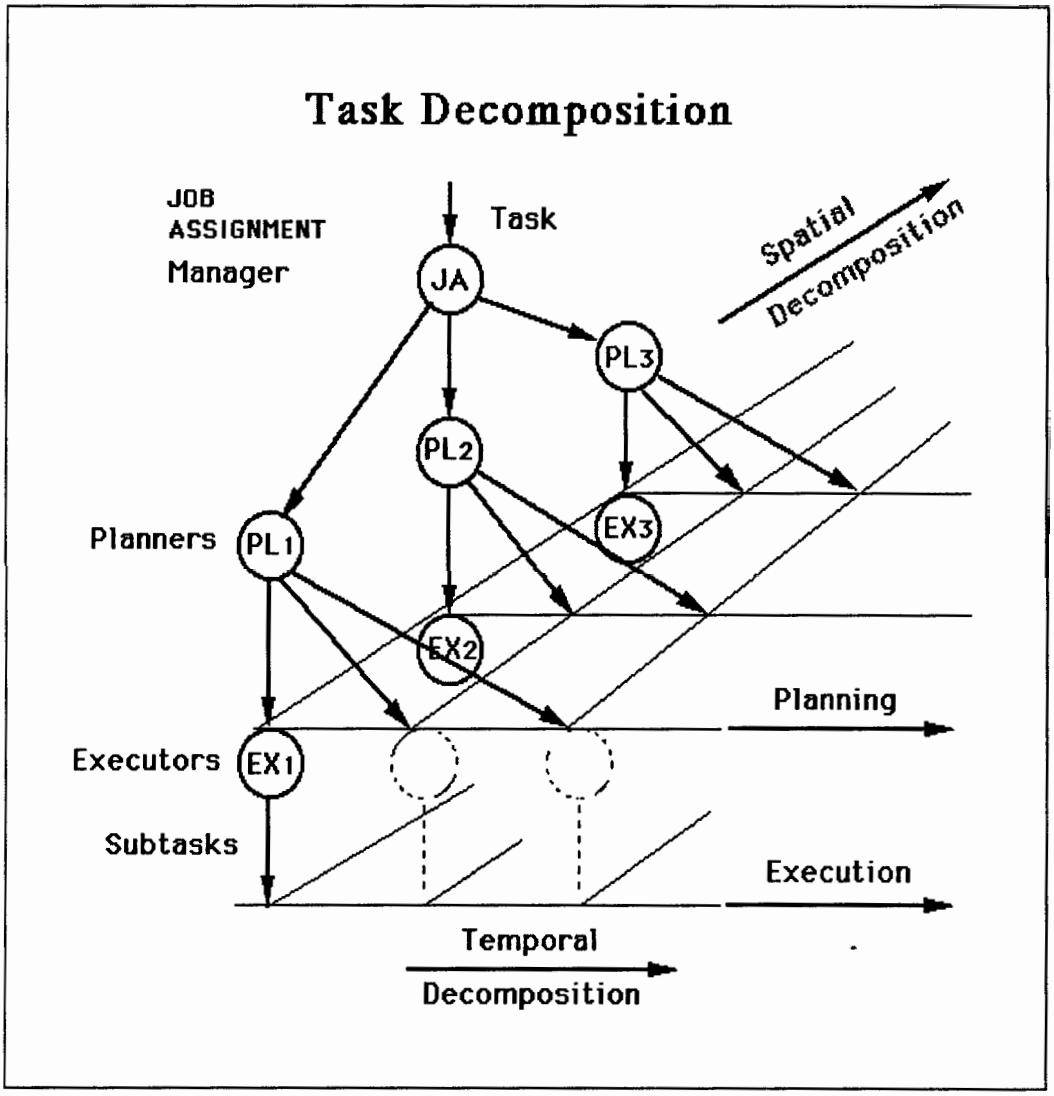


Figure 5.

2.3.4. Task Decomposition

In Figure 5, the task execution model for planning and execution between levels is decomposed into a repeating pattern of ***Job Assignment (JA), Planners (PL), and Executors (EX)***. We define the ***Behavior Generation (BG)*** function as containing the subfunctions JA, PL and EX. ***Job Assignment*** involves commanding subordinates to carry out concurrent tasks. Stored or generated plans temporally decompose tasks into sequences of sub-goals to be accomplished, to the limit of the appropriate planning horizon for a level. ***Planners*** are responsible for selecting pre-stored plans and/or generating new plans to be instantiated by the Executors. ***Executors*** instantiate the next step in the current plan (selected by the PL) based on the current state of the world as viewed via the World Model. Executors pass instantiated task commands to the next lower level JA, where this pattern is repeated, down the hierarchy, in a pipelined refinement of task detail. In general, subordinate levels deal with less abstract task details, at faster sub-goal completion rates.

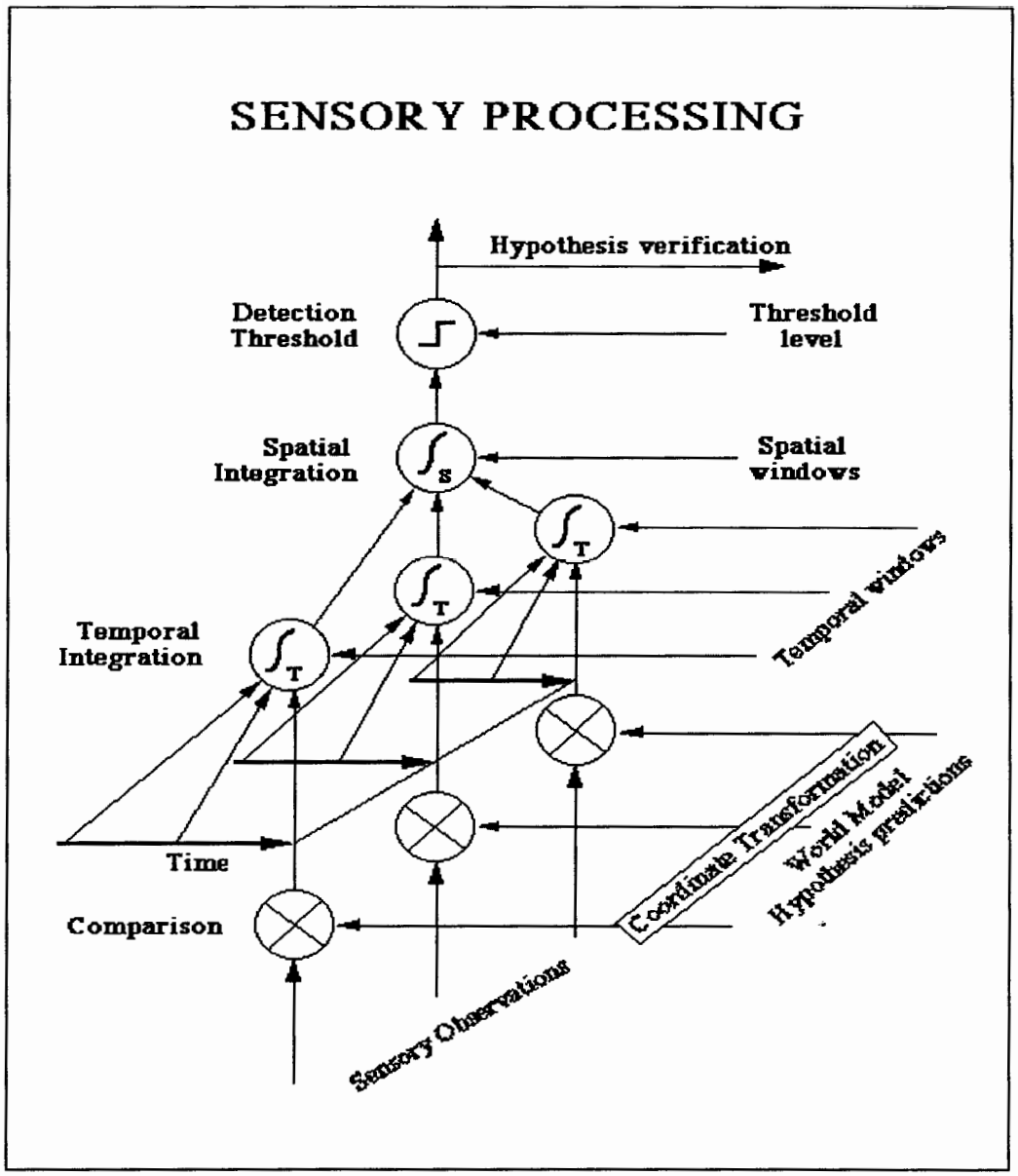


Figure 6.

2.3.5. Sensory Processing

Figure 6 deals with ***Sensory Processing (SP)***. The ***Sensory Processing*** function is responsible for filtering incoming sensor data, comparing the data stream with predicted values supplied by the World Model, integrating sensor data over time, extracting a historical trace of the data values, performing coordinate transformations on the data to allow data fusion from multiple sensors, and applying windows to the data stream in order to detect sensory input which has exceeded an event threshold. Sensory Processing works with the World Model to maintain the best estimate of the state of the world in real-time.

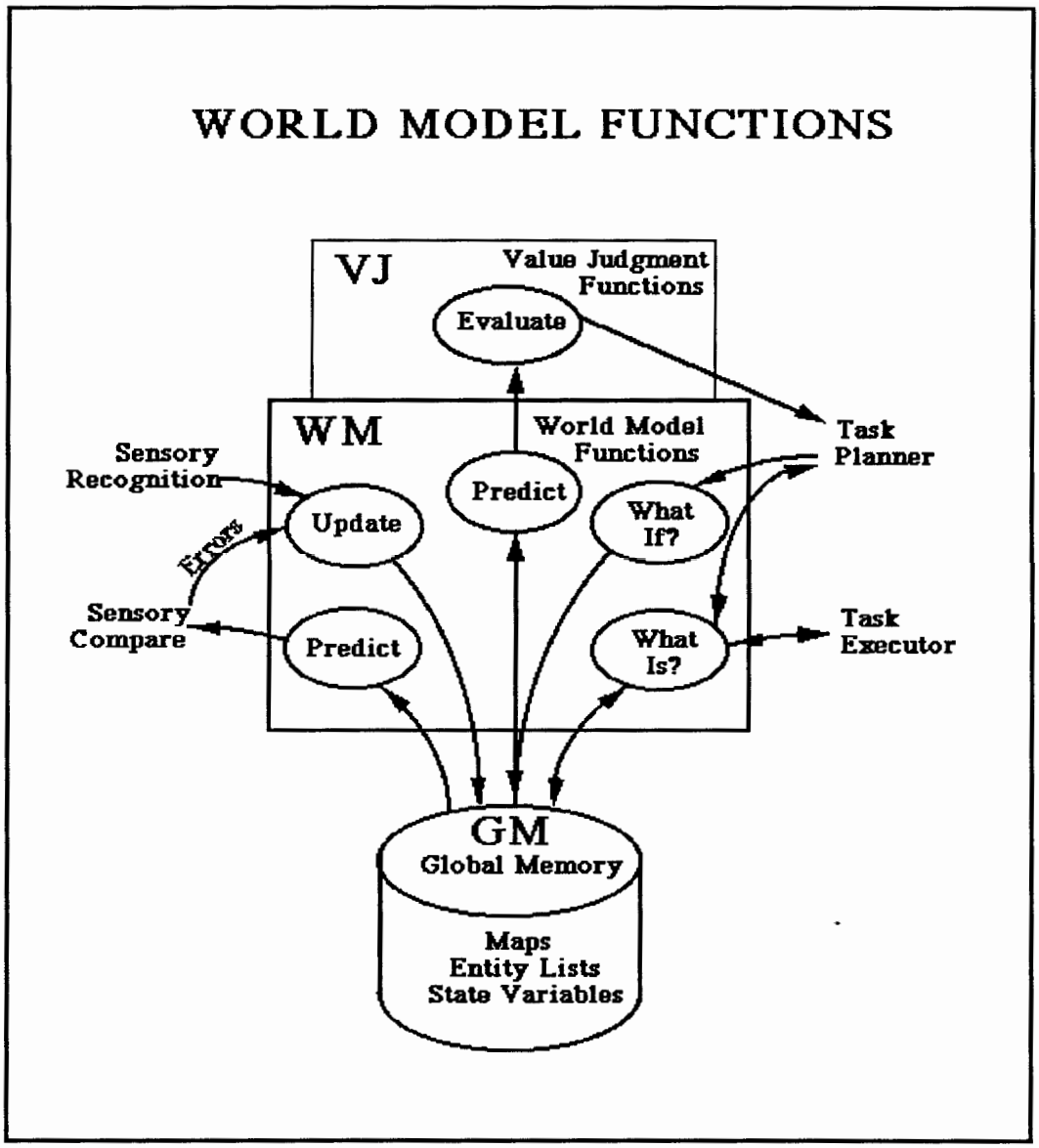


Figure 7.

2.3.6. The World Model

Figure 7 shows the World Modeling functions. ***World Modeling (WM)*** functions act as servers to the SP and BG functions (BG contains the JA, PL, and EX functions discussed above) by answering queries, accepting updates and generating predictions. WM also shares responsibility for performing value judgments and goal selection. ***Global Memory (GM)*** is the complete collection of globally defined variables. It may be thought of as the repository or knowledge base where controller nodes (i.e., SP, WM, and BG functions) store information to be shared with other controller nodes. In many applications Global Memory is implemented as a distributed database. GM can also be viewed as a combination of the communications mechanisms and the database repository necessary for implementing an RCS application.

The terms, ***World Model or World View***, are used to describe the intelligent machine's collective capability to perceive the world in which it functions (both external and internal). When we use these terms we are referring to algorithms for understanding the world, WM server functions, and the information stored in the Global Memory.

SECTION 3. RCS METHOD TENETS

The Random House College Dictionary, [Ra 82], defines ***tenet*** as, "any opinion, doctrine, dogma etc., held to be true". It defines ***canon*** as, "a body of rules, principles, or standards accepted as axiomatic and universally binding", and it defines ***canonical*** as, "pertaining to, established by, or conforming to a canon or canons". We will use the word tenet, here, to mean guidelines and engineering rules of thumb which characterize the RCS Methodology. Together the RCS Architecture definition and these tenets form a basic set of rules or systems integration standards for building real-time control systems. These standards can be considered the canons of RCS and they define a canonical form for real-time control systems design.

The RCS Methodology defined here draws upon elements of established software engineering methodologies (e.g., functional decomposition and structured design) and emerging methods (e.g., object oriented design and relational methods) while placing particular emphasis on the RCS ***task*** decomposition approach which has been developed at NIST and elsewhere over the last decade.

This RCS Methodology is based on the following empirically established method tenets which we will discuss in more detail in the remainder of this section. Tenets 1) through 5) are generally applicable to all RCS Methodologies while tenets 6) through 10) are expressed in terms of the Barbera approach and may differ from other interpretations of the RCS canonical form.

1) Use task oriented decomposition (driven by scenarios)
2) Use hierarchical organization and assign responsibility and authority
3) Organize the control hierarchy around tasks top-down and equipment bottom-up
4) Partition by an order of magnitude between levels (spatial and temporal resolution) and roughly ten decisions or less per plan
5) Use seven + or - two subordinates per supervisor and only one supervisor at a time
6) SP/WM/BG functions are distributed throughout RCS and assumed to exist in each node
7) Allow human I/F at any node
8) Controller modules are finite state machines communicating through Global Memory
    * Use a controller template as the basic building block
    * Use cyclic sampling rather than interrupts for context switching
    * Surround everything with data buffers
    * Use non-blocking input/output (I/O)
    * Implement Global Memory using a One Writer, Many Readers Paradigm
    * Match the control cycle time to the demands of the control application
9) Design for concurrent processing
    * Measure execution time performance
    * Allocate sufficient computing resources
10) Use synchronous control at the lowest levels transitioning to asynchronous control at the highest levels

3.1. Use Task Oriented Decomposition

Every methodology seeks to devise a model of the problem domain which explicitly represents and emphasizes the most critical components of the problem while simplifying those aspects which have a lesser impact on the solution being sought. Every model is, after all, a simplification of the real world. The trick is to choose an abstraction (or a set of abstractions) that highlights the parts of the problem that make a difference in the understandability, quality and efficiency of the solution. That is why it is so important to choose a methodology which matches the problem domain. In the domain of intelligent machine control systems we believe ***tasks*** are the driving factor.

The RCS Methodology approaches this problem by attempting to create a generic organizational modeling structure that formats information in a way that is patterned after human techniques of abstraction. It assumes that humans have a very limited capacity for breadth of information management and must resort to layers of abstractions so they never deal with too many pieces of information at a time [MI 56]. This technique is evident in most, if not all, human organizational structures. The layering of abstractions creates hierarchical organizational frameworks with detailed information at the bottom and higher and higher levels of information abstraction as one proceeds up through the layers. The RCS Methodology has, therefore, attempted to create a generic information representation mechanism of generic units that can be assembled into hierarchical structures with detailed processing at the bottom layers and higher level abstractions occurring at the upper layers. Thus, the methodology is not so much a model of the problem domain of a particular control application, as it is a model designed to be synergistic with human information representation techniques.

As previously mentioned, we define the domain of intelligent machine control systems in this document to include the control of electro-mechanical devices designed to perform some useful work using computerized control. Intelligent machines are further assumed to possess the capability to respond to the physical environment in some intelligent way, in real-time. This definition is intended to cover automation systems, embedded systems and robotic systems.

In this problem domain, practical control systems solutions are always defined within the context of the tasks to be performed (electro-mechanical actions to be taken). Our design models should strive to explicitly define all critical information related to carrying out these physical actions. We generally consider things like printed reports, displays to be generated, and other more traditional data processing chores to be of secondary concern.

The process of designing practical systems solutions always involves trading-off general purpose capabilities against task specific requirements. Examples of such trade-offs can be seen in common human transportation systems. People have developed automobiles, trains, buses, aircraft, rockets, elevators and escalators all for transportation but each for a specialized class of transportation tasks. Even though these examples can all be categorized as transportation functions, the specific systems solutions are all very different from one another. The tasks to be performed, the tools and machines to be used, the objects to be manipulated and the human interaction required, are the elements that distinguish any one system application from all others.

The attributes of tools, machines, and physical objects can be described independently of each other, but, task definitions are dependent on all three of these object types as well as being dependent on the human interaction required.

Recognizing this fact, this tenet suggests that ***task decomposition*** should be explicitly represented in modeling intelligent real-time control systems. Further it stipulates that task knowledge (i.e., control flow, processes, procedures, strategies and tactics), task specific object knowledge and task specific data processing knowledge should be encapsulated within modules which generate commands to their subordinates to accomplish system tasks when stimulated by task commands from their supervisors or in response to other communications (task state knowledge) arriving through Global Memory.

From a practical point of view this tenet suggests that it makes no sense to develop any sensory processing algorithm, data representation or world model algorithm if the parameters computed and/or stored are not required to accomplish some useful system task. It suggests that the only universe of object representations that are required are those which are meaningful within a task context (objects are the focus of attention for tasks). See Figure 1. New algorithms, data representations, and parameters can easily be added to an implementation when new tasks are added by either encapsulating them within existing modules or by adding new modules.

One of the first steps the systems engineer and the domain expert should complete is to produce a written narrative description or ***scenario*** of the operation of the intelligent machine system to be designed. A scenario should outline the expected sequence of events required to successfully accomplish system tasks under normal conditions as well as the event sequences required to deal with exception conditions and emergencies. We have found scenarios to be an excellent means of guiding the enumeration of all of the relevant task elements in an application-specific problem. The knowledge to be embedded in the computer based control system comes from human expertise and human memory. While the human mind is extraordinary in what it can contain, it is limited in its ability to retrieve information. The human mind appears to be an associative system rather than a random access system (i.e., one thought triggers another associated with it). The currently popular way for human domain experts to recall all of the relevant knowledge about a task is have them pursue a scenario of task activities that allows the associative triggering of all sorts of relevant details in the context established by each sequential element of the scenario description.

Albus [Al 91b] uses "task frames" to capture task knowledge in a nearly compilable form. When using Albus' technique, scenarios are required as a source of the knowledge to be stored in the task frames, as well as for testing and modification of the code that is generated from the task frames.

3.2. Use Hierarchical Organization and Assign Responsibility and Authority

The primary objectives of the RCS Methodology are to improve human understanding of the design, to manage software complexity, to provide for robust, verifiable, efficient, coordinated, real-time performance, and to provide for extensibility, portability and software reuse. Hierarchical organization has been found to be a key element in realizing these goals.

