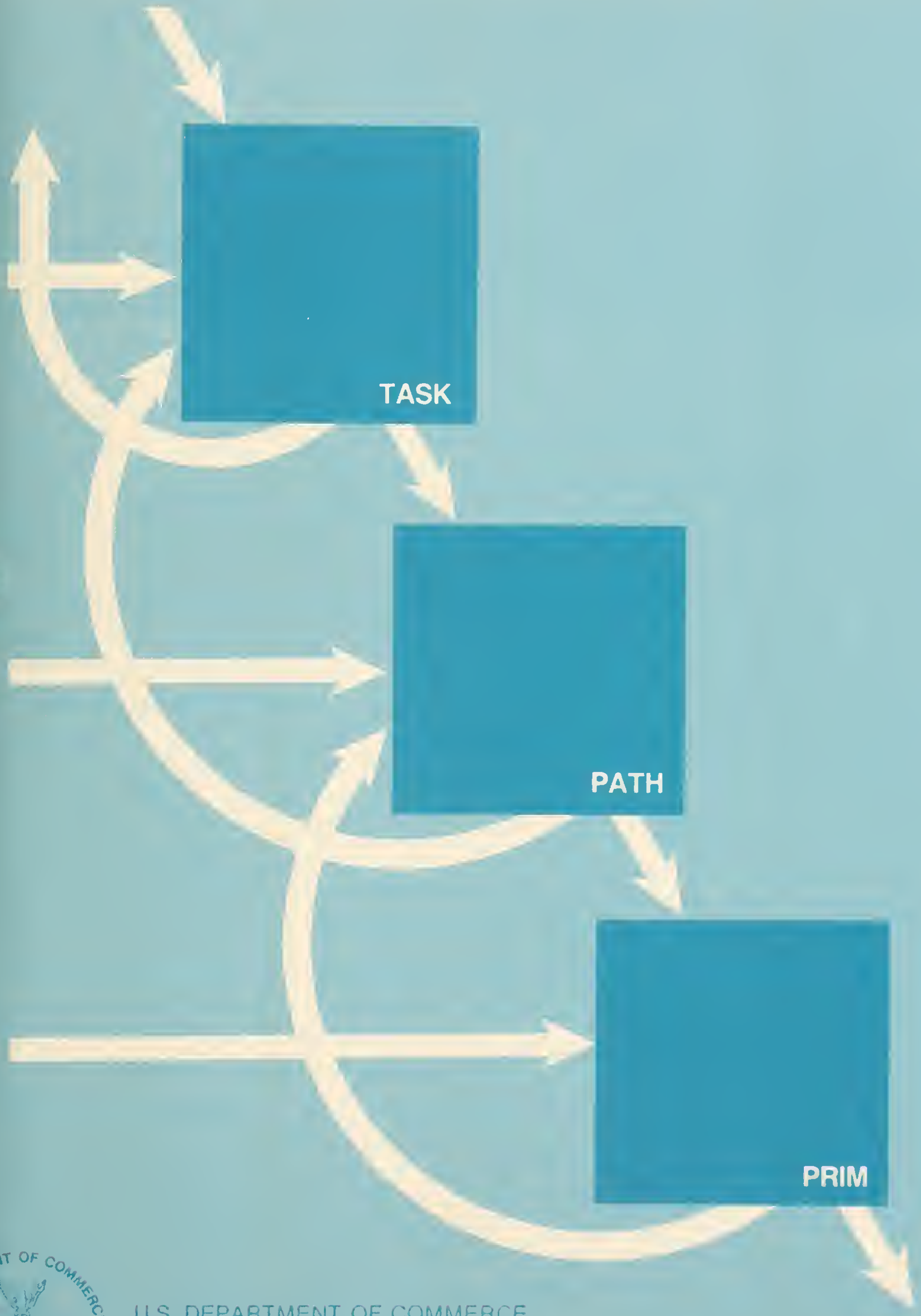
NIST TECHNICAL NOTE 1250
(formerly National Bureau of Standards)

The NBS Real-time Control System



User's Reference Manual

Version 2.2c

OCTOBER 1988

Stephen A. Leake
and
Roger D. Kilmer

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
(formerly National Bureau of Standards)
Robot Systems Division

NIST Technical Note 1250

The NBS Real-time Control System User's Reference Manual

Stephen A. Leake and Roger D. Kilmer

Center for Manufacturing Engineering
National Engineering Laboratory
National Institute of Standards and Technology
(formerly National Bureau of Standards)
Gaithersburg, MD 20899

NOTE: As of 23 August 1988, the National Bureau of Standards (NBS) became the National Institute of Standards and Technology (NIST) when President Reagan signed into law the Omnibus Trade and Competitiveness Act.

October 1988

U.S. Department of Commerce
C. William Verity, Secretary

National Institute of Standards and Technology
(formerly National Bureau of Standards)
Ernest Ambler, Director

National Institute of Standards
and Technology
Technical Note 1250
Tech. Note 1250
381 pages (Oct. 1988)
CODEN: NBTNAE

U.S. Government Printing Office
Washington: 1988

For sale by the Superintendent
of Documents,
U.S. Government Printing Office,
Washington, DC 20402

ERGO® is a registered trademark of Micro-Term, Inc.

MULTIBUS® is a registered trademark of Intel Corporation.

Polaroid™ is a trademark of Polaroid Corporation.

polyFORTH 1 is a licensed product of FORTH, Inc.

VAL™ is a trademark of Unimation, Inc., A Westinghouse Company, Danbury, Connecticut.

Commercial equipment is identified in this paper in order to adequately describe the systems under development. In no case does such identification imply recommendation by the National Bureau of Standards, nor does it imply that this equipment was necessarily the best for the purpose.

FORTH, Inc.

111 N. Sepulveda Blvd.
Manhattan Beach, CA 90266
(213) 372-8493

ORDER FORM AND LICENSE AGREEMENT FOR FORTH, INC. PROGRAM PRODUCTS

FORTH, INC. by acceptance of this Order and Agreement by signature at its headquarters agrees to grant and Customer agrees to accept on the terms and conditions of the agreement, including those set forth on the reverse side, a non-transferable and non-exclusive license to use the Licensed Programs, listed below, and those which are ordered from time to time by the Customer subject to written confirmation by FORTH, INC.

PROGRAM PRODUCTS

FORTH, Inc. Product Code	Designated Computer System	Charges
polyFORTH system to be provided	NBS Real-time Control System	$150.00
by the National Bureau of Standards		

California Sales Tax ____________

TOTAL ____________

THE CUSTOMER ACKNOWLEDGES THAT HE HAS READ THIS AGREEMENT, UNDERSTANDS IT, AND AGREES TO BE BOUND BY ITS TERMS AND, FURTHER, AGREES THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN THE PARTIES, WHICH **SUPERSEDES ALL PROPOSALS**, ORAL OR IMPLIED AND ALL OTHER COMMUNICATIONS BETWEEN THE PARTIES RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

FORTH, INC.

____
Signature

____
Name and Title

____
Date

____
Company Name

____
Address

____
Authorized Signature

____
Name of Signer and Title

____
Date

LICENSED PROGRAMS

Licensed Programs means the FORTH software and related materials furnished to Customer under this Agreement. Software may include without limitation: subroutines, dictionaries, peripheral handlers/drivers, source listings, compiler, meta-assembler, assembler, interpreter, utility routines, mathematical routines, optional routines, multiprogrammer, and text editor. Related materials refer to all materials and documentation furnished to Customer in support of software such as, without limitation, training manuals, specifications, drawings, and the like.

OPTIONAL MATERIALS

For certain Licensed Programs FORTH, Inc. will offer to Customer related Optional Materials as a separate item under this Agreement. When Optional Materials are ordered by the Customer, the term "Licensed Programs" shall also refer to the Optional Materials.

LICENSE

Each license granted to the Customer hereunder authorizes the Customer only to use the Licensed Program on or for the designated single Computer System for which it was ordered. Customer understands and agrees that a separate license and license fee is required to use Licensed Programs or any part thereof on or for any other Computer System.

COPYING

Customer may copy Licensed Programs solely for archival purposes on the designated single Computer System. No other right to reproduce or copy Licensed Programs in whole or in part is granted hereby except in a target-compiled form which contains neither assembler nor compiler. FORTH, Inc. has no liability or obligation with respect to any such target-compiled Licensed Programs.

TITLE

Title to and ownership of Licensed Programs, or any copies thereof, shall at all times remain in FORTH, Inc.

PROTECTION OF LICENSED PROGRAMS

Licensed Programs include valuable information which is proprietary and confidential. Customer agrees to maintain the confidentiality of Licensed Programs and to not reproduce, provide, disclose, distribute, or otherwise make available any item of Licensed Programs, in any form, to any person other than Customer or FORTH, Inc. employees without prior written consent of FORTH, Inc., except that Licensed Programs may be included in Customer's products in a cross-compiled or compressed form which is not extensible. Additional copies of Licensed Programs may be licensed from FORTH, Inc. at charges then in effect.

Customer agrees to take appropriate action by instruction, agreement, or otherwise with his employees or other persons permitted access to Licensed Programs to satisfy his obligations with respect to use, copying, protection, and security of Licensed Programs.

TERM AND TERMINATION

This Agreement is effective from the date on which it is accepted by FORTH, Inc. and shall remain in force until the Agreement is terminated as hereinafter provided.

In the event Customer neglects or fails to perform or observe any of the obligations or conditions under this Agreement, and if such obligations or conditions are not remedied within twenty (20) days after written notice therof has been given to Customer, this Agreement and all licenses granted hereunder shall immediately terminate.

Within two (2) weeks after any such termination, Customer shall certify in writing to FORTH, Inc. that through its best efforts and to the best of its knowledge, the original and any and all copies, in any form, including partial copies or modifications, of Licensed Programs have been destroyed.

WARRANTY

Licensed Program ordered by Customer shall be free from defects in material and workmanship and shall operate in accordance with applicable FORTH, Inc. specifications. This warranty is for sixty (60) days following receipt of Licensed Programs by Customer. During this warranty period, FORTH, Inc. will at its option, repair or replace any software which proves to be defective. THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE and the stated express warranty is in lieu of all liabilities or obligations of FORTH, Inc. for damages including, but not limited to, consequential damages occurring out of or in connection with the use or performance of Licensed Programs.

The Customer acknowledges that it has not been induced to enter into this Agreement by any representations or statements, oral or written, not expressly contained herein or expressly incorporated by reference.

LIMITATION OF LIABILITY

FORTH, Inc. will refund the purchase price or replace or correct any defective item at FORTH, Inc.'s expense provided prompt written notice of the defect is given to FORTH, Inc.

The Licensed Programs licensed hereunder are for use on the standard Computer System designated by the Customer. FORTH, Inc. assumes no liability for any malfunction resulting from the use of Licensed Programs with other than such a standard Computer System or from equipment defects. Should Customer desire to have such a malfunction corrected, Customer shall pay for all work performed by FORTH, Inc. in defining and correcting the malfunction based upon FORTH, Inc.'s standard consulting rates then prevailing.

Customer agrees that FORTH, Inc.'s liability hereunder for damages including but not limited to liability for patent and copyright infringement shall not exceed the charges paid by Customer for the particular Licensed Program involved. Customer further agrees that FORTH, Inc. will not be liable for any lost profits, nor for any claim or demand against the Customer by any other party. No action, regardless of form, arising out of the transactions under this Agreement, may be brought by either party more than one year after the cause of action has accrued, except that an action for non-payment may be brought within one year after the date of last payment. IN NO EVENT WILL FORTH, INC. BE LIABLE FOR INDIRECT OR CONSEQUENTIAL DAMAGES EVEN IF FORTH, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

DELIVERY

Licensed Programs will be shipped to Customer generally with one (1) month after order confirmation unless Customer requests a later delivery date. However, FORTH, Inc. does not represent nor warrant any such shipment date and in no case will FORTH, Inc. be liable for late or nondelivery due to war, acts of God, strikes, shortages of equipment, or other causes beyond its reasonable control.

TRAINING

FORTH, Inc. will provide with the Licensed Programs furnished to Customer reasonable training materials for use of the Licensed Programs on the designated Computer System. Upon the Customer's request, FORTH, Inc. will provide additional training at charges then prevailing.

CONSULTING

Upon request, FORTH, Inc. will provide consulting services to Customer at FORTH, Inc.'s standard rates then prevailing.

TERMS

The changes for the Licensed Programs set forth on the reverse side hereof are payable on delivery unless designated otherwise.

RETURN POLICY

NO RETURNS ACCEPTED WITHOUT PRIOR WRITTEN AUTHORIZATION. Normal restocking charge on returns is 25% of P.O. amount. Merchandise held over 30 days from invoice date is not subject to return.

ASSIGNMENT—SUBLICENSE

This Agreement, the Licenses granted hereunder, and Licensed Programs transferred hereunder may not be assigned, sublicensed, or transferred by the Customer without prior written consent from FORTH, Inc.

GENERAL

The term "this Agreement" as used herein includes any future written amendments, modifications, or supplements thereto.

The terms of this Agreement will take precedence over the terms of any present or future order from the Customer for any license hereunder. Customer agrees that acceptance of future delivery of any Licensed Programs from FORTH, Inc. is conclusive evidence of his agreement that the license for such Program is governed by the terms of this Agreement.

If any of the provisions, or portions thereof, of this Agreement are invalid under any applicable statute or rule of law, they are to that extent to be deemed omitted.

This Agreement will be governed by the laws of the State of California.

INSTRUCTIONS FOR ORDERING RCS SOFTWARE

Required Items to be Sent to NBS:

(1) Copy of valid Order Form and License Agreement from FORTH, Inc. This is obtained by submitting an Order Form and License Agreement form for polyFORTH (a copy is provided on the next page) to FORTH, Inc. at the listed address. There is a fee of $150 which must be sent to FORTH along with the Order Form and License Agreement.

(2) Four (4) Intel Model D2732A EPROMs (or equivalent) for each single-board computer in the system. For a complete RCS system (four single-board computers), a total of sixteen (16) EPROMs are required.

(3) One (1) DataTech Model QuadPlus 1/2" tape (or equivalent), 2400-foot length and 6250 bpi rating.

(4) Completed Order Form from the bottom of this page.

Send these items to the following address:

Robot Systems Division
National Bureau of Standards
Building 220, Room B127
Gaithersburg, MD 20899

ATTN: RCS User's Group

Include the following with your order:

NBS RCS SOFTWARE ORDER FORM

Company Name: ______________________________

Mailing Address: ______________________________

____

____

Telephone No.: ______________________________

Person to Contact: ______________________________

TABLE OF CONTENTS

Page

Chapter 1
INTRODUCTION

Chapter 2
RCS OVERVIEW

Chapter 3
RCS ARCHITECTURE

Chapter 6
BASIC RCS OPERATIONS

Chapter 7
SMACRO

Chapter 8
COMMUNICATIONS

Page

Chapter 9
ROBOT SENSOR LANGUAGE (RSL)

Chapter 10
RSL CONTROL LEVELS

Chapter 11
RSL EXTENSIONS

Appendix C
SYSTEM MAPS

Appendix D
RCS DISK BLOCK ORGANIZATION

Appendix E
8Ø87 OPERATION CODES FOR RCS

Appendix F
JOYSTICK SCHEMATICS

Appendix G
GLOSSARY

LIST OF FIGURES

Page

LIST OF TABLES

Page

Chapter 1
INTRODUCTION

This chapter gives a brief description of the Real-time Control System (RCS) developed by the National Bureau of Standards (NBS) and describes how this manual is organized. The chapter also describes the documentation conventions used throughout the manual to indicate different types of information. The conventions include how the manual presents syntax descriptions of the languages within RCS. Finally, the chapter describes who should read this manual and points out which chapters will be most useful to you, depending on your purpose in using RCS.

1.1 THE PURPOSE OF RCS

RCS is a microprocessor-based system for the real-time control of automated systems. The applications described in this manual are for robotic systems; however, RCS can be used to control other types of automated devices.

RCS has two unique features that generally are not found in other control systems. First, because of the hierarchical control architecture of RCS, you can adapt RCS to control different systems. The hierarchical architecture makes RCS modular, with well-defined interfaces. NBS has been developing these interfaces using RCS for over 1Ø years. Second, RCS provides for real-time control based on sensor measurements. This feature enables you to modify RCS control commands, in real time, based on this sensor data.

1.2 ABOUT THIS MANUAL

This section introduces the organization of the manual, the documentation conventions used in the manual, and the conventions used for RCS words.

How This Manual is Organized

This manual is organized into 13 chapters, six appendices, a glossary, a bibliography, and an index. This chapter, "Introduction", introduces RCS and explains how to use this manual.

Chapter 2, "RCS Overview", provides a brief description of RCS, including its purpose and the basic processing structures implemented.

Chapter 3, "RCS Architecture", is a detailed description of RCS processing structures. This chapter provides the theoretical background necessary for understanding the design of RCS and how it operates.

Chapter 4, "RCS Components", describes the required and optional hardware and software for RCS.

Chapter 5, "RCS Installation Procedures", explains how to configure and install the hardware required to run RCS. It also explains how to install the RCS software supplied on magnetic tape by NBS, and how to get the system running for the first time.

Chapter 6, "Basic RCS Operations", provides step-by-step procedures for several RCS operations and utilities. It explains how to start up and shut down the system, edit and execute code, and use printer and tape utilities.

Chapter 7, "SMACRO", is a complete description of the SMACRO language created by NBS for use with RCS. SMACRO consists of word extensions to the FORTH programming language and operating system.

Chapter 8, "Communications", is a detailed description of the RCS communications process that enables communication of real-time data between the various parts of the system.

Chapter 9, "Robot Sensor Language (RSL)", describes RSL, a task-description language developed by NBS as an application for RCS.

Chapter 1Ø, "RSL Control Levels", describes the structure of the four levels of the RSL application. Using this information, you can modify the RSL code for your own application.

Chapter 11, "RSL Extensions", explains how to create your own extensions to RSL.

Chapter 12, "RCS Application Examples", provides two example robot applications that use RCS. Each example is a step-by-step description of an application, including a partial listing of the SMACRO or RSL code with explanations. Each example traces the execution of a robot task.

Chapter 13, "Debugging Techniques", describes the interactive debugging procedures you can use with RCS.

Appendix A contains information on error messages. Appendices B through F include a user word summary, system maps, a disk directory, information on the 8Ø87 operation codes relevant to RCS, and joystick schematics and interfacing information.

Finally, the manual includes a glossary of RCS and RSL terms, a bibliography listing additional reference material, and an index.

Documentation Conventions

This manual uses the following documentation conventions:

- The name of each keyboard key mentioned in the text appears in uppercase letters. For example, the carriage-return key appears as RETURN.

- In a control-key sequence, a caret (^) represents the CONTROL key. For example, control-C appears as ^C.

- Information that appears on the terminal screen appears in bold print. For example, the **:R rsl>** prompt appears on the terminal when the system is awaiting input from the terminal keyboard.

- Information that you must enter exactly as it appears in the manual is underlined. For example, "Enter 3 LOAD" means to type 3 LOAD and press RETURN.

- Variable data that you are to enter is represented within square brackets ([ ]). The instruction "Enter [block #] LOAD" indicates that you are to replace [block #] with a specific block number when you enter the LOAD command. For example, you might type 3 LOAD and press RETURN.

- The default string delimiter is a carriage return. The characters ^, \, and a space may also delimit strings. You need to use the correct character delimiter to include strings in source code blocks. String delimiters are listed in syntax descriptions.

Note: Memory address locations, board numbers, and port addresses are hexadecimal numbers.

System Word Conventions

RCS defines many words to perform real-time task decomposition but you do not need all of these words to create application programs. This manual describes only the RCS, RSL, and FORTH words used to create application programs.

For the syntax descriptions of RCS, RSL, and FORTH words, this manual shows each command and its operands in the order in which you enter them.

1.3 WHO SHOULD USE THIS MANUAL

This manual is for NBS personnel and for robotics researchers in government, industry, and universities. The manual assumes that you are conducting research and developing applications for the real-time control of robots using sensors, hierarchical task decomposition, and multiple CPUs. This manual is intended for system developers and not for factory-floor operators of robotic systems. Although the manual includes the basic RCS operating procedures, it does not include the information these operators would need for the daily operation of a turnkey system.

This manual assumes that you are familiar with the concepts and terminology of robotics and with the FORTH programming language used extensively in RCS. The polyFORTH 1 Reference Manual and the polyFORTH 8Ø86 Operations Manual from FORTH, Inc., of Hermosa Beach, California, provide in-depth information on the FORTH language and operating system used for RCS.

Also, if you intend to create or modify any RCS 8Ø86 or 8Ø87 assembler code for research or applications, the manual assumes you are already familiar with assembly language programming. Appendix E, "8Ø87 Operation Codes for RCS", lists the 8Ø87 operation codes that affect RCS operation.

Researchers using RCS can be divided into three classes: those who want to enter data for an existing application, those who want to modify an existing application while retaining the basic configuration, and those who want to create new applications.

Entering Data for an Existing Application

To use the sample application written in RSL, you need to edit the blocks of RSL code to enter the specific application data for your system. This robot-control data is specific for each application at a specific installation, and includes defined or recorded robot poses, defined locations and objects, and defined trajectories.

If you want to use RCS to enter specific data for the supplied sample application, be sure to read Chapter 6, "Basic RCS Operations", which describes how to edit blocks of code, and Chapter 9, "Robot Sensor Language (RSL)", which describes the data types used with RSL and how to edit RSL code.

Modifying an Existing Application

A typical modification to an existing application is to add a simple sensor, such as a ranging sensor or switch. When you add a simple sensor, the basic structure and configuration of RSL do not change. You need to edit approximately 1ØØ lines of SMACRO or RSL code.

If you want to use RCS to add a sensor to an existing application, be sure to read Chapter 6, "Basic RCS Operations"; Chapter 7, "SMACRO", which describes SMACRO code; Chapter 9, "Robot Sensor Language (RSL)"; Chapter 1Ø, "RSL Control Levels", which includes information on sensor preprocessing; and Chapter 11, "RSL Extensions", which describes how you can add RSL extensions.

Creating New Applications

To create new applications, you need to modify the basic structures (such as adding a new control level). Be sure to read Chapters 1 through 11 for a complete understanding of RCS and RSL.

Chapter 2 briefly describes the purpose and theory of RCS, and outlines the basic processing structure of RCS architecture. This overview is presented without technical detail; Chapter 3, "RCS Architecture", describes the architecture in greater detail.

2.1 WHAT IS RCS?

RCS is a research tool for investigating the use of real-time sensory-interactive control. The current implementation of RCS uses a multiprocessor, hierarchically structured architecture for control of robotic systems. RCS is a stand-alone system that enables you to develop, compile, and test all necessary programs without a host. RCS provides an interactive programming environment, while retaining the run-time speed of a compiled system.

Multiprocessor System

RCS is designed around a multiprocessor computer system to provide the high-speed processing required to control a multi-axis robot in real time. The multiprocessor architecture also provides a natural modularity, which helps divide the control function of the system into easily understood control levels.

Hierarchical Control Levels

Each hierarchical control level receives a task from the level above and decomposes this task into a set of more primitive tasks, which it sends to the level below. The lowest control level (the robot controller) supplies the machine instructions to the robot. Each control level communicates with the adjacent levels through a common memory that all control levels share.

Modular Architecture

The modularity of RCS also simplifies the task of integrating different kinds of hardware into the system. For example, you can design a module to accept and process input from sensors and then modify existing modules to use this sensory information.

Control Function Techniques

RCS uses the following techniques to simplify the programming of control functions:

- Software modularization into well-bounded functions with clearly defined interfaces to enhance extensibility of the system by clarifying the information that passes between two "black box" modules.

- Use of generic software module structures to simplify module creation, debugging, and maintenance.

- Specification of well-defined data interfaces.

- Use of a common memory structure to simplify integration of independently developed modules, provide access to additional processes, and provide extensive diagnostic and status reporting capabilities.

- Replacement of separate internal looping structures with cyclic execution that repeats at globally specified periodic intervals to ensure that state changes within the system are synchronized with each other.

- Use of state table programming structures that provide an explicit representation of branch conditions. A state table provides an advantage over a nested IF-THEN-ELSE structure because a state table for a complex branch condition is easier to understand and modify.

2.2 THE ATTRIBUTES OF RCS

RCS provides a modular, flexible, and easily understandable computer hardware and software architecture for experimenting with sensory systems and robotic devices operating in real time.

Modularity

Just as a structured program consists of a series of well-defined modular components, the RCS hardware and software is modular. This modularity makes the system more manageable, enabling developers to concentrate their efforts on one module at a time.

The modular nature of the system software lets you reconfigure all or part of the system software simply by loading different software modules. This modularity ensures that if you make changes to one module you do not necessarily have to reconfigure other modules.

The following are examples of RCS modules:

- Initialization procedures
- Real-time control function execution
- Communications
- Preprocessors and postprocessors
- Device drivers
- Diagnostics
- Monitor systems
- Error reporting

Flexibility

To provide hardware flexibility and extensibility, RCS uses the Intel MULTIBUS. The MULTIBUS can support multiple processors, and compatible board products are widely available.

The RCS software is flexible and extensible by its modular nature. Developers can create sample input data for each module and examine the resulting output data to analyze the operation of the module.

Understandability

NBS designed a generic software module structure as the basis for the various modules operating within the system. After you understand the generic module structure, you can more easily understand the specific modules within the system, which are derived from this generic structure.

The modular design of RCS reduces the considerable complexity of a real-time control system to a set of simple, well-defined, and easily understood segments. You can comprehend the operation of the entire system by analyzing the operation of each module separately.

RCS simplifies the analysis of system operation by providing you with interactive diagnostic software tools to monitor system operation continuously, display the system status, and capture data in real time.

2.3 RCS AS AN INPUT-PROCESS-OUTPUT STRUCTURE

The fundamental building block of the RCS architecture is the input-process-output structure, which specifies functional modules that communicate through well-defined data interfaces. The input-process-output structure is shown in Figure 2-1.

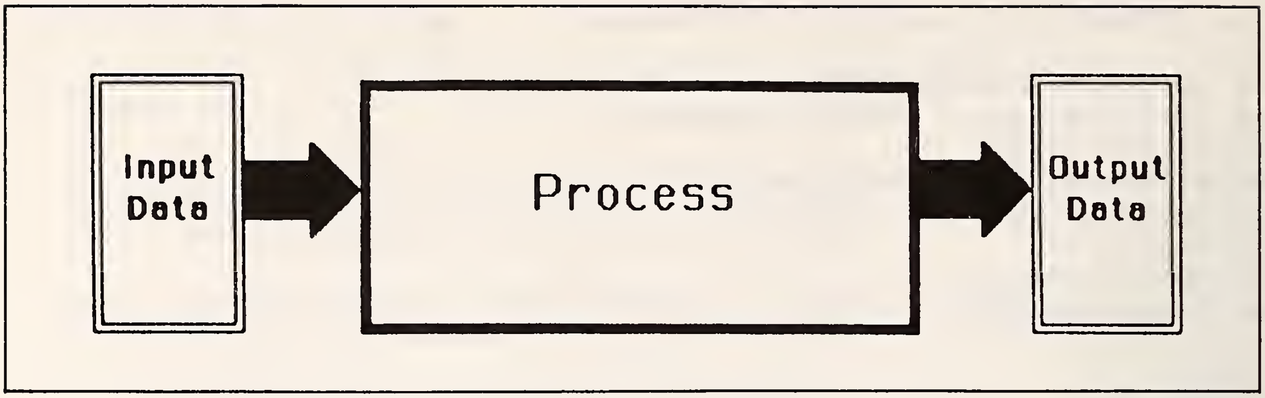


Figure 2-1. The input-process-output structure.

In this model structure, a functional module processes its input data set to generate a set of output data. The function that each module performs is independent of other processing that might occur within the system. Information required by more than one module passes through the interface data sets stored in common memory. You can understand the function of a module by examining its input data, its output data, and a description of the processing that occurs within the module.

You can define new modules to provide new capabilities, such as new sensors, end effectors, or robot trajectory algorithms, and integrate these modules with existing components of the system through the appropriate data interfaces.

In addition to providing a clear understanding of the system, well-bounded modules facilitate module testing and debugging. Simply by supplying input data to a module, you can test that module in isolation before adding it to the system or when diagnosing a problem.

2.4 PROCESSING WITHIN A GENERIC CONTROL LEVEL

Although using the basic input-process-output structure is an efficient method of organizing a complex system, the large number of such structures is a problem in itself.

An effective solution to this problem is the use of generic software module structures. This method helps you understand the system by standardizing the basic operation of a module because you can view each module function as a variation of the generic processing structure.

Within a module, you can view processing as a sequence of three processing operations--preprocessing, decision processing, and postprocessing. (See Figure 2-2.)

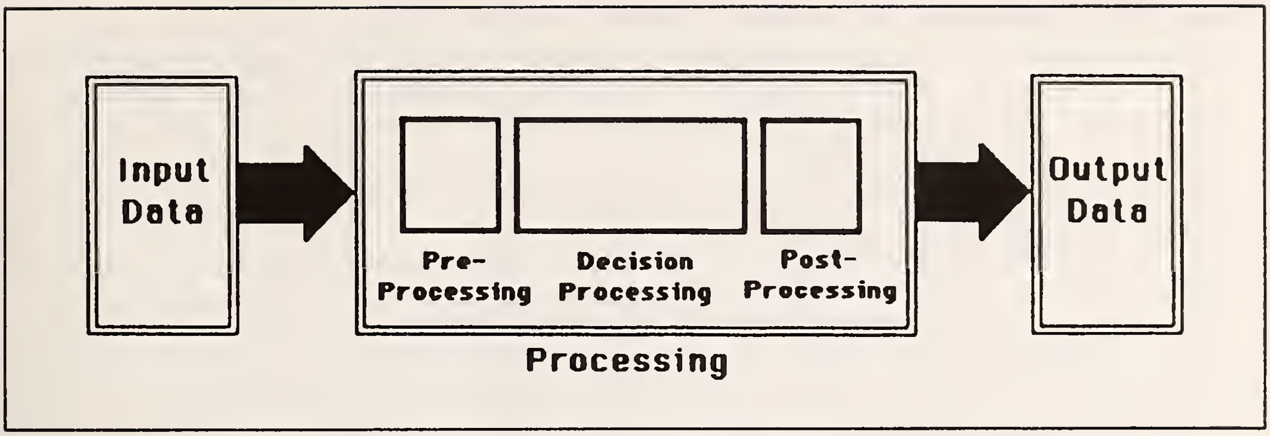


Figure 2-2. The processing structure within a module.

Preprocessing

In the preprocessing stage, the system evaluates, scales, and reduces the input data, and transforms it into the appropriate set of variable values required for decision processing.

Decision Processing

Decision processing is the primary mechanism that directs program execution within a module. Decision processing selects the appropriate procedures for execution based on the values of several high-level variables.

Decision processing defines the various test conditions and the corresponding output procedures for the system to execute if the input from preprocessing satisfies the test conditions. In RCS, state tables usually control this decision-making process.

For a real-time control system, state table programs are preferable to control-path diagrams and IF-THEN-ELSE structures because state tables provide explicit representation of the branch conditions occurring during decision processing. This representation makes branching easier to follow. The state table is an enclosed version of the case statement used in programming languages such as Pascal and C.

Figure 2-3 illustrates the general form of a state table:

Inputs				Outputs
A	B	C	D	
T	X	T	T	5
T	X	T	F	4
T	X	F	X	3
F	T	X	X	2
F	F	X	X	1

T = TRUE F = FALSE X = DON'T CARE

Figure 2-3. The general form of a state table.

The state table in Figure 2-3 contains four input variables: A, B, C, and D, each of which can be either true or false. The system executes the first output where the values of the input variables match all the listed inputs on that line. For example, assume inputs A, B, and C are true and input D is false. When you look through the table, you see that this set up condition generates output 4. In this case, since input B does not affect the outcome, it could be either true or false.

Figure 2-4 shows an implementation of a state table that determines an output command for controlling a robot gripping an object based on the status of touch and force sensors and the position of the gripper.

In this example, there are five input variables: INPUT COMMAND, TOUCH SENSOR #1, TOUCH SENSOR #2, GRIPPER OPENING, and GRIPPER FORCE. The output consists of the command to be executed, OUTPUT COMMAND, and the status of the task currently being executed, OUTPUT STATUS. Each line in the state table corresponds to a set of conditions that can occur when the robot tries to grasp an object with its gripper.

For example, the third row of state table conditions corresponds to a situation where the object is touching one of the gripper fingers (TOUCH SENSOR #1 has contact) but not the other. In this case, the output commands the robot to move Ø.Ø5 mm to try to center the object between the gripper fingers and also commands the gripper to close Ø.1 mm to grasp the object. The OUTPUT STATUS indicates that RCS is executing the GRASP command and that contact with the object has been made. Other lines in the state table correspond to other situations, including errors which might occur, and the corresponding output response that the robot and gripper should make.

INPUT COMMAND	TOUCH SENSOR #1	TOUCH SENSOR #2	GRIPPER OPENING	GRIPPER FORCE	OUTPUT COMMAND	OUTPUT STATUS
GRASP	NO CONTACT	NO CONTACT	X	X	CLOSE .1 MM	GRASP-EXECUTING
GRASP	NO CONTACT	CONTACT	X	X	CLOSE .1 MM MOVE +.05 MM	GRASP-EXECUTING CONTACT
GRASP	CONTACT	NO CONTACT	X	X	CLOSE .1 MM MOVE -.05 MM	GRASP-EXECUTING CONTACT
GRASP	CONTACT	CONTACT	≥ object size	< grip force	CLOSE .1 MM	GRASP-EXECUTING FORCE
GRASP	CONTACT	CONTACT	≥ object size	≥ grip force	PAUSE	GRASP-FINISHED OBJ-IN-HAND
GRASP	CONTACT	CONTACT	< obj. size ≠ closed	X	PAUSE	GRASP-FINISHED NOT-OBJ, SIZE-VALUE
GRASP	CONTACT	CONTACT	CLOSED	X	PAUSE	GRASP-FINISHED NO-OBJ-IN-HAND

Figure 2-4. Example of a state table.

Postprocessing

The postprocessing stage performs any additional processing required after decision processing is complete. Saving internal variables and reformatting algorithm results for output or transfer to other modules are examples of postprocessing tasks.

This chapter describes the ways the control-level software running on the single-board computers can communicate and cooperate to perform control functions. It includes descriptions of the independence of data, program structures, hierarchical control architecture, interlevel communications, decision processing, and the contents of common memory.

3.1 HIERARCHICAL CONTROL ARCHITECTURE

Hierarchical control levels enable you to break a complex control task into manageable procedures. Each control level represents a well-defined, clearly bounded control function with a small number of inputs and outputs.

In this control scheme, you enter a high-level command such as "transfer the part" to the top level of a hierarchical system of control levels. The top level decomposes the task into sequences of more primitive commands (subtasks) to the next lower control level in the hierarchy. The bottom level in the control-level hierarchy generates the outputs to the actuator that performs the action.

Structure of a Control Level

The atomic unit within a control level is a functionally bounded module. This module consists of inputs, a process, and outputs (see Figure 3-1).

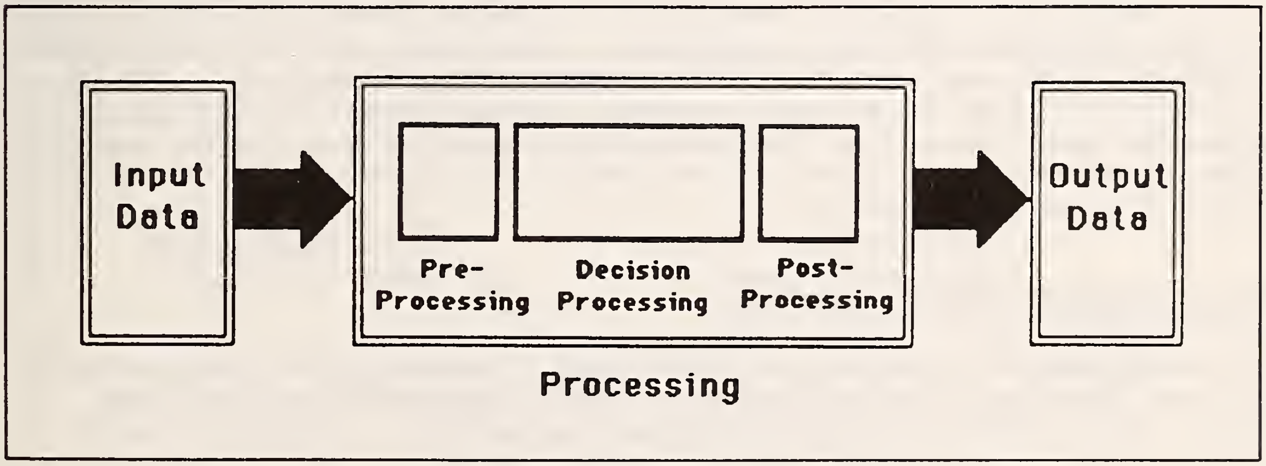


Figure 3-1. Components of a functionally bounded module.

The process shown in this figure includes the preprocessing, decision-processing, and postprocessing stages described in Chapter 2, "RCS Overview". The preprocessing stage accepts and conditions the inputs for the decision-processing stage of the module, and the postprocessing stage prepares the output and performs the handshaking required for data transmission.

You can group functionally bounded modules under a named high-level routine (a routine containing a series of statements that execute other routines). You can then execute this group of modules by entering the name of the high-level routine. RCS uses this method to maintain hierarchical relationships between routines.

A control level consists of groups of routines. The routines perform the functions of preprocessing the input, decision processing based on the input, and postprocessing the output.

Cyclical Control-Level Processing

RCS does not require the use of interrupts to respond to external events. Instead, RCS performs its processing in a cycle based on the periodic communications it conducts between different control levels. This cycle is called the communications cycle.

Real-time control depends on producing a response to changes in input data quickly enough for that response to be stable and effective. The system can provide continuous real-time control without interrupts only if the communications cycle repetition rate is fast enough (that is, the period of the communications process is short enough) to give the processing modules time to respond to the new input data.

RCS uses multiple processors to achieve the high-speed parallel processing needed to support a fast repetition rate. Each control level executes once during every communications cycle.

Cyclic execution has a major impact on the way control-level decision processing is implemented. Because a typical task takes many cycles to complete, the level must enable you to enter the algorithm description each cycle without losing track of what has already been done. RCS uses state variables to accomplish cyclic execution. State variables record the state of the algorithm at the end of each cycle and are used at the beginning of the next cycle to pick up algorithm execution.

Control-Level Decision Processing

You can program the decision-processing stage of a control level using a state table and state variables or using a series of IF-THEN-ELSE control structures. Each of these options provides certain benefits.

A state table has the advantage of separating the definition of a function from the timing of its execution. However, this separation can be a disadvantage when you are trying to trace the sequence of program execution. A series of IF-THEN-ELSE statements is easy to follow when you are tracing the sequence of program execution, but hard to understand and follow if it becomes long and complex.

The SMACRO language developed by NBS supports the following format for state tables:

```
     state-table [state-var1] [state-var2] . . . [state-var7]
     state:      [test-var1]  [test-var2]  . . . [test-var7]  [routine A]
     state:      [test-var8]  [test-var9]  . . . [test-var14] [routine B]
     .
     .
     .
     default-state: [default-routine]
     end-state-table
```

During execution, the system searches through the state table until it finds a line in which the values of all test variables match the values of the corresponding state variables. After the state table finds a line with matching variable values, it executes the routine at the end of that line. All lines after the first matching line are ignored. The state table executes the default routine if no line matches.

The following list contains examples of questions that a state table can answer for the decision-processing stage by examining test-variable status information:

- Has the level received a new command from the next higher level?
- Has the lower level returned its status to the current level?
- Has the lower level confirmed that it has received the last command?
- How far has the procedure progressed through its execution sequence?
- Have the sensors specified values for all symbolic variables?
- Has a specified event occurred?

Figure 3-2 illustrates the sources of information that a control-level procedure uses to determine the command to send to the next lower level and the status to send to the next higher level.

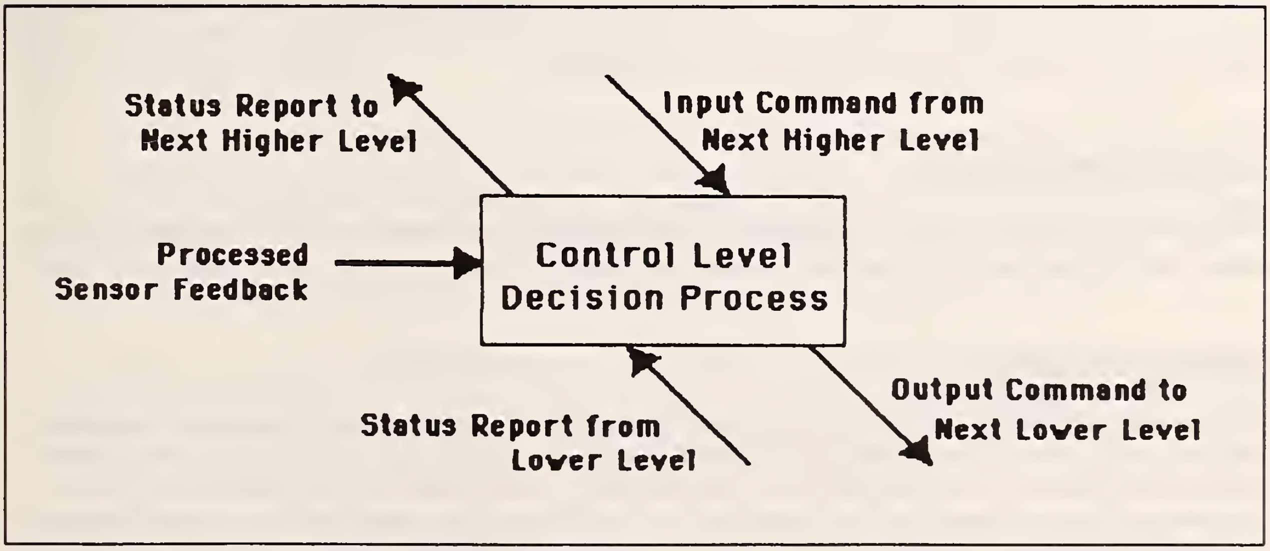

Status Report to Next Higher Level
Input Command from Next Higher Level
Processed Sensor Feedback
Control Level Decision Process
Output Command to Next Lower Level
Status Report from Lower Level

Figure 3-2. Inputs and outputs from control-level decision processing.

Each control level may require multiple cycles to complete a command. After a control level completes its command, it sends a done status message to the next higher level, commands the lower levels to pause, and waits for a new command.

Interlevel Communications

The information that the communications process transfers to any control level consists of commands issued from the level above and status information returned from the level below.

The commands passed to a control level from the level above vary greatly from level to level and from application to application. However, the status information passed to a control level from the level below (or from other components such as sensor systems) usually reflects one of three possible results: executing, done, or error. The error status message can also indicate the reason for the failure.

The communications process is described in more detail in Chapter 8, "Communications".

3.2 RCS PROGRAMS

A complete control system based on RCS includes multiple programs running in parallel on the various boards within the system. These programs access the data they require from variables stored in common memory. To make the programs data-independent, you must use symbolic variables to represent the physical system and the objects manipulated by the system.

Programs also use common memory to pass command and status information to programs running on other boards. To enable communications between levels to be asynchronous, common memory provides double buffering for the input and output of the different control levels in the system.

3.3 RCS COMMON MEMORY

The common memory of RCS contains three major components: the system dictionary, the communication buffers, and the data files.

System Dictionary

The system dictionary contains records for every variable, procedure, state table, and list of variables in the system while it is operating. The system dictionary makes the interactive features of RCS possible by enabling you to locate, modify, and test any section of code quickly without having to modify the entire system.

After loading the application software, the contents of the system dictionary enable you to perform the following functions:

- Execute procedures (functionally bounded modules)
- Create and initialize new variables
- Define lists of variables
- Show the value of a variable or a list of variables

The system dictionary maintains pointers to procedures and RCS updates these pointers automatically when you reload a procedure. This automatic updating enables you to modify a procedure, reload the procedure, and then run the system without having to modify other procedures to integrate the altered procedure.

Communication Buffers

The communication buffers provide communication paths between control levels, sensor systems, robots, grippers, and other subsystems. RCS simplifies the transfer of information between control levels by providing high-level commands that make the details of the communications process transparent to the control-level procedure.

To program a level to receive input commands or status, you identify the name of the sending level and the name of the list of variables to be transferred. Similarly, to program a level to send output commands or status, you identify the name of the receiving level and the name of the list of variables to be transferred.

The communications cycle reads output buffers every cycle and fills input buffers on the same cycle only if those buffers are ready to accept input. The communications protocol is described in more detail in Chapter 8, "Communications".

Data in Common Memory

The factory of the future will maintain a database that describes every system and object within its domain, and this data will be supplied to the control algorithms during the manufacturing process. However, for RCS you must enter manually the data required to specify the geometry of each object, the location of each critical point, the approach path, the departure path, and intermediate trajectories for each application of the system.

To maintain the data independence of a control algorithm, you should always separate data from the algorithm that uses the data. Separating the data and task definitions ensures that only the data, and not the algorithm, will require alteration if the physical aspects of the system or the object change.

The common memory stores the data for all control levels and serves as the interface between communicating control levels and between the control levels and input/output systems. When a procedure requires data, its control level assigns data from common memory to the appropriate symbolic variables within the procedures.

Chapter 9, "Robot Sensor Language (RSL)", describes the NBS-developed language for robot-control applications. Therefore, this chapter presents only a brief overview of the kinds of information RCS needs in common memory to control a robot. Consider the example of a robot wrist and gripper. The kind of data you place in common memory to define the physical characteristics of a system with a robot wrist and gripper would include poses, objects, locations, and trajectories.

Pose data

Poses comprise the most fundamental data required to control a robot. You use the name of each pose to identify a specific position and orientation in the workspace of the robot.

To create a pose, you enter numerical information from the keyboard or use a joystick to describe a specific position and orientation. Alternatively, you can program the robot system to "learn" poses by using sensor systems to capture a pose you select manually.

A pose in the example robot gripper system consists of a specific position and orientation for the wrist and gripper. Figure 3-3 illustrates some different poses for this system.

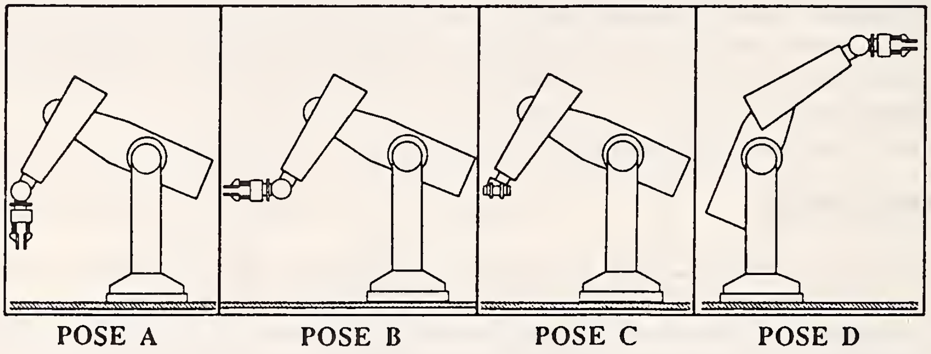


Figure 3-3. Examples of poses.

Object data

Object data specifies such things as the grip size and grip location on an object that the robot is manipulating. The grip size specifies the width that the gripper needs to accommodate and the grip location specifies the location at which to grip the object. Figure 3-4 illustrates the grip size and grip location on an object.

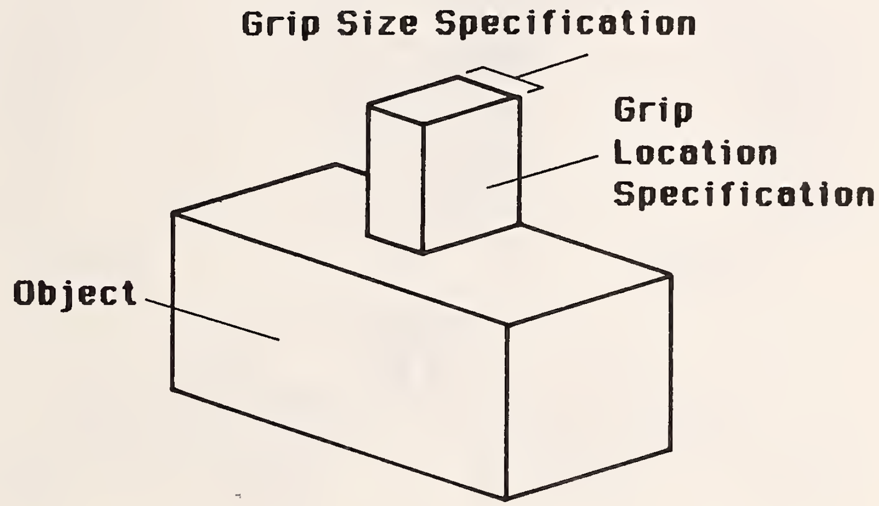


Figure 3-4. Example object data specifications.

Location data

Location specifications refer to coordinates within the workspace of the robot. These specifications describe points such as the goal point of a move-to command.

Figure 3-5 illustrates an example of a location where the robot is to place an object.

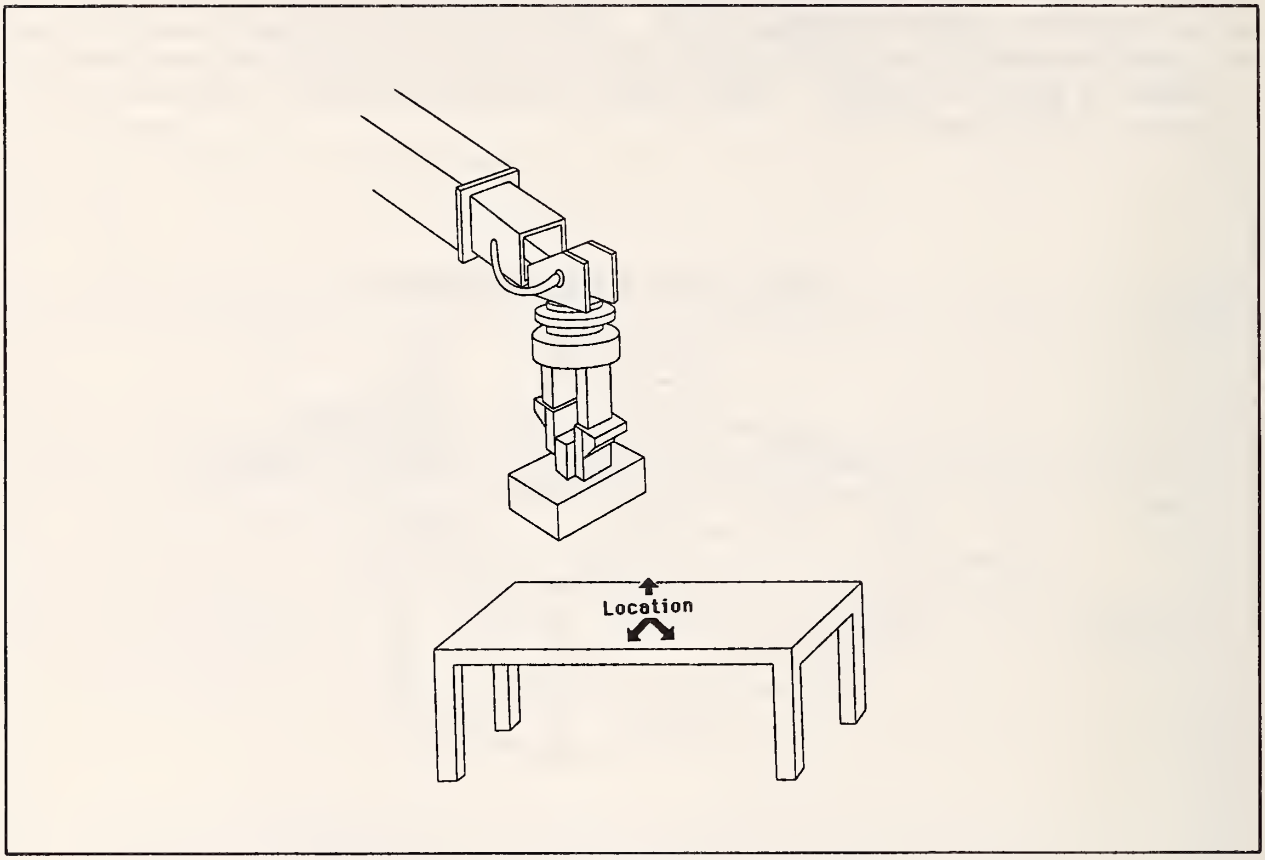


Figure 3-5. Example location.

Trajectory data

Trajectories define how the robot is to move from one location to another. Each object and location has specific requirements as to how the gripper should approach and depart. You also need to provide intermediate trajectory data to guide the gripper from one departure to the next approach.

Trajectory data can refer to objects, location points, or arrays of location points. The specifications for a trajectory can include specific beginning or end location points (as well as objects or arrays), maximum acceleration, maximum velocity, and braking distance.

Figure 3-6 illustrates some of the possible components of trajectory data.

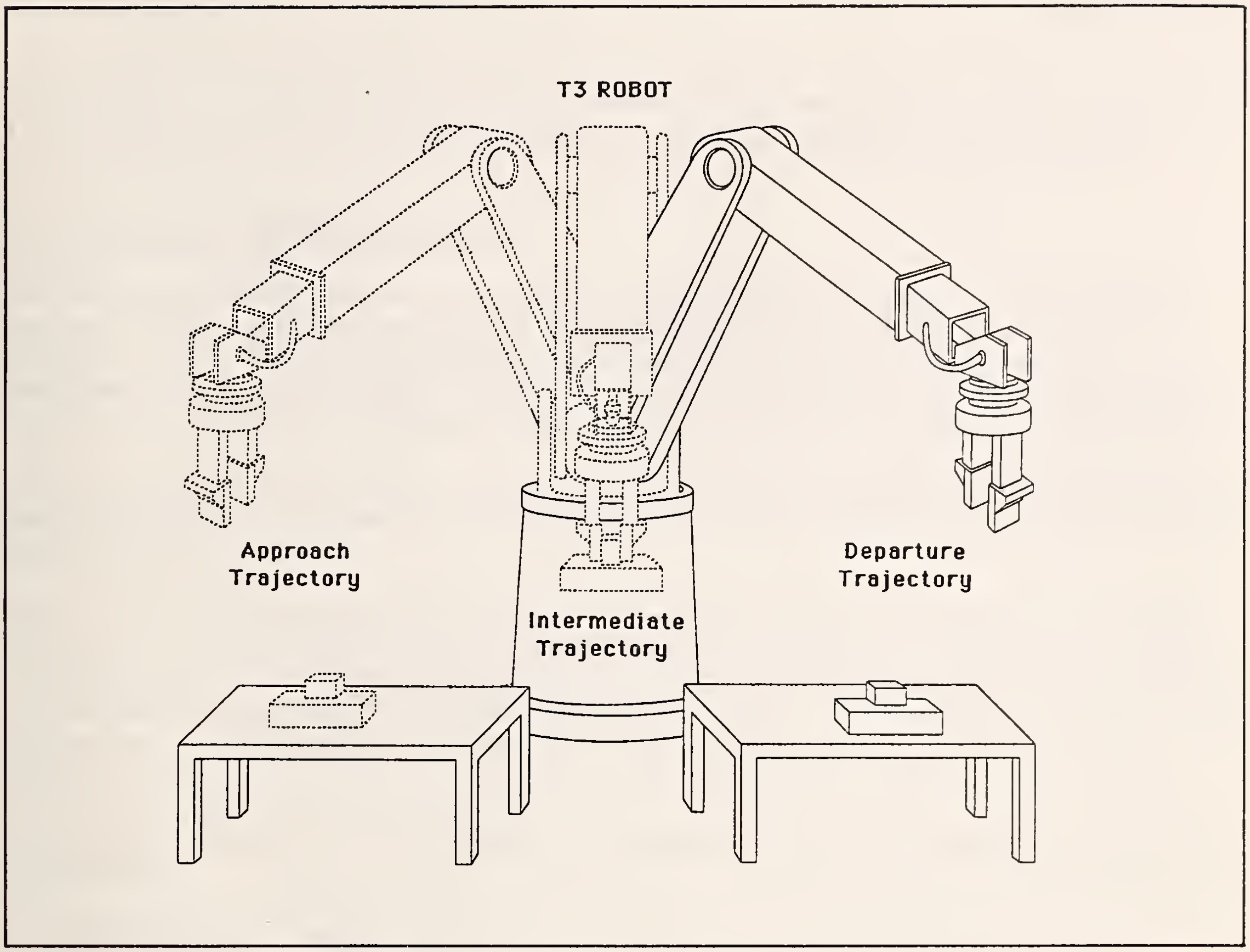


Figure 3-6. Examples of trajectories.

This chapter describes the hardware and software components of RCS, including the organization of the memory address space, I/O space, and disk blocks.

4.1 HARDWARE COMPONENTS OF RCS

RCS is a MULTIBUS-based multiprocessor microcomputer system. The MULTIBUS defines a bus structure supported by many vendors with compatible board products. RCS currently supports up to seven Intel 86/3Ø single-board computers, each with an 8Ø87 floating point coprocessor. Each 86/3Ø computer can run one or more RCS control levels, depending on timing considerations.

Each single-board computer includes local RAM and contains the FORTH operating system in PROM. All computers in the system share access to a common memory and Winchester disk.

One of the single-board computers is the master board; the others are slave boards. The user terminal attaches to the master board. Except for unusual circumstances (such as a system crash), all user communication with the slave boards occurs indirectly through the master board. If necessary, you can add a terminal switch box to the system to communicate directly with a slave board.

In addition to one or more single-board computers, RCS includes the following components:

- Shared-memory boards
- Numeric data processors
- Disk/tape controller board
- Winchester disk drive
- Nine-track tape drive
- Terminal

Depending on the RCS application, it may also contain optional components, such as:

- Serial I/O boards
- Parallel I/O boards
- Analog I/O boards
- Joystick
- Printer

RCS can connect directly to, and control the operation of, a wide variety of robotic equipment. See Chapter 5, "RCS Installation Procedures", for a complete description of the system hardware.

MULTIBUS Address Space Organization

In RCS, the address space for the disk control block, FORTH PROMs, and semaphores are fixed. The user allocates address space for processors and common memory. RCS supports either 2Ø-bit or 24-bit addressing (24-bit addressing requires the P2 connector on the MULTIBUS).

With 24-bit addressing, the address space is divided into 16 one-megabyte pages. Processors must be in page Ø. You can distribute common memory over any two (but only two) other pages.

The RSL application uses 2Ø-bit addressing. See Section C.1, "Multibus Address Space", for the exact addresses.

MULTIBUS I/O Space Organization

RCS allocates the MULTIBUS I/O space as shown in Section C.2, "Multibus I/O Space Organization". All unused I/O address space and the J3 and J4 ports are available for user applications.

RCS Interrupt Structure

Each processor board in the system uses interrupts to control RCS operation. See Section C.3, "RCS Interrupt Assignments", for the use of each interrupt.

RCS Disk Block Organization

RCS uses the FORTH block structure for all source code. The disk block map in Appendix D, "RCS Disk Block Organization", shows how RCS allocates blocks on the Winchester disk for the RSL 1.6 application.

RCS uses a 1Ø-block directory structure within these source code blocks. The system does not automatically maintain this directory information; you must update the block-directory blocks when you add, delete, or move blocks. For more information on directories, see Section 6.3, "Locating Source Code".

4.2 SOFTWARE COMPONENTS OF RCS

The RCS operating system software developed by NBS is based on a compiled version of the FORTH programming language. This operating system supports multiple processors, interprocessor communications, and a mid-level high-speed compiled language called SMACRO.

The operating system provides an environment for the development of application programs for real-time control. One such application program is RSL, which runs under RCS.

The RCS software includes an operating system with system utilities and macros. The utilities enable you to edit source code, communicate with individual computers within the system, debug application programs, back up programs and data, and perform other tasks. The macros form the SMACRO language you can use to create application programs.

Other components of the RCS software include SMACRO files in common memory, system dictionary vocabularies, and the communications utility, COMM.

The Operating System

The RCS operating system includes utilities for booting the system, editing source code, and performing other tasks. It provides multiprocessor operation by using one master processor board to control several slave processor boards. In addition to these features, the RCS operating system supports background tasks, interrupts, and four modes of operation.

System utilities

Many RCS system utilities perform simple tasks, such as printing a block of source code or selecting a specific type of terminal. Other utilities (such as the MBOOT, CUSTOM, and EDITING utilities) are more powerful.

Some MBOOT and CUSTOM utilities enable you to boot the system, load the system software, and load the custom software required for your particular applications. Other MBOOT and CUSTOM utilities enable you to save memory images on the Winchester disk, and then reload them into memory. These utilities are useful when you need to recover from a system crash.

The EDITING utilities enable you to examine and modify source code. RCS supports three kinds of terminals: the TeleVideo Model 95Ø, the Datamedia Elite 1521A, and the DEC VT1ØØ.

Multiprocessor operation

To provide multiprocessor operation, the RCS operating system enables you to define one processor board as the master board, and the other processor boards as slave boards. The terminal can communicate directly with only the master processor board. Communication with the slave boards occurs indirectly through the master board.

Background tasks

The RSL application developed by NBS includes tasks that run in the background. Refer to the RSL source code for an example of background task operation. The polyFORTH 1 Reference Manual and the polyFORTH 8Ø86 Operations Manual provide more information on running tasks in background mode.

Interrupt routines

RCS includes specific SMACRO words for interrupt routines. The enter-interrupt SMACRO word prepares an 8Ø86 processor to perform an interrupt routine first by pushing the contents of all registers and then by pointing the DS and ES registers to specific segments.

The exit-interrupt SMACRO word should appear at the end of an interrupt routine, to restore the contents of the 8Ø86 registers and then execute an IRET. You can use the ~INTERRUPT SMACRO word to set the 8Ø86 interrupt vector to point to a selected SMACRO routine.

The operation of these SMACRO words is explained in detail in Section 7.6, "Interrupts and Assembly Language". Refer to Section C.3, "RCS Interrupt Assignments", for a list of the reserved interrupts.

Operating modes

RCS can run in one of four modes: Run, Show, Locate, and Compile. Run mode is the normal operating mode for controlling robotic equipment. Show mode enables you to examine the values of variables. Locate mode enables you to search for the source block of any SMACRO word. Compile mode is for compiling blocks of RCS code.

RCS SMACRO Language

The RCS operating system is based on the FORTH language and operating system. NBS has extended FORTH by adding words to the standard set of FORTH words. This set of extensions comprises the SMACRO language.

Each SMACRO variable is a member of a set of variables belonging to a variable owner. SMACRO variable types include: integer, byte, floating point, array, string, segment, and sequential. The owners in SMACRO include: variable owners, sequential owners, and list owners. In SMACRO, unlike FORTH, you can define a word that is longer than one block by using the SMACRO words routine and end-routine. For a complete description of SMACRO, see Chapter 7, "SMACRO". For a complete list of SMACRO words, see Appendix B, "User Word Summary".

SMACRO Files

RCS operation is based on SMACRO files. A SMACRO file is stored using a linked-list record structure, residing in common memory. The memory-resident SMACRO files provide a means of communication between the processor boards within a system.

Disk Files

RCS does not have traditional disk files. RCS stores source code in blocks. You keep track of which block holds what source code by maintaining the RCS system of directory blocks. The directory blocks help you locate code; RCS does not use the directory blocks.

RCS also provides a mechanism for storing compiled code on the disk. This method is similar to the method for storing an object code file on a traditional system. The compiled code is stored in D>Ms, where D>M is simply a copy of the local RAM for a board. D>Ms are less flexible than the traditional object code file, but they are faster to load because no linking is required.

Robot Sensor Language (RSL)

NBS has developed RSL as a sample RCS application you can use and modify for your own robot system. It includes four control levels, each representing one level of task decomposition.

The four RSL control levels are TASK, PATH, PRIM, and JOINT. TASK decomposes a task into a sequence of paths. PATH decomposes each path into sequences of goal poses. PRIM decomposes the poses into a sequence of intermediate poses. JOINT decomposes the intermediate poses into commanded joint angles.

System Dictionary Vocabularies

RCS maintains a system dictionary in common memory. The system dictionary contains entries for every variable and routine in the system. A vocabulary is a group of related entries in the dictionary. RCS includes a system vocabulary on each board, and a vocabulary for each control level, including RSL. You can include up to five user-defined vocabularies per board (although you usually have only one vocabulary per control level).

The Communications Utility (COMM)

The RCS operating system includes the COMM communications utility, which performs communications between processor boards within the system. This utility enables you to restart the timer, set up communication buffers, and specify the communications timing. See Chapter 8, "Communications", for more information on the communications process.

This chapter lists the hardware required to support the RCS and RSL software, and explains the hardware and software installation procedures.

5.1 HARDWARE REQUIREMENTS

NBS supplies the software for an RCS system; you supply the hardware (see Figure 5-1).

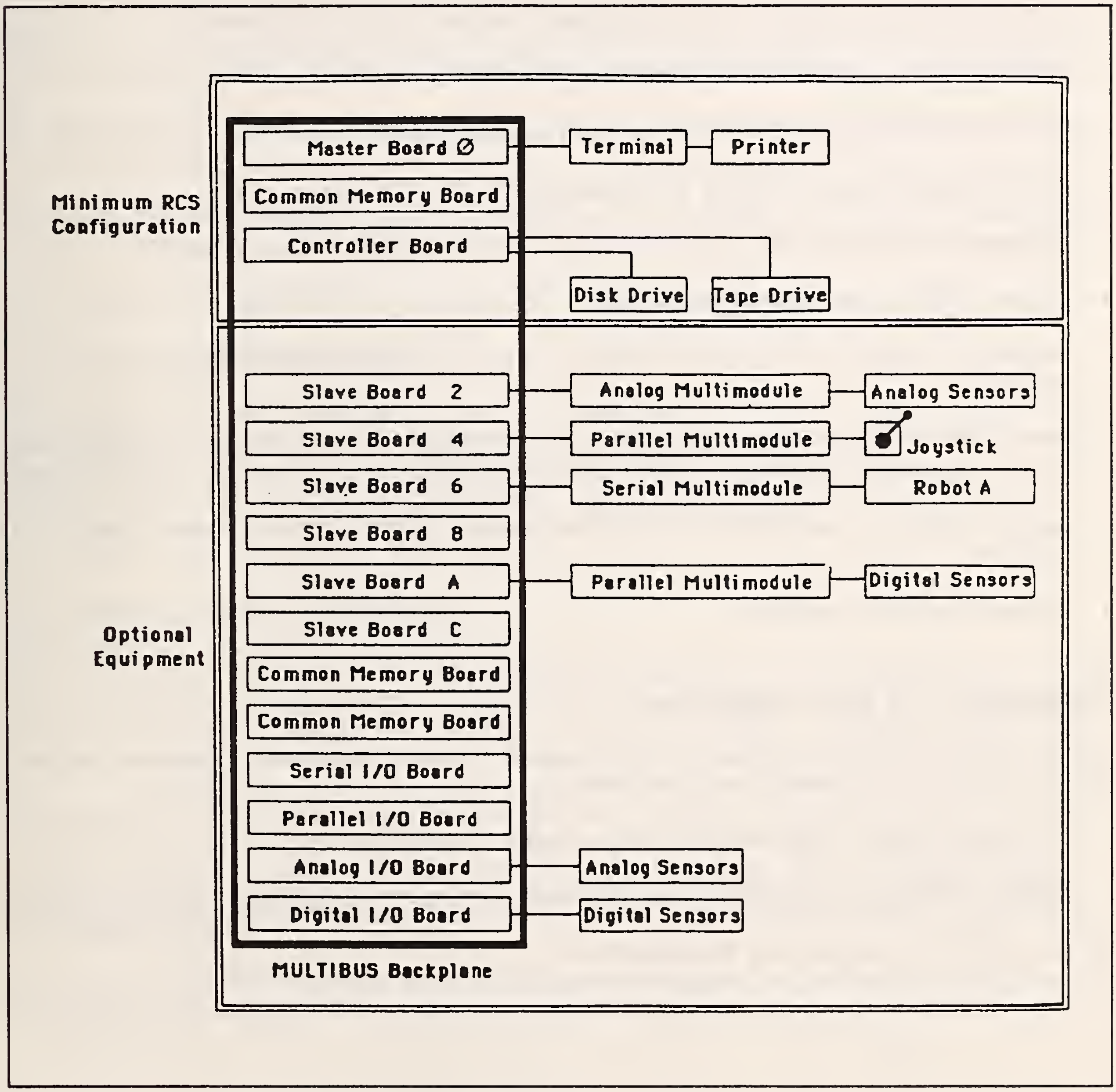


Figure 5-1. Example RCS hardware configuration.

Minimum Hardware Requirements

The following list describes the minimum hardware configuration needed to run RCS:

- Equipment rack.
- MULTIBUS backplane and power supply (for example, the ETI Model 8223 23-slot chassis, including a parallel priority circuit).
- Intel iSBC 86/30 single-board computer configured with a bus clock.
- Intel iSBC 337A numeric data processor module for 86/30 board.
- Plessey Model PSM 512A (or equivalent) memory board.
- Ciprico Rimfire 45 disk and tape controller board.
- Priam Diskos Model 6650-10 66-megabyte Winchester disk drive.
- Priam Model 3 00106-04 ANSI-standard disk interface adapter board.
- Cipher F880 nine-track tape drive.
- TeleVideo Model 950, Datamedia Elite 1521A, or DEC VT100-compatible terminal.

  Note: The NBS-supplied system tape is configured for a Micro-Term ERGO 301 terminal, which is VT100 compatible.

- Apple Imagewriter, DECwriter III, or Integral Data Systems Paper Tiger printer.
- Interconnection cables.

Unimate Puma 760 Robot Requirements

The following list describes the additional system components required to run RSL to control a Unimate Puma 760 robot.

- Three additional iSBC 86/30 single-board computers.
- Three additional iSBC 337A numeric data processor modules.
- One iSBX 351 Serial I/O Multimodule.

Additional Components

To modify RCS to develop other applications or control other types of robotic machinery, you can add the following components:

- Intel MULTIBUS boards and multimodules
  - Up to three additional iSBC 86/3Ø computers (for a maximum of seven)
  - iSBC 534 Serial I/O board
  - iSBC 519 Parallel I/O board
  - iSBX 351 Serial I/O Multimodule
  - iSBX 35Ø Parallel I/O Multimodule
  - iSBX 311 Analog Input Multimodule
  - iSBX 328 Analog Output Multimodule

- One to four additional Plessey Model PSM 512A (or equivalent) memory boards (this application requires 24-bit addressing capability).

- MULTIBUS P2 backplane connector installed in chassis (required for 24-bit addressing).

- A switch box, compatible with RS-232 and including one input and seven outputs (required only if you need direct communication with slave boards).

- Joystick, as specified in Appendix F, "Joystick Schematics".

- Robot with optional sensors.

Your system may not require all of the hardware included in these lists. The configuration of your system depends on the requirements of your robot. For example, the Analog Input Multimodule is not required for RCS operation, but may be needed to provide sensor input. However, you should plan ahead by ordering a spare for each critical component in the system.

5.2 ROBOT INTERFACE REQUIREMENTS

RCS provides an environment for developing programs that operate in real time to control robotic equipment. RCS makes no assumptions about the machine it controls; the machine does not even need to be a robot. You must supply the software interface for the specific machine you want to control using RCS.

NBS provides interface software designed to control a Unimate Puma 76Ø robot through a serial I/O multimodule. This software also can control Unimate robots other than the Puma 76Ø. For more information, see the Unimate Puma Control System SLAVE Interface Specification for External Computer Path Control Using VAL II.

To control robotic devices from other manufacturers, you must develop your own interface software according to the specific requirements of each device.

5.3 SOFTWARE REQUIREMENTS

The software required to run RCS applications includes PROMs containing the basic FORTH operating system, and a magnetic tape containing the SMACRO extensions to FORTH.

The magnetic tape supplied by NBS also includes RSL, an application developed by NBS. You can modify RSL for your application or copy selected RSL procedures as a basis for your own RCS application.

5.4 HARDWARE INSTALLATION PROCEDURES

To install the hardware required to run RCS, you must configure the processor boards, the common memory board, the disk and tape controller board, the disk drive, the terminal, and the printer.

Configuring the Processor Boards

Before installing the 86/30 processor boards in the system, you must change the factory-default configuration of the boards, add the PROMs containing the FORTH operating system, install the 8087 numeric data processors, add heat sinks, set up the bus clock on one of the boards, and set up RCS communications on one of the boards.

Installing processor-board jumpers

Configure each processor board by performing the jumper modifications indicated in Table 5-1. These modifications are from the Intel default jumper configurations.

Table 5-1. Processor-Board Jumper Modifications.

Modification	Result
Add 204-206	Enables bus lock
Pull 205-207, pull 208-209	Disables bus clock
Pull 210-211	Disables serial priority
Pull 213-214, add 212-213	Grounds CBRQ/
Pull 219-225	Sets the RAM address to 2000
Add 218-223, add 238-239	Selects 24-bit addressing
Add 124-125	Selects 32K PROM size
Pull 111-112, add 112-113	Selects 32K PROM size
Pull 151-152, pull 147-158	Deletes interrupt assignments
Add 164-166	Selects 8087 error interrupt
Add 136-154, add 153-157	Selects onboard serial port interrupts
Add 133-165	Assigns the 6-msec time-out interrupt
Pull 175-176, add 175-184	Sets PIT clocks
Add 76-77	Connects RTS to CTS

After making these jumper modifications, cut connection line 32 of the P2 connector to disconnect a non-tristate output.

You may need additional jumpers to configure a processor board for the onboard parallel port or for one of the add-on multimodule ports.

To select the address for the RAM on each 86/30 processor board, configure the jumpers for each starting address as shown in Table 5-2.

Table 5-2. Processor-Board Starting-Address Jumpers.

Board Numbers	Starting Address	Jumpers	RSL Control Level
0	00000	232-233	RSL
2	20000	None	PRIM
4	40000	219-226, 232-233	TASK/PATH
6	60000	219-226	JOINT/COMM
8	80000	219-225, 232-233	not used for RSL
A	A0000	219-225	not used for RSL
C	C0000	219-225, 219-226, 232-233	not used for RSL

Installing the FORTH PROM circuits

Install on each 86/30 processor board the four PROMs containing the FORTH operating system supplied by NBS.

Installing the 8087 numeric data processors

For each processor board, carefully remove the 8086 microprocessor from the board and install it on a small board (Intel 337A) with an 8087 numeric data processor. Next, install the 8087 board into the 8086 socket on the processor board. If you are using an old 8087 that runs at a maximum clock speed of 5 MHz, add jumper 36-37 to the processor board.

Installing heat sinks

After installing the 8087 board, install a heat sink (using silicone heat sink compound) on top of the 8086 and 8087 processors to ensure adequate cooling.

Installing the bus clock

You must enable the bus clock on one, and only one, of the processor boards (usually board 0) within the system. Install jumpers 205-207 and 208-209 to enable the bus clock. These jumpers should not be present on any other processor board in the system.

Configuring interprocessor communications

The processor board responsible for RCS communications between levels is the COMM board (although it may also perform functions not related to communications).

Configure the COMM board to send the COMM bit to the other boards by adding jumper 245-251. This jumper enables status register bit 5 to invoke bus interrupt 2.

Configure the other processor boards to receive the COMM bit by adding jumper 147-148. This jumper assigns bus interrupt 2 to bit 2 of the Programmable Interrupt Controller (PIC).

Configuring the Common Memory Board

Modify the factory-default configuration of the common memory board (Plessey Model PSM 512A) by modifying the jumper wires as indicated in Table 5-3.

Table 5-3. Common Memory Board Jumper Modifications.

Modification	Result
Pull LK19	Disables the PWRfail interrupt
Pull LK21	Selects parallel priority operation
Pull LK22-31	Disables the error interrupt

To run RSL, set the address and interrupt switches of the common memory board to the settings shown in Table 5-4.

Table 5-4. Common Memory Board Switch Settings for RSL.

Switch Settings									Result
Pole:	1	2	3	4	5	6	7	8	(Pole 1 is at the left)
SW1:	on	off	off	off	off	off	off	off	Disables interrupts
SW2:	off	off	off	off	off	off	off	off	Selects megabyte page
SW3:	off	on	on	on	on	on	on	on	Selects 80000 as first address
SW4:	off	off	off	off	off	off	on	off	Selects EFFFF as last address

Configuring the Disk and Tape Controller Board

Modify the factory-default configuration of the Rimfire 45 disk and tape controller board by installing the jumper wires indicated in Table 5-5.

Table 5-5. Disk and Tape Controller-Board Jumper Modifications.

Modification	Result
Pull 15-16, add 15-17	Selects 16-bit bus operation
Pull 23-26	Selects parallel priority operation
Add 13-14	Enables bus time-out interrupt
Pull 31 through 5Ø	
Add 5Ø-49-48-47-45-44-43-42	
Add 46-41	Sets the initialization address to EFCØ6
Add 4Ø-39-38	
Add 37-36-35-34-33-32-31	

For operation with the RCS system, you must set the channel attention of the Rimfire 45 controller board to ØØ52 and the bus width to 16. You must also set the DIP switches on the Rimfire controller board to the settings shown in Table 5-6.

Table 5-6. Disk and Tape Controller-Board Switch Settings.

Switch Settings

Bit:	1	2	3	4	5	6	7	8
SW1:	off	on	off	on	off	off	on	on
SW2:	off	off	off	off	off	off	off	off

Configuring the Disk Drive

You can order Priam Diskos Model 665Ø-1Ø disk drives from the factory, equipped with an ANSI standard adapter. Refer to the Priam documentation if you have a drive that requires installing this adapter.

Modify the factory-default configuration of the Priam 665Ø-1Ø disk drive by setting the Sector DIP switch (located under the adapter board) for 18 sectors per track according to the settings shown in Table 5-7.

Table 5-7. Disk Drive Switch Settings.

Switch Settings

Bit:	1	2	3	4	5	6	7	8
Sector Switch:	off	on	off	off	on	off	off	off

Caution: Make sure you have unlocked the disk heads and spindle before turning on the disk power. New Priam disk drives have a label indicating which lever performs this function. Old drives have two levers; one releases the heads and the other releases the spindle. Refer to the manufacturer's instructions for more information on preparing the disk drive for use.

Configuring the Terminal

Installing the terminal requires configuring both the hardware and software. The steps involved depend on the terminal you are using. This section describes the steps for the terminals that RCS supports.

To configure the hardware, set up the terminal to communicate with RCS. The computer port on the terminal must be configured for RS-232C, 9600 baud, 8 data bits, no parity, and one stop bit. Refer to the terminal manual for information on how to set up the terminal. If you have Micro-Term ERGO 3Ø1, the function key format must be set to TV970.

To configure the software, modify the system to load the proper block for the terminal being used and define the programmable function keys (if any). See Steps 6 and 7 in Section 5.5, "Software Installation Procedure", for the exact steps involved.

Your NBS-supplied system tape is configured for an ANSI terminal, specifically the Micro-Term Model ERGO 3Ø1. The Micro-Term has programmable function keys and tabs. The RCS VT1ØØ utility contains additional words to program these keys. Similarly, the RCS TeleVideo utility contains words to program the TeleVideo function keys. Tables 5-8 and 5-9 list the programmable function keys as defined for these two terminals. Note that the Datamedia 1521A does not have function keys.

Note: Terminals that are compatible to VT1ØØ terminals conform to the ANSI standard for terminals. You can use any ANSI standard terminal with the RCS VT1ØØ screen editor. However, most terminals add features on top of the ANSI standard, such as programmable function keys and tabs. Therefore, the protocol for programming function keys and tabs differs from one terminal to another, even if both terminals are ANSI. If you do not have a Micro-Term or TeleVideo terminal, you have to write your own words to program the function keys for your terminal.

Table 5-8. Function Keys Defined on the Micro-Term ERGO 3Ø1.

Key	Function	SHIFT-Key Function	CTRL-Key Function
F1	printer on	X	printer off
F2	form feed	X	X
F3	^X	X	X
F4	move to TASK&PATH board	X	X
F5	move to PRIM board	X	X
F6	move to JOINT/COMM board	X	X
F7	X	X	X
F8	X	X	X
F9	X	boot commands*	X
F1Ø	X	2 D>M	X
F11	X	3 D>M	X

Table 5-8. Function Keys Defined on the Micro-Term ERGO 3Ø1. (continued)

Key	Function	SHIFT-Key Function	CTRL-Key Function
F12	X	5 D>M	X
F13	X	X	X
F14	X	X	X
F15	X	X	X
F16	^E	X	X
PF1	home	X	X
PF2	clear	X	X
PF3	X	X	X
PF4	X	X	X

*The boot commands are HEX F7 C9 Ø CBOOT Ø MBBOOT init-cm. SHIFT-F9 does not include a carriage return.

Note: The keys PF1 through PF4 are not programmable. The keys F1 through F16 are programmed in the RCS Function-Keys block. The SHIFT-Key sequences are programmed in EAROM.

For the VT1ØØ, pressing ^<ENTER> toggles the selection of the printer.

Table 5-9. Function Keys Defined on the TeleVideo 95Ø.

Key	Function	SHIFT-Key Function	CTRL-Key Function
F1	printer on	X	X
F2	printer off	X	X
F3	printer form feed	X	X
F4	move to TASK/PATH board	X	X
F5	move to PRIM board	X	X
F6	move to JOINT/COMM board	X	X
F7	X	X	X
F8	X	X	X
F9	X	X	X
F10	boot commands	X	X
F11	2D>M	X	X

Configuring the Printer

Connect the printer to the printer port on the terminal you are using. Make sure the printer and terminal have the same port configurations, such as baud rate and stop bits. Consult the printer and terminal manuals for more information. If your terminal does not have a transparent mode (where the terminal passes all control codes through to the printer), some of the RCS printing utilities will not work. In this case, you need to write a small amount of FORTH code to communicate with a separate printer port. The TeleVideo 95Ø and VT1ØØ terminals have transparent modes.

In addition to configuring the ports on the printer and terminal, it is necessary to modify the system for the particular printer being used. The steps involved are given in Section 5.5 in Step 8.

Preparing to Install the RCS Software

After configuring the boards, the disk, the printer, and the terminal, install them and any other components of the system according to the manufacturer's instructions. Although different applications require different configurations of the system hardware, you must have at least the minimum hardware configuration defined in Section 5.1, "Hardware Requirements", to install the RCS system on a blank disk. If you want to install the full RSL application, you must have three additional 86/3Ø processor boards configured as shown in Table 5-2 (board numbers 2, 4, and 6).

5.5 SOFTWARE INSTALLATION PROCEDURE

You install the RCS and RSL software by transferring it from the NBS-supplied magnetic tape to your Winchester disk, and then compiling it on your system. To install the software, follow this procedure:

Note: If you make an error in typing any of the required input statements, use the "delete" key to backspace over the error and correct it. Do not use the "backspace" or cursor control keys to correct an error.

1. Make sure the disk heads are unlocked. Then turn on the system and verify that the disk is spinning.

2. At the terminal, type the first digit of the master board address (that is, the board number) followed by a space and the word CBOOT. For example type RPB 4 ERASE Ø CBOOT and press RETURN.

   After you press RETURN, the system initializes the disk semaphore and control block for that board. Then the system loads the heads on the disk to prepare it for use and displays the FORTH **ok** prompt.

3. Formatting the disk takes approximately 3 minutes. To format the disk, enter the following commands exactly as shown:

```
HEX
RPB 1C ERASE
24 RPBPREP
Ø4ØØ RPB 4 + ORM
4 RPB ØA + !
RIMFIRE-GET RDOP CØ8Ø FFFF ?WERROR RIMFIRE-FREE
DECIMAL
```

Refer to the Rimfire controller manual for information on any error messages that appear during the formatting process. Repeat the commands listed in this step if formatting does not proceed as expected. If the problem persists, make sure that the addressing jumpers and switches are correctly positioned on the memory board, the processor board, and the Rimfire controller board.

4. To rewind the tape, move the tape past the first file mark, and read in the MBOOT file, load the RCS system tape into the tape drive and enter the following commands exactly as shown:

```
TREWIND
TSKIP
1ØØØ 1349 88 T/F-TAPE
```

Note: Appendix B, "User Word Summary", does not contain these words because they are used only for this initial installation.

5. If the commands listed in Step 4 execute correctly, enter the following commands to load MBOOT and the normal TAPE utility, and to rewind the tape. Enter the commands exactly as shown, but substitute the [#] with the first hex digit of the starting address for the 86/3Ø processor board designated for RSL. If you configured your system as described in this chapter, the first hex digit of the starting address for the RSL board is Ø.

```
[#] MBOOT
Basic TAPE
TREWIND
```

Now load the rest of the source code from the tape. After each FROM-TAPE command, the specified blocks are read from the tape, and the message **tape is at file #** (where # is the current file number on the tape) is displayed. If an error message appears, do not execute the rest of the commands (see Section 6.10, "Using Tape Utilities", for information on solving the tape problem). To load the source code, enter:

```
2 TGOTO
135Ø 1499 FROM-TAPE
15ØØ 1599 FROM-TAPE
7ØØØ 7999 FROM-TAPE
8ØØØ 8999 FROM-TAPE
9ØØØ 9999 FROM-TAPE
1ØØØØ 1Ø999 FROM-TAPE
11ØØØ 11999 FROM-TAPE
12ØØØ 12999 FROM-TAPE
```

After each FROM-TAPE command, the specified blocks are read from tape, and the message **tape is at file #** will be displayed, (where # is the current file # on the tape). If any error messages appear, do not execute the rest of the commands (see Section 6.10, "Using Tape Utilities").

6. Create the tape and print D>M by pressing the master reset switch and entering the following commands on the master board:

```
HEX F7 C9 OUTPUT
RPB 4 ERASE
Ø CBOOT
Ø MBOOT
```

Depending on the type of terminal being used, enter:

```
[#] CUSTOM
```

Where [#] is 7 for Datamedia, 8 for TeleVideo, and 9 for VT1ØØ-compatible terminals. Note that after you enter [#] CUSTOM, the system prompt changes to **list&tape>**.

Now enter:

```
Ø OFFSET !
5 MEM>DISK
```

7. If you are using a Micro-Term ERGO 3Ø1 terminal, skip to Step 8. If you are using either a TeleVideo or Datamedia terminal, you must reconfigure the system.

For the TeleVideo 95Ø, perform the following steps:

- In block 137Ø, the Ø CUSTOM block, use the screen editor to change VT1ØØ in line 4 to Televideo.

- Change VT1ØØ-Terminal in line 6 to Televideo-terminal.

- Enter the following commands to copy block 1166 to block 14Ø5:

```
1ØØØ OFFSET !
166 4Ø5 1 BLOCKS
```

For the Datamedia, perform the following steps:

- In block 137Ø, the Ø CUSTOM block, use the screen editor to change VT1ØØ in line 4 to Datamedia.

- Change VT1ØØ-Terminal in line 6 to Datamedia-terminal.

- In block 14Ø5, insert %% at the beginning of line Ø. Inserting %% disables the programming of function keys since the Datamedia 1521A has none.

Note: For information on using the screen editor, see Section 6.7, "Editing a Block of Code".

8. If you are using an Imagewriter printer, skip to Step 9. If you are using either a DECwriter III or Integral Data Systems Paper Tiger printer, you must reconfigure the system. For the DECwriter III, change Image-writer to Dec-writer in blocks 1377, 1378, and 1379. For the Paper Tiger, change Image-writer to Paper-tiger in these three blocks.

9. Create the base system by pressing the master reset button to turn the hardware on and off, and then entering the following commands to the master 86/30 processor board. (If the system does not have 24-bit addressing the first command is not needed.) After you enter RCS, the system responds with a variety of loading messages, followed by a **:R** prompt. After you enter 1471 LOAD, the system prompt becomes **:R rsl>**.

```
HEX F7 C9 OUTPUT
0 CBOOT
0 MBOOT
init-cm
RCS
1471 LOAD
```

After you enter these commands, the system loads 0 CUSTOM and performs the 1 PRESERVE and 1 PRESERVE-FILE commands. CUSTOM loads a block that customizes the board for a specific application (RSL on the tape supplied by NBS). PRESERVE writes the current system dictionary to disk. PRESERVE-FILE writes the current user files to disk.

10. As prompted, enter the 1 MEM>DISK command. This command writes an image of the board's local RAM to the disk. Then enter Y at the **make image?** prompt.

11. A system running RSL requires four processor boards. Other applications may require more or fewer processor boards. If you are installing equipment other than that listed for a system running RSL, you must edit blocks 1472 through 1479 to reconfigure the base system to accommodate these components. For information on how to edit blocks, see Section 6.7, "Editing a Block of Code".

12. Enter the following command to load each slave board through CUSTOM and to execute the 1 MEM>DISK command:

```
1470 LOAD
```

Loading the base system takes about 14 minutes for the three slave boards in a system configured to run RSL. The previous command causes each slave board to execute the following commands:

```
[#] MBOOT
Slave
REMOTE-SLAVE
RCS
[#] CUSTOM
1 PRESERVE 1 PRESERVE FILE 1 MEM>DISK
```

Where [#] is the first hex digit of the 86/30 processor board address given in Table 5-2.

13. Enter the following command to load each slave board with the corresponding control level of the RSL sample application:

    ```
    1480 0LOAD
    ```

    Loading RSL takes about 7 minutes. If you are using an application other than RSL, you must first define slave address constants in block 1404 (loaded by the master board) and reconfigure BOOT-SYSTEM. Blocks 1460 through 1469 are reserved for booting slave boards; BOOT-SYSTEM loads block 1460.

14. When loading is complete, the system prompts you to enter the following commands on the master board:

    ```
    8000 OFFSET !
    2 MEM>DISK
    Y
    3 MEM>DISK
    Y
    ```

At this point, the RCS and RSL software is installed.

This chapter provides the step-by-step procedures for performing basic RCS operations. The operations include starting up the system; finding, loading, and editing code; saving and rebooting the system; using printer and tape utilities; and shutting down the system.

The procedures in this chapter describe how to use RCS user command words to perform these basic operations. For a complete list of RCS user words, see Appendix B, "User Word Summary".

6.1 STARTING THE SYSTEM

This section describes how to start the system when you have a working application in the 2 D>M area of the disk. A D>M is a 12Ø consecutive block area on the disk. Your system tape comes with the RSL application in the 2 D>M area of memory.

RCS enables you to maintain up to five copies of local memory on the disk. Each copy is called a disk image or D>M. The RCS word D>M enables you to move one of the copies from the disk to memory. Alternatively, MEM>DISK moves a disk image from local memory to the disk. On the system tape provided by NBS, 1 D>M contains the base RCS version 2.2, 2 D>M contains RSL version 1.6, 3 D>M and 4 D>M are free for expansion, and 5 D>M contains additional RCS features such as the editor and the print and tape utilities.

For information on starting the system for the first time, including a more detailed explanation of the start-up command words, see Chapter 5, "RCS Installation Procedures".

Note: If you make an error in typing any of the required input statements, use the "delete" key to backspace over the error and correct it. Do not use the "backspace" or cursor control keys to correct an error.

To start the system, follow these steps:

1. Turn on rack power to the RCS hardware.

2. Press the RETURN key until the terminal responds **ok**.

3. Enter RPB 4 erase Ø CBOOT Ø MBOOT init-cm 2 D>M and wait for the **:R rsl>** prompt. (Remember that the system is case sensitive. You must enter the uppercase and lowercase characters as shown.)

   The commands in Step 3 boot the RSL application. The Ø CBOOT command initializes the Winchester disk, Ø MBOOT loads the base FORTH system, init-cm initializes memory, and 2 D>M moves the application image to memory and executes the auto-load block. The auto-load block initializes the communications process, the 8Ø87 chip, the function keys, the system dictionary, and the SMACRO files. The auto-load block also calls BOOT-SYSTEM to boot the slave boards in the system.

Note: At this point you can use the system for programming and debugging without running the robot.

4. Turn on power to the robot.

5. Prepare the robot for RCS control. The exact procedures for this step depend on the type of robot you are using with RCS. Typically, the steps include turning on power to the robot controller, initializing the controller and the interface to RCS, turning on arm power to the robot, and moving the robot to a safe starting position.

6. Enter GO to start all the control levels and the communications process. RCS is now waiting for the robot to start communications.

7. Command the robot controller to start the RCS interface.

8. Enter ^X to return to the master board. You must press RETURN after ^X.

RCS now controls the robot.

6.2 MOVING FROM ONE BOARD TO ANOTHER

To find a block of code in the memory on a specific board, you must move to that board to search for the block. Moving to a board means that you make the terminal communicate directly with the specified board. RCS enables you to use either software switches or a hardware switch for moving between boards. After you move to the correct board, you must use the correct vocabulary on that board.

The RSL example includes four processor boards. The prompt that appears on the terminal identifies the board with which the terminal is currently communicating. In addition, the first two characters of the prompt indicate the current operating mode of the system.

Table 6-1 lists the boards and their prompts for the RSL example system.

Table 6-1. Board Names and Prompts.

Board Name	Prompt
RSL (the master board)	**:R rsl>**
TASK&PATH	**:R task&path>**
PRIM	**:R prim>**
JOINT	**:R joint/comm>**

Note: The RCS prompts for each board were named with the RSL application in mind. If you add a new application, you can change the prompts to correspond to the control levels of your application. The prompt for each board is defined by the word :OK in the CUSTOM block for that board.

Using Software Switches

The software switch is the code on the master and slave boards that enables you to select the board with which the terminal is communicating. For example, enter [slave board name] REMOTE to make your terminal communicate with a specified slave board. Enter ^X to make your terminal communicate with the master board again.

The NBS-supplied system tape includes code to define the function keys on the Micro-Term terminal. Function keys correspond to each slave board, eliminating the need to type [slave board name] REMOTE. See Table 5-8 in Chapter 5, "RCS Installation Procedures", for the definition of all Micro-Term function keys.

If you have a different terminal, you can customize the function keys for that terminal by editing the block that defines the word Function-keys (absolute block number 1405). If you have defined function keys for the slave boards in your system, you can move between boards by pressing the function key for the board you want.

You must move to a slave board from the master board; you cannot move from one slave board to another. To return to the master board from any slave board, enter ^X.

Using a Hardware Switch

If your system includes a switch box for the terminal, you can communicate with another board by moving the switch to the position corresponding to the appropriate board.

You can also use this switch to perform a soft reset of a board by setting the switch for that board and entering ^C. You can perform a hard reset of the entire system by using the master reset button on the chassis.

Changing Vocabularies

Each board contains a system vocabulary, called SDEF, and up to five user-defined vocabularies. To locate code on the system, you must be on the correct board and in the correct vocabulary on that board. To display the current vocabulary name and size, enter V-SIZE. Enter SDEF to change to the system vocabulary on any board.

The RSL application includes a predefined vocabulary for each control level. These vocabulary names are the same as the names of the RSL control levels: TASK, PATH, PRIM, and JOINT. The word $DEF sets the vocabulary to be the vocabulary of the current board. For example, if you are on the PRIM board, enter $DEF to move to the PRIM vocabulary. Because the TASK and PATH levels are on the same board, RSL defines the words TDEF and PDEF to move between the TASK and PATH vocabularies on the TASK&PATH board.

Alternatively, you can enter [name] DEFINITIONS, where [name] is the name of the desired vocabulary. For example, enter PRIM DEFINITIONS to use the PRIM vocabulary.

6.3 LOCATING SOURCE CODE

Locating source code requires an understanding of the LOC command, the Locate mode, directory block conventions, relative and absolute blocks, and offsets.

This section discusses these topics, including an example of how to locate source code using the RCS directory block system.

Most code creates entries in the system dictionary. This code can be located by name using the LOC command or Locate mode. To locate code which does not make entries in the system dictionary, you must use the directory block system.

Using the LOC Command

When you are on the correct board, using the correct vocabulary, use the LOC command to find the block where a specific word is defined. Enter LOC followed by a space and the word you want to locate. The system displays a listing of the source block in which the word is defined.

Some words cause RCS to issue the message **Pre-locate.** This message indicates that the word is a system word, not a user application word, and that you cannot access it. However, some system words are accessible using the LOC command.

Using the Locate Mode

You can also find a specific block using Locate mode. To use Locate mode, when you are on the correct board, using the correct vocabulary, enter :L. In Locate mode, :L prefixes the system prompt.

Enter the word that you want to locate and the system lists the block that defines that word, instead of executing the word. To exit Locate mode, enter :R. (For more information on RCS modes, see Section 7.4, "Modes".)

Understanding Directory Block Conventions

The RCS disk is organized into sections containing 1ØØØ blocks each. For example, the source code for RCS resides in the 7ØØØ-block section, and the source code for RSL resides in the 8ØØØ-block section. For a complete map showing these blocks, see Appendix C, "System Maps".

All source code on the disk is organized according to the FORTH block structure; a block contains 16 lines of 64 characters each. RCS uses a hierarchical directory block structure to organize these blocks on the disk. A directory block exists for each 1000-, 100-, and 10-block section of disk.

Note that if a block is 0 or a multiple of 10 or 100, the next block (block 1, block 11, or block 101, respectively) is also a directory block that lists blocks at the next directory level. For example, block 11 lists the directory for blocks 12 through 19.

The directory blocks contain comments to help you locate code on the disk. You must maintain directory blocks. When you add, delete, or change code, be sure to update the affected directory blocks.

Using Relative and Absolute Block Addresses

RCS uses a system of relative block addresses to help you locate blocks. The absolute block number is the actual block number on the disk; the relative block number is the absolute block number minus the offset.

Specifying an Offset

The offset is the value of the special FORTH variable OFFSET. The offset is usually a multiple of 1000 that indicates the board with which you are currently communicating. Using an offset decreases the number of characters you must enter to identify a block.

If a board boots with the offset set at 8000, you can access blocks 8000 through 8999 using the relative block numbers 0 through 999.

Note that there may be more than one control level on the same processor board. In this case, the first level resides in blocks 0 through 999, the second level resides in blocks 1000 through 1999, and so forth.

Use !, the FORTH store command, to change the offset. For example, to set the offset at 8000, enter 8000 OFFSET !.

Note: If you reconfigure the disk, you must edit the code in the application-load blocks to change the offsets.

An Example of Using the RCS Directory Block System

This section explains how to use the RCS directory block system to locate a block of code. Figure 6-1 shows the directory blocks for a 1000-block section of the disk.

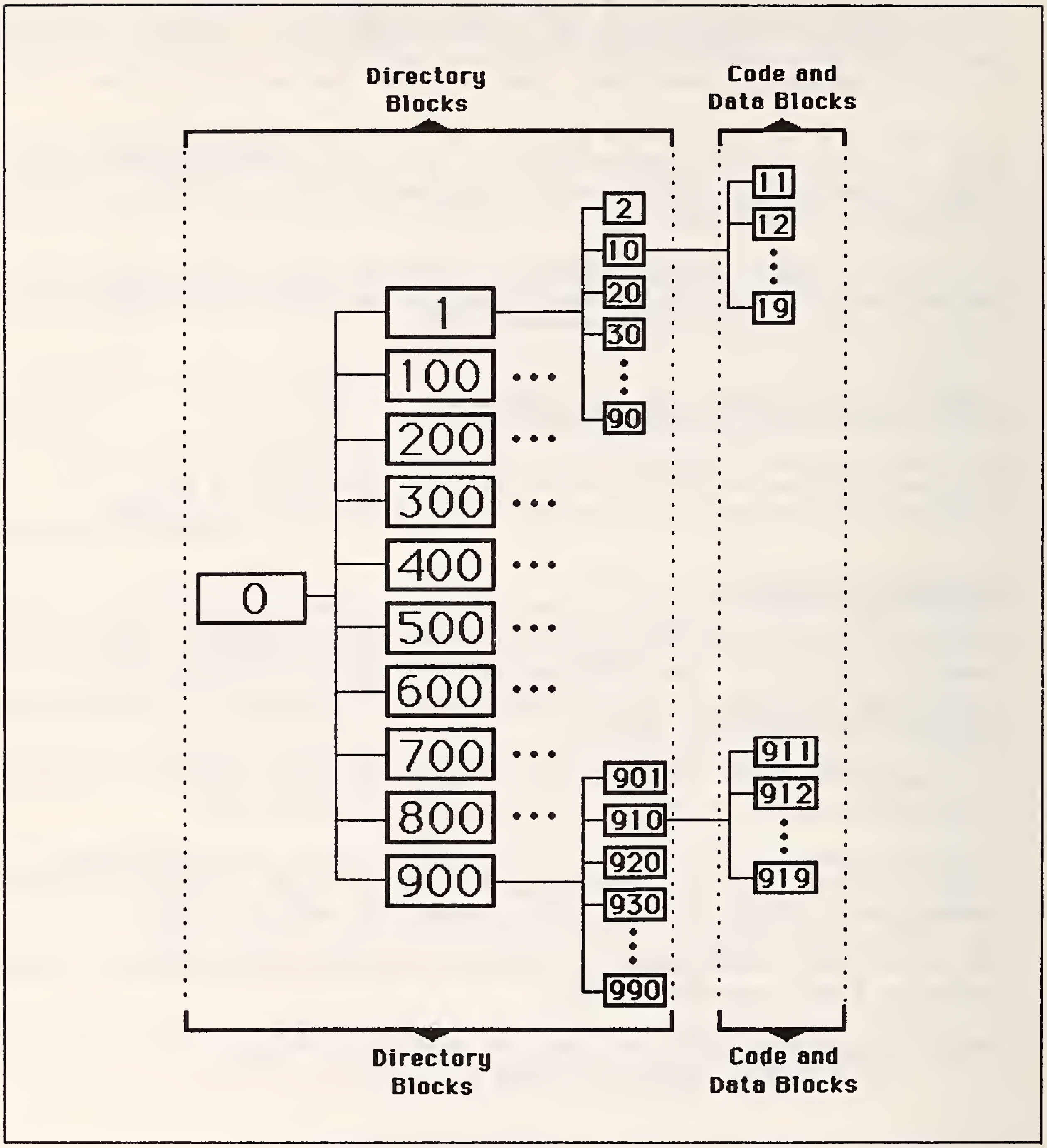


Figure 6-1. Directory blocks for a 1ØØØ-block section of the disk.

You use these directory blocks to locate code; for example, to find the block that contains a TRANSFER task. Tasks are contained in blocks on the TASK level. To find where TASK is located on the disk, boot the system (Steps 1 through 3 in Section 6.1, "Starting the System"), then enter MAP to list the first block of the directory of the disk.

```
-1Ø64Ø      136Ø ABS

22Ø/A366 Development Disk Directory Ø3/Ø5/86
 *******  SYSTEM BLOCKS ************

     Ø -   999   FLOPPY BLOCKS
  1ØØØ -  1349   MBOOT 2.2
  135Ø -  1499   CUSTOM
  15ØØ -  1599   screen EDIT
  16ØØ -  5999   D>M images
  6ØØØ -  6999   TAPE TRANSFER
  7ØØØ -  7999   RCS 2.2
  8ØØØ -  8999   RSL  1.6
  9ØØØ -  9999   TASK 1.6
 1ØØØØ - 1Ø999   PATH 1.6
 11ØØØ - 11999   PRIM 1.6
 12ØØØ - 12999   JOINT 1.6


 :R rsl>
```

As you can see, MAP lists absolute block 136Ø, which shows TASK at 9ØØØ OFFSET. (You can see the rest of the MAP directory block by entering N L.) Enter 9ØØØ OFFSET ! to set the offset to the location of the TASK level. Enter Ø LIST to list the first directory block for the TASK level.

```
Ø
 Ø  ( TASK 1.6 5/19/86  for RSL 1.6b
 1 ------------------------------------------------------
 2      1  load, list, debug
 3    1ØØ  variables
 4    2ØØ  pre-process
 5    3ØØ  command : primitives
 6    4ØØ  command : path search
 7    5ØØ  command : tasks, COMMAND-PROCESS
 8    6ØØ
 9    7ØØ
1Ø    8ØØ  post-process
11    9ØØ  user commands, level, power-up
12 ------------------------------------------------------
13
14
15
:R rsl>
```

Because the TRANSFER command is a user command, you can see on line 11 that TRANSFER should be in the 9ØØ 1ØØ-block section. Enter 9ØØ LIST to display the next directory-level listing.

```
9ØØ
 Ø  ( TASK : user commands, level, power-up
 1 ------------------------------------------------------
 2   Ø1  Display, time
 3   1Ø  user commands
 4   2Ø
 5   3Ø
 6   4Ø
 7   5Ø
 8   6Ø
 9   7Ø
1Ø   8Ø  power-up load block
11   9Ø  TASK LEVEL
12 --------------------------------------------------)
13   ØØ ref-blk
14    Ø1 load ( 1Ø load   2Ø load   3Ø load   4Ø load
15    5Ø load   6Ø load   7Ø load   8Ø load ) 9Ø load
:R rsl>
```

You can see on line 3 that user commands are in the 91Ø 1Ø-block section. Enter 91Ø LIST to display the next directory-level listing.

```
91Ø
 Ø  ( TASK :  user commands
 1 ------------------------------------------------------
 2   1  RESTART
 3   2  PAUSE
 4   3  TRANSFER
 5   4  MOVE-TO nul ARRAY-SAFE
 6   5
 7   6
 8   7
 9   8
1Ø   9  single-step
11 ------------------------------------------------------)
12   Ø  ref-blk
13
14 (  1 load   2 load   3 load   4 load   5 load
15    6 load   7 load   8 load   9 load )  ~MEM
:R rsl>
```

Finally, you can see on line 4 that TRANSFER is in block 913. Enter 913 LIST to display the TRANSFER command block.

```
913
 Ø
 1 :R
 2   inc-command-#-in ~@ 1+ ==> inc-command-TASK DEFINITIONS#-in
 3   '' TRANSFER nul Ø1 arr ARRAY ØØ Ø8 ;; loc CONVEYOR "
 4     ==> input-command
 5 :S
 6
 7
 8
 9
1Ø
11
12
13
14
15
:R  rsl>
```

6.4 LOADING CODE

Loading code in RCS is similar to loading code in FORTH. In fact, in RCS, the word "load" means the FORTH LOAD operation, which reads a block from the disk to local RAM, and then interprets all the words in the block. In RCS, the SMACRO compiler interprets or compiles the words in a block.

The SMACRO compiler in RCS is an incremental compiler. It compiles, links, and loads code in small pieces, typically one routine at a time. No separate linker and loader exists as in other systems; the code is completely linked and ready to run when the compiler finishes with it. The SMACRO compiler is called by the SMACRO defining words, mainly the word "routine" and variable declarations. Thus, to compile code, you simply load the blocks containing it.

Note that loading a block does not necessarily result in compiling code. The block may contain words that perform other tasks, such as initializing ports or executing routines. Blocks used this way resemble command files on other systems.

The SMACRO compiler also enables you to redefine routines. After a complete system is loaded, you can edit and reload (recompile) any routine at any time, without reloading or even relinking the rest of the system. Any routine that calls the edited routine now calls the new version. Combined with the block screen editor, the ability to redefine routines is extremely useful for debugging, because bug fixes can be tested with very short edit and compile cycles.

SMACRO is also interpretive in that you can execute any routine by typing its name, and you can view and set variables. You cannot execute statements or fragments in this way, only compiled routines. This restriction retains the speed of compiled systems, while providing the user-friendliness of interpreted systems.

To load a block, enter the relative block number followed by the word LOAD, or the absolute block number followed by the word ØLOAD. You can save the time required to reload code by saving the FORTH and SMACRO segments (the D>M) and the system dictionary to disk, using the MEM>DISK and PRESERVE commands.

The following procedures describe two special situations in which you need to load code: reloading an edited routine and loading code still under development.

Editing and Reloading a Routine

To edit and reload a routine, follow these steps:

1. Boot the system as described in Section 6.1, "Starting the System".

2. Edit the routine as described in Section 6.7, "Editing a Block of Code".

3. Move to the board containing the routine.

4. Use the command LOAD or ØLOAD to reload the source code for the routine.

5. Enter [preserve#] PRESERVE [preserve#] PRESERVE-FILE, where [preserve#] is the number of the PRESERVE area containing the final application, to save the system dictionary and your files.

6. Enter [d>m#] MEM>DISK, followed by a comment, such as the date and a description of the changes you made. The comment must begin with a double quotation mark.

   The MEM>DISK command saves the FORTH and SMACRO segments that are on the board. For example, 2 MEM>DISK " Added code for new sensor, saves the 2 D>M image to disk.

(This procedure is illustrated in Section 6.8, "Example RCS Dialogue".)

Loading Routines Under Development

The procedure for loading routines under development is similar to the procedure for reloading a routine. However, because an application under development may contain programming errors, you should load it in segments. After you load each segment of the application and ensure that the code works properly, save the application and the system dictionary to the disk, using the D>M and PRESERVE commands.

To load an application under development, follow these steps:

1. Make the board containing the application the current board.

2. Enter [d>m#] D>M to copy the working Forth and SMACRO segments from the disk into memory, where [d>m#] is the last image saved.

3. Enter [Preserve#] RESTORE [preserve#] RESTORE-FILE to copy the working system dictionary and SMACRO files from the disk into memory.

4. Load the code under development. If the system detects an error, RCS aborts the loading procedure and you must edit the block to correct the error.

   An error may be nonfatal or fatal. If the error is nonfatal (for example, a misspelled variable name), simply reload the block. If the error is fatal (for example, not allowing enough space for a variable owner), reload the application using the D>M, RESTORE, and RESTORE-FILE commands.

   At this point, all of the code is loaded. However, memory space may be wasted as a result of nonfatal errors. To recover this wasted space, repeat Steps 2 through 4 before continuing to Step 5.

5. Enter [preserve#] PRESERVE [preserve#] PRESERVE-FILE, where [preserve#] is the number of the PRESERVE area containing the final application, to save your files and the system dictionary.

6. Enter [d>m#] MEM>DISK, followed by a comment, to save the updated FORTH and SMACRO segments to disk.

Using this procedure, you can restore a system quickly without reloading a list of blocks.

6.5 EXECUTING TASKS AND ROUTINES

With RCS, you can execute robot tasks with an operating robot, or the routines that define that task without an operating robot. RCS also enables you to operate in different modes, control output to the screen, and abort an executing task.

Executing Robot Tasks

To execute a robot task, load the block that defines that task. For example, if the RSL transfer task is defined in block 913 on the TASK board, enter Task/path REMOTE to move to the TASK&PATH board, and then enter 913 LOAD to execute a transfer in which the robot transfers an object to a different location.

Executing Routines

Executing routines without an operating robot is useful when you are debugging an application. First, issue a HALT command to stop all control levels, the communication process, and execution of the robot interface. Then assign values to the input variables needed by the routine, and enter the name of the routine to execute it. If the routine includes many input variables, you may want to set up a load block to assign values to the input variables. This procedure is illustrated in the example in Chapter 13, "Debugging Techniques".

Understanding Modes

RCS includes four operating modes: Run, Show, Locate, and Compile. In Run mode, the system executes routines previously compiled. In Show mode, the system displays on the screen the contents of variables. In Locate mode, the system displays the block defining the word you enter. Section 7.4, "Modes", explains how these modes affect SMACRO words. RCS uses Compile mode when you load a block containing defining words.

Controlling Screen Output

To suspend output to the terminal, press ^S. To resume output to the terminal, press ^Q.

Aborting an Executing Task

To send a soft reset to the board to which the terminal is currently connected (usually the master board), press ^C. If you are using a software switch to communicate with a slave board, pressing ^C aborts the software switch and returns you to the master board. The executing task on the slave board is not aborted. To abort a task on a slave board, use the switch box to access that board and then press ^C.

6.6 SAVING AND REBOOTING THE SYSTEM

When you change existing code or create new applications, you usually want to save the changes or the new code to the disk. You may also need to recreate the base system to reflect hardware changes. This section describes the procedures for performing these operations.

To understand these procedures, you need to understand the structure of RCS described in earlier chapters of this manual. RCS stores compiled code, variables, and routines in the local memory of each processor board. Common memory contains the system dictionary and user files. RCS also stores compiled code on the disk in D>Ms, PRESERVEs, and PRESERVE-FILEs, which contain copies of the compiled code from local and common memory.

Figure 6-2 shows the action of the RCS words D>M, MEM>DISK, PRESERVE, RESTORE, PRESERVE-FILE, and RESTORE-FILE, used for saving and restoring copies of code.

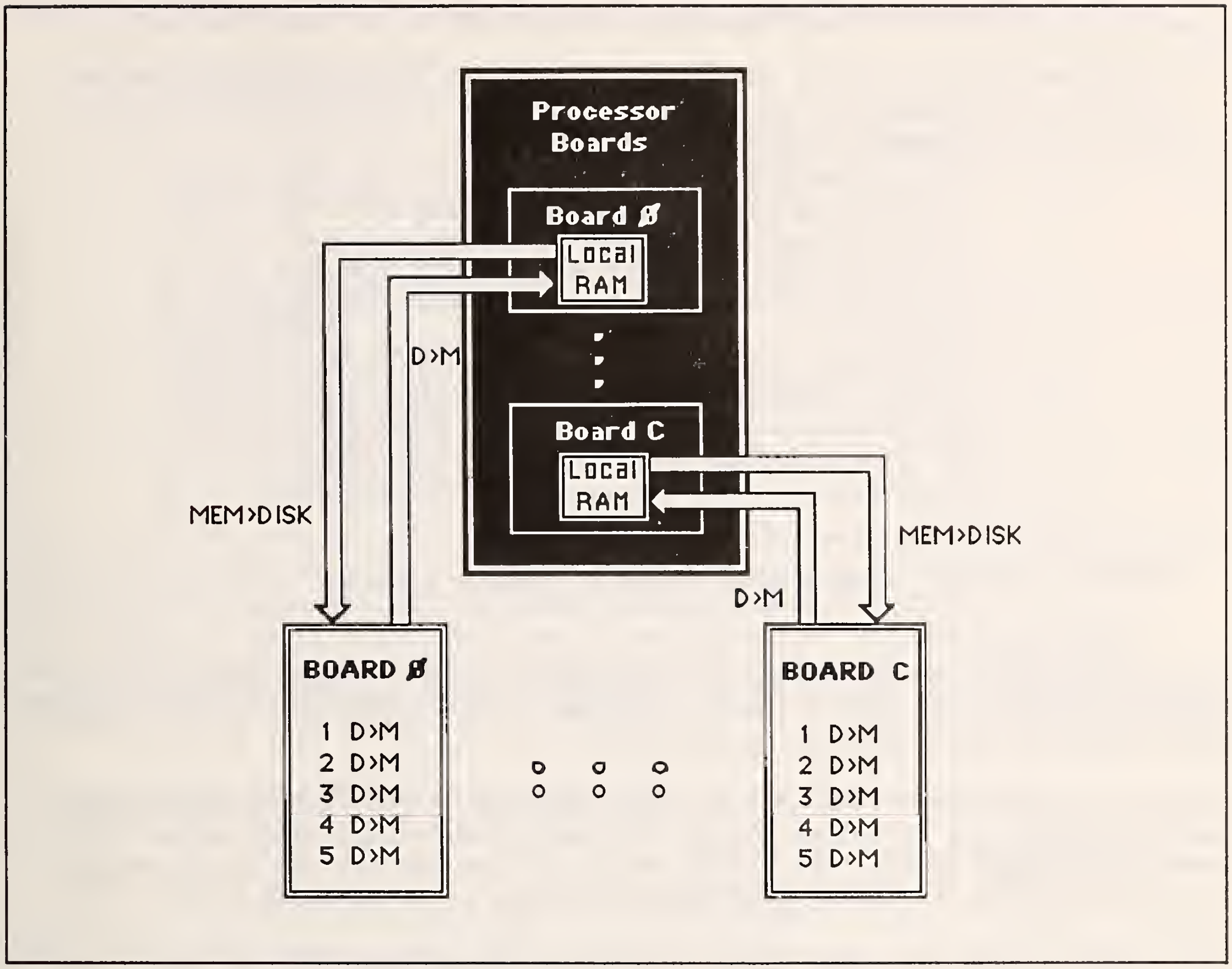


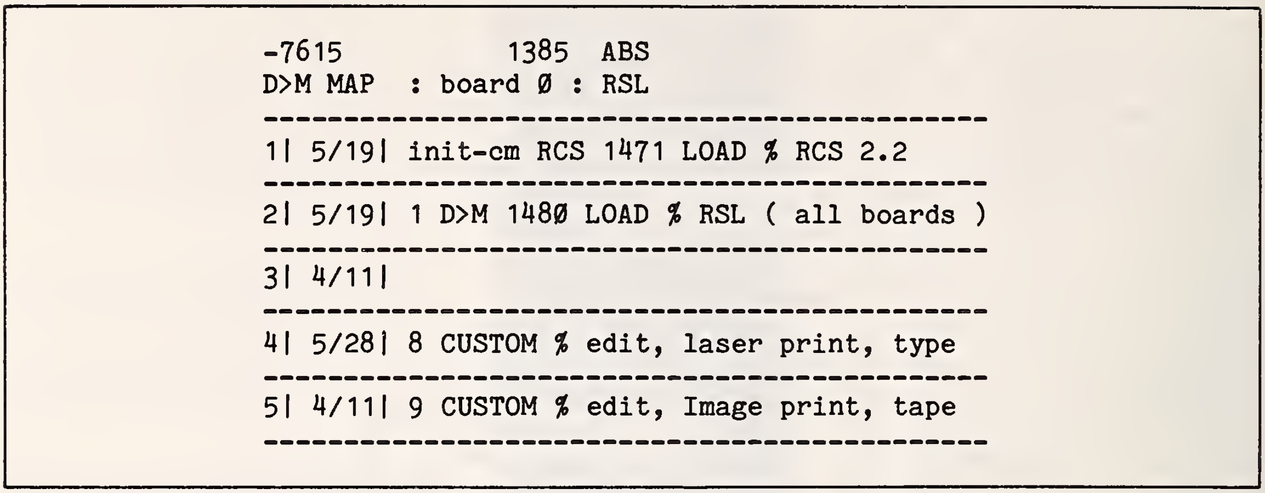
Figure 6-2. Restoring and saving code.

Saving and Restoring Disk Images

A D>M is the name given to a disk area of 12Ø consecutive blocks containing the FORTH and SMACRO segments for a processor board. A D>M is a disk image of the contents of that 12Ø-block area. D>M is also an RCS word that moves the data in the D>M to local memory.

The disk includes space for seven boards, each with five disk images (D>Ms). Enter [board#] DM? to list the D>Ms on a board. This listing is called the D>M map block. The word MBOOT lists the D>M map block for board Ø, which you see every time you start RCS.

Figure 6-3 shows the D>M map block for board Ø of the RSL application.



```
-7615           1385  ABS
D>M MAP  : board Ø : RSL
---------------------------------------------
1| 5/19| init-cm RCS 1471 LOAD % RCS 2.2
---------------------------------------------
2| 5/19| 1 D>M 148Ø LOAD % RSL ( all boards )
---------------------------------------------
3| 4/11|
---------------------------------------------
4| 5/28| 8 CUSTOM % edit, laser print, type
---------------------------------------------
5| 4/11| 9 CUSTOM % edit, Image print, tape
---------------------------------------------
```

Figure 6-3. Example D>M map block.

In this D>M map, you create the 1 D>M image by entering the commands summarized on line 1. To create the 5 D>M image, enter 9 CUSTOM. You must update this block for your specific application.

The RCS word D>M moves the data in that D>M area on the disk to local memory, and then loads the auto-load block. A reserved auto-load block exists for each D>M on each board. The auto-load block executes all the words necessary to start the D>M, including initializing hardware and software.

For example, when you enter 2 D>M to boot an application, RCS copies the 2 D>M image from disk to local RAM and the auto-load block executes, restoring the system dictionary and SMACRO files from the disk.

The RCS word MEM>DISK copies the contents of the on-board RAM into a D>M space on the disk, saving the current segments. Any words you enter on the same line after the MEM>DISK are executed both after the MEM>DISK word and after the corresponding D>M is executed. Enter words after the MEM>DISK command to add comments to describe the D>M.

The following dialogue illustrates D>Ms, input lines, and auto-load blocks:

```
:R rsl>Joint/comm REMOTE
```

Switch to the Joint/comm board.

```
:R joint/comm>2 AUTO?
```

Display the auto-load block for the 2 D>M on the Joint/comm board.

```
    -10564 1436 ABS

    % board 6 2 D>M
    CR " Joint/comm level  "
    " NO RESTORE "
    TIME-OUT-INIT 87INIT
     HEX 7FFF ==> control-cycle-#-clks DECIMAL
     RESTART-COMM-TIMER
    CR " Joint init " 12000 980 OLOAD
    CR " Joint/comm inited "
    CR
```

When you load this block, it prints the message **Joint/comm level NO RESTORE**, initializes the time-out and the 8087, initializes the variable control-cycle-#-clks, starts the communications timer, prints the message **Joint init**, loads block 12980 absolute, and prints the message **Joint/comm inited.** Block 12980 loads other blocks that initialize several variables for the JOINT level.

```
:R joint/comm>2 MEM>DISK "hi there-this is the input line"


make image? y
```

Make a new 2 D>M; it overwrites the old 2 D>M. Confirm the command.

```
Joint/comm level NO RESTORE
Joint init
Joint/comm inited
hi there-this is the input line
```

RCS loads the auto-load block when MEM>DISK executes. The auto-load block displays the message on the input line.

```
:R joint/comm>2 D>M
```

Executing 2 D>M now reads into RAM the copy just written, loads the auto-load block, and displays the message on the input line.

```
Joint/comm level NO RESTORE
Joint init
Joint/comm inited
hi there-this is the input line
```

All essential start-up procedures should reside in the auto-load block. Use the D>M input line only for comments.

Saving and Restoring the System Dictionary

Use the words PRESERVE and RESTORE to move the system dictionary between common memory and the disk. PRESERVE moves the system dictionary from common memory to the disk, and RESTORE moves the system dictionary from the disk to common memory. The disk contains space for nine copies of the system dictionary. The base RCS system dictionary resides in the 1 PRESERVE area.

Saving and Restoring User Files

Use the words PRESERVE-FILE and RESTORE-FILE to save and retrieve SMACRO files. (RCS creates user files using the SMACRO FILE declaration. See Section 7.2, "SMACRO Syntax", for more information.) PRESERVE-FILE saves your files to the disk, and RESTORE-FILE retrieves them. The disk contains space for nine copies of your files.

Note: Be careful when using more than one version of an application. If you do not enter the PRESERVE, PRESERVE-FILE, and MEM>DISK commands after making changes to the system, the versions in the D>M and the source code will differ.

6.7 EDITING A BLOCK OF CODE

RCS provides line-editing commands and a screen editor. Both these editing features are loaded in the 2 D>M and 5 D>M on the master board. You have to be on the master board to use the screen editor.

To edit the current block of code using the screen editor, enter ED. To edit any other block, enter [block#] ED. RCS displays the block and puts you in the screen editor. In the editor, a block consists of 16 lines of 64 characters each. The screen editor does not recognize characters that exceed the FORTH block boundaries. Be careful not to lose characters off the end of a line or the end of a block.

The source code on your supplied tape is spread out over many blocks. The empty blocks are provided so you can expand the system. If new or added code does not fit in a block, use editing commands to move part of the block to the next block and load both blocks together. Do not put a vocabulary name in the continuation block. This technique is shown for an example SMACRO routine in Figure 6-4.