Humans have used hierarchical organizations to manage complexity and real-time coordination problems throughout history. There are also numerous examples of the power of hierarchical organization in the animal world. Humans, many insects, and other animals instinctively form social hierarchies to build habitats, hunt for or produce food, fight their enemies, reproduce, nurture their young, and in general improve their quality of life and their chances of survival.

People use hierarchies whenever complex real-time coordination of more than one individual is necessary. Notable examples are sports teams (managers, coaches, team captains/signal callers, player positions and specialized team groupings), enterprise structures (corporations, chief officers, divisions, plants, departments, sections, groups, worker specialties), and military organizations (Commander in Chief, Army, Theater, Corps, Division, Brigade, Battalion, Company, Platoon, Squad, Soldier). ***These organizational structures employ strategies and tactics in their procedures in order to deal with unstructured environments and uncertain information in an ordered way.*** Based on hierarchically structured plans (strategies and tactics) measures of performance can be defined to help refine real-time performance. These highly organized groups typically spend a significant amount of time training and practicing their designated functions in order to maximize their real-time performance and the quality of the result. Practice, training and performance measurement, often driven by scenarios that highlight critical parameters, are the techniques employed in order to develop, improve, and verify strategies and tactics (uninstantiated a priori plans).

In unorganized social situations people's behaviors can be defined independently of each other and each human being can be considered to possess roughly an equal capacity for reasoning and performing work. Therefore they can be said to be independent agents. When people are organized hierarchically, such as in a military situation, they agree to function within a limited range of responsibility, they are assigned a limited authority for decision making and they agree to be constrained in their actions by a chain-of-command. Even so, the people in such a structure still have roughly the same capacity for reasoning and performing work, no matter where they are placed in the chain-of-command. Each person simply performs his or her work at a different level of abstraction, at a different resolution, and within a different time horizon, given his or her position in the structure (using knowledge bases, strategies and tactics appropriate to his or her station).

The RCS analogy to these examples is a highly modular partitioning of process intelligence codified into a hierarchy of modules. Each module has a bounded clearly defined range of authority and responsibility, each deals with a particular layer of abstraction and timing horizons within the problem domain and each has clearly defined vertical, horizontal, and point to point channels of communication.

A popular software organizational structure often viewed as an alternative to a hierarchical structure is the notion of "independent cooperating agents". In this technique software agents are created which are roughly equal in authority and responsibility. These agents must then negotiate for a share of control over a finite pool of resources (computing time, actuators, sensors, fuel, power, goal designation, direction and speed of movement, etc.) in order to achieve a desired set of system goals. They negotiate according to some established set of arbitration rules (priorities, voting, ordered sets of constraint conditions, etc.) and usually use message passing over a computer network as a communications scheme (i.e., client/server systems).

The independent cooperating agent technique is analogous to the human parliamentary organizational structure. Humans use this type of structure to establish policy, to define arbitration rules, to divide resources, and to create or modify rules or laws of social interaction (e.g., government, civic organizations, standards bodies, etc.). Such human organizations are not known for their efficient real-time performance. The results achieved using this type of structure are often unpredictable (non-deterministic). Seemingly insignificant perturbations of the inputs to such a system can result in quite unexpected major changes in the resulting behavior. It is difficult to verify all of the possible behaviors that might be exhibited given responses to changes in the input space for such systems. The principal utility of such structures is to ensure some measure of "fairness" in the result as well as a social sense of "due process".

In short, we don't believe the use of the independent cooperating agent technique alone (without hierarchical organization) is a better choice for achieving complexity management or robust, verifiable, efficient, coordinated, real-time performance. On the other hand, an RCS implementation can certainly benefit from the judicious use of these negotiation techniques when the application warrants them. This can be particularly evident when implementing a system which, by its physical nature, requires negotiated resource sharing.

RCS accommodates both hierarchical command and control as well as negotiated resource sharing between controller modules (i.e., cooperating agents). An RCS system can be defined as a hierarchy with a fixed chain-of-command where each controller module has a fixed set of plans and algorithms built in. However, RCS can also be implemented with conditional responses or modes of operation such that the command tree can be reconfigured from moment to moment depending on conditions reflected in the World Model. RCS planners (PL functions) can work from a fixed set of pre-computed plans or they may generate plans based on heuristic search strategies or game theoretic statistical methods. Modules may negotiate among themselves for assignment of subordinates, or the allocation of shared resources (e.g., time, tools, fuel, etc.) needed for their respective assigned jobs. RCS in no way prohibits or even inhibits the use of cooperating agents technology. RCS simply embeds negotiation within a regular hierarchical computing structure where the unpredictable nature of these methods can be safely handled.

3.3. Organize the Control Hierarchy Around Tasks Top-Down and Equipment Bottom-Up

The process of developing an RCS application and organizing its control components is an iterative one which might begin with a top-down decomposition of tasks to be performed (forming a task tree structure) and the organization of a hierarchy of modules which will be

responsible for coordinating the tasks to be performed in a bottom-up design procedure. The order in which these two activities take place is not specified and is less important than realizing that the objective of this iterative design process is to map the task tree onto the controller hierarchy, as shown in Figure 8.

By organizing an RCS hierarchy around the equipment to be controlled (machines, actuators and sensors) in a bottom-up process we can minimize and simplify the difficult real-time problems of resource contention, conflict resolution and scheduling. Every man-made machine ultimately has a finite number of actuators and sensors through which it can influence the problem environment.

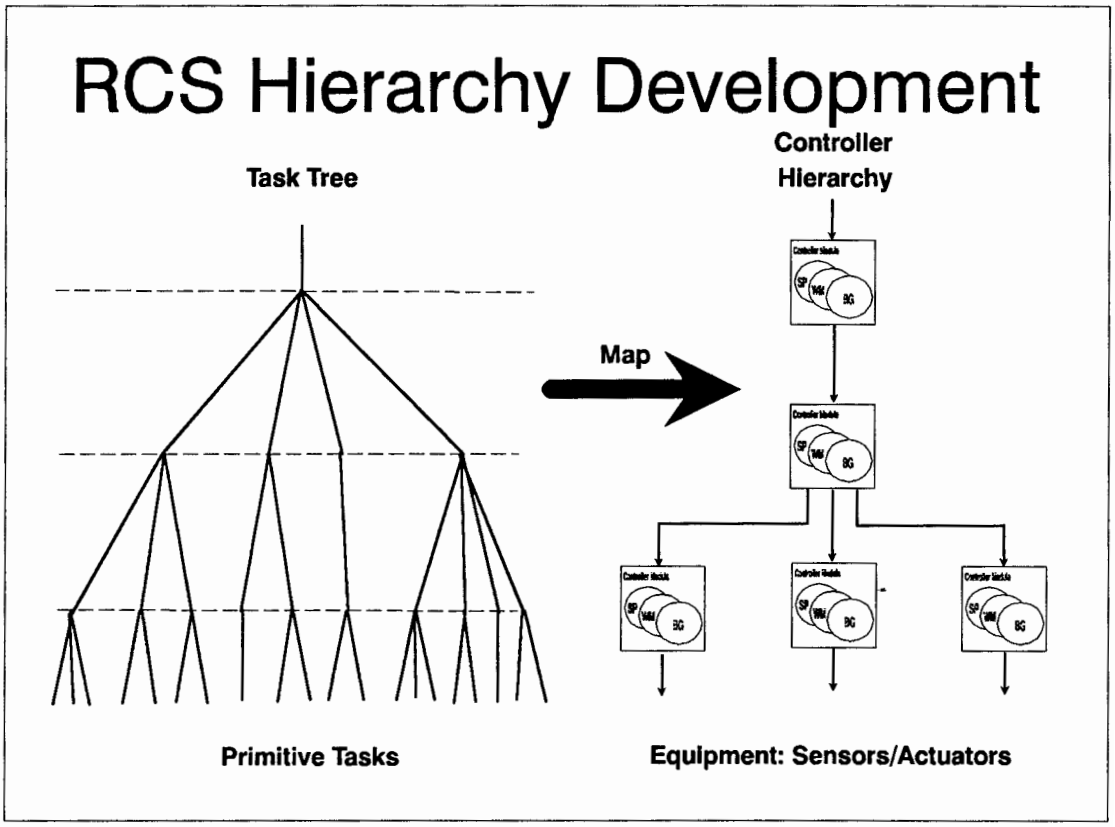


Figure 8.

We can capitalize on this fact by structuring a supervisor-subordinate hierarchy of intelligent controllers using a bottom-up approach starting from the actuators and sensors that the intelligent machine will be directly controlling. We can ensure that every subordinate is directed by only one supervisor at any instant in time (one software read/compute/execute cycle) and that there is a clearly defined supervisory module for each individual actuator and sensor at any instant in time. Using this structure it is then possible to define coordination and scheduling plans to be used by each controller module within the scope of its level of authority and responsibility. Furthermore because ***the hierarchical structure predefines the responsibility and authority of each controller module*** and the resources it may control at any instant in time, the problems of resource contention, conflict resolution and scheduling are dealt with locally within each supervisory controller module (as defined by its local library of plans) and are independent of the concurrent actions of all other controller modules at the same level of authority.

We imply, above, that the hierarchy may be redefined in between control cycles. In fact, this has been accomplished in laboratory demonstrations involving reassigning a mobile robot cart to different work cells in the NIST AMRF, as it transported parts trays from work cell to work cell, [Mc 82]. Another AMRF example is the automatic changing of end-effectors (a gripper) on a robot in real-time. Resource sharing or restructuring the RCS hierarchy in real-time, however, is still very much a topic for further research.

Whenever practical the designer should group hardware components (actuators and sensors) and software components (RCS modules) so as to minimize the number of interfaces necessary in order to implement closed-loop control through the RCS hierarchy at the lowest level practical. The intent of this guideline is to localize the design of closed-loop control within RCS (into modular subsystems) in order to minimize the potential for "ripple effect" when evolving, expanding or maintaining the implementation and to minimize the need for and extent of high bandwidth feedback loops. Communications bottlenecks can also be eliminated by judicious grouping of closely coordinated components.

Physical constraints within the problem domain often will conflict with this guideline thereby imposing engineering trade-off decisions (e.g., important sensors or actuators may be physically located on different parts of the machine or on separate structures). Other RCS tenets listed here might also conflict with this guideline. For example the " seven + or - two" tenet suggests adding hierarchy (and therefore additional interfaces) in order to manage complexity and improve human understanding. The "order of magnitude" tenet can also suggest the addition of more levels in the hierarchy and additional interfaces again in the interest of complexity management and understandability. As always, the engineer's function is to perform the engineering analysis necessary to arrive at a viable engineering design trade-off.

3.4. Partition by an Order of Magnitude Between Levels and Roughly Ten or Less Decisions per Plan

In robotic applications and other control systems problems that are concerned with achieving smooth motion and closed-loop stability, a widely accepted rule of thumb suggests that nested (or hierarchical) control loops should be separated by an order of magnitude in closed-loop bandwidth.

In dealing with the issues of minimizing the impact of complexity and providing for human understanding of the design result, we have devised a rule of thumb that suggests limiting the scope of the problem domain for any individual controller module or RCS level to roughly one order of magnitude in terms of the resolution of the maps and the geometric models it uses for planning and in terms of the timing horizon it uses for planning and scheduling. This rule of thumb should not be strictly interpreted especially at the higher RCS levels since its intent is to limit the number of planning decisions or steps any one planner needs to plan into the future. In some cases the number of required planning steps is event driven rather than strictly related to the passage of time. Good engineering judgment needs to be exercised when applying this rule of thumb since the objective here is to create modularity and thereby to limit the scope of complexity

in the coordination of tasks and to limit the number of planning decisions or steps to be planned into the future to roughly less than ten. Some individual steps may have very long durations.

The amount of task decomposition between levels is typically reflected by the number of subordinates a supervisory module directs. If there is a single subordinate module under a supervisor, then the decomposition of a task by the supervisor, from its supervisor, to some set of output commands to its single subordinate is in the range of one to ten (i.e., the task is decomposed into less abstract subtasks). A module implementing this type of decomposition emphasizes Planner and Executor subfunctions. If, however, there are a number of subordinates being coordinated by a supervisor, then the amount of task decomposition occurring within the supervisory module may be much less (i.e., the output commands from the supervisory module to its subordinates are almost at the same level of abstraction as the input commands to the module). In this case, the role of the supervisor may be primarily in coordination or synchronization of multiple subordinates, rather than in decomposition of tasks from a higher level of abstraction into significantly simpler subtasks. In this instance the supervisory module is performing a Job Assignment sub-function.

3.5. Use Seven + or - Two Subordinates per Supervisor and Only One Supervisor at a Time

This tenet seeks to manage task knowledge complexity and human understanding of the design by adding hierarchy when more than roughly nine subordinates must be supervised by a controller. This rule takes its name from George Miller's early work, [Mi 56]. He empirically found that human beings are able to manage seven + or - two things at a time without losing control. This guideline suggests adding a supervisory controller module whenever more than nine subordinates must be controlled and that the optimum number of subordinates per supervisor is roughly five. Again software engineering judgment is always called for in applying these guidelines. It is possible that in a particular application only one supervisor may be needed for a much greater number of subordinates (e.g., twenty or more) if all of the subordinates are performing very similar tasks. We sometimes see this pattern in human organizations such as in assembly line production.

3.6. SP/WM/BG Functions are Distributed Throughout RCS and Assumed to Exist in Each Node

The RCS architecture has evolved at NIST from a controls system point of view. It is grounded in the notion that closed-loop control is a fundamental construct of the architecture style. With this in mind we have defined a ***generic controller module*** (Figure 9) as the basic processing entity through which closed-loop control can be implemented, therefore a generic controller module must be capable of performing the basic closed-loop control functions of Sensory Processing (SP), World Modelling (WM) and Behavior Generation (BG). An RCS architecture is then built of RCS controller modules interconnected to form a hierarchical tree structure of controller modules with one or more modules forming a ***node*** in the hierarchy. Furthermore the modules are designed to be interconnected to form nested closed-loop control with each node contributing to one layer of problem abstraction and decomposition.

While the RCS architecture philosophy focuses on highly modular closed-loop control, it does not require closed-loop control within every implemented module. In fact it would be possible to implement an entire RCS compatible architecture without including real-time sensory input for closed-loop control. This implies that while SP, WM and BG functions are distributed throughout the hierarchy and may exist within every controller node, ***there is no requirement that more than one of these functions (or their subfunctions - JA, PL, and EX) be non-trivial, or even exist, in any one implemented controller node.*** In practice open-loop control nodes often occur in RCS especially when implementing a simple on/off control function. In this case closed-loop control might be implemented at the next higher level through a supervisory controller node.

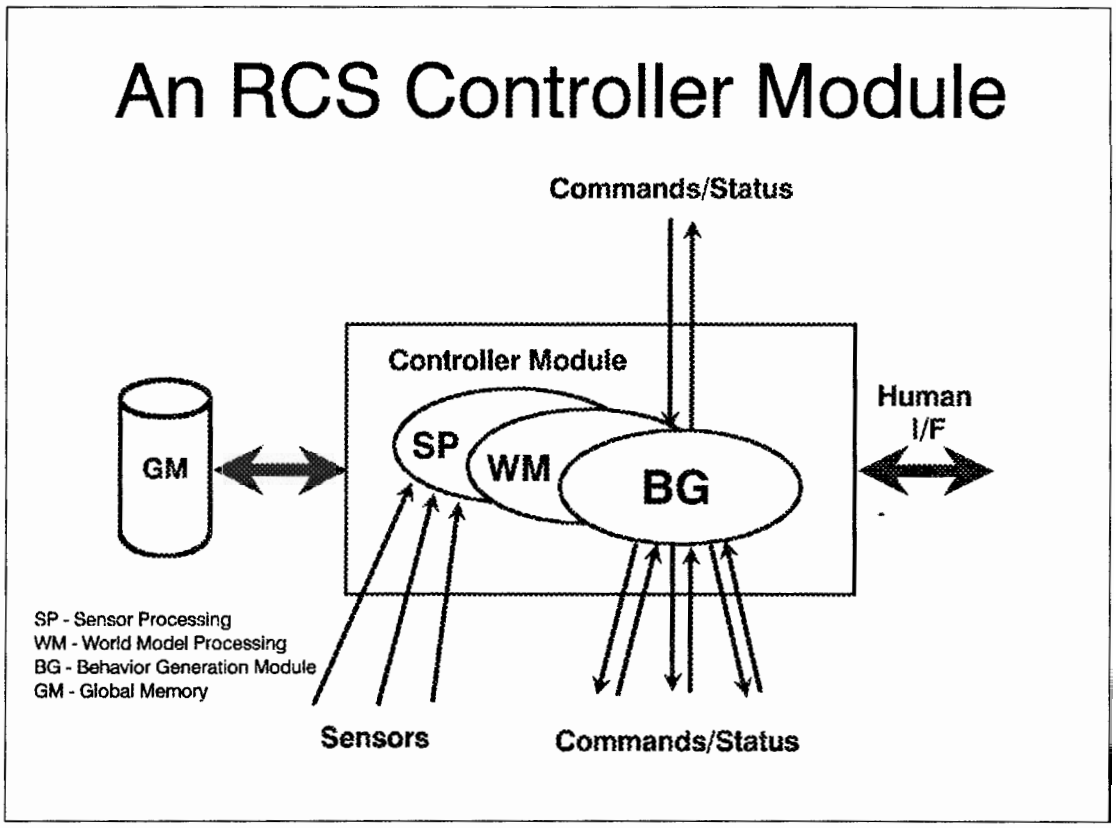


Figure 9.

3.7. Allow Human Interface at any Node

Recognizing that almost every practical system must have a human interface at some level and that it is usually desirable to develop intelligent machine systems in an evolutionary manner, we believe it is prudent to build the possibility for human interface into every controller node as part of the system design philosophy as opposed to adding the human interface capability in an ad hoc manner as the need arises. This objective can be realized by providing for human interface in the design of the generic RCS controller template to be discussed in a subsequent section of this document.

The important philosophical point to be made about the human interface is that human interaction is managed by the control system in the same manner as all other input/output (I/O). Any branching required, as a result of human interaction, is handled without the use of interrupts. If the human operator wishes to cause a certain command at a particular level to execute, he does so by requesting that the supervisory module, with the authority and responsibility to do so, issue that command. In this manner, the supervisory module is able to control the access to its subordinates, allowing human interaction only when and where appropriate, thereby preserving the chain-of-command. The human is never allowed to directly break or bypass the chain-of-command. This results in a known controlled interaction, allowing the system to maintain knowledge as to the state of the system during human interaction, which simplifies the problem of resuming automatic behavior after the human withdraws. Human interface is always buffered and accomplished using non-blocking I/O in order to ensure that the RCS control system is always running and capable of detecting and reacting to exception conditions even while interacting with a human operator.

3.8. Controller Modules are Finite State Machines Communicating Through Global Memory

In keeping with the controls systems origins of the RCS philosophy as well as the emphasis placed on programming for concurrent multiprocessor environments this tenet defines an RCS controller module (Figure 9) as a finite state machine. Finite state machines are characterized as functioning on a read/compute/write execution cycle. Such machines divide time into discrete increments based on their execution cycle duration. An instant of time in this context is considered to be one read/compute/write execution cycle (the time period of one control cycle). Finite state machines deal with external stimuli as snap-shots in time and have a deterministic and verifiable response which depends solely on the internal state of the machine and the event or external stimulus being presented to the machine (through Global Memory) at any given instant in execution time.

An integrated set of rather simple finite state machines can exhibit amazingly complex and intelligent dynamic responses to uncertain stimuli even though they are completely deterministic and verifiable. These properties make them an excellent choice as the basic execution model for RCS applications.