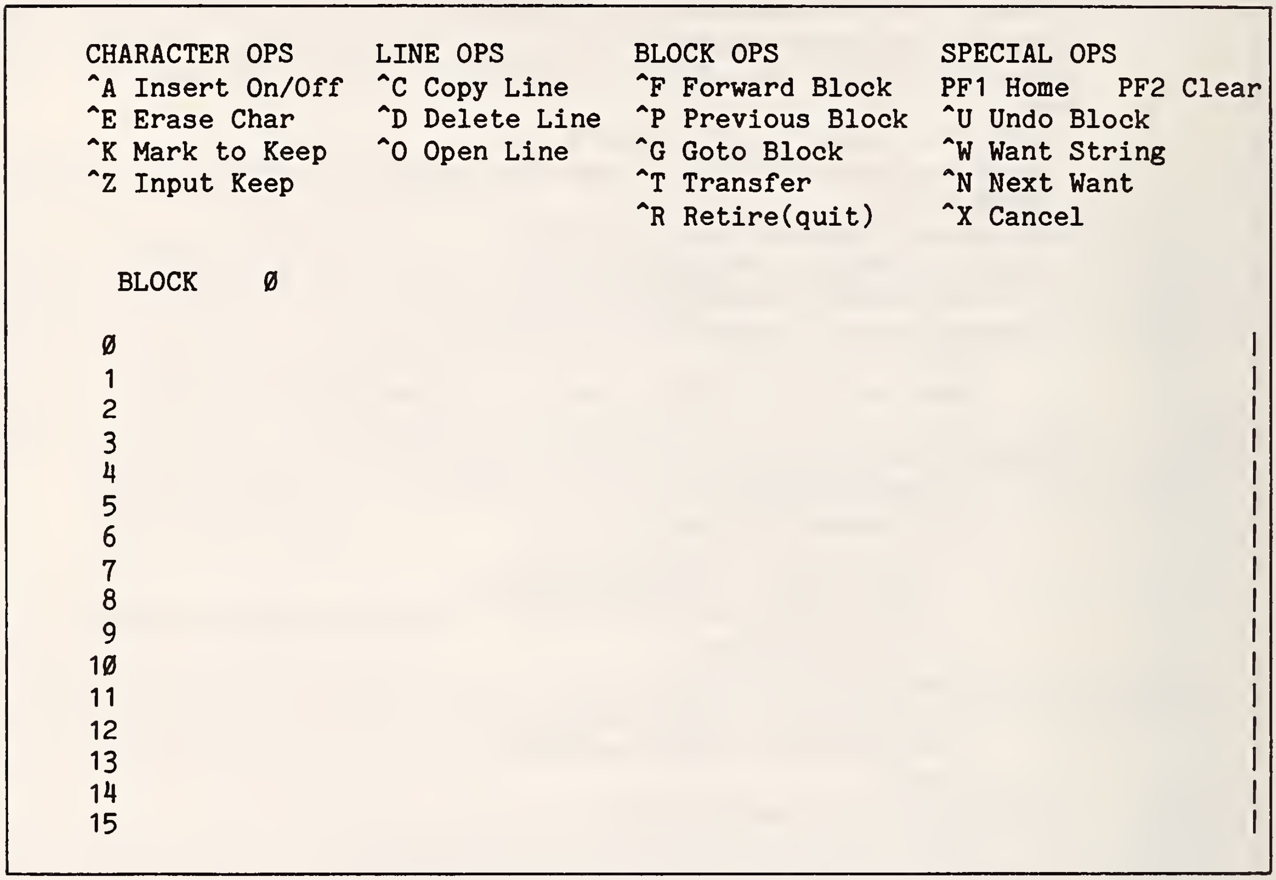
```
                          JOINT DEFINITIONS
routine CARTESIAN
 %  Input:  pose-^-in, scaling-var
 %  Output: servo-com-joint, scaling-var
 if servo-status (EQ) error
 then error => status-report lower-level => status-arg-out
 else
  POSE-FILE pose-^-in => record#
  retrieve-from-fields pose conf-flag  pose
  REACH-CHECK
  if status (EQ) noerror
  then CART>JOINT  CART-CONFIGURE
   if status (EQ) noerror
   then 1.# .=>. j-acc 1.# .=>. j-vel
    SCALE
    servo-point => output-command inc-command-#-out INC

( CARTESIAN cont )
     JOINT-LIMIT-TEST
     is status (EQ) error
     then error => status-report
      status-arg => status-arg-out
      old-ja S=> servo-com-joint
     else CART-STATUS
     endif
    else error => status-report
     status-arg => status-arg-out
    endif
   else error => status-report
    status-arg => status-arg-out
   endif
  endif
end-routine
```

Figure 6-4. Using continuation blocks.

Using Screen-Editor Commands

To move the cursor in the screen editor, use the four arrow keys. If you hold down an arrow key, the key repeats until you release the key. The cursor never leaves the block; if the cursor reaches the end of a line, it wraps around to the beginning of that line. The HOME key moves the cursor to the top left corner of the block. All screen-editor commands use relative block numbers.

When you enter the screen editor, RCS displays the four categories of editing commands, as shown in Figure 6-5. The vertical lines on the right of the figure indicate the end of the 64-character line.



```
    CHARACTER OPS      LINE OPS         BLOCK OPS           SPECIAL OPS
    ^A Insert On/Off   ^C Copy Line     ^F Forward Block    PF1 Home    PF2 Clear
    ^E Erase Char      ^D Delete Line   ^P Previous Block   ^U Undo Block
    ^K Mark to Keep    ^O Open Line     ^G Goto Block       ^W Want String
    ^Z Input Keep                       ^T Transfer         ^N Next Want
                                        ^R Retire(quit)     ^X Cancel

      BLOCK    Ø

     Ø                                                                          |
     1                                                                          |
     2                                                                          |
     3                                                                          |
     4                                                                          |
     5                                                                          |
     6                                                                          |
     7                                                                          |
     8                                                                          |
     9                                                                          |
    1Ø                                                                          |
    11                                                                          |
    12                                                                          |
    13                                                                          |
    14                                                                          |
    15                                                                          |
```

Figure 6-5. RCS screen-editor display for the VT1ØØ.

The control characters listed on the screen are for the VT1ØØ. Other terminals may have other control characters that perform the same function. The commands the control characters perform are listed on this screen. Table 6-2 describes these editing commands.

Note: The screen editor must be configured for the terminal being used. See Section 5.5, "Software Installation Procedure", for the procedure.

Character operations

In the screen editor, you can edit characters using the four character-editing commands shown in Table 6-2.

Table 6-2. Character-Editing Commands.

Command	Description
^A (Insert On/Off)	Inserts characters when on and writes over characters when off. After you press ^A, *** INSERT *** appears at the top of the screen, and any characters you enter are inserted at the cursor position. When you press ^A again, the insert indicator leaves the screen and any character you enter replaces the character at the cursor position. **Caution:** Remember not to shift characters past the end of the line.
^E (Erase Character)	Erases the character at the cursor position and moves the remaining characters in the line one space to the left.
^K (Mark to Keep)	Marks code to be copied into a Keep buffer that you can copy to other locations within the block or to another block. Use ^K to delimit the string you want to copy to the buffer. Position the cursor on the first character to include and press ^K once. *** KEEP *** appears at the top of the screen. Then move the cursor to the first position after the last character to include and press ^K again. If the last character you want to save is the last character in the line, position the cursor at the first character in the next line and press ^K.
^Z (Input Keep)	If the insert indicator is on, inserts the string contained in the Keep buffer starting at the cursor position. If the insert indicator is off, enters the string in the Keep buffer at the cursor position, overwriting the characters already there.

Line operations

In the screen editor, you can edit lines using the three line-editing commands shown in Table 6-3.

Table 6-3. Line-Editing Commands.

Command	Description
^C (Copy Line)	Moves all lines after the line containing the cursor down one line, and copies the line containing the cursor on the resulting blank line. The last line of the block is lost; it does not move to the next block or wrap around to the top of the block.
^D (Delete Line)	Deletes the line containing the cursor and moves all subsequent lines up one line, leaving the last line in the block empty.
^O (Open Line)	Moves the line containing the cursor and all subsequent lines down one line, providing a blank line for new input. The last line of the block is lost. If you want to insert more lines than the block can hold and the next block is empty, use the Keep buffer or the Transfer command to copy part of the block into the next block. Make sure both blocks are loaded together.

Block operations

You can save and move code between blocks using the five block-operation commands shown in Table 6-4.

Table 6-4. Block-Operation Commands.

Command	Description
^F (Forward Block)	Saves changes to the current block and displays the next block. The next block appears in the screen editor, with the cursor at the same position as it was in the saved block.
^P (Previous Block)	Saves changes to the current block and displays the previous block. The previous block appears in the screen editor, with the cursor at the same position as it was in the saved block.

Command — Description

^G (Goto Block)

Displays the block you specify. When you enter ^G, the system displays a **Go To Block** _____ prompt at the bottom of the screen, requesting a block number. After you enter the relative block number, the current block is saved and the specified block appears in the editor.

^T (Transfer)

Transfers lines within a block or from one block to another. Start the transfer operation with the cursor at the destination location.

When you enter ^T, the system displays the prompt **Transfer Block** _____. Enter the relative block number of the block containing the information you want to transfer. The system then displays the prompt **From Line** _____. Enter the number of the first line you want to transfer. Then respond to the **To Line** _____ prompt with the number of the last line you want to transfer. The transfer lines overwrite existing characters in the destination block.

For example, to transfer lines Ø through 5 from block 99 to block 1ØØ, display block 1ØØ and place the cursor at the position where you want the transferred information to appear. Then press ^T, and respond to the three prompts with 99, Ø, and 5, respectively.

Caution: Insert mode should be off. Transfer lines overwrite existing characters in the destination block.

^R (Retire)

Saves the current block and exits the screen editor. The message **Bye** appears on the screen, and you return to the master board.

Special operations

Table 6-5 describes the commands you can use for special editing operations in Screen Edit mode.

Table 6-5. Special Editing Commands.

Command	Description
PF1-HOME	Moves the cursor to the home position in the block.
PF2 (Clear)	Erases the contents of the block. If you press the CLEAR key unintentionally, enter ^U to undo the Clear operation.
^U (Undo Block)	Restores the screen to its condition before the last Save operation. Note that several editing commands save the contents of a block before performing their assigned functions. For example, ^F saves a block before moving to the next block.
^W (Want String)	Enables you to search for a string of up to 2Ø characters. After you press ^W, the system displays the prompt **Search Input** _____. Enter the search string. After you enter the search string, the system displays the **From Block** _____ and **To Block** _____ prompts, enabling you to specify beginning and ending block numbers for the search. After you answer these prompts, the system displays the first block in the specified range that contains the search string. The characters must match exactly. While the system is searching for the string, **AT** and the number of the block currently being searched appear at the top of the screen.
^N (Next Want)	Repeats the last Want String operation specified using ^W, finding the next occurrence of the search string within the specified range.
^X (Cancel Operation)	Cancels an editing operation. For example, to cancel a transfer operation after pressing ^T, press ^X before responding to all three Transfer command prompts.

Using Line-Editor Commands

You can also edit blocks outside the editor by using the RCS line-editor commands. Common line-editor commands are listed alphabetically in Table 6-6. Line-editor commands enable you to perform operations such as listing a block, or clearing or copying a range of blocks. For a complete list of editing commands, see Appendix B, "User Word Summary".

Table 6-6. Line-Editor Commands.

Command	Description
B	Decrements the current block number.
[source block] [destination block] [# of blocks] BLOCKS	Copies blocks, starting at the source block and continuing for the number of blocks specified, to the destination block. The source and destination blocks may overlap. For example, to copy blocks 1ØØ, 1Ø1, and 1Ø2 to blocks 4ØØ, 4Ø1, and 4Ø2, enter 1ØØ 4ØØ 3 BLOCKS.
[start block] [end block] CLEAR	Clears the specified range of blocks with spaces.

Note: Use the BLOCKS and CLEAR commands with extreme caution, because you can lose large sections of source code with a simple typing error.

Command	Description
[start block] [end block] FIND [string^]	Displays each line containing the string in the specified range of blocks. The display shows the line number in the block at the right margin, along with the start block number on the first line followed by the end block number on the last line.
[start block] [end block] FIND-R [string^]	Replaces all occurrences of the string in the specified block range with the string in the insert buffer. You can load the insert buffer using the command R-WITH, described later in this table.
L	Lists the current block, including the line numbers. N L lists the next block, B L lists the previous block.
[block#] LIST	Lists the specified block, including the line numbers.
[absolute block#] ØLIST	Lists the specified block, including the line numbers. You must specify the absolute block number.
[block#] LLIST	Lists the specified block without line numbers.
[absolute block#] ØLLIST	Lists the specified block without line numbers. You must specify the absolute block number.
N	Increments the current block number.

Command	Description
[start block] [end block] QR	Searches for a string and either replaces it or does not replace it, based on your answer to a query. QR prompts you for the search and replace strings. The system searches for the first occurrence of the search string. When the system finds the string, QR prompts you to confirm the replacement by positioning the cursor at the end of the string. You enter Y, y, or space to replace the search string with the replace string. You can enter ? to display a help menu.
R-WITH [string^]	Loads the insert buffer with the specified string.

6.8 EXAMPLE RCS DIALOGUE

This section gives a possible dialogue for a user of the NBS RSL application. The dialogue contains examples of most of the procedures described in sections 6.2 through 6.7. The dialogue locates the word Display in the vocabulary of the PATH level on the TASK&PATH board. The word Display displays the contents of certain variables.

After the dialogue locates Display, it executes Display, edits Display to label the variables displayed, and saves the updated routine on the disk. The dialogue indicates each of these steps with a separator shown in all uppercase characters. Information that you type appears underlined in the dialogue. Information that the system displays appears in boldface in the dialogue. Brief explanations are included in parentheses.

Remember that this example is specific to the NBS implementation of RCS, and the RSL application of RCS. If you are working with the NBS implementation, you may want to use this dialogue as a tutorial to get acquainted with basic RCS procedures. Centered headings appear in the tutorial to indicate the function of each group of commands.

To start the tutorial, boot the system as described in Steps 1 through 3 of Section 6.1, "Starting the System". After you boot the system, it displays the prompt for the master RSL board.

STEP 1: LOCATE Display FOR THE PATH LEVEL

```
  :R rsl> Task/path REMOTE          (Moves to the task/path board.)

  :R task&path> PDEF                (Makes the path vocabulary current.)

  :R task&path> LOC Display         (Lists the source block for Display.)

  1909      10909  ABS
                                             PATH DEFINITIONS
  routine Display
    if print-f (EQ) true
    then
      ~F CR
      ~PRINT" path: "
      ~PRINT inc-command-#-in
      ~PRINT input-command
      ~PRINT status-report
      ~PRINT status-arg-out
      ~PRINT ppt-command
      ~PRINT ppt-done
    endif
  end-routine
```

STEP 2: EXECUTE Display

```
  :R task&path> Display             (Executes the word Display.)

  path:  0 0 0 0 0 0
```

STEP 3: EDIT Display TO ADD A LABEL

```
  :R task&path> ^X                  (Returns to the master board containing the
                                    editor.)

  :R rsl> red                       (Edits the block most recently accessed on
                                    the most recent slave board.)
```

STEP 3: EDIT Display TO ADD A LABEL (cont.)

```
     CHARACTER OPS      LINE OPS        BLOCK OPS           SPECIAL OPS
     ^A Insert On/Off   ^C Copy Line    ^F Forward Block    PF1 Home   PF2 Clear
     ^E Erase Char      ^D Delete Line  ^P Previous Block   ^U Undo
     ^K Mark to Keep    ^O Open Line    ^G Goto Block       ^W Want string
     ^Z Input Keep                      ^T Transfer         ^N Next Want
                                        ^R Retire(quit)     ^X Cancel

     29Ø9         1Ø9Ø9
                                            PATH DEFINITIONS
   Ø
   1  routine Display
   2    if print-f (EQ) true
   3    then
   4       ~F CR
   5       ~PRINT" path: "
   6       ~PRINT inc-command-#-in
   7       ~PRINT input-command
   8       ~PRINT status-report
   9       ~PRINT status-arg-out
  1Ø       ~PRINT ppt-command
  11       ~PRINT ppt-done
  12    endif
  13  end-routine
  14
  15
```

Use the arrow keys to position the cursor at the first character in line 1Ø.

^A — (Enables you to insert characters. Notice the insert indicator appears after the absolute block number.)

~PRINT" ppt: " — (Labels the path-point output. Do not put a space before the first quotation mark. Do put a space before and after the second quotation mark.)

^A — (Turns insert off.)

^R — (Exits the editor.)

```
Bye

:R rsl> L 29Ø9
```

(Lists the current block with your editing changes.)

STEP 3: EDIT Display TO ADD A LABEL (cont.)

```
29Ø9
                                               PATH DEFINITIONS
   routine Display
     if print-f (EQ) true
     then
        ~F CR
        ~PRINT" path: "
        ~PRINT inc-command-#-in
        ~PRINT input-command
        ~PRINT status-report
        ~PRINT status-arg-out
        ~PRINT" ppt: " ~PRINT ppt-command
        ~PRINT ppt-done
     endif
   end-routine


```

STEP 4: LOAD THE BLOCK AND EXECUTE THE ROUTINE

```
:R rsl> ^X                      (Returns to the most recent slave board.)

:R task&path>
:R task&path> 19Ø9 ELOAD      Display REDEFINED

                                (Empties the local block buffers and then
                                loads the specified block.)

:R task&path> Display           (Executes the word.)

path: Ø Ø Ø Ø ppt: Ø Ø
:R task&path>
```

STEP 5: SAVE THE UPDATED ROUTINE

```
:R task&path> 6 PRESERVE 6 PRESERVE-FILE

                                (Writes the system dictionary and SMACRO
                                files to the 6 PRESERVE area on the disk.)
```

```
:R task&path> 3 MEM>DISK " Patched Display 5/23 "
```

(Writes the current D>M, D>M number 3, from the local RAM on the TASK&PATH board, to the disk. The text within quotation marks is a comment briefly describing the update you made, followed by the date. The comment appears as a message the next time you execute a 3 D>M. You may want to save comments in a logbook to track the changes you make to the system.)

STEP 5: SAVE THE UPDATED ROUTINE (cont.)

```
make image? Y
```

(Confirms the MEM>DISK command and writes the image to disk.)

```
Task/path levels
NO RESTORE
Task init
Path init
sonar not inited

Task/path inited
Patched Display 5/23
:R task&path>
```

6.9 USING PRINTING UTILITIES

To use the RCS printing utilities, enter 5 D>M on the master board for the printing application. After you enter 5 D>M, you can use the printing words described in the following sections and in Appendix B, "User Word Summary". These commands enable you to print a range of blocks and to print block directories and programs. For information on configuring your printer, see Section 5.4, "Hardware Installation Procedures".

Printing a Range of Blocks

You can print a range of blocks with 1Ø blocks on each page by using the CPRINT command. Enter the start block and the end block, separated by a space. Then enter the word CPRINT. For example, to print blocks 1Ø through 19 on one page, enter 1Ø 19 CPRINT.

To print a range of blocks with three blocks on each page, use the same procedure, but enter the word Print instead of CPRINT. For example, to print blocks 1 through 9 with three blocks on a page, enter 1 9 Print.

Printing Block Directories and Programs

The words List-directory and List-programs enable you to print the block directories and the programs, respectively. For more detailed information on these commands, see Appendix B, "User Word Summary".

Note: When using a TeleVideo 95Ø, prefix the printer commands with the word Thru-tv. Using these words puts the terminal in transparent mode so that printer control codes are passed to the printer; for example for a Televideo 95Ø, Thru-tv 1Ø 19 CPRINT.

6.1Ø USING TAPE UTILITIES

To use the RCS tape utilities, enter 5 D>M on the master board for the tape application. After you enter 5 D>M, you can use the tape words described in the following sections and in Appendix B, "User Word Summary". These commands enable you to back up the system on tape, solve a bad-tape problem, and read a tape.

Backing Up the System on Tape

The word BACKUP lists the backup block that contains the commands to back up the system on tape. BACKUP uses the ADD-TO-TAPE command to write a file to the tape. To back up the system on tape, load the backup block. You must edit the backup block to conform to your applications.

Solving Bad-Tape Problems

If you receive a bad-tape message for an ADD-TO-TAPE command, use the command BAD-TAPE preceded by the start and end blocks of the file to extend the file to cover the bad section and preserve the file numbering sequence. You cannot use this command if the bad section of tape is in the first file. In that case, start the tape operation using a new tape.

Reading a Tape

To read a tape, use the command FROM-TAPE preceded by the start and end blocks of the file you want to read. FROM-TAPE reads the range of blocks from the tape, beginning from the current tape position. To use this command, you need a listing of the BACKUP block that contains the current block numbers for the files.

6.11 SHUTTING DOWN THE SYSTEM

To shut down the system, follow this procedure:

Note: If you are using the system without running the robot, start with Step 4.

1. Move the robot to a safe place.

2. Enter HALT. The HALT command aborts communications through the robot interface, turns off the background task on each board, and turns off the communications process.

3. Turn off the robot.

4. If you want, save any changes you made to the code, using the MEM>DISK, PRESERVE, and PRESERVE-FILE commands. Making and saving changes are described under Section 6.6, "Saving and Rebooting the System", earlier in this chapter.

5. Enter WUNLOAD. The WUNLOAD command shuts down the Winchester disk. Always enter this command before you turn off rack power to the RCS hardware.

6. Turn off rack power to the RCS hardware.

The shut-down procedure is now complete.

Chapter 7
SMACRO

RCS contains code written in SMACRO, FORTH, and 8Ø86 assembly language. This chapter describes when to write code in the different languages and gives a comprehensive description of SMACRO, the primary language used in RCS. The chapter assumes that you know the FORTH programming language.

7.1 OVERVIEW

The base operating system of RCS is polyFORTH 1. SMACRO is an extension of polyFORTH, containing a set of macros that enable you to write structured programming code similar to high-level languages like Pascal and C. SMACRO is a high-level assembler and, like most assemblers, it generates code that usually executes faster and is more compact than similar code written in a high-level, structured language.

In addition to the speed of SMACRO, SMACRO is easy to debug because of the interactive nature of FORTH and SMACRO. You can execute a single SMACRO routine, or you can edit and recompile one routine of a larger application and then execute the application without recompiling the rest of the code.

NBS developed SMACRO to make writing structured programming code easy. However, SMACRO may not be suited for all tasks. If SMACRO is not suited for a particular task, you can include FORTH or 8Ø86 assembly language within a block of SMACRO code, or you can write blocks of code entirely in FORTH or assembly.

For example, if timing is important for a routine, you may need to use assembly language, to make the routine operate fast enough to execute in one control cycle. Also, because SMACRO does not support block and disk access, you may have to write some routines in FORTH. The read routines in the TASK level that access the system dictionary are written in FORTH.

Your NBS-supplied system tape also contains Robot Sensor Language (RSL) code. RSL is a language for controlling robot motions; it is one specific application that runs in RCS. Chapter 9, "Robot Sensor Language (RSL)", describes RSL in more detail.

7.2 SMACRO SYNTAX

This section describes the syntax of the SMACRO language, including programming conventions, variable declarations, members and owners, file declarations, operators, and statements. See Appendix B, "User Word Summary", for a complete list of SMACRO words.

In general, the SMACRO compiler does not detect syntax errors. Using DBG-ON catches some errors, but not all. You should test routines to ensure that the algorithm, as well as the syntax, is correct. To verify that each routine runs correctly, test each routine individually.

Programming Conventions

The SMACRO code that is on the system tape follows several programming conventions. These conventions include:

- The vocabulary name is in the top right corner of the first line of the block.
- A comment describing the purpose of the routine and the input and output variables appears at the beginning of the routine.
- Routine names appear in uppercase characters.
- Variable names appear in lowercase characters.
- Code is indented two spaces.

To keep your system organized and easy to maintain, continue to follow these conventions.

Figure 7-1 shows an example block of SMACRO code to illustrate SMACRO programming conventions:

```
                                                                     SDEF
routine SMACRO-EXAMPLE
  % Example routine to illustrate SMACRO programming conventions
  % Input: variable1, variable2, ...
  % Output: variableA, variableB, ...

  SMACRO routine statements

end-routine
```

Figure 7-1. Example SMACRO routine.

The vocabulary identifier on the first line of a SMACRO block automatically makes the vocabulary you specify the current vocabulary when you reload the block. The line also displays the current vocabulary when you use the LOC command.

Percent symbols in SMACRO code indicate that the rest of the line is a comment. List the input and output variables at the beginning of each routine to make code easier to read and debug.

The convention of using all uppercase characters for routine names and all lowercase characters for variable names is used throughout the system. RCS will be easier to maintain if you follow this convention when you add code to the system.

Variable Declarations

You must declare a variable before you use it, and the variable you declare must be in a vocabulary. RCS already contains a system vocabulary for each board. RSL contains the vocabularies TASK, PATH, PRIM, and JOINT. Make sure you are in the correct vocabulary before you declare a variable. For information on changing vocabularies, see Section 6.2, "Moving From One Board to Another".

Table 7-1 shows how to declare a variable for the SMACRO variable types.

Table 7-1. SMACRO Variables.

Variable Type	SMACRO Declaration
16-bit integer	iv [variable name]
8-bit integer byte	bv [variable name]
32-bit floating point	fv [variable name]
1-dimensional integer array	[# elements] 1:a [variable name]
2-dimensional integer array	[# rows] [#columns] 2:a [variable name]
1-dimensional floating point array	[# elements] 1:fa [variable name]
2-dimensional floating point array	[# rows] [#columns] 2:fa [variable name]
string	[# bytes] strv [variable name]
segment	[segment] [address] [# bytes] segv [variable name]
sequential	seqv [variable name]

On your system tape, variables tend to be declared at the beginning of each control level, as they are in high-level languages like Pascal. However, in SMACRO and FORTH, this method is only a convention; you can declare variables at any place before you use them.

Remember that all variables in RCS are global within a vocabulary. A SMACRO variable name can be up to 31 characters long, and can contain any ASCII character, except a space. Variable names must be unique within a vocabulary.

To refer to an array element in SMACRO code, enclose the indices in braces. For example, A1 { x } is the xth element in array A1. A2 { x y } is the element in the xth row and yth column of array A2. The indices must be integer or sequential variables, and the first element in the array is Ø.

In addition to the SMACRO variable types listed in Table 7-1, RCS uses matrix, pose, quaternion, and vector variables to represent robot positions and movements. SMACRO represents these variables in floating point arrays.

A vector consists of three consecutive floating point values. A pose consists of seven consecutive floating point numbers. The first four numbers represent the quaternion (or orientation) part of the pose, and the last three numbers represent the vector (or position) part of the pose.

A rotation matrix consists of nine consecutive floating point numbers, stored in the order: m11, m21, m31, m12, m22, m32, m13, m23, m33. Using this order, the matrix represents a coordinate frame with the first column representing the x unit vector, the second column representing the y unit vector, and the third column representing the z unit vector.

A quaternion consists of four consecutive floating point numbers. The first number represents the scalar part of the quaternion, and the last three numbers represent the vector part of the quaternion.

Table 7-2 gives an example declaration for representing a vector, pose, matrix, and quaternion.

Table 7-2. Matrix, Pose, Quaternion, and Vector Declarations.

Variable Type	SMACRO Declaration
Vector named V1	3 1:fa V1
Pose named P1	7 1:fa P1
Pose named PQ	4 1:fa PQ 3 1:fa PV
Matrix named M1	9 1:fa M1
Matrix named M2	3 3 2:fa M2
Quaternion named Q1	4 1:fa Q1

Members and Owners

Variables are always grouped as members under an owner. Members include all SMACRO variable types. The three types of owners include: variable owners (VAR-O), sequential variable owners (S-VAR-O), and list owners (LIST-O).

Variable owners

The following list gives the syntax, function, and members allowed for a variable owner.

Syntax: [# bytes] bytes VAR-O [name]

Function: Enables you to group variables by allocating space in the SMACRO segment.

Members: integer, byte, floating point, string, segment, one-dimensional integer array, two-dimensional integer array, one-dimensional floating point array, two-dimensional floating point array.

Figure 7-2 presents an example variable owner that contains each type of member a variable owner can have.

```
 Ø                                                      SDEF
 1       125  bytes      VAR-O        VO-EXAMPLE
 2                 iv       I1
 3                 bv       B1
 4                 fv       F1
 5            1Ø   1:a      IARRAY1
 6          3  3   2:a      IARRAY2
 7             4   1:fa     FARRAY1
 8          2  6   2:fa     FARRAY2
 9            1Ø   strv     STR1
1Ø      9ØØØ  8Ø 4 segv     SEG1
11
12
13
14
15
```

Figure 7-2. Example variable owner.

Table 7-3 gives the action of each line in Figure 7-2. In Table 7-3, the value in parentheses is the number of bytes allocated.

Table 7-3. The Variable Owner VO-Example.

Line Number	Action
Ø	Puts the contents of this block in the SDEF vocabulary.
1	Allocates a 125-byte buffer, named VO-EXAMPLE, in the SMACRO segment. Defines the variable owner, VO-EXAMPLE. The number of bytes, 125, is the sum of the bytes required for all the members. The word "bytes" is a null word to make the source code more readable. If the members allocate more than the specified number of bytes, RCS issues a buffer overflow error message.
2	Assigns a 16-bit integer variable, I1, to the VAR-O. (2 bytes)
3	Assigns an 8-bit integer variable, B1, to the VAR-O. (1 byte)
4	Assigns a 32-bit floating point variable, F1, to the VAR-O. (4 bytes)
5	Assigns a 1Ø-element one-dimensional 16-bit integer array, IARRAY1, to the VAR-O. (2Ø bytes)

Line Number	Action
6	Assigns a 3 x 3 two-dimensional 16-bit integer array, IARRAY2, to the VAR-O. (18 bytes)
7	Assigns a 4-element one-dimensional 32-bit floating point array, FARRAY1, to the VAR-O. (16 bytes)
8	Assigns a 2 x 6 two-dimensional 32-bit floating point array, FARRAY2, to the VAR-O. (48 bytes)
9	Assigns a string with a maximum of 1Ø characters, STR1, to the VAR-O. (12 bytes: 1Ø for the string, 1 for the maximum character count, and 1 for the actual character count)
1Ø	Assigns a 4-byte section of common memory, SEG1, whose absolute address is at offset 8Ø, segment 9ØØØ. (6 bytes: 2 bytes for length and 4 bytes for the 32-bit address)

Sequential variable owners

The following list gives the syntax, function, and allowable members for a sequential variable owner.

Syntax: [# members] mem S-VAR-O [name]

Function: Enables you to group sequential variables by allocating space in the SMACRO segment

Members: sequential variables

Figure 7-3 presents an example sequential variable owner that contains three members:

```
        Ø                                                      SDEF
        1    3   mem        S-VAR-O    SVO-EXAMPLE
        2          seqv               SA
        3          seqv               SB
        4          seqv               SC
        5
        6
        .
        .
        .
        14
        15
```

Figure 7-3. Example sequential variable owner.

Table 7-4 lists the action of each line in the preceding figure:

Table 7-4. The Sequential Variable Owner SVO-Example.

Line Number	Action
Ø	Puts the contents of this block in the SDEF vocabulary.
1	Creates a 6-byte buffer, named SVO-EXAMPLE, in the SMACRO segment. The buffer contains consecutive integers starting at Ø. The word "mem" is a null word to make the source code more readable. The maximum number of members is 3.
2	Assigns the variable SA with a value of Ø to the S-VAR-O.
3	Assigns the variable SB with a value of 1 to the S-VAR-O.
4	Assigns the variable SC with a value of 2 to the S-VAR-O.

List owners

The following list gives the syntax, function, and allowable members for a list owner.

Syntax: LIST-O [name]

Function: Enables you to group any type of member together for display

Members: Any type of variable

Figure 7-4 presents an example list owner that contains an integer and two owners.

```
Ø                                           SDEF
1        LIST-O     LO-EXAMPLE
2     m   A
3     m   IARRAY1
4     m   SA
5
6
.
.
.
14
15
```

Figure 7-4. Example list owner.

Table 7-5 lists the action of each line in the preceding block.

Table 7-5. The List Owner LO-EXAMPLE.

Line Number	Action
Ø	Puts the contents of this block in the SDEF vocabulary.
1	Groups the members under the owner LO-EXAMPLE.
2	Assigns an integer variable A to the LIST-O.
3	Assigns an integer array, IARRAY1, to the LIST-O.
4	Assigns a sequential variable, SA, to the LIST-O.

Note: You must define all list owner members before you add them to the list owner. The numbers must be in the same vocabulary as the owner.

For a complete list of the RCS words for members and owners, see Appendix B, "User Word Summary".

File Declaration

Use the SMACRO word FILE to declare SMACRO files. This declaration creates a SMACRO file, using the most recently declared variable owner as the template. (Include the file declaration in the source block that declares the variable owner.)

RCS stores SMACRO files in common memory, providing the primary method of communication between control levels. SMACRO files consist of a linked list of records. Record numbers identify individual records in a file. The record number is the address of the record in the file segment of common memory.

Header variables contain information about the file, such as the number of records available, the address of the first record, and the record size. The header variables are stored in common memory with the file and retrieved to local variables when you open the file.

SMACRO Operators for Standard Programming Operations

The SMACRO operators that enable you to perform most standard programming operations are:

- Arithmetic
- Assignment
- Relational
- File

In SMACRO, you perform operations using infix notation as in mathematics. Different operators are available for different types of variables. The operator types include: integer, byte, floating point, string, pose, matrix, quaternion, and vector.

In general, SMACRO encloses most integer operators in parentheses, precedes most byte operators with a capital B, and encloses most floating point operators in periods. Operands must be of the same type as the operator. Conversion operators convert integers to floating point numbers and floating point numbers to integers.

Arithmetic operators

SMACRO includes arithmetic operators for adding, subtracting, multiplying, and dividing integers, bytes, and floating point numbers. Table 7-6 lists these operators.

Table 7-6. SMACRO Arithmetic Operators.

Integer Operators	Byte Operators	Floating Point Operators
(+)	B+	.+.
(-)	B-	.-.
(*)	B*	.*.
(/)	B/	./.

Integer operations produce integer results. When you use integer division, remember that SMACRO truncates the fractional part of the result. For example, if you enter 5 (/) 3, the result is 1. SMACRO operations proceed from left to right; there is no precedence for integer and byte expressions.

You can use parentheses enclosed in periods to enforce precedence, but only for floating point numbers. You can nest scalar operations up to eight levels. You can nest vector operations only one level, and pose and quaternion operations to any level.

Here are some example SMACRO arithmetic operations and the results:

Statement	Result
I1 (*) I2	Multiplies the integer variables I1 and I2.
B1 B+ B2	Adds the byte variables B1 and B2.
F1 .-. F2	Subtracts the floating point value of F2 from the floating point value of F1.
I1 (+) I2 (*) I3	Adds integers I1 and I2 and multiplies the result by integer I3.
F1 .+. .(F2 .*. F3).	Multiplies the floating point numbers F2 and F3, and then adds the result to the floating point value in F1.

Be careful not to mix variable types in arithmetic operations. You may not get an error message, but you might get incorrect results. SMACRO does not allow literals in statements. To include constants in a statement, declare a variable with an appropriate name and initialize it.

Assignment operators

SMACRO includes assignment operators for integer, byte, floating point, string, pose, quaternion, matrix, and vector variables.

An assignment operator stores the variable or expression on the left side of the operator in the variable location specified on the right side. The arrow indicates the direction of the assignment. (Most high-level languages use the opposite direction.) Table 7-7 lists the assignment operators.

Table 7-7. SMACRO Assignment Operators.

Operator	Type
=>	Integer
B=>	Byte
.=>.	Floating Point
S=>	String
.P=>.	Pose
.Q=>.	Quaternion
.M=>.	Matrix
.V=>.	Vector

Here are some example assignment operators:

I1 (+) I2 => I1	Assigns the sum of integer variables I1 and I2 to the integer variable I1.
A2 { x y } => I1	Stores the element in row x, column y, of the integer array A2, to the integer variable I1.
B1 B* B2 B=> B3	Assigns the product of byte variables B1 and B2 to the byte variable B3.

Relational operators

Relational operators enable you to compare variables of the same type. You can use relational operators only in the conditional statements "if then else endif", "while do end-do", and "repeat until end-repeat". You cannot use relational operators in assignment statements. Table 7-8 lists the SMACRO relational operators.

Table 7-8. SMACRO Relational Operators.

Integer Operators	Byte Operators	Floating Point Operators	
(EQ)	EQ_B	.EQ.	.EQZ.
(NE)	NE_B	.NE.	.NEZ.
(LT)	LT_B	.LT.	.LTZ.
(LE)	LE_B	.LE.	.LEZ.
(GT)	GT_B	.GT.	.GTZ.
(GE)	GE_B	.GE.	.GEZ.

These operators perform the comparisons equal to, not equal to, less than, less than or equal to, greater than, and greater than or equal to, respectively. The second column of floating point operators compares floating point expressions to Ø. The floating point expression goes before the operator. Relational operators produce Boolean results of true or false. For example:

I1 (EQ) I2 — results in the Boolean value of true if integer I1 is equal to integer I2.

F1 .GEZ. — results in the Boolean value of true if the floating point variable F1 is greater than or equal to Ø.

You can also use arithmetic expressions on the left side of a relational test. For example:

I1 (+) I2 (/) I3 (EQ) I4 — results in a Boolean value of true if the result of the integer expression I1 (+) I2 (/) I3 equals integer I4.

SMACRO also includes a string comparison operator, S-EQ, which compares two strings of equal length using the ASCII character values.

The SMACRO logical (AND) enables you to compare two Boolean expressions. The result is true only if both expressions are true.

File operators

File operators enable you to perform operations such as storing data from the local template to a record, retrieving the current record from the current file, and returning the current list to the free list. Appendix B, "User Word Summary", contains a list of the operators for manipulating files.

SMACRO also includes four Boolean file operators: matches, matches-r, matches-fields, and matches-fields-r. Use these Boolean file operators to search for a variable in the file and, if desired, to retrieve the local template of the first record that matches the variable.

In other systems, files reside on the disk, and you must open and close them. You can open only one file at a time, and you can only read or write to that file. Closing the file enables other processes to access it.

In SMACRO, files reside in common memory, and you must open but you do not need to close them. To open a file, you execute its name. You can open a file from any number of boards; no inherent protection mechanism exists. You must ensure that two boards do not try to access the same file simultaneously. This structure increases the access speed of the files. Potential read/write conflicts between files arise rarely, so RSL handles these conflicts as a special case using round-robins.

The control levels of RCS communicate through SMACRO files in common memory. The file structure consists of linked lists of records. For each file declaration, SMACRO sets up a file using the most recently declared variable owner as the template. The members of the variable owner define the fields for the records in the linked list. The file declaration also adds two fields to each record: a link field that points to the next record in the linked list, and a source block field that points to the source block that contains the data in the record.

The following SMACRO code declares a file named DATA-FILE:

```
16 bytes VAR-O file-rec
15 strv "field1"
   iv   field2

 5 rec FILE DATA-FILE
```

The first three lines of the example define a variable owner named file-rec that serves as the local template for DATA-FILE. The last line of the example allocates space in common memory for a file named DATA-FILE with five records. The file declaration fills the file with a list of empty records called the free list. The linked list for the file is terminated by a record whose link field contains Ø.

Figure 7-5 shows a representation of the file as initialized by re-init-file.

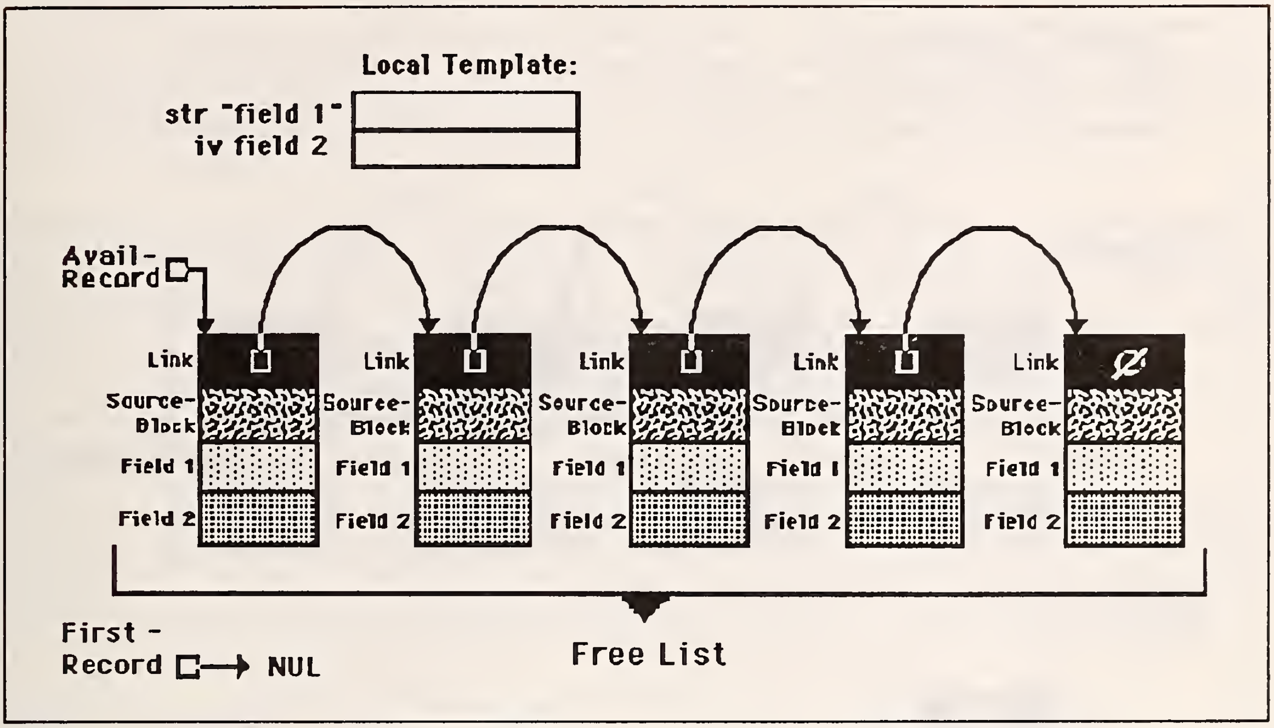


Figure 7-5. An example SMACRO file in common memory.

You create the file represented in Figure 7-5 by loading the block that contains the file declaration and initializing the file with the work re-init-file. RCS maintains the free list. When you add a record to a list, the record comes from the free list. When you remove a record from a list, the record returns to the free list.

You open a file by executing the file name. When you execute the file name, RCS retrieves header variables that contain information about the file from common memory to local variables.

Figure 7-6 shows the file after a record is added. Assume that the local template variables contain the data shown in the figure. To use DATA-FILE, you execute DATA-FILE to open the file, and then execute add-record to store the data in the local template to a record, remove the record from the free list, and add the record to the end of the current list. After add-record executes, DATA-FILE looks like Figure 7-6.

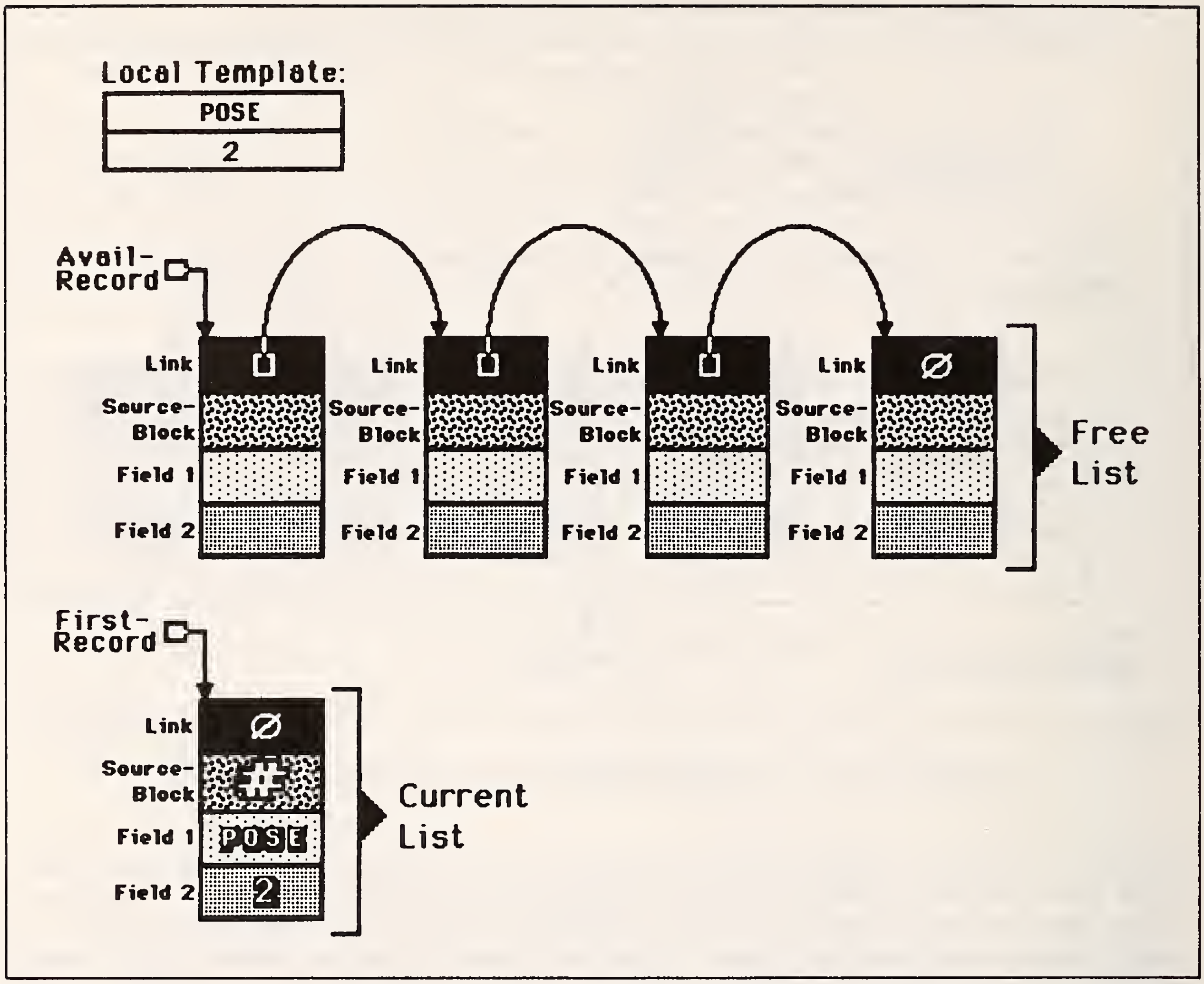


Figure 7-6. SMACRO file after the execution of add-record.

Additional SMACRO Operators

SMACRO includes additional operators that enable you to perform the following types of operations:

- Bit
- I/O
- Stack
- Matrix
- Pose
- Quaternion
- Vector
- Others

The bit, I/O, and stack operators enable you to perform operations similar to those in assembly language. The matrix, pose, quaternion, and vector operators make representing robot positions and movements easier. For the exact syntax of these operators, see Appendix B, "User Word Summary".

Bit operators

Several operators perform bit-wise operations. The operator ^EQ^ performs a bit-wise logical AND of two integer variables, compares the result with the second variable, and returns true if the variables are equal. The operators set-bit-in and zero-bit-in enable you to define a bit mask to set specific bits in a word.

Use the bit operators 1-? and Ø-? in conditionals. The operator 1-? performs a bit-wise logical AND of two integer variables and returns a true value if the result is not Ø. The operator Ø-? takes the complement on the first integer variable and then performs a bit-wise logical AND with the second integer variable. The operator returns a true value if the result is not Ø.

I/O operators

SMACRO includes the in-port and out-port I/O operators. The in-port operator enables you to read a value from the port you specify. The out-port operator enables you to send an integer expression to the port you specify. Use these operators when you want to communicate with sensors in the system.

Stack operators

Stack operators enable you to transfer values to and from the 8Ø86 processor stack. Table 7-9 lists the integer and byte stack operators. (You also can access an internal 8Ø87 stack through assembler code.)

Table 7-9. SMACRO Stack Operators.

Integer Operators	Byte Operators
to-stack	B-to-stack
from-stack	

The to-stack operator stores a variable or the result of a calculation onto the stack. For example:

I1 (+) I2 to-stack — Moves the sum of integers I1 and I2 to the stack.

A2 { x y } to-stack — Moves the element in row x, column y, of integer array A2, to the stack.

B1 B+ B2 B-to-stack — Moves the sum of byte variables B1 and B2 to the stack.

The from-stack operator takes values from the processor stack. You can assign the values to variables or use the values in calculations. The from-stack operator must appear on the left side of an assignment. For example:

from-stack => I1	moves the integer value from the processor stack and stores it in the variable I1.

Matrix, pose, quaternion, and vector operators

RCS uses matrices, poses, quaternions, and vectors, stored as floating point arrays, to represent robot positions and movements. Most other robot applications use 4 x 4 homogeneous matrices to represent robot movements. RCS uses quaternions, which are more efficient. SMACRO includes operators to make vectors, matrices, and quaternions easier to use.

For more information on how to use quaternions in robotics, refer to Robot Motion: Planning and Control, part of the MIT Press series in artificial intelligence. You will find specific information for quaternions on pages 24Ø to 265.

Note: All quaternions in RCS are rotation quaternions with a magnitude of 1 and a non-negative scalar element. When performing quaternion operations, RCS uses the 8Ø86 stack for temporary storage because these operations require the use of all 8Ø87 registers.

Therefore, you can nest quaternion operations to a depth limited only by the size of the 8Ø86 stack. However, because RCS stores all vectors in the 8Ø87 registers, a sequence such as vexp .V+. .(qexp .Q*V. vexp). overflows. On the other hand, a sequence such as qexp .Q*V. vexp .V+. vexp does not overflow.

The SMACRO matrix operator words operate on 3 x 3 rotation matrices only.

A pose consists of a quaternion part and a vector part. The quaternion part is a rotation quaternion. The 8Ø86 stack is used for temporary storage of poses, so you can nest pose expressions to any depth for which there is enough space on the stack.

Other SMACRO operators

SMACRO includes many additional operators that perform operations such as calculating trigonometric functions, performing indirect addressing, converting an integer to a floating point number or a floating point number to an integer, reading and writing strings from the current input stream, and indexing arrays. You can also use FORTH operators. The SMACRO word ~F calls a FORTH word. For a complete list of SMACRO operators, see Appendix B, "User Word Summary".

SMACRO Statements

SMACRO includes the structured programming statements if, while, repeat, and case. SMACRO also includes an enhanced version of the case statement called a state-table. Using the state-table statement, you can test the state of up to seven variables. If all the variables are equal to the corresponding test variables, then the statement executes the code following that state line.

Table 7-10 presents the syntax for each of the SMACRO statements. Additional statements are in Appendix B.

Table 7-10. Syntax for SMACRO Statements.

Statement Name	Syntax
if	if [Boolean exp] then [code] else [code] endif
while	while [Boolean exp] do [code] end-do
repeat	repeat [code] until [Boolean exp] end-repeat
case	case [ivar1] case: [ivar2] [code] . . . . default: [code] end-case
state-table	state-table [state-var1] [state-var2] ... [state-var7] state: [test-var1] [test-var2] ... [test-var7] [routineA] . [test-var8] [test-var9] ... [test-var14] [routineB] . . . . . . default-state: [default-routine] end-state-table

7.3 SMACRO ROUTINES

All SMACRO routines start with the word routine and end with the word end-routine. Figure 7-7 presents an example SMACRO routine:

```
BLOCK 8106

$DEF
routine POSE-^ ( - pose name - )
  % find record# of named pose
  % output:  record# (0 if not found)
  POSE-FILE WREAD S=> "pose-name"
  first-record => record#
  if "pose-name" matches "pose-name"
  then endif
end-routine
```

Figure 7-7. Example SMACRO routine.

In SMACRO, routines can be longer than one block. To make a routine longer than one block, put the additional code that does not fit in the first block into the next block. Use the screen editor described in Section 6.7, "Editing a Block of Code", to move existing code to another block. Do not put a vocabulary name in the continuation block.

7.4 MODES

SMACRO words function differently depending on the current mode of RCS. The three RCS modes are Run, Show, and Locate. To set or change the mode, enter :R, :S, or :L (or, :r, :s, or :l), respectively. To change modes and re-execute the most recent word in the new mode, enter \R, \S, or \L (or \r, \s, or \l). To display the most recent word, enter \N (or \n).

A fourth mode called compile mode, indicated by :C, is used exclusively by the SMACRO compiler. You see the :C prompt only when an error occurs while a routine is compiling. Change to one of the other modes to continue.

In Locate mode, entering a SMACRO word lists the source block for that word. In Run and Show modes, the results of a SMACRO word depend on the type of the SMACRO word. Table 7-11 lists the results of using SMACRO words in Run and Show modes.

Table 7-11. SMACRO Words in Run and Show Modes.

SMACRO Word Type	In Run mode	In Show mode
VAR-O, S-VAR-O	Returns the address of the first member on the stack.	Displays all members.
LIST-O	Displays all members.	Displays all members.
iv, bv, fv, 1:a, 1:fa, strv, seqv	Returns the address of the variable to the stack.	Displays the variable name and value.
2:a, 2:fa	Returns the address of the variable to the stack.	Displays the variable name and values, one row per line.
segv	Returns the segment and address of the variable to the stack, leaving the address on the top of the stack.	Displays the name and contents of the data area of the variable, as unsigned 8-bit integers, in hex.
routine	Executes the routine.	Executes the routine.
FILE	Makes the file current.	Makes the file current.

7.5 BOARD LEVEL PROCESSES

RCS executes tasks in the foreground and the background. You run a task in the foreground when you execute a task at the terminal, and when you single-step each control level. You cannot access the terminal or perform any terminal operations from a routine running in the background task.

RSL has one background task per board that runs the control task, named BACK-TASK, for that board. You execute the background task by entering the word BGO. For examples of background tasks and broad level processes, see the RSL application (blocks 8020 through 8049).

7.6 INTERRUPTS AND ASSEMBLY LANGUAGE

You can write interrupt routines by beginning the routine with the word enter-interrupt and ending the routine with the word exit-interrupt. The word ~INTERRUPT sets the specified interrupt vector to point to your interrupt routine.

Figure 7-8 shows how to set up an interrupt routine.

```
                                                            SDEF
        routine  INTERRUPT-SERVICE
            %
            %
            enter-interrupt

               (interrupt routine code)

            exit-interrupt

        end-routine




        8 ~INTERRUPT INTERRUPT-SERVICE
```

Figure 7-8. Setting up an interrupt routine.

To enable the interrupt, you have to install the appropriate jumpers and clear the interrupt mask in the Programmable Interrupt Controller. Refer to the manual and schematics that come with the board for details. Also be careful not to disturb any of the interrupts currently implemented in the system. See Appendix C, "System Maps", for a list of the interrupts.

If you need to use assembly language code to write an interrupt routine or another routine, refer to the FORTH documentation for 8086 assembler, and Appendix E, "8087 Operation Codes For RCS", for 8087 assembler.

In addition to data, RCS can pass command and status information between control levels. Normal data consists of information such as robot poses, task descriptions, and relative moves. Command and status information consists of the control-level inputs and outputs that control task decomposition.

This chapter briefly explains that memory-resident files provide the medium for data transfer and then describes the communications process provided by NBS to transfer command and status information between control levels. Following a description of how the communications process works, this chapter explains how to program the communications process to transfer command and status information between control levels on different boards and between control levels on the same board.

8.1 USING MEMORY FILES TO PASS DATA

The different control levels within the system use common memory to pass data in the form of files. The control levels must implement the protocol to resolve file read/write conflicts. RSL uses round-robins to resolve these conflicts.

8.2 USING COMM TO PASS COMMAND AND STATUS INFORMATION

COMM is an RCS utility that provides a communications process for passing command and status information between control levels. Each control level can have an input status buffer, an input command buffer, an output status buffer, and an output command buffer. The output command buffer and the input status buffer of a control level communicate with the level below. The input command buffer and the output status buffer communicate with the level above. (See Figure 8-1.)

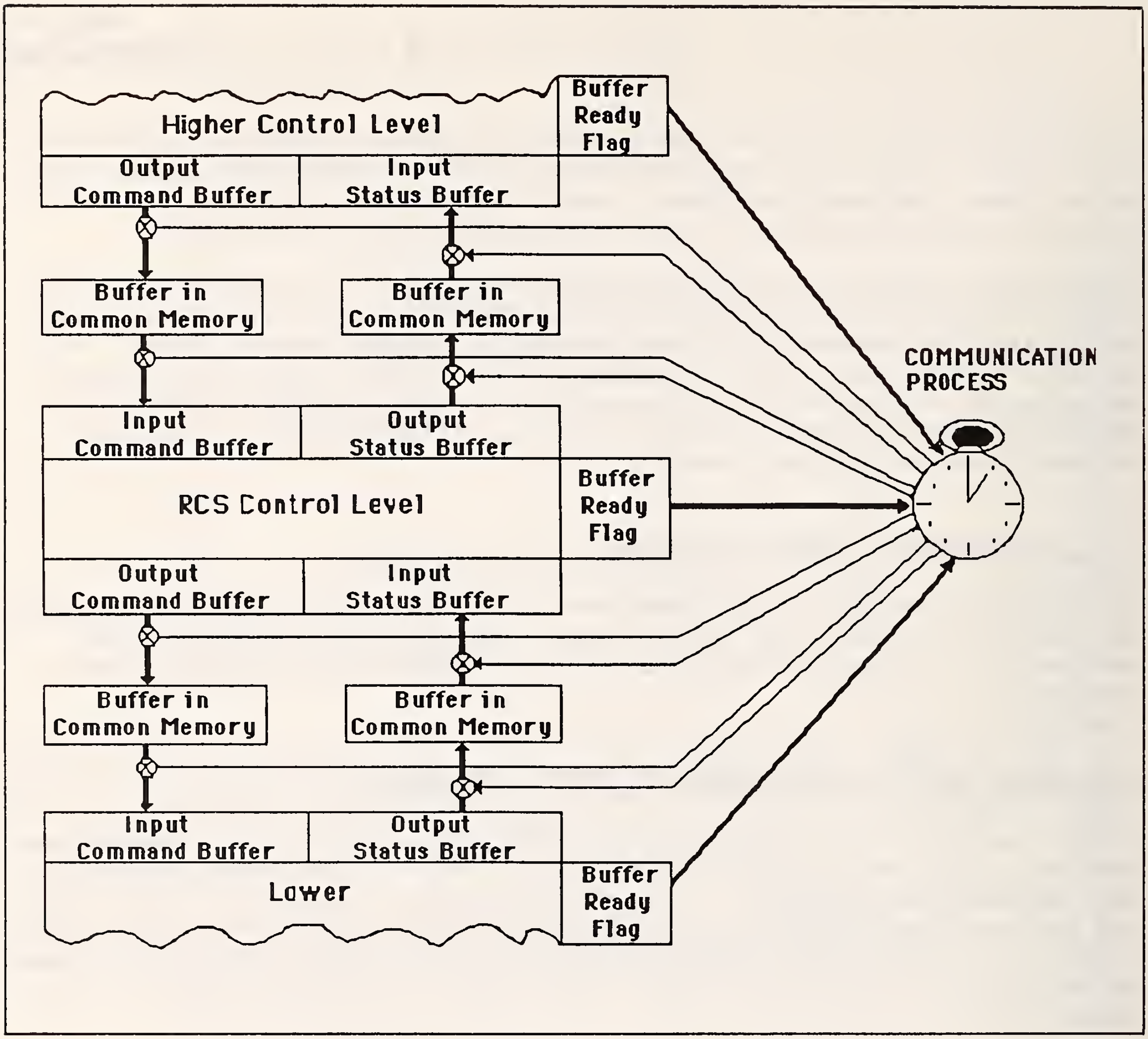


Figure 8-1. Command and status communication paths.

The communications process passes command and status information by moving the contents of the output buffer on one level to the corresponding input buffer on another level. This process operates in cycles to synchronize the flow of command and status information. Communication buffers in common memory enable the communications process to proceed in an asynchronous fashion. That is, these buffers can temporarily store command and status information until the control level is ready to accept it.

The period of the communications cycle is user-defined using a SMACRO variable called control-cycle-#-clks. COMM uses timer Ø to generate the communications cycle period. You can also synchronize the communications cycle to the operations of the robot.

A combination of ready flags on each level and a communications bit on the bus provide a means of resolving read/write conflicts. See Section 5.4, "Hardware Installation Procedures", for information on how to use one of the MULTIBUS interrupt lines as the communications bit.

No other process can execute while the communications bit is set and the communications process is moving the contents of the communication buffers between control levels. At the beginning of each cycle, the communications process resets the communications bit. Each level waits for this bit to be reset before starting execution.

While each level is executing, it sets the variable buffer-ready-f to not-ready to indicate that its buffers are not ready. When the level finishes executing, it waits for the system to reset the communications bit (which normally occurs immediately). Then the level sets buffer-ready-f to ready to indicate that the buffer is ready.

At the end of each cycle, the communications process sets the communications bit and moves all ready output buffers to an intermediate buffer in common memory. It then moves all data from the intermediate buffer to the corresponding input buffer, assuming the input buffer is ready. This protocol allows levels to overrun the cycle time without affecting communications.

The communications period during which no other process may execute is called communications dead time. To minimize this dead time, you must restrict buffers to the smallest possible size.

For example, to minimize the communications dead time, RSL communicates pose information between levels by passing pointers into a round-robin group of poses. This technique reduces the required buffer size (and speeds communication) because the levels communicate only the name of each pose rather than the information to describe a pose completely.

8.3 PROGRAMMING COMM

The program for the communications process consists of a table of buffer moves called a communications table. You must first enter the ERASE-COMM-TABLE statement and then repeat the following two statements for each internal buffer move desired:

```
TRANSFER-FROM [vocabulary] [output buffer]
TO-DESTINATION [vocabulary] [input buffer]
```

The ERASE-COMM-TABLE statement clears the contents of the communications tables on all the boards in the system. The TRANSFER-FROM statement selects the output buffer from which to send command and status information. The TO-DESTINATION statement selects the corresponding input buffer that is to receive the command and status information.

Each vocabulary specified with TRANSFER-FROM or TO-DESTINATION must include a semaphore variable named buffer-ready-f, which COMM uses to synchronize communications. Thus, each vocabulary actually has its own buffer-ready-f rather than a flag for each buffer. The communications process determines the address of this variable when it compiles the communications table. If you compile the communications table and this variable is missing, RCS returns the "no buffer-ready-f" message.

8.4 COMMUNICATING ON THE SAME BOARD

Two processes running on the same processor board do not need the communications process to resolve read/write conflicts. Processes running on the same processor board can use the S=> SMACRO operator.

For example, in RSL you can use the following two statements to move the contents of TASK's output command buffer to PATH's input-status-buffer:

```
TASK path-command-var PATH S=> input-command-var
PATH output-status-var TASK S=> path-status-var
```

This chapter describes the Robot Sensor Language (RSL), an example of a real-time application that NBS developed within the RCS environment. RSL is a high-level, task description language designed for programming robotic tasks, in which the control system, RCS, uses sensors to control the robot. This chapter defines RSL, describes RSL syntax, and gives procedures for entering, editing, and executing RSL code.

9.1 WHAT IS RSL?

RSL is a data-driven, semi-interpreted, user-extensible language that supports user-designed sensors, hierarchical task decomposition, and real-time execution. RSL is extensible; you can add a new sensor to a robot, and then extend RSL, enabling you to program robot tasks that use the new sensor.

RSL consists of two main programming structures: data and algorithms. Data is information about the robot environment such as the name, size, or location of an object. Robot sensors supply data to RSL, or you may enter the data directly from the keyboard.

An algorithm is the RSL code you use to accomplish a task. RSL compiles algorithms into a linked-list representation. Then the RSL control levels interpret the algorithm, using environmental data to command the robot to execute a task.

RSL decomposes tasks hierarchically, from the top down, using the RCS methods of cyclic execution, task decomposition, and control levels described in Chapter 2, "RCS Overview", and Chapter 3, "RCS Architecture". The four RSL-specific control levels are: TASK, PATH, PRIM, and JOINT, as shown in Figure 9-1.

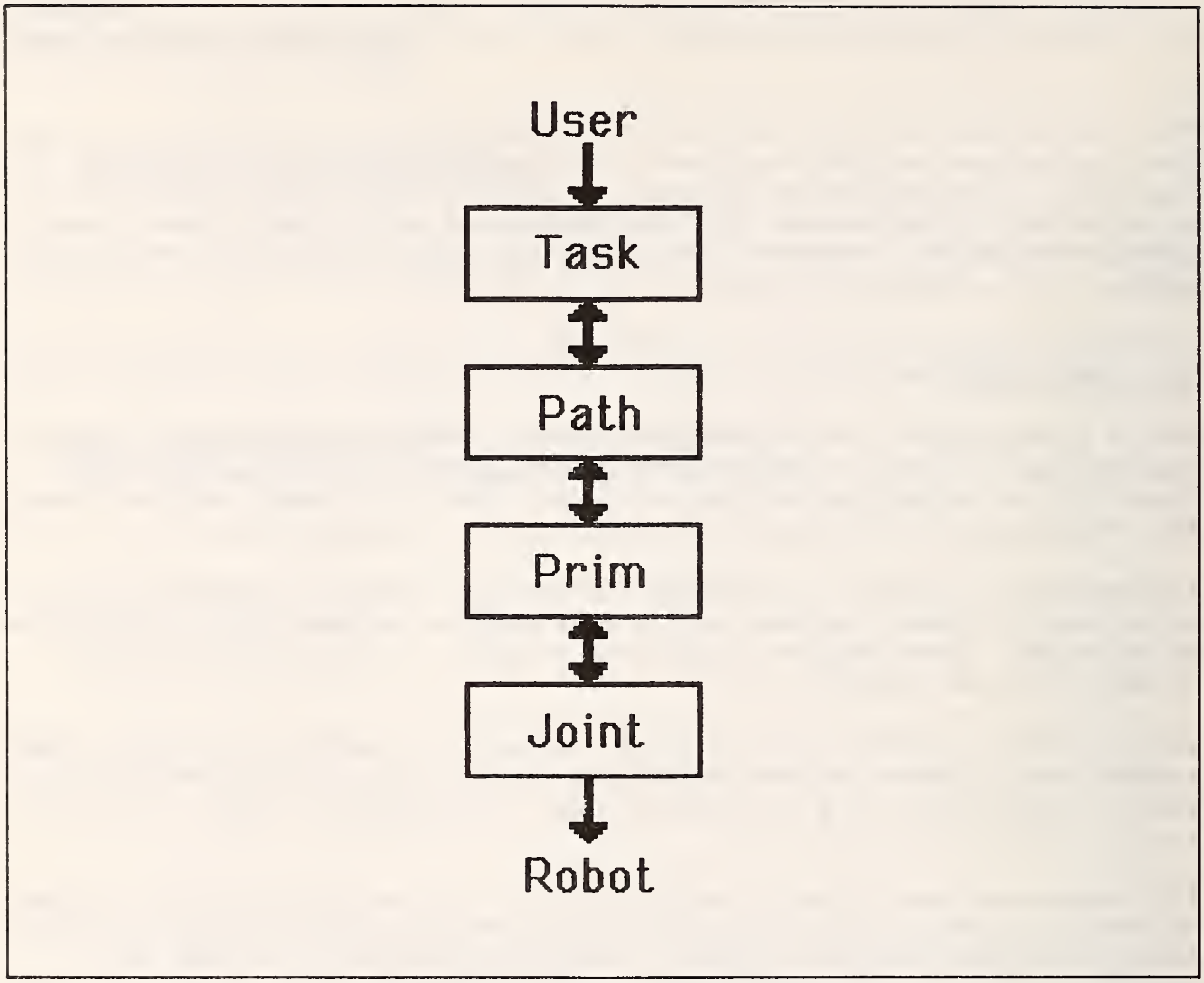


Figure 9-1. RSL control levels.

The code in these control levels defines the types of tasks that you can accomplish with RSL. You define tasks by writing algorithms, which define how to accomplish the task. The algorithms include operations such as transferring parts from a tray to a buffer or loading tools into a machining center. These algorithms become the input for the RSL control levels.

The TASK level decomposes all tasks into a sequence of path types. The path type and parameters of the current task identify the specific path to execute. RSL defines the path types: move-to, approach-pickup, depart-pickup, approach-release, and depart-release. You can define additional path types as you need them. A path algorithm specifies a simple path such as moving between locations or grasping objects. The specific path becomes the input for the PATH level.

The PATH level decomposes paths into a sequence of path-points. Each path-point consists of a single motion or sensor command. A path-point command performs operations, such as initiating a motion that a sensor condition terminates. You must define additional commands for each type of motion and sensor combination you need. The PATH level further decomposes the path-points into a sequence of trajectories, which become the input for the PRIM level.

The PRIM level decomposes trajectories into a sequence of intermediate points. A trajectory consists of a goal point and parameters describing the path to the goal point. RSL defines the straight-line Cartesian trajectory types and the joint-interpolated trajectory type. The intermediate points become the input for the JOINT level.

The JOINT level transforms intermediate points from Cartesian space to joint space. The joint values are sent to the servo controls in the manufacturer's robot controller.

Note: Keep in mind the different kinds of code in RSL. RSL contains data, such as the name of an object; algorithms, consisting of paths, path-points and trajectories, such as the RSL steps to accomplish a transfer task; and SMACRO code that defines the control levels.

9.2 RSL OVERVIEW

This section describes the syntax of RSL statements. See Appendix B, "User Word Summary", for the exact syntax of each RSL statement. The section also includes some example RSL code.

The Pose Statement

The pose statement, -pose-, defines a position, or the physical location, of the robot. A pose consists of a translation, represented by a vector, and a rotation, represented by a quaternion. The translation and rotation give the tool frame relative to the base frame of the robot.

With a six-axis robot, you may need to use up to three robot arm configuration flags to resolve the ambiguities of the inverse transform operation that transforms a pose into joint angles. These flags depend on the robot you use and the interface between RCS and the manufacturer's robot controller.

You may enter poses directly from the keyboard; or you may position the robot with a joystick or sensor-based algorithm and then record the pose. For information on how to enter a pose, see Section 9.3, "Entering and Editing RSL Source Code", later in this chapter.

The Movetable Statements

A movetable defines relative positions, using a combination of rotations and translations. That is, a movetable defines a coordinate transform. Adding a movetable to an existing pose produces a new location.

For example, if you define a pose at one corner of a table, you can define the other three corners of the table using movetables. If you want to reposition the table, you have to update only one pose--the movetables do the rest.

Defining a movetable requires several RSL statements such as -mtb-, ---, and -mtb-end-. Each transform line, ---, specifies either a vector translation or a rotation. You can also define a movetable that is the exact inverse of another movetable using the -imtb- statement.

The Location Statement

The location statement, -loc-, defines a location as the combination of a pose and a movetable. Locations may be goal points, or the origin of a coordinate frame in which to define motions. RSL adds the movetable and the pose just before the control system needs the result, enabling you to modify the data structures in real time. You can fine-tune a recorded pose by defining a movetable without using the robot. Also, you can define several related locations in terms of the same pose, so that RSL redefines the locations when it redefines the pose.

The Array Statement

Arrays may be one-, two-, or three-dimensional. The statement -arr- defines an array. The origin of an array is a location. A movetable specifies the displacement from one sector to the next sector along each dimension of the array. A sector is an element of an array. Use arrays to make palletizing operations easier.

The Object Statement

The object statement, -obj-, defines an object that consists of the object name and a list of grip numbers and movetables. The grip numbers define different orientations for gripping the object. RSL finds the grip orientation by adding the movetable to the object frame. Note that you may never refer to an object without a grip number.

The Path Statements

A path is an algorithm for performing simple tasks such as moving between locations or grasping an object. The path algorithm may be a simple path in space that guarantees no collisions, or the path may be more complex, requiring sensor input to help locate an object. Paths consist of a sequence of steps called path-points.

The path statement syntax has two parts: the path statement and a sequence of path-point lines. The statement -path- defines the type and goal of the path; the -ppt- lines give the sequence of steps in the algorithm.

The path type identifies the intended purpose of the path. RSL provides five path types: move-to, approach-pickup, depart-pickup, approach-release, and depart-release. You may define other path types, using these types as models. The rest of this section describes each of these five types of paths:

- The move-to path specifies how to move the robot quickly while it is carrying an object from near the starting location to near the destination location. The approach and depart paths handle slower movement closer to the starting and destination locations.

- The approach-pickup path specifies how to move the robot, with an empty end effector, toward an object at the goal location. The approach-pickup path starts from a safe location near the goal and stops at the goal location.

- The depart-pickup path specifies how to move the robot to pick up an object at a goal location. The depart-pickup path starts at the goal location and stops at a safe nearby location while the robot grasps the object.

- The approach-release path specifies how to move the robot while it is grasping an object. The approach-release path starts at a safe location near the goal and stops at the goal location.

- The depart-release path specifies how to move the robot at a goal location to release an object. The depart-release path starts at the goal location and stops at a safe nearby location, leaving the object at the goal location.

The Path-Point Statement

Each path-point consists of a single command. Typically, the commands initiate a motion, which a sensor condition terminates. The structure of the path-points depends heavily on the specific sensors and applications you are using. In general, path-points consist of a command phrase, a location phrase, and a trajectory phrase, although you may define other structures.

The command phrase gives the motion type, and specifies the sensor condition that terminates the motion. The location phrase specifies a location that RSL uses in one of two ways: as an intermediate goal point, or as a coordinate frame with translation along or rotation about an axis. The trajectory phrase specifies the trajectory type (Cartesian straight-line or joint interpolated) and the parameters (acceleration, maximum velocity, and neighborhood) RSL uses during execution of the path-point.

RSL provides only the goto path-point. The goto path-point specifies a motion of the robot to be a goal location using either straight-line Cartesian or joint-interpolated trajectories. You can add other path-points.

The Round-Robin Statement

The round-robin statement defines a round-robin or a circular list in a file. RSL uses round-robins to pass information between control levels. When one control level writes a record to the next lower control level, the lower level reads the record previously written. For a more detailed explanation of round-robins, see Section 10.2, "RSL Data Structures" in Chapter 10.

Example RSL Code

This section gives some simple examples of RSL code, and explains what each example does.

```
     -pose - TABLE 0.04361     0.999     0.0005033     0.00166
                   -39.83      -8.702    0.0216
                   1.0         1.0       -1.0
```

This code defines a pose named TABLE. The first four numbers give the orientation expressed as a quaternion. The next three give the position expressed as a Cartesian vector. The last three give the arm configuration. (For a PUMA 760, the arm configuration is lefty, elbow up, wrist flipped.) This pose defines the position and orientation of a table.

```
     -mtb- CORNER
       --- t-tool  20.0    0.0    0.0
     -mtb-end-

     -mtb- TABLE-SAFE
       --- t-tool  10.0    15.0   10.0
       --- r-tool  Z       20.0
     -mtb-end-

     -mtb- UP5
       --- t-tool  0.0     0.0    5.0
     -mtb-end-
```

This code defines three movetables, intended for use with the TABLE pose. The first movetable defines a corner of the table. The TABLE pose is at one corner of the table, with the x axis along one edge, the y axis along the other edge, and z up. CORNER is 20 inches away along the x axis. TABLE-SAFE is a point above the center of the table that is a pass-through point for motions to and from the table. The point is rotated 20 degrees about the z axis. UP5 is a point 5 inches up along the z axis. It defines an intermediate point for moving to points on the table.

```
     -loc- TABLE  TABLE  nul

     -loc- CORNER  TABLE  CORNER

     -loc- TABLE-SAFE  TABLE  TABLE-SAFE
```

This code defines locations. Locations are always the sum of a pose and a movetable. They are used in paths and path-points (poses cannot be used directly). These four locations associate the movetables defined above with the TABLE pose, indicating the appropriate positions and orientations for the table.

```
     -path- approach-pickup nul  loc CORNER
       -ppt- goto loc TABLE_SAFE joint 10.0  40.0  20.0
       -ppt- goto goal UP5  cart  0.03  0.03  1.0
                                  0.25  1.5  6.0
       -ppt- goto goal nul  cart  0.03  0.10  0.0
                                  0.25  0.5  0.0
```

This code defines a path used for moving to the corner of the table. The first path-point moves towards the location TABLE-SAFE (above the middle of the table). The trajectory is joint-interpolated, with acceleration limited to 10 percent of maximum and velocity limited to 40 percent of maximum. When all joints are within 20 degrees of the TABLE-SAFE location, the system steps to the next path-point.

The second path-point goes towards the intermediate goal defined by the location CORNER, plus the movetable UP5. Thus, the intermediate goal is TABLE + CORNER + UP5, or 5 inches above the corner of the table. The motion is a Cartesian straight line. For information on the velocity and acceleration parameters, see Appendix B, "User Word Summary". The second path-point is complete when the tool-point is within 1.0 inch and 6.0 degrees of the intermediate goal. The last path-point moves to the goal TABLE + CORNER and stops there.

9.3 ENTERING AND EDITING RSL SOURCE CODE

When you build a new application or change the data for an existing application, you need to enter or edit RSL source code. To enter new source code, type the data into a block using the RCS screen editor, described in Section 6.7, "Editing a Block of Code". Also use the screen editor to edit existing source code.

You can enter new robot poses by using the joystick to position the robot. Then enter RECORD-POSE [pose-name]. RECORD-POSE stores the current robot position in the pose with the specified pose-name. RECORD-POSE creates a new pose record if the pose-name is not already defined.

You can also enter the pose directly into a block by specifying the block number, the line number, and the name of the pose. For example, to enter SECTOR into block 23, line 5, enter 23 5 TYPE-POSE SECTOR. TYPE-POSE enters the pose into the block beginning at the line specified.

9.4 COMPILING RSL SOURCE CODE

Like SMACRO code, you compile RSL code by loading the blocks containing the source code. The compiling words in RSL compile the code into a linked list representation in common memory files. Like SMACRO code, you can recompile all RSL code; you may edit and recompile any piece of code, and any other code that uses it will use the new version.

Unlike SMACRO, RSL has garbage collection; a path may be recompiled several times without wasting any space in the files. Other named data structures have no garbage collection; you cannot remove a pose, for example. Thus, you may run out of space if you keep defining more and more data structures such as poses and locations. To avoid running out of space, restore the system to a clean state and recompile all the RSL source code.

Recompiling code is easier with RSL than with SMACRO; to restore the system to a clean state, you simply reinitialize all the files using ~re-init-file (rather than using D>M and RESTORE on several boards). Block 990 on RSL is set up to do this restoration, as well as reload the RSL code. (You must edit block 994 and your directory blocks to be sure all of your code is loaded.) Note that the control system must not be running the robot when the RSL code is completely reloaded; 990 executes HALT to guarantee that the robot is not running. Small pieces of code may be loaded with the control system running, as long as the control system is not actually using that code. To be safe, HALT the system before loading RSL code.

If you run out of file space (indicated by a "FILE FULL" message), but do not have extra code loaded, you can increase the number of records allotted to each file by editing the file declaration (in the 100 block section on the RSL board). You must recompile the SMACRO code for all levels and the RSL code for this change to take effect. You may run out of the common memory space allotted to files; type F-MEM to display the amount of memory left. To change the allocation of common memory, edit absolute block 1402 and reload the base and application code.

In general, RSL checks for errors and gives appropriate error messages. (The appendices of this manual do not list these messages.) If you get an error message, simply edit the block to correct the error, then reload the code. You do not have to use the RESTORE or D>M commands to fix errors with RSL.

When you load block 990 on RSL, it reloads the initialization blocks for each level after compiling all the RSL source code. This is done because the levels use several reserved records in RSL (mainly the round-robins) and must look up record numbers, which may have changed as a result of recompilation.

9.5 EXECUTING RSL COMMANDS

To execute RSL commands, you load a block containing a command for the TASK level. For example, move to the TASK&PATH board and use the directory block structure to locate the TRANSFER command. Listing block 0 shows that user commands are defined in the 900 section. Listing block 900 shows that the input commands for the task level are in the 910 section. Listing block 910 shows that the TRANSFER command is in block 913.

Load block 913. RCS executes the TRANSFER command specified in block 913. You can also execute the TRANSFER in Show mode, in which case RSL displays the status-report, status-arg-out, and joint angles for each level as the robot executes the transfer.

To see the status of the command being executed, load block 928 on RSL. This block displays the status-report and status-arg-out variables for each level. A status-report of Ø means the command is executing, of 1 means the command is done, and of 2 means an error occurred. The status-arg-out variable gives the error types. (Refer to Chapter 10, "RSL Control Levels", for a description of these errors.) Alternatively, you can display the variable error-list, the owner of the error-report variables, on each level.

To issue another command, simply load another block. Blocks 91Ø through 979 on TASK are available for user commands. Most RSL commands are interruptible. If you issue a new command, the old command aborts and the new command starts executing. To suspend and then resume command execution, press the HOLD-SET button on the joystick. The HOLD-SET button causes the robot to stop moving and command execution to be suspended. To resume execution, press the HOLD-CLEAR button.

Note that the GO and HALT commands are not commands to the control system. The GO and HALT commands start and stop the control system.

The first three sections of this chapter provide an overview of RSL and describe the RSL data structures and compiler. The last four sections describe TASK, PATH, PRIM, and JOINT, the four hierarchical control levels that make up RSL. These control levels decompose each task you enter at the keyboard into a series of primitive commands that a robot can execute to accomplish a task. The sections describe the function of each control level and provide command descriptions, status information, variable descriptions, control-level routine descriptions, and a discussion of the error conditions for each control level.

10.1 OVERVIEW OF RSL CONTROL LEVELS

The RSL system contains five main segments of code: the RSL compiler and the four control levels TASK, PATH, PRIM, and JOINT. Each segment has its own vocabulary. This chapter refers to segments by using the vocabulary name for each segment. (The context of the discussions distinguishes RSL as the whole system from RSL as the compiler.)

Figure 10-1 gives an overview of the relationships between RSL and the control levels.

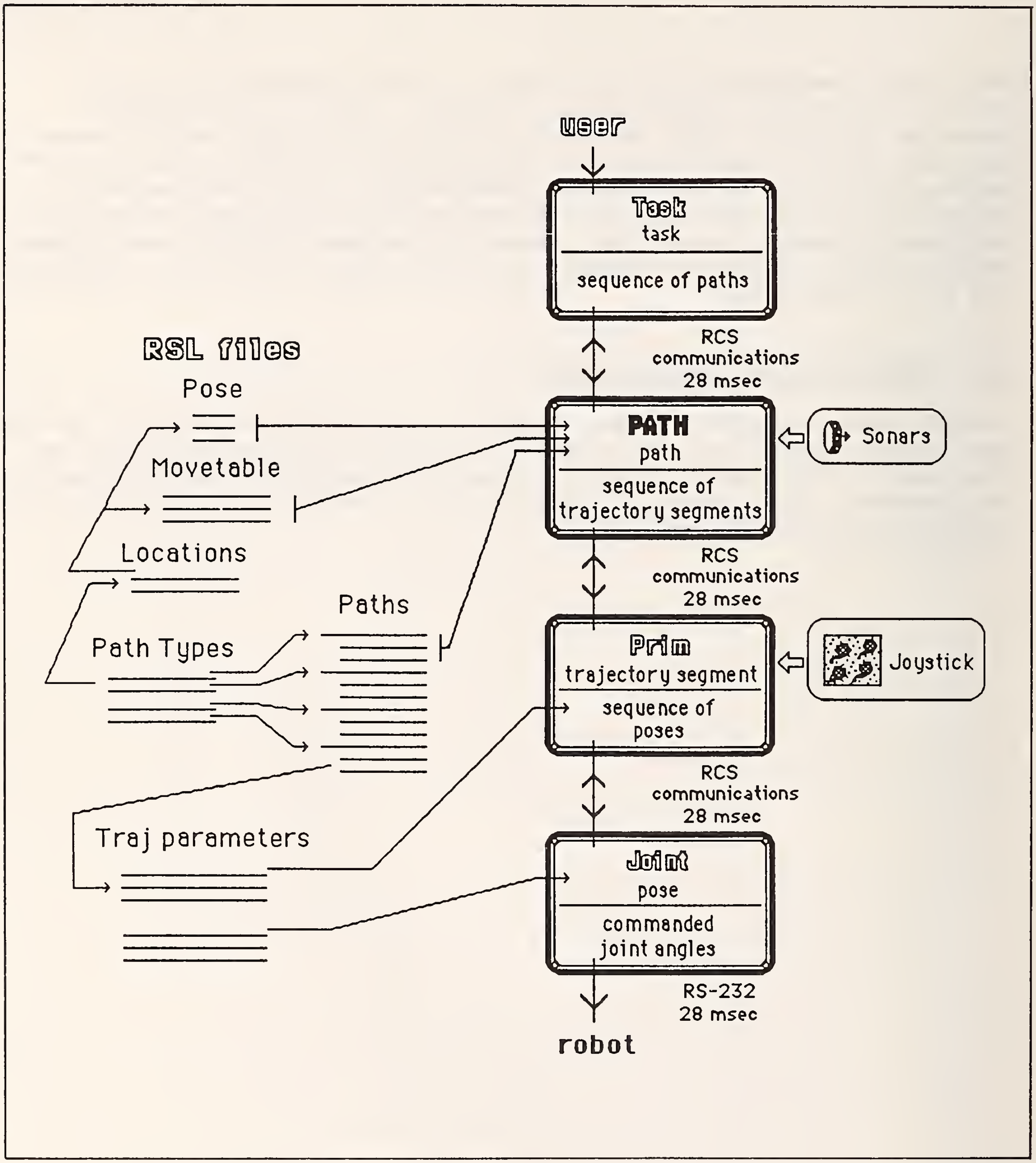


Figure 10-1. RSL files and control levels.

RSL compiles environmental data and algorithms into files in common memory. The control levels use the files to execute commands. The TASK level decomposes user commands into a series of paths. The PATH level decomposes those paths into path-points and then into trajectories. The PRIM level executes Cartesian trajectories, by sending a new pose to the JOINT level every cycle. The JOINT level transforms the poses to joint values, and sends them to the servo controls in the manufacturer's robot controller. For joint trajectories, PRIM sends the command to JOINT, which executes it, sending new joint values to the servo mechanisms every cycle.

The rest of this chapter gives an introductory description of the source code for all five segments and describes the RSL data structures. The source code descriptions introduce you to the overall structure of RSL code.

The descriptions of individual routines are not exact translations of the SMACRO code into English. Rather, they describe the functions of the routines. When you understand the descriptions in this chapter, you should be able to understand the source code without the functional descriptions.

The internal structures of the four control levels are similar, but they are not exactly the same. Each section of this chapter begins with a description of the input commands and status feedback for the level, followed by an overview of the level's structure, the preprocess routines, the command-process routines, and the postprocess routines.

Note that when a variable owner is given as an input or output for a routine, the members of the owner are the actual inputs and outputs.

The command descriptions include the command syntax. Commands to the TASK level are different from those to the other levels. The TASK input is an ASCII string; the other levels accept numerical data. The user never gives commands directly to the lower levels, only to the TASK level. So the TASK level needs to be very flexible and user-friendly. The lower levels have less flexible inputs and use fixed record descriptions. Thus, the syntax descriptions for the lower levels only indicate which of the fields in the input command record are actually used by that command.

The command syntax shows parameter names enclosed in square brackets. When you enter the commands, separate the parameters with spaces.

10.2 RSL DATA STRUCTURES

RSL represents data structures and algorithms as a linked list of records, kept in files in common memory. These files are the user files described in earlier chapters of this manual. The data within these files resides in the user file portion of common memory.

Typically, the records contain pointers to other files, numeric data, and names. For linear displacement values, the unit of measurement is inches. For angles, the unit of measurement is degrees. Internally, RSL represents angles as radians. Names in RSL can have up to 15 ASCII characters, excluding spaces.

RCS provides each file with a free list of records for garbage collection, and SMACRO file operators to maintain the free list.

The following sections explain the RSL data structures: pose, movetable, location, array, object, path, path-point, trajectory, and round-robin. Each section describes the record structure of a file for the RSL data structure type. RSL uses the RCS language SMACRO to define the data files. SMACRO provides the linked-list programming structure used for RSL files. A SMACRO variable owner defines the record, with each member of the owner defining a field in the record. The SMACRO file declaration specifies the maximum number of records allowed in the file. You may change this number to fit the size of your application.

The Pose Data Structure

The pose file in common memory has the following structure:

```
55 bytes VAR-0 pose-rec
   15 strv "pose-name"
   7 1:fa  pose
   3 1:fa  conf-flag

50 rec FILE POSE-FILE
```

where:

- "pose-name" contains the name of the pose.

- pose contains the seven numbers giving the rotation quaternion and the translation vector for the pose.

- conf-flag contains the three robot arm configuration flags.

The pose file is a single linked list. When you define a new pose, RSL adds the pose to the end of the list. RSL does not include a provision for removing poses. When you reload the source code for a pose, RSL updates all the fields.

The Movetable Data Structure

The movetable file in common memory has the following structure:

```
127 bytes VAR-0 mtb-rec
   15 strv "mtb-name"
   iv 1st-line-type 3 1:fa 1st-line-para
   iv 2nd-line-type 3 1:fa 2nd-line-para
   iv 3rd-line-type 3 1:fa 3rd-line-para
   iv 4th-line-type 3 1:fa 4th-line-para
   iv 5th-line-type 3 1:fa 5th-line-para
   iv 6th-line-type 3 1:fa 6th-line-para
   iv 7th-line-type 3 1:fa 7th-line-para
   iv 8th-line-type 3 1:fa 8th-line-para

100 rec FILE MOVETABLE-FILE
```

where:

- "mtb-name" contains the name of the movetable.

- 1st-line-type through 8th-line-para contain the lines of the movetable. The first integer in each line contains a code number that identifies the line type, and the next three floating-point numbers contain the parameters for that type of line. Some line types require less than three parameters. (A movetable may contain up to eight lines.) Table 10-1 lists the line types and parameters.

  Note: The line types given in Table 10.1 are used for the internal representation only and are not the same as the line types in the syntax for defining movetables.

Table 10-1. Movetable Line Types.

Code	Line Type	Parameters
0	nul	none.
1	tool-trn	x, y, z components of the vector in the tool frame.
2	base-trn	x, y, z components of the vector in the base frame.
3	tool-rotx	sine and cosine of half the angle about the x axis of the tool frame.
4	tool-roty	sine and cosine of half the angle about the y axis of the tool frame.
5	tool-rotz	sine and cosine of half the angle about the z axis of the tool frame.

The movetable file is a single linked list. When you define a new movetable, RSL adds the movetable to the end of the list. RSL does not include a provision for removing movetables. When you reload the source code for a movetable, RSL updates all the fields.

The Location Data Structure

The location file in common memory has the following structure:

```
19 bytes VAR-O loc-rec
   15 strv "loc-name"
   iv      loc-pose-^
   iv      loc-mtb-^

20 rec FILE LOCATION-FILE
```

where:

- "loc-name" contains the name of the location.

- loc-pose-^ contains a pointer to the pose for "loc-name" in the POSE-FILE. RSL sets the pointer when it defines the location.

- loc-mtb-^ contains a pointer to the movetable for "loc-name" in the MOVETABLE-FILE. RSL sets the pointer when it defines the location.

The location file is a single linked list. When you define a new location, RSL adds the location to the end of the list. RSL does not include a provision for removing locations. When you reload the source code for a location, RSL updates all the fields.

The Array Data Structure

The array file in common memory has the following structure:

```
88 bytes VAR-O arr-rec
   1Ø strv "arr-name"
   iv      base-loc-^
   iv      x-#-sectors
   iv      y-#-sectors
   iv      z-#-sectors
   iv      #-sectors
   iv      x-mtb-^
   iv      y-mtb-^
   iv      z-mtb-^
   2Ø 1:a  sector-list
   iv      arr-pose-^

1Ø rec FILE ARRAY-FILE
```

where:

- "arr-name" contains the name of the array.

- base-loc-^ contains a pointer to the location in LOCATION-FILE that is the origin of the array.

- x-#-sectors, y-#-sectors, and z-#-sectors contain the number of sectors along the x, y, and z axes of the location frame.

- #-sectors contains the total number of sectors.

- x-mtb-^, y-mtb-^, and z-mtb-^ contain pointers to the movetables in MOVETABLE-FILE for the displacement from one sector to the next sector along the respective axes of the location frame. The movetables should have only lines of type tool-trn.

- sector-list contains a list of sectors giving the order in which RSL loads or unloads the array.

- arr-pose-^ contains a pointer to the pose that gives the position of the current sector during loading or unloading.

All fields, except sector-list, are set when RSL defines the array. TASK sets the field sector-list when it receives a TRANSFER command to load or unload the array.

RSL reserves a pose record for arr-pose-^ when you declare the array. TASK calculates the value of the pose from the base location and sector movetables at the beginning of execution of a path involving the sector as the source or destination.

RSL numbers sectors starting at the origin of the location frame, sector Ø. Then it increments along the x, y, and z axes of the location frame, incrementing x first, y second, and z third.

The array file is a single linked list. When you define a new array, RSL adds the array to the end of the list. RSL does not include a provision for removing arrays. When you reload the source code for an array, RSL updates all the fields, clearing the sector-list.

The Object Data Structure

The object files in common memory have the following structure:

```
17 bytes VAR-O obj-name-rec
   15 strv "obj-name"
   iv      obj-grip-L-^

5 rec FILE OBJ-NAME-FILE

4 bytes VAR-O obj-grip-rec
   iv     grip#
   iv     grip-mtb-^

2Ø rec FILE OBJ-GRIP-FILE
```

where:

- "obj-name" contains the name of the object.

- obj-grip-L-^ contains a pointer to a list of the grips for each object in OBJ-GRIP-FILE.

- grip# contains the grip number of the object.

- grip-mtb-^ contains a pointer to a movetable that defines the grip in MOVETABLE-FILE.

Typically, grip 1 has a nul movetable and RSL defines other grips relative to grip 1.

OBJ-NAME-FILE contains a linked list of all object names. OBJ-GRIP-FILE contains a list for each object. RSL does not include a provision for removing objects or object grips. When you reload the source code that defines a grip definition, RSL updates the field grip-mtb-^ and preserves all other fields.

The Path Data Structure

The central path file in common memory is the PATH-POINT-FILE. The PATH-POINT-FILE has the following structure:

```
     4 bytes VAR-O ppt-rec
        iv   ppt-command
        iv   ppt-para^

     1ØØ rec FILE PATH-POINT-FILE
```

where:

- ppt-command contains the path-point command.

- ppt-para^ contains a pointer to the command parameter file.

The path-point file contains one list for each defined path. RSL does not include a provision for removing paths. When you reload a path, RSL returns the current list for that path to the free list and creates a new list. RSL also returns the associated records in the command parameter file to their free lists and creates new lists as needed. You may redefine paths as often as you like without wasting file space.

Each path-point command has one command parameter file. This file contains any parameters related to the path-point. RSL provides only the goto path-point, and thus, only the GOTO-FILE command parameter file.

The move-to path type

The move-to file in common memory has the following structure:

```
     12 bytes VAR-O move-to-rec
        iv    move-to-obj-^
        iv    move-to-start-type
        iv    move-to-start-^
        iv    move-to-dest-type
        iv    move-to-dest-^
        iv    move-to-path-^

     4Ø mem FILE MOVE-TO-FILE
```

where:

- move-to-obj-^ contains a pointer to the object grip record in OBJ-GRIP-FILE.

- move-to-start-type contains the starting location type. The location type must be either loc or arr.

- move-to-start-^ contains a pointer to the starting location in LOCATION-FILE or ARRAY-FILE.

- move-to-dest-type contains the destination location type (loc or arr).

- move-to-dest-^ contains a pointer to the destination.

- move-to-path-^ contains a pointer to the path in the PATH-POINT-FILE.

The move-to file is a single linked list. A new record is added to this file when a move-to path for a new object/start location/destination is defined. RSL does not include a provision for removing paths. When you reload a move-to path, RSL updates the field move-to-path-^ to point to the new path.

The structures of the other path types are similar to the structure of move-to. Because of the similarity, the following sections give the structure of the path type records without a complete description.

The approach-pickup path type

The approach-pickup file in common memory has the following structure:

```
8 bytes VAR-O approach-pickup-rec
   iv   appr-pick-loc-type
   iv   appr-pick-loc-^
   iv   appr-pick-obj-^
   iv   appr-pick-path-^

4Ø mem FILE APPROACH-PICKUP-FILE
```

The depart-pickup path type

The depart-pickup file in common memory has the following structure:

```
8 bytes VAR-O depart-pickup-file
   iv   dept-pick-loc-type
   iv   dept-pick-loc-^
   iv   dept-pick-obj-^
   iv   dept-pick-path-^

4Ø mem FILE DEPART-PICKUP-FILE
```

The approach-release path type

The approach-release file in common memory has the following structure:

```
     8 bytes VAR-O approach-release-rec
        iv   appr-rel-loc-type
        iv   appr-rel-loc-^
        iv   appr-rel-obj-^
        iv   appr-rel-path-^

     40 mem FILE APPROACH-RELEASE-FILE
```

The depart-release path type

The depart-release file in common memory has the following structure:

```
     8 bytes VAR-O depart-release-file
        iv   dept-rel-loc-type
        iv   dept-rel-loc-^
        iv   dept-rel-obj-^
        iv   dept-rel-path-^

     40 mem FILE DEPART-RELEASE-FILE
```

The Path-Point Data Structure

The goto file in common memory has the following structure:

```
     8 bytes VAR-O goto-rec
        iv   goto-loc-type
        iv   goto-loc-^
        iv   goto-traj-type
        iv   goto-traj-^

     100 rec FILE GOTO-FILE
```

where:

- goto-loc-type contains the type of location.

- goto-loc-^ contains a pointer to the location.

- goto-traj-type contains the trajectory type: cart or joint.

- goto-traj-^ contains a pointer to the trajectory parameters in CARTESIAN-FILE or JOINT-FILE.

The goto file is a single linked list. When you define a goto path-point, RSL adds the goto path-point to the end of the list. RSL removes the goto path-point whenever a path containing goto path-points is deleted as part of redefinition.

The Trajectory Data Structure

The files for the Cartesian and joint trajectory types in common memory have the following structure:

```
24 bytes VAR-O cart-rec
   fv    amax
   fv    vmax
   fv    trn-nbrhd
   fv    gmax
   fv    wmax
   fv    rot-nbrhd

100 rec FILE CARTESIAN-FILE
```

where:

- amax contains the maximum allowed magnitude of the translational acceleration, in inches per cycle per cycle.
- vmax contains the maximum allowed magnitude of the translational velocity, in inches per cycle.
- trn-nbrhd contains the translational distance in inches from the current goal location at which the trajectory routine reports done.
- gmax contains the maximum allowed magnitude of the rotational acceleration, in degrees per cycle per cycle.
- wmax contains the maximum allowed magnitude of the rotational velocity, in degrees per cycle.
- rot-nbrhd contains the rotation neighborhood, the rotational distance in degrees from the current goal location at which the trajectory routine reports done.

```
12 bytes VAR-O joint-rec
   fv    j-acc
   fv    j-vel
   fv    j-delta

100 rec FILE JOINT-FILE
```

where:

- j-acc contains the maximum allowed acceleration of any joint, given as the percentage (between 0 and 100) of the hardware maximum for each joint.
- j-vel contains the maximum allowed velocity of any joint, given as the percentage (between 0 and 100) of the hardware maximum for each joint.

- j-delta contains the difference in degrees from the location joint values at which the trajectory routine reports done. When the trajectory reports done, all joint values are within this delta.

Each file is structured as a single linked list. RSL adds records to the appropriate list whenever a path-point is defined. RSL removes records whenever a path-point containing the trajectory is deleted as part of a path redefinition.

The Round-Robin Data Structure

A round-robin is a circular list in a file, called the containing file. Use round-robins to store information temporarily. For example, if you are using a slow sensor, such as a vision sensor that may require up to 15 cycles to take a reading, use a round-robin to store robot poses for the previous 15 cycles. This technique enables the sensor to access the position of the robot at every cycle since the sensor started the reading.

RSL also uses round-robins to pass information between control levels. When one control level writes a record to the next lower level, the lower level reads the record written previously.

As illustrated in Figure 1Ø-2, for example, in cycle 1 PATH writes the goal pose to record 1 and puts the pointer in goal-pose-^-out. Also in cycle 1, PRIM reads the pose from record 3 (put there by PATH in the last cycle). This pose is pointed to by goal-pose-^-in. After cycle 1, COMM moves PATH's output command buffer (containing goal-pose-^-out) to PRIM's input command buffer (containing goal-pose-^-in). Then in cycle 2, PATH writes the goal pose to record 2 and PRIM reads from record 1.

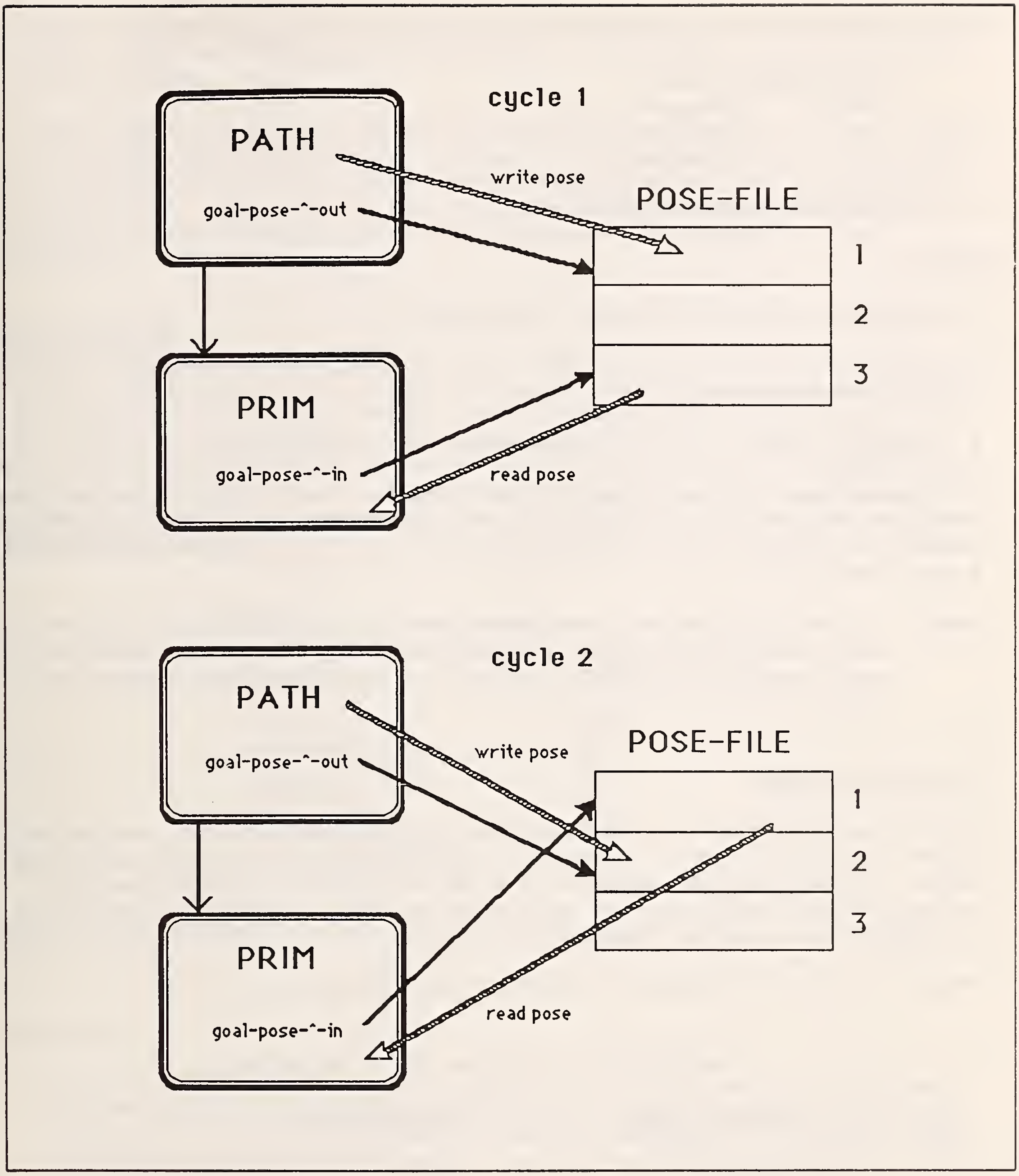


Figure 1Ø-2. Using a round-robin to transfer information between control levels.

The round-robin file in common memory has the following structure:

```
     17 bytes VAR-O r-rec
        15 strv   "rr-name"
        iv         rr-^
        iv         rr-size

     10 mem FILE ROUND-ROBIN-FILE
```

where:

- "rr-name" contains the name of the round-robin.

- rr-^ contains the record number of the beginning of the circular list in the containing file.

- rr-size contains the number of records in the round-robin.

The round-robin file is a single linked circular list. RSL does not include a provision for removing round-robins. When you redefine a round-robin, RSL returns the records in the containing file to the free list, and then creates a new circular list.

Note: The containing file is not identified by any field in this record. You must manually keep track of which containing file is associated with each round-robin. In RSL, the name of the round-robin denotes the containing file.

10.3 RSL COMPILER

The RSL compiler compiles RSL source code describing locations, movetables, objects, and paths into a linked-list representation, stored in common memory files. The RSL compiler consists of a set of routines that interpret the source code and build the data and algorithm files. Each data type has its own compiling routine or routines.

Table 10-2 lists the compiling words for each data type.

Table 10-2. Data Type Compiling Words.

RSL Data Type	Compiling Word
pose	-pose-
movetable	-mtb-, ---, -mtb-end-, -imtb-
location	-loc-
array	-arr-
object	-obj-
path	-path-, -ppt-
round-robin	-rr-

Note that the compiling word is an abbreviation of the data type name enclosed by dashes. RSL uses these same abbreviations consistently when naming variables and routines.

The rest of this section describes some RSL compiling routines. If you extend RSL to include new path-points, path types, or trajectory types, you have to add new compiling routines, similar to the ones described here to accommodate the extensions. These compiler routines call utility routines to perform common functions. For more information on extensions see Chapter 11, "RSL Extensions".

Utility Routines

Each file that contains a data type name has an associated search routine that reads a name from the input stream, stores the name in the name field of the local template for the file, and searches the file for that name. If the search routine finds the name in the file, the routine sets the record# to the found record. If the search routine does not find the record name in the file, the routine sets the record# to Ø. Table 1Ø-3 lists the files and the associated routines:

Table 1Ø-3. Data Type Search Routines.

RSL Files	Routines
POSE-FILE	POSE-^
MOVETABLE-FILE	MTB-^
LOCATION-FILE	LOC-^
ARRAY-FILE	ARR-^
OBJ-NAME-FILE	OBJ-^
ROUND-ROBIN-FILE	RR-^

SAME-BLOCK? is an example of a utility routine called by many of the other compiling routines.

SAME-BLOCK?

> The SAME-BLOCK? utility routine compares the current source block number with the block number stored in the current record. If the numbers are different, the routine displays the message **redefined in different block,** and displays the old block number. This message should prevent you from using the same name twice.

RSL includes the location types loc, arr, tool, and goal. When you add location types, you need to add to the routine READ-LOC-PHRASE, which is defined next.

READ-LOC-PHRASE [loc type] [loc name]

The READ-LOC-PHRASE utility routine stores the location type in the variable loc-type. Then it searches the corresponding file for loc name and stores the record# in loc-^. READ-LOC-PHRASE aborts execution if it does not find the location. Because READ-LOC-PHRASE does not add records to a file, it does not have a STORE or REMOVE routine.

Environmental Data Routines

Environmental data routines compile information about objects in the environment such as the position of an object relative to the robot. The routine -pose- is described here as an example.

-pose- [name] [pose numbers] [configuration flags]

This environmental data routine calls POSE-^ to read the pose name and check that it is not already defined. Then -pose- calls #.READ 1Ø times to read in the pose numbers and configuration flags, storing them in the corresponding fields in the POSE-FILE local record, pose-rec. If the pose is already defined, it calls SAME-BLOCK? and stores the local template to the found record. Otherwise, it adds the local template to the list in POSE-FILE, pointed to by the variable first-record.

Routines to Compile Trajectories

READ-TRAJ-PHRASE, STORE-TRAJ-PHRASE, and REMOVE-TRAJ-PHRASE are the highest level compiler routines used with trajectories. Use these routines when you add other trajectory types to RSL.

READ-CART-TRAJ [cart traj parameters]

READ-CART-TRAJ is an example trajectory type routine that calls #.READ five times to read in the Cartesian trajectory parameters, storing them in the local template of CARTESIAN-FILE.

READ-JOINT-TRAJ [joint traj parameters]

READ-JOINT-TRAJ is an example trajectory routine that calls #.READ three times to read in the joint trajectory parameters, storing them in the local template of JOINT-FILE.

READ-TRAJ-PHRASE [traj type] [traj parameters]

READ-TRAJ-PHRASE is a compiling routine that calls #READ to read in the trajectory type, storing it in traj-type. READ-TRAJ-PHRASE aborts if the trajectory type is invalid. READ-TRAJ-PHRASE also executes a case statement to call a routine to read the trajectory parameters for the type.

STORE-TRAJ-PHRASE

STORE-TRAJ-PHRASE is a compiling routine that stores the traj-type local template to its file, adding to the list of file records. STORE-TRAJ-PHRASE sets traj-^ to the record# stored. The path-point routine calls this routine after checking the rest of the path-point syntax.

REMOVE-TRAJ-PHRASE

REMOVE-TRAJ-PHRASE is a compiling routine that deletes the trajectory parameter record pointed to by the variable traj-^ from the traj-type file.

Routines to Compile Path-Points

RSL compiles paths in two stages. First, RSL compiles the path and then the path-points. The routine -path- compiles the identifying information for the path, and creates a list in PATH-POINT-FILE. The routine -ppt- compiles the path-points, adding them to the list in PATH-POINT-FILE.

You must provide two routines for each path-point you add to RSL; one that compiles the path-point and one that removes it for garbage collection. By convention, the path-point command consists of lowercase characters, the compiling routine has the same name in uppercase characters, and the removing routine has the same name in uppercase characters with -REMOVE appended to the end. For example, the goto path-point command is goto, the compiling routine is GOTO, and the removing routine is GOTO-REMOVE.

The compiling routine must read in all the fields of the path-point, checking that any referred-to items, such as movetables and locations, are defined before modifying any files. Then the compiling routine stores the results in the appropriate files. One of these files must be the primary path-point parameter file. The record# of the record added to that file must be put in the variable ppt-para-^ to be stored in PATH-POINT-FILE by the routine -ppt-.

You must add the new compiling routine to the case statement in -ppt-, and you must add the removing routine to the case statement in PPT-REMOVE. Use the directory block system in the source code to find empty blocks in which to write your routines.

The compiling routines for the RSL path-point goto are described below:

GOTO [loc phrase] [traj phrase]

This routine calls READ-LOC-PHRASE and copies loc-type and loc-^ into the variables goto-loc-type and goto-loc-^. (These variables are fields in the GOTO-FILE local template goto-rec.) GOTO then calls READ-TRAJ-PHRASE and copies traj-type into goto-traj-type. At this point, all fields have been read and any errors detected so the records may be written to the files. GOTO writes the records to the files by calling STORE-TRAJ-PHRASE, copying traj-^ to goto-traj-^. Then GOTO adds the local template to GOTO-FILE and sets ppt-para-^ to the record# stored.

GOTO-REMOVE

This routine retrieves the record from GOTO-FILE pointed to by ppt-para-^, copies the variables goto-traj-type and goto-traj-^ into traj-type and traj-^, calls REMOVE-TRAJ-PHRASE, and removes the record from GOTO-FILE.

-ppt- [ppt command] [ppt parameters]

This routine stores the ppt command type in ppt-command, executes a case statement to call the corresponding compiling routine, then adds the ppt record to the current path in PATH-POINT-FILE. (Note that -path- puts an invalid ppt record at the beginning of the list when it is created. Therefore, -ppt- checks first-ppt to see if it should store to that record or add a record to the list.)

Routines to Compile Paths

Each path type you add to RSL must have enough parameters to identify individual paths of that type. You provide a routine that reads the identifying parameters, creates a list in PATH-POINT-FILE, and stores the data to the path-type file. When a path is redefined, you must call PATH-CLEAR to delete the old path from PATH-POINT-FILE and the path-point command parameter files. You must also modify the path files. You must add your routine to the case statement in -path-.

Two typical parameters that identify a path are locations and objects. The utility routine READ-LOC-PHRASE may be used to read location phrases. A routine used to read object phrase, a compiling routine for the system path type move-to, and some associated routines are described below.

READ-OBJ-PHRASE [obj name] [grip#]

This routine searches the object files for the name and grip number. If they are not found, the routine aborts. Then the routine sets obj-grip-^ to the record# in the OBJ-GRIP-FILE.

PATH-CLEAR

This routine deletes the path in PATH-POINT-FILE starting at path-^, executing PPT-REMOVE for each path-point. PPT-REMOVE deletes all records added to path-point parameter files for the path.

MOVE-TO [obj phrase] [start loc phrase] [destination loc phrase]

This routine calls READ-OBJ-PHRASE and copies obj-grip-^ to move-to-obj-^. MOVE-TO then calls READ-LOC-PHRASE, and copies loc-type and loc-^ into move-to-start-type and move-to-start-^. Then MOVE-TO calls READ-LOC-PHRASE again, and copies loc-type and loc-^ into move-to-dest-type and move-to-dest-^. MOVE-TO has now read all input fields, and is ready to write the data to the files.

MOVE-TO searches MOVE-TO-FILE for the object, start location, and destination location. If a match is found, the old path is removed. If not found, a record is added to the MOVE-TO-FILE list. In either case, path-own-^ is set to the record# in MOVE-TO-FILE, so that move-to-path-^ may be set later.

After the correct MOVE-TO-FILE record is found, a list is created in PATH-POINT-FILE, and move-to-path-^ is set. The ppt-command is set to -1 for a path with no path-points. If such a path is redefined, PPT-REMOVE does not abort.

-path- [path-type] [parameters]

This routine sets path-type to the path type. It then executes a case statement to execute the corresponding routine, which reads the parameters and creates a list in PATH-POINT-FILE. The variable first-ppt is set to true, so that -ppt- stores to the first record in the list, rather than adding a record to the list.

10.4 THE TASK LEVEL

The TASK level interprets and executes robot task commands entered at the keyboard, or more commonly, loaded from a block. You can define your own tasks using state tables, or you can use the MOVE-TO and TRANSFER tasks provided on your system tape. You must use the TASK utility routines when you create routines to perform other tasks.

TASK Commands

The following commands execute at the TASK level: RESTART, PAUSE, MOVE-TO, and TRANSFER. This section explains the function and provides the syntax of each command.

RESTART command

The syntax of the RESTART command is:

```
RESTART
```

RESTART sends RESTART to PATH, and sets the current location and object to nul. After sending the RESTART command, TASK reports its status as executing, until the PATH level reports done. Then TASK level also reports its status as done.

PAUSE command

The syntax of the PAUSE command is:

```
PAUSE
```

PAUSE sends PAUSE to the PATH level, and sets the current location and object to nul. After you issue PAUSE, the TASK level reports its status as executing until the PATH level reports its status as done. Then the TASK level also reports its status as done.

MOVE-TO command

The syntax of the MOVE-TO command is:

```
MOVE-TO [object] [grip#] [destination location type] [destination]
     [sector]
```

where:

- object specifies the name of the object you are directing RSL to move.
- grip# specifies the number of the grip for the specified object.
- destination location type can be array (arr) or location (loc).
- destination specifies the destination for the MOVE-TO command.
- sector specifies the array sector containing the destination. This field is included only if the destination location type is an array (arr).

MOVE-TO sends a single move-to path to the PATH level. MOVE-TO uses the current location as the start location. The destination becomes the current location when the path is complete.

MOVE-TO reports an error if it does not find the path. After you issue MOVE-TO, the TASK level reports its status as executing until the PATH level reports its status as done. Then the TASK level also reports its status as done.

TRANSFER command

The syntax of the TRANSFER command is:

```
TRANSFER [object phrase] [source location phrase] [source sector list]
     [destination location phrase] [destination sector list]
```

where:

- object phrase contains the object name and grip number.
- source location phrase contains the location type and name of the source.
- source sector list is a list of numbers, terminated by ;; and giving the sectors of the source array. This field is included only if the source location type is an array (arr).
- destination location phrase contains the location type and name of the destination.
- destination sector list is a list of numbers, terminated by ;; and giving the sectors of the destination array. This field is included only if the destination location type is an array (arr).

TRANSFER sends a sequence of path commands to the PATH level and updates the current location each time it completes the execution of a path.

TRANSFER sends the following sequence of paths, in the order shown, to the PATH level:

1. move-to [object] [source] [destination]
2. approach-pickup [object] [source]
3. depart-pickup [object] [source]
4. move-to [object] [source] [destination]
5. approach-release [object] [destination]
6. depart-release [object] [destination]

If the source or the destination is an array, RSL repeats the specified sequence of paths for each sector in the sector list. If both the source and the destination are arrays, the sector lists must have the same lengths.

After you issue TRANSFER, the TASK level reports its status as executing until the PATH level indicates that all paths are complete by reporting its status as done. Then the TASK level also reports its status as done.

TASK Input Command Buffer

The TASK input command buffer consists of the following variables:

inc-command-#-in
The user increments this integer variable each time the user issues a new command.

input-command
The user stores each new command in this 128-character string variable.

TASK Status Information

To provide status feedback, the TASK level returns an echo of the input command, and a status report.

The TASK output status buffer consists of the following five variables:

cycle-#-status-out
Returns the cycle count when TASK returns its status. The changing value of this integer variable verifies that the TASK level is running.

inc-command-#-echo-out
Returns a copy of the inc-command-#-in variable. This information enables you to know for which command TASK is returning status information.

command-echo
Returns a copy of the string stored in input-command. This information enables you to know for which command TASK is returning status information.

status-report
Returns the current status of TASK. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	executing	The current command is still executing.
1	done	The current command is finished executing.
2	error	An error occurred. (See status-arg-out.)

status-arg-out
Returns the error status. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	noerror	TASK detected no errors.
1	path-error	PATH reported an error to TASK.
2	command-error	TASK received an invalid command or parameters.
3	nul-path	TASK could not find one of the specified paths.
4	programmer	TASK detected a programming error.

TASK Variables

This section describes internal TASK variables.

t->IN
TASK uses this variable as a pointer to the current position in the input-command buffer variable.

t-word
: This variable holds a word read from the current position in the input-command buffer variable.

t-#
: This variable holds a number read from the current position in the input-command buffer variable.

state-label
: This variable holds the label for the current state. TASK routines derived from state graphs use this variable.

cur-loc-type
: The cur-loc-type variable specifies the current location type.

cur-loc-^
: The cur-loc-^ variable points to the current location.

cur-obj-^
: The cur-obj-^ variable points to the current object.

com-loc-type
: The com-loc-type variable holds the commanded location type determined by READ-LOC-PHRASE.

com-loc-^
: The com-loc-^ variable holds the commanded location pointer determined by READ-LOC-PHRASE.

com-obj-^
: The com-obj-^ variable holds the commanded object pointer determined by READ-OBJECT-PHRASE.

search-result
: This variable owner contains the variables that hold the output of path-search routines.

goal-type, goal-^, obj-^, path-^
: These variables hold the output of path-search routines. These variables are members of the search-result variable owner.

arr-pose-^
: This variable points to the pose for the current sector in the current array. ARRAY-POSE is used as a dummy name for this pose.

TASK Errors

TASK uses the variable status as an error flag. At the beginning of each cycle, the value of status is noerror. Any routine that detects an error sets status to indicate an error. TASK notes this error, sets status-arg-out to the appropriate value, and sends a PAUSE command to the PATH level. TASK does not finish executing the current task after detecting an error.

TASK Utilities

This section provides general descriptions of the TASK utilities listed below. Use the LOC command to find the exact code for each routine on your system.

WREAD-COMMAND
#READ-COMMAND
READ-OBJ-PHRASE
READ-LOC-PHRASE
READ-SECTOR-LIST
CALC-SECTOR-POSE
WAIT-PATH-DONE
TASK=>PATH

The description of these utilities begins with a list of the input and output variables.

WREAD-COMMAND

Input Variables	Output Variables
input-command	t-word
t->IN	t->IN
	status

Reads a word, delimited by a space, from the input-command buffer variable. The routine stores the word in t-word and increments t->IN by the number of characters in the word.

#READ-COMMAND

Input Variables	Output Variables
input-command	t-#
t->IN	t->IN
	status

Reads a string, delimited by a space, from the input-command buffer variable and increments t->IN by the number of characters in the string. It then searches for the string in the system dictionary, using the current vocabulary. If the string is in the dictionary, #READ-COMMAND checks to see that it is an integer or sequential variable and, if it is, stores the value in t-#.

If the string is not in the system dictionary or if it is not an integer or sequential variable, #READ-COMMAND converts the string to an integer and stores the result in t-#. If the string is not a valid integer, #READ-COMMAND aborts.

READ-OBJ-PHRASE

Input Variable	Output Variables
input-command	com-obj-^
t->IN	status

Calls WREAD-COMMAND and #READ-COMMAND to read an object name and grip number from the input command buffer. READ-OBJ-PHRASE then finds the record# of the corresponding entry in the object data structure.

READ-LOC-PHRASE

Input Variable	Output Variables
input-command	com-loc-type
	com-loc-^
	status

Calls #READ-COMMAND and WREAD-COMMAND to read a location type and name from the input command buffer. READ-LOC-PHRASE stores the location type in the com-loc-type variable. READ-LOC-PHRASE searches the location data structure for the name and sets com-loc-^ to the record# of the record containing the name.

READ-SECTOR-LIST

Input Variables	Output Variables
input-command	sector-list
com-loc-^	status

Calls #READ-COMMAND until it reads the entire sector list, which is terminated by ;;.

CALC-SECTOR-POSE

Input Variables	Output Variable
sector	ARRAY-POSE
arr-rec	

Calculates the pose for the sector of the array in arr-rec and stores the pose in the RSL data structure POSE-FILE at the position pointed to by arr-pose-^.

WAIT-PATH-DONE

Input Variable	Output Variables
path-status	status-report
	cur-var

Checks if the path-status variable indicates done and, if so, sets the status-report variable to done, sets cur-loc-type to equal goal-loc-type, sets cur-loc-^ to equal goal-loc-^, and sets cur-obj-^ to equal obj-^-out. WAIT-PATH-DONE does nothing if the path-status variable indicates executing.

TASK=>PATH

Input Variable	Output Variable
search-result	path-command-var
inc-command-#-out	inc-command-#-out

Sets the value of output-command to PATH, copies the value of search-result to the corresponding fields in path-command-var, and increments the value of inc-command-#-out.

TASK Processing

The three stages for the TASK level are PRE-PROCESS, COMMAND-PROCESS, and POST-PROCESS, which correspond to the preprocessing, decision-processing, and postprocessing stages discussed in Chapter 2, "RCS Overview".

PRE-PROCESS reads the command and status inputs, and sets the flags needed by COMMAND-PROCESS. COMMAND-PROCESS, the decision-processing stage, reads the command parameters, searches for paths, and sends commands to the command buffer of the PATH level. POST-PROCESS performs the "housekeeping" required to prepare TASK for the next command.

TASK preprocessing

The TASK routine PRE-PROCESS calls READ-COMMAND and PATH-STATUS.

READ-COMMAND

Input Variables	Output Variables
input-command	"command-name"
inc-command-#-in	new-command
old-inc-command-#-in	status
	status-arg
	t->IN
	old-inc-command-#-in

Checks if the value of inc-command-#-in has changed since the last cycle and, if so, sets the status report and status-arg-out variables to indicate no error, sets the new-command variable to true, and copies the value of inc-command-#-in to old-inc-command-#-in. Calls WREAD-COMMAND to read the first word contained in the input-command buffer variable. Stores the first word in the "command-name" variable. (The "command-name" variable includes the double quotation marks.)

PATH-STATUS

Input Variables	Output Variable
status-in	path-status
inc-command-#-out	
inc-command-#-echo-in	

Verifies that inc-command-#-out matches inc-command-#-echo-in, indicating that the PATH level responded to the current command and that the value of the status-in variable is valid. PATH-STATUS then copies the value of status-in to the path-status variable. If the PATH level has not responded, PATH-STATUS sets path-status to executing.

TASK decision processing

The TASK routine COMMAND-PROCESS executes the current task routine. If the value in status-report is error, COMMAND-PROCESS executes only the RESTART command.

If neither status-report nor status indicates an error, COMMAND-PROCESS executes the task routine corresponding to the "command-name" set in PRE-PROCESS. If "command-name" does not contain a valid command, COMMAND-PROCESS sets status-report to error and sets status-arg to command-error. After COMMAND-PROCESS executes the specified task routine, it copies the value of status-arg to status-arg-out.

The following is a list of routines used in COMMAND-PROCESS for the RESTART, PAUSE, and MOVE-TO commands. This list does not include the routines for TRANSFER, most of which are similar to the routines for MOVE-TO.

TASK-RESTART
TASK-PAUSE
MOVE-TO
MOVE-TO-PARAM
MOVE-TO-SEARCH

The descriptions of these routines follow.

TASK-RESTART

Input Variable	Output Variables
new-command	path-command-var
inc-command-#-out	process-var
	status-report

Sends RESTART to the PATH level and waits for prim-status to indicate done.

TASK-PAUSE

Input Variable	Output Variables
new-command	path-command-var
	status-report

Sends PAUSE to the PATH level.

MOVE-TO

Input Variables	Output Variables
new-command	path-command-var
input-command	status-report

Reads the command parameters and finds the path by calling MOVE-TO-PARAM and MOVE-TO-SEARCH. If MOVE-TO-SEARCH finds the path without finding any errors, MOVE-TO calls TASK=>PATH. If new-command indicates false, MOVE-TO calls WAIT-PATH-DONE. MOVE-TO sets the status as described under the "MOVE-TO" command.

MOVE-TO-PARAM

Input Variables	Output Variables
input-command	move-to-rec
cur-loc-type	sector
cur-loc-^	ARRAY-POSE
	status
	status-arg

Reads the rest of the parameter information in the input-command buffer variable by calling READ-OBJ-PHRASE, READ-LOC-PHRASE, and #READ-COMMAND. MOVE-TO-PARAM then uses this information to set the parameters for MOVE-TO-SEARCH.

MOVE-TO-SEARCH

Input Variable	Output Variables
move-to-rec	search-result
	status
	status-arg

Searches the MOVE-TO-FILE for a record that matches move-to-rec and sets the fields of search-result if it finds a match.

TASK postprocessing

The TASK routine POST-PROCESS calls CLEAN-UP.

CLEAN-UP

Input Variable	Output Variable
input-command-var	output-status-var

Echoes the input-command and inc-command-#-in, and then sets cycle-#-status-out.