The following guidelines are offered as an extension of the finite state machine implementation method:

* Use a controller template as the basic building block
* Use cyclic sampling rather than interrupts for context switching
* Surround everything with data buffers
* Use non-blocking input/output (I/O)
* Implement Global Memory using a one writer, many readers paradigm
* Match the control cycle time to the demands of the control application

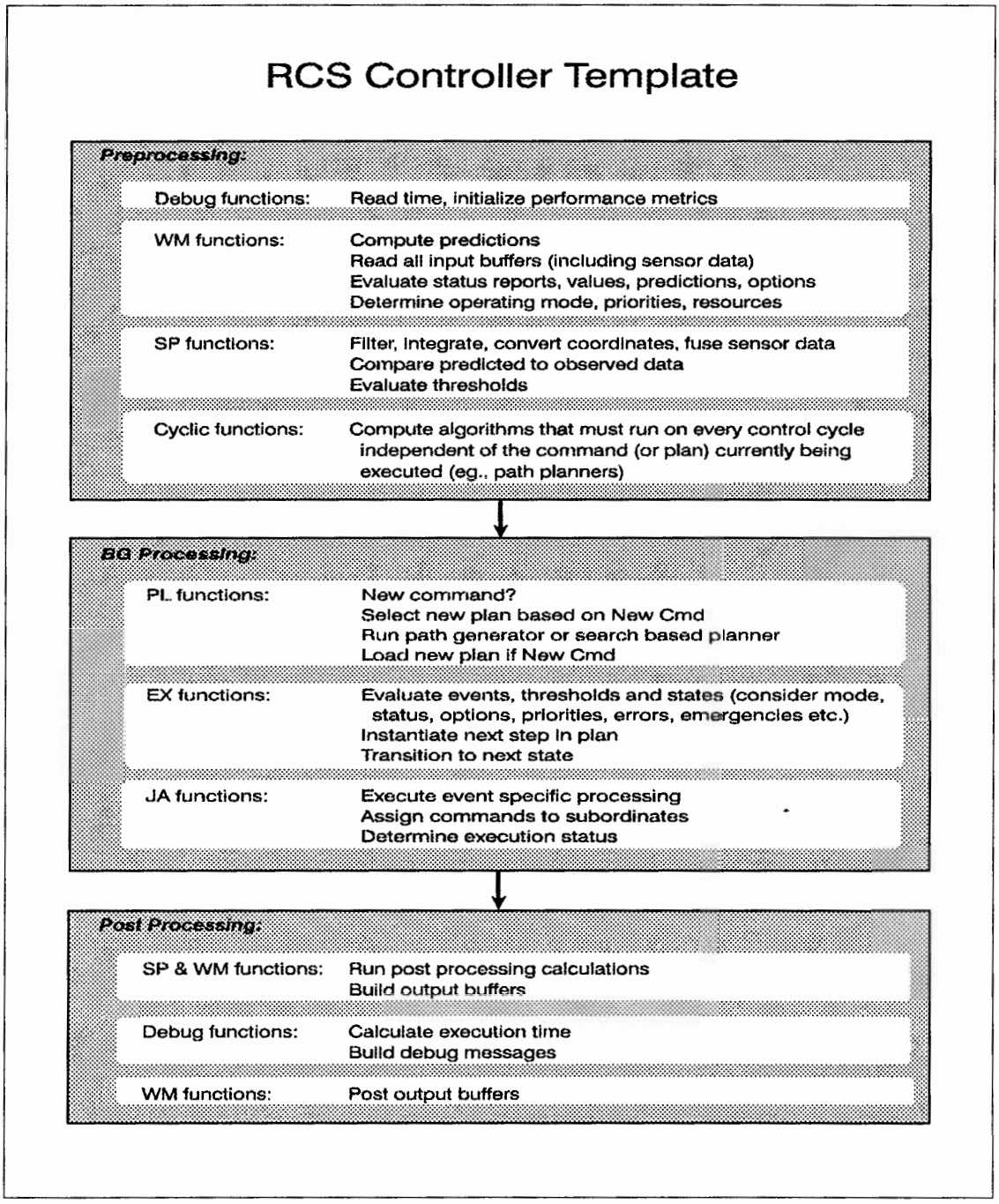


Figure 10.

3.8.1. Use a controller template as the basic RCS building block.

A controller template can be implemented in almost any computer language to provide the basic "hooks" for communicating between controller modules, as well as to contain most of the overhead functions necessary to comply with the RCS integration standards (including the ability to measure execution time performance). Such a generic template, as shown in Figure 10, can

then be replicated and filled in (instantiated) with task specific plans, state tables, and algorithms at design time. Appendix A provides example C language code which we used in implementing controller module templates for our submarine automation project.

A controller template can be used to implement any of the fundamental RCS subfunctions (e.g., SP, WM, JA, PL, EX). A controller module built using a template may contain any number of SP and WM functions but only one BG function (JA, PL, EX). It is also possible to construct a controller node using a controller module for each sub-function. This technique would resemble Fiala's [Fi 90a] implementation of atomic units. In many cases a controller node can be implemented with a controller module which emphasizes one or two subfunctions while the rest exist only trivially. In other words, a particular controller module instance might only implement a planner function. In another instance a module might perform a Job Assignment function, coordinating several subordinates. Another module might be responsible for executing one or two pre-stored rule plans (an Executor function). In still another instance an implemented module might perform mostly Sensory Processing and/or World Modeling functions while only performing trivial BG functions.

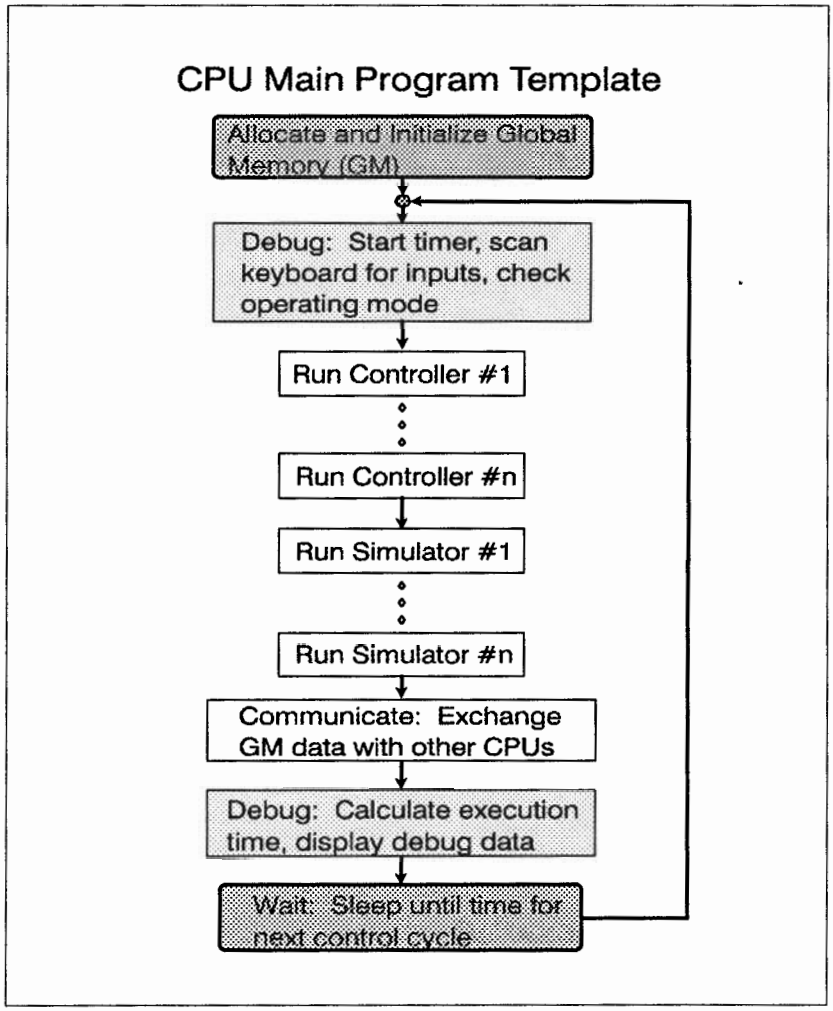


Figure 11.

3.8.2. Use cyclic sampling rather than interrupts for context switching.

Control flow within an RCS design is altered by executing plans (state tables) within controllers (state machines), resulting in verifiable deterministic behavior. At the very lowest levels, hardware interrupts are used to latch or buffer incoming sensor data that might happen between control cycles. Once the data is latched and buffered it is sampled on the next control cycle. Data is not queued up in serial data buffers, however. The RCS control cycle time must be selected and implemented to ensure that every significant incoming piece of data can be sampled and processed before the next sample arrives.

To implement the basic "heartbeat" control cycle in an RCS implementation we can use a simple wait loop or we might also use a time based interrupt, as shown in Figure 11. However, interrupts are never used to alter control flow in real-time, in a Barbera approach RCS implementation. Allowing a real-time operating system to insert a branching action in the control sequence (not explicitly modeled in an RCS plan), is not acceptable. Such interrupt branching is non-deterministic and usually difficult if not impossible to verify. Interrupt techniques also make it difficult to calculate and measure execution time performance.

Interrupt driven multi-tasking environments are expressly designed for the purpose of sharing the available computing resource of single CPU systems. RCS accomplishes multi-tasking by requiring the designer to make an explicit assignment of software modules (controllers) to specific CPUs at design time. Execution time is then measured to ensure that the "worst case" execution performance falls within the required real-time performance specifications. Multi-tasking is accomplished within RCS by implementing a "Main Program" for each CPU in the system (see Figure 11). The Main Program is then used to simply schedule, in sequential fashion, each controller that will share that particular CPU. Any number of controller modules can share a CPU as long as the Main Program can complete the execution all of its assigned modules within the design requirement for the cyclic execution time of one control cycle.

3.8.3. Surround everything with data buffers.

The independence of RCS controller modules is preserved in an RCS implementation by surrounding every controller with data buffers in a triple buffering scheme, as shown in Figure 12. This allows the designer to reassign software modules (controllers) to any CPU and to any position in the sequential order of execution within the execution schedule of one CPU (controlled by the Main Program). Triple buffering means each controller maintains a set of buffers for all of its own I/O, and each CPU maintains a set of buffers (Global Memory buffers) for all of the I/O data its assigned controllers use, therefore there are three sets of buffers involved in a controller to controller communication (the writer's local buffers, the global memory buffers, and the reader's local buffers). Each controller reads its inputs and writes its outputs once each control cycle. The Main Program is responsible for communicating with all other CPUs in the system. Once each cycle the Main Program determines which of its data buffers need to be written (updated) to other CPUs and it reads (receives updates) data written by other CPUs (see Figure 11). Communications is discussed in more detail in later sections. Appendix B provides example C language code for a Main Program.

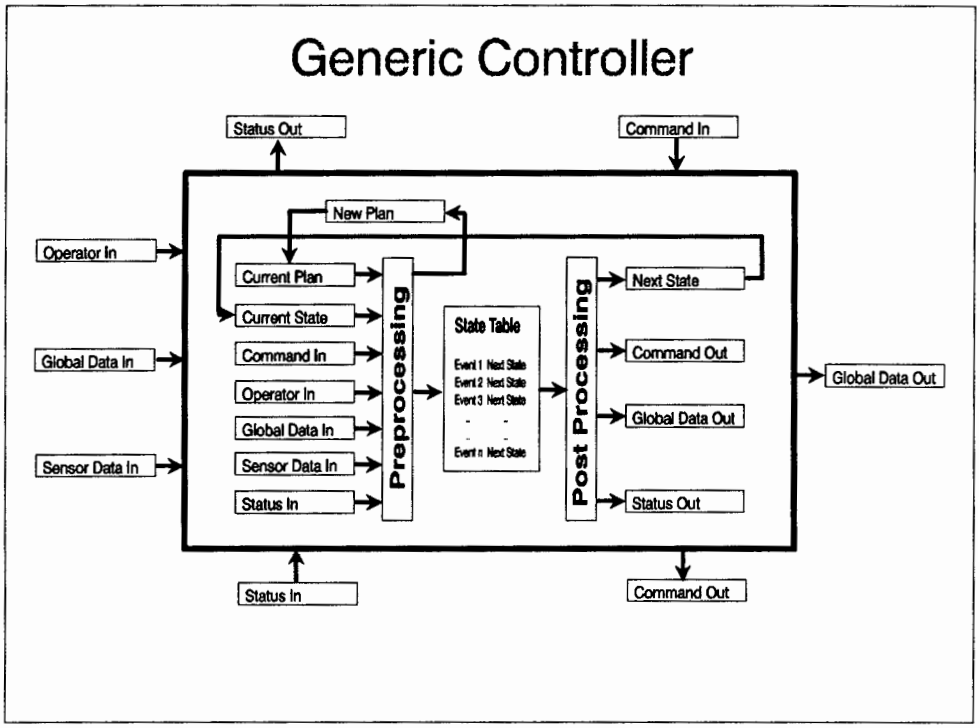


Figure 12.

3.8.4. Use non-blocking input/output (I/O).

The Barbera RCS approach uses non-blocking I/O, **only**, in order to ensure that the RCS system is always capable of responding to changes in its inputs (human commands or sensory input). This also means that RCS is always operating on the latest available data (data queuing is not used). In an RCS implementation the control cycle is never blocked waiting for an input or an acknowledgement that a receiver has read output data. Since this RCS Method is based on cyclic sampling, non-blocking communication is required in order to preserve controller response time.

3.8.5. Implement Global Memory using a One Writer, Many Readers Paradigm.

In many RCS applications Global Memory is implemented as a distributed database. Data is written by only one writer and may be read by any number of readers. Data is linked between writers and readers at design time through the definition of interface buffers for each controller module in an RCS hierarchy. The highest communication bandwidth typically exists between SP, WM, and BG processes within an RCS controller module. It is important to note, however, that it is possible to exchange data between controllers at any level in the hierarchy through appropriate communications mechanisms. Also note that only commands and status are constrained to communicate in a hierarchical fashion between superior and subordinate controllers, even though we generally implement commands and status as Global Memory interface buffers just like all other data. Data transfers through Global Memory may be horizontal, vertical or point-to-point between any writer and any set of readers. This allows us to easily implement debug monitoring of the communication between controllers.

Commands are never queued up in a subordinate's input buffer. Instead, supervisors either assign a new task to a subordinate after receiving a "done" status, or they abort the subordinate's active task by issuing a "new" command. New commands automatically supersede any assigned task in this RCS Methodology. This means that we never expect a subordinate to resume a task that has been superseded, without being commanded to do so (usually through a series of commands which re-initialize the task to its previous state) by its supervisor.

3.8.6. Match the control cycle time to the demands of the control application.

In an RCS application the basic control cycle time and sampling rate must be matched to the physical demands of the control application. Åström and Wittenmark define sampling in the context of control and communication as, "a continuous-time signal replaced by a sequence of numbers, which represent the values of the signal at certain times", [Ås 90]. The choice of sampling rate depends on the properties of the analog and discrete sensory input signals to be sampled, the signal reconstruction method chosen, and the purpose of the system. The closed-loop servo rate is chosen to effect stable control and smooth motion given the natural frequency of the physical (mechanical) system being controlled. In RCS systems the reaction time of the system to detected events is determined by the CPU control cycle time.

In many RCS applications the sampling rate, closed-loop servo rate, and CPU control cycle time are all set to be equal to the heartbeat rate. A worst case analysis is performed to select the heartbeat rate. If the analysis shows that there is insufficient computing speed available or that communications latencies make it impractical to chose a single worst case rate as the heartbeat then other options should be considered (e.g., filtering and detecting very high frequencies as a preprocess before sampling, multiple rates among multiple CPUs, variable rates, etc.).

Our first concern is that the sampling rate must be chosen to accurately represent the highest frequency analog input signal in the control application. The sampling rate must generally be at least twice the rate of the highest sensory input signal frequency as a first approximation. In most real-time systems zeroth order hold devices are used for causal reconstruction since reconstruction delays are unacceptable. The accuracy of signal reconstruction can be analyzed using the Nyquist Sampling Theorem. Signal sampling accuracy can sometimes require a rate of six to ten times the highest frequency of concern. In some applications the signal rise time might require setting the sample rate to as much as ten samples per rise. Digital signal frequencies (e.g., switch closures) must also be analyzed to ensure the latching and processing of every occurrence of the discrete input signal with the highest potential frequency.

In robotics and many other control applications we must ensure that our closed-loop servo rate is chosen for control stability and we are often interested in achieving smooth motions. Stability and smooth motion requirements may dictate higher servo loop rates. The Nyquist stability criterion [Ku 67] or other analysis methods should be used to evaluate stability. Analysis often dictates closed-loop servo rates of two to ten times the highest desired position control bandwidth. To achieve smooth motions the closed-loop servo rate might be set as much as fifty times the desired position control bandwidth.

Communications and compute time latencies can be a principle cause of instability in high speed servo control systems. These problems become apparent when implementing very high speed servo-loops while computing complex algorithms (e.g., large matrix inversions). Robot Systems Division researchers have implemented RCS control systems using pipe-lined multiprocessors running at different cycle rates when dealing with control applications which push the state-of-the-art in real-time high speed computing. Fiala explores methods of dealing with these issues in [Fi 90a] and [Fi 90b].

We must also consider the reaction time of the RCS system to events being monitored. The CPU control cycle rate chosen must meet the worst case response time requirement for reacting to event inputs (intelligent context switching).

The sampling rate we chose will set the basic control cycle time or heartbeat for the RCS system. Selecting a heartbeat cycle time is a trade-off problem between the criteria discussed above and the number of instructions per cycle that can be executed on any given CPU. In general we would like to choose the longest heartbeat cycle possible which still meets all of the worst case requirements for sampling, stability, smooth motion and reaction time. By doing so we can achieve a good balance between computing capacity per cycle and closed-loop real-time performance.

3.9. Design for Concurrent Processing

RCS is an inherently parallel programming method which seeks to develop software processes (controller modules) which will execute in a cyclical and concurrent manner on multiple CPUs. By designing for concurrent processing we can more easily match the control system implementation to the concurrent nature of most real world intelligent control system problems. The software processes we design can then be easily distributed across a multiprocessor hardware computing environment in order to meet real-time execution constraints. These controller modules should be thought of as finite state machines executing in parallel, preferably on a fixed read/process/write control cycle which can be viewed as the system's "heartbeat". As a consequence of this tenet the RCS Method differs from traditional procedural programming methods, which are aimed at producing sequential code to efficiently execute on a single CPU.

In general the Barbera RCS method is less concerned with optimizing the utilization of any one CPU and more concerned with managing software complexity and providing a means to ensure deterministic and verifiable real-time performance.

This tenet suggests that for many practical applications it is often more cost effective to add microcomputer hardware (single board computers and memory) to a multiprocessor environment in order to meet real-time needs. The other alternative is to procure a single central processing unit (CPU) with sufficient computing speed to meet the worst case real-time demands of the application. This can often lead to the purchase of very expensive computing platforms. If we elect to use real-time operating system interrupt services to dynamically manage the central processing unit's computing power, we will often increase the complexity of our control system application programs and make code verification more difficult, as part of the bargain.

Fortunately we now live in an age when microcomputer hardware has become quite affordable for many applications. In the early days of computerized control systems, engineers had to go to extremes to preserve computing resources because they were so expensive. Many applications were simply not feasible from an economic perspective. Operating system services like interrupt processing, multi-tasking, and time slicing were some of the techniques employed to help engineers allocate this costly resource. Today we can trade-off inexpensive hardware against the requirements of managing complexity and achieving verifiable performance especially in very large systems. Given today's hardware prices, size, and power requirements, we can now afford to explicitly design for concurrent computing in a multiprocessor environment without resorting to non-deterministic interrupt processing.

The following ideas are directly connected to this emphasis on concurrent processing:

* Measure execution time performance
* Allocate sufficient computing resources

3.9.1. Measure execution time performance.

Real-time is defined as generating a reasonable and effective solution in time to maintain stable control of the problem. An optimum solution arriving too late can often result in disaster. The lesson in this definition is that real-time systems must always be designed to meet or exceed "worst case" timing requirements. It is incumbent on the systems engineer to measure and verify real-time performance. The implementation approach described in this paper includes building in execution time monitoring in every RCS controller and within each "Main Program" (defined later) in order to guarantee the design meets the real-time requirements of the application, as shown in Figure 10.

It is easy to redistribute the SP, WM, or BG functions (or any set of sub-functions) which might logically reside within a single controller module across more than one controller when one function or sub-process is a "compute hog". When a particular process requires more CPU processing power than remains available (within the established execution cycle time) on a shared CPU, it is reduced to an "atomic algorithm" and implemented within its own separate controller module. That controller can then be assigned to execute on its own dedicated CPU or a shared CPU with sufficient reserve computing capacity to run concurrently in a multi-processing environment.

An example of this could be a compute intensive, algorithmic path planner which is assigned to run concurrently on a dedicated CPU. A controller containing only this path planner would operate under a single task command, "run", and it would compute a new path plan by reading in the appropriate state variables and computing a solution on every control cycle. If such an algorithm could not be made to run in a single control cycle, then it might be allocated two or more cycles to complete a new path solution. Each time its solution becomes available the controller would post the result (the path plan) in Global Memory to be read by the appropriate controller (the controller that executes the path plan generated).

3.9.2. Allocate sufficient computing resources.

The Barbera RCS design philosophy advocates trading-off additional hardware in the interest of minimizing software complexity. Rather than trying to create a customized point solution which attempts to maximize the utilization of existing computing hardware, we would simply add another CPU whenever the computing capacity is overloaded by additional software demands. In the RCS Method described here, controllers may be assigned to execute on a shared CPU as long as their combined execution time performance does not exceed the required basic control cycle time. If adding another controller module to the Main Program or another plan, or sub-process to an existing controller module causes the cycle time to be overrun, then add a CPU and redistribute (assign) the software modules (controllers) to the expanded hardware resources.

3.10. Use Synchronous Control at the Lowest Levels Transitioning to Asynchronous at the Highest Levels

A Barbera RCS implementation employs periodic sampling and communication as opposed to interrupt processing (for those processes executing within a backplane) in order to ensure deterministic and verifiable behavior (in terms of execution time and response time), particularly at the lowest levels of the architecture. This synchronous sampling approach has been implemented using a simple software wait-loop or a synchronizing timer to establish the "heartbeat" control cycle with commercially available real-time operating systems.

At higher levels of an implementation it is often convenient to transition to an asynchronous communications technique. One obvious transition point is whenever the control system must coordinate physically separate machines. For example a milling machine (Individual Machine Level) might need to communicate over a local area or wide area network with its supervisor (a workstation controller at the Workstation Level) or any of its peers (e.g., a robot, a conveyor, etc.) in order to form a workstation group in a factory.

Asynchronous communications in RCS are handled by attaching a "handshake" identifier to every buffer being transferred. The simplest mechanism used involves incrementing the identifier count by one whenever a buffer contains new information. The receiver may then echo the identifier number to confirm receipt of data. This mechanism is also useful in implementing fault detection, isolation, and recovery. If a handshake is not echoed within some reasonable time (e.g., a "time-out" value), this fact can be used in developing fault handling routines.

SECTION 4. RCS PLANS

A ***plan*** is defined as, "a method of action or procedure; a design or scheme of arrangement", in [Ra 82]. Here, we define ***plan*** as a computer information model which specifies tasks or actions to be performed (spatial and/or temporal procedures) and their precedence ordering. ***Rule plans*** are uninstantiated plans (or plan schemas) which can be represented using some form of *If-Then-Else* construct. Rule plans specify branching conditions. ***Path plans*** are an ordered set of instantiated poses, knot points, commands, or other variables specifying a sequential order of execution. Path plans do not specify branching conditions. A rule plan is required to specify conditions to be monitored in order to interrupt the execution of a path plan for branching (i.e., out-of-tolerance condition branching). A path plan can be generated by fully instantiating a rule plan (assuming or given a time sequenced set of input conditions) and storing the result.

An RCS control system can be viewed as an integrated collection of finite state machines which are capable of selecting or generating and executing RCS plans in real-time to cause an electro-mechanical system to perform useful work. This requires a method of representing task knowledge in plans and a set of integration and decomposition standards for distributing those plans within the RCS hierarchy. The RCS Methodology described here uses both rule plans and path plans.

Path plans are normally combined with rule plans in order to instantiate commands to be issued to the next lower RCS level. Rule plans are used to monitor the state of the world to detect out-of-tolerance conditions and path plans generate the next goal point to be achieved by a "move" command assigned to a subordinate RCS level.

Plans may be generated using numerous methods. Plans can be constructed by concatenating pre-stored path plan fragments, plans may be generated by slightly modifying existing plans, plans may be generated by algorithms, or by traversing a decision tree. Alternative plans can be retrieved using an indexing strategy (hypothesis generation) from a plan repository, as in case based reasoning systems, and evaluated according to some value judgment criteria sometimes called a cost-benefit analysis or objective function. In most cases people develop rule plans off-line. Alternatively rule plans could be developed interactively with computer assistance in near-real-time. Path plans are often generated algorithmically in real-time without human input. They may also be generated interactively on or off-line. Path plans are sometimes developed without human interaction off-line. Algorithms used to generate path plans are almost always developed and coded off-line.

4.1. Path Plans

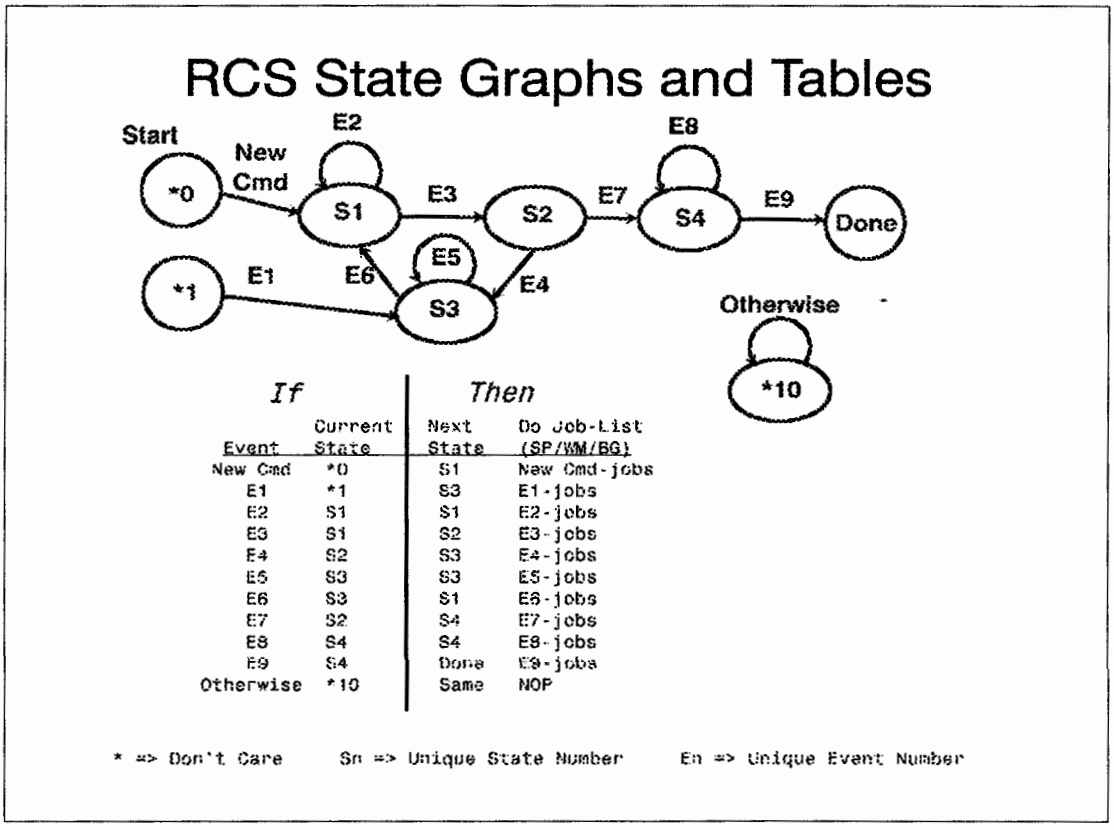
Path plans can be generated by computing a trajectory, by applying some control law, or by generating path "knot points" using a search based algorithm (e.g., A* search) operating within some physical configuration space or in decision space. A path plan might resemble a musical score, in that such a plan would contain a set of pose specifications (position or force, orientation, velocity, etc.) usually occurring at regular intervals of time (a set of interpolated trajectory points). An algorithm might also be used to traverse a "game tree" in order to select a "good" set

of next moves. Path plans (i.e., algorithmically generated plans) are normally found at the Elementary-Move, Primitive, and Servo levels of the RCS Architecture. The knot points specified in a path plan are instantiated goals to be achieved by the finite state machine.

The RCS Methodology described here emphasizes applications which require rule plans (representations of strategy, tactics or process knowledge). This implementation approach is completely compatible with the use of path plans, as well, but we will not cover this topic at the same level of detail in this paper.

4.2. Rule Plans

In the approach discussed here, we use state graphs and state tables to represent RCS rule plans. These plans embody strategies, tactics and process knowledge also called plan schemas. A plan schema is a set of uninstantiated rules for accomplishing some task. Rule plans are used throughout the RCS Architecture. Rule plans are often very simple at the low levels and are often more complex at the higher levels.



If		Then	
Event	Current State	Next State	Do Job-List (SP/WM/BG)
New Cmd	*0	S1	New Cmd-jobs
E1	*1	S3	E1-jobs
E2	S1	S1	E2-jobs
E3	S1	S2	E3-jobs
E4	S2	S3	E4-jobs
E5	S3	S3	E5-jobs
E6	S3	S1	E6-jobs
E7	S2	S4	E7-jobs
E8	S4	S4	E8-jobs
E9	S4	Done	E9-jobs
Otherwise	*10	Same	NOP

* => Don't Care Sn => Unique State Number En => Unique Event Number

Figure 13.

Plans decompose tasks and goals (input commands) into temporal and spatial sequences of subtasks to be performed by subordinate controllers at the next lower level in the hierarchy. Rule plans can be represented as state graphs, as shown in Figure 13. Rule plans specify constraints such as monitoring, threshold, and timing conditions which constitute events or edges in a state

graph. One general plan failure condition that plans may incorporate is a plan time-out monitor. An error status is returned to the higher level supervisory controller whenever a plan failure condition occurs. Plans may specify the conditions for accepting human input as well as states that require and request human input. Thus an edge in a plan might trigger a job list which requests human input and another edge might specify transition conditions that cause the machine to wait (by looping back) at the current state until a human response is received.

A servo loop may also be implemented in a state graph by an edge which loops back on a state node. An edge can specify an *If* test on a sensor signal and a computation to instantiate a command for an incremental actuator movement in response to the detection of an out-of-tolerance condition. Such a servo loop can be used to continuously monitor and null a difference signal according to a control law equation.

Edges or arcs in a state graph indicate both the prerequisites for transition from state to state upon the occurrence of the next state clock cycle and the activities to be performed if those prerequisites occur. Bubbles or nodes in a state graph define and name the allowable states a finite state machine may enter. Normally each edge in a plan will initiate a list of jobs to be accomplished as a result of satisfying the specified transition conditions. Upon initiating these jobs, which are typically task commands to subordinate controllers and edge specific Sensory Processing and/or World Modeling interim calculations, the machine steps to the next state. The edges emanating from the next state are subsequently evaluated on the next control cycle (state clock cycle).

Many real-time control problems can be solved by preplanning responses to the allowable input condition space of a controller. Uninstantiated plan schemas, developed off-line, in the form of state graphs can be stored in the plan repository for each input command allowed and each possible exception condition. Such state graph plans can produce an amazingly complex set of responses to a wide variety of input conditions by incorporating variables which are only instantiated at execution time through sensory feedback. When these plans are executed in a hierarchical RCS control system, each level only instantiates the current step in its plan at any moment in time. As mentioned before, path plans are often combined with rule plans at the lower levels of RCS. Path planners generate and instantiate "move" goal points, in real-time, (specified as variables in the rule plan) and together they instantiate commands for the next lower level subordinate. This produces a control system that can react to exception conditions very quickly while also exhibiting very complex real-time intelligent behavior.

Since all levels of the hierarchy are sampling their input conditions and evaluating their rule plans at the control cycle clock rate, the entire controller is extremely responsive, in a very organized manner, to any changes or transitions at any level of detail throughout the entire system. This response time capability (i.e., to essentially react to any change at any level within one control cycle) is the basis of the "real-time" capability provided by the RCS system. It is the RCS equivalent of event driven real-time context switching (i.e., interrupt handling). This response time concept is as important to achieving real-time behavior as the concepts of layering the hierarchy by orders of magnitude in timing horizons or measuring the length of time each command takes to execute to completion within a controller module.

4.2.1. RCS State Graphs

Figure 13 illustrates how rule plans may be represented in the form of RCS State Graphs and corresponding State Tables. In Figure 13, bubbles (or nodes) are states and arrows are edges, arcs, or transition events. An RCS state graph resembles a PERT chart in that it explicitly represents the precedence of transitions between states. A finite state machine capable of executing an RCS state graph must remember which state it is in, from control cycle to control cycle, and it must evaluate each of the transition conditions specified by the state graph edges, on each cycle. The machine then matches transition conditions to its current state to determine which next state it must transition to on each control cycle. There is an implied (not shown) job list associated with each edge in a state graph. The job list is often not shown in a state graph in order to minimize clutter and to make it easier to draw the graph. A plan should not contain more than roughly ten *If-Then* rules in order to make it easy to understand.

In the example shown, the plan transitions to state S1 on "New Cmd", to S2 on E3, to S4 on E7, and it completes the principal decision sequence to "Done" on E9. A machine executing this plan would follow this sequence if no exception conditions are encountered during execution. The example shows exception conditions and wait or servo loops as follows: E2 loops back on state S1 until E3 happens, E5 loops on state S3 until E6, and E8 loops at state S4 until E9 occurs; In addition an exception loop is shown as moving along E4 from S2 to S3 and E6 from S3 to S1. These examples illustrate that a state graph is not simply a sequential set of steps, rather it is a specification for a plan schema which can handle many combinations and permutations of events over an indefinite time span. The plan schema is, however, completely deterministic in that it specifies all recognized states and transition events.

The state graph in Figure 13 contains an edge labeled "otherwise". This means, if no match is found on a given control cycle, initiate the job list associated with the "otherwise" edge and move to the state pointed to by the edge. Often the "otherwise" condition contains a "no operation (NOP)" instruction or job and the specified next state transition is to stay in the current state (no change). In Figure 13, the "otherwise" edge loops back to state "*10" which illustrates this NOP example. If the "otherwise" condition represents an exception detection (e.g., an error) then a controller executing this plan would issue an error status and it would transition to an exception state in the state graph.

A shorthand notation is used to indicate "don't care" states, they are labeled with an "*" followed by a number in Figure 13. "Don't care" states are not really states at all, they are simply a notation convention. The equivalent state graph for a "don't care" state would be drawn with an edge leading from every state in the graph to the state pointed to by the edge leaving the "don't care" state.

There is a "don't care" state labeled "*0", in Figure 13, with an edge labeled "New Cmd" leading from it into state "S1". This is our notation for the entry point into the plan. Whenever a "new" command is received the finite state machine must immediately initiate the job list associated with the "New Cmd" edge and transition to state "S1" in this example. Receiving a "new" command has the effect of resetting the current plan to the entry point if the plan was already active. In

RCS a handshaking communications protocol employing command serial numbers is used to distinguish between the receipt of a "new" command versus simply receiving the same command buffer (no change) on subsequent control cycles. If the task command and its serial number have not changed, it is considered to be a repeat of the last command and it will be ignored for purposes of plan execution.

4.2.2. RCS State Tables

Figure 13 also shows a state table which is a one-to-one mapping of the state graph shown. The state table is in the form of a list of *If-Then* rules. By mapping the state graph to a state table our representation becomes one step removed from actual software code. The *If-Then* rule table can be implemented using a number of software language constructs such as if-then-else statements or case statements. The table is organized in precedence order for evaluating event conditions (edges). Each event is logically *anded* with the current state in precedence order from the top of the list. A match (logical true condition) causes the plan to transition to the next state and initiate (run) the job list associated with the triggering event. In our approach only the first match found is acted upon during a control cycle. This process is repeated on each control cycle until the plan reaches the "Done" state. After reaching "Done" a NOP is executed thereafter.

In this example we have labeled the event edges and the "don't care" states in precedence order of evaluation. This makes it very easy to code the "don't care" states and to understand that "*0" takes precedence over E1, E2 over E3, and so on. Also notice that the job list associated with each edge in the plan is explicitly shown in the state table representation.

The RCS Executor is the function that evaluates events and interprets transitions in a state table. RCS Job Assignment is the function that interprets and initiates (runs) the job list in a state table.

Appendix C provides examples of a state graph, a state table, and "C" code used to implement them in our, DARPA sponsored, submarine automation demonstration project.

SECTION 5. IMPLEMENTING A CONTROLLER TEMPLATE, THE BASIC RCS BUILDING BLOCK

Building understandable large systems designs requires defining systems integration standards. If we can define a small set of primitive generic building blocks which can be replicated and integrated using a concise set of integration standards, we are likely to have more success in building even more complex systems structures. The ideal situation is to be able to build a complex structure using a single standard type of building block.

The RCS implementation approach described here requires only two basic building blocks: An RCS Controller Template and a Main Program Template. This section introduces the RCS Controller Template concept.

Figure 9 shows a block diagram of a generic RCS controller module. This model complies with the RCS Reference Model Architecture, presented earlier, in that it provides a software execution structure within which the basic functions of Sensory Processing, World Modeling, and Behavior Generation may be implemented. In addition this model begins to address the interface definitions required to integrate a set of controllers to form an RCS hierarchy. The controller must be able to accept task commands from its superior and send commands to its subordinates or to actuators if the controller is at the lowest level of its branch in the RCS hierarchy. It must be capable of accepting status from its subordinates and sending status to its superior. A controller module must be capable of accommodating a human interface and it must have the capacity for communicating with other controllers and the knowledge base through some set of Global Memory communications mechanisms. A controller must also be capable of directly accepting sensor data for processing.

Since a controller module is a finite state machine its response to stimulus is deterministic for any given execution cycle. Its output is only a function of its current state and its input event space. Furthermore its execution time can be measured or calculated for every event-state pair in a given plan. Any algorithm implemented within a controller module must be designed to execute in a cyclic manner, with a definite execution time for each execution cycle. Such an algorithm could, however, be allocated any number of execution cycles in order to produce a solution or a set of more and more optimum solutions on each subsequent execution cycle. An example might be an algorithm that requires a number of sample points in order to converge on a solution within a reasonable margin of error. Such algorithms often produce increasingly accurate results with each new data sample processed.

5.1. Grouping JA, PL, and EX Functions

Figure 14 shows an RCS hierarchy constructed of linked sets of Job Assignment, Planner, and Executor functions. Figure 14 extends the task decomposition diagram (Figure 5), discussed in Section 2, in order to illustrate how the pattern of JA, PL, and EX functions conceptually repeat in an RCS implementation. These patterns roughly delineate the RCS levels and they also denote the opportunities for concurrency within a hierarchy. In practice, when implementing an RCS compliant design, some controller nodes will often emphasize one sub-function over others.

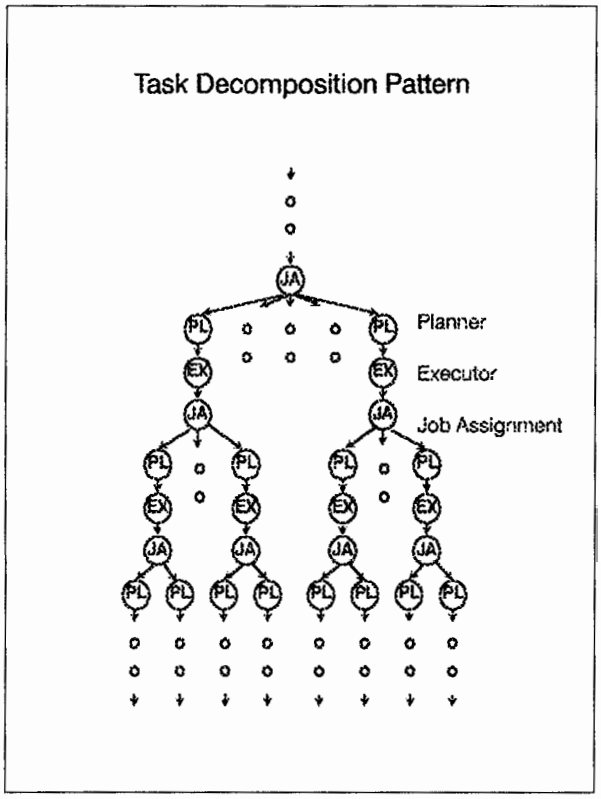


Figure 14.