10.5 THE PATH LEVEL

The PATH level interprets and executes path and path-point commands received from the TASK level. PATH executes paths by retrieving each successive path-point and waiting for the lower levels, PRIM and JOINT, to complete the execution of the path-point. PATH executes path-points by reading the appropriate sensors and sending motion commands to the PRIM level.

PATH Commands

The following commands execute at the PATH level: RESTART, PAUSE, and PATH. This section describes the function and provides the syntax of each command.

RESTART command

The syntax of the RESTART command is:

```
RESTART
```

RESTART sends RESTART to PRIM and resets the debug variables. After issuing the RESTART command, PATH reports its status as executing until PRIM reports done. Then PATH reports done to TASK.

PAUSE command

The syntax of the PAUSE command is:

```
PAUSE
```

PAUSE sends PAUSE to the PRIM and JOINT levels. After issuing PAUSE, PATH reports done to the TASK level immediately, without requesting confirmation from PRIM and JOINT.

PATH command

The syntax of the PATH command is:

```
PATH [path-^] [goal-type] [goal-^] [obj-^]
```

where:

- path-^ points to the path to be executed.

- goal-type can be either a location (loc) or an array (arr).

- goal-^ points to the goal location or the array describing the path. Path-point commands use this information to determine paths for the PRIM and JOINT levels to execute.

- obj-^ points to the current object. Some path-point routines use this information to obtain the grip movetable or other object-dependent data.

PATH executes the path pointed to by path-^. The PATH level reports its status as executing until it completes the execution of all path-points. Then the PATH level reports done to the TASK level.

PATH Input Command Buffer

The PATH input command buffer consists of the following variables:

inc-command-#-in
TASK increments this integer variable each time it sends a new command to the PATH level.

input-command
TASK sends this integer variable to PATH to identify the command. Commands are numbered as follows: Ø-RESTART, 1-PAUSE, and 2-PATH.

path-^
TASK sends this integer pointer variable to PATH to identify the path.

goal-type
TASK sends this integer variable to PATH to indicate the type of goal location.

goal-^
TASK sends this integer pointer variable to PATH to identify the goal location.

obj-^
TASK sends this integer pointer variable to PATH to identify the current object.

PATH Status Information

To provide status feedback to the TASK level, the PATH level returns a status report, and echoes the input command and incremental command number.

PATH uses the following five variables to return status to the TASK level:

cycle-#-status-out
Returns the cycle count when PATH returns its status. The changing value of this integer variable verifies that the PATH level is running.

inc-command-#-echo-out
Echoes the integer count of new commands received by the PATH level as indicated by the inc-command-#-in variable. This information enables TASK to know for which command PATH is returning status information.

command-echo
Echoes the string stored in inc-command-#-in. This information enables you to know for which command PATH is returning status information.

status-report
Returns the current status of PATH. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	executing	The current command is still executing.
1	done	The current command is finished executing.
2	error	An error occurred. (See status-arg-out.)

status-arg-out
Returns the error status. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	noerror	PATH detected no errors.
1	prim-error	PRIM reported an error to PATH.
2	command-error	PATH received an invalid command or parameters.
3	ppt-para	PATH detected invalid ppt-command parameters.

PATH Errors

When PATH detects an error of any type, it sets the status-report variable to indicate an error, stores the label for the error in the status-arg-out variable, and sends the PAUSE command to the PRIM level. PATH does not complete the execution of a path-point after detecting an error. (The ppt-done variable remains set to false.)

PATH utility routines set the status and status-arg variables. The calling routine must set the status-report and status-arg-out variables.

PATH Processing

The three processes for the PATH level are PRE-PROCESS, COMMAND-PROCESS, and POST-PROCESS. PRE-PROCESS reads the command, status, and sensor inputs for COMMAND-PROCESS. COMMAND-PROCESS, the decision-processing stage, interprets the command and the command parameters, executes paths and path-points, and sends commands to the command buffer of the PRIM level. POST-PROCESS sends the TASK level confirmation for each command received and performs the "housekeeping" required to prepare PATH for the next command.

The output command from PATH to PRIM consists of a pose, configuration flags, trajectory type, and trajectory parameters. The path-point routines in PATH store the pose, configuration flags, and trajectory type in the variable owner prim-com-var, and the trajectory parameters in the local template of the file corresponding to the trajectory type. The POSE-ROUND and TRAJ-ROUND routines in the POST-PROCESS stage of the PATH level put the trajectory parameters into the PATH>PRIM round-robins.

PATH preprocessing

The PATH routine PRE-PROCESS calls NEW-COMMAND?, PRIM-STATUS, and TOOL-POSE-^. If you add sensors to your system, you can have PRE-PROCESS call your sensor routines also.

NEW-COMMAND?

Input Variables	Output Variables
inc-command-#-in	new-command
old-inc-command-#-in	old-inc-command-#-in
	status
	status-arg

Sets the new-command flag to true if the value of inc-command-#-in has changed since the last cycle. (That is, if inc-command-#-in does not equal old-inc-command-#-in.) NEW-COMMAND? also sets the status and status-arg variables to noerror.

PRIM-STATUS

Input Variables	Output Variables
status-in	prim-status
inc-command-#-out	
inc-command-#-echo-in	

Sets prim-status to the same value as status-in if inc-command-#-out matches inc-command-#-echo-in. This match indicates that the PRIM level responded to the current command, and that the status-in variable represents the current PRIM status. If these variables do not match, PATH sets prim-status to executing.

TOOL-POSE-^

Input Variables	Output Variables
tp-cycles	tool-pose-^
rbt-pose-^	

Sets tool-pose-^ to the best guess of the tool pose of the robot for the current cycle. For the best guess, TOOL-POSE-^ uses the pose stored in the RBT-POSE round-robin tp-cycles before the current cycle.

PATH decision processing

The PATH routine COMMAND-PROCESS reads the input command and uses a case statement to determine which of the COMMAND-PROCESS commands to execute. If the status-report variable indicates an error, COMMAND-PROCESS freezes the error state and will not execute any command other than RESTART.

See "PATH Status Information" earlier in this chapter for a description of the status information that the COMMAND-PROCESS routines return in the status-report and status-arg-out variables.

PATH-RESTART

Input Variables	Output Variables
new-command	prim-command-var
prim-status	status-report
	status-arg-out

Sends RESTART to the PRIM level and waits for prim-status to indicate done.

PATH-PAUSE

Input Variables	Output Variables
new-command	prim-command-var
prim-status	status-report
	status-arg-out

Sends PAUSE to the PRIM level.

PATH-PATH

Input Variables	Output Variables
new-command	prim-command-var
prim-status	status-report
input-command-var	status-arg

Initializes the PATH variables if new-command is true. Then, if the last path-point is complete, PATH-PATH calls NEW-PPT? to retrieve the next path-point and calls PATH-POINT to execute the path-point. If prim-status is error, PATH-PATH sets status-report to error, sets status-arg to prim-error, and does not execute PATH-POINT.

NEW-PPT?

Input Variables	Output Variables
ppt-done	ppt-rec
last-ppt	new-ppt
next-ppt-^	last-ppt
	ppt-done
	path-done
	next-ppt-^

Reads the next path-point record from the current path and sets flags ppt-done, new-ppt, and last-ppt. NEW-PPT? stores the current path-point information in ppt-rec, the PATH-POINT-FILE local template. NEW-PPT? does not retrieve the ppt command parameter record; the path-point routines (for example, GOTO) retrieve this record.

To indicate the start of a new path-point, NEW-PPT? sets new-ppt to true during the cycle in which a path-point routine is retrieving a new path-point record. NEW-PPT? sets new-ppt to false during other cycles.

The path-point routines set ppt-done to true when they finish executing each path-point. NEW-PPT? sets ppt-done to false when it retrieves the next path-point record.

NEW-PPT? sets last-ppt to true if the current path-point is the last path-point in the current trajectory.

GOAL-POSE

Input Variables	Output Variables
goal-type	prim-com-conf-flag
goal-^	pose-add
obj-^	status
	status-arg

Calculates the goal pose for the current path. If the goal-type is invalid, GOAL-POSE sets status to error and status-arg to command-error.

TRAJ-PHRASE

Input Variables	Output Variables
traj-type	prim-com-traj-type
traj-^	traj para file local template
	status
	status-arg

Sets prim-com-traj-type to traj-type and retrieves the trajectory parameters and stores them in the local template of the corresponding file. Your sensor routine can modify the trajectory parameters before PATH sends them to PRIM so that sensor path-point servo algorithms can control the velocity and acceleration of the robot.

TRAJ-PHRASE sets status to error and status-arg to ppt-para if the traj-type parameter is invalid.

LOC-PHRASE

Input Variables	Output Variables
loc-type	prim-com-pose
loc-^	prim-com-conf-flag
	status
	status-arg

Interprets a location phrase in a path-point by calculating the pose for the specified location and storing the pose in the prim-com-pose and prim-com-conf-flag variables. LOC-PHRASE sets status to error and status-arg to ppt-para if the loc-type parameter is invalid.

GOTO

Input Variables	Output Variables
ppt-rec	prim-com-var
new-ppt	traj para file local record
prim-status	ppt-done
	status
	status-arg

Executes the goto path-point. If new-ppt is true, GOTO retrieves the ppt parameters from GOTO-FILE and sets the specified pose and trajectory parameters by calling LOC-PHRASE and TRAJ-PHRASE. If new-ppt is false, GOTO waits for prim-status to indicate done and then sets ppt-done to true.

PATH postprocessing

The PATH routine POST-PROCESS sends the TASK level confirmation for each command received and performs the "housekeeping" required to prepare PATH for the next command. POST-PROCESS calls CLEAN-UP, POSE-ROUND, and TRAJ-ROUND.

CLEAN-UP

Input Variables	Output Variables
inc-command-#-in	inc-command-#-echo-out
cycle-count	cycle-#-status-out
input-command	command-echo

Echoes the command and inc-command-# to the TASK level.

POSE-ROUND

Input Variables	Output Variables
prim-com-pose	path-pose-^
prim-com-conf-flag	goal-pose-^-out
path-pose-^	PATH>PRIM-POSE round-robin

Puts prim-com-pose and prim-com-conf-flag into the PATH>PRIM-POSE round-robin.

TRAJ-ROUND

Input Variables	Output Variables
prim-com-traj-type	path-cart-^
traj-para-file-local-record	path-joint-^
path-cart-^	traj-type-out
path-joint-^	PATH>PRIM-CART round-robin
	PATH>PRIM-JOINT round-robin

Puts the traj-para-file-local-record corresponding to prim-com-traj-type into the PATH>PRIM-CART or PATH>PRIM-JOINT round-robin.

10.6 THE PRIM LEVEL

The PRIM level receives trajectories from the PATH level, transforms these trajectories into positions, and sends these positions to the JOINT level. The JOINT level converts the positions to the primitive commands sent to the robot controller.

PRIM Commands

The following commands execute at the PRIM level: RESTART, PAUSE, and TRAJ. This section explains the function and provides the syntax of each command.

RESTART command

The syntax of the RESTART command is:

```
RESTART
```

RESTART clears any error conditions, sets the hold-set variable to false, and sends RESTART to the lower level, JOINT. After executing RESTART, PRIM reports its status as executing until the JOINT level reports its status as done. When JOINT reports done, PRIM sets the current location to the feedback position, sets the current velocity and acceleration to Ø, and resets the debug variables. Then PRIM reports done to PATH.

PAUSE command

The syntax of the PAUSE command is:

```
PAUSE
```

PAUSE sends the PAUSE command to JOINT when the joystick is inactive. The PAUSE command ensures that JOINT performs its processing only when you are using the joystick. After executing PAUSE, PRIM reports its status as executing when the joystick is active, and done when the joystick is inactive.

TRAJ command

The syntax of the TRAJ command is:

```
TRAJ [traj-type] [traj-para-^] [goal-pose-^]
```

where:

- traj-type specifies the type of trajectory, either Cartesian or joint.
- traj-para-^ points to the trajectory parameters.
- goal-pose-^ points to the goal pose.

TRAJ executes the specified trajectory. The traj-type parameter can have a value of either Ø or 1 as follows:

Ø (Cartesian)
Specifies that the type of trajectory is a straight line in Cartesian space from the start pose to the goal pose. While executing a Cartesian TRAJ command, PRIM reports its status as executing until the system

detects that the actuator is within a specified range of the goal. The variables trn-nbhrd and rot-nbhrd specify this range. After the system verifies that the robot is within range, PRIM reports its status as done to PATH.

1 (Joint)
Specifies that the type of trajectory is linear interpolation in joint space from the start pose to the goal pose. While executing a joint TRAJ command, PRIM reports its status as executing until the robot reports that all joint angles are within the target range. The variable j-delta specifies the target range for joint angles. PRIM reports its status as done when the system verifies that the joint angles are within range.

Note: The joint trajectory algorithm does not decelerate to a stop at the goal. If the robot reaches the goal, it oscillates around the goal rather than stops. Use the Cartesian trajectory algorithm to stop the robot at a goal.

PRIM Input Command Buffer

The PRIM input command buffer consists of the following variables:

inc-command-#-in
PATH increments this integer variable each time it sends a new command to the PRIM level.

input-command
PATH sends this integer variable to PRIM to identify the command. Commands are numbered as follows: Ø-RESTART, 1-PAUSE, and 2-TRAJ.

traj-type-in
PATH sends this integer variable to PRIM to indicate the type of trajectory.

traj-^-in
PATH sends this integer pointer variable to PRIM to identify the trajectory parameters, which are in a file corresponding to the trajectory type in traj-type-in.

goal-pose-^-in
This integer pointer variable identifies the current pose in POSE-FILE.

PRIM Status Information

To provide status feedback to the PATH level, the PRIM level returns a status report and a feedback pose, and echoes the input command and its incremental command number.

PRIM uses the following six variables to return status to the PATH level:

cycle-#-status-out
Returns the cycle count when PRIM returns its status. The changing value of this integer variable verifies that the PRIM level is running.

inc-command-#-echo-out
Echoes the integer count of new commands received by the PRIM level as indicated by the inc-command-#-in variable. This information enables PATH to know for which command PRIM is returning status information.

command-echo-out
Echoes the string stored in inc-command-#-in. This information enables you to know for which command PRIM is returning status information.

status-report
Returns the current status of PRIM. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	executing	The current command is still executing.
1	done	The current command is finished executing.
2	error	An error occurred. (See status-arg-out.)

status-arg-out
Returns the error status. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	noerror	PRIM detected no errors.
1	joint-error	JOINT reported an error to PRIM.
2	command-error	PRIM received an invalid command or parameters.
3	hold	The HOLD-SET joystick button paused the robot.

Note: The hold status argument is not an error. PRIM sets the status-report variable to executing when you push the HOLD-SET button. You can clear this status by pushing the HOLD-CLEAR button or by sending the RESTART command to PRIM.

rbt-pose-^
Returns a pointer to the feedback pose calculated by adding the tool movetable to the JOINT feedback pose.

PRIM Trajectory Information

RSL supports two types of trajectories: Cartesian straight-line and joint interpolated. The PRIM level executes the Cartesian trajectories and the JOINT level executes the joint trajectories. Therefore, this section discusses the joint trajectories only when they affect the execution of Cartesian trajectories. See Section 10.7, "The Joint Level", for information on joint trajectories.

The differences between the two types of trajectories can cause problems when you follow one type of trajectory with the other. Ideally, the transition from the tool-point velocity and acceleration of one trajectory to the next should be as smooth as possible. To accomplish this smooth transition, PRIM must track the output of the joint algorithm and maintain current information on position and velocity.

To maintain current position and velocity data, PRIM uses the joint feedback pose to update this data for the Cartesian trajectory algorithm during every cycle of a joint trajectory motion. Because PRIM and JOINT execute on separate boards, communication timing delays the feedback pose information by one cycle. Output commands incur an additional delay of one cycle. Therefore, PRIM calculates the appropriate current position by adding twice the current velocity (in inches per cycle) to the feedback pose. PRIM uses the history of the feedback pose to calculate the approximate velocity.

Another problem arises during Cartesian motion. Because JOINT limits the joint velocity and acceleration, the actual output from JOINT may be different from the command that PRIM sends to JOINT. If JOINT reports its status as next-point or done, PRIM executes the trajectory routine to generate the next point. However, if JOINT reports its status as executing, PRIM updates the current position using the history of the feedback pose, rather than generating a new point. This procedure produces errors in the velocity and acceleration profiles, but minimizes path errors. To minimize velocity and acceleration profile errors, try to set the Cartesian trajectory parameters so that JOINT does not have to limit them.

Joystick HOLD-SET and HOLD-CLEAR Buttons

The PRE-PROCESS stage of the PRIM level sets the hold-set variable to true when you push the HOLD-SET button on the joystick controller. When the hold-set variable is true, COMMAND-PROCESS ignores the input command (except RESTART) and sends PAUSE to the JOINT level to stop the robot. In this case, PRIM sets the status-report variable to executing and the status-arg-out variable to hold. When you push the HOLD-CLEAR button, PRIM sets hold-set to false, sets new-command to true, and restarts the current input command.

PRIM Variables

This section describes internal PRIM variables.

joint-com-pose

This variable (along with joint-com-conf-flag) holds the current command that PRIM is sending to JOINT. The joint-com-pose and current-pose variables may contain different values when JOINT is executing a joint trajectory or scaling a Cartesian point to limit the motion of the joint.

joint-com-conf-flag

The joint-com-conf-flag (along with joint-com-pose) holds the current command that PRIM is sending to JOINT. The joint-com-conf-flag and current-conf-flag variables may contain different values when JOINT is executing a joint trajectory or scaling a Cartesian point to limit the motion of the joint.

current-pose

This variable (along with current-conf-flag) holds the current position of the robot. The command routine executing during the current cycle sets the current-pose variable.

current-conf-flag

The current-conf-flag variable (along with current-pose) holds the current position of the robot. The command routine executing during the current cycle sets the current-conf-flag variable.

current-vel

This six-element array holds the current velocity. The first three elements represent the rotation velocity vector and the last three elements represent the translation velocity vector. The command routine executing during the current cycle sets the current-vel variable.

last-pose

This variable holds the last position of the robot. At the beginning of each cycle, PRE-PROCESS copies the value of current-pose to last-pose. The command routines use last-pose as the current position when they build a trajectory.

last-vel

This six-element array holds the last velocity. As with the current-vel array, the first three elements of the last-vel array represent the rotation velocity vector and the last three elements represent the translation velocity vector. At the beginning of each cycle, PRE-PROCESS copies the value of current-vel to last-vel. The command routines use last-vel as the current velocity when they build a trajectory.

feedback-pose

PRIM transforms the feedback position from JOINT to the tool-point pose and stores this information in the feedback-pose variable.

feedback-conf-flag

PRIM copies the feedback configuration flags from JOINT and stores this information in the feedback-conf-flag variable.

last-feedback-pose

The last-feedback-pose variable (along with last-feedback-conf-flag) holds the last feedback position. Before updating the feedback-pose variable, PRE-PROCESS copies the value of feedback-pose to last-feedback-pose.

last-feedback-conf-flag

The last-feedback-conf-flag variable (along with last-feedback-pose) holds the last feedback position. Before updating the feedback-conf-flag variable, PRE-PROCESS copies the value of feedback-conf-flag to last-feedback-conf-flag.

hold-set

This variable indicates the state of the HOLD-SET and HOLD-CLEAR buttons on the joystick controller. The hold-set variable indicates true after you press the HOLD-SET button, and false after you press the HOLD-CLEAR button.

PRIM Errors

When PRIM detects an error of any type, it sets the status-report variable to indicate an error, stores the label for the error in the status-arg-out variable, and sends the PAUSE command to the JOINT level. Error reporting overrides status reporting. PRIM does not send any commands to JOINT after detecting an error. One exception to the above rule is if the current input command is PAUSE and JOINT is reporting a joint-limit error, PRIM does not report an error. In this case, the CREEP button on the joystick commands CREEP to JOINT, clearing the error.

PRIM Processing

The three processes for the PRIM level are PRE-PROCESS, COMMAND-PROCESS, and POST-PROCESS. PRE-PROCESS reads the command, joystick status, and JOINT status, and prepares for COMMAND-PROCESS. COMMAND-PROCESS, the decision-processing stage, interprets the command and the command parameters, and sends joint commands to the command buffer of the JOINT level. POST-PROCESS sends the PATH level confirmation for each command received and performs the "housekeeping" required to prepare PRIM for the next command.

PRIM preprocessing

The PRIM routine PRE-PROCESS calls NEW-COMMAND?, JOY-STATUS, and JOINT-STATUS.

NEW-COMMAND?

Input Variables	Output Variables
inc-command-#-in	new-command
old-inc-command-#-in	old-inc-command-#-in
	status
	status-arg

Sets the new-command flag to true if the value of inc-command-#-in has changed since the last cycle. (That is, if inc-command-#-in does not equal old-inc-command-#-in.) NEW-COMMAND? also sets the status and status-arg variables to noerror.

JOY-STATUS

Input Variables

hold-set
joystick ports

Output Variables

joy-1a
joy-1b
joy-2a
joy-2b
hold-set
joy-status
new-command

Reads the joystick ports and stores the binary image of the switches in the variables joy-1a, joy-1b, joy-2a, and joy-2b. (See Appendix F, "Joystick Schematics", for more information.) JOY-STATUS sets the joy-status variable to active if the joystick enable switch is on and a motion switch is active. JOY-STATUS sets the joy-status variable to inactive if the joystick enable switch is off or a no-motion switch is active.

JOY-STATUS sets the new-command variable to true if the hold-clear switch is active and the hold-set variable indicates true. If the hold-set switch is active, JOY-STATUS sets the hold-set variable to true. If the hold-clear switch is active, JOY-STATUS sets the hold-set variable to false.

JOINT-STATUS

Input Variables

status-in
inc-command-#-out
inc-command-#-echo-in
joint-pose-^
tool movetable

Output Variables

prim-status
feedback-pose
feedback-conf-flag
RBT-POSE round-robin

Sets joint-status to the same value as status-in if inc-command-#-out matches inc-command-#-echo-in. This match indicates that the JOINT level has responded to the current command, and that the status-in variable represents the current JOINT status. If these variables do not match, PRIM sets joint-status to executing.

If the current input command is not RESTART, JOINT-STATUS does two things. First, it adds the specified tool movetable to the joint feedback pose and places this combination in the RBT-POSE round-robin and in the feedback-pose variable. By checking for the RESTART command, PRIM avoids trying to add the tool movetable to a meaningless pose. Second, JOINT-STATUS copies the values of the feedback configuration flags to the RBT-POSE round-robin and the feedback-conf-flag variable.

PRIM decision processing

If the hold-set variable is false and the status-report variable does not indicate an error, the PATH routine COMMAND-PROCESS uses a case statement to determine which of the commands in this section to execute. If hold-set is true and input-command is not RESTART, COMMAND-PROCESS commands JOINT to pause. If the status-report variable indicates an error, COMMAND-PROCESS freezes the error state and does not execute any command other than RESTART.

PRIM-RESTART

Input Variables

new-command
joint-status

Output Variables

joint-command
inc-command-#-out
hold-set
current-pose
current-conf-flag
current-vel
debug variables

Checks if new-command is true and, if so, sets the value of the joint-command variable to joint-restart, increments inc-command-#-out, sets hold-set to false, and resets the debug variables. If new-command is false, PRIM-RESTART waits for joint-status to indicate done and then sets current-vel to zero and sets current-pose and current-conf-flag to the combination of the feedback pose and the tool movetable.

PRIM-PAUSE

Input Variables

new-command
joint-status
status-arg-in
joy-status
joy-1a
joy-1b
joy-2a
joy-2b

Output Variables

joint-com-pose
joint-com-conf-flag
joint-command
inc-command-#-out
current-pose
current-conf-flag
current-vel

Checks if new-command is true and, if so, sets the value of the joint-command variable to joint-pause, increments inc-command-#-out, and sets current-vel to Ø. If new-command is true and joy-status is active, PRIM-PAUSE increments the inc-command-#-out variable and calls JOYSTICK and STRAIGHT-LINE to interpret the joystick command. If new-command is true and joy-status is inactive, PRIM-PAUSE sets the current velocity and acceleration to Ø.

If the value of joint-status is error and the value of status-arg is joint-error when you press the CREEP button on the joystick console, PRIM-PAUSE sets the joint-command variable to joint-creep and increments inc-command-#-out. In response to other error conditions, PRIM-PAUSE sets joint-command to PAUSE.

TRAJ

Input Variables

last-pose
last-vel
traj-type-in
traj-^-in
goal-pose-^-in

Output Variables

joint-com-pose
joint-com-conf-flag
joint-command
inc-command-#-out
current-pose
current-vel

Calls CART-TRAJ or JOINT-TRAJ depending on whether the value of the traj-type-in variable is cart or joint.

CART-TRAJ

Input Variables

last-pose
last-vel
traj-^-in
goal-pose-^-in

Output Variables

joint-com-pose
joint-com-conf-flag
joint-command
inc-command-#-out
current-pose
current-conf-flag
current-vel

Executes the specified Cartesian straight-line trajectory. CART-TRAJ calls STRAIGHT-LINE to build the trajectory and monitors the joint status. In response to any error conditions, CART-TRAJ sets joint-command to PAUSE.

JOINT-TRAJ

Input Variables

traj-^-in
goal-pose-^-in
status-in

Output Variables

joint-com-pose
joint-com-conf-flag
joint-command
inc-command-#-out
current-pose
current-conf-flag
current-vel

Sends a joint trajectory to the JOINT level and monitors joint status. In response to any error conditions, JOINT-TRAJ sets joint-command to PAUSE.

JOYSTICK

Input Variables	Output Variables
joy-1a	Rg
joy-1b	xg
joy-2a	vmax
joy-2b	amax
last-pose	wmax
	gmax

Adds the motion indicated by the joystick to last-pose and stores the result in Rg and xg for use by STRAIGHT-LINE. JOYSTICK sets vmax to the value resulting from the equation (joy-max-v) * 2**(-vel) and sets wmax to the value resulting from the equation (joy-max-w) * 2**(-vel), where vel represents the setting of the velocity switch on the joystick. JOYSTICK sets variables amax and gmax equal to vmax and wmax, respectively.

STRAIGHT-LINE

Input Variables	Output Variables
Rg	current-pose
xg	current-vel
vmax	
amax	
wmax	
gmax	
last-pose	
last-vel	

Executes the straight-line algorithm.

PRIM postprocessing

The PRIM routine POST-PROCESS sends to the PATH level confirmation for each command received and performs the "housekeeping" required to prepare PRIM for the next command. POST-PROCESS calls CLEAN-UP and, if the current command is not RESTART, calls COMMAND-RR.

CLEAN-UP

Input Variables	Output Variables
input-command-var	output-status-var

Echoes the command and inc-command-# to the PATH level.

COMMAND-RR

Input Variables	Output Variables
joint-com-pose	PRIM>JOINT-POSE round-robin
joint-com-conf-flag	PRIM>JOINT-TRAJ round-robin
joint-rec	

Adds the inverse tool movetable to the contents of the joint-com-pose variable and puts the resulting pose and the joint trajectory parameters in the PRIM>JOINT round-robins.

10.7 THE JOINT LEVEL

The JOINT level receives positions expressed in Cartesian coordinates from the PRIM level, transforms these positions to robot joint coordinates, and controls joint velocities and accelerations.

The design of the JOINT level is modular so that you can replace the existing robot-dependent modules (the interface routines and transforms) with new modules designed to control your specific robot.

The two basic modes of motion for JOINT are Cartesian-space motion and joint-space motion. In the Cartesian-space mode, the PRIM level interpolates Cartesian paths between the start and end positions by generating and executing a new point along a Cartesian path each cycle. The JOINT level transforms each path-point to joint coordinates and, if necessary, scales the move to meet velocity and acceleration limits.

In the joint-space mode, the PRIM level passes the goal position from the PATH level to the JOINT level. The JOINT level generates a joint-interpolated trajectory to the goal position.

The routines that the JOINT level executes fall into two categories: robot-independent and robot-dependent. The functions of the robot-independent routines do not change if you use a different robot. The functions of the robot-dependent routines are specific to the Puma 760 robot. You must provide your own routines to accommodate other robots.

JOINT Commands

The following commands execute at the JOINT level: RESTART, PAUSE, CARTESIAN, JOINT-TRAJ, and CREEP. This section provides the syntax and explains the function of each command.

RESTART command

The syntax of the RESTART command is:

```
RESTART
```

RESTART clears all errors, sets the current joint velocity and acceleration to Ø, and resets the debug variables. When the robot interface reports its status as done, RESTART reads the feedback joint values and then sends these values back to the JOINT level on succeeding cycles to maintain the current position. RESTART ignores acceleration limits.

While executing RESTART, the JOINT level reports its status as executing until the robot interface reports done. Then JOINT reports done to PRIM.

PAUSE command

The syntax of the PAUSE command is:

```
PAUSE
```

PAUSE continues sending the most recently commanded joint values to the robot interface. PAUSE ignores acceleration limits. After sending PAUSE, JOINT reports its status as done immediately.

CARTESIAN command

The syntax of the CARTESIAN command is:

```
CARTESIAN [pose-^]
```

where:

- pose-^ points to the position to which the robot is being moved.

CARTESIAN is best used to move through a series of points in a Cartesian motion, where the distances between successive points are small. JOINT transforms each pose it receives from PRIM into a set of goal joint values and moves the robot to the position indicated by pose-^. If necessary, JOINT scales the commanded trajectory to stay within the specified hardware limitations as it interpolates each step along the trajectory.

While executing CARTESIAN, JOINT reports its status as executing if the interpolated step size is larger than the system parameter next-point-scale-threshold. PRIM should not send any new points to JOINT when the JOINT status is executing.

If the step size is the same or smaller than this system parameter, JOINT reports its status as next-point. The next-point status indicates that PATH should send the next Cartesian step on the next cycle.

JOINT reports its status as done when the commanded joint values equal the goal joint values. The done status indicates that PRIM should send a new goal point on the next cycle.

JOINT-TRAJ command

The syntax of the JOINT-TRAJ command is:

```
JOINT-TRAJ [pose-^] [traj-para-^]
```

where:

- pose-^ points to the position to which the robot is being moved.
- traj-para-^ points to the trajectory parameters.

JOINT-TRAJ is best used to move through large joint-interpolated path segments. JOINT transforms each pose it receives from PATH into a set of goal joint values and moves the robot to the position indicated by pose-^. If necessary, JOINT scales each step to stay within the limitations specified by traj-para-^ as it interpolates the trajectory. While executing JOINT-TRAJ, JOINT reports its status as executing if the commanded joint values are outside the range of the goal joint values specified by the j-delta variable. JOINT reports done after the commanded values are within the j-delta range.

CREEP command

The syntax of the CREEP command is:

```
CREEP
```

CREEP recovers from the joint-limit error state by commanding all joints that are within the creep-delta distance from a joint limit to move away slowly from the limit at a speed specified by creep-vel. The creep-delta distance and the creep-vel velocity are user-defined values.

While executing CREEP, JOINT reports its status as executing until all joint movements are complete. Then CREEP reports done.

JOINT Input Command Buffer

The JOINT input command buffer consists of the following variables:

inc-command-#-in
PRIM increments this integer variable each time it sends a new command to JOINT.

input-command
PRIM sends this integer to JOINT to identify the command. Commands are numbered as follows: Ø-RESTART, 1-PAUSE, 2-CARTESIAN, 3-JOINT-TRAJ, and 4-CREEP.

pose-^-in
PRIM sends JOINT this pointer to the commanded pose in POSE-FILE.

traj-^-in
PRIM sends JOINT this pointer to the traj-para in the PRIM>JOINT-TRAJ round-robin.

JOINT Status Information

To provide status feedback to the PRIM level, the JOINT level returns a status report, returns a feedback pose, and returns echoes of the input command and its incremental command number.

JOINT uses the following six variables to return status information to the PRIM level:

cycle-#-status-out
Returns the cycle count when JOINT returns its status. The changing value of this integer variable verifies that the JOINT level is running.

inc-command-#-echo-out
Echoes the integer count of new commands received by the JOINT level as indicated by the inc-command-#-in variable. This information enables PRIM to know for which command JOINT is returning status information.

command-echo-out
Echoes the integer stored in inc-command-#-in. This information enables PRIM to know for which command JOINT is returning status information.

status-report
Returns the current status of JOINT. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	executing	The current point is still executing.
1	done	The joint has reached its goal location.
2	error	An error occurred. (See status-arg-out.)
3	next-point	The joint reached an intermediate path-point.

status-arg-out

Returns the error status. The possible values for this integer variable and their meanings are:

Value	Meaning	Description
Ø	noerror	PRIM detected no errors.
1	lower-level	The servo reported an error to JOINT.
2	command-error	JOINT received an invalid command or parameters.
3	joint-limit	A joint has reached its limit.
4	reach-limit	The goal pose is outside the reach of the robot.
5	configuration	The commanded configuration is invalid.

pose-^-out

Contains the pointer to the feedback pose, which is the output joint values transformed to the wrist pose. During the execution of a RESTART command, the feedback pose gives the startup position of the robot. During the execution of the CARTESIAN and JOINT-TRAJ commands, the feedback pose can update the Cartesian trajectory routines with the actual commanded position.

JOINT Errors

When JOINT detects an error of any type, it sets the status variable to indicate an error, stores the label for the error in the status-arg-out variable, and sends the last valid joint values to the robot. At this point, COMMAND-PROCESS does not execute any command other than RESTART, with one exception. If the error is joint-limit, COMMAND-PROCESS allows the CREEP command to correct the error.

JOINT Processing

The three processes for the JOINT level are PRE-PROCESS, COMMAND-PROCESS, and POST-PROCESS. PRE-PROCESS reads the command from the PATH level and the status from the servo. COMMAND-PROCESS, the decision-processing stage, calculates trajectories for robot motion and scales the motion to meet velocity and acceleration limits. POST-PROCESS sends the JOINT level confirmation for each command received and performs the "housekeeping" required to prepare JOINT for the next cycle.

JOINT preprocessing

The JOINT routine PRE-PROCESS calls NEW-COMMAND? and SERVO-STATUS.

NEW-COMMAND?

Input Variables	Output Variables
inc-command-#-in	new-command
old-inc-command-#-in	old-inc-command-#-in
	status
	status-arg

Sets the new-command flag to true if the value of inc-command-#-in has changed since the last cycle (That is, if inc-command-#-in does not equal old-inc-command-#-in.) NEW-COMMAND? also sets the status and status-arg variables to noerror.

SERVO-STATUS

Input Variables	Output Variables
inc-command-#-out	servo-status
inc-command-#-echo-in	
status-in	

Checks if the values of inc-command-#-out and inc-command-#-echo-in are equal (indicating that status-in contains the status of the last command). If these values are equal, SERVO-STATUS sets the value of servo-status to the value of status-in. If these values are not equal, SERVO-STATUS sets the value of servo-status to executing.

JOINT decision processing

If the status-report variable does not indicate an error, the JOINT routine COMMAND-PROCESS uses a case statement to determine which of the routines in this section to execute. If the status-report variable indicates an error other than joint-limit, COMMAND-PROCESS freezes the error state and does not execute any command other than RESTART. COMMAND-PROCESS allows CREEP to execute if the error is joint-limit.

JOINT-RESTART

Input Variables	Output Variables
joints-in	output-command
new-command	inc-command-#-out
servo-status	servo-com-joint
	status-report
	v-current
	overflow cycle
	max-process-time
	min-process-time

Checks if new-command is true and, if so, sets the value of the output-command variable to RESTART, increments inc-command-#-out, sets v-current to Ø, and resets the debug variables. If new-command is false, JOINT-RESTART waits for servo-status to indicate done and then calls READ-SERVO-STATUS to set servo-com-joint to the feedback joint values.

JOINT-PAUSE

Input Variables	Output Variables
none	inc-command-#-out
	v-current

Pauses the robot, leaves the current joint values unchanged in the servo-com-joint variable, sets v-current to Ø, and increments inc-command-#-out.

CARTESIAN

Input Variables	Output Variables
pose-^-in	servo-com-joint
v-current	v-current
old-ja	

Retrieves the commanded pose and configuration flags, calls CART>JOINT and CART-CONFIGURE to convert the pose to joint values, calls SCALE to limit the joint velocities and accelerations (using the hardware maximum limits), stores the result in servo-com-joint, and calls JOINT-LIMIT-TEST. If JOINT-LIMIT-TEST does not result in an error, CARTESIAN increments inc-command-#-out. If JOINT-LIMIT-TEST results in an error, CARTESIAN sets the value of servo-com-joint to that of old-ja.

JOINT-TRAJ

Input Variables	Output Variables
pose-^-in	servo-com-joint
traj-^-in	v-current
v-current	
old-ja	

Retrieves the commanded pose, configuration flags, and trajectory parameters; calls CART>JOINT and JOINT-CONFIGURE to convert the pose to joint values; calls SCALE to limit joint velocities and accelerations (using the commanded trajectory parameters), stores the result in servo-com-joint, and calls JOINT-LIMIT-TEST. If JOINT-LIMIT-TEST does not result in an error, JOINT-TRAJ increments inc-command-#-out. If JOINT-LIMIT-TEST results in an error, JOINT-TRAJ sets the value of servo-com-joint to that of old-ja.

SCALE

Input Variables	Output Variables
xg	servo-com-joint
scaling-var	scaling-var

Scales the step from servo-com-joint to xg to meet the velocity and acceleration limits, and ensures that the joint motions are coordinated to reach their goals simultaneously.

JOINT postprocessing

The JOINT routine POST-PROCESS sends to the PRIM level confirmation for each command received, and performs the "housekeeping" required to prepare JOINT for the next command. POST-PROCESS calls CLEAN-UP and, if the input-command-var is not RESTART (or if the command is RESTART and the status-report variable indicates done), calls FEEDBACK-POSE and SET-SERVO-COMMAND.

CLEAN-UP

Input Variables	Output Variables
input-command-var	output-status-var
v-current	v-previous
servo-com-joint	old-ja

Echoes the command and inc-command-# to the PRIM level, sets the value of v-previous to that of v-current, and sets the value of old-ja to that of servo-com-joint.

FEEDBACK-POSE

Input Variable	Output Variable
servo-com-joint	RBT-POSE round-robin

Calls JOINT>CART to transform the value of servo-com-joint to a pose, and stores that pose in the RBT-POSE round-robin.

JOINT Robot-Dependent Routines

The following commands control JOINT operations that are specific to the type of robot the system is controlling. You must edit or replace these routines to accommodate different robots.

READ-SERVO-STATUS

Input Variable	Output Variable
joint-in	servo-com-joint

READ-SERVO-STATUS converts the value of the joint-in variable from the servo format to radians and returns the converted value in servo-com-joint.

SET-SERVO-COMMAND

Input Variable	Output Variable
servo-com-joint	joint-out

SET-SERVO-COMMAND scales the value of the servo-com-joint value from radians to the servo format and returns the converted value in joint-out.

REACH-CHECK

Input Variable	Output Variable
pose	status
	status-arg

REACH-CHECK checks the pose variable to see if the commanded pose is within the work volume of the robot. If the pose is not within the work volume of the robot, REACH-CHECK sets the status variable to error and the status-arg variable to reach-limit.

JOINT-LIMIT-TEST

Input Variable	Output Variable
servo-com-joint	status
	status-arg

JOINT-LIMIT-TEST tests the servo-com-joint variable to see if it exceeds the joint limits. If the servo-com-joint value exceeds the joint limits, JOINT-LIMIT-TEST sets the status variable to error and the status-arg variable to joint-limit.

JOINT>CART

Input Variable	Output Variable
servo-com-joint	pose
	conf-flag

JOINT>CART transforms the value of servo-com-joint to a pose and stores the result in the pose and conf-flag variables.

CART>JOINT

Input Variable	Output Variable
pose	xg (goal joint values)
conf-flags (POSE-FILE local template)	status
	status-arg

CART>JOINT converts the value of the pose and configuration flags to joint values and stores the result in xg, the goal joint values variable. The conf-flag variable contains the configuration flags it reads from the POSE-FILE local template.

In addition to the pose and conf-flag variables, CART>JOINT uses the old-ja variable to determine the new joint values. The variable old-ja stands for old-joint-angle. It holds the commanded joint values from the previous cycle. It is set to servo-com-joint by POST-PROCESS. This process is similar to the way PRIM handles current versus old position and velocity. See the transform documentation for your robot for details on poses and configurations.

Warning: CART>JOINT may generate a floating-point error interrupt that aborts the control process if the pose is outside the work area of the robot. Use REACH-CHECK to test the pose before using CART>JOINT. (See also REACH-CHECK.)

CART-CONFIGURE

Input Variable	Output Variable
xg	xg
old-ja	

CART-CONFIGURE reads xg and old-ja and then modifies xg to minimize joint motion by taking advantage of multiple configurations for the same pose. See the transform documentation for your robot for definitions of configurations.

JOINT-CONFIGURE

Input Variable	Output Variable
xg	xg
conf-flag	

JOINT-CONFIGURE reads xg and conf-flag and modifies the transform output to match the commanded configuration. See the transform documentation for your robot for definitions of configurations.

This chapter discusses how to incorporate user extensions to RSL. It assumes that you have loaded the NBS-supplied system tape as described in Section 5.5, "Software Installation Procedure", and that you have not yet modified the code.

The extensions described in this chapter are extensions made while developing RSL. The code is included as part of RSL on your system tape. To understand the general procedures for making similar extensions to RSL, refer to the blocks mentioned in this chapter.

11.1 UNDERSTANDING THE TYPES OF EXTENSIONS

RSL was developed in the context of the transfer task application and may be inappropriate for other applications, such as tracing an object for a deburring operation. To overcome these potential limitations, you can extend the RSL application by adding new tasks, paths, path-points, and trajectories to the existing RSL control structure.

For example, you need a new trajectory for any type of motion other than Cartesian straight-line or joint interpolated, such as circular motion. You need new path-points whenever you add another sensor to the system. When you develop a new application, both new tasks and new paths are required to provide a hierarchical decomposition of that application.

After you understand the basic procedures, you may want to make more complex extensions to RSL. For example, you can modify RSL to perform a quick change operation that changes the tool movetable while a task is executing, or you can add the ability to download RSL programs from a supervising computer. You can also add new control levels, execute control levels in parallel, or run two robots simultaneously.

11.2 ADDING A TASK TO RSL

You can extend RSL by adding a new task. Adding a task is a complex operation, that usually involves adding additional paths and path-points. This section describes how to add the MOVE-TO task as if it were a new extension to RSL, assuming the move-to path and goto path-point are present.

The syntax for the MOVE-TO task is:

```
     MOVE-TO  [object] [grip#] [destination location type] [destination]
          [sector]
```

The MOVE-TO task issues a move-to path, using the current location as the starting location. If the destination location type is an array (arr), then the destination is the array sector given by the optional parameter sector. MOVE-TO updates the current location when it completes the path. Refer to the indicated blocks on the system tape for the actual MOVE-TO code.

To add the MOVE-TO task to RSL, you must make four additions. Table 11-1 lists the additions and their associated block numbers. These routines are all on the TASK level on the TASK&PATH board. The block numbers listed assume that you are on the TASK&PATH board with the OFFSET set at 9ØØØ. Remember that you have to update the directory/load blocks when you add new routines to RSL.

Table 11-1. Additions for the MOVE-TO Task.

Routine	Addition
MOVE-TO	Add this routine to execute the task, send a move-to path to the PATH level, and wait for the PATH level to report status done. Refer to block 513 for the routine MOVE-TO.
MOVE-TO-PARAM	Add this routine to read the input parameters, looking up record numbers in the object and location files. Refer to blocks 511 and 512 for the routine MOVE-TO-PARAM.
MOVE-TO-SEARCH	Add this routine to search for the proper MOVE-TO-FILE record. Refer to block 4Ø2 for the routine MOVE-TO-SEARCH.
COMMAND-PROCESS	Add MOVE-TO to the if-then-else list of legal commands that the RSL routine COMMAND-PROCESS uses to execute tasks on the TASK level. COMMAND-PROCESS is in block 591.

11.3 ADDING A PATH TO RSL

You can extend RSL by adding a new path. This section describes how to add the move-to path type as if it were a new addition to RSL. Note that lower-case move-to is a path type and not the same as the MOVE-TO task described in Section 11.2, "Adding a Task to RSL".

The syntax for the move-to path type is:

```
     -path-  [move-to] [object] [grip#] [start location type] [start location]
          [destination location type] [destination location]
```

The move-to path type moves the robot quickly from near the start location to near the destination location. The approach and depart path types move the robot when it is near the start location or the destination location. Refer to the indicated blocks on the system tape for the actual move-to code.

Remember that you have to update the directory/load blocks when you add new routines to RSL.

The following steps use the move-to path as an example of the general procedure you follow to add a similar path to RSL. The block numbers listed assume that you are on the RSL board with the OFFSET set at 8ØØØ.

1. Define an RSL data structure for the new path type by creating a SMACRO variable owner that lists the parameters for the new path type. The variable owner defines the record format for the move-to data structure file, described in Section 9.2, "RSL Overview".

   Insert the new file declaration in an empty block between blocks 141 and 149, the blocks reserved for RSL path-type file declarations. List block 141 to see the move-to path parameter file MOVE-TO-FILE.

2. Add the new path type to the list of legal path types. Block 195 contains the sequential variable owner path-type-list that defines the legal RSL path types. List block 195 to see the addition of the move-to sequential variable to the path-type-list owner.

3. Add a routine that compiles the path parameters and starts the path-point (ppt) list. The routine reads the new command and all its parameters, looks up the record numbers for the locations and object, and creates a path-point list in the RSL PATH-POINT-FILE. The routine should store the results of reading the command in the file created in Step 1. List blocks 411 and 412 to see the routine MOVE-TO.

   List blocks 413 and 414 to see the optional routine ?MOVE-TO that displays information about all currently defined move-to paths. You can create a similar routine for other path types.

4. Add the new path type to the case statement in the RSL routine -path- that starts compiling an RSL path. List block 491 to see the move-to path type in the case statement.

5. Initialize the new path file created in Step 1, by using the SMACRO file operator ~re-init-file. List block 993 to see the initialization of the MOVE-TO-FILE.

11.4 ADDING A PATH-POINT TO RSL

You can extend RSL by adding a new path-point. This section describes how to add a range path-point that positions the tool point at a given distance from a surface, using a sonar range finder. Although the range path-point is included with RSL, you cannot use this path-point without a sonar and additional hardware circuitry to enable RCS to read the range.

The syntax for the range path-point is:

```
     -ppt- range  [sonar#] [range] [threshold] [{X, Y, Z}] [loc-phrase]
          [traj-phrase]
```

The range path-point translates the tool point along the axis of the loc-phrase frame, until the range read by the specified sonar is within the threshold of the range. Refer to the indicated blocks on the system tape for the actual range code.

Remember that you have to update the directory/load blocks when you add new routines to RSL. To add the range path-point, you have to add code on the RSL board and the TASK&PATH board.

Adding Code on the RSL Board

The following steps describe the additions you need to make on the RSL board to add the range path-point or a similar path-point to RSL. The block numbers listed assume that you are on the RSL board with the OFFSET set at 8ØØØ.

1. Define an RSL data structure for the new path-point by creating a SMACRO variable owner that lists the parameters for the new path-point. The variable owner defines the record format for a new data structure file, similar to the data structures defined in Section 9.2, "RSL Overview".

   Insert the new file declaration in an empty block between blocks 161 and 169, the blocks reserved for RSL path-point command parameters file declarations. List block 162 to see the range path-point parameter file RANGE-FILE.

2. Add the new path-point to the list of legal path-points. Block 192 contains the sequential variable owner ppt-command-list that defines the legal RSL path-points. List block 192 to see the addition of the range path-point to the ppt-command-list owner.

3. Add a routine to compile the new ppt, read ppt syntax fields, look up the record numbers for the location, and add records to the trajectory, range, and path-point files. The routines must read and check all of the ppt syntax fields before storing any records. You must leave the files in a clean state in case any errors occur in the ppt syntax fields. List block 343 to see the routine RANGE.

4. Add a garbage collection routine to remove records added by the routine in Step 3. The routine should remove the records when a path containing a range ppt is redefined. List block 344 to see the routine RANGE-REMOVE.

5. Add the new path-point to the case statement in the RSL routine -ppt- that defines path-points. List block 391 to see the addition of the range path-point to the case statement.

6. Add the garbage collection routine in Step 4 to the case statement in the RSL routine PPT-REMOVE. List block 395 to see the addition of the routine RANGE-REMOVE to the case statement.

7. Initialize the new path-point file created in Step 1, by using the SMACRO file operator ~re-init-file. List block 993 to see the initialization of the RANGE-FILE.

Adding Code on the TASK&PATH Board

The following steps describe the additions you need to make on the TASK&PATH board to add the range path-point or a similar path-point to RSL. All the additions on the TASK&PATH board are to the PATH level. The block numbers listed assume that you are on the TASK&PATH board in the PATH vocabulary with the OFFSET set at 1ØØØØ.

1. Add the variables needed to execute routines for the new path-point. The variables you need to add depend heavily on the type of path-point you are adding to RSL. List block 141 to see the variables needed for the range path-point. List block 142 to see the additional variables that give addresses of the ports for the sonar interface, and some constants used in the interface protocol.

2. Add any routines needed to process information from a sensor. These routines depend heavily on the path-point and sensor you are adding to RSL. List block 251 to see the addition of the range path-point routine SONAR-PORT-INIT that initializes the sonar interface port. List block 252 to see the addition of the routine SONAR-READ that reads the sonar range, converts the range to inches, and stores the result in sonar-range.

3. Add any read routines specified in Step 2 to the RSL routine PRE-PROCESS. List block 299 for the addition of SONAR-READ to the PRE-PROCESS routine for the PATH level. This addition enables you to read the sonar every cycle, making data available as soon as needed. (The SONAR-READ routine is currently commented out.)

4. Add the command routines needed to execute the new path-point in empty blocks between blocks 3ØØ and 399. For the range path-point, list block 331 to see the routine TRANSLATE, which translates the current tool pose by a specified delta along the specified axis. TRANSLATE stores the result in prim-com-pose.

   List block 332 to see the routine HALT, which sets prim-com-pose to the current tool pose and sets the neighborhood fields of the current trajectory template to Ø. HALT commands the robot to halt at the current tool pose.

   List block 411 to see the routine R-NEW-PPT, which retrieves the range path-point parameters and initializes the range path-point variables.

   List blocks 412 and 413 to see the routine RANGE, which executes the range path-point.

5. Initialize the new path-point routines and variables. List blocks 982 and 983 to see the initialization of these variables and routines for the range path-point.

11.5 ADDING A TRAJECTORY TYPE TO RSL

You can extend RSL by adding a new trajectory type. This section describes how to add a servo trajectory type to RSL. The code that implements this trajectory type is included on the system tape and is loaded when you load RSL.

The syntax for the servo trajectory phrase is:

```
     servo  [sensor-mtb] [inv-sensor-mtb] [goal-mtb] [trn-max-acc]
          [trn-max-vel] [trn-min-vel] [max-range] [trn-nbrhd] [rot-max-acc]
          [rot-max-vel] [rot-min-vel] [max-angle] [rot-nbrhd]
```

The servo trajectory type moves the sensor toward the goal (target pose plus goal-mtb). The servo trajectory reports done when the robot is within the neighborhood you specify in trn-nbrhd and rot-nbrhd. Servo decreases the translation velocity linearly from trn-max-vel (for ranges greater than or equal to the max-range) to trn-min-vel at the goal (range=Ø). Servo decreases the rotation velocity linearly from rot-max-vel (for angles greater than or equal to max-angle) to rot-min-vel at the goal (angle=Ø). Refer to the indicated blocks on the system tape for the actual servo code.

Remember that you have to update the directory/load blocks when you add new routines to RSL.

To add the servo trajectory type, you have to add code on the RSL board, the TASK&PATH board, and the PRIM board.

Adding Code on the RSL Board

The following steps describe the additions you need to make on the RSL board to add the servo trajectory type or a similar trajectory to RSL. The block numbers listed assume that you are on the RSL board with the OFFSET set at 8ØØØ.

1. Define an RSL data structure for the new trajectory type by creating a SMACRO variable owner that lists the parameters for the new trajectory type. The variable owner defines the record format for a new data structure file, similar to the data structures defined in Section 9.2, "RSL Overview".

   Insert the new routine in an empty block between blocks 121 and 129, the blocks reserved for RSL trajectory file declarations. List block 123 to see the servo trajectory-parameter file SERVO-FILE.

2. Add the new trajectory type to the list of legal trajectory types. Block 194 contains the sequential variable owner traj-list that defines the legal RSL trajectory types. List block 194 to see the addition of the servo sequential variable to the traj-list owner.

3. Add a routine to read the new command and all its parameters. The routine should store the results of reading the command in the local template of the file created in Step 1. List block 322 to see the routine READ-SERVO-TRAJ.

4. Add the routine created in Step 3 to the case statement in the RSL routine READ-TRAJ-PHRASE. List block 337 to see the addition of READ-SERVO-TRAJ to the case statement.

5. Add the routine created in Step 1 to the case statement in the RSL routine STORE-TRAJ-PHRASE. List block 338 to see the addition of SERVO-FILE to the case statement.

6. Add the routine created in Step 1 to the case statement in the RSL routine REMOVE-TRAJ-PHRASE. List block 339 to see the addition of SERVO-FILE to the case statement.

7. Add a round-robin to pass the new trajectory type parameters from the PATH level to the PRIM level. Add the round-robin in an empty block between blocks 511 and 519, the blocks reserved for RSL round-robins. The round-robin enables the PATH level to change parameters every cycle, if necessary. List block 511 to see the servo round-robin PATH>PRIM-SRV.

8. Initialize the new trajectory file created in Step 1, by using the SMACRO file operator ~re-init-file. List block 993 to see the initialization of the SERVO-FILE.

Adding Code on the TASK&PATH Board

The following steps describe the additions you need to make on the TASK&PATH board to add the servo trajectory type or a similar trajectory type to RSL. All the additions on the TASK&PATH board are to the PATH level. The block numbers listed assume that you are on the TASK&PATH board in the PATH vocabulary with the OFFSET set at 10000.

1. Add an integer variable to the SMACRO variable owner reserved-record# to access the round-robin created in Step 7 of the previous section. List block 173 to see the addition of the servo variable path-srv-^ to the reserved-record# owner.

2. Add the new trajectory type file to the case statement in the RSL routine TRAJ-PHRASE. This addition enables you to retrieve the appropriate record from the data structure file, and also makes the new trajectory type legal for the PATH level. List block 322 to see the addition of SERVO-FILE to the case statement.

3. Add the new trajectory type to the case statement in the RSL routine HALT block 332.

4. Add the new trajectory type and file to the case statement in the RSL routine TRAJ-ROUND. TRAJ-ROUND sends the trajectory parameters to the trajectory round-robin. List block 804 to see the addition of the servo trajectory type and SERVO-FILE for the PATH>PRIM-SRV round-robin.

5. Initialize the new trajectory variable created in Step 1. The variable identifies the round-robin record for the RSL routine TRAJ-ROUND. List block 981 to see the initialization of path-srv-^.

Adding Code on the PRIM Board

The following steps describe the additions you need to make on the PRIM board to add the servo trajectory type or a similar trajectory type to RSL. The block numbers listed assume that you are on the PRIM board with the OFFSET set at 11000.

1. Add the variables and routines for the new trajectory type. The variables you need to add depend heavily on the type of trajectory you are adding to RSL. Refer to blocks 730 through 739 for the servo trajectory variables and routines.

   The Cartesian straight-line trajectory type is the basis for the servo trajectory. The routines transform the velocity of the tool point to the sensor frame, command the goal pose as the goal of both the rotation and translation parts of the trajectory, compute the maximum velocity as a function of the range to the goal, and transform the result back into a motion of the tool point.

2. Add the new trajectory type to the case statement in the RSL routine TRAJ. This addition makes the new trajectory type legal for the PRIM level. List block 791 to see the addition of the servo trajectory type to the case statement.

This chapter describes two example applications of RCS. NBS developed each application for a different project. The first application describes a machining workstation that is part of a totally automated factory project. The second application describes a field materiel-handling robot that is part of an Army-sponsored research project.

12.1 A MACHINING STATION IN THE AUTOMATED MANUFACTURING RESEARCH FACILITY

The National Bureau of Standards is developing an experimental factory called the Automated Manufacturing Research Facility (AMRF). The AMRF will operate as a small, totally automated, batch machine shop.

Currently the AMRF consists of five workstations: three machining stations, a cleaning and deburring station, and an inspection station. The AMRF maintains a central database that provides information for the five workstations. Each workstation uses robotic material handling for tasks such as machine-tool loading and unloading, deburring parts, and loading a collet into a lathe. The horizontal machining station and the cleaning and deburring station use RCS to maintain real-time control over the robot.

The following sections discuss the horizontal machining workstation in more detail and give an example of unloading an incoming parts tray.

For more information, read the IEEE Computer Society reprint entitled, "A Hierarchically Controlled, Sensory Interactive Robot in the Automated Manufacturing Research Facility". This paper is a reprint from the IEEE International Conference on Robotics and Automation, St. Louis, Missouri, March 25 to 28, 1985.

The Horizontal Machining Workstation

This example discusses the horizontal machining workstation. This workstation contains a horizontal-machining center and a materiel-handling robot, the Cincinnati Milacron T3 hydraulic, six-degree-of-freedom manipulator. The function of the T3 robot is to load part blanks into the horizontal-machining center fixtures and then unload the machined parts. The T3 robot also loads and unloads the parts from the part trays and buffers of the workstation.

To meet the requirements of the AMRF, NBS equipped the T3 with a hierarchical real-time control system (RCS), a 3-D vision system, a watchdog safety system, an active pedestal, a quick-change device, and an instrumented servo-controlled gripper.

Figure 12-1 shows the major robot-related components of this workstation and the workstation control system, which sends commands to the T3 RCS.

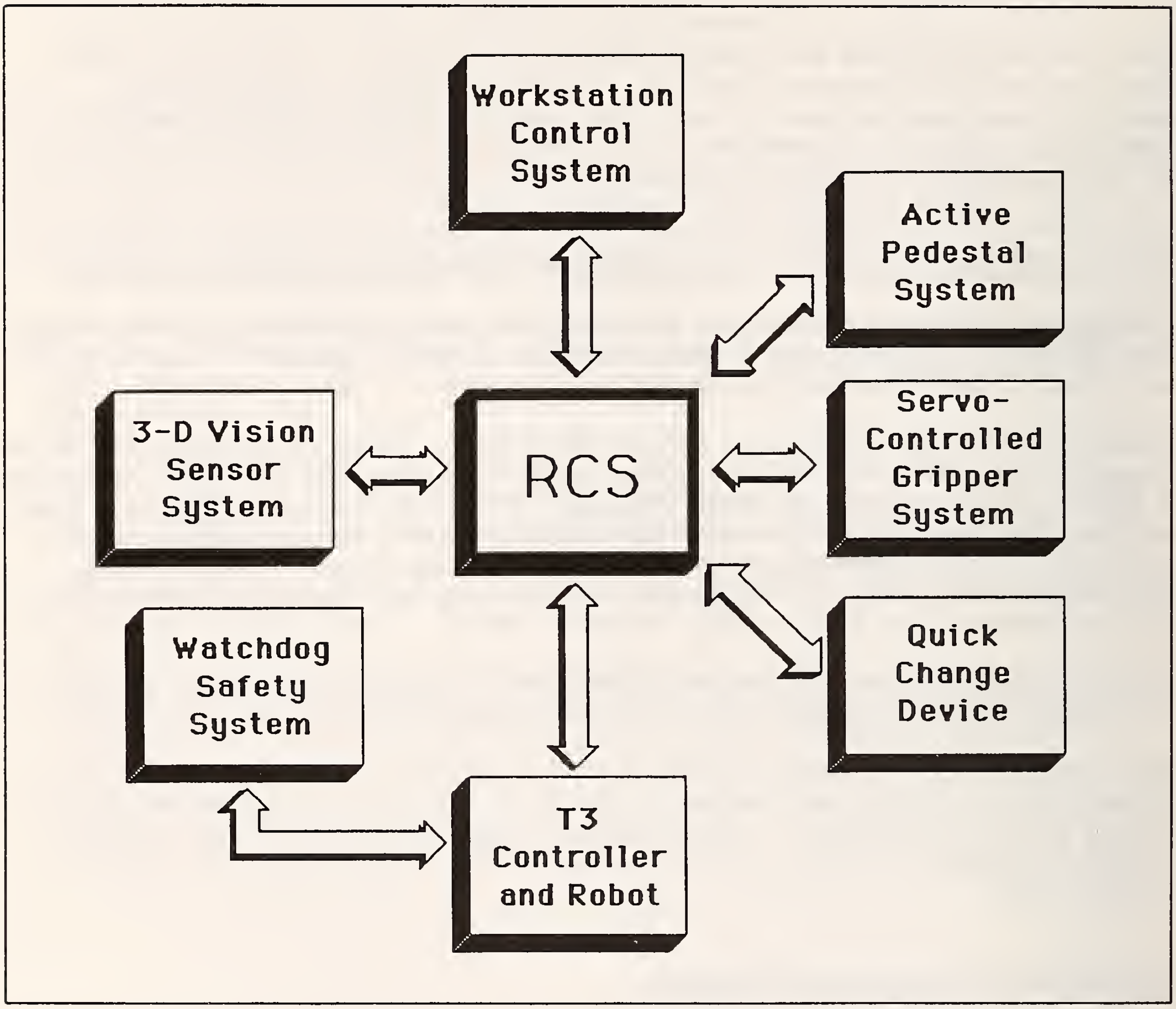


Figure 12-1. Major robot-related components of the horizontal machining workstation.

Table 12-1 lists the functions of the major robot-related components of the horizontal machining workstation.

Table 12-1. Functions of the Major Robot-Related Components of the Horizontal Machining Workstation.

Component	Function
RCS	Controls the operation of the T3 robot, the active pedestal system, and the quick change device.
3-D Vision Sensor System	Locates and identifies objects.
Watchdog Safety System	Monitors individual joint and tool-point motions of the robot to prevent the robot from damaging itself or other equipment.
Active Pedestal System	Regrips parts.
Quick Change	Changes end effectors.
Servo-Controlled Gripper System	Enables the T3 to open a gripper to a specified width or close a gripper with a specified force.

The hardware and software configuration of the T3 RCS is more complex than the RSL configuration described in earlier chapters, because the T3 RCS supports additional devices.

Hardware configuration of the T3 RCS

The hardware configuration of RCS for the horizontal machining workstation consists of six Intel 86/30 processor boards communicating via a MULTIBUS. Three of the 86/30 boards contain the task decomposition levels: TASK, SUBTASK, E-MOVE (elemental move), PRIM (primitive), PARTS GRIPPER, VALVE GRIPPER, and QUICK CHANGE.

The final robot-dependent level, T3, is on a separate 86/30 board. Of the other two remaining 86/30 boards, one contains a diagnostics system and the communications process (COMM), and the other controls the graphics display monitor.

The 86/3Ø boards have parallel I/O and A/D ports for communicating with sensors and actuators. For example, the board containing the E-MOVE and PRIM control levels has access to proximity sensors and gripper beam-break sensors through the parallel I/O ports of the board.

This implementation of RCS also includes six other boards: two memory boards for the common memory, which contains user files, communications buffers, and the system dictionary; a DMA (Direct Memory Access) board that provides communications with the vision system; two 68ØØØ processor boards that provide access to other workstations on the network; and the board for controlling the hard disk and tape drive.

Figure 12-2 shows the T3 RCS hardware configuration.

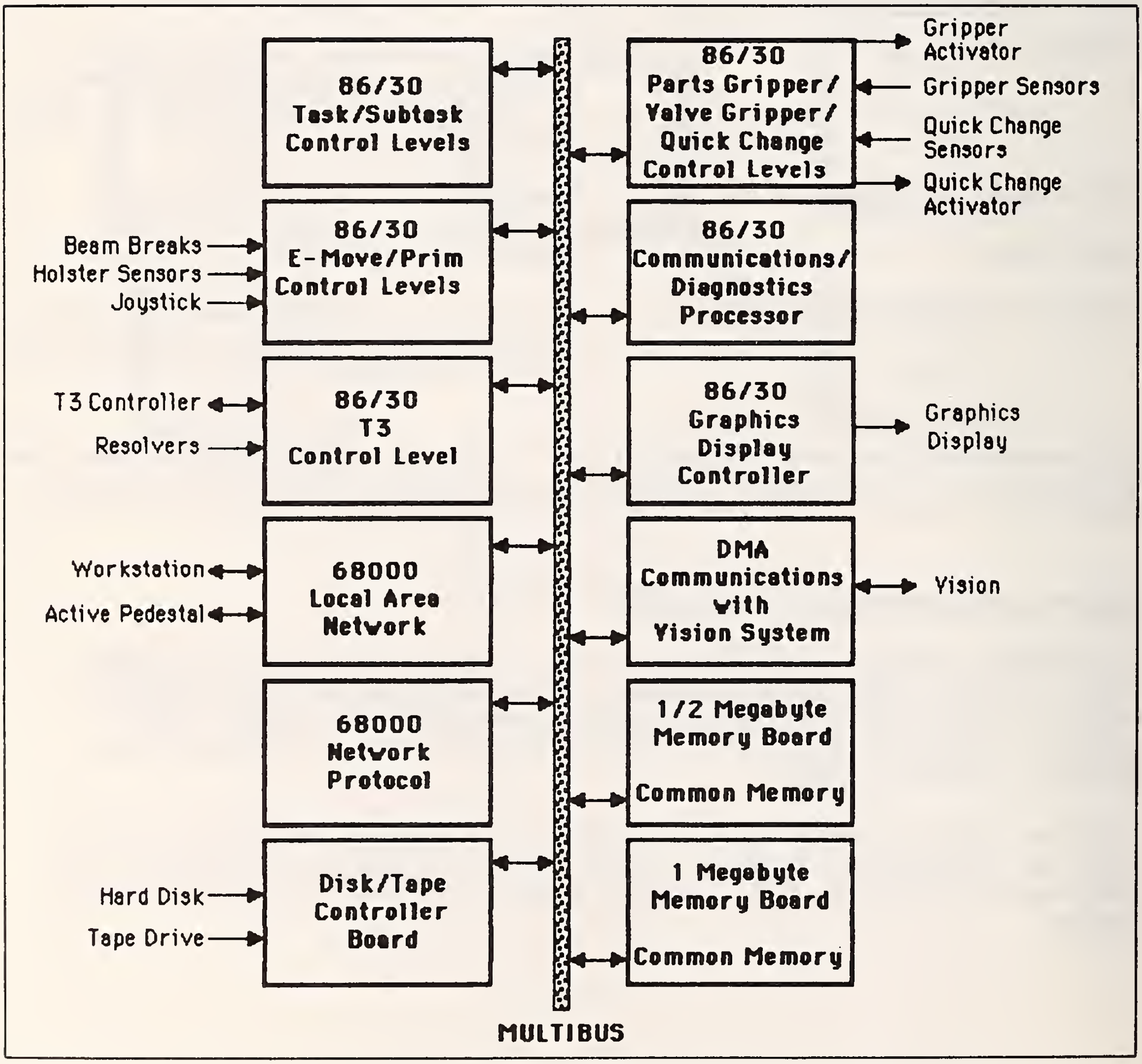


Figure 12-2. T3 RCS hardware configuration.

Software configuration of the T3 RCS

The RCS software configuration for the horizontal machining workstation consists of task decomposition control levels similar to the RSL application control levels. This application of RCS decomposes tasks using the control levels: TASK, SUBTASK, E-MOVE, PRIM, and T3. The T3 level is robot dependent, transmitting the appropriate commands to the T3 controller.

In addition, the control levels PARTS GRIPPER, VALVE GRIPPER, and QUICK CHANGE execute in parallel with the PRIM level, enabling the gripper system to position the gripper independently of the T3 robot motion. The control levels PARTS GRIPPER, VALVE GRIPPER, and QUICK CHANGE transmit commands to the two grippers and to the quick change actuator, respectively. Figure 12-3 shows the control-level hierarchy.

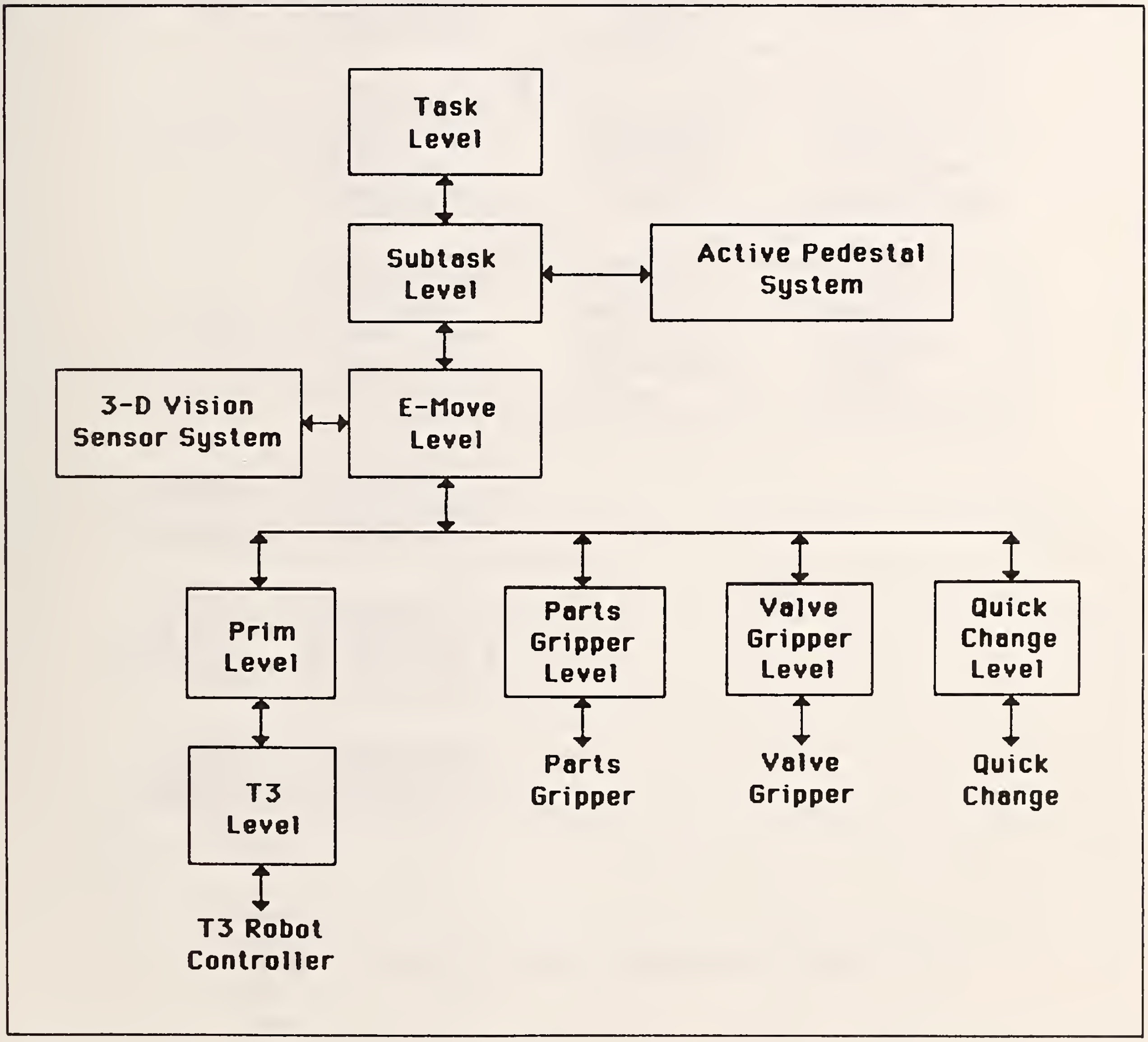


Figure 12-3. T3 RCS control-level hierarchy.

The COMMUNICATIONS/DIAGNOSTICS processor manages communications between levels. This processor transfers the command and status buffers between levels every 4Ø milliseconds, as set by the fixed cycle time of the T3 RCS.

Each control level processes a set of commands of decreasing complexity. A typical command at the TASK level decomposes into several robot movements, while a command at the PRIM level typically becomes one simple robot movement.

For example, a TRANSFER command consists of the following sequence of steps: moving to the specified object at the specified location, identifying the object orientation using the vision system, grasping the object, moving the object to the destination point, and releasing the object. The other control levels decompose these steps into simpler and simpler actions. For example, the command DELTA-MOVE at the PRIM level moves the robot a short distance from its current position.

Figure 12-4 shows the commands that each control level can process.

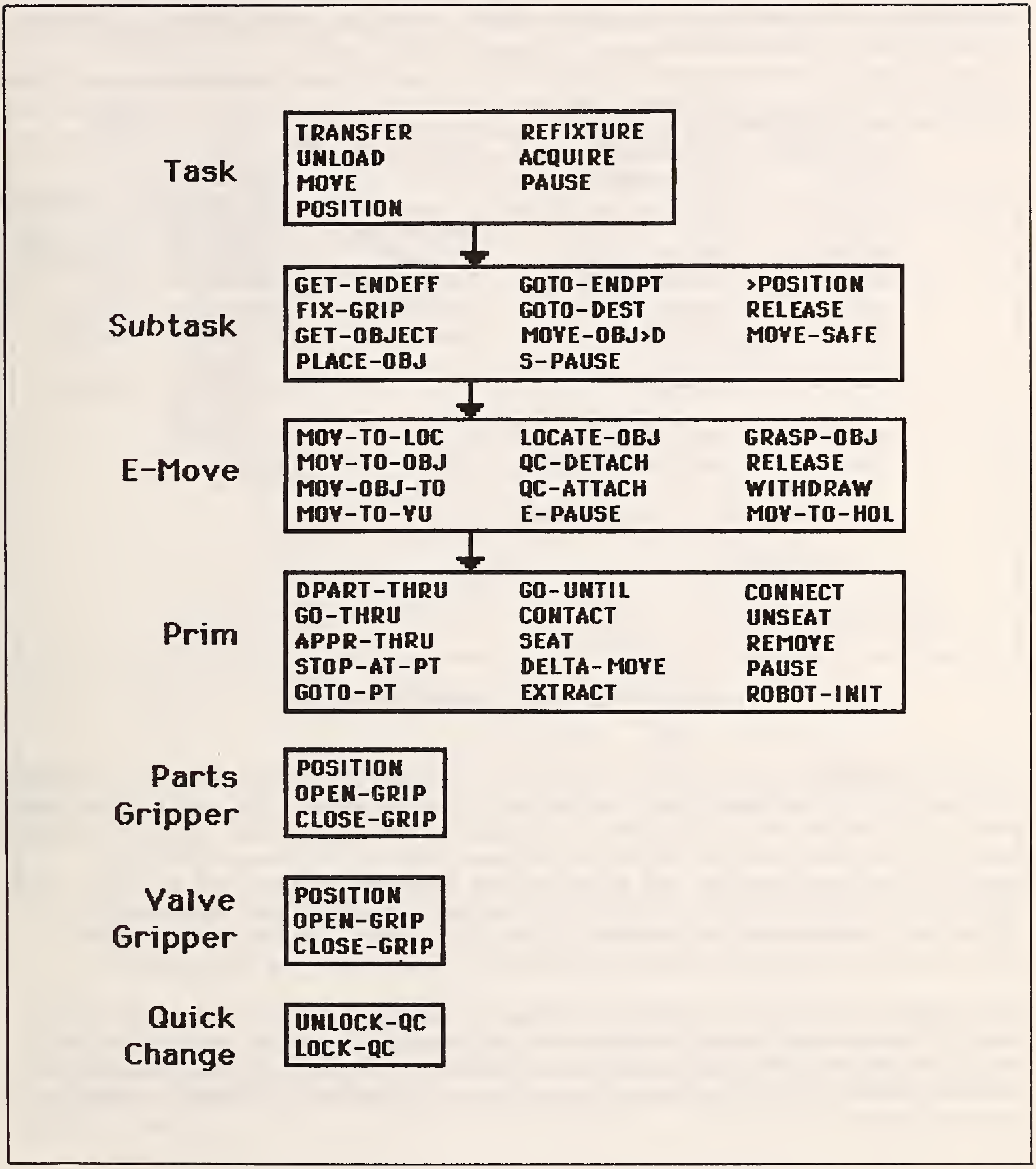


Figure 12-4. Control level commands for the T3 RCS.

Whenever possible, this RCS application uses the SMACRO state table statement to decompose tasks. Figure 12-5 shows an example state table routine, QC-DETACH, which decomposes a quick change operation, disengaging the current end effector on the robot.

```
routine QC-DETACH-ST                                        EDEF
   state-table   ref-action         "qc-status-in"
     state:      new-command        XX            EXECUTING
                                                  GET-HOLSTER-D CONTACT
     state:      XX                 "executing"   EXECUTING
     state:      contact-made       "locked"      EXECUTING  QC-UNLOCK
     state:      contact-made       "unlocked"    EXECUTING  SEAT
     state:      seated             "unlocked     EXECUTING  EXTRACT
     state:      extracted          "unlocked"    EXECUTING  VERIFY-
                                                    DETACH
     state:      detached           XX            EXECUTING  DEPART-
                                                    HOLSTER
     state:      executing          XX            EXECUTING
     state:      delta-move-done    XX            DONE

     default-state:  STATE-ERR-MSG

  end-state-table

end-routine
```

Figure 12-5. The QC-DETACH routine.

In the QC-DETACH routine, RCS first gets the correct holster position information and moves the robot into contact with the holster. The application uses proximity sensors on the holster to determine when the robot contacts the holster. After the contact, QC-DETACH commands the quick change device to unlock, disconnecting the end effector and dropping it into the holster. The robot then slides the end effector to the seat position and moves away slightly. RCS checks the sensors in the robot wrist to verify the detach. After the robot detaches the end effector, it departs from the holster.

Each control level uses similar routines to decompose the commands at that level. Each control level consists of an executing routine that calls the routines PRE-PROCESS, STATE-TABLE, and POST-PROCESS. These routines perform the RCS functions of preprocessing, decision processing, and postprocessing, respectively.

T3 UNLOAD Tray Example

One function of the T3 robot is to unload incoming parts from a tray and store them in a buffer area. When the workstation sensors detect an incoming parts tray, the workstation controller sends an UNLOAD command to the TASK level. The workstation controller also sends the tray location point and the unload buffer area as UNLOAD command parameters. RCS must obtain information about the contents of the parts tray from the AMRF central database.

For this example, suppose the tray contains two rectangular part blanks and a valve body, as shown in Figure 12-6.

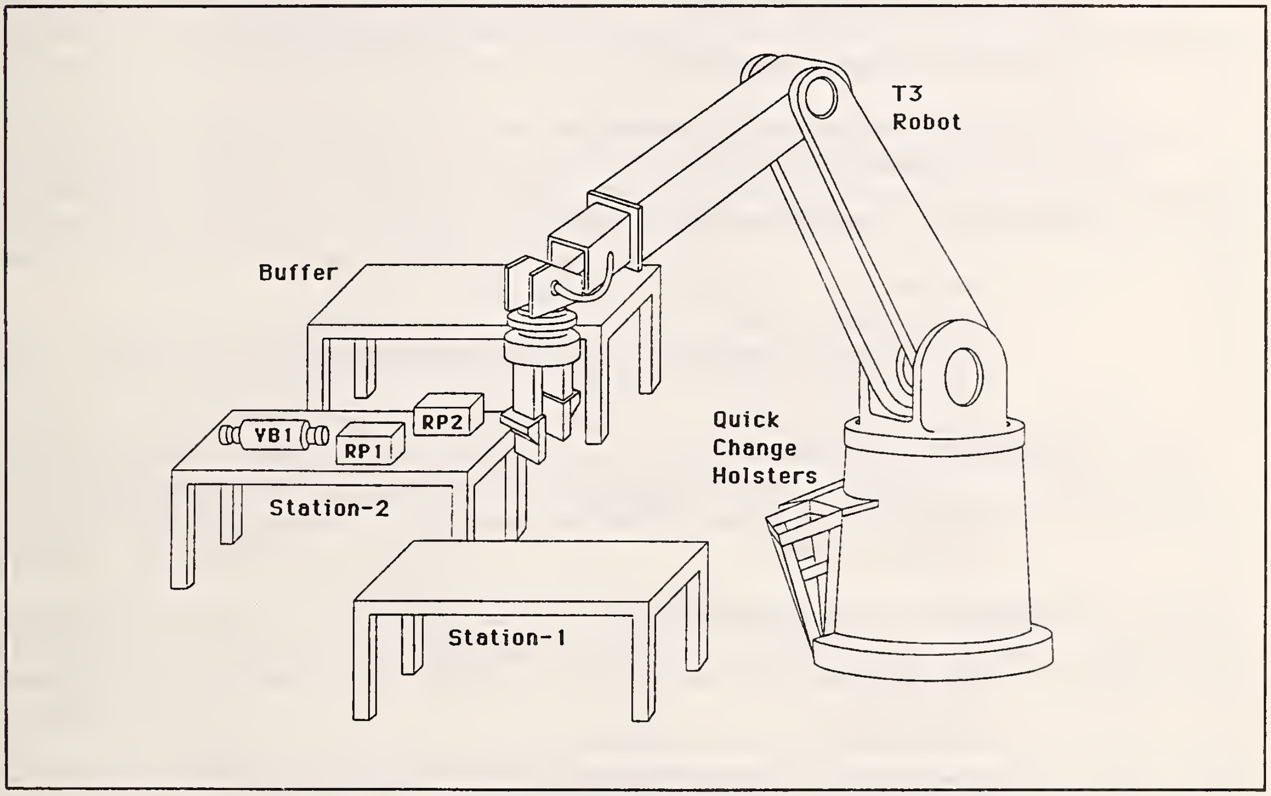


Figure 12-6. Example UNLOAD situation for the T3 robot.

The rectangular parts, RP1 and RP2, require the T3 to use a parts gripper, and the valve body, VB1, requires the T3 to use a valve gripper. Assume that the rectangular parts gripper is attached to the robot. Figure 12-7 gives a pseudocode example of the sequence of control statements generated by the UNLOAD command.

Figure 12-7 lists statements down to the E-MOVE control level. Each indentation indicates statements at the next lower control level.

```
UNLOAD ( STATION-2, BUFFER, SAFE )

     GET-OBJECT ( RP1, STATION-2, SECTOR-3 )

          MOV-TO-VU ( STATION-2, SECTOR-3 )
          LOCATE-OBJ ( RP1 )
          MOV-TO-OBJ ( RP1 )
          GRASP-OBJ ( RP1 )

     PLACE-OBJ ( RP1, BUFFER )

          MOV-OBJ-TO ( RP1, BUFFER )
          RELEASE ( RP1 )

     GET-OBJECT ( RP2, STATION-2, SECTOR-4 )

          MOV-TO-VU ( STATION-2, SECTOR-4 )
          LOCATE-OBJ ( RP2 )
          MOV-TO-OBJ ( RP2 )
          GRASP-OBJ ( RP2 )

     PLACE-OBJ ( RP2 BUFFER )

          MOV-OBJ-TO ( RP2, BUFFER )
          RELEASE ( RP2 )

     GET-ENDEFF ( VALVE-GRIPPER )

          MOV-TO-HOL ( )
          QC-DETACH ( )
          MOV-TO-HOL ( VALVE-GRIPPER )
          QC-ATTACH ( VALVE-GRIPPER )

     GET-OBJECT ( VB1, STATION-2, SECTOR-2 )

          MOV-TO-VU ( STATION-2, SECTOR-2 )
          LOCATE-OBJ ( VB1 )
          MOV-TO-OBJ ( VB1 )
          GRASP-OBJ ( VB1 )

     PLACE-OBJ ( VB1, BUFFER )

          MOV-OBJ-TO ( VB1, BUFFER )
          RELEASE ( VB1 )

     GOTO-ENDPT ( SAFE )
```

Figure 12-7. Pseudocode example of a T3 UNLOAD command.

STATION-2 and BUFFER are location-point arrays and SAFE is a location point. The SECTOR parameter indicates which array position contains the specified part. The TASK level obtains the value of SECTOR from the AMRF database before the first SUBTASK command, GET-OBJECT.

The objects RP1, RP2, and VB1 have associated grip and vision information. The control-level routines obtain much of the information needed to execute the UNLOAD task from the files in common memory. For example, when the T3 must move from the BUFFER to the holsters for a quick change operation, the system searches the trajectory file for a trajectory between the array BUFFER and the fixed holster location point HOL-SAFE. If the system finds a trajectory, it uses the path-points and velocity parameters to move the robot; otherwise, it uses the end points and default parameters.

The UNLOAD example includes three sequences of GET-OBJECT/PLACE-OBJ operations--one sequence for each object. The example includes an additional GET-ENDEFF operation after the first two objects, because the T3 must change grippers to pick up the valve body part. After the T3 unloads the valve body part, it moves to the location labeled SAFE.

12.2 THE FIELD MATERIEL-HANDLING ROBOT CONTROL SYSTEM

The U.S. Army Human Engineering Laboratory is developing a Field Materiel-Handling Robot (FMR). When completed, the FMR project will include a specially developed robot designed to automate materiel handling. The Human Engineering Laboratory is sponsoring NBS to develop an RCS-based robot controller for the FMR project.

One proposed task for the FMR is to locate, acquire, and transfer a randomly oriented pallet load from a truck bed to a conveyor. The following sections describe the hardware and software configurations required to accomplish the transfer task.

The FMR application uses an extended version of RSL. The extensions add path-points to use the sensors mounted on the FMR end effector. The discussion of this example assumes that you are familiar with the information on RSL in Chapter 9, "Robot Sensor Language (RSL)", and Chapter 1Ø, "RSL Control Levels".

Hardware Configuration for the FMR

The specifications for the FMR robot include a reach of 25 feet with the ability to move a 4,ØØØ-pound load at speeds of up to 15Ø inches per second. Commercially available robots cannot meet these specifications. NBS uses the Unimate 4ØØØ robot to test software and sensor development.

NBS developed a forklift-like end effector for the Unimate 4ØØØ robot. The fork enables the robot to insert tines into a pallet to transport the materiel securely on the pallet. The fork also provides a structure on which to place the sensors needed to position the fork for picking up a randomly oriented pallet. Figure 12-8 shows the fork and sensor hardware and the associated coordinate frame.

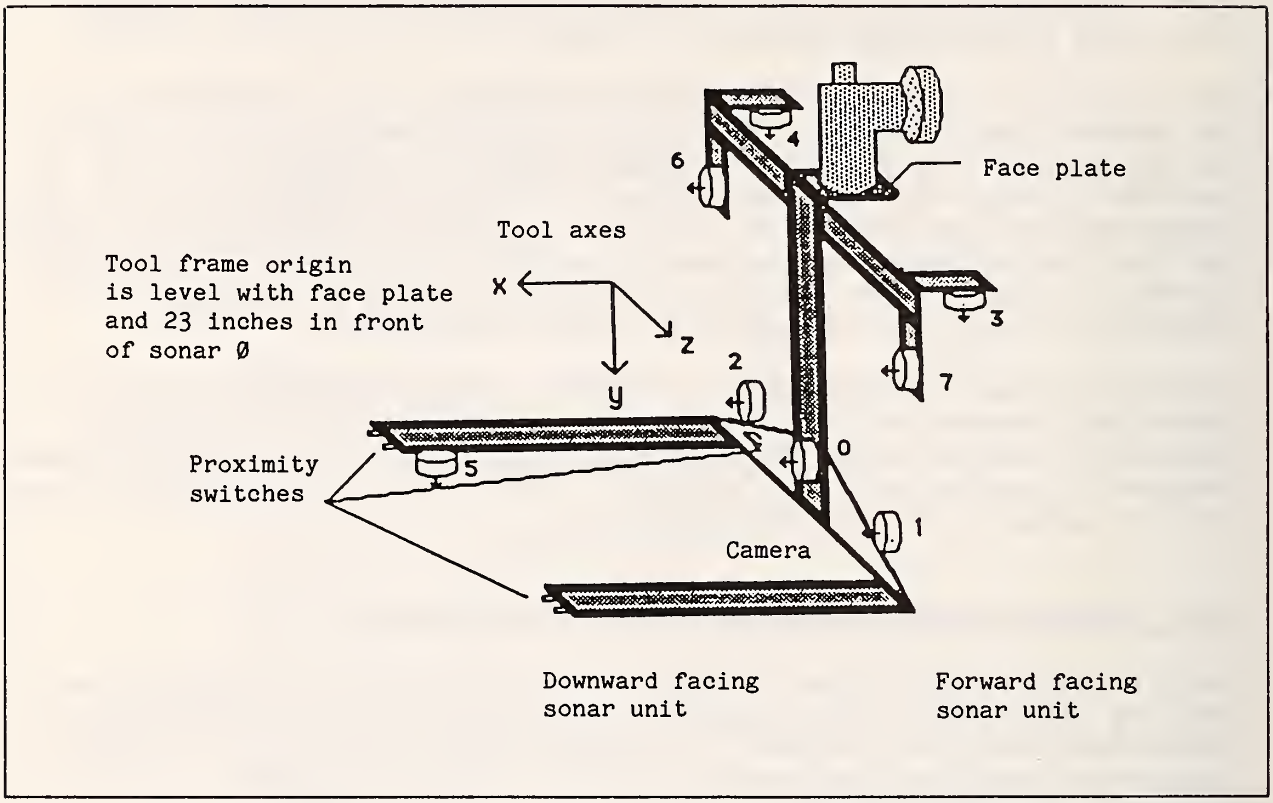


Figure 12-8. Fork and sensor hardware for the FMR.

The sensor hardware enables the robot to orient the fork with the truck bed, to align the fork with the desired entry side of the pallet, and to guide the fork under the pallet. The sensors are Polaroid ultrasonic ranging devices or sonars, and optical proximity switches. Routines can control robot motions, such as translations along and rotations about the tool axes, using the sonar sensors shown in Figure 12-8. For example, routines can control a roll motion about the X-axis using sonars 3 and 4.

The FMR application uses the same RCS hardware configuration as RSL. The FMR contains four 86/3Ø processor boards containing the four RSL control levels TASK, PATH, PRIM, and JOINT; a 512K memory board for common memory; and the tape and disk controller board. The boards communicate through a MULTIBUS.

Figure 12-9 shows the hardware configuration for the FMR example.

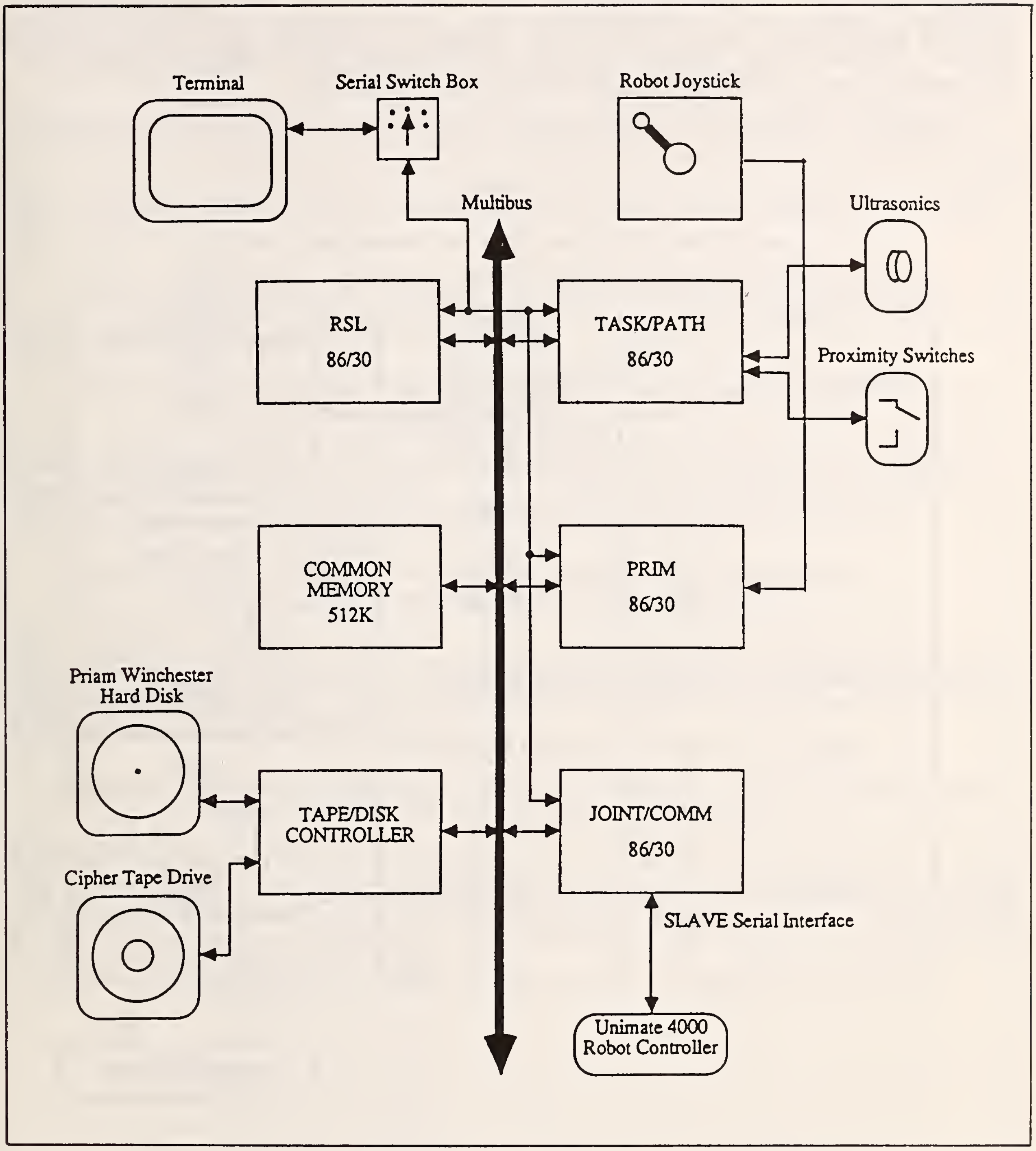


Figure 12-9. RCS hardware configuration for the FMR.

Software Configuration for the FMR

The FMR application uses the same RCS software configuration as the RSL application. Figure 12-1Ø reviews the RSL control-level hierarchy and shows the RSL data types and common memory data structure files.

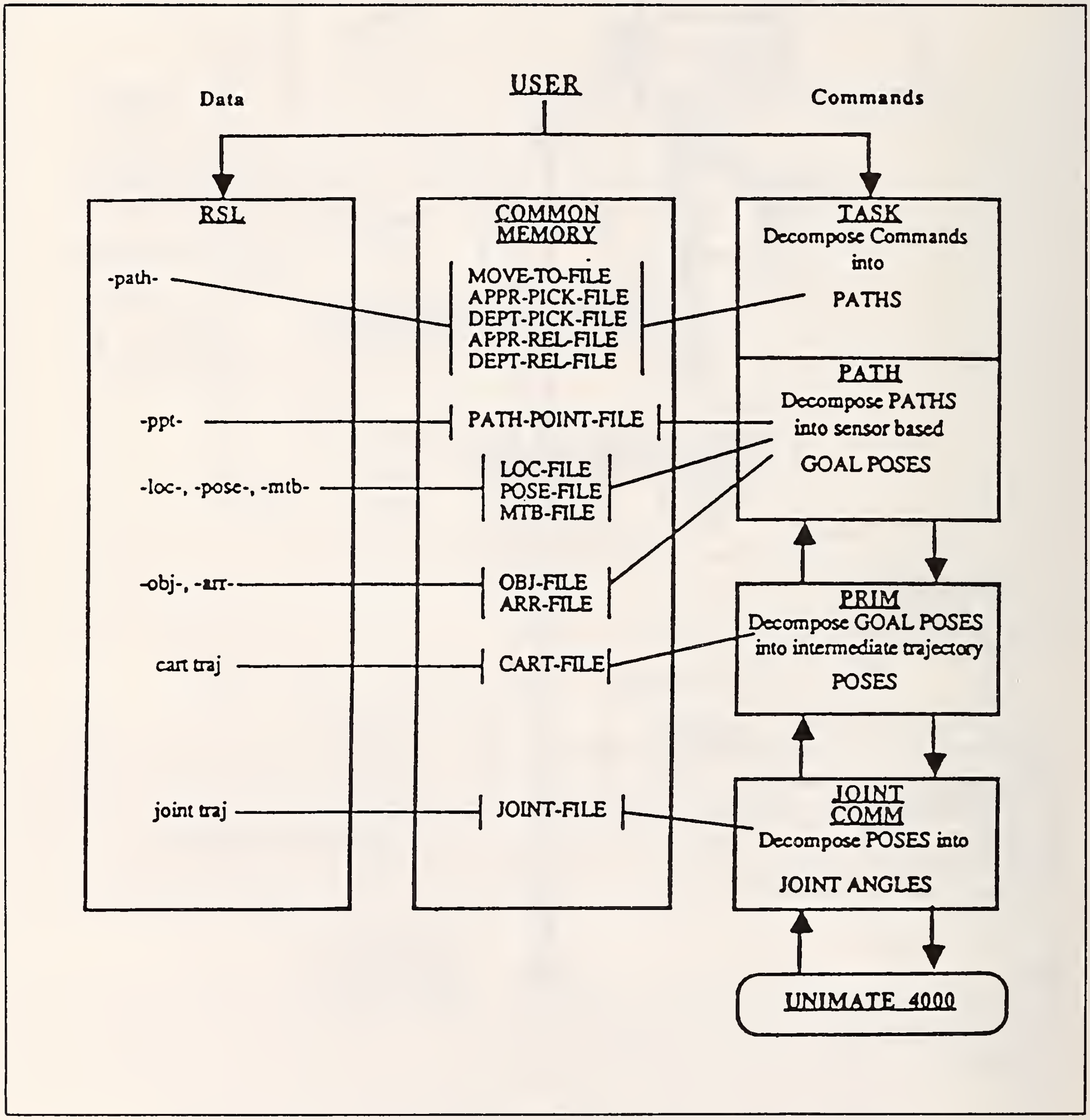


Figure 12-1Ø. RCS software configuration for the FMR.

To locate a randomly oriented pallet, the FMR application extended RSL by adding path-points that use the sonar and proximity sensors. The new path-points control individual sensor motion, scan the work volume and analyze the orientation of a pallet, and combine sequences of lower-level path-points so that you can execute these sequences of path-points in parallel.

The new path-points are edge, range, equate, scan, align-grip, approach-pallet, and pickup-pallet. The application also uses the RSL goto path-point.

The following list provides the syntax and a brief description of each FMR path-point.

```
     goto  [location type] [location name] [traj type] [traj parameters]
```

Moves the robot directly to the location.

```
     edge  [sonar#] [range] [delta] [location type] [location name]
           [traj type] [traj parameters]
```

Moves the robot toward the location name looking for an edge. The delta parameter describes how large the edge is, and range specifies at what distance to start looking for the edge.

```
     range  [sonar#] [range] [threshold] [axis] [location type]
            [location name] [traj type] [traj parameters]
```

Moves the robot along the specified axis of the Cartesian frame specified by the location type and location name. The robot moves to the range measured by sonar (within threshold). The commanded translation is positive if the range minus the sonar is greater than Ø.

```
     equate  [sonar#1] [sonar#2] [threshold] [offset] [axis] [location type]
             [location name] [traj type] [traj parameters]
```

Rotates about the axis of the Cartesian frame specified by the location type and location name until the two selected sonars take the same reading (within threshold). The rotation is positive if sonar#1 minus sonar#2 is greater than Ø. If two sonars are on different but parallel planes, you can specify the difference between them in offset.

```
scan  [sonar#] [skip readings] [max object range] [min range delta]
      [extra] [pose name] [location type] [location name] [traj type]
      [traj parameters]
```

Determines the nominal location of a rectangular object. The robot scans an area taking range readings starting from the current position and ending at the location name. When the sonars take readings, the routines associated with scan link the readings to the position of the robot and store the positions in a file. The scan path-point returns the nominal pose of the object in pose name.

```
align-grip  [perm-edge-val] [perm-equal-val] [s1s2-offset] [extra]
            [long-integer] [min-total-long] [max-short-side]
            [side-equate-thresh] [location type] [location name]
            [goto-traj-type] [traj parameters] [edge traj type]
            [traj parameters] [equate traj type] [traj parameters]
```

Uses information from the scan path-point and the current sonar data to determine if the long side of the pallet faces the fork. By calling the edge, equate, and goto commands, the align-grip path-point uses the trajectory parameters to align the fork with the long side of the pallet.

Figure 12-11 shows a sequence of path-points similar to the path-points that align-grip might execute. However, the figure makes some assumptions that cannot be made until after sonar data processing, such as which direction to move when looking for the edge.

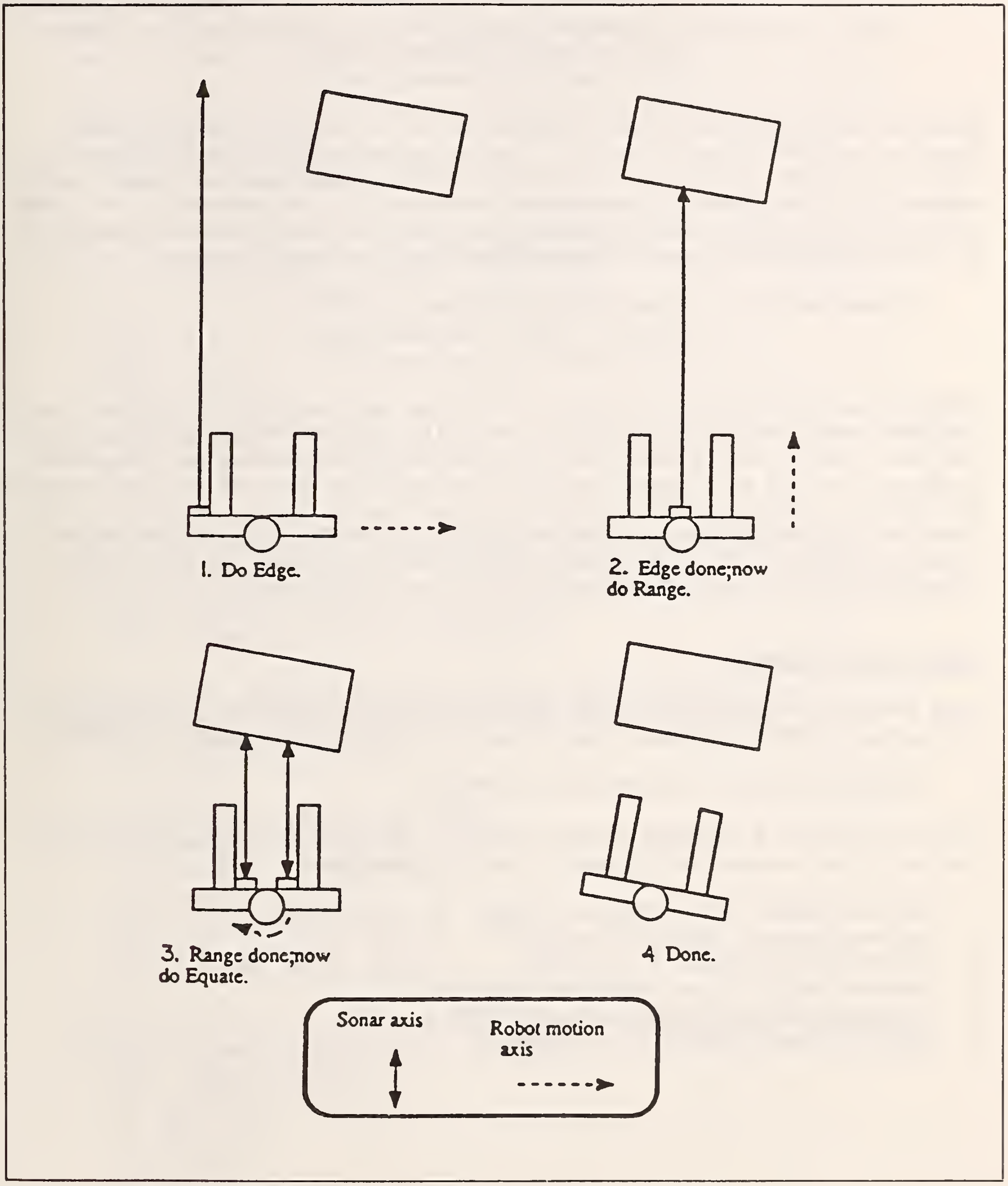


Figure 12-11. FMR path-point examples.

```
     approach-pallet  [t-x-sonar#] [t-x-range] [t-x-threshold]
                      [t-y-sonar#] [t-y-range] [t-y-threshold]
                      [r-x-sonar#1] [r-x-sonar#2] [r-x-offset] [r-x-threshold]
                      [r-y-sonar#1] [r-y-sonar#2] [r-y-offset] [r-y-threshold]
                      [location type] [location name]
                      [traj type] [traj parameters]
```

Achieves the proper height and orientation of the fork before the robot inserts the fork under the pallet. The approach-pallet path-point calculates four tool motions similar to the calculations for the range and equate path-points. The t-x- and t-y- parameters correspond to the parameters for range path-points along the x and y axes. The r-x- and r-y- parameters correspond to the parameters for equate path-points about the x and y axes.

```
     pickup-pallet    [sonar#] [z-correction] [extra]
                      [location type] [location name]
                      [traj type] [traj parameters]
```

Ensures that the fork tines do not hit the pallet feet while the robot inserts the fork under the pallet. Optical proximity sensors detect when the tines are too close to the pallet feet. The specified sonar# obtains the nominal distance the fork must travel before a switch at the rear of the fork engages the pallet. When a proximity sensor senses a pallet foot, the robot translates in the appropriate z direction until the sensor is no longer active. Because of the narrow beam of the sensors, the robot also travels an additional z-correction after the sensor has cleared the foot.

FMR Transfer Example

This example provides the source code for the paths required to execute the following TRANSFER command, which transfers PALLET 1 from TRUCK to CONVEYOR.

```
     TRANSFER PALLET 1 loc TRUCK loc CONVEYOR
```

The six steps of a TRANSFER command specify the paths for the robot to pick up, move, and release the pallet. The six steps are:

1. move-to PALLET 1 loc HOME loc TRUCK
2. approach-pickup PALLET 1 loc TRUCK
3. depart-pickup PALLET 1 loc TRUCK
4. move-to PALLET 1 loc TRUCK loc CONVEYOR
5. approach-release PALLET 1 loc CONVEYOR
6. depart-release PALLET 1 loc CONVEYOR

To execute the TRANSFER command, you must define and compile all six path steps from the RSL board. The RSL code for the paths and path-points needed to accomplish the TRANSFER command follow. The % character indicates a comment line.

```
% Move to a starting position (from home) before scanning the truck.
  -path-  move-to PALLET 1 loc HOME loc TRUCK

   % Move to the predefined scan start location using a fast joint
   % trajectory motion.
     -ppt-  goto loc SCAN-START
       joint 3Ø.Ø 3Ø.Ø 5.Ø

% Scan the truck from left to right searching for a pallet, align
% with long side of pallet, and engage fork.
  -path-  approach-pickup PALLET 1 loc TRUCK

   % Orient fork with the truck bed
   % adjust roll.
     -ppt-  equate 4 3 .5 .Ø X tool nul
       cart .2 .15 .25 .3 .2 .25
   % adjust pitch.
     -ppt-  equate 4 5 .5 2Ø.Ø Z tool nul
       cart .2 .15 .25 .3 .2 .25

   % Fork is now parallel with truck bed.  Scan for pallet, return
   % position in PALLET-POSE.  SCAN-MTB causes the fork to travel in a
   % base rotation motion.
     -ppt-  scan Ø Ø 5Ø.Ø .5 Ø PALLET-POSE
       tool SCAN-MTB joint 25.Ø 25.Ø 5.Ø

   % Move to loc PALLET-LOC which consists of PALLET-POSE and a
   % movetable to position the fork in front of the pallet.
     -ppt-  goto loc PALLET-LOC
       joint 25.Ø 25.Ø 1.Ø

   % Find the long side of the pallet.
     -ppt-  align-grip
       4.Ø 8.Ø Ø.Ø .75 15.Ø 14.Ø 5
       tool nul
       cart .3 .15 .25 .3 .15 .25
       cart .3 .15 .25 .3 .15 .25
       cart .3 .15 .25 .3 .15 .25

   % Lower the fork to the truck bed.  Ensure that the fork is parallel
   % with the face of pallet and truck bed.
     -ppt-  approach-pallet
       Ø  17.75 .5
       4  28.5  .25
       4 3 .Ø .25
       7 4 .Ø .5
       cart .3 .Ø7 .25 .3 .Ø7 .25
```

```
 % Guide fork under the pallet until the rear switch signals engagement.
  -ppt-  pickup-pallet 7 .5 1.Ø
   cart .3 .3 .25 .3 .15 .25

% Depart by picking up the pallet and tilting it back.
 -path-  depart-pickup PALLET 1 loc TRUCK

 % PALLET-UP1 is an offset up from the current position under the
 % pallet.
  -ppt-  goto tool PALLET-UP1
   cart .3 .3 .25 .3 .3 .25

 % PALLET-UP2 tilts the pallet back.
  -ppt-  goto tool PALLET-UP2
   cart .3 .3 .25 .3 .3 .25

% Move to an intermediate safe position.
 -path-  move-to PALLET 1 loc TRUCK loc CONVEYOR

 % CONV-SAVE is a predefined safe position near the conveyor, which
 % can be approached using a high-joint velocity.
  -ppt-  goto loc CONV-SAFE
   joint 1Ø.Ø 3Ø.Ø 5.Ø

% Approach conveyor, a predefined location, and release pallet.
 -path-  approach-release PALLET 1 loc CONVEYOR

 % CONV-UP1 is an offset up from CONVEYOR goal.  Goal type
 % takes the movetable (CONV-UP1) and adds it to the path goal
 % (CONVEYOR).
  -ppt-  goto goal CONV-UP1
   cart .3 .3 .25 .3 .3 .25

 % Place pallet at the goal CONVEYOR.
  -ppt-  goto goal nul
   cart .3 .3 .25 .3 .3 .25

% Extract fork tines.
 -path-  depart-release PALLET 1 loc CONVEYOR

 % CONV-BACK is an offset up from the CONVEYOR goal.
  -ppt-  goto goal CONV-BACK
   cart .3 .25 .25 .3 .25 .2

  -ppt- goto loc CONV-SAFE
   joint 1Ø.Ø 3Ø.Ø 5.Ø
```

Chapter 13
DEBUGGING TECHNIQUES

This chapter describes several RCS debugging techniques, including debugging interactively using Show mode, single-stepping through a routine, single-stepping control levels, isolating problems, creating debugging routines, and using other debugging techniques.

13.1 DEBUGGING INTERACTIVELY USING SHOW MODE

In most high-level languages (when an interactive debugging system is unavailable), you debug routines, or procedures, by inserting write or print statements within the code, recompiling the code, and executing the code again. SMACRO is interactive. You can perform the following functions at the terminal: execute individual routines, assign values to variables, and display the contents of variables.

Show mode is a useful debugging tool, enabling you to display information at the terminal. In Show mode, when you enter the name of an owner, RCS displays the name of the owner and the contents of all the members. When you enter the name of a variable, RCS displays the value of the variable.

For example, if you are using the RSL application, move to the TASK&PATH board and change to Show mode by entering :S. Enter TDEF to ensure that you are in the TASK vocabulary and then enter input-command, the variable that contains the command that the user passes to the TASK level. RCS displays the contents of the input-command, for example, PAUSE.

To see the values of all members in the output status buffer for the TASK level, enter the name of the owner, output-status-var. RCS displays the name of the owner and the contents of the members of the owner:

```
output-status-var
(iv) cycle-#-status-out = 15
(iv) inc-command-#-echo-out = 1
(strv) command-echo-out = PAUSE

(iv) status-report = Ø
(iv) status-arg-out = Ø
```

For example, to test the robot transforms, set up block 21 on the JOINT/COMM board as follows:

```
:S  joint/comm>21 LIST
21
 Ø                                      $DEF
 1  :R
 2  1Ø.deg  2Ø.deg  3Ø.deg
 3  4Ø.deg  5Ø.deg  6Ø.deg ==> servo-com-joint
 4  TIME-P JOINT>CART
 5  TIME-P CART>JOINT
 6  :S
 7  CR JS servo-com-joint
 8  CR JS xg
 9  CR PMS pose
1Ø  CR conf-flag
11
12
13
14
15
```

Line 2 sets servo-com-joint to the desired joint values. Line 4 executes JOINT>CART and displays the execution time. The input to JOINT>CART is in servo-com-joint, the output is in pose and conf-flags. Line 5 executes CART>JOINT and displays the execution time. The input to CART>JOINT is in pose and conf-flags, the output is in xg. Lines 7 through 1Ø display the inputs and outputs. Load block 21 by entering 21 LOAD. RCS yields the following output:

```
JOINT>CART 975Ø usec
CART>JOINT 83ØØ usec
servo-com-joint
1Ø.         2Ø.         3Ø.         4Ø.         5Ø.         6Ø.
xg
1Ø.         2Ø.         3Ø.         4Ø.         5Ø.         6Ø.
pose as matrix
            x^          y^          z^          xyz
            -Ø.6366     Ø.Ø2272     Ø.77Ø9      25.66
            Ø.7712      Ø.Ø296      Ø.6359      9.Ø82
            -Ø.ØØ8369   Ø.9993      -Ø.Ø3636    39.23
(1:fa) conf-flag =
-1.         1.          1.
:S  joint/comm>
```

Because xg has the same values as servo-com-joint, the two transforms are inverses of each other, as required. Block 21 displays the pose as a matrix for easier interpretation. You can set the joint angles to simple values (for example, Ø or 9Ø degrees) to check the pose and conf-flag computation. To check the computation, return to the master board, edit line 2 of block 12021, move to the JOINT board, and reload block 21 (using ELOAD). Edit block 21 to contain the simple joint angles in the following listing:

```
0                                         $DEF
 1  :R
 2  0.deg  0.deg  90.deg
 3  0.deg  0.deg  0.deg ==> servo-com-joint
 4  TIME-P JOINT>CART
 5  TIME-P CART>JOINT
 6  :S
 7  CR JS servo-comm-joint
 8  CR JS XG
 9  CR PMS pose
10  CR conf-flag
11
12
13
14
15
:S rsl
```

Now load the block by entering 21 ELOAD. The output is:

```
JOINT>CART 7920 usec
CART>JOINT 6750 usec
servo-com-joint
0.0          0.0          90.          0.0          0.0          0.0
xg
0.0          0.0          90.          0.0          0.0          0.0
pose as matrix
             x^           y^           z^           xyz
             0.0          0.0          1.           23.62
             0.0          1.           0.0          4.488
             -1.          0.0          0.0          25.59
(1:fa) conf-flag =
-1.          1.           1.
:S  joint/comm>
```

These particular values correspond to the pose for the PUMA 760 with the elbow at 90 degrees.

13.2 SINGLE-STEPPING THROUGH A ROUTINE

To locate an error within a routine, single-step through the routine one line at a time. You can execute routines line by line in two ways: by making the entire routine a comment and then restoring the routine one line at a time, or by executing the FORTH equivalent of the code in the routine line by line at the terminal (if possible).

To make an entire routine into a comment, you should first copy the routine to an empty block. Then use the screen editor to enclose the routine, except for the first line of SMACRO code, in parentheses. Leave the screen editor, load the block, and execute the routine. If you do not get an error, go back into the screen editor and move the first parenthesis to include another line of code. Continue this procedure until you find the problem.

Executing the code at the terminal works best for routines that manipulate SMACRO files, because you can execute many of the SMACRO file operators directly at the terminal. These file operators are:

add-list
add-record
clear-list
~re-init-file
remove
retrieve
retrieve-l&r
store
store-l&r
store-source
update-header

Many other SMACRO operators have equivalents in FORTH that you can use to execute lines of code at the terminal. For example, the FORTH equivalent for the SMACRO assignment operator (=>) is the FORTH store command (~!).

13.3 SINGLE-STEPPING THROUGH THE RSL CONTROL LEVELS

Often the best way to isolate an error is to single-step through the control levels. In single-step, each step executes one communications cycle. To single-step through the control levels, follow this procedure:

1. On the RSL board, enter 923 LIST.

2. Use the screen editor to select between dummy and real execution. Making line 6 a comment selects single-step without the robot. Making line 5 a comment selects single-step with the robot.

3. Enter Task/path REMOTE to move to the TASK&PATH board.

4. Issue a command by loading a block that contains an RSL command. On the NBS system tape, the directory block 910 lists the blocks containing RSL commands. Enter 913 LOAD to issue a transfer task.

5. Enter ^X to return to the master board.

6. Enter HALT whenever you return to the master board from a slave board. Entering HALT makes the single-step output easier to read.

7. Enter PAGE TT to execute one communications cycle.

Figure 13-1 shows sample output that RCS displays when you enter TT.

```
task:  2 transfer nul Ø1 arr ARRAY ØØ Ø8 ;; loc CONVEYOR
path:  13 2 1 Ø Ø 1
133Ø usec
237Ø usec
:S task&path>
prim:  22 2 1 Ø Ø
 (fv) |xs| = Ø.Ø
 (fv) angRs = Ø.Ø
1Ø94Ø usec
:S prim>
joint: 533 2 1 Ø 1. 1.
servo-com-joint
53.3 -26.38 -79.Ø8 -174.4 55.98 161.4
3188Ø usec
```

Figure 13-1. Single-step output.

TT commands each board to load block 919 in its offset. (TASK&PATH loads TASK's 919, and not PATH's 919.) Block 919 on each level contains the words that execute that level, and any debug or display words you may have added. Here is block 919 on the TASK&PATH board:

```
                                 TASK DEFINITIONS
:R 1 remote-active !
   BOARD-PROCESS
:S
   PDEF
\ CR ppt-started
\ CR new-command
\ CR inc-command-#-out

CR :R TDEF process-time ~@ PRINT-TIME :S
CR :R PDEF process-time ~@ PRINT-TIME :S





```

Here is block 919 on the PRIM board:

```
                                     PRIM DEFINITIONS
:R 1 remote-active !
   BOARD-PROCESS
:S
  CR |xs|
  CR angRs
% CR current-vel
  CR :R process-time ~@  PRINT-TIME :S
% CR PMS joint-com-pose
```

Here is block 919 on the JOINT/COMM board:

```
% single step dummy mode           JOINT DEFINITIONS
:R
     1 remove-active !
     COMM-PROCESS JOINT-LEVEL
     done ~@ ==> status-in joint-out joint-in  12 ~MOVE
     inc-command-#-out ~@ ==> inc-command-#-echo-in
  CR JS servo-com-joint
% CR JS xg
% CR JS v-current
:S
% CR PMS pose
% CR conf-flag

CR :R process-time ~@ PRINT-TIME :S
```

Note: The TASK&PATH and PRIM boards simply run BOARD-PROCESS, but the JOINT/COMM board is more complex. The difference is because the JOINT/COMM board must run without the robot communications, which are usually included in BOARD-PROCESS for that board.

The following example shows output from TT taken from the middle of a TRANSFER task execution. The times shown are for 5 MHz boards.

```
     task: 9 TRANSFER nul Ø1 arr ARRAY ØØ Ø8 ;; loc CONVEYOR
           Ø Ø 2 Ø
     path: 8 2 Ø Ø Ø Ø
     117Ø usec
     254Ø usec
     :S task&path>
     prim: 1Ø 2 Ø Ø Ø
     (fv) |xs| = 5.263
     (fv) angRs = Ø.Ø
     1445Ø usec
     :S prim>
     joint: 78 2 1 Ø 1. 1.
     servo-com-joint
     13.45     Ø53.7     -54.Ø4     -Ø.ØØ3327     -72.47     18.45
     32Ø3Ø usec
     :S joint/comm>
     :S rsl>
```

The Display routine on each level generates the lines labeled with the level name. (For example, the PRIM level Display routine generates the line prim: 1Ø 2 Ø Ø Ø. TT sets the print-f of each level to true.) Other words in the 919 blocks on each level generate the additional output lines. The rest of this section describes the output in detail, line by line.

```
     task: 9 TRANSFER nul Ø1 arr ARRAY ØØ Ø8 ;; loc CONVEYOR
           Ø Ø 2 Ø
```

These lines are from the TASK Display routine. It shows the values of inc-command-#-in (9), input-command (TRANSFER nul Ø1 arr ARRAY ØØ Ø8 ;; loc CONVEYOR), status-report (Ø is executing), status-arg-out (Ø is noerror), state-label (2), and sector-count (Ø).

```
     path: 8 2 Ø Ø Ø Ø
```

This line is from the PATH Display routine. It shows the values of inc-command-#-in (8), input-command (2 - PATH), status-report (Ø - executing), status-arg-out (Ø - noerror), ppt-command (Ø - goto), and ppt-done (Ø - false).

```
     117Ø usec
```

This line is generated from the line "CR :R TDEF process-time ~@ PRINT-TIME :S" in TASK block 919. It displays the execution time for the TASK level.

```
    2540 usec
```

This line is generated by the line "CR :R PDEF process-time ~@ PRINT-TIME :S" in TASK block 919. It displays the execution time for the PATH level.

```
    :S task&path>
```

This line is the prompt for the TASK&PATH board.

```
    prim:  1Ø 2 Ø Ø Ø
```

This line is from the PRIM Display routine. It shows the values of inc-command-#-in (1Ø), input-command (2 - TRAJ), status-report (Ø - executing), status-arg-out (Ø - noerror), and traj-type-in (Ø - cart).

```
    (fv) |xs| = 5.263
```

This line is generated by the line "CR |xs|" in PRIM block 919. It displays the translational distance from the goal.

```
    (fv) angRs = Ø.Ø
```

This line is from the line "CR angRs" in PRIM block 919. It displays the rotational distance from the goal.

```
    1445Ø usec
```

This line is generated by the line "CR :R process-time ~@ PRINT-TIME :S" in PRIM block 919. It displays the execution time for the PRIM level.

```
    :S prim>
```

This line is the PRIM board prompt.

```
    joint: 78 2 1 Ø 1. 1.
```

This line is from the JOINT Display routine. It shows the values of inc-command-#-in (78), input-command (2 - CARTESIAN), status-report (1 - done), status-arg-out (Ø - noerror), v-scale-min (1.), and a-scale-min (1.).

```
    servo-com-joint
    13.45     Ø53.7     -54.Ø4     -Ø.ØØ3327     -72.47     18.45
```

This line is generated by the line "CR JS servo-com-joint" in JOINT block 919. It displays the output joint angles in degrees.

```
     32Ø3Ø usec
```

This line is generated by the line "CR :R process-time ~@ PRINT-TIME :S" in JOINT block 919. It displays the execution time for the JOINT level.

```
     :S joint/comm>
     :S rsl>
```

These lines are the prompts for the JOINT/COMM and RSL boards.

The word Display on each control level, and any variables executed in Show mode in block 919 on each control level, generate this output. You can execute a number of cycles and watch the single-step display change by entering [#cycles] TTT. Single-stepping through the control levels is often helpful in locating errors.

13.4 ISOLATING PROBLEMS

When the system aborts a robot task or does not complete a task, debug the problem first by determining where the problem occurred. If you can see that the robot is not performing a task correctly, finding the cause of the problem is usually easy. For example, if the robot starts a transfer task and hits an object while moving directly to the first sector of an array, you have to add a path-point to the transfer routine that first moves the robot to a safe location.

If the error is in the software, you have to determine where the problem occurred and correct it. To illustrate this procedure, the following example introduces an error onto the system tape, isolates the problem by single-stepping the control levels and tracing the execution of the task, and restores RCS to its previous condition.

1. Start the system in background mode by entering the following commands:

   ```
   Ø CBOOT Ø MBOOT init-cm 2 D>M
   922 LOAD
   ```

2. Load block 928 to show the contents of the status-report and status-arg-out variables on each level by entering the command:

   ```
   928 LOAD
   ```

3. Edit block 981 on the PRIM level, as follows:

   3981 ED

   Introduce an error by editing block 981 to make line 9 a comment and to include a new line 8 that sets tool-mtb-^ to Ø. The block should now look as follows:

```
CHARACTER OPS        LINE OPS          BLOCK OPS            SPECIAL OPS
^A Insert On/Off     ^C Copy Line      ^F Forward Block     PF1 Home    PF2 Clear
^E Erase Char        ^D Delete Line    ^P Previous Block    ^U Undo Block
^K Mark to Keep      ^O Open Line      ^G Goto Block        ^W Want string
^Z Input Keep                          ^T Transfer          ^N Next Want
                                       ^R Retire(quit)      ^X Cancel

  BLOCK    3981   11981

 Ø                                                          PRIM DEFINITIONS |
 1 :R                                                                        |
 2                                                                           |
 3 RR-^ PRIM>JOINT-POSE retrieve  rr-^ ~@ ==> prim>joint-pose-rr-^           |
 4 RR-^ PRIM>JOINT-TRAJ retrieve  rr-^ ~@ ==> prim>joint-traj-rr-^           |
 5                                                                           |
 6 RR-^ RBT-POSE       retrieve  rr-^ ~@ ==> rbt-pose-rr-^                   |
 7                                                                           |
 8 Ø ==> tool-mtb-^                                                          |
 9 % MTB-^ TOOL-MTB     retrieve record# ~@ ==> tool-mtb-^                   |
1Ø MTB-^ INV-TOOL-MTB retrieve record# ~@ ==> inv-tool-mtb-^                 |
11                                                                           |
12                                                                           |
13                                                                           |
14                                                                           |
15                                                                           |
```

4. Load the edited block containing the error by halting RCS execution and moving to the PRIM board, as follows:

   HALT
   Prim REMOTE
   981 ELOAD

5. Now single-step through the control levels until the error occurs. Enter:

   ^X
   HALT
   PAGE TT

   The error message **BOARD-PROCESS ABORTED** appears after the first cycle of single-step. Because the message appears before the prompt for the PRIM board, the error occurred on the PRIM level.