Obviously, an RCS hierarchy can be developed by implementing each JA, PL, and EX sub-function as a separate software module. Each sub-function would still need some set of SP and WM support functions in order to provide for closed-loop feedback. These could also be implemented as individual software modules. Alternatively, we might group the repeating pattern of JA, PL, and EX functions into a Behavior Generation function and group the BG function with the necessary supporting SP and WM functions. This is attractive if we wish to develop a single controller module template for use as an "integration wrapper". Using this approach we can still implement the sub-functions as separate modules if we wish or they can be developed in any combination which makes sense in a particular controller node instance.

We will consider two possible groupings of the JA, PL, and EX functions necessary to form a Behavior Generation functional group. Both of these groupings contain a repeating pattern of JA, PL, and EX functions. Both are also consistent with the RCS Architecture Reference Model, task decomposition structure discussed earlier (see Section 2).

Figure 15 illustrates these two functional grouping structures side by side. In the first grouping (Grouping#1) the BG function is formed by grouping JA, PL, and EX functions in a pattern exactly like the one presented in the RCS Architecture Reference Model for task decomposition (see Figure 5). Using this grouping pattern, concurrency (assigning tasks to be executed in parallel by subordinates), occurs within a BG function, since JA is responsible for job assignments. This pattern also suggests that a Behavior Generation functional grouping will contain one JA and any number of pairs of PL and EX functions, depending on the number of

concurrent tasks to be assigned to subordinate levels. Robot Systems Division researchers have successfully implemented RCS designs (for the NASA FTS project and some others) containing the two lowest levels of RCS (Servo and Primitive Levels) using this grouping pattern. This grouping structure is not easy to replicate as a template, however, since it may contain an instance dependent number of PL and EX functions for each JA.

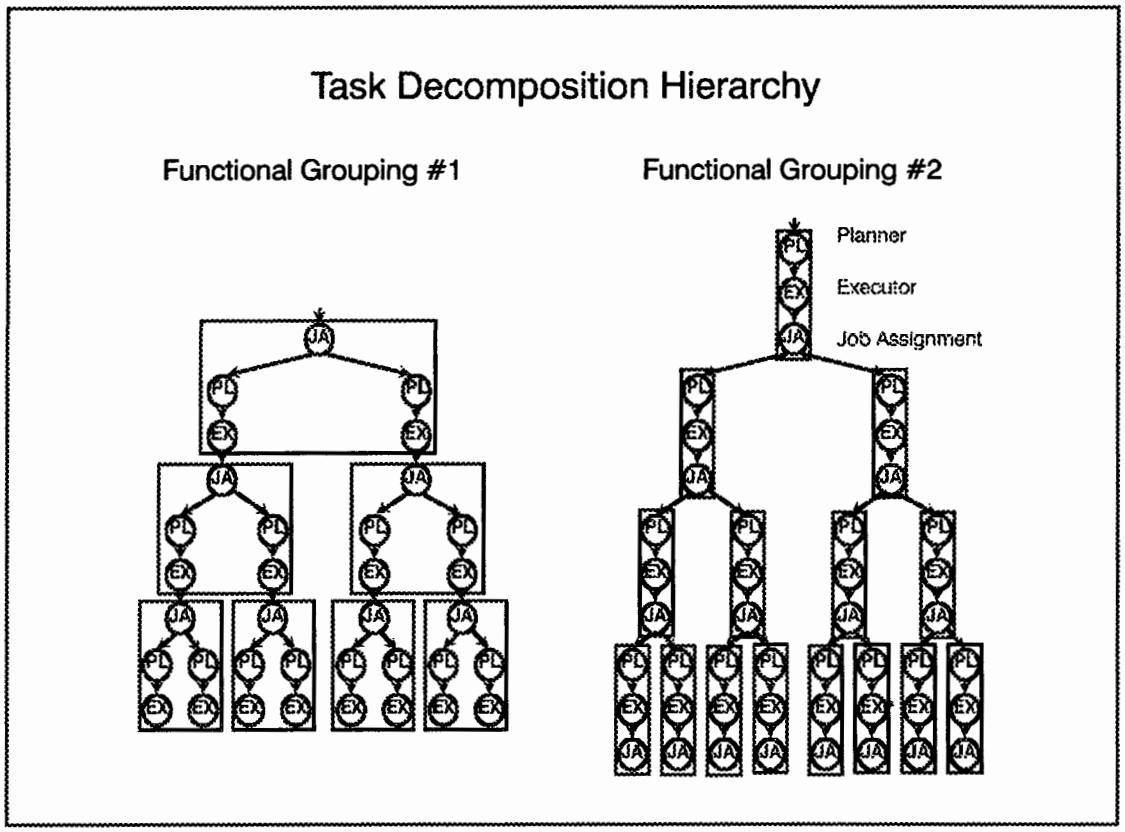


Figure 15.

Grouping#2, in Figure 15, shows an alternate way of grouping the JA, PL, and EX functions. The hierarchy resulting from this second grouping pattern is identical to the one shown for the first grouping pattern, except at the top and bottom of the structure.

At the bottom of Grouping#2 the structure contains a redundant JA function as the leaf node of each tree branch. This JA is redundant because, by definition, its subordinate will always be a single actuator or "black box" device. Since only one subordinate will be directed by the leaf node JAs, we could just as easily issue the commands from the EX function, as in Grouping#1. The top of the Grouping#2 structure begins with a PL and an EX instead of a JA function as in Grouping#1. By developing a controller module that emphasizes the JA function with trivial PL and EX functions an equivalent top node can be implemented. The important point is that functionally identical intelligent control systems applications can be designed using either of these two structures.

The Barbera approach uses Grouping#2 for the following reasons: **1)** the repeating pattern is always the same, (i.e., one PL, one EX, and one JA), **2)** it is a simpler repeating pattern (fewer functions are implemented per module) which results in simpler code, and **3)** Concurrency (assigning tasks to an indefinite number of subordinates to be executed in parallel) is handled at the interface between groupings instead of within a grouping by issuing task commands to subordinates using the GM communications primitives (write and read). These attributes combine to make it relatively straight forward to create a single RCS Controller Module Template for use as an RCS building block.

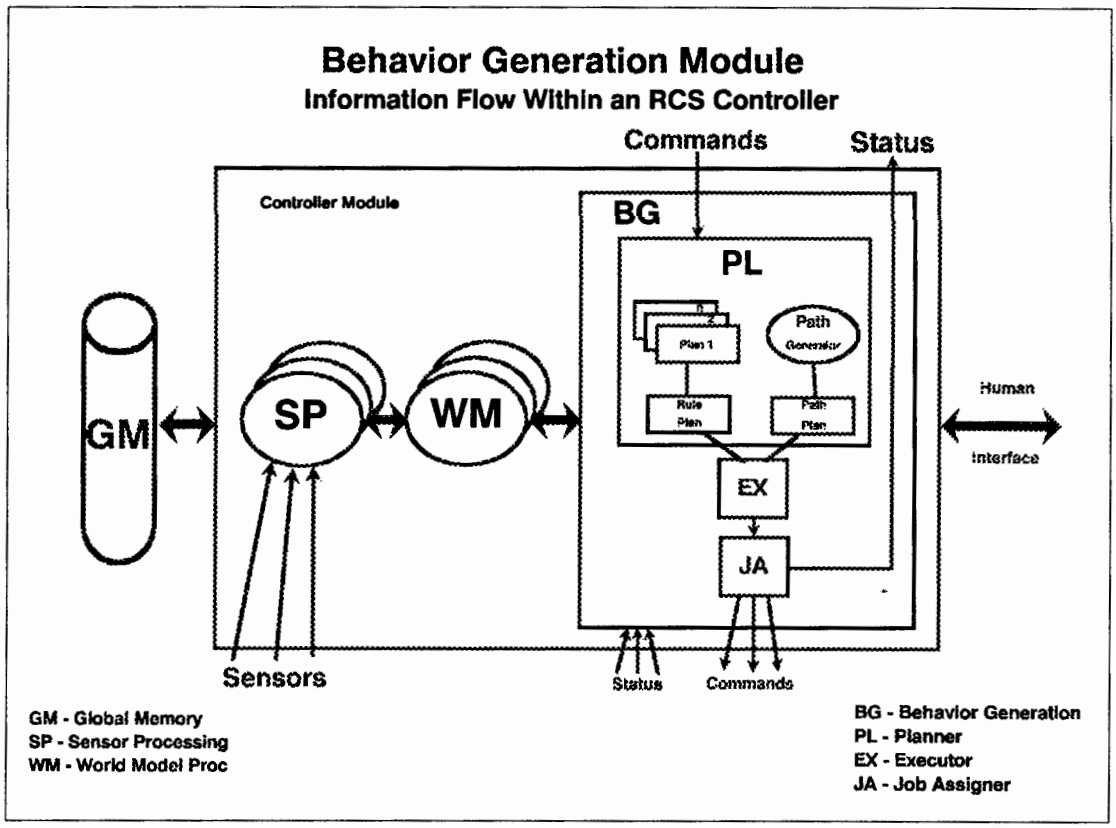


Figure 16.

5.2. BG Decomposition

Figure 16 shows how a Behavior Generation function is further decomposed (within a controller module) using the second grouping pattern option discussed above. The controller communicates using a triple buffering scheme (through Global Memory) with other controllers, its supervisor, its subordinates, its direct sensory inputs, and the human interface (I/F) for this level.

Using this pattern the controller module reads in its next task command, then the Planner runs selecting or generating and passing a rule plan and/or a path plan to the Executor for instantiation. The Executor cyclically executes the plan, stepping once (that is evaluating one set of GM inputs) on each control cycle, and triggering appropriate Job Assignment functions. JA processing results in generating commands for each subordinate as well as the posting of other outputs. In this

implementation approach each RCS controller module may contain at most one BG function and any number of SP and WM functions. The BG function, however, may encapsulate (or retrieve from the knowledge base) any number of plans. At least one plan must exist (or the system must be capable of generating a plan in real-time) for each task command the controller understands and is responsible for acting upon.

5.3. The RCS Controller Template

The RCS Controller Template flow chart diagram in Figure 10 provides another view of the process discussed above. Every controller module must be capable of communicating with other modules in a manner which complies with the RCS Methodology tenets. One way to ensure compliance is to replicate a standard RCS Controller Template incorporating all of the necessary "hooks" for integration. This overhead structure is included in every module to make it easy for humans to integrate modules and to understand and maintain the design. This is done even though some of the overhead may not be needed in every controller module instance. A controller module built from an RCS Controller Template performs Preprocessing, then Behavior Generation (also referred to as Decision Processing), followed by Post-Processing on each control cycle.

5.3.1 Preprocessing within a Controller Template

Preprocessing includes four types of subfunctions: Debug or overhead functions, World Model functions, Sensory Processing functions, and other cyclic functions. These subfunctions are cyclical, meaning they are performed on every control cycle no matter which plan the controller module happens to be executing. On each control cycle Preprocessing reads in all of the input buffers, which includes accepting commands from the next higher level (supervisory controller) and it passes this information to its Behavior Generation (BG) function after some pre-conditioning. The purpose of the pre-conditioning is to process the somewhat raw form of the data in the input buffers into a more symbolic representation for testing in the state tables.

The Debug Preprocessing function is responsible for initializing any performance metrics to be measured on each control cycle. The most basic of these is initializing the counters for measuring execution time performance including the time it takes the controller to execute a given command during the current control cycle as well as the minimum time and maximum time the controller module has taken to execute any of its commands since reinitializing the entire system. Other measures might involve keeping statistics on commands received, sensor reading change rates, etc.

Preprocessing includes World Model functions such as computing predictions, evaluating status reports from subordinates, determining the current operating mode, evaluating priorities, and resources available, etc. These WM functions instantiate variables (i.e., numeric, logical and string variables) and/or arrays of variables that are then passed to and used in Behavior Generation and Post-Processing.

The Sensory Processing functions performed during Preprocessing include algorithms to filter and pre-condition the incoming data stream, convert coordinate systems for data fusion, fuse data,

compare World Model predictions to observed sensor data, and to set detection variables according to established threshold conditions. All of these SP functions result in instantiating variables to be used by Behavior Generation and Post-Processing. Again, the basic operation is to prepare the high level symbolic variables for testing in the state tables.

Other cyclic functions performed during Preprocessing include running any algorithm which must be computed cyclically, independent of the command (or plan) currently being executed. Often path planners are computed cyclically at the lower levels of RCS. Other examples might include simulation algorithms. Algorithms which require large amounts of CPU time can be segregated and implemented in controller modules dedicated to running only one cyclic algorithm. In this case the BG function typically includes rule plans for commands like initialize, run, pause, and perhaps some other debug mode commands. Such controllers may be assigned to execute on separate CPUs.

Preprocessing WM and SP functions might also be responsible for extracting data trends for use and storage as historical traces. This might be thought of as implementing both short term and long term memory. Historical trace variables are subsequently posted to Global Memory during Post-Processing.

The controller module is also responsible for monitoring events which might require sending an error status to its supervisor in response to an emergency condition or an exception condition. Evaluating the world view for this purpose is normally done during Preprocessing. Behavior Generation uses this information to produce error status reports that are then posted during Post-Processing.

5.3.2. Behavior Generation within a Controller Template

The Behavior Generation (BG) function consists of three sub-functions: planning, execution, and job assignment. When the BG function is implemented within a controller module it contains one Job Assignment (JA) function, one Executor (EX) function and one Planner (PL) function.

5.3.2.1. Planner Functions

The Planner may perform rule planning or algorithmic path planning, or both. Rule planning involves selecting options according to a set of rules by applying value judgments. Planners may interact with the World Model to evaluate possible alternative plans against some value judgment criteria. A Planner might hypothesize alternatives by selecting plans from the repository and using the World Model to simulate and predict the performance of the plan based on current conditions and recent historical traces. The result of this planner evaluation process becomes the basis for it to select the best alternative plan for execution.

A Planner may also be implemented so that human input is required in making a plan selection or in generating a new plan. If human interactive planning is done, all human interface I/O must be non-blocking. Requests for human input are posted on each control cycle (along with a handshaking serial number) as well as any data to be displayed. Human responses are read, through Global Memory, in the same fashion.

The most basic Planner simply matches the incoming task command identifier (from the next higher level supervisory module) to the plans pre-stored in its repository. Upon finding a match the Planner selects the plan and passes it to the Executor for instantiation and execution. If no match is found then an error status is posted. In more complex planners, several optional plans might be available in the repository to choose from.

Path planners typically generate plans containing at least two steps into the future and should be limited to generating not more than roughly ten steps. Planners should be designed to operate within a bounded planning horizon. One of the guidelines for designing planners and plans for an RCS system is to limit the number of steps a path plan contains to roughly ten or less as well as the number of branch points (decision points) in a rule plan. New plans generated or selected by a Planner may be loaded into the EX function for immediate execution and/or stored in the plan repository for reuse.

Planners and their supporting WM servers may need to draw upon historical traces stored in Global Memory in order to generate or select future plans. These historical traces should contain data samples covering a time span roughly equal to the future planning horizon for the plans to be generated or used.

There must be at least one plan in the repository for every possible input command or the Planner must be capable of generating a new plan in real-time. The Planner maintains in local memory the identity (and serial number) of the currently active command. This function is performed using a unique command identifier and a serial number to effect a handshaking communication with the next higher level controller.

5.3.2.2. Executor Functions

The Executor (EX) is a reactive planner that is responsible for executing the current step of the current rule plan on each state clock cycle by instantiating variables using inputs from Global Memory (GM) and locally calculated results generated by the SP and WM functions during Preprocessing as discussed above. The EX function is capable of executing plans (state graphs) passed to it by the PL function. The EX function monitors its input event space generating an event vector (using SP and WM calculations) to form a transition trigger event in the current plan which in turn determines its next state. The EX initiates task specific SP and WM jobs as dictated by the current plan whenever it transitions to a new state in a plan. The results of these SP and WM calculations are stored as local interim values within the controller and/or as Global Memory variables for use on subsequent execution cycles or by other controllers. These SP and WM calculations constitute the event constraints and conditions necessary to instantiate and execute the current step in the active plan.

If the Executor is executing a path plan it simply steps to the current knot point in the currently active path plan (passed to it by the Planner), then it steps to the next knot point on the next control cycle, and so on. As the Executor steps through the path plan it triggers the JA function to generate a "move" command to the appropriate subordinate with this knot point as its next goal.

5.3.2.3. Job Assignment Functions

The JA function is responsible for generating outputs on each execution cycle. It must return status to its next higher level supervisor and it forwards commands to each of its subordinates at the next lower level as dictated by the task specific output list associated with each transition edge in the active plan being executed.

A supervisory controller issues a new command with a new serial number to a subordinate under three possible conditions: **1)** when a subordinate reports that it has successfully completed its last assignment and it is cycling, waiting for new input (subordinates are normally capable of completing a ten step plan before their supervisor sends the next command), **2)** upon receiving a "new" command from its supervisor (the next higher level) requiring new tasks to be performed immediately, and **3)** in response to a transition event or error, including emergency conditions, that may cause the supervisory controller to issue a new command before being notified by a subordinate that the current assignment has been completed. Errors detected at this level in the hierarchy or signaled by the status feedback from the next lower level are passed up the chain-of-command so that the supervisor with the decision authority and the appropriate world view can issue new commands to respond to the emergency.

5.3.3. Post-Processing within a Controller Template

Post-Processing may include Sensory Processing functions, World Model functions and Debug functions. All controller modules build output buffers and post them to Global Memory during Post-Processing. They also update local variables/pointers in preparation for the next control cycle (e.g., current plan pointer, current state pointer, etc.). All controllers compute any required performance measures like execution time during Post-Processing. In addition, any cyclic SP or WM functions which need to be computed after Behavior Generation runs are performed during Post-Processing.

It is during Post-Processing that the counter values are used to calculate the execution time of this controller during this cycle, as well as the minimum and maximum length of time the controller has taken to execute to completion while executing any of its commands since the RCS system was reinitialized. Additional Post-Processing routines continuously compare the present execution time of a command to its average, and if the command is taking significantly longer to complete, these routines set status variables that may be used to alert the operator to problems within detailed subtask activities, thereby focusing in on one or two hardware components or objects that might be contributing to the delay. This provides one of the basic capabilities in real-time fault diagnostics.

Other Post-Processing routines, especially in the lower level controllers which are issuing actuator drive signals, act as intelligent watch dogs, checking that an actuator has not been left on too long, or an incorrect combination of actuator commands has not been issued.

5.4. Multi-Tasking on a Shared CPU, using a Main Program Template

Multitasking within a shared CPU is implemented in RCS with a CPU Main Program Template (see Figure 11). The Main Program allows initialization of CPU parameters such as declaring global variables (externals) and the loading of starting values in Global Memory, in preparation for RCS execution. After initialization the Main Program begins running the heartbeat control cycle for the CPU. First a Debug function runs to check for operator inputs indicating a change in operating mode (e.g., debug single step, normal run mode, etc.) and to start a timer to measure control cycle execution time. Once that completes each RCS controller module (including simulation modules) runs in sequence according to the precedence order of the execution schedule established by the programmer. Of course, in compliance with the RCS tenets, all of the controllers must be able to complete their execution within the established heartbeat control cycle time. If the controllers overrun the cycle time then the RCS designer must reassign one or more to different CPUs or increase the cycle time.

At the end of each control cycle the communications controller modules are executed to exchange Global Memory data with other CPUs within the backplane and over any networks being used. The total execution time is calculated as a last step to be sure that the heartbeat control cycle time has not been exceeded (a debug error is posted if it ever does). At that point the Main Program enters a wait loop (or goes to sleep) until time for the next control cycle.

If a random access memory (RAM) board is installed in the backplane, Global Memory variables can be declared as externals and buffered on the RAM board. If this is done and the host operating system supports common memory declarations on a separate RAM board, then communications between controllers on the backplane are handled as common memory communications.

If RCS is running within a real-time operating system environment, a preemptive high priority timer interrupt can be used to wake-up the RCS Main Program for each control cycle. If this technique is used the interrupt system must be turned-off upon entering the Main Program and turned-on again when the Main Program goes to sleep. This technique could be used to allow RCS to coexist with other (lower priority) applications within a conventional real-time interrupt driven system.