Note: Some errors do not occur until RCS has single-stepped through many cycles. Single-stepping through the control levels is often a useful debugging tool.

6. Move to the PRIM board and re-execute the word that caused the error. The error occurs again, enabling you to locate the block that aborted. Enter:

   ```
   Prim REMOTE
   LOC BOARD-PROCESS
   ```

7. As you can see on the display, the routine BOARD-PROCESS executes several other routines. Because PRIM-LEVEL is the only routine an RCS user is likely to have changed, locate that routine by entering:

   ```
   LOC PRIM-LEVEL
   ```

8. The routine PRIM-LEVEL also executes several routines, but an RCS user is likely to have changed only the routines PRE-PROCESS, COMMAND-PROCESS, POST-PROCESS, and Display. Execute these routines in the same order as PRIM-LEVEL executes them, stopping when an error occurs. Enter:

   ```
   PRE-PROCESS
   COMMAND-PROCESS
   ```

   Executing COMMAND-PROCESS should cause the error to occur again.

9. Enter the following commands to display the last word executed and to locate that word:

   ```
   \N
   \L
   ```

1Ø. Trace the execution of the routine COMMAND-PROCESS by displaying variables in Show mode to isolate the problem further. Enter:

   ```
   :S
   status-report
   error
   hold-set
   false
   input-command
   prim-restart
   ```

11. At this point, COMMAND-PROCESS executes the routine PRIM-RESTART. Enter:

   ```
   PRIM-RESTART
   ```

   The error should occur again. Now locate PRIM-RESTART by entering:

   ```
   \L
   ```

12. Trace the execution of the routine PRIM-RESTART by displaying variables in SHOW mode to isolate the problem further, as follows (the command N L displays the rest of the routine):

```
:S
new-command
true
joint-status
error
N L
joint-status
done
```

13. At this point, PRIM-RESTART executes the routine FEEDBACK-POSE. Execute FEEDBACK-POSE, as follows:

```
FEEDBACK-POSE
```

The error should occur again. Now locate FEEDBACK-POSE by entering:

```
\L
```

14. Because the original error message said RCS aborted because of a Ø record#, look for statements in FEEDBACK-POSE that assign values to record# and display those variables. Enter:

```
:S
joint-pose-^
tool-mtb-^
```

Note: You cannot execute retrieve-from-field from the terminal.

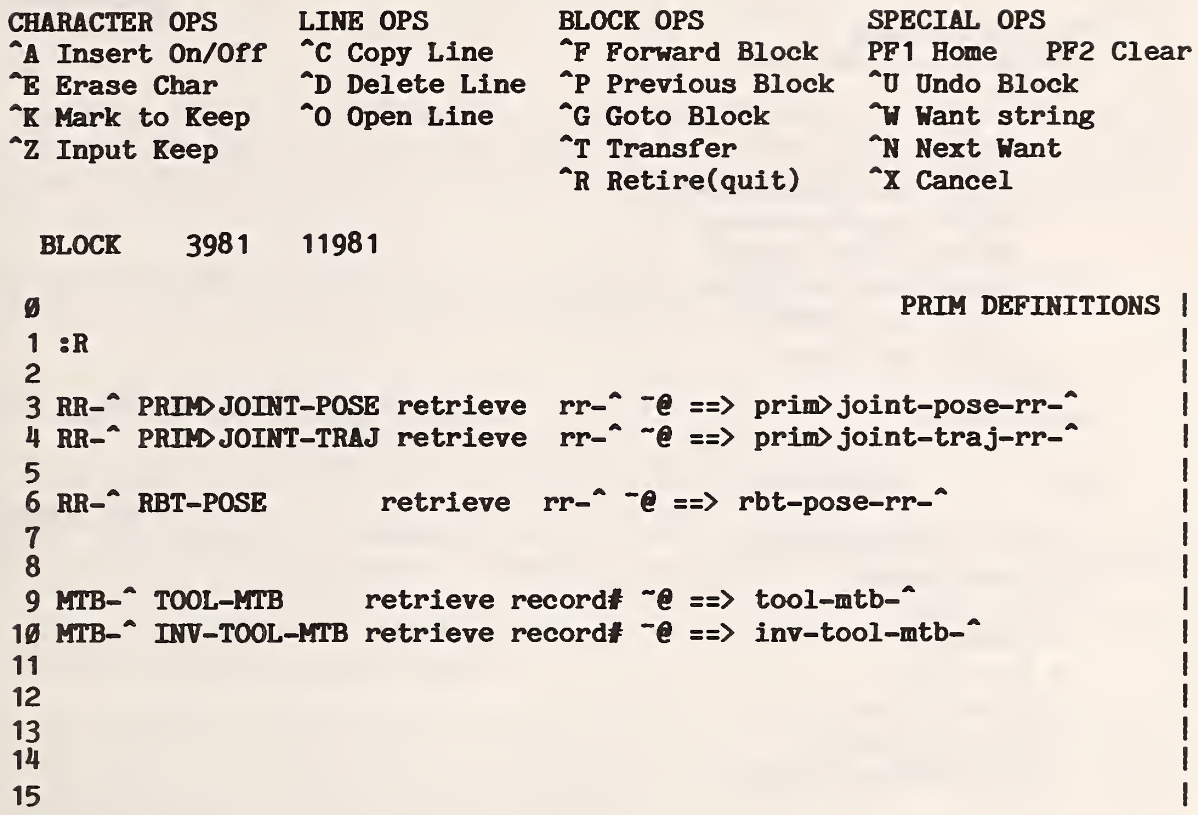
15. Because tool-mtb-^ equals Ø, you need to find where the variable tool-mtb-^ is initialized. Use the directory block structure to determine where PRIM level round-robin variables are initialized by entering:

```
Ø LIST
9ØØ LIST
98Ø LIST
981 LIST
```

16. Return block 981 to its original state by entering:

```
^X
RED
```

Edit the block to contain the following code:



```
CHARACTER OPS      LINE OPS        BLOCK OPS           SPECIAL OPS
^A Insert On/Off   ^C Copy Line    ^F Forward Block    PF1 Home   PF2 Clear
^E Erase Char      ^D Delete Line  ^P Previous Block   ^U Undo Block
^K Mark to Keep    ^O Open Line    ^G Goto Block       ^W Want string
^Z Input Keep                      ^T Transfer         ^N Next Want
                                   ^R Retire(quit)     ^X Cancel

 BLOCK   3981   11981

                                                             PRIM DEFINITIONS |
:R                                                                            |
                                                                              |
RR-^ PRIM>JOINT-POSE retrieve  rr-^ ~@ ==> prim>joint-pose-rr-^               |
RR-^ PRIM>JOINT-TRAJ retrieve  rr-^ ~@ ==> prim>joint-traj-rr-^               |
                                                                              |
RR-^ RBT-POSE       retrieve  rr-^ ~@ ==> rbt-pose-rr-^                       |
                                                                              |
                                                                              |
MTB-^ TOOL-MTB     retrieve record# ~@ ==> tool-mtb-^                         |
MTB-^ INV-TOOL-MTB retrieve record# ~@ ==> inv-tool-mtb-^                     |
                                                                              |
                                                                              |
                                                                              |
                                                                              |
                                                                              |
```

When you extend RSL, use this procedure to debug errors.

13.5 CREATING DEBUGGING ROUTINES

You can create your own routines to help debug problems. RSL contains several debugging routines that display variables. Each control level contains a routine called Display and a list owner called List-display. Figure 13-2 lists the routines Display and List-display for the PATH level. You can write similar routines to display other variables.

```
    1909       10909 ABS
                                                        PATH DEFINITIONS
    routine Display
      if print-f (EQ) true
      then
        ~F CR
        ~PRINT" path: "
        ~PRINT inc-command-#-in
        ~PRINT input-command
        ~PRINT status-report
        ~PRINT status-arg-out
        ~PRINT ppt-command
        ~PRINT ppt-done
      endif
    end-routine

    1908       10908 ABS
                                                        PATH DEFINITIONS
      LIST-O  List-display
      m new-command
      m ppt-done
      m new-ppt
      m path-done
      m prim-status
      m output-command
      m inc-command-#-out
```

Figure 13-2. Display routine and List-display list owner for the PATH level.

Note: In debugging routines when RSL is running in the background, you cannot see the intermediate results of calculations. To show the intermediate results, you can run RSL in the foreground and include ~PRINT statements. However, ~PRINT statements slow RCS operation.

13.6 USING OTHER DEBUGGING TECHNIQUES

You can also use other techniques to debug RCS hardware, RCS software, and RSL software.

RCS Hardware Debugging Techniques

You can add the following optional RCS hardware to make some debugging operations easier:

- A hardware switch box enables you to use the hardware switch to communicate directly with the other boards in the system. When only one board in the system malfunctions, use the switch box to reset that board without rebooting the entire system.

- An oscilloscope on the bus busy enables you to see the signal on the MULTIBUS. The signal goes down when the bus is busy.

RCS Software Debugging Techniques

Some techniques for debugging RCS software problems include:

- Check your spelling. Remember, reserved words in the FORTH dictionary have three significant characters. SMACRO words have 31 significant characters.

- If the vocabulary of a control level is full, increase the size allocated in block 9 of that control level.

- If the system dictionary for a board is full, edit the system vocabulary declaration for that board to increase the size of the dictionary. These declarations are located in absolute blocks 14Ø6 to 1412. You must reload the base system for this to be effective.

- If you cannot access a slave board, try entering [board name] Board-sem-initialize.

- Make sure you did not put a vocabulary identifier in routines that are more than one block long.

RSL Software Debugging Techniques

Some techniques for debugging RSL software problems include:

- To check the current status of all control levels, enter 8928 ØLOAD to display the status and status-arg-out information from all control levels.

- Check that all RSL source code blocks contain :R. The RSL compiler does not work in other modes.

- If you are not sure what RSL code is loaded, enter HALT to halt the system and then enter 899Ø ØLOAD to reload the code.

- If the robot continues to reach a wrist joint-limit when moving to a pose, try changing the wrist configuration by editing the wrist configuration flag in the pose definition. Note that you cannot change the wrist configuration when using the joystick.

- Write debugging routines to display variables.

This appendix lists the error messages that RCS displays to indicate error conditions. The error messages include RCS, 8Ø87, and disk and tape error messages. RCS error messages consist of explanatory text. 8Ø87 error messages consist of a display of the 8Ø87 register contents. Disk and tape error messages consist of an error code described in the Rimfire manual.

A.1 RCS ERROR MESSAGES

RCS error messages describe fatal and nonfatal errors. Fatal errors are the result of problems in allocating memory space for an application. A fatal error requires you to execute at least a RESTORE and a D>M, rather than simply reload a single block, to recover from the error. RESTORE and D>M return the system to the "clean" state that existed before you encountered the error. The fatal error messages include BUFFER OVERFLOW, DICTIONARY FULL, FILE FULL, FILE SPACE FULL, SYSDICT FILE FULL, SYSDICT SPACE FULL, and vocab redefined larger. The remaining error messages indicate nonfatal errors.

The following list describes the RCS error messages.

~var only
    RCS detected something other than a SMACRO variable following ==>. (Nonfatal)

[name] ?
    The specified name is undefined. (Nonfatal)

6 msec time-out interrupt
    An application program attempted to access an I/O port or a RAM address that is not physically present in the system. (Nonfatal)

ALREADY ASSIGNED HERE
    RCS found more than one occurrence of the same variable in a list owner (LIST-O). (Nonfatal)

base<>1Ø
    RCS detected that you were trying to load a block with DBG-ON without first setting BASE to decimal. The block is not loaded. (Nonfatal)

buffer length mismatch
    RCS detected that two buffers specified for a COMM transfer have different lengths. (Nonfatal)

[name] BUFFER OVERFLOW

The current variable owner does not have room for the name to be defined. BUFFER OVERFLOW is the most common fatal error. This error results from the user specifying an insufficient number of bytes during the variable owner declaration. The error actually occurs when the SMACRO compiler assigns the members of that variable owner to the allocated space, which is too small to hold the members. (Fatal)

comm table full

RCS cannot accept more than 6Ø COMM transfer definitions. (Nonfatal)

DICTIONARY FULL

The FORTH dictionary space is full. This condition results from defining too many FORTH words or encountering too many errors while compiling SMACRO routines. If the condition results from defining too many FORTH words, you may be able to circumvent the condition by reloading the base system with LOCATE-OFF. (RCS loads RSL with LOCATE-OFF automatically.)

RCS compiles SMACRO routines in FORTH space before moving them to SMACRO space. If an error occurs during the compilation, RCS does not move the part of the SMACRO routine already compiled to SMACRO space, thus using up FORTH space. (Fatal)

[name] F-EXISTS

The specified name is a previously defined FORTH word. This error message is a warning only; RCS defines the SMACRO word. (Nonfatal)

FILE FULL

The current SMACRO file is full. This error can occur when you load the RSL code. To correct this condition, delete some of the RSL code for that file or increase the size of the file.

Deleting some RSL code and reloading the current set (enter <u>99Ø LOAD</u> for RSL) may solve the problem if the file does not use garbage collection (such as the POSE-FILE or the LOCATION-FILE).

If you choose to increase the file size, you must use RESTORE and D>M to return the system to a "clean" state before the file was declared on any level. Then declare the file with the new size and reload the system. Examine block 4 on each level to determine if the level declares the file. (Fatal)

FILE SPACE FULL

The SMACRO file being declared is too large for the remaining space in the common memory area allocated for user files. You can check the current size of the file space by entering <u>F-MEM</u>. To correct this condition, you must redefine the values of the fpage, ftop, and fbottom variables in block 14Ø2, and reload the base system. (Fatal)

floating point only

RCS detected something other than a floating-point number in the input for the #.READ word. (Nonfatal)

ILLEGAL

RCS detected an illegal value following <=. (Nonfatal)

illegal type

RCS detected something other than a string, array, byte variable, integer variable, floating-point variable, or a segment variable following ==>. (Nonfatal)

invalid preserve#

The preserve (or preserve-file) number is outside the range 1 through 9. (Nonfatal)

invalid image#

The D>M image number is outside the range 1 through 5. (Nonfatal)

[name] Pre-locate

The indicated name cannot be found using LOC. (Nonfatal)

record# = Ø

The variable record# was Ø when the program tried to manipulate a file. (Nonfatal)

remove record not on list

The SMACRO file-manipulation word, remove, could not find the specified record number on the current list. This error indicates that remove did not remove any records. (Nonfatal)

STACK EMPTY

Program operation caused the FORTH stack to underflow. Syntax errors in SMACRO statements may cause this error. (Nonfatal)

SYSDICT FILE FULL

RCS detected an attempt to add a word to a vocabulary that is already full. Enter V-SIZE to determine the size of the current vocabulary. To increase the size of a vocabulary, you must first use RESTORE and D>M to return the system to a "clean" state where that vocabulary is not defined. Then define the larger vocabulary. (Fatal)

SYSDICT SPACE FULL

The vocabulary being declared is too large for the remaining space in the common memory area allocated for the system dictionary. You can check the current size of the system dictionary by entering F-MEM. To correct this condition, you must redefine the values of the dpage, dtop, and dbottom variables in block 14Ø2, and reload the base system. (Fatal)

u-STACK-underflow

A stack used in compiling SMACRO routines has underflowed. This message indicates a syntax error, such as if you spelled "endif" as "end-if". (Nonfatal)

vocab already exists

RCS detected an attempt to define more than one vocabulary with the same name. Each vocabulary name must be unique throughout the entire system. (Nonfatal) vocab redefined larger

RCS detected an attempt to increase the size of a vocabulary when you redefined it. (Fatal)

var-owner only

RCS detected something other than a variable owner in a COMM transfer definition. (Nonfatal)

A.2 8Ø87 ERROR MESSAGES

If the 8Ø87 detects an error, it aborts the current operation and displays the control, status, and tag registers (in binary). It also displays the instruction address (in either the FORTH or SMACRO segment), the opcode, and the operand address (in either the FORTH or SMACRO segment). You can enter Ø.Ø Ø.Ø F/ for an example of this display.

A.3 RIMFIRE DISK AND TAPE ERROR MESSAGES

Disk and tape errors have the following format:

```
tape error [error code] block [block#] [word]
```

The error code gives the two Rimfire status bytes. The lower 6 bits of the high order byte give the error code. See the Rimfire manual listed under "Disk and Tape Controller Board" in the bibliography for a description of the error codes. The block# gives the block where the error occurred, and word is the error encountered.

This appendix describes the RCS and RSL user words. The word descriptions are grouped according to function and labelled with headings (See the Appendix B portion of the table of contents at the beginning of this manual for a list of the headings.) Within each group, the words are listed in ASCII order. The descriptions include the required syntax showing the word and its parameters in the order in which you should enter them.

Table B-1 lists the abbreviations and Table B-2 lists the phrases used to describe the syntax of the words. Table B-3 helps you find the description of a specific word in this appendix. This table lists all RCS and RSL user words in ASCII order. The table consists of the word followed by its group heading, the page number on which it is found in this appendix, and the page number where it is discussed in the manual. The bottom of each page of the table lists the ASCII character sequence to help you locate specific words. Because of the length of this table, horizontal lines separate every five words.

Table B-1. List of Abbreviations Used in the Syntax Descriptions

Abbreviation	Meaning
var	Any variable, including subscripted arrays.
ivar	An integer or byte variable.
iexp	An integer expression, composed of integer variables and operators. Note that integer expressions do not contain parentheses.
fvar	A scalar floating-point variable, including subscripted floating-point arrays.
fexp	A scalar floating-point expression.
vvar	A vector variable.
vexp	A vector expression.
qvar	A quaternion variable.
qexp	A quaternion expression.
pvar	A pose variable.
pexp	A pose expression.
mvar	A matrix variable.
mexp	A matrix expression.

Table B-2. List of Phrases Used in the Syntax Descriptions.

Phrase	Meaning
local template	The variable owner that serves as a template for records in the file. The members define the fields in the record.
current file	The last file made current by executing its name. All file operations refer to this file.
current record	The record in the current file pointed to by the variable record#. If record# is Ø, no current record exists.
current list	The linked list in the current file starting at the current record. If record# is Ø, no current list exists.
free list	The linked list of available records in the current file. The variable avail-record points to the free l.

Table B-3. Complete List of RCS and RSL User Words in ASCII Order.

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
!	FORTH words	B-46	6-5
"	Miscellaneous RCS words	B-35	6-9
""	SMACRO operator words	B-59	
#.READ	SMACRO operator words	B-60	10-16
#LLIST	Editing words	B-29	
#READ	SMACRO operator words	B-60	
$DEF	RSL operating system words	B-94	6-5
%	Miscellaneous RCS words	B-35	6-14
%%	Miscellaneous RCS words	B-35	5-12
''	Miscellaneous RCS words	B-35	6-9
(	Miscellaneous RCS words	B-36	6-8
(( ))	SMACRO operator words	B-60	
()	SMACRO operator words	B-60	
(*)	SMACRO operator words	B-60	7-9
(*/)	SMACRO operator words	B-61	
(+)	SMACRO operator words	B-61	7-9
(-)	SMACRO operator words	B-61	7-9
(/)	SMACRO operator words	B-61	7-9
(ABS)	SMACRO operator words	B-61	
(AND)	SMACRO Boolean-operator words	B-74	7-11
(COM)	SMACRO operator words	B-61	
(EQ)	SMACRO Boolean-operator words	B-74	6-17
(GE)	SMACRO Boolean-operator words	B-74	7-11
(GT)	SMACRO Boolean-operator words	B-74	7-11
(LE)	SMACRO Boolean-operator words	B-74	7-11
(LT)	SMACRO Boolean-operator words	B-74	7-11
(MAX)	SMACRO operator words	B-61	
(MIN)	SMACRO operator words	B-62	
(MOD)	SMACRO operator words	B-62	
(NE)	SMACRO Boolean-operator words	B-75	7-11
*	FORTH words	B-46	10-46
+	FORTH words	B-46	
-	FORTH words	B-46	
---	Language words	B-98	9-4
-1#	Predefined SMACRO variable words	B-57	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _   { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
-arr-	Language words	B-98	9-4
-imtb-	Language words	B-98	9-4
-loc-	Language words	B-99	9-4
-mtb-	Language words	B-99	9-4
-mtb-end-	Language words	B-100	9-4
-obj-	Language words	B-100	9-4
-path-	Language words	B-100	9-5
-pose-	Language words	B-101	9-3
-ppt-	Language words	B-101	9-5
-rr-	Language words	B-101	10-14
.	FORTH words	B-46	7-17
.( ).	SMACRO operator words	B-62	7-9
.*.	SMACRO operator words	B-62	7-9
.*2**.	SMACRO operator words	B-62	
.+.	SMACRO operator words	B-62	7-9
.-.	SMACRO operator words	B-62	7-9
./.	SMACRO operator words	B-63	7-9
.=>.	SMACRO operator words	B-63	6-17
.ABS.	SMACRO operator words	B-63	
.ATAN,SIN,COS.	SMACRO operator words	B-63	
.ATAN.	SMACRO operator words	B-63	
.AXIS,ANGLE>Q.	SMACRO quaternion-operator words	B-70	
.AXIS,SIN,COS>Q.	SMACRO quaternion-operator words	B-70	
.CROSS.	SMACRO vector-operator words	B-68	
.D	FORTH words	B-47	
.DOT.	SMACRO vector-operator words	B-68	
.EQ.	SMACRO Boolean-operator words	B-75	7-11
.EQZ.	SMACRO Boolean-operator words	B-75	7-11
.Esc	Miscellaneous RCS words	B-36	
.FRAC.	SMACRO operator words	B-63	
.GE.	SMACRO Boolean-operator words	B-75	7-11
.GEZ.	SMACRO Boolean-operator words	B-75	7-11
.GT.	SMACRO Boolean-operator words	B-75	7-11
.GTZ.	SMACRO Boolean-operator words	B-75	7-11
.INT.	SMACRO operator words	B-63	

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
.LE.	SMACRO Boolean-operator words	B-75	7-11
.LEZ.	SMACRO Boolean-operator words	B-76	7-11
.LT.	SMACRO Boolean-operator words	B-76	7-11
.LTZ.	SMACRO Boolean-operator words	B-76	7-11
.M=>.	SMACRO matrix-operator words	B-71	7-10
.M>Q.	SMACRO matrix-operator words	B-71	
.MAX.	SMACRO operator words	B-64	
.MIN.	SMACRO operator words	B-64	
.MINUS.	SMACRO operator words	B-64	
.MOD+-18Ø.	SMACRO operator words	B-64	
.NE.	SMACRO Boolean-operator	B-76	7-11
.NEZ.	SMACRO Boolean-operator words	B-76	7-11
.P=>.	SMACRO pose-operator words	B-72	7-10
.PINV.	SMACRO pose-operator words	B-72	
.PP*.	SMACRO pose-operator words	B-72	
.PQ*.	SMACRO pose-operator words	B-72	
.PV+.	SMACRO pose-operator words	B-72	
.Q=>.	SMACRO quaternion-operator words	B-70	7-10
.Q>AXIS,ANGLE,SIN,COS.	SMACRO quaternion-operator words	B-70	
.Q>AXIS,SIN,COS.	SMACRO quaternion-operator words	B-71	
.Q>M.	SMACRO matrix-operator words	B-72	
.QINV.	SMACRO quaternion-operator words	B-71	
.QP*.	SMACRO pose-operator words	B-73	
.QQ*.	SMACRO quaternion-operator words	B-71	
.QV*.	SMACRO quaternion-operator words	B-71	
.ROT,SIN,COS.	SMACRO pose-operator words	B-73	
.ROT.	SMACRO pose-operator words	B-73	
.SIGN.	SMACRO operator words	B-64	
.SIN,COS.	SMACRO operator words	B-64	
.SQ.	SMACRO operator words	B-64	
.SQRT.	SMACRO operator words	B-65	
.TRN.	SMACRO pose-operator words	B-73	
.UNIT.	SMACRO vector-operator words	B-69	
.V*S.	SMACRO vector-operator words	B-69	
.V+.	SMACRO vector-operator words	B-69	7-16

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _   { | } ~

		APPENDIX	MANUAL
WORD	GROUP HEADING	PAGE #	PAGE #
.V-.	SMACRO vector-operator words	B-69	
.V/S.	SMACRO vector-operator words	B-69	
.V=>.	SMACRO vector-operator words	B-69	7-10
.V>Q.	SMACRO quaternion-operator words	B-71	
.VINV.	SMACRO vector-operator words	B-69	
.VMAG.	SMACRO vector-operator words	B-70	
.VMINUS.	SMACRO vector-operator words	B-70	
.VP+.	SMACRO pose-operator words	B-74	
.^	Miscellaneous RCS words	B-36	
/	FORTH words	B-47	
Ø-?	SMACRO Boolean-operator words	B-76	7-15
Ø-SMACRO	Miscellaneous SMACRO words	B-90	
Ø.#	Predefined SMACRO variable words	B-58	
ØED	Editing words	B-30	
Øed	Editing words	B-30	
ØLIST	Editing words	B-30	6-23
ØLLIST	Editing words	B-30	6-23
ØLOAD	Miscellaneous RCS words	B-36	5-14
1#	Predefined SMACRO variable words	B-57	
1-?	SMACRO Boolean-operator words	B-76	7-15
1.#	Predefined SMACRO variable words	B-58	6-17
1Ø#	Predefined SMACRO variable words	B-57	
1Ø-SMACRO	Miscellaneous SMACRO words	B-90	
11#	Predefined SMACRO variable words	B-57	
12#	Predefined SMACRO variable words	B-57	
12-SMACRO	Miscellaneous SMACRO words	B-90	
13#	Predefined SMACRO variable words	B-57	
14#	Predefined SMACRO variable words	B-57	
15#	Predefined SMACRO variable words	B-57	
16#	Predefined SMACRO variable words	B-57	
17#	Predefined SMACRO variable words	B-57	
18#	Predefined SMACRO variable words	B-57	
19#	Predefined SMACRO variable words	B-57	
1:a	SMACRO declaration words	B-54	7-3
1:fa	SMACRO declaration words	B-54	7-3

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
2!	FORTH words	B-47	
2#	Predefined SMACRO variable words	B-57	
2-SMACRO	Miscellaneous SMACRO words	B-90	
2.#	Predefined SMACRO variable words	B-58	
2:a	SMACRO declaration words	B-54	7-3
2:fa	SMACRO declaration words	B-55	7-3
2@	FORTH words	B-47	
2IN	FORTH words	B-47	
2OUT	FORTH words	B-47	
2pi	SMACRO operator words	B-65	
3#	Predefined SMACRO variable words	B-57	
32767#	Predefined SMACRO variable words	B-57	
4#	Predefined SMACRO variable words	B-57	
4-SMACRO	Miscellaneous SMACRO words	B-90	
5#	Predefined SMACRO variable words	B-57	
6#	Predefined SMACRO variable words	B-57	
6-SMACRO	Miscellaneous SMACRO words	B-90	
7#	Predefined SMACRO variable words	B-57	
8#	Predefined SMACRO variable words	B-57	
8-SMACRO	Miscellaneous SMACRO words	B-90	
87INIT	Miscellaneous RCS words	B-36	6-15
9#	Predefined SMACRO variable words	B-57	
:	FORTH words	B-47	
:C~EXEC	FORTH words	B-48	
:L	Miscellaneous RCS words	B-36	6-4
:l	Miscellaneous RCS words	B-36	7-18
:L~EXEC	FORTH words	B-48	
:OK	Miscellaneous RCS words	B-37	6-2
:R	Miscellaneous RCS words	B-37	6-4
:r	Miscellaneous RCS words	B-37	7-18
:S	Miscellaneous RCS words	B-37	6-9
:S~EXEC	FORTH words	B-48	
:s	Miscellaneous RCS words	B-37	7-18
;	FORTH words	B-48	
<=	SMACRO declaration words	B-55	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
<PAGE	Miscellaneous RCS words	B-37	
==>	Miscellaneous RCS words	B-37	6-9
=>	SMACRO operator words	B-65	7-10
=>SEG	SMACRO operator words	B-65	
>PAGE	Miscellaneous RCS words	B-37	
?	FORTH words	B-48	6-24
?ARR	RSL operating system words	B-94	
?LOC	RSL operating system words	B-94	
?MOVE-TO	RSL operating system words	B-95	11-3
?MTB	RSL operating system words	B-95	
?OBJ	RSL operating system words	B-95	
?POSE	RSL operating system words	B-95	
@	FORTH words	B-48	
a#	SMACRO operator words	B-65	
a-sysmax	JOINT parameters	B-105	
AA>Q	Miscellaneous RCS words	B-38	
abort	SMACRO statement words	B-84	
abort-f	Control-level words	B-108	
ABT	RSL operating system words	B-95	
add-list	SMACRO file-operation words	B-80	13-4
add-record	SMACRO file-operation words	B-80	7-13
ADD-TO-TAPE	Tape words	B-28	6-29
approach-pickup	Language words	B-101	9-5
approach-release	Language words	B-102	9-5
arr	Language words	B-102	6-9
ass/rec	SMACRO file-operation header variable words	B-78	
AUTO?	MBOOT/CUSTOM words	B-19	6-15
avail-record	SMACRO file-operation header variable words	B-78	
B	Editing words	B-30	6-23
B*	SMACRO operator words	B-65	7-9
B+	SMACRO operator words	B-65	7-9
B-	SMACRO operator words	B-66	7-9
b-fill	SMACRO statement words	B-84	
B-to-stack	SMACRO operator words	B-66	7-15
B.	FORTH words	B-49	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _   { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
B/	SMACRO operator words	B-66	7-9
B=>	SMACRO operator words	B-66	7-10
BACKGROUND	FORTH words	B-49	
BACKUP	Tape words	B-28	6-29
BAD-TAPE	Tape words	B-28	6-29
BASE	Miscellaneous RCS words	B-38	
Basic	MBOOT-CUSTOM words	B-19	5-11
BI	Miscellaneous RCS words	B-38	
blank	SMACRO statement words	B-84	
BLOCKS	Editing words	B-30	6-23
Board-sem-initialize	Remote-slave words	B-33	13-15
BOOT-BASE-SYSTEM	MBOOT/CUSTOM words	B-19	D-1
BOOT-SYSTEM	MBOOT/CUSTOM words	B-20	5-14
buffer-ready-f	COMM words	B-92	8-3
BUILD	FORTH words	B-49	
bv	SMACRO declaration words	B-55	7-3
bytes	SMACRO declaration words	B-55	7-4
bytes/record	SMACRO file-operation header variable words	B-79	
cart	Language words	B-102	9-7
case	SMACRO statement words	B-85	7-17
case:	SMACRO statement words	B-85	7-17
CBOOT	MBOOT/CUSTOM words	B-20	5-10
CLEAR	Editing words	B-30	6-23
clear-list	SMACRO file-operation words	B-80	13-4
CLR-BIT	FORTH words	B-49	
COMM	COMM words	B-92	4-3
COMM-PROCESS	COMM words	B-92	
control-cycle-#-clks	COMM words	B-92	6-15
cos{angle}	Predefined SMACRO variable words	B-58	
CPRINT	Printing words	B-26	6-28
CR	Miscellaneous RCS words	B-38	6-15
creep-delta	JOINT parameters	B-106	10-49
creep-vel	JOINT parameters	B-106	10-49
CUSTOM	MBOOT/CUSTOM words	B-20	4-3
CUSTOM?	MBOOT/CUSTOM words	B-20	D-1

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
cycle-count	Predefined SMACRO variable words	B-58	
cycle-time	Control-level words	B-108	
D>M	MBOOT/CUSTOM words	B-20	4-5
Datamedia	Terminal words	B-24	5-12
Datamedia-terminal	Editing words	B-31	5-12
DBG-OFF	Miscellaneous RCS words	B-38	
DBG-ON	Miscellaneous RCS words	B-38	7-1
dbottom	MBOOT/CUSTOM words	B-20	A-3
DC	Terminal words	B-24	
Dec-writer	Printing words	B-26	5-13
DECIMAL	Miscellaneous RCS words	B-39	6-15
default:	SMACRO statement words	B-85	7-17
default-state:	SMACRO statement words	B-85	3-3
DEFINITIONS	Miscellaneous SMACRO words	B-90	6-4
deg	Miscellaneous RCS words	B-39	
depart-pickup	Language words	B-102	9-5
depart-release	Language words	B-102	9-5
Disk-fix	MBOOT/CUSTOM words	B-21	
Display	Control-level words	B-109	6-8
DM?	MBOOT/CUSTOM words	B-21	6-13
do	SMACRO statement words	B-85	7-10
dpage	MBOOT/CUSTOM words	B-21	A-3
dtop	MBOOT/CUSTOM words	B-21	A-3
DUMP	FORTH words	B-49	
ED	Editing words	B-31	6-16
ed	Editing words	B-31	
Ed-extend	Editing words	B-31	
EDIT	Editing words	B-31	
ELOAD	Remote-slave words	B-34	6-27
else	SMACRO statement words	B-85	6-17
end-case	SMACRO statement words	B-85	7-17
end-do	SMACRO statement words	B-85	7-10
end-repeat	SMACRO statement words	B-85	7-10
end-routine	SMACRO declaration words	B-55	4-4
end-state-table	SMACRO statement words	B-85	3-3

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
endif	SMACRO statement words	B-86	6-17
enter-interrupt	SMACRO statement words	B-86	4-4
EQ_B	SMACRO Boolean-operator words	B-77	7-11
erase	SMACRO statement words	B-86	6-1
ERASE-COMM-TABLE	COMM words	B-92	8-3
error-list	Control-level words	B-109	
Esc	Miscellaneous RCS words	B-39	
exit-interrupt	SMACRO statement words	B-86	4-4
F*	FORTH words	B-49	
F+	FORTH words	B-50	
F-	FORTH words	B-50	
F-ABS	FORTH words	B-50	
F-MEM	Miscellaneous RCS words	B-39	9-8
f-segment	SMACRO file-operation header variable words	B-79	
f-template	SMACRO file-operation header variable words	B-79	
F.	FORTH words	B-50	
F/	FORTH words	B-50	A-4
F>I	SMACRO operator words	B-66	
false	Predefined SMACRO variable words	B-58	13-11
Fastbaud	Terminal words	B-24	
fbottom	MBOOT/CUSTOM words	B-21	A-2
FILE	SMACRO declaration words	B-55	7-12
file-start	SMACRO file-operation header variable words	B-79	
File-var	SMACRO file-operation variable words	B-78	
FIND	Editing words	B-31	6-23
FIND-R	Editing words	B-32	6-23
first-record	SMACRO file-operation header variable words	B-79	
FLOAT	FORTH words	B-50	
fpage	MBOOT/CUSTOM words	B-22	A-2
from-stack	SMACRO operator words	B-66	7-15
FROM-TAPE	Tape words	B-29	5-11
ftop	MBOOT/CUSTOM words	B-22	A-2
Function-keys	Terminal words	B-24	
fv	SMACRO declaration words	B-56	7-3
GET-FROM-DISK	Miscellaneous RCS words	B-39	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

		APPENDIX	MANUAL
WORD	GROUP HEADING	PAGE #	PAGE #
GE_B	SMACRO Boolean-operator words	B-77	7-11
GO	RSL operating system words	B-96	6-2
goal	Language words	B-103	9-7
goto	Language words	B-103	9-5
GT_B	SMACRO Boolean-operator words	B-77	7-11
H.	FORTH words	B-50	
HALT	RSL operating system words	B-96	6-11
HEX	Miscellaneous RCS words	B-39	6-15
I>F	SMACRO operator words	B-66	
identity-pose	Predefined SMACRO variable words	B-58	
if	SMACRO statement words	B-86	6-17
Image-writer	Printing words	B-26	5-13
in-port	SMACRO operator words	B-67	7-15
inc-command-#-in	Control-level words	B-109	6-9
index-on	SMACRO operator words	B-67	
init-cm	MBOOT/CUSTOM words	B-22	5-13
INPUT	FORTH words	B-51	
INV-TOOL-MTB	RSL parameters	B-108	
IRR	FORTH words	B-51	
iv	SMACRO declaration words	B-56	7-3
joint	Language words	B-103	9-7
joint-traj-delay	PRIM parameters	B-107	
Joint/comm	RSL operating system words	B-96	6-15
joy-max-v	PRIM parameters	B-107	10-46
joy-max-w	PRIM parameters	B-108	10-46
JS	Control-level words	B-109	13-2
L	Editing words	B-32	6-23
LE_B	SMACRO Boolean-operator words	B-77	7-11
LIST	Editing words	B-32	6-8
List-directory	Printing words	B-26	6-29
List-display	Control-level words	B-109	
LIST-O	Miscellaneous SMACRO words	B-91	7-4
List-programs	Printing words	B-26	6-29
lj-limit	JOINT parameters	B-106	
LOAD	Miscellaneous RCS words	B-40	5-13

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

		APPENDIX	MANUAL
WORD	GROUP HEADING	PAGE #	PAGE #
load	Miscellaneous RCS words	B-40	6-7
LOC	Miscellaneous RCS words	B-40	6-4
loc	Language words	B-103	6-9
LOCATE-OFF	FORTH words	B-51	
LOCATE-ON	FORTH words	B-51	
LT_B	SMACRO Boolean-operator words	B-77	7-11
m	Miscellaneous SMACRO words	B-91	13-14
MAP	MBOOT/CUSTOM words	B-22	6-7
mask-with	SMACRO operator words	B-67	
Master	Remote-slave words	B-34	
matches	SMACRO Boolean file-operation words	B-83	7-11
matches-fields	SMACRO Boolean file-operation words	B-83	7-11
matches-fields-r	SMACRO Boolean file-operation words	B-83	7-11
matches-r	SMACRO Boolean file-operation words	B-84	7-11
max-cycle-time	Control-level words	B-109	
max/rec	SMACRO file-operation header variable words	B-79	
MBOOT	MBOOT/CUSTOM words	B-22	4-3
mem	SMACRO declaration words	B-56	7-6
MEM>DISK	MBOOT/CUSTOM words	B-22	5-12
min-cycle-time	Control-level words	B-110	
MONITOR	Remote-slave words	B-34	
MOVE	FORTH words	B-51	
move	SMACRO statement words	B-87	
move-to	Language words	B-103	3-7
MOVE-TO	TASK command words	B-104	6-8
MS	Miscellaneous RCS words	B-40	
N	Editing words	B-32	
next-point-scale-threshold	JOINT parameters	B-106	
next-record	SMACRO file-operation variable words	B-78	
NE_B	SMACRO Boolean-operator words	B-77	7-11
not-ready	Predefined SMACRO variable words	B-58	8-3
NQUIT	Miscellaneous RCS words	B-40	
nul	Language words	B-103	6-8
nz	Miscellaneous RCS words	B-40	
OFFSET	Miscellaneous RCS words	B-41	5-14

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _   { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
OLOAD	Miscellaneous RCS words	B-41	
Opaq	Terminal words	B-25	
out-port	SMACRO operator words	B-67	7-15
OUTPUT	FORTH words	B-51	
overflow-cycle	Control-level words	B-110	
PAGE	Miscellaneous RCS words	B-41	13-4
page	Miscellaneous SMACRO words	B-91	
PAGE-MOVE	Miscellaneous RCS words	B-41	
PAGE-SEG-ERASE	Miscellaneous RCS words	B-41	
Paper-tiger	Printing words	B-27	5-13
PAUSE	TASK command words	B-104	6-8
PDEF	Control-level words	B-110	6-3
PDUMP	Miscellaneous RCS words	B-41	
PI	FORTH words	B-51	
pi	SMACRO operator words	B-67	
pi/2	SMACRO operator words	B-67	
PIC	FORTH words	B-52	
PMAP	MBOOT/CUSTOM words	B-23	
PMS	Miscellaneous RCS words	B-42	13-2
POFF	RSL operating system words	B-96	
Poff	Terminal words	B-25	
PON	RSL operating system words	B-96	
Pon	Terminal words	B-25	
PRESERVE	MBOOT/CUSTOM words	B-23	5-13
PRESERVE-FILE	MBOOT/CUSTOM words	B-23	5-13
Prim	RSL operating system words	B-97	
Print	Printing words	B-27	6-28
print-f	Control-level words	B-110	6-26
PRINT-TIME	Miscellaneous RCS words	B-42	13-5
Printing	Printing words	B-27	
Pset-tab	Printing words	B-28	
QMS	Miscellaneous RCS words	B-42	
QR	Editing words	B-32	6-24
QS	Miscellaneous RCS words	B-42	
Query-replace	Editing words	B-32	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _   { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
r-tool	Language words	B-104	9-6
R-WITH	Editing words	B-33	6-24
rad	Miscellaneous RCS words	B-42	
RCS	MBOOT/CUSTOM words	B-23	5-13
READ"	SMACRO operator words	B-67	
ready	Predefined SMACRO variable words	B-58	8-3
rec	SMACRO declaration words	B-56	7-12
record#	SMACRO file-operation variable words	B-78	6-17
RECORD-POSE	RSL operating system words	B-97	9-7
RED	Editing words	B-33	13-12
red	Editing words	B-33	6-25
ref-blk	Miscellaneous RCS words	B-42	6-8
REMOTE	Remote-slave words	B-34	6-3
REMOTE-SLAVE	Remote-slave words	B-34	5-13
remove	SMACRO file-operation words	B-80	13-4
repeat	SMACRO statement words	B-87	7-10
reset	Predefined SMACRO variable words	B-58	
RESTART	TASK command words	B-105	6-8
RESTART-COMM-TIMER	COMM words	B-93	6-15
RESTORE	MBOOT/CUSTOM words	B-23	6-11
Restore	Printing words	B-28	
RESTORE-FILE	MBOOT/CUSTOM words	B-23	6-11
retrieve	SMACRO file-operation words	B-81	13-4
retrieve-from-field	SMACRO file-operation words	B-81	
retrieve-from-fields	SMACRO file-operation words	B-81	6-17
retrieve-l&r	SMACRO file-operation words	B-81	13-4
retrieve-link	SMACRO file-operation words	B-81	
retrieve-source	SMACRO file-operation words	B-81	
rot-angle	Predefined SMACRO variable words	B-58	
rot-axis	Predefined SMACRO variable words	B-59	
ROUND	FORTH words	B-52	
routine	SMACRO declaration words	B-56	4-4
S-EQ	SMACRO Boolean-operator words	B-77	
S-VAR-O	SMACRO declaration words	B-56	7-4
S=>	SMACRO statement words	B-87	6-17

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
SAVE-ON-DISK	Miscellaneous RCS words	B-43	
sd	Miscellaneous RCS words	B-43	
SDEF	Miscellaneous SMACRO words	B-92	6-3
SEG-DUMP	Miscellaneous RCS words	B-43	
SEG-MOVE	Miscellaneous RCS words	B-43	
SEG=>	SMACRO operator words	B-68	
segv	SMACRO declaration words	B-57	7-3
set	Predefined SMACRO variable words	B-59	
SET-BIT	FORTH words	B-52	
set-bit-in	SMACRO statement words	B-87	7-15
seqv	SMACRO declaration words	B-57	7-3
SHOW-TABLE	COMM words	B-93	
sin{angle}	Predefined SMACRO variable words	B-59	
Slave	Remote-slave words	B-34	5-13
Small	Printing words	B-28	
source-blk	SMACRO file-operation variable words	B-78	
SQRT	FORTH words	B-52	
state	SMACRO statement words	B-87	
state-table	SMACRO statement words	B-88	2-5
store	SMACRO file-operation words	B-82	13-4
store-l&r	SMACRO file-operation words	B-82	13-4
store-link	SMACRO file-operation words	B-82	
store-source	SMACRO file-operation words	B-82	13-4
store-to-field	SMACRO file-operation words	B-82	
store-to-fields	SMACRO file-operation words	B-82	
strv	SMACRO declaration words	B-57	7-3
SYNC	Remote-slave words	B-34	
t-base	Language words	B-104	
t-tool	Language words	B-104	9-6
TAB	Miscellaneous RCS words	B-43	
TALK\	Remote-slave words	B-35	
TAPE	Tape words	B-29	5-11
Task/path	RSL operating system words	B-97	6-11
TDEF	Control-level words	B-110	6-3
TDOC	Tape words	B-29	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

		APPENDIX	MANUAL
WORD	GROUP HEADING	PAGE #	PAGE #
Televideo	Terminal words	B-25	5-12
Televideo-terminal	Editing words	B-33	5-12
TGOTO	Tape words	B-29	5-11
then	SMACRO statement words	B-88	6-17
THRU	Miscellaneous RCS words	B-43	
Thru-tv	Terminal words	B-25	6-29
TIME-OUT-INIT	Miscellaneous RCS words	B-44	6-15
TIME-P	Miscellaneous RCS words	B-44	13-2
TIMER-READ	Miscellaneous RCS words	B-44	
TIMER-START	Miscellaneous RCS words	B-44	
TO-DESTINATION	COMM words	B-93	8-3
to-stack	SMACRO operator words	B-68	7-15
tool	Language words	B-104	12-20
TOOL-MTB	RSL parameters	B-108	
tp-cycles	Path parameters	B-107	
Tran	Terminal words	B-25	
TRANSFER	TASK command words	B-105	6-7
TRANSFER-FROM	COMM words	B-93	8-3
TREWIND	Tape words	B-29	5-11
true	Predefined SMACRO variable words	B-59	6-26
TT	RSL operating system words	B-97	13-4
TTT	RSL operating system words	B-97	13-9
TUNLOAD	Tape words	B-29	
TYPE-POSE	RSL operating system words	B-97	9-7
uj-limit	JOINT parameters	B-106	
until	SMACRO statement words	B-88	7-10
V-SIZE	Miscellaneous RCS words	B-44	6-3
v-sysmax	JOINT parameters	B-107	
VAR-O	SMACRO declaration words	B-57	7-4
VOCABULARY	SMACRO declaration words	B-57	
VT1ØØ	Terminal words	B-26	5-12
VT1ØØ-terminal	Editing words	B-33	5-12
w-fill	SMACRO statement words	B-88	
WAIT-FOR-NEXT-COMM-CYCLE	COMM words	B-93	
WAIT-TASK-DONE	Control-level words	B-110	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _ { | } ~

WORD	GROUP HEADING	APPENDIX PAGE #	MANUAL PAGE #
WFIX	MBOOT/CUSTOM words	B-24	
while	SMACRO statement words	B-89	7-10
WREAD	SMACRO operator words	B-68	7-18
WUNLOAD	MBOOT/CUSTOM words	B-24	6-30
XX	Miscellaneous SMACRO words	B-89	
X Y Z	Predefined SMACRO variable words	B-59	
zero-bit-in	SMACRO statement words	B-89	7-15
\L	Miscellaneous RCS words	B-45	7-18
\l	Miscellaneous RCS words	B-45	7-18
\N	Miscellaneous RCS words	B-45	7-18
\n	Miscellaneous RCS words	B-45	7-18
\R	Miscellaneous RCS words	B-45	7-18
\r	Miscellaneous RCS words	B-45	7-18
\S	Miscellaneous RCS words	B-45	7-18
\s	Miscellaneous RCS words	B-45	7-18
\^	Miscellaneous RCS words	B-45	
^EQ^	SMACRO Boolean-operator words	B-78	7-15
^x	Remote-slave words	B-35	6-2
{ }	SMACRO operator words	B-68	7-10
~	Miscellaneous SMACRO words	B-89	
~!	FORTH words	B-52	13-4
~?	FORTH words	B-52	
~@	FORTH words	B-53	6-9
~C!	FORTH words	B-53	
~C@	FORTH words	B-53	
~DUMP	Miscellaneous RCS words	B-45	
~F	SMACRO statement words	B-89	6-25
~F!	FORTH words	B-53	
~F?	FORTH words	B-53	
~F@	FORTH words	B-53	
~FR?	FORTH words	B-53	
~INTERRUPT	Miscellaneous SMACRO words	B-92	4-4
~MEM	Miscellaneous RCS words	B-46	
~MOVE	Miscellaneous RCS words	B-46	
~PRINT	SMACRO statement words	B-89	6-25
~PRINT"	SMACRO statement words	B-89	6-25
~re-init-file	SMACRO file-operation words	B-83	13-4
~update-header	SMACRO file-operation words	B-83	

ASCII Sequence:

! " # $ % & ' ( ) * + , - . / Ø...9 : ; < = > ? @ Aa...Zz [\ ] ^ _   { | } ~

B.1 RCS WORDS

This section describes user words defined in RCS, including SMACRO and COMM. The RCS words include:

- Operating system words
- FORTH words
- SMACRO words
- COMM words

All block references are absolute, unless otherwise indicated.

Operating System Words

The operating system words include:

- MBOOT/CUSTOM words
- Terminal words
- Printing words
- Tape words
- Editing words
- Remote-slave words
- Miscellaneous words

MBOOT/CUSTOM words

AUTO?

    [d>m#] AUTO?

    Lists the auto-load block for the specified D>M# on the current board.

Basic

    Basic

    Loads the second half of the basic FORTH system. Use this word after MBOOT and before other load words.

BOOT-BASE-SYSTEM

    BOOT-BASE-SYSTEM

    Loads absolute block 1490 to boot the base system (1 D>M) on each board.

BOOT-SYSTEM

BOOT-SYSTEM

Loads block 1460 to boot the slave boards in the system. You may edit blocks 1460 through 1469 to configure the boot process.

CBOOT

[board#] CBOOT

Boots the Winchester disk from the board that you specify using board#. Use CBOOT at power-up only.

CUSTOM

[custom#] CUSTOM

Loads a block (block 1370 + custom#) that loads a custom set of utilities on the current board for an application. You must set up and maintain this custom# block.

CUSTOM?

[custom#] CUSTOM?

Lists the custom block if the argument is present; lists the custom map block if the argument is not present.

D>M

[d>m#] D>M

Loads a specified D>M image from disk to memory and loads the auto-load block.

dbottom

dbottom

Specifies the bottom 20-bit address used by the system dictionary. The pointer value in this FORTH variable consists of a segment and an offset within the megabyte page specified by the dpage variable. You must set the value above 1FFFF because a processor board interprets any lower address as an on-board address instead of a common memory address. You set dbottom in block 1402. (See also dpage and dtop)

Disk-fix

Disk-fix

Loads the disk-patching utility.

DM?

[board#] DM?

Lists the D>M map block. If the argument is present, this word lists the D>M map for the specified board. If the argument is not present, this word lists the D>M map for the current board.

dpage

dpage

Specifies which megabyte page (using a pagecode) in common memory contains the system dictionary. You can use >PAGE to store a pagecode value in this FORTH variable. You set the dpage variable in block 14Ø2. (See also >PAGE, under "Miscellaneous RCS words", dtop, and dbottom)

dtop

dtop

Specifies the top 2Ø-bit address used by the system dictionary. The pointer value in this FORTH variable consists of a segment and an offset within the megabyte page specified by the dpage variable. You set the dtop variable in block 14Ø2. (See also dpage and dbottom)

fbottom

fbottom

Specifies the bottom 2Ø-bit address used by the SMACRO files. The pointer value in this FORTH variable consists of a segment and an offset within the megabyte page specified by the fpage variable. You must set the value above 1FFFF because a processor board interprets any lower address as an on-board address instead of a common memory address. You set the fbottom in block 14Ø2. (See also fpage and ftop) fpage

fpage

Specifies which megabyte page (using a pagecode) in common memory contains the SMACRO files. You can use >PAGE to store a pagecode value in this FORTH variable. You set the fpage variable in block 14Ø2. (See also >PAGE, under "Miscellaneous RCS words", ftop, and fbottom)

ftop

ftop

Specifies the top 2Ø-bit address used by the SMACRO files. The pointer value in this FORTH variable consists of a segment and an offset within the megabyte page specified by the fpage variable. You set the ftop variable in block 14Ø2. (See also fpage and fbottom)

init-cm

init-cm

Initializes the common memory space by filling it with zeros. Block 14Ø5 defines the area to be initialized. (When editing block 14Ø5, you should also edit block 14Ø2, which sets the common memory pointer variables)

MAP

MAP

Lists the first map block.

MBOOT

[board#] MBOOT

Loads the first half of the basic FORTH system by loading block 9 from 1ØØØ OFFSET. Use MBOOT at power-up and following a reset, and follow it with Basic to load the second half of the basic FORTH system, or D>M to boot a finished system.

MEM>DISK

[d>m#] MEM>DISK

Writes a D>M image of memory to disk and assigns a number to the image.

PMAP

PMAP

Lists the preserve map block.

PRESERVE

[preserve#] PRESERVE

Writes an image (identified by preserve#) of the current system dictionary to disk. (See also RESTORE)

PRESERVE-FILE

[preserve-file#] PRESERVE-FILE

Writes the current user files to disk. (See also RESTORE-FILE)

RCS

RCS

Loads RCS. Note that the system ignores any words typed after this word on the same line.

RESTORE

[preserve#] RESTORE

Restores the specified system dictionary image from disk. (See also PRESERVE)

RESTORE-FILE

[preserve-file#] RESTORE-FILE

Restores a specified set of user files from disk. (See also PRESERVE-FILE)

WFIX

[block#] WFIX

Reformats the portion of the disk containing the specified block. This word displays the range of blocks that would be affected by the reformatting and asks you to confirm that the reformatting is to take place. This reformatting process usually repairs blocks that generate disk errors.

WUNLOAD

WUNLOAD

Unloads the heads and powers down the drive motor of the Winchester disk drive. You must use this command before you turn off the system.

Terminal words

Datamedia

Datamedia

Loads the control words required to use a Datamedia Elite 1521A terminal as the system console. (See also EDIT under "Editing words")

DC

[row] [column] DC

Positions the cursor at the specified row and column.

Fastbaud

Fastbaud

Sets the communication rate of the board to 19,2ØØ baud. After issuing this word, you must also set the terminal communication rate to 19,2ØØ baud. For the TeleVideo 95Ø, Fastbaud sets the rate to 19,2ØØ automatically. For the VT1ØØ, you must set the baud rate manually. Fastbaud is inappropriate for the Datamedia because the Datamedia does not operate at 19,2ØØ baud.

Function-keys

Function-keys

Programs the function keys by loading block 14Ø5. You may edit block 14Ø5 to customize the operations of the function keys.

Opaq

Opaq

Stops sending output to the printer. This word cancels the action of Opaq. This word operates only when used with a TeleVideo 95Ø terminal. (See also Tran)

Poff

Poff

Disables the printer port on the terminal so that output sent to the terminal is not also sent to the printer. (See also Pon)

Pon

Pon

Enables the printer port on the terminal so that output sent to the terminal is also sent to the printer. (See also Poff)

Televideo

Televideo

Loads the control words required to use a TeleVideo 95Ø terminal as the system console. (See also EDIT under "Editing words")

Thru-tv

Thru-tv [words]

Executes Tran to send output to the printer, executes the words you enter at the keyboard, then executes Opaq to stop sending the output to the printer. This word operates only when used with a TeleVideo 95Ø terminal or a VT1ØØ terminal.

Tran

Tran

Sends all output to the system printer, including all control sequences that the terminal normally intercepts and interprets. This word operates only when used with a TeleVideo 95Ø terminal. (See also Opaq)

VT1ØØ

VT1ØØ

Loads the control words required to use a DEC VT1ØØ or compatible terminal as the system console. (See also EDIT under "Editing words")

Printing words

CPRINT

[start block#] [end block#] CPRINT

Prints the specified range of blocks at a density of 1Ø blocks per page.

Dec-writer

Dec-writer

Loads the control words required to use a Digital Equipment Corporation DECwriter III as the system printer. (See also Printing)

Image-writer

Image-writer

Loads the control words required to use an Apple Imagewriter as the system printer. (See also Printing)

List-directory

List-directory

Places RCS in the list-directory mode, which is similar to the list-programs mode except that list-directory mode prints the comments (lines 3 through 11) in the load blocks instead of listing the program blocks. These lines should provide a directory to the code contained in each block.

List-programs

List-programs

Places RCS in the list-programs mode. In this mode, RCS prints at least 1Ø blocks per page (more if the blocks contain blank lines). The actual number of blocks printed per page depends on the format of the 1Ø-block load block structure.

The best way to use this utility is to set up a block that lists an entire level as follows:

```
List-programs
Small
100 LOAD
200 LOAD
301 load 380 load
Restore
```

The word List-programs redefines "load" to be the listing word at the beginning of the block, and the word Restore puts back the FORTH definition at the end of the block. The word Small instructs the printer to use condensed type. 100 LOAD prints the blocks normally loaded by 100; this block is assumed to use "load" (and not "LOAD") to load 10-block segments.

RCS prints only the nonblank lines from each block actually loaded. RCS enhances the printing of the first line of each 10-block load block to serve as a label for the load block. (It does not print the rest of the 10-block load block)

RCS does not print the first line of each program block because it assumes that the first line contains a vocabulary identifier. RCS prints the block number on the line above the block and, if the current page does not have enough room to print a complete 10-block load block, RCS begins a new page.

Paper-tiger

```
Paper-tiger
```

Loads the control words required to use an Integral Data Systems Paper Tiger as the system printer. (See also Printing)

Print

```
[start block#] [end block#] Print
```

Prints the specified range of blocks at a density of three blocks per page.

Printing

```
Printing
```

Loads the printing utilities. Use this word after you select a specific printer. (See also Dec-writer, Image-writer, and Paper-tiger)

Pset-tab

[column#] Pset-tab

Sets a printer tab stop at the specified column. (The first column is column 1)

Restore

Restore

Returns you to the mode you were using before you entered the List-programs word or the List-directory word.

Small

Small

Selects condensed printing for the printers.

Tape words

ADD-TO-TAPE

[absolute start block#] [absolute end block#] ADD-TO-TAPE

Begins at the current tape position and writes the specified range of blocks to tape.

BACKUP

BACKUP

Lists the backup block. You can edit this block to specify customized tape backup procedures.

BAD-TAPE

[absolute start block#] [absolute end block#] BAD-TAPE

Extends the previous file (while preserving file numbers) on the tape to cover a bad section detected by an attempted ADD-TO-TAPE. BAD-TAPE cannot work if the first file on the tape is bad.

FROM-TAPE

[absolute start block#] [absolute end block#] FROM-TAPE

Begins at the current tape position and loads the specified range of blocks into memory.

TAPE

TAPE

Loads the tape utility.

TDOC

[absolute start block#] [absolute end block#] [file#] TDOC

Moves the tape to the start of the specified file and then displays the start and end blocks. TDOC also leaves the start block# and end block# on the stack.

TGOTO

[file#] TGOTO

Moves the tape to the beginning of the specified file.

TREWIND

TREWIND

Rewinds the tape to the beginning.

TUNLOAD

TUNLOAD

Unloads the tape. RCS does not wait for the tape to finish unloading before executing the next word you enter.

Editing words

#LLIST

[block#] #LLIST

Lists a block (specified by its relative block number) with relative and absolute block numbers, but without line numbers.

ØED

[absolute block#] ØED or [absolute block#] Øed

Makes a block (specified by its absolute block number) the current block and then enters the screen editor. (Loaded by EDIT)

Øed

(see ØED)

ØLIST

[absolute block#] ØLIST

Lists a block (specified by its absolute block number) with relative and absolute block numbers and with line numbers.

ØLLIST

[absolute block#] ØLLIST

Lists a block (specified by its absolute block number), but without line numbers.

B

B

Decrements the current block number for listing or editing nearby blocks.

BLOCKS

[source block#] [destination block#] [#blocks] BLOCKS

Copies a number of blocks (specified by #blocks) from the source block number to the destination block number. The source and destination block ranges can overlap. (Loaded by Ed-extend)

CLEAR

[start block#] [end block#] CLEAR

Clears the specified range of blocks by filling the blocks with spaces. (Loaded by Ed-extend)

Datamedia-terminal

Datamedia-terminal

Customizes the screen editor for the Datamedia terminal. (Loaded by EDIT)

ED

[block#] ED or [block#] ed

Makes a block (specified by its relative block number) the current block and then enters the screen editor. If no block number is entered, the screen editor is entered at the current block. (Loaded by EDIT)

ed

(See ED)

EDIT

EDIT

Loads the screen editor. One of the following terminal customization words must follow the word EDIT: TeleVideo-terminal, Datamedia-terminal, or VT1ØØ-terminal. EDIT also loads the following screen editor words: TeleVideo-terminal, Datamedia-terminal, VT1ØØ-terminal, ED, ed, ØED, Øed, RED, and red.

Ed-extend

Ed-extend

Loads the following command extensions to the line editor: BLOCKS, CLEAR, FIND, FIND-R, and R-WITH.

FIND

[start block#] [end block#] FIND [string^]

Locates and displays each line within the specified block range containing the specified string. (Loaded by Ed-extend)

FIND-R

[start block#] [end block#] FIND-R [string^]

Locates and replaces all occurrences of the specified string within the specified block range. You must precede FIND-R with R-WITH to specify the replacement string. (Loaded by Ed-extend) (See also R-WITH)

L

L

Lists the current block with line numbers.

LIST

[block#] LIST

Lists a block (specified by its relative block number) with line numbers.

N

N

Increments the current block number for listing or editing nearby blocks.

QR

[start block#] [end block#] QR

Prompts you for the search and replace strings, then searches through the specified range of blocks for the first occurrence of the search string. After matching the search string, QR waits for you to enter a space, Y, or y, before replacing the string and then continuing to search for the next occurrence.

You can enter a question mark (?) to receive help information. Entering any other character causes QR to leave the matched string unchanged and search for the next occurrence. (Loaded by Query-replace)

Query-replace

Query-replace

Loads the QR utility. (See also QR)

R-WITH

R-WITH [string^]

Places the specified string into the insert buffer. FIND-R uses the string in the insert buffer as the replacement text for its search and replace operation. (Loaded by Ed-extend) (See also FIND-R)

RED

RED or red

Makes current for editing the most recently accessed block on the most recently accessed slave board. You must load RED by using 1124 ØLOAD in CUSTOM after you load the words Master and EDIT.

red

(See RED)

Televideo-terminal

Televideo-terminal

Customizes the screen editor for the TeleVideo 95Ø terminal. (Loaded by EDIT)

VT1ØØ-terminal

VT1ØØ-terminal

Customizes the screen editor for the VT1ØØ terminal. (Loaded by EDIT)

Remote-slave words

Board-sem-initialize

[board segment] Board-sem-initialize

Initializes the semaphore used to control remote/slave operation. This semaphore must be initialized by the master board for each slave board at power-up.

ELOAD

[block#] ELOAD

Empties the local block buffers, then executes LOAD. You must use ELOAD to reload a block onto a slave board after you have edited it from the master board.

Master

Master

Loads the master utility.