If a particular RCS controller module requires significantly longer execution time (e.g., a module doing complex planning), the preferred solution is to move it to a CPU with sufficient excess computing time to accommodate it. If necessary this CPU may run with a longer heartbeat cycle. Algorithms which are compute intensive and are atomic algorithms (i.e., they don't easily decompose any further) are typically encapsulated in their own RCS controller module whose only job is to cyclically execute the algorithm and post results in Global Memory. Such a controller might accept only a single task command, ***run*** (in addition to the typical debug commands like initialize, pause, calibrate, etc.). This technique of segregating compute "hogs" is very effective in preserving the responsiveness of the overall control system while still accommodating complex, compute intensive, algorithms.

5.5. Handshaking

The RCS approach discussed here relies only on a handshaking mechanism to synchronize task command and status communication between modules. We don't rely on the synchronous nature of the "heartbeat" control cycle for this purpose. This handshaking mechanism is also applied to all data shared (through Global Memory) between CPUs. This is done to allow the designer the freedom to reassign controller modules to any CPU in the architecture without being concerned about synchronizing the control cycles of every CPU. In fact, by using a handshaking protocol, we can easily establish different control cycle rates for each CPU, if required. This also means that the RCS design will actually run completely asynchronously if we choose not to impose a synchronous heartbeat on each CPU (allowing every CPU to execute all of its assigned modules as fast as possible with no control cycle "wait" mechanism).

A simple mechanism for implementing the RCS handshaking protocol is to increment a counter in each controller module each time a new command is to be issued to a subordinate. In this way, the count is used as a serial number which when appended to a command identifier (or any data buffer) serves to uniquely identify each instance of the message type. This allows us to determine when a command received is a "new" command as opposed to simply a repeat of the last command received. If the serial number has changed it is a new command, if not it isn't. The subordinate controller module echoes the serial number of the command it is working on in its status reports. This is useful in matching completion status or error reports to the command being executed.

If a heartbeat control cycle is being used a cycle count value can be incremented by the Main Program to be used as a time-stamp, for time-out purposes, since the count will reflect the number of control cycles which have passed and the heartbeat control cycle time is known. This value is appended to every command and data buffer in addition to the handshake serial number. Remember that the serial number only increments on issuing a new command or data buffer while the control cycle value increments on every control cycle to keep track of the passage of time. This can be very useful for implementing fault detection and isolation logic. Hardware failures can be easily detected by using time-out monitors to alert the system that a particular controller is no longer communicating. This type of mechanism is often called a "watchdog timer". This technique can also be used to trigger the automatic or semi-automatic reconfiguration of resources given the detection of a fault condition.

The watchdog timer capability can be implemented as follows. The cycle count value is incremented each control cycle and placed in Global Memory by the Main Program running on each CPU. This is the value each controller uses to recognize the relative control cycle time on any CPU it is communicating with. The control cycle values on different CPUs don't have to be synchronized as long as the normal buffer update rate is known. The update rate must take into account any delays imposed due to the physical communications links being used for a given command or data buffer. The cycle count value can be appended to each controller's output buffers as a form of time-stamp of when those buffers were last communicated. The controllers that read these buffers can compare the present clock cycle count value to the value appended to the last data buffer received. If the difference gets too large (as compared to the known nominal

update rate), it indicates that the data buffer has not been updated for some time, either because of a problem with the sending controller or a communication link problem. Every controller can test for this as part of a Preprocessing function, and watch for any significant delays occurring in receipt of data. In this way, every controller has the capacity to watch other controllers they communicate with, and they can immediately report delays they detect. This is another real-time diagnostic attribute easily supported by the structure of the RCS Methodology.

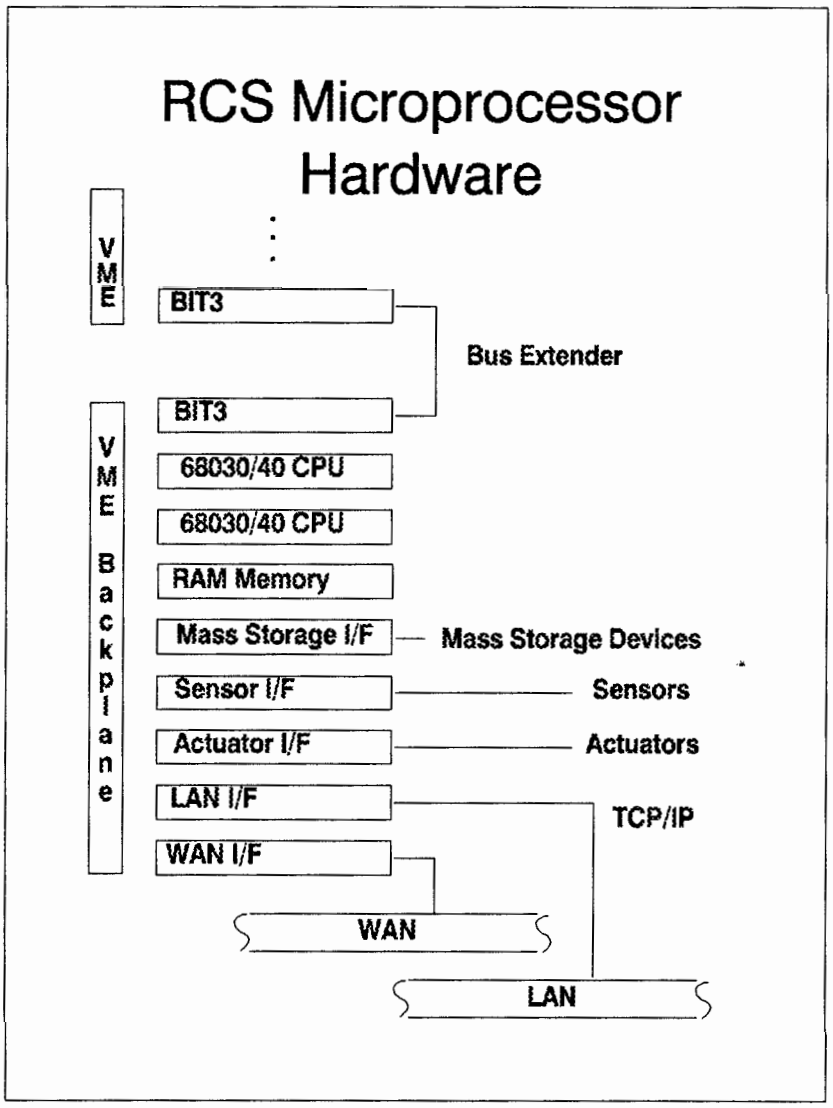


Figure 17.

5.6. RCS Target Hardware

Figure 17 illustrates a typical suite of microprocessor hardware for implementing an RCS control system. The block diagram shows a VME backplane multiprocessor system. Any multiprocessor backplane hardware suite may be used (e.g., Multibus, Nubus, etc.). Bus extenders, such as BIT3, can be used to expand a backplane, adding additional card slots. Many standard commercial boards are available today for such systems at very reasonable prices. Such boards

include: CPUs, RAM memory, interface cards for standard computer peripheral devices (e.g., disc drives, CRTs, mouse, keyboard, printers, plotters, etc.), graphics boards, video boards, analog to digital (A/D) and D/A boards, as well as local area and wide area network interface boards. Software operating systems and device drivers are also widely available for these systems. Using this type of hardware suite in an RCS implementation makes it very easy to tailor the hardware selection and the communications network to meet the real-time requirements of the application. It is also very easy to extend such systems as the system evolves. In addition, if the RCS software is written using standard software language compilers such as "C", we can improve the prospects for portability of the code to newer generations of hardware.

5.7. Required Operating System Services

An RCS implementation generally runs in a host operating system (OS) environment on a given hardware platform. RCS requires the following operating system services and capabilities:

1) ***OS Kernel*** - The operating system should provide basic kernel services like scheduling a designated user application program after boot-up (power on). The kernel should provide interfaces to standard device drivers (discs, keyboards, mouse, CRT screen, printer, plotter, etc.) for application programs.

2) ***Memory-Locking*** - the host operating system must allow a program or sub-program to be defined as memory-locked. Locking a program in RAM memory guarantees its existence and eliminates the need to roll programs into high speed RAM memory from a mass storage device (a disc drive) at execution time. This allows programs to run with very fast response times. The ability to spawn copies of programs during operation is of little or no use in real-time applications, and is not required.

3) ***Timers*** - the host operating system should support timing facilities for maintaining wall-clock time and a calendar. High speed timer precision, at the microsecond level, is required for computing various performance measures previously discussed. Private timers for use locally within a program are also useful.

4) ***Multiprocessor*** - The OS should support a multiprocessor environment, allowing programs to be memory-locked on multiple CPUs in a shared backplane. The OS must also support communications mechanisms such as common memory management for the multiprocessor environment.

5) ***Communications*** - In addition to common memory management the OS should support standard interprocessor communications such as mail boxes, datagrams, sockets, etc., as well as standard local area network (LAN) and wide area network (WAN) communications mechanisms (e.g., TCP/IP and Ethernet).

6) ***File Management*** - The OS should provide file management services for a mass storage device (e.g., disc storage).

In addition to the services listed above, the following ***real-time operating system*** services are useful but not required when implementing an RCS control system:

1) ***Interrupts*** - Real-time context switching driven by interrupt handling. The ability to turn off interrupt handling within a program is required.

2) ***Synchronization*** - Synchronization tools like semaphores, signals, clocked synchronization and priority queues.

3) ***Preemptive scheduling*** - The ability to preempt lower priority processing to allow higher priority tasks to run immediately with guaranteed response time.

Real-time applications must respond to the physical environment within some set of time limits. This means that real-time applications (like RCS) must be able to guarantee that they exist in the computing environment (memory-locking) at all critical times and that they can respond to environmental stimuli within some "worst case" time limit. Generally this means that the RCS application must be able to preempt and/or take over complete control of the computing platform during real-time operation. Single user operating systems such as DOS can meet these requirements as well as most real-time operating systems. Unix operating systems and other non-real-time, time sharing, multitasking, operating systems, on the other hand, usually don't allow an application (e.g., RCS) to take over control, therefore only operating systems with real-time (RT) extensions (e.g., RT-Unix, Lynx OS, VxWorks, etc.) should be considered as host environments.

SECTION 6. RCS METHODOLOGY DEVELOPMENT STEPS

The method steps listed in Table 1 begin with the definition of at least one conceptual design. Such a conceptual solution would propose the use of some set of people, machines, actuators, sensors, and an information structure to be organized into an integrated system, for operation within one or more physical environments and aimed at addressing an overall highest level system goal (see Figure 1). These RCS Method steps are intended to provide a procedural and systematic approach to developing and implementing practical intelligent machine systems which would typically perform some physical work in a given environment (e.g., operate a vehicle, produce parts, perform construction or mining tasks, operate a sensor or weapon system, etc.). If several competing conceptual designs exist, the method could be used to evaluate the feasibility and expected performance of each design, as part of the engineering analysis and selection process. The method might also be used to analyze the potential benefits and trade-offs which might result from selecting among different system components within a particular conceptual design (e.g. selecting different types or sets of sensors and actuators or integrating "black box" off-the-shelf subsystems).

The methodology described in this paper should be interpreted as an iterative, "rapid prototyping", real-time software development method. The steps listed in Table 1 are roughly in the sequential order of a first pass through the method to achieve a skeleton of the overall RCS architecture to be implemented. Once a skeleton is developed the developer(s) should iterate within the steps to develop executable controller modules in a bottom-up process. As these controller modules are developed, as executable code, their behavior can be studied and their performance measured to further refine the control hierarchy. This process should involve revisiting and revising the original problem description and requirements as well as the organizational structure and definition of the RCS controller modules.

As the running system evolves it should be used as a tool to enhance the dialogue with the domain experts and sponsors. By demonstrating the evolving system, developers, experts and customers are better able to refine requirements and explicitly capture the domain expert's knowledge.

Table 1. Summary of the RCS Methodology Steps

1) CONCEPT DEVELOPMENT
   A) Gather domain knowledge.
      a) Interview domain experts.
      b) Define processes, object and workspace geometry, machine and load kinematics, dynamics, and coordinate systems.
   B) Develop the problem description / scenario
      a) Define the goals and requirements of the control system.
      b) Identify physical devices (actuators and sensors) to be controlled and the human interfaces required.
      c) Illustrate the important spatial and geometric dimensions of the problem domain as well as the their coordinate frames.
      d) Define control system performance constraints.
      e) Describe the temporal span of control.
         1. What minimum reaction time is required?
         2. What is the maximum sampling rate required?
         3. What will the longest planning horizon be?
   C) Conceptualize the application in terms of:
      - the RCS Controller Hierarchy,
      - the Operator I/F System,
      - the Data Management System and
      - the Communications Management System.
      a) Perform task decomposition engineering analysis of the RCS Controller Hierarchy
         1. Define tasks, attributes, parameters, constraints, and procedures (scripts or plans), including timing and synchronization at each hierarchical level.
         2. Estimate the timing horizons for each level.
      b) Perform Human I/F engineering analysis
      c) Perform Data Management System engineering analysis
         1. Define the set of physical objects to be acted upon and list their relevant attributes.
         2. Represent the objects and their attributes (including geometry) in world model database structures.
         3. Define a set of maps that represent the space in which objects reside, and in which tasks on those objects will be performed. Define appropriate scale, resolution, and coordinate systems for the maps at each hierarchical level.
         4. Estimate the size of the data structures required and the update and retrieval rates expected.
      d) Perform Communications System engineering analysis
         1. Define communications protocols, syntax, and semantics of messages between control modules.
         2. Estimate the size of the messages required and the frequency of message traffic expected.

2) DESIGN THE RCS HIERARCHY USING TASK DECOMPOSITION

A) Develop a task tree (hierarchical decomposition of tasks).

B) Choose a "thread" of tasks which span the task tree from the highest node to the bottom of the tree.

C) Design the controller hierarchy by iteratively adding task threads in a rapid prototyping fashion.

a) Define the set of agents (people, sensors and actuators) that will perform the tasks.

b) Organize and name a set of controllers in a hierarchy according to their respective levels of authority and responsibility (supervisors and subordinates) for executing tasks through the task agents.

c) Define command verbs for each controller by mapping the task tree to the organizational hierarchy of controllers.

D) Design controller software by adding design detail using generic RCS templates in a bottom-up process.

a) Design state diagrams (plans) or scripts for each command verb.

b) Identify inputs

1. Command inputs.
2. Operator inputs
3. Status inputs.
4. World Model inputs (global parameters).
5. Sensor inputs.

c) Define Sensor Processing algorithms and other input preprocessing required.

d) Define World Model processing required.

e) Define post processing and outputs required (commands, messages and data to be posted in the global memory).

f) Identify status reports required.

g) Identify parameters computed which should be included (posted) in Global Memory.

h) Identify command verbs to be output to next lower level (subordinates).

3) CODING AND TESTING RCS SOFTWARE

A) Incrementally develop code for each controller using generic RCS coding templates in a bottom-up fashion.

B) Incrementally develop simulators to drive each controller in a closed-loop fashion.

C) Incrementally develop simulators for the human interfaces required.

D) Measure the performance of each controller in terms of execution time.

E) Map the controller modules (software processes) onto the computer hardware (processors).

a) Measure the communications latency of messages flowing between hardware devices (inter-board, intra-board, LAN and WAN communications).

b) Map processes onto processors by grouping closely coordinated modules.

4) PORT THE RCS SOFTWARE TO THE TARGET HARDWARE SYSTEM.

5) INCREMENTALLY INTEGRATE AND TEST THE RCS CONTROLLERS WITH THE ROBOTIC SYSTEM'S SENSORS AND ACTUATORS.

A) Perform lab tests.
B) Perform field tests.

6) DEVELOP A SIMULATOR TO ANIMATE THE ROBOTIC SYSTEM IN THE ENVISIONED PHYSICAL ENVIRONMENT (WORKSPACE).

7) DESIGN, CODE AND TEST THE OPERATOR I/F SYSTEM, DATA MANAGEMENT SYSTEM AND THE COMMUNICATIONS MANAGEMENT SYSTEM

8) INTEGRATE THE RCS CONTROLLER HIERARCHY WITH THE OPERATOR I/F SYSTEM, DATA MANAGEMENT SYSTEM AND THE COMMUNICATIONS MANAGEMENT SYSTEM.

9) PRODUCE FINAL DOCUMENTATION FOR THE SYSTEM VERSION OR RELEASE.

10) ITERATE ALL OF THE STEPS ABOVE EXTENDING THE RCS SYSTEM, IN A "RAPID PROTOTYPING" FASHION, BY ADDING NEW CONTROLLERS AND/OR PROCESSING MODULES TO EXECUTE ADDITIONAL TASK THREADS.

SECTION 7. CONCLUDING REMARKS

When discussing the RCS Method with people familiar with emerging software engineering practices, we are often asked, "is RCS object oriented?" and "is RCS a functional decomposition?". We will attempt to partially answer those questions here.

7.1. Is RCS Object Oriented?

The approach taken here can be viewed as an object oriented method, as defined by Coad and Yourdon [Co 91], in that the objects used to drive the design (in a bottom-up fashion) are the sensors, actuators, and the controller modules. The processes embedded within each controller can be viewed as encapsulated methods. Further, controller objects can inherit execution and communications methods from a generic controller template. In fact, several Robot Systems Division researchers have experimented, achieving some success, with using object oriented techniques to implement RCS designs. Using the Barbera RCS approach, the organization of the controller module objects, for message passing and processing, is driven by the task decomposition method (rather than data flow analysis) using an iterative top-down and bottom-up design approach.

In implementing RCS a task tree is created that defines the set of command verbs each respective controller object (nouns) must be able to execute. The services performed by the controller objects include processing their input buffers on each control cycle and writing to output buffers (commands, status and Global Memory variable updates) according to processing rules which can be represented using state diagrams. This differs slightly from the typical object oriented approach in that this decomposition is derived and analyzed by emphasizing control flow rather than data flow analysis. Of course, both approaches result in specifying data flow and control flow, only the decomposition technique emphasis is different.

It appears that the sensor processing object hierarchy needed to support the RCS development method could also benefit from an object oriented analysis and design approach. Sensor processing is concerned with answering the, "what is the state of the world?", question. The ability to answer this question demands an organized representation of all of the objects which are important to the goals of the robotic system. An object oriented approach addresses this representation by forming a taxonomy of objects in successive layers of abstraction (see Figure 1). Such a taxonomy can be represented very efficiently by using the notion of inheritance. In a machine vision example, at the bottom of the tree are pixels and point sources of sensor information, the next layer is features, then surfaces, then objects, then groups of objects, etc. Object oriented methods are very good at defining software processes that encapsulate knowledge about these types of Sensor Processing and World Model objects.

The World Model clearly must bridge the gap between a sensory processing (object oriented) view of the world and the task decomposition (control flow) view of the actions to be performed. It seems clear that entity relationship diagrams and object oriented taxonomies should be investigated as a means of representing a priori information to be stored in the World Model as

well as any Global Memory data which must be stored as a historical trace (post priori, short term and long term memory).

If one attempts to build object oriented taxonomies for the Global Memory databases, one cautionary note is that in RCS, this knowledge must be distributed across the hierarchy of controllers and may not wind up residing in one central database. This might frustrate an attempt to combine traditional object oriented database design with RCS methods.

7.2. Is RCS a Functional Decomposition or a Structured Analysis Method?

We consider RCS to be a special case of functional decomposition. It is also very highly organized and modular as is typical of structured methods.

The decomposition rules for RCS design concentrate on ***task*** functions (physical actions to be performed) rather than functions in general. More abstract functions such as Sensory Processing, World Modeling, and Behavior Generation are distributed across the entire hierarchy of controllers. Within each RCS controller module SP, WM, and BG functions are further decomposed. In addition the sub-decomposition of SP, WM, and BG functions is accomplished using a generic, replicating, template approach. In a more traditional functional decomposition approach, SP, WM, and BG functions would typically be implemented as single large modules. There are many other examples of abstract functions (e.g., safety, situation assessment, attention focusing, execution monitoring, etc.) which are distributed by layers of abstraction and according to timing or decision making horizons, in RCS. These functions might otherwise be implemented as single large modules, with very complex inter and intra module communications, if we applied traditional functional decomposition methods.

7.3. Conclusions

In this paper we have attempted to begin to define a consistent set of systems engineering rules for building, evolving, and maintaining large, complex, intelligent control systems. We are particularly motivated by the need to make the systems engineering development process more compatible with the human thought process. The modularity guidelines presented recognize the limited natural capacity of the human mind to understand and deal with concurrent tasks in a complex real-time system. Our approach has been:

1) To develop an RCS Methodology based on the RCS Architecture Reference Model developed by Albus, Barbera, and others over the last two decades.
2) To use task scenarios in the knowledge engineering process in order to capitalize on the human associative memory capacity.
3) We have emphasized hierarchical organization as a powerful method of complexity management.
4) We have selected cyclic sampling and the finite state machine as our execution model in order to ensure our designs are deterministic and verifiable.
5) We have emphasized rule plan knowledge models (state graphs and state tables) which are compatible with and can be directly executed by finite state machines.

6) We have described a primitive communications mechanism (triple buffering and Global Memory) which is compatible with cyclic sampling and provides for non-blocking I/O. We have also described how other communications mechanisms can be implemented on top of this primitive mechanism using communications controller modules.
7) We have defined generic RCS Controller Module Templates and the RCS Main Program as our basic systems integration wrapper mechanism to simplify the development and integration process.
8) We have presented an outline of a set of rapid prototyping steps which can be used as a systems development life cycle approach.

There are many areas we haven't discussed in detail. Some of the important topics not fully covered include:

1) How to develop the Sensory Processing algorithms required to answer the "what is?" question.
2) How to develop the World Model "what is?" and "what if?" processing algorithms required.
3) What methods to use and how to develop algorithms to handle the processing of uncertain information (during Preprocessing) in order to recognize a transition event.
4) How to implement a complete communications network.
5) How to develop the human I/F system.
6) How to develop a database management system and populate it with information models as well as data.
7) What information models to use for the object taxonomy, object geometry, track files, and map files.
8) How to develop the simulation algorithms necessary.
9) How to develop the animation system.
10) How to deal with a shared physical resource. How to restructure the hierarchy between control cycles in order to move a sub-tree from one supervisor to another in real-time.
11) How to develop very high speed servo-loops for algorithmically complex control system applications.
12) How to develop path planners and path planning algorithms.

These topics are not addressed in detail for two reasons. Because the objective of this paper is to define how to organize a real-time control system architecture which can accommodate (provide a home for) all of these functions while efficiently handling the problems of: managing complexity; extensibility; resource contention; conflict resolution; verifiable, robust, deterministic, measurable performance; closed-loop control; etc.; and because we still don't have all the answers.

The NIST Robot Systems Division is currently conducting a long term research program, called the Intelligent Machines Initiative, which is focusing on Sensory Processing and World Modeling for machine vision in particular as well as many of the other issues not addressed in detail here.

SECTION 8. REFERENCES

[Al 91a] J.S. Albus, R. Quintero, R. Lumia, M. Herman, R.D. Kilmer, K.R. Goodwin, *A Reference Model Architecture for ARTICS*, ASME and IIE, Manufacturing Review Volume 4, Number 3, September 1991.

[Al 91b] J.S. Albus, *Outline for a Theory of Intelligence*, IEEE Journal, Transactions on Systems, Man and Cybernetics, Volume 21, Number 3, May/June 1991.

[Al 90a] J.S. Albus, *The Role of World Modeling and Value Judgment in Perception*, Proceedings of the Fifth IEEE International Symposium on Intelligent Control, Philadelphia, PA., September 5-7, 1990.

[Al 90b] J.S. Albus, *Hierarchical Interaction Between Sensory Processing and World Modeling in Intelligent Systems*, Proceedings of the Fifth IEEE International Symposium on Intelligent Control, Philadelphia, PA., September 5-7, 1990.

[Al 89a] J.S. Albus, H.G. McCain, and R. Lumia, *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, NIST (formerly NBS) Technical Note 1235, April 1989 Edition.

[Al 89b] J.S. Albus, R. Quintero, H. Huang, M. Roche, *Mining Automation Real-Time Control System Architecture Reference Model (MASREM)*, NIST Technical Note 1261 Volume 1, May 1989.

[Al 88] J.S. Albus, *System Description and Design Architecture for Multiple Autonomous Undersea Vehicles Project*, NIST Technical Note 1251, September 1988, p. 126.

[Al 81] J.S. Albus, *Brains, Behavior and Robotics*, BYTE/McGraw- Hill, Petersborough, NH, 1981.

[Ås 90] K. J. Åström, and B. Wittenmark, Computer Controlled Systems Theory and Design, Prentice Hall, NJ, 1990.

[Ba 84] A.J. Barbera, J.S. Albus, M.L. Fitzgerald, and L.S. Haynes, *RCS: The NBS Real-Time Control System*, Robots 8 Conference and Exposition, Detroit, MI, June 1984.

[Co 91] Peter Coad and Edward Yourdon, *Object Oriented Analysis*, YOURDON Press, Englewood, NJ, 1991.

[Fi 90a] J. Fiala, *Note on NASREM Implementation*, NISTIR 89-4215, March 90.

[Fi 90b] J. Fiala, R. Lumia, *An Approach to Telerobot Computing Architecture*, NISTIR 4357, June 90.

[Hu 92] H. Huang, J. Horst, R. Quintero, *A Motion Control Algorithm for a Continuous Mining Machine Based on a Hierarchical Real-Time Control System Design Methodology*, Journal of Intelligent and Robotic Systems 5: 79-99, 1992, Kluwer Academic Publishers, Dordrecht, The Netherlands.

[Hu 91] H. Huang, R. Quintero, J.S. Albus, *A Reference Model, Design Approach, and Development Illustration toward Hierarchical Real-Time Control System for Coal Mining Operations*, Control and Dynamic Systems: Advances in Theory and Applications Volume 46: Manufacturing and Automation Systems: Techniques and Technologies, Part 2 of 5, , Edited by C. T. Leondes, Academic Press,1991.

[Ku 67] Benjamin C. Kuo, *Automatic Control Systems*, Second Edition, Prentice-Hall, Electrical Engineering Series, 1967.

[Mc 86] H. G. McCain, et al., *A Hierarchically Controlled Autonomous Robot for Heavy Payload Military Field Applications*, Proceedings of the International Conference on Intelligent Autonomous Systems, Amsterdam, The Netherlands, December 8-11,1986.

[Mc 82] C. McLean, H. Bloom, T. Hopp, *The Virtual Manufacturing Cell*, IFAC/IFIP Conference on Information Control Problems in Manufacturing, Gaithersburg, MD., Oct., 1982.

[Mi 56] George A. Miller, *The Magical Number Seven , Plus or Minus Two: Some Limits on Our Capacity for Processing Information"*, The Psychological Review, 63, pp. 71-97, 1956.

[OD 91] J. O'Donnell, B. Lee, *Flight Telerobotic Servicer Task Analysis Methodology*, report prepared for Goddard Space Flight Center, by Ocean Space Systems under contract NAS5-30897, 2 April 1991.

[Ra 82] Random House College Dictionary, 1982, Revised Edition

[Si 83] J.A. Simpson, R.J. Hocken, and J.S. Albus, *The Automated Manufacturing Research Facility of the National Bureau of Standards*, Journal of Manufacturing Systems, Vol.1, No. 1, pg. 17, 1983.

[Sz 88] S. Szabo, H. A. Scott, R. D. Kilmer, *Control System Architecture for the TEAM Program*, Proceedings of the Second International Symposium on Robotics and Manufacturing Research, Education and Applications, Albuquerque, NM, November 16-18, 1988.

Appendix - A. Controller Module Template, Example C Language Code

```
/*
* Example of Generic Controller Module for RCS on a PC Compatible in
* Microsoft C v6.00
* This is the Template used for the RCS Software Demonstration
* The following prefixes are used:
      PR - this module
      TB - subordinate to this module
      SM - supervisor of this module

* The following are non-standard C conventions used in this source
  code:

  #define ST_BGN            if(
  #define THEN      ){
  #define ST          }else if(
  #define DEFAULT     }else {
  #define ST_END            }

  #define CASE_BGN      if(
  #define CASE        }else if(
  #define CASE_END    }

  #define COPY_BUFFER(x,y,z)  (memcpy((char *)(x),(char *)(y),(z)))

  dprintf( ) is a specialized routine for writing to the VGA display
             rapidly.

* The template may be easily extended by adding plans and Sensory
* Processing & World Modeling functions
* Search for !!! for places where the plan in Appendix C can be added
*/

/*
* Include Files
*/
#include "global.h"     /* global definitions */

/*
* Definitions and Macros
*/
#define PR_DEBUG_LINE               5   /* line used for debug screen */

/*
* Private Global Variables
*/
/* Start Time - Used for Performance Metrics */
static unsigned                     pr_cycle_start_time;

/* Initialize Current State */
static int                          pr_cur_state = S0;

/* Declare Local Copies of Interface Buffers */
static PR_BUFFER                    pr;     /* command from above */

/* Commands to and Status from Subordinates, TB */
static TB_COMMAND                   tb_co;
static TB_STATUS                    tb_si;

/*
* World Model Data Local Copies
*/
static enum OPER_MODE               w_op_mode;

/*
* Function Prototypes - Listed by Category
*/

/* Primary Controller Module Routine, calls all others */
void pr_controller();

/* Behavior Generation - Planning, Execution, & Job Assignment */
static void pr_decision_process     (void);
static void pr_check_if_new_command (void);

/* Sensory Processing and World Modelling */
static void pr_pre_process          (void);
static void pr_post_process         (void);

/* Debug Functions for Display */
static void pr_print_in_data        (void);
static void pr_print_out_data       (void);

/* Commands which may be received - State table plans */
static void pr_init                 (void);
static void pr_halt                 (void);
/* !!! Declare more commands here !!! */

/*QQ*********************************************************************
**
RTNS:  None.
PRPSE: External Routine used by main to execute this task.  This is the
       only entrance to this task. Note that no parameters are passed
       explicitly in the function call. All parameter passing is done
       in the pre_process by copying the interface buffers.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  Ron Hira & Hui-Min Huang 11/20/92
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************
*/
void pr_controller()
{
/*  READ START TIME, COPY IN INTERFACING BUFFERS AND CONTROL MODEL */
/*  CHECK SUBORDINATE STATUS                                       */
       pr_pre_process();

/*  EXECUTE COMMON FUNCTIONS SENSORY PROCESSING & WORLD MODELLING  */
       /* Function calls made for SP & WM */


/*  CHECK IF NEW COMMAND                                           */
       pr_check_if_new_command();

/*  BG/JA/PL/EX -SELECT STATE TABLE AND EXECUTE                    */
       pr_decision_process();

/*  EXECUTE COMMON FUNCTIONS SENSORY PROCESSING & WORLD MODELLING  */
       /* Function calls made for SP & WM */

/*  COPY OUT INTERFACING BUFFERS & CONTROL MODEL, DISPLAY DATA &   */
/*  DEBUG, CALCULATE PERFORMANCE                                   */
       pr_post_process();
}

/*QQ*********************************************************************
**
```

```
RTNS:  None.
PRPSE: This routine's function is to plan the control via a specific
         state table based on the command in as well as other inputs and
         sensor data Behavior Generation.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**************************************************************************
*/
static void pr_decision_process(void)
{
/* DEBUG - for running modes */
        /*
        * If run flag in not set return
        */
        if (pr.modei.dont_run == TRUE)
        {
                return;
        }

        /*
        * If single step flag is set but single step number has not changed
        *     return
        */
        if (
        (pr.modei.single_step == TRUE) &&
        (pr.modei.single_step_num == pr.perfo.single_step_num))
        {
                return;
        }

        /*
        * If single step flag is set and single step number has changed
        *     cycle once.
        */
        if (
        (pr.modei.single_step == TRUE) &&
        (pr.modei.single_step_num != pr.perfo.single_step_num))
        {
                pr.perfo.single_step_num = pr.modei.single_step_num;
        }


/* BEHAVIOR GENERATION STATE TABLE - PLANNING, EXECUTION, AND JOB ASSIGNMENT */
        /*
         * Execute the proper command
         */
        ST_BGN
                pr.ci.command == PR_INIT
                THEN
                        dprintf(PR_DEBUG_LINE,8,"init     ");
                        pr_init();
        ST
                pr.ci.command == PR_HALT
                THEN
                        dprintf(PR_DEBUG_LINE,8,"halt     ");
                        pr_halt();

/* !!! Commands may be added here in the same format as above  !!! */
        DEFAULT
                dprintf(PR_DEBUG_LINE,8,"INVALID   ");
        ST_END
}

/*QQ**********************************************************************
**
RTNS:  None.
PRPSE: This routine handles all of the preprocessing this level.  It
        includes:  Reading the global interface buffers,
        Checking executing status of children;
ATHR:  Hui-Min Huang
CRTD:  11/21/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**************************************************************************
*/
static void pr_pre_process(void)
{
/* DEBUG - Display */
        dprintf(PR_DEBUG_LINE,1,"PR");

/* Performance */
        /*
        * Starting time for PR controller module
        */
        pr_cycle_start_time = W->timer_counter;

/* Copying Interfacing Buffers */
        /*
        * Get incoming commands from superior module.
        */
        COPY_BUFFER(&pr,&(G->pr_buf),sizeof(PR_BUFFER));

        /*
        * Get status back from subordinates (Turbine - TB)
        */
        COPY_BUFFER(&tb_si,&(G->tb_buf.so),sizeof(TB_STATUS));
        COPY_BUFFER(&tb_co,&(G->tb_buf.ci),sizeof(TB_COMMAND));

        /*
        * Copy in world model data to local copies
        */
        COPY_BUFFER(&w_op_mode,&(W->pr_w_op_mode),sizeof(enum OPER_MODE));

/* DEBUG  */
        /*
        * If the subordinate status back echo number does not
        * match its command number, assume the subordinate is still executing.
        */
        if (tb_si.status_num != tb_co.command_num)
        {
                tb_si.status = TB_EXECUTING;
        }


        /*
        * Debug Display - Execute print_in_data
        */
        pr_print_in_data();
}


/*QQ**********************************************************************
```

```
**
RTNS:  None.
PRPSE: This is the state table (plan) for the INIT command of this level.
ATHR:  Hui-Min Huang
CRTD:  10/14/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************
*/
static void pr_init(void)
{
        /* Plan for INIT command */
        ST_BGN
                new_command     /* conditions */
                THEN
                        /* Set current state of PR control */
                        pr_cur_state = S1;
                        /* Command and command number to subordinate TB */
                        tb_co.command = TB_INIT;
                        tb_co.command_num ++;
        ST
                pr_cur_state == S1      &&
                tb_si.status == TB_DONE
                THEN
                        pr_cur_state = NOP;
                        pr.so.status = PR_DONE;
        DEFAULT
        ST_END
}

/*QQ*************************************************************************
**
RTNS:  None.
PRPSE: This is the state table (plan) for the HALT command of this level.
ATHR:  Hui-Min Huang
CRTD:  10/14/91
MDFD:  commented out other actuator stuff 12/31/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************
*/
static void pr_halt(void)
{
        /* Plan for HALT command */
        ST_BGN
                new_command
                THEN
                        pr_cur_state = S1;
                        tb_co.command = TB_HALT;
        ST
                pr_cur_state == S1          &&
                tb_si.status == TB_DONE
                THEN
                        pr_cur_state = NOP;
                        pr.so.status = PR_DONE;
        DEFAULT
        ST_END
}

/* !!! Extra Plans may be added here, in similar format as above. !!! */
```

```
/*QQ*************************************************************************
**
RTNS:  None.
PRPSE: This routine handles the writing out of all interface buffers as
       well as any other required Sensory Processing & World Modeling.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************
*/
static void pr_post_process(void)
{
/* Copy Interface Buffers */
        /*
        * Write status back to superior module contained in the PR_BUFFER
        */
        COPY_BUFFER(&(G->pr_buf.so),&pr.so,sizeof(PR_STATUS));
        COPY_BUFFER(&(G->pr_buf.perfo),&pr.perfo,sizeof(PR_PERFORMANCE));

        /*
        * Write commands to subordinate TB
        */
        COPY_BUFFER(&(G->tb_buf.ci),&tb_co,sizeof(TB_COMMAND));

        /*
        * Write world data
        */
        COPY_BUFFER(&(W->pr_w_op_mode),&w_op_mode,sizeof(enum OPER_MODE));

/* DEBUG */
        /*
        * Calculate performance metrics for module execution
        */
        pr.perfo.last_cycle_time = (W->timer_counter-pr_cycle_start_time) * 21;

        if (pr.perfo.min_cycle_time == 0)
        {
                pr.perfo.min_cycle_time = 0xFFFF;
        }
        if (pr.perfo.last_cycle_time > pr.perfo.max_cycle_time)
        {
                pr.perfo.max_cycle_time = pr.perfo.last_cycle_time;
        }
        if (pr.perfo.last_cycle_time < pr.perfo.min_cycle_time)
        {
                pr.perfo.min_cycle_time = pr.perfo.last_cycle_time;
        }

        /*
        * Display print out data
        */
        pr_print_out_data();
}

/*QQ*************************************************************************
**
RTNS:  None.
PRPSE: This routine checks to see if the command receieved from above is
        a new command.  It checks this by comparing the current command
```

```
        in number with the last command in number it received (stored
        in status out number). If they are different, it is assumed that
        it is a new command.
ATHR:  David G. Quigley
CRTD:  10/09/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**************************************************************************
*/
static void pr_check_if_new_command(void)
{
        if (pr.ci.command_num != pr.so.status_num)
        {
                dprintf(PR_DEBUG_LINE,5,"NC");
                pr.so.status_num = pr.ci.command_num;
                new_command = TRUE;
                pr_cur_state = S0;
                pr.so.status = PR_EXECUTING;
        }
        else
        {
                new_command = FALSE;
                dprintf(PR_DEBUG_LINE,5,"  ");
        }
}


/*QQ************************************************************************
**
RTNS:  None.
PRPSE: This routine prints the command and command number received this
        cycle to the screen if the Debug for this level is active.
ATHR:  Hui-Min Huang
CRTD:  11/21/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**************************************************************************
*/
static void pr_print_in_data(void)
{
        dprintf(PR_DEBUG_LINE,35,"%5.5u",pr.ci.command_num);
        dprintf(PR_DEBUG_LINE,30,"%4.4d",pr.ci.command);

        dprintf(PR_DEBUG_LINE+20,1,"PR");
        if (pr.modei.dont_run == TRUE)
        {
                dprintf(PR_DEBUG_LINE+20,10,"STOP");
        }
        else
        {
           dprintf(PR_DEBUG_LINE+20,10,"RUN ");
        }

        if (pr.modei.single_step == TRUE)
        {
                dprintf(PR_DEBUG_LINE+20,16,"SINGLE");
        }
        else
        {
                dprintf(PR_DEBUG_LINE+20,16,"AUTO  ");
        }

        if (pr.modei.simulate == TRUE)
        {
                dprintf(PR_DEBUG_LINE+20,23,"SIMU");
        }
        else
        {
           dprintf(PR_DEBUG_LINE+20,23,"REAL");
        }
}

/*QQ************************************************************************
**
RTNS:  None.
PRPSE: This routine prints the status, status number, and state matched
        for this cycle to the screen if the Debug for this level is active.
ATHR:  Hui-Min Huang
CRTD:  11/21/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
**************************************************************************
*/
static void pr_print_out_data(void)
{
        dprintf(PR_DEBUG_LINE,48,"%5.5u",pr.so.status_num);
        dprintf(PR_DEBUG_LINE,43,"%4.4d",pr.so.status);
        if (pr_cur_state == NOP)
        {
                dprintf(PR_DEBUG_LINE,55,"NOP  ",pr_cur_state);
        }
        else
        {
                dprintf(PR_DEBUG_LINE,55,"S%d  ",pr_cur_state);
        }
        if ((pr.modei.dont_run == TRUE) ||
                (pr.modei.single_step == TRUE))
        {
                dprintf(PR_DEBUG_LINE+20,35,"
        }
        else
        {
                dprintf(PR_DEBUG_LINE+20,35,"%5.5u  %5.5u  %5.5u",

pr.perfo.last_cycle_time,

pr.perfo.min_cycle_time,

pr.perfo.max_cycle_time);
        }
}
```

```
/*
* PR buffer definitions for the Generic Controller Template
*/
enum pr_commands
{
        PR_INIT = 2100,
        PR_HALT,
/* !!! Extra commands (plans) may be defined here !!! */
};

enum pr_responses
{
        PR_NOT_READY = 0,
        PR_EXECUTING,
        PR_DONE,
        PR_ERROR,
};

/*
* Module buffer interface structures:
*/
typedef struct
{
    unsigned                command_num;
    enum pr_commands        command;
/* !!! Command parameters may be added here, e.g., ship_speed !!! */
} PR_COMMAND;

typedef struct
{
     unsigned                status_num;
     enum pr_responses       status;
     unsigned                error_num;
} PR_STATUS;


typedef struct
{                              /* pr mode structure */
     boolean                 dont_run;
     boolean                 single_step;
     boolean                 simulate;
     unsigned                single_step_num;
} PR_MODE;

typedef struct
{                              /* pr performance */
     unsigned                last_cycle_time;
     unsigned                min_cycle_time;
     unsigned                max_cycle_time;
     unsigned                single_step_num;
} PR_PERFORMANCE;


typedef struct
{                              /* pr command-response buffer */
     PR_COMMAND              ci;
     PR_STATUS               so;
     PR_MODE                 modei;
     PR_PERFORMANCE          perfo;
/* Sensors and Actuators may be added here */
} PR_BUFFER;
```

Appendix - B. Main Program Template, Example C Language Code

```
/*
* Main.C - Example Main Program Template for RCS to Run on a PC Compatible
           in Microsoft C v6.00
* This is the Template used for RCS Software Demonstrations

* The following are non-standard C conventions used in this source
  code:

  #define COPY_BUFFER(x,y,z)  (memcpy((char *)(x),(char *)(y),(z)))

  qprintf( ) is a specialized routine for writing to the VGA display
             rapidly.

Controllers with prefixes, 'sup', are modules which allow human
interaction via decision aiding.

For clarity, module execution and some important variables from the
submarine demonstration are included.

*/

/*
* Include Files
*/
#include "global.h"                 /* global definitions for GM variables  */

/*
* Definitions and Macros
*/

/*
* Private Global Variables
*/
static int              single_step     = FALSE;
static int              paused          = FALSE;
float                   delta_time;
float                   start_time;
static unsigned         end_time;
static CC1_COMMAND      cc1_co;         /* command to below     */
static CC1_STATUS       cc1_si;         /* response from below  */


/*
* Public Global Variables
*/
GLOBAL_DATA *G;         /* Interface Buffers */
WORLD_DATA  *W;         /* World Model       */
SIM_DATA    *S;         /* Simulation Model  */

/*
* Function Prototypes
*/
void main_exit_rcs                  (void);
void main_transmit_init             (void);
void main_transmit_halt             (void);
void main_single_step               (void);
void main_pause                     (void);
void main_go                        (void);

void main_pre_process               (void);
void main_post_process              (void);
void main_init_global               (void);
void main_set_default_world_data    (void);
void main_world_data_print          (void);
```

```
/*QQ*********************************************************************
RTNS:  None.
PRPSE: This is the main routine.  Handles the multitasking, initialization
       of the system, and system restoration upon completion.
ATHR:  Ron Hira
CRTD:  12/16/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*********************************************************************QQ*/
int main()
{

/* Allocate Global Memory, World Model, and Interface Buffering Space: */
        if ((G = (GLOBAL_DATA *) malloc(sizeof(GLOBAL_DATA))) == NULL)
        {
                fprintf(stderr, "*** Couldn't allocate global memory.\n");
                exit(1);
        }
        if ((W = (WORLD_DATA *) malloc(sizeof(WORLD_DATA))) == NULL)
        {
                fprintf(stderr, "*** Couldn't allocate world memory.\n");
                exit(1);
        }

        if ((S = (SIM_DATA *) malloc(sizeof(SIM_DATA))) == NULL)
        {
                fprintf(stderr, "*** Couldn't allocate sim data memory.\n");
                exit(1);
        }


/* Initialization of Timer Interrupts, Keyboard, Display, Global
   Memory */

        /* This function zeros all of the global buffer information. */
        main_init_global();

        /*This function initializes the level and command arrays by loading
        *  them in from the files LEVEL.DAT and COMMAND.DAT respectively.
        */
        kbd_init_arrays();

        /* This function replaces the current timer interrupt (0x08) routine
        *       with the rcs timer interrupt routine.
        */
        timer_initialize_interrupt();

        /* Blanks out the Display */
        draw_screen();

        /* This function transmits the command init to the process CC1.
         *  CC1 is the top level module
         */
        main_transmit_init();

        /* Write_comm_buffers */
        main_post_process();

        /* Part of user interface for debugging */
        kbd_clear_previous_commands();
```

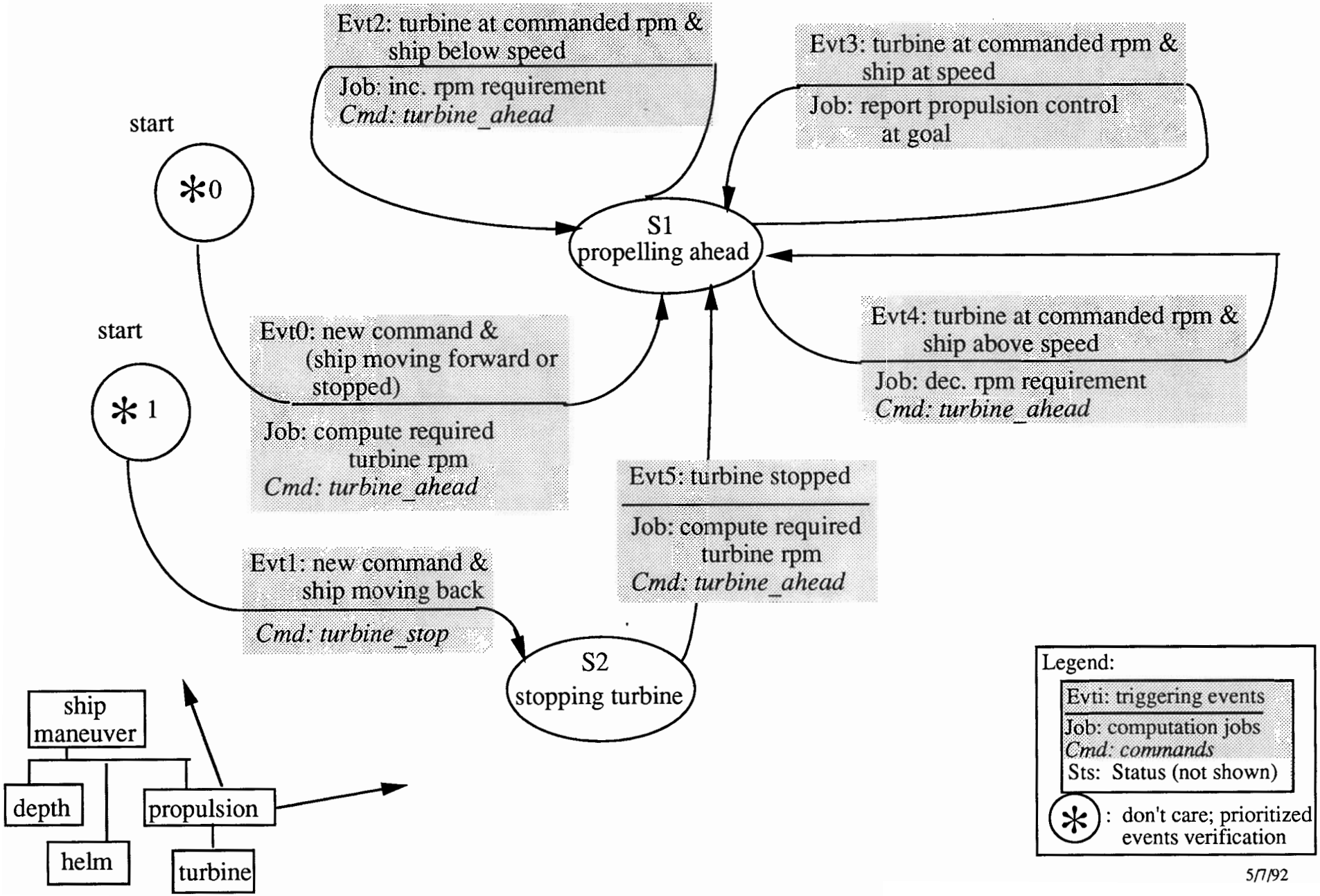
```
/*
* Set the default values for the World Model
*/
main_set_default_world_data();

/*
* Main event loop - Endless While Loop
*/
while (1)
{
/* Read Cycle Start Time */
        start_time = W->timer_counter;

/* Scan Keyboard for Inputs and Evaluate Command */
        while (kbhit()) kbd_evaluate_board_input();

/* Breakpoint for Source Level Debugger */
        if (W->main_enter_debug)
        {
                W->main_enter_debug = FALSE;  /* set breakpoint here */
        }

/* Check Operating Mode - Single Step or Free Running */
        if (!paused || single_step)
        {
                single_step = FALSE;

                /*
                * Run the Ship Simulators, including cyclic execution
                * of actuator simulators and ship dynamics
                */
                sh_controller();

                /*
                * Run the Individual Controllers in a Top Down Order
                */
                cc1_controller();
                sm_controller();
                dp_controller();
                hl_controller();
                pr_controller();
                dr_controller();
                ha_controller();
                tr_controller();
                bl_controller();
                sp_controller();
                sl_controller();
                li_controller();
                pu_controller();
                bv_controller();
                hv_controller();
                rd_controller();
                tb_controller();

                /*
                * Execute Decision Aiding Modules
                */
                sup_dp_controller();
                sup_sm_controller();

                /*
                * Display the data
                */
                main_world_data_print();

                /*
                * Read and write the bit3 interface - For communication
                  with the SGI workstation
                */
                read_write_bit3();

        }

        /* Calculate Performance for Last Execution Cycle */
        delta_time = ((float)(W->timer_counter) - start_time) *.211;

        /* Wait for the specified 30 msec execution cycle time */
        while(timer_check_for_timer_pulse() == 0);
    }
}

/*QQ*********************************************************************
RTNS:  Returns to dos an exit code of 0.
PRPSE: This routine restore the system to its state before the start of
            the program.  Also clears the screen.
ATHR:  Nat Frampton
CRTD:  12/16/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*********************************************************************/
void main_exit_rcs()
{
        timer_restore_interrupt();

        _setbkcolor(0);
        _clearscreen(_GCLEARSCREEN);
        _displaycursor(_GCURSORON);
        exit(0);
}

/*QQ*********************************************************************
RTNS:  None.
PRPSE: This function transmits the command init to the process CC1.
        CC1 is the top level module
ATHR:  Hui-Min Huang
CRTD:  12/01/91
MDFD:  None.
NOTES: Main_post_process is called to load the command into the global
              memory.
PRBLM: None.
OTHER: None.
*********************************************************************/
void main_transmit_init()
{
        main_pre_process();
        single_step = TRUE;
        cc1_co.command = CC1_INIT;
        cc1_co.command_num++;
        S->sh_buf.ci.command_num = 0;
        S->sh_buf.ci.command = SH_INIT;
        S->sh_buf.ci.command_num++;
        main_post_process();
}

/*QQ*********************************************************************
RTNS:  None.
```

```
PRPSE: This function transmit the command halt to the process CC1.
ATHR:  Hui-Min Huang
CRTD:  12/01/91
MDFD:  None.
NOTES: Main_post_process is called to load the command into the global
         memory.
PRBLM: None.
OTHER: None.
*****************************************************************************/
void main_transmit_halt()
{
        main_pre_process();
        single_step = TRUE;
        cc1_co.command = CC1_HALT;
        cc1_co.command_num++;
        main_post_process();
}



/*QQ*************************************************************************
RTNS:  None.
PRPSE: This command stops execution of the tasks by setting the paused
                flag to true.  It also prints a red "[paused]" to the screen.
ATHR:  David G. Quigley
CRTD:  10/07/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************/
void main_pause()
{
        paused = TRUE;
        qprintf(0,70,RED,WHITE,TRUE,"[PAUSED] ");
}


/*QQ*************************************************************************
RTNS:  None.
PRPSE: This function pauses the execution of the tasks and sets the
                single_step flag to true allowing one pass through the levels.
ATHR:  Ron Hira
CRTD:  12/15/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************/
void main_single_step()
{
        paused = TRUE;
        qprintf(0,70,RED,WHITE,TRUE,"[PAUSED] ");
        single_step = TRUE;
}

/*QQ*************************************************************************
RTNS:  None.
PRPSE: This function sets the paused flag to FALSE, effectively turning
                on the free running of the modules.
ATHR:  Ron Hira
CRTD:  12/16/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************/
void main_go()
{
        paused = FALSE;
        qprintf(0,70,GREEN,WHITE,FALSE,"[RUNNING]");
}



/*QQ*************************************************************************
RTNS:  None.
PRPSE: This routine handles all of the preprocessing this level.  It
         includes:  Reading the interface buffers,
                Checking executing status of children;
ATHR:  Nat Frampton
CRTD:  12/16/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************/
void main_pre_process()
{
        /*
        * read_comm_buffers
        */
        COPY_BUFFER(&cc1_si,&(G->cc1_buf.so),sizeof(CC1_STATUS));
        COPY_BUFFER(&cc1_co,&(G->cc1_buf.ci),sizeof(CC1_COMMAND));

        /*
        * If the echo number does not match the command number, set status to
        * EXECUTING
        */
        if (cc1_si.status_num != cc1_co.command_num)
        {
                cc1_si.status = CC1_EXECUTING;
        }
}

/*QQ*************************************************************************
RTNS:  None.
PRPSE: This routine handles the writing of all global data as well as
         any other required post processing.
ATHR:  Hui-Min Huang
CRTD:  12/20/91
MDFD:  None
NOTES: None.
PRBLM: None.
OTHER: None.
*****************************************************************************/
void main_post_process()
{
        /* write_comm_buffers   */
        COPY_BUFFER(&(G->cc1_buf.ci),&cc1_co,sizeof(CC1_COMMAND));
}


/*QQ*************************************************************************
RTNS:  None.
PRPSE: This function zeros all of the global buffer information.
ATHR:  Ron Hira
CRTD:  12/16/91
```

```
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*************************************************************************/
void main_init_global(void)
{
        char far *tmp;
        int  i;

        tmp = (char far *) G;
        for (i=0;i<sizeof(GLOBAL_DATA);i++)
                *tmp++ = 0;

        tmp = (char far *) S;
        for (i=0;i<sizeof(SIM_DATA);i++)
                *tmp++ = 0;

        tmp = (char far *) W;
        for (i=0;i<sizeof(WORLD_DATA);i++)
                *tmp++ = 0;
}

/*QQ*********************************************************************
RTNS:  None
PRPSE: This routine prints out the World Data contents at the top of
        the screens. It's used for the debug.  It calls qprintf.
ATHR:  Ron Hira
CRTD:  12/31/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*************************************************************************/
void main_world_data_print (void)
{
        qprintf(0,0,BLACK,WHITE,FALSE,"HEADING: %6.4f",W->hl_w_ship_heading);
        qprintf(1,0,BLACK,WHITE,FALSE,"BUBBLE : %6.4f",W->dr_w_ship_bubble_angle);
        qprintf(2,0,BLACK,WHITE,FALSE,"DEPTH  : %6.4f",W->hv_w_ship_depth);
        qprintf(0,20,BLACK,WHITE,FALSE,"SPEED  : %6.4f",W->pr_w_ship_speed);
        qprintf(1,20,BLACK,WHITE,FALSE,"X_POS  : %6.4f",W->hl_w_x_pos);
        qprintf(2,20,BLACK,WHITE,FALSE,"Y_POS  : %6.4f",W->hl_w_y_pos);
        qprintf(0,40,BLACK,WHITE,FALSE,"STERN  : %6.4f",W->sp_w_stern_plane_angle);
        qprintf(1,40,BLACK,WHITE,FALSE,"SAIL   : %6.4f",W->sl_w_sail_plane_angle);
        qprintf(2,40,BLACK,WHITE,FALSE,"RUDDER : %6.4f",W->rd_w_rudder_angle);
/*      qprintf(2,60,BLACK,WHITE,FALSE,"CMAC_SG : %6.4f",W->sgi_w_suggested_heading);
        qprintf(1,60,BLACK,WHITE,FALSE,"SM_SG   : %6.4f",W->sgi_w_fathometer);
*/      qprintf(0,60,BLACK,WHITE,FALSE,"TIME(ms): %6.4f",delta_time);
}

/*QQ*********************************************************************
RTNS:  None
PRPSE: This is the initial settings of the World Model. The values may be
        adjusted for various "what if?" type scnarios.
ATHR:  Ron Hira
CRTD:  12/31/91
MDFD:  None.
NOTES: None.
PRBLM: None.
OTHER: None.
*************************************************************************/
void main_set_default_world_data (void)
{
        W->timer_counter                    = 0;
        W->sim_timer_counter                = 0;
        W->sim_times_real                   = 1;
        W->main_enter_debug                 = FALSE;
        W->supervisor_control               = FALSE;
        W->ccl_w_num_tracks                 = 0;


        S->sh_sim_w_sim_bubble_angle        = 0.0;
        S->sh_sim_w_sim_ship_last_speed     = 0.0;
        S->sh_sim_w_sim_sea_pressure        = 50.0 * 0.003967;
        S->sh_sim_w_sim_vertical_vel        = 0.0;
        S->sh_sim_w_sim_density             = 1.00;

        W->dp_w_bottom_depth                = 0.0;
        W->dr_w_ship_bubble_angle           = 0.0;
        W->sup_dp_sgi_message.message_num       = 1;
        W->sup_sm_sgi_message.message_num       = 1;

}
```

Appendix - C. Example RCS Plan (State Graph, State Table, and C Language Code)

PROPULSION CONTROL: AHEAD, AT DESIRED SPEED



5/7/92

Propulsion Ahead Plan State Table

IF		THEN		
			Do Job-List (SP/WM/BG)	
Event	Current State	Next State	Computation	Command/Status
E0: New Cmd & (Ship Forward || Stopped)	*0	S1	WM/BG: Compute Required Turbine RPM	TB-Turbine Ahead
E1: New Cmd & Ship Backward	*1	S2		TB-Turbine Stop
E2: Turbine at Commanded RPM & Ship below Commanded Speed	S1	S1	WM/BG: Increase RPM Requirement	TB-Turbine Ahead
E3: Turbine at Commanded RPM & Ship at Commanded Speed	S1	S1		Report Propulsion Control at Goal
E4: Turbine at Commanded RPM & Ship above Commanded Speed	S1	S1	WM/BG: Decrease RPM Requirement	TB-Turbine Ahead
E5: Turbine Stopped	S2	S1	WM/BG: Compute Required Turbine RPM	TB-Turbine Ahead

Appendix C: A Propulsion Ahead State Table in C

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

PURPOSE: This is the state table for the AHEAD command of this level.

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

static void pr_ahead(void)
{
        ST_BGN
                new_command &&
                (ship_dir == MOVING_AHEAD || ship_dir == STOPPED)
                THEN
                        pr_cur_state = S1;
                        calc_fwd_prop_speed();
                        tb_co.command = TB_AHEAD;
                        tb_co.command_num ++;
                        tb_co.rpm = prop_speed;
        ST
                new_command &&
                ship_dir == MOVING_BACK
                THEN
                        pr_cur_state = S2;
                        tb_co.command = TB_STOP;
                        tb_co.command_num ++;
        ST
                pr_cur_state == S2      &&
                tb_si.status == TB_DONE
                THEN
                        pr_cur_state = S1;
                        calc_fwd_prop_speed();
                        tb_co.command = TB_AHEAD;
                        tb_co.command_num ++;
                        tb_co.rpm = prop_speed;
        ST
                pr_cur_state == S1      &&
                tb_si.status == TB_DONE     &&
                sub_speed_status == BELOW_SPEED
                THEN
                        ++ prop_speed;
                        tb_co.command = TB_AHEAD;
                        tb_co.command_num ++;
                        tb_co.rpm = prop_speed;
        ST
                pr_cur_state == S1      &&
                tb_si.status == TB_DONE     &&
                sub_speed_status == ABOVE_SPEED
                THEN
                        -- prop_speed;
                        tb_co.command = TB_AHEAD;
                        tb_co.command_num ++;
                        tb_co.rpm = prop_speed;
```

```
        ST
                pr_cur_state == S1      &&
                tb_si.status == TB_DONE     &&
                sub_speed_status == AT_SPEED
                THEN
                        pr.so.status = PR_AT_GOAL;
        DEFAULT
        ST_END
}
```