MONITOR

[board segment] MONITOR

Displays all output from the specified board and then returns to the master board when the slave becomes idle (waiting for input).

REMOTE

[board segment] REMOTE [string\]

Places the master board into remote mode to communicate with the specified slave board, and then sends the slave board the specified string.

REMOTE-SLAVE

REMOTE-SLAVE

Places the current board into slave mode.

Slave

Slave

Loads the slave utility.

SYNC

[board segment] SYNC

Waits for the specified board to become idle (waiting for input).

TALK\

[board segment] TALK\ [string]

Sends the specified string to the specified board.

^x

^x (This word is control-x, not caret-x)

Terminates slave mode on the current board and returns to the master board, or returns from the master board to the most recently accessed slave board.

Miscellaneous RCS words

"

" [string"] (This word consists of one double quotation mark character)

Displays the specified string. You can use this word in load blocks to display comments during loading.

%

%

Indicates that the rest of the line is a comment.

%%

%%

Indicates that the rest of the block is a comment.

''

'' [string"] (This word consists of two single quotation mark characters)

Reads a string of up to 256 characters from the keyboard or block. RCS stores the string in a temporary area so you can use ==> to initialize a string variable.

(

([string])

Enters a comment that terminates with a closing parenthesis. You cannot nest this kind of comment.

.Esc

.Esc [string]

Sends to the terminal an ASCII ESC code followed by the specified string. You can use this word to program terminal options, such as function keys. This word may be used only in FORTH words, and cannot be used in load blocks. (See also Esc)

.^

.^ [string] (This word consists of a period and a caret)

Converts the ASCII string into the corresponding control characters and sends the control-character string to the terminal. You can use this word to program terminal options, such as function keys. This word may be used only in FORTH words, and cannot be used in a load block.

ØLOAD

[absolute block#] ØLOAD

Loads the specified block from Ø OFFSET, then restores the previous OFFSET.

87INIT

87INIT

Initializes the 8Ø87. All boards must execute 87INIT after booting RCS at power-up.

:L

:L or :l

Places RCS in Locate mode.

:l

(See :L)

:OK

:OK [string"]

Sets the system prompt to the specified string.

:R

:R

Places RCS in Run mode.

:r

See :R

:S

:S

Places RCS in Show mode.

:s

See :S

<PAGE

[pagecode] <PAGE

Converts a pagecode to a page number and leaves this page number on the stack. (See also >PAGE)

==>

[values] ==> [variable]

Moves information from the stack to the specified variable in the current vocabulary. RCS assumes that the information on the stack is correct for the type of variable you specify.

>PAGE

[page#] >PAGE

Converts a page number to a pagecode and leaves the pagecode on the stack. (See also <PAGE)

AA>Q

[vector] [angle] AA>Q

Normalizes the specified vector and converts the vector and the access angle (specified in radians) into a quaternion. This word leaves the quaternion on the stack so that you can use ==> to initialize quaternion variables.

BASE

BASE

Contains the current number base. The value of this variable affects all input and output numbers. (See also BI, DECIMAL, and HEX)

BI

BI

Specifies base 2 operation. The action of this word affects all input and output numbers. (See also BASE, DECIMAL, and HEX)

CR

CR

Sends to the terminal an ASCII carriage-return code followed by an ASCII line-feed code.

DBG-OFF

DBG-OFF

Turns off debug load. (See also DBG-ON)

DBG-ON

DBG-ON

Turns on debug load. Debug load performs three tasks: displays each block number before it is loaded, aborts the load if BASE is not decimal, and displays any numbers still on the stack after each block is loaded. Some of these numbers appear while loading SMACRO code, but any numbers remaining on the stack when loading is complete indicate one or more errors. (See also DBG-OFF)

DECIMAL

DECIMAL

Specifies base 1Ø operation. The action of this word affects all input and output numbers. (See also BASE, BI, and HEX)

deg

[angle in degrees] deg

Converts floating-point angle measurements from degrees to radians and leaves the converted value on the stack.

Esc

Esc [string]

Sends to the terminal an ASCII ESC code, followed by the specified string. You can use this word to program terminal options, such as function keys. This word may be used only in a load block, and cannot be used in a FORTH word. (See also .Esc)

F-MEM

F-MEM

Displays the current, minimum, and maximum highest address for the system dictionary and user files.

GET-FROM-DISK

[page] [segment] [page address] [start block#] [#blocks] GET-FROM-DISK

Moves #blocks * 1Ø24 bytes of data from the disk beginning at the start block# (specified by its absolute block number) to the specified RAM page address. (See also SAVE-ON-DISK)

HEX

HEX

Specifies base 16 operation. The action of this word affects all input and output numbers. (See also BASE, BI, and DECIMAL)

LOAD

[block#] LOAD

Loads the specified block.

load

[block#] load

Loads the specified block relative to the current ref-blk. (See also ref-blk)

LOC

LOC [word]

Lists the source block for the specified word. The error message, **Pre-locate**, indicates that you cannot list the specified word.

MS

MS [matrix variable]

Displays the name of the specified matrix variable and then displays its contents.

NQUIT

NQUIT

Corrects a problem (indicated by the loss of :R in the prompt) that occurs in rare situations.

nz

nz

Contains the specification for the maximum number of zeros FORTH displays following the decimal point and before the most significant digit of a floating-point number. You can set this FORTH variable to display up to 32 characters in a floating-point number.

OFFSET

OFFSET

Contains the value of the current offset. RCS adds the value of this FORTH variable to relative block numbers to generate the absolute block numbers.

OLOAD

[offset] [block#] OLOAD   (Oh-LOAD, not ØLOAD)

Changes the current OFFSET to the specified offset, loads the specified block, and then restores the previous OFFSET.

PAGE

PAGE

Sends a PAGE character to the terminal. This PAGE (also called a form feed) character may be different for different types of terminals.

PAGE-MOVE

[source seg] [address] [dest seg] [address] [#bytes] [pagecode] PAGE-MOVE

Moves #bytes from the specified address in the source segment to the specified address in the destination segment. Either the source or the destination segment must be in the current board. The other segment must be in the page specified by the pagecode.

PAGE-SEG-ERASE

[page] [segment] PAGE-SEG-ERASE

Fills with zeros the 64-kilobyte segment beginning at the specified page and segment address.

PDUMP

[page] [segment] [address] [#bytes] PDUMP

Displays #bytes (in hex) beginning at the specified page address.

PMS

PMS [pose variable]

Displays the name of the pose variable and shows it as a rotation matrix with a translation vector.

PRINT-TIME

[#tics] PRINT-TIME

Displays the result (in microseconds) of TIMER-READ execution. (See also TIMER READ and TIMER-START)

QMS

QMS [quaternion variable]

Displays the name of the specified quaternion matrix variable and shows the quaternion as a matrix.

QS

QS [quaternion variable]

Displays the name of the specified quaternion variable and shows the quaternion value as an axis and an angle.

rad

[angle in radians] rad

Converts floating-point angle measurements from radians to degrees and leaves the converted value on the stack.

ref-blk

[block#] ref-blk

Updates the current ref-blk by making it the sum of the specified block number and the current block number. (See also load)

SAVE-ON-DISK

```
[page] [segment] [page address] [start block#] [#blocks] SAVE-ON-DISK
```

Moves #blocks * 1024 bytes of data from the specified RAM page address to the disk, beginning at start block# (specified by its absolute block number). (See also GET-FROM-DISK)

sd

```
sd
```

Contains the specification for the number of significant digits FORTH displays in floating-point numbers and leaves this value on the stack. You can set this FORTH variable to display up to 8 significant digits in a floating-point number.

SEG-DUMP

```
[segment] [address] [#bytes] SEG-DUMP
```

Displays #bytes (specified in hex) beginning at the specified segment address.

SEG-MOVE

```
[source address] [dest address] [#bytes] [source seg] [dest seg] SEG-MOVE
```

Moves #bytes from the source segment address to the destination segment address.

TAB

```
TAB
```

Sends an ASCII TAB character to the terminal.

THRU

```
[start block#] [end block#] THRU
```

Loads the specified range of blocks.

TIME-OUT-INIT

TIME-OUT-INIT

Enables the 6-millisecond time-out interrupt, which aborts access to a memory location or I/O port if the hardware does not respond within 6 milliseconds of the request. This interrupt generates the message **6 msec time-out interrupt.** To enable this interrupt, the system must execute TIME-OUT-INIT at power-up.

TIME-P

TIME-P [routine]

Displays the name of the specified routine, executes the routine, and displays the time of execution. TIME-P subtracts the appropriate overhead time required to call the routine from the terminal task to provide an accurate measurement of the time required to execute the routine. TIME-P uses the same timer (timer 1) as TIMER-READ and TIME-START.

The overhead factor is correct for 8-MHz operation. You should enter the following lines in the D>M auto-load block for each board running at 5-MHz:

```
16 fnul-word-time !
81 snul-word-time !
```

TIMER-READ

TIMER-READ

Reads the timer and returns on the stack the number of ticks from the 153-KHz clock. TIMER-READ uses the same timer (timer 1) as TIME-P and TIMER-START. (See also PRINT-TIME and TIMER-START)

TIMER-START

TIMER-START

Starts the timer. TIMER-START uses the same timer (timer 1) as TIME-P and TIMER-READ. (See also PRINT-TIME and TIMER-READ)

V-SIZE

V-SIZE

Displays the current vocabulary, the number of words defined in it, and the maximum number of words the vocabulary can hold.

\L

Re-executes the last SMACRO word executed, this time in Locate mode.

\l

(See \L)

\N

Displays the name of the last SMACRO word executed.

\n

(See \N)

\R

Re-executes the last SMACRO word executed, this time in Run mode.

\r

(See \R)

\S

Re-executes the last SMACRO word executed, this time in Show mode.

\s

(See \S)

\^

^ [string] (This word consists of a caret)

Converts the ASCII string into the corresponding control characters and sends this control-character string to the terminal. You can use this word to program terminal options, such as function keys. This word may be used only in a load block, and may not be used in a FORTH word. (See also .^)

~DUMP

[address] [#bytes] ~DUMP

Displays #bytes (in hex) from the specified SMACRO segment address.

~MEM

~MEM

Displays the number of bytes remaining in the SMACRO segment.

~MOVE

[source] [destination] [#bytes] ~MOVE

Moves #bytes (specified in hex) from the SMACRO segment source to the SMACRO segment destination.

FORTH Words

!

[value] [FORTH address] !

Stores the specified 16-bit integer value in the specified FORTH address.

*

[value1] [value2] *

Calculates the product of two 16-bit integer values and leaves the result on the stack.

+

[value1] [value2] +

Calculates the sum of two 16-bit integer values and leaves the result on the stack.

-

[value1] [value2] -

Calculates the difference between two 16-bit integer values by subtracting value2 from value1 and leaves the result on the stack.

.

[value] .

Displays the specified 16-bit integer value.

.D

[value] .D

Displays the specified 16-bit decimal integer value, but does not change the current base.

/

[value1] [value2] /

Calculates the quotient of two 16-bit integer values by dividing value1 by value2 and leaves the result on the stack.

2!

[address] [value] 2!

Stores a 32-bit integer in a FORTH variable.

2@

[address] [value] 2@

Retrieves a 32-bit integer from a FORTH variable.

2IN

[port address] 2IN

Reads the 8-bit port twice and leaves the resulting 16-bit value on the stack. This word reads the low-order byte of this 16-bit value before the high-order byte.

2OUT

[value] [port address] 2OUT

Sends the specified 16-bit value to the specified output port as 2 bytes. This word sends the low-order byte of this 16-bit value before it sends the high-order byte.

:

:[name]

Creates a FORTH word and causes the system to enter the FORTH Compile mode.

:C~EXEC

:C~EXEC [SMACRO word]

Sets the mode temporarily to Compile, compiles a call to the specified SMACRO word, and restores the previous mode after the SMACRO word executes. This word may be used only in FORTH words, not in a load block.

:L~EXEC

:L~EXEC [SMACRO word]

Sets the mode temporarily to Locate, compiles a call to the specified SMACRO word, and restores the previous mode after the SMACRO word executes. This word may be used only in FORTH words, not in a load block.

:S~EXEC

:S~EXEC [SMACRO word]

Sets the mode temporarily to Show, compiles a call to the specified SMACRO word, and restores the previous mode after the SMACRO word executes. This word may be used only in FORTH words, not in a load block.

;

;

Causes the system to exit from the FORTH Compile mode.

?

[FORTH address] ?

Displays the 16-bit integer value contained in the specified FORTH address.

@

[FORTH address] @

Retrieves a 16-bit integer value from the specified FORTH address and leaves it on the stack.

B.

[value] B.

Displays the specified 16-bit integer value in binary, but does not change the current base.

BACKGROUND

[stack size] [return stack size] [# user variables] BACKGROUND

Allots space for a background task by creating an entry in the FORTH dictionary with the parameter fields providing (in the following order) the background task address, stack origin, and the user area size (expressed as an 8-bit value). FORTH adjusts the task address upwards to fall on a 16-byte boundary to simplify segment register manipulations. (See also BUILD)

BUILD

[task address] BUILD

Inserts the background task from the specified task address into the FORTH round-robin. BUILD places the new background task immediately after the current task in the round-robin, copies the user variables declared for the background task from the operator task, and sets the background stack pointer to its origin. (See also BACKGROUND)

CLR-BIT

[bit#] [port address] CLR-BIT

Clears the specified bit in the specified port to Ø. Other bits remain unaffected. You can use this word to clear interrupt masks with PIC. (See also SET-BIT)

DUMP

[FORTH address] [#bytes] DUMP

Displays #bytes beginning at the specified FORTH address.

F*

[value1] [value2] F*

Calculates the product of two 32-bit floating-point values and leaves the result on the stack.

F+

[value1] [value2] F+

Calculates the sum of two 32-bit floating-point numbers and leaves the result on the stack.

F-

[value1] [value2] F-

Calculates the difference of two 32-bit floating-point values by subtracting value2 from value1 and leaves the result on the stack.

F-ABS

[value] F-ABS

Calculates the absolute 32-bit floating-point value of the specified value and leaves the result on the stack.

F.

[value] F.

Displays the specified 32-bit floating-point value.

F/

[value1] [value2] F/

Calculates the quotient of two 32-bit floating-point values by dividing value1 by value2 and leaves the result on the stack.

FLOAT

[integer] FLOAT

Converts a 16-bit integer value to a 32-bit floating-point value and leaves the converted value on the stack. (See also ROUND)

H.

[value] H.

Displays the specified 16-bit integer value in hex, but does not change the current base.

INPUT

[port address] INPUT

Reads the 8-bit value from the specified input port and leaves the value on the stack.

IRR

IRR

Returns on the stack the port address for the Interrupt Request Register in the Programmable Interrupt Controller. This FORTH constant is useful for determining if interrupts are connected correctly.

LOCATE-OFF

LOCATE-OFF

Turns off locate for FORTH words.

LOCATE-ON

LOCATE-ON

Turns on locate for FORTH words. You cannot locate words defined with locate off. Turning off locate saves 2 bytes of FORTH space per word. (Locate is always on for SMACRO words)

MOVE

[source FORTH address] [destination FORTH address] [#bytes] MOVE

Moves #bytes from the source FORTH address to the destination FORTH address.

OUTPUT

[value] [port address] OUTPUT

Sends the specified 8-bit value to the specified output port.

PI

PI

Leaves the 32-bit floating-point value of pi on the stack.

PIC

PIC

Returns on the stack the port address for the Programmable Interrupt Controller. You use this port to mask and unmask interrupts.

ROUND

[float] ROUND

Converts a 32-bit floating-point value to a 16-bit integer value through rounding and leaves the converted value on the stack. (See also FLOAT)

SET-BIT

[bit#] [port address] SET-BIT

Sets the specified bit in the specified port to 1. Other bits remain unaffected. You can use this word to set interrupt masks with PIC. (See also CLR-BIT)

SQRT

[value] SQRT

Calculates the square root of the specified 32-bit floating-point value and leaves the result on the stack.

~!

[value] [SMACRO address] ~!

Stores the specified 16-bit integer value in the specified SMACRO address.

~?

[SMACRO address] ~?

Displays the 16-bit integer value contained in the specified SMACRO address.

~@

[SMACRO address] ~@

Retrieves the 16-bit integer value contained in the specified SMACRO address and leaves the value on the stack.

~C!

[value] [address] ~C!

Stores data in a SMACRO byte variable.

~C@

[address] [value] ~C@

Retrieves data from a SMACRO byte variable.

~F!

[value] [SMACRO address] ~F!

Stores the specified 32-bit floating-point value in the specified SMACRO address.

~F?

[SMACRO address] ~F?

Displays the 32-bit floating-point value contained in the specified SMACRO address.

~F@

[SMACRO address] ~F@

Retrieves the 32-bit floating-point value contained in the specified SMACRO address.

~FR?

[address] ~FR?

Reads a 32-bit floating-point value from the specified address, converts it from radians to degrees, and then displays the converted value.

SMACRO Words

The SMACRO words include the following types:

- SMACRO declaration words
- SMACRO predefined SMACRO variable words
- SMACRO operator words
- SMACRO vector-operator words
- SMACRO quaternion-operator words
- SMACRO matrix-operator words
- SMACRO pose-operator words
- SMACRO boolean-operator words
- SMACRO file-operation variable words
- SMACRO file-operation header variable words
- SMACRO file-operation words
- SMACRO boolean file-operation words
- SMACRO statement words
- Miscellaneous SMACRO words

SMACRO declaration words

1:a

```
[#elements] 1:a [name]
```

Creates a one-dimensional integer array, which belongs to the most recently declared variable owner. The parameter #elements must be less than 256.

1:fa

```
[#elements] 1:fa [name]
```

Creates a one-dimensional floating-point array, which belongs to the most recently declared variable owner. The parameter #elements must be less than 256.

2:a

```
[#rows] [#columns] 2:a [name]
```

Creates a two-dimensional integer array, which belongs to the most recently declared variable owner. SMACRO stores the array by rows. The product of the #rows and the #columns must be less than 256.

2:fa

[#rows] [#columns] 2:fa [name]

Creates a two-dimensional floating-point array, which belongs to the most recently declared variable owner. SMACRO stores the array by rows. The product of the #rows and the #columns must be less than 256.

<=

<= [values]

Stores the specified values in the most recently declared variable. The values are assumed to be of the correct type and number for the variable type. This declaration word works only for string, byte, and integer variables. (See ==> for other variable types) The parameter <= is used to initialize variables as they are declared, not in routines.

bv

bv [name]

Creates an 8-bit byte variable, which belongs to the most recently declared variable owner.

bytes

bytes

Performs no function. You can use this nul word to make source code more readable (for example, 2 bytes VAR-0).

end-routine

end-routine

Terminates the Compile mode. (See also routine)

FILE

[max #records] FILE [file name]

Creates a file with the specified name, using the most recently declared variable owner as a template. Include this statement in the source block that declares the variable owner. The max #records parameter specifies the size of the new file.

fv

fv [name]

Creates a 32-bit floating-point variable, which belongs to the most recently declared variable owner.

iv

iv [name]

Creates a 16-bit integer variable, which belongs to the most recently declared variable owner.

mem

mem

Performs no function. You can use this nul word, which is an abbreviation for members, to make source code more readable (for example, 1Ø mem S-VAR-O).

rec

rec

Performs no function. You can use this nul word, which is an abbreviation for records, to make source code more readable (for example, 1Ø rec FILE).

routine

routine [name]

Creates a routine with the specified name and enters the Compile mode. (See also end-routine)

S-VAR-O

[#members] S-VAR-O [name]

Creates a sequential variable owner with the specified name. The sequential variable can accept a maximum of #members.

segv

[segment] [address] [#bytes] segv [name]

Creates a segment variable, which belongs to the most recently declared variable owner.

seqv

seqv [name]

Creates a sequential variable, which belongs to the most recently declared sequential variable owner. The first sequential variable receives a value of Ø, the second variable receives a value of 1, and each successive variable receives the next sequential integer value.

strv

[#bytes] strv [name]

Creates a string variable #bytes long with the specified name, which belongs to the most recently declared variable owner.

VAR-O

[#bytes] VAR-O [name]

Creates a variable owner with the specified name and allots to this variable owner #bytes of space in the SMACRO segment for variable storage.

VOCABULARY

[assembler voc#] [SMACRO voc#] [#entries] VOCABULARY [name]

Creates a vocabulary. Enter the vocabulary numbers (voc#s) in hexadecimal. Both voc#s are four-digit numbers. The first three digits for the assembler vocabulary must be 513. The first three digits for the SMACRO vocabulary must be 531. The fourth digit of each number must be the same; must be 7, 9, B, D, or F; and must be different from any other vocabulary declared on the same board. The vocabulary name must be unique in the system, because RCS does not check for duplicate vocabulary names. The system may contain up to 2Ø vocabularies.

Predefined SMACRO variable words

-1# 1# 2# 3# 4# 5# 6# 7# 8# 9# 1Ø# 11# 12# 13# 14# 15# 16# 17# 18# 19# 32767#

Each of these predefined integer variables contains the indicated value (for example, -1# contains the value -1).

Ø.# 1.# 2.#

Each of these predefined floating-point variables contains the indicated value (for example, 2.# contains the floating-point value 2.Ø).

cos{angle}

Predefined floating point variable used as output for certain quaternion operators.

cycle-count

A segment variable that provides the current cycle count. The communications process updates the cycle count.

false

Predefined sequential variable with integer value Ø. You use the predefined sequential variables false, true, set, reset, ready, and not ready for logic tests. Their integer values are Ø, 1, 2, 3, 4, and 5, respectively.

identity-pose

This predefined pose variable contains the identity pose, 1.Ø Ø.Ø Ø.Ø Ø.Ø Ø.Ø Ø.Ø Ø.Ø.

not-ready

Predefined sequential variable with integer value 5. (See also false)

ready

Predefined sequential variable with integer value 4. (See also false)

reset

Predefined sequential variable with integer value 3. (See also false)

rot-angle

Predefined floating-point variable used as output for certain quaternion operators.

rot-axis

Predefined floating-point variable used as output for certain quaternion operators.

set

Predefined sequential variable with integer value 2. (See also false)

sin{angle}

Predefined floating-point variables used as outputs of certain quaternion operators.

true

Predefined sequential variable with integer value 1. (See also false)

X Y Z

These predefined integer variables contain Ø, 1, and 2, respectively. You can use these predefined variables with pose operators to identify frame axes.

SMACRO operator words

The general spelling convention for SMACRO operators is as follows: operators surrounded by () are integer, operators surrounded by .. are floating point, and operators ending in _B are byte. (There may be exceptions to this convention) See Table B-1 earlier in this appendix for definitions of the abbreviations used in syntax descriptions.

""

""

Returns the result of the most recent integer expression. This word may operate incorrectly if you place other code between the most recent integer expression and the pair of double quotation mark characters.

#.READ

#.READ

Reads a word (bounded by ASCII spaces) from the current input and tries to find the word in the current vocabulary. If #.READ finds the word in the current vocabulary, it assumes that the word is a floating-point variable and returns the value of the variable. Otherwise, #.READ assumes that the word is a floating-point number and returns this value. You can use the #.READ word in place of a floating-point variable except as the destination of the .=>. SMACRO word.

#READ

#READ

Reads a word (bounded by ASCII spaces) from the current input and tries to find the word in the current vocabulary. If #READ finds the word in the current vocabulary, it assumes that the word is an integer variable and returns the value of the variable. Otherwise, #READ assumes that the word is an integer number and returns this value. You can use #READ in place of an integer variable except as an array index or as the destination of the => SMACRO word.

(( ))

(([ivar]))

Causes SMACRO to treat the integer variable, ivar, as a pointer.

()

[var] ()

Specifies an address by adding the current value of the index register to the value of var. There is no space between the parentheses. (See also Index-on)

(*)

[iexp] (*) [ivar]

Returns the product of iexp and ivar (iexp * ivar).

(*/)

[iexp] (*/) [ivar1] [ivar2]

Returns the result of iexp * ivar1/ivar2. The intermediate results of this expression can occupy up to 32 bits to maintain accuracy.

(+)

[iexp] (+) [ivar]

Returns the sum of iexp and ivar (iexp + ivar).

(-)

[iexp] (-) [ivar]

Returns the difference of ivar subtracted from iexp (iexp - ivar).

(/)

[iexp] (/) [ivar]

Returns the quotient of iexp divided by ivar (iexp/ivar).

(ABS)

(ABS) [iexp]

Returns the absolute value of iexp.

(COM)

(COM) [iexp]

Returns the one's complement of iexp.

(MAX)

[iexp1] (MAX) [iexp2]

Returns the signed maximum integer of iexp1 and iexp2.

(MIN)

[iexp1] (MIN) [iexp2]

Returns the signed minimum integer of iexp1 and iexp2.

(MOD)

[iexp1] (MOD) [iexp2]
Performs a modulo operation (iexp1 modulo iexp2) and returns the result using the same sign as iexp1.

.( ).

.( ).

Specifies the processing order for floating-point operations within an expression. Scalar operations can support up to eight levels of nesting, vector operations can support up to one level of nesting, and pose and quaternion operations can support any number of levels of nesting.

.*.

[fexp1] .*. [fexp2]

Returns the floating-point product of fexp1 and fexp2.

.*2**.

[fexp1] .*2**. [fexp2]

Rounds fexp2 to an integer value and then returns a value computed according to the following expression:

$$\text{fexp1} * 2^{\text{fexp2}}$$

.+.

[fexp1] .+. [fexp2]

Returns the floating-point sum of fexp1 and fexp2.

.-.

[fexp1] .-. [fexp2]

Returns the floating-point difference of fexp1 minus fexp2.

./.

[fexp1] ./. [fexp2]

Returns the floating-point quotient of fexp1 divided by fexp2.

.=>.

[fexp] .=>. [fvar]

Stores fexp into fvar.

.ABS.

.ABS. [fexp]

Returns the absolute value of fexp.

.ATAN,SIN,COS.

.ATAN,SIN,COS. [fexp1] [fexp2]

Returns the arctangent, in radians of the quotient resulting from dividing fexp1 by fexp2 (fexp1/fexp2). This word also returns the SIN and COS values of the quotient in the predefined variables sin{angle} and cos{angle}, respectively.

.ATAN.

.ATAN. [fexp1] [fexp2]

Returns the arctangent in radians of the quotient resulting from dividing fexp1 by fexp2 (fexp1/fexp2).

.FRAC.

.FRAC. [fexp]

Returns the fraction portion of fexp as a floating-point value.

.INT.

.INT. [fexp]

Returns the integer portion of fexp as a floating-point value.

.MAX.

[fexp1] .MAX. [fexp2]

Returns the signed maximum floating-point value of fexp1 and fexp2.

.MIN.

[fexp1] .MIN. [fexp2]

Returns the signed minimum floating-point value of fexp1 and fexp2.

.MINUS.

.MINUS. [fexp]

Multiplies fexp by -1.Ø and returns the result.

.MOD+-18Ø.

.MOD+-18Ø. [fexpr]

Forces the result of fexpr into the range ± Pi radians (±18Ø degrees).

.SIGN.

.SIGN. [fexp]

Returns the floating-point sign of fexp as +1.Ø, -1.Ø, or Ø.Ø if the value of fexp is greater than Ø, less than Ø, or equal to Ø, respectively.

.SIN,COS.

.SIN,COS. [fexp]

Returns the sine and cosine of fexp in radians in the predefined SMACRO variables sin{angle} and cos{angle}. This word is a statement in itself and cannot be used as part of another expression.

.SQ.

.SQ. [fexp]

Returns the square of fexp.

.SQRT.

.SQRT. [fexp]

Returns the square root of fexp. This word generates an 8Ø87 error if fexp has a negative value.

2pi

2pi

Returns the indicated floating-point value.

=>

[iexp] => [ivar]

Stores iexp into ivar.

=>SEG

[var] [#bytes] =>SEG [segvar]

Moves #bytes from var to the segment address identified in pointer variable, segvar.

a#

a# [var]

Returns the address, not the value, of var.

B*

[iexp] B* [ivar]

Returns the byte product of iexp and ivar.

B+

[iexp] B+ [ivar]

Returns the byte sum of iexp and ivar.

B-

[iexp] B- [ivar]

Returns the byte difference of iexp minus ivar.

B-to-stack

[iexp] B-to-stack

Converts iexp to a byte value and then pushes it onto the 8Ø86 stack.

B/

[iexp] B/ [ivar]

Returns the byte quotient of iexp divided by ivar.

B=>

[iexp] B=> [bvar]

Converts iexp to a byte value and stores the resulting value in bvar.

F>I

F>I [fexp]

Returns the integer value obtained by rounding the floating-point value of fexp.

from-stack

from-stack

Pops the 8Ø86 stack and returns the resulting integer value. (See also to-stack)

I>F

I>F [iexp]

Converts the integer value of iexp to a floating-point value.

in-port

[ivar] in-port

Returns the byte value read from the 8-bit I/O port specified by the address value in ivar.

index-on

index-on [ivar]

Loads ivar into the indexing register for use with the () operator.

mask-with

[iexp] mask-with [ivar]

Returns the result of a bitwise AND using iexp and ivar (iexp AND ivar).

out-port

[iexp] out-port [ivar]

Truncates iexp to an 8-bit value and sends it to the 8-bit output port whose address is contained in ivar.

pi

pi

Returns the indicated floating-point value.

pi/2

pi/2

Returns the indicated floating-point value.

READ"

READ"

Reads a string terminated by an ASCII double quotation mark character (") from the current input. You must follow READ" with S=> to store the string in a variable.

SEG=>

[segvar] [#bytes] SEG=> [var]

Moves #bytes to var from the segment address identified in pointer variable, segvar.

to-stack

[iexp] to-stack

Pushes iexp onto the 8Ø86 stack. (See also from-stack)

WREAD

WREAD

Reads a string terminated by an ASCII space character. WREAD must be followed by S=> to store the string in a variable. If the input string is to contain spaces, use READ" instead of WREAD.

{ }

array-var {[ivar1] [ivar2]}

Provides a means of indexing arrays. In the case of one-dimensional arrays, ivar1 specifies the element number (beginning at Ø) and the second variable, ivar2, is not included. In the case of two-dimensional arrays, ivar1 specifies the row number and ivar2 specifies the column number.

SMACRO vector-operator words

.CROSS.

[vexp1] .CROSS. [vexp2]

Returns the cross-product of vexp1 and vexp2.

.DOT.

[vexp1] .DOT. [vexp2]

Returns the dot-product of vexp1 and vexp2.

.UNIT.

.UNIT. [vexp]

Returns a unit vector parallel to vexp.

.V*S.

[vexp] .V*S. [fexp]

Returns the vector obtained by multiplying each element of vexp by fexp.

.V+.

[vexp1] .V+. [vexp2]

Returns the vector obtained by adding vexp1 to vexp2, element by element.

.V-.

[vexp1] .V-. [vexp2]

Returns the vector obtained by subtracting vexp2 from vexp1, element by element.

.V/S.

[vexp] .V/S. [fexp]

Returns the vector obtained by dividing the elements of vexp by fexp.

.V=>.

[vexp] .V=>. [vvar]

Stores vexp into vvar.

.VINV.

.VINV. [vexp]

Returns the generalized inverse of vexp (that is, vexp/$|\text{vexp}|^2$).

.VMAG.

.VMAG. [vexp]

Returns the magnitude of vexp.

.VMINUS.

.VMINUS. [vexp]

Returns the negative of vexp.

SMACRO quaternion-operator words

.AXIS,ANGLE>Q.

.AXIS,ANGLE>Q. [vexp] [fexp]

Treats vexp as an axis and fexp as an angle in radians, and returns the equivalent quaternion. The vector vexp must be a unit vector. (See also .UNIT. in the section "SMACRO vector-operator words")

.AXIS,SIN,COS>Q.

.AXIS,SIN,COS>Q. [vexp] [fexp1] [fexp2]

Treats vexp as an axis, fexp1 as sin{angle}, fexp2 as cos{angle}, and returns the equivalent quaternion. The vector vexp must be a unit vector. (See also .UNIT.)

.Q=>.

[qexp] .Q=>. [qvar]

Stores qexp into variable qvar.

.Q>AXIS,ANGLE,SIN,COS.

.Q>AXIS,ANGLE,SIN,COS. [qexp]

Converts qexp into an axis and angle representation. This word returns its results in the predefined variables rot-axis, rot-angle in radians (Ø <= rot-angle <=pi), sin{angle}, and cos{angle}. .Q>AXIS,SIN,COS. is faster than .Q>AXIS,ANGLE,SIN,COS. but it does not compute the angle. This word is a statement in itself and cannot be used as part of another expression.

.Q>AXIS,SIN,COS.

.Q>AXIS,SIN,COS. [qexp]

Converts qexp into an axis and angle representation. This word returns its results in the predefined variables rot-axis, sin{angle}, and cos{angle}. .Q>AXIS,SIN,COS. is faster than .Q>AXIS,ANGLE,SIN,COS. because it does not compute the angle. This word is a statement in itself and cannot be used as part of another expression.

.QINV.

.QINV. [qexp]

Returns the inverse of qexp.

.QQ*.

[qexp2] .QQ*. [qexp1]

Returns the quaternion product of qexp2 and qexp1.

.QV*.

[qexp] .QV*. [vexp]

Returns the vector obtained by applying the rotation qexp to vexp.

.V>Q.

.V>Q. [vexp]

Returns the quaternion equivalent to the rotation vector, vexp, by treating .UNIT. vexp as an axis and .VMAG. vexp as an angle in radians.

SMACRO matrix-operator words

.M=>.

[mexp] .M=>. [mvar]

Stores mexp into mvar.

.M>Q.

.M>Q. [mexp]

Returns the quaternion equivalent to mexp.

.Q>M.

    .Q>M. [qexp]

Returns the matrix equivalent to qexp.

SMACRO pose-operator words

.P=>.

    [pexp] .P=>. [pvar]

Stores pexp into pvar.

.PINV.

    .PINV. [pexp]

Returns the inverse of pexp.

.PP*.

    [pexp2] .PP*. [pexp1]

Returns the POSE product of pexp2 and pexp1.

.PQ*.

    [pexp] .PQ*. [qexp]

Returns a pose computed by multiplying the quaternion part of pexp by qexp. The vector part of pexp becomes the vector part of the result. The operator .PQ*. rotates the pose frame by a rotation expressed in the pose frame.

.PV+.

    [pexp] .PV+. [vexp]

Returns a pose computed by multiplying the quaternion part of pexp by vexp, and then adding the result to the vector part of pexp. The operator .PV+. translates the pose frame by a translation expressed in the pose frame.

.QP*.

[qexp] .QP*. [pexp]

Returns a pose computed by multiplying the qexp by the quaternion part of pexp. The vector part of the result is the vector part of pexp. The operator .QP*. rotates the pose frame by a quaternion expressed in the base frame.

.ROT,SIN,COS.

[pexp] .ROT,SIN,COS. [ivar] [fexp1] [fexp2]

Treats ivar as the axis of the pose frame and uses sin (fexp1) and cos (fexp2) to define the angle of rotation, and then returns a pose computed by rotating the quaternion part of pexp about the axis through the specified angle. The axis is one of the x, y, or z axes of the pose frame as indicated by an ivar value of Ø, 1, or 2, respectively. (Use the predefined variables X, Y, and Z) The vector part of the result is the vector part of pexp. .ROT,SIN,COS. operates faster than .PQ*. or .ROT. to rotate the pose frame by a quaternion expressed in the pose frame.

.ROT.

[pexp] .ROT. [ivar] [fexp]

Treats ivar as one of the axes of the pose frame and fexp as the angle of rotation in radians, and then returns a pose computed by rotating the quaternion part of pexp about the axis through the specified angle. The axis is one of the x, y, or z axes of the pose frame as indicated by an ivar value of Ø, 1, or 2, respectively. (Use the predefined variables X, Y, and Z) The vector part of the result is the vector part of pexp. The operator .ROT. operates faster than .PQ*. to rotate the pose frame by a quaternion expressed in the pose frame.

.TRN.

[pexp] .TRN. [ivar] [fexp]

Treats ivar as the axis of the pose frame and fexp as the translation distance, and then returns a pose computed by translating the vector part of pexp along the specified axis by the specified distance. The axis is one of the x, y, or z axes of the pose frame as indicated by an ivar value of Ø, 1, or 2, respectively. (Use the predefined variables X, Y, and Z) The quaternion part of the result is the quaternion part of pexp. The effect of this word is to translate the pose frame by a vector expressed in the pose frame. The operator .TRN. operates faster than .PV+. to translate the pose frame by a vector expressed in the pose frame.

.VP+.

[vexp] .VP+. [pexp]

Returns a pose computed by adding vexp to the vector part of pexp. The quaternion part of the result is the quaternion part of pexp. This word translates the pose frame by a vector expressed in the base frame.

SMACRO Boolean-operator words

(AND)

[boolean exp] (AND) [boolean exp]

Returns true only if both Boolean expressions are true.

(EQ)

[iexp] (EQ) [ivar]

Returns true only if the operands are equal.

(GE)

[iexp] (GE) [ivar]

Returns true only if iexp is greater than or equal to ivar.

(GT)

[iexp] (GT) [ivar]

Returns true only if iexp is greater than ivar.

(LE)

[iexp] (LE) [ivar]

Returns true only if iexp is less than or equal to ivar.

(LT)

[iexp] (LT) [ivar]

Returns true only if iexp is less than ivar.

(NE)

[iexp] (NE) [ivar]

Returns true only if the operands are not equal.

.EQ.

[fexp1] .EQ. [fexp2]

Returns true only if the operands are equal.

.EQZ.

[fexp] .EQZ.

Returns true only if fexp is equal to Ø.

.GE.

[fexp1] .GE. [fexp2]

Returns true only if fexp1 is greater than or equal to fexp2.

.GEZ.

[fexp] .GEZ.

Returns true only if fexp is greater than or equal to Ø.

.GT.

[fexp1] .GT. [fexp2]

Returns true only if fexp1 is greater than fexp2.

.GTZ.

[fexp] .GTZ.

Returns true only if fexp is greater than Ø.

.LE.

[fexp1] .LE. [fexp2]

Returns true only if fexp1 is less than or equal to fexp2.

.LEZ.

[fexp] .LEZ.

Returns true only if fexp is less than or equal to Ø.

.LT.

[fexp1] .LT. [fexp2]

Returns true only if fexp1 is less than fexp2.

.LTZ.

[fexp] .LTZ.

Returns true only if fexp is less than Ø.

.NE.

[fexp1] .NE. [fexp2]

Returns true only if the operands are not equal.

.NEZ.

[fexp] .NEZ.

Returns true only if fexp is not equal to Ø.

Ø-?

[ivar1] Ø-? [ivar2]

Returns true only if the result of a bitwise AND of ivar2 with the one's complement of ivar1 is not Ø.

1-?

[ivar1] 1-? [ivar2]

Returns true only if the result of a bitwise AND of ivar1 with ivar2 is not Ø.

EQ_B

[iexp] EQ_B [ivar]

Truncates the operands iexp and ivar to 8 bits and returns true only if the truncated operands are equal.

GE_B

[iexp] GE_B [ivar]

Truncates iexp and ivar to 8 bits and returns true only if the truncated iexp is greater than or equal to the truncated ivar.

GT_B

[iexp] GT_B [ivar]

Truncates iexp and ivar to 8 bits and returns true only if the truncated iexp is greater than the truncated ivar.

LE_B

[iexp] LE_B [ivar]

Truncates iexp and ivar to 8 bits and returns true only if the truncated iexp is less than or equal to the truncated ivar.

LT_B

[iexp] LT_B [ivar]

Truncates iexp and ivar to 8 bits and returns true only if the truncated iexp is less than the truncated ivar.

NE_B

[iexp] NE_B [ivar]

Truncates the operands iexp and ivar to 8 bits and returns true only if the truncated operands are not equal.

S-EQ

[svar] S-EQ [svar]

Returns true only if the ASCII values of two string variables are equal. S-EQ assumes the lengths of the strings are equal.

^EQ^

[ivar1] ^EQ^ [ivar2]

Returns true only if the Boolean value of ivar2 is equal to the result of a bitwise AND of ivar1 with ivar2.

SMACRO file-operation variable words

File-var

Contains all the file-operation and header variables. Showing this variable owner can be useful in debugging file operations.

next-record

Points to the next record in the file.

record#

Points to the current record in the file. All file access procedures use this variable.

source-blk

Specifies the block (if any) that contains the source code for that record.

SMACRO file-operation header variable words

ass/rec

ass/rec

Contains the number of records removed from the free list and assigned to other lists.

avail-record

avail-record

Points to the beginning of the free list.

bytes/record

bytes/record

Specifies the number of bytes in each record in the file, including the link and source fields.

f-segment

f-segment

Specifies the segment address for the file.

f-template

f-template

Specifies the address of the local template.

file-start

file-start

Specifies the address of the first physical record in the file segment.

first-record

first-record

Points to the first record in a list in the current file. This header variable provides a simple way to manage files that consist of a single list. The first-record variable contains the pointer to the beginning of the list, and is updated as needed by the file operators. This variable has other uses in multiple-list files.

max/rec

max/rec

Contains the total number of records allocated for the file.

SMACRO file-operation words

See Table B-2 at the beginning of this appendix for definitions of the phrases used in the syntax descriptions.

add-list

add-list

Stores the data from the local template to a record, removes the record from the free list, and makes the record the beginning of a new list. This operator leaves record# and first-record set to the record stored, increments ass/rec, sets source-blk and the source block field to the current block, and sets next-record and the link field to Ø. The word add-list aborts if the file is full.

add-record

add-record

Stores the data from the local template to a record, removes the record from the free list, and adds the record to the end of the current list. This operator leaves record# set to the record stored, increments ass/rec, sets source-blk and the source block field to the current block, and sets the next-record and link field to Ø. If there is no current list (that is, if record# is Ø), SMACRO executes add-list automatically in place of add-record. The word add-record aborts if the file is full.

clear-list

clear-list

Returns the entire current list to the free list. This operation does not abort if record# is Ø, but in this case no action takes place because there is no list to clear.

remove

remove

Removes the current record from the list pointed to by first-record, and returns it to the free list, sets first-record to the beginning of the resulting list, sets next-record to the record following the removed record, and decrements ass/rec. The operation of remove aborts if there is no current record, or if the current record is not on the list.

retrieve

retrieve

Retrieves the current record from the current file and leaves the data in the local template. The word retrieve aborts if record# is Ø.

retrieve-from-field

retrieve-from-field [field] [destination]

Retrieves the data from the specified field in the current record and stores the data in the specified destination. The word retrieve-from-field aborts if record# is Ø.

retrieve-from-fields

retrieve-from-fields [field1] [field2] [destination]

Retrieves the data from fields in the range field1 through field2 in the current record and stores the data in the specified destination. The destination must be the beginning of a set of consecutive variables whose total byte count matches the total of the fields being retrieved. The word retrieve-from-fields aborts if record# is Ø.

retrieve-l&r

retrieve-l&r

Retrieves the following three items: the current record to the local template, the link field to the variable next-record, and the source field to the variable source-blk. The word retrieve-l&r aborts if record# is Ø.

retrieve-link

retrieve-link [var]

Retrieves the record# from the link field of the current record and stores it in var. This word does not abort if record# is Ø.

retrieve-source

retrieve-source [var]

Retrieves the data from the source block field of the current record and stores it in var. This word does not abort if record# is Ø.

store

store

Stores the data present in the local template in the current record and sets the source block field to the current block. The word store aborts if record# is Ø.

store-l&r

store-l&r

Stores the following three items: the local template to the current record, the variable next-record to the link field, and the variable source-blk to the source field. The word store-l&r aborts if record# is Ø.

store-link

store-link [var]

Stores the value in var to the link field in the current record. The var must contain a valid record# for the current file. This operation does not abort if record# is Ø.

store-source

store-source

Sets the source block field in the current record to the current block as specified by BLK + OFFSET. This word does not abort if record# is Ø.

store-to-field

store-to-field [field] [source]

Stores the data from the specified source to the specified field in the current record. The source and the field must contain an equal number of bytes. The word store-to-field aborts if record# is Ø.

store-to-fields

store-to-fields [field1] [field2] [source]

Stores the data from the source to the range of fields, field1 through field2, in the current record. The source must be the beginning of a set of consecutive variables whose total byte count matches the total of the fields stored. The word store-to-fields aborts if record# is Ø.

~re-init-file

~re-init-file

Assigns all records in the current file to the free list. This operator sets variables first-record and ass/rec to Ø, and fills all record fields (except the link field) with zeros.

~update-header

~update-header

Stores the data present in the local copies of the header variables to the header of the current file.

SMACRO Boolean file-operation words

matches

[var] matches [field]

Searches the current list, testing to see if the contents of the specified var equal the contents of the specified field in each record. This operation returns a true value and sets record# to that record when it finds a match between the two operands. The word matches aborts if record# is Ø.

matches-fields

[var] matches-fields [field1] [field2]

Searches the current list, testing to see if the contents of the specified var equal the contents of the specified range of fields in each record. This operation .returns a true value and sets record# to that record when it finds a match between the operands. The word matches-fields aborts if record# is Ø.

matches-fields-r

[var] matches-fields-r [field1] [field2]

Searches the current list, testing to see if the contents of the specified var equal the contents of the specified range of fields in each record. This operation returns a true value, sets record# to that record, and retrieves the record to the local template when it finds a match between the operands. The word matches-fields-r aborts, if record# is Ø.

matches-r

[var] matches-r [field]

Searches the current list, testing to see if the contents of the specified var equal the contents of the specified field in each record. This operation returns a true value, sets record# to that record, and retrieves the record to the local template when it finds a match between the operands. The word matches-r aborts if record# is Ø.

SMACRO statement words

abort

abort

Aborts the current process and displays the message **ABORTED.**

b-fill

[var] [#bytes] [char] b-fill

Starts at var and fills #bytes with the value of char (an 8-bit ASCII character, not a variable). The parameter var cannot be a subscripted array. (See also w-fill)

blank

[var] [#bytes] blank

Starts at var and fills #bytes with the ASCII space character. The parameter var cannot be a subscripted array.

case

```
case [var1]
case: [var2] [codeA]
case: [var3] [codeB]
.
.
.
case: [var41] [codeC]
default: [default codeD]
end-case
```

Compares the value of var1 to the values of the case variables (var2 through var41) and executes the code following the first variable matching var1. This statement executes the default code if none of the variables match var1. You can use up to 40 cases (the statement compiles as a nested IF-THEN-ELSE comparison so the same maximum nesting limit applies). The operands var1 through var41 must be either integers or sequential variables.

case:

(See case)

default:

(See case)

default-state:

(See state-table)

do

(See while)

else

(See if)

end-case

(See case)

end-do

(See while)

end-repeat

(See repeat)

end-state-table

(See state-table)

endif

(See if)

enter-interrupt

enter-interrupt

Pushes all 8Ø86 registers, except CS, SS, and S; pushes the variable page; sets DS to the SMACRO segment; sets ES to the FORTH segment; and sets the 24-bit page register to page Ø. The enter-interrupt statement does not clear the interrupt mask. You can use this statement to preserve the state of the processor registers at the beginning of an interrupt routine. (See also exit-interrupt)

erase

[var] [#bytes] erase

Starts at var and fills #bytes with Ø. The parameter var cannot be a subscripted array.

exit-interrupt

exit-interrupt

Restores the registers pushed on the stack by enter-interrupt and then executes a return from interrupt (IRET). You can use this statement at the end of an interrupt routine to restore the registers before continuing with normal processing. (See also enter-interrupt)

if

if [boolean exp] then [code] else [code] endif

Provides the standard if-then-else control structure up to a nesting level of 4Ø.

move

[var1] [var2] [#bytes] move

Moves #bytes from var1 to var2. The parameters var1 and var2 cannot be subscripted arrays.

repeat

repeat [code] until [boolean exp] end-repeat

Provides the standard repeat-until control structure up to a nesting level of 4Ø.

S=>

[var1] S=> [var2]

Moves var1 to var2. This statement does not affect the rest of var2 if var1 is shorter than var2. The variables var1 and var2 may be strings, owners, or arrays.

set-bit-in

[var1] set-bit-in [var2]

Sets to 1 the var2 bits corresponding to ones in bit mask var1.

state

(See state-table)

state-table

```
state-table    [state-var1]     [state-var2] . . .  [state-var7]
state:         [test-var1]      [test-var2]  . . .  [test-var7]   [codeA]
state:         [test-var8]      [test-var9]  . . .  [test-var14]  [codeB]
state:         [test-var15]     [test-var16] . . .  [test-var21]  [codeC]
.
.
.
default-state: [default code]
end-state-table
```

Compares the values of all state variables (state-var1 through state-var7) to the values of the test-variables (test-var1 through test-var7) in each line of the state table and executes the code on the first line containing a match for every state variable. Following the match, lines within the state table are ignored. SMACRO executes the default code if none of the lines contains a match for every state variable.

You must use the same number of test variables in each line as there are state variables. You may have up to seven state variables. A test variable can be a variable, a negated variable (using the ~ prefix character to indicate "not-equal"), or XX. XX is a don't-care variable that matches any state variable. Corresponding state and test variables must be of the same type. State tables accept only types strv, iv, bv, and seqv.

State tables cannot be nested. The total number of test variables (not counting XX don't-care variables) must be less than 4Ø.

then

(See if)

until

(See repeat)

w-fill

```
[var] [#words] [integer] w-fill
```

Starts at var and fills #words with the specified integer value. The parameter var cannot be a subscripted array. (See also b-fill) while

while [boolean exp] do [code] end-do

Provides the standard while-do control structure up to a nesting level of 4Ø.

XX

(See state-table)

zero-bit-in

[ivar1] zero-bit-in [ivar2]

Sets to Ø the ivar2 bits corresponding to ones in bit mask ivar1.

~

(See state-table)

~F

~F [FORTH word]

Calls the specified FORTH word.

~PRINT

~PRINT [var]

Displays the value of the specified variable. For an array variable, the entire array is displayed. ~PRINT does not work with a subscripted array.

~PRINT"

~PRINT" [string"]

Displays the specified string.

Miscellaneous SMACRO words

Ø-SMACRO

Ø-SMACRO

Represents board Ø in the system vocabulary. (All boards have even numbers)

1Ø-SMACRO

1Ø-SMACRO

Represents board 1Ø in the system vocabulary. (All boards have even numbers)

12-SMACRO

12-SMACRO

Represents board 12 in the system vocabulary. (All boards have even numbers)

2-SMACRO

2-SMACRO

Represents board 2 in the system vocabulary. (All boards have even numbers)

4-SMACRO

4-SMACRO

Represents board 4 in the system vocabulary. (All boards have even numbers)

6-SMACRO

6-SMACRO

Represents board 6 in the system vocabulary. (All boards have even numbers)

8-SMACRO

8-SMACRO

Represents board 8 in the system vocabulary. (All boards have even numbers)

DEFINITIONS

DEFINITIONS

Enables the current vocabulary. You must precede DEFINITIONS with the name of a vocabulary.

LIST-O

LIST-O [name]

Creates a list owner.

m

m [name]

Adds a variable with the specified name to the most recently created list owner. You must declare this named variable in the same vocabulary as the list owner. (For example, you cannot add a variable from the SDEF vocabulary if the most recently created list owner is in the $DEF vocabulary)

page

page

Contains a copy of the four high-order address bits. Routines that perform address calculations require this FORTH variable because the system does not allow access to the actual address register. To keep the information in this variable current, all code that sets the address register must also set this variable. In particular, an interrupt routine that changes the four high-order bits must first preserve the page variable before making the changes. The interrupt routine must then restore the page variable and reset the address register bits before executing an IRET. (See also enter-interrupt and exit-interrupt in the section "SMACRO statement words")

SDEF

SDEF

Sets the system vocabulary.

~INTERRUPT

[interrupt#] ~INTERRUPT [routine]

Sets the specified interrupt# to call the specified routine.

COMM Words

buffer-ready-f

buffer-ready-f

Indicates when the buffer is ready. You must include this integer variable in every vocabulary that COMM accesses.

COMM

COMM

Loads the communications utility.

COMM-PROCESS

COMM-PROCESS

The routine that performs the communications process. You must include this routine in the board process on the COMM board.

control-cycle-#-clks

control-cycle-#-clks
Contains the user-defined number of counts for the communications timer based on a 153-KHz counter cycle. The default value for this SMACRO variable is 13ØØ (hex). For debugging, you can change this value to 7FFF to obtain the longest possible interval between cycles.

ERASE-COMM-TABLE

ERASE-COMM-TABLE

Erases all entries from the communications table of a board, thus deleting all communication paths between control levels.

RESTART-COMM-TIMER

RESTART-COMM-TIMER

Restarts the communications timer (timer Ø on the COMM board).

SHOW-TABLE

SHOW-TABLE

Displays the contents of the current communications table. The first column contains unimportant information which you can ignore. Each line in the table represents either a transfer path from an output buffer to common memory or a transfer path from common memory to an input buffer on another level. Thus, two lines comprise a complete transfer path.

TO-DESTINATION

(See TRANSFER-FROM)

TRANSFER-FROM

TRANSFER-FROM [vocabulary buffer] TO-DESTINATION [vocabulary buffer]

Enters a buffer transfer into the communications table of a board, thus adding a communication path between control levels.

WAIT-FOR-NEXT-COMM-CYCLE

WAIT-FOR-NEXT-COMM-CYCLE

Delays program execution until the communications timer begins the next communications cycle. You can optionally include this statement in the board process on the COMM board.

B.2 RSL WORDS

The following list describes the user words defined in RSL. All block references are absolute, unless otherwise indicated. The RSL words fall into the following categories:

- RSL operating system words
- Language words
- TASK command words
- RSL system parameter words
- Control level words

RSL Operating System Words

$DEF

```
$DEF
```

Enables all levels to load their file definitions from the same source block. Though the file definitions come from the same source block, $DEF declares them in the correct vocabulary. You must define this word on all control levels to set their vocabularies.

?ARR

```
?ARR [array name]
```

Displays the following information for the specified array: record number, absolute source block number, number of sectors for each dimension, movetable name for each dimension, sector pose name, and current sector list. ?ARR sets the editor default block to the last block number displayed.

?LOC

```
?LOC [location name]
```

Displays the following information for the specified location: record number, absolute source block number, pose name, movetable name, and a matrix representing the resulting pose. ?LOC sets the editor default block to the last block number displayed.

?MOVE-TO

?MOVE-TO

Displays the following information for all currently defined move-to paths: record number, absolute source block number, path-^, object, start location, and destination location. ?MOVE-TO pauses after displaying the information for each move-to path and waits for you to press any key before displaying the information for the next move-to path. You can press ^C to abort the operation of ?MOVE-TO before it displays all current move-to paths. ?MOVE-TO sets the editor default block to the last block number displayed.

?MTB

?MTB [movetable name]

Displays the specified movetable, its record number, and its absolute source block number. ?MTB sets the editor default block to the last block number displayed.

?OBJ

?OBJ [object name] [grip#]

Displays the following information for the specified grip# on the specified object: OBJECT-NAME-FILE record number, OBJ-GRIP-FILE record number, absolute source block number for the object grip, and name of the grip movetable. ?OBJ sets the editor default block to the last block number displayed.

?POSE

?POSE [pose name]

Displays a matrix representation of the specified pose and its configuration flags, record number, and absolute source block number. ?POSE sets the editor default block to the last block number displayed.

ABT

ABT [level name]

Aborts the control process of the specified level by setting the abort-f variable to true. The level can be TASK, PATH, PRIM, or JOINT.

GO

GO

Starts the execution of all control level and communications processes and issues the RESTART command to the TASK level by loading block 8921. GO prompts you to initialize the robot interface.

HALT

HALT

Stops the execution of all control-level processes, the communications process, and the robot interface by loading block 8924. HALT clears all slave-board output buffers, sets offsets and prepares the system to execute the next GO command it receives. Block 8925 performs the same functions as block 8924 except that block 8925 leaves the robot interface active so that you can use the TT command. (See also TT)

Joint/comm

Joint/comm

Specifies the board segment of the JOINT/COMM board. You use this FORTH constant with remote-slave words described in the section "Remote-slave words".

POFF

POFF [level name]

Turns off the debug print facility for the specified control level by setting the print-f variable to false. The level can be TASK, PATH, PRIM, or JOINT. (See also PON)

PON

PON [level name]

Turns on the debug print facility for the specified control level by setting the print-f variable to true. The level can be TASK, PATH, PRIM, or JOINT. (See also POFF)

Prim

Prim

Specifies the board segment of the PRIM board. You use this FORTH constant with remote-slave words described in the section "Remote-slave words".

RECORD-POSE

RECORD-POSE [pose name]

Stores the current robot pose in the named pose variable. If the pose name is not yet defined, RECORD-POSE defines it.

Task/path

Task/path

Specifies the board segment of the TASK&PATH board. You use this FORTH constant with remote-slave words described in the section "Remote-slave Words".

TT

TT

Executes one cycle of the full system and displays debug output for all control levels. You must edit block 8923 to activate or deactivate the robot interface during the operation of TT. You can edit block 919 on each level to display the additional debug output. You must precede TT with the HALT command if the robot interface is to be inactive. Alternatively, you must precede TT with 8925 LOAD if the robot interface is to be active. (See also HALT)

TTT

[#cycles] TTT

Performs the TT single-step function a number of times equal to #cycles. (See also TT)

TYPE-POSE

[block#] [line#] TYPE-POSE [pose name]

Enters source code for the specified pose into the specified block, beginning at the specified line.

(See -mtb-)

-arr-

```
-arr-[array name] [base location] [x-#-sectors] [x-movetable]
     [y-#-sectors] [y-movetable] [z-#-sectors] [z-movetable]
     [sector pose]
```

Defines an array with the specified name and dimensions.

Where:

- array name contains the name of the array.

- base location specifies the location that is the origin (sector Ø) of the array.

- x-#-sectors, y-#-sectors, and z-#-sectors specify the number of sectors along the x, y, and z axes of the array frame. Currently, the total number of sectors must be less than or equal to 2Ø. The size of sector-list limits the total number of statements. You can increase the size of sector-list.

- x-movetable, y-movetable, and z-movetable contain the names of the movetables that give the displacements between sectors along the x, y, and z axes of the array frame. The movetables should consist only of tool frame motions. The array retains its shape if you move the location.

- sector pose contains the name of the pose RSL uses to store the position of the current sector during execution.

-imtb-

```
-imtb- [inverse name] [source name]
```

Defines an inverse movetable with the specified name based on the specified source movetable.

Where:

- inverse name contains the name for the new movetable.

- source name contains the name of the original movetable.

-loc-

```
-loc- [location name] [pose] [movetable]
```

Defines a location with the specified name. The specified pose and movetable combine to define the new location.

Where:

- location-name contains the location name.

- pose contains the name of the pose.

- movetable contains the name of the movetable.

-mtb-

```
-mtb- [name]

    --- [frame] [X] [Y] [Z]
    --- [r-tool] [axis] [angle]
    .
    .
    .
    -mtb-end-
```

Defines a movetable with the specified name. There are two parts to the movetable syntax: the name line (-mtb-) and the sequence of transform lines (---) terminated by -mtb-end-. Each transform line specifies either a vector translation or a rotation. You specify translations by giving the x, y, and z components of the vector (expressed in either the base or tool frame). You specify rotations by giving the angle and the axis (x, y, or z) of the tool frame. The unit of measurement for translations is inches. For rotations the unit is degrees, within the range +180.00 to -179.99.

Where:

- name contains the movetable name.

- frame contains either t-base or t-tool to specify the base frame or the tool frame, respectively.

- X contains the x component of the translation vector.

- Y contains the y component of the translation vector.

- Z contains the z component of the translation vector.

- r-tool specifies a rotation about the tool frame.

- axis contains X, Y, or Z to specify the axis about which to rotate.

- angle contains the angle to rotate in degrees.

-mtb-end-

(See -mtb-)

-obj-

-obj- [object phrase] [grip movetable]

Defines an object grip with the specified name. The grip# specifies the grip position on the object and the specified grip movetable defines the grip position for the robot.

Where:

- object phrase contains the object name and grip number.

- grip-mtb contains the name of the movetable that defines the grip position.

-path-

-path- [path type] [object phrase] [location phrase]

or,

-path- move-to [object phrase] [start location phrase] [destination location phrase]

The path statement syntax contains two parts: the path statement and a sequence of path-point lines. The -path- line defines the type and the parameters of the path; the -ppt- lines give the sequence of steps in the algorithm. See -ppt- for the path-point syntax.

Where:

- path type is one of the RSL path types: approach-pickup, depart-pickup, approach-release, or depart-release.

- object phrase contains the object name and the grip number for this path.

- location phrase contains the type of goal (loc or arr) and the name that defines the goal of this path.

- start location phrase contains the location type (loc or arr) and the start location name.

- destination location phrase contains the location type (loc or arr) and the destination location name.

### -pose-

-pose- [name] [pose numbers] [configuration flags]

Defines a pose with the specified name. The definition includes pose numbers and configuration flags. The pose numbers are the seven floating-point numbers that specify a quaternion and a vector. Refer to the robot transform documentation for information on the three configuration flags.

Where:

- name contains the name of the pose.

- pose numbers contains the seven floating point numbers for the quaternion and vector that represent the pose.

- configuration flags contains the three configuration flags, usually +1.Ø or -1.Ø. See your robot transform documentation for details.

-ppt-

-ppt- goto [location phrase] [trajectory phrase]
Adds a goto path-point to the current path.

Where:

- location phrase contains the type (loc, arr, goal, or tool) and the name of the goal location.

- trajectory phrase contains the type of trajectory (cart or joint) and the trajectory parameters. (For descriptions of the trajectory parameters, see cart and joint)

-rr-

-rr- [name] [file] [size]

Defines a round-robin with the specified name and size in the specified file.

Where:

- name contains the name of the round-robin.

- file contains the name of the round-robin file.

- size specifies the number of records in the round-robin file.

approach-pickup

(See -path-)

approach-release

(See -path-)

arr

arr [name]

Indicates an array in a location phrase. (See -path- and -ppt)

cart

cart [translational max acceleration] [translational max velocity]
[translational neighborhood] [rotational max acceleration]
[rotational max velocity] [rotational neighborhood]

Indicates a cartesian trajectory in a path-point trajectory phrase. When the trajectory routine reports done, the robot is within both the translational and rotational neighborhood of the goal.

Where:

- translational max acceleration contains the maximum allowed magnitude of the translational acceleration, in inches per cycle per cycle.

- translational max velocity contains the maximum allowed magnitude of the translational velocity, in inches per cycle.

- translational neighborhood contains the translational distance in inches from the current goal location at which the trajectory routine reports done.

- rotational max acceleration contains the maximum allowed magnitude of the rotational acceleration in degrees per cycle per cycle.

- rotational max velocity contains the maximum allowed magnitude of the rotational velocity, in degrees per cycle.

- rotational neighborhood contains the rotational distance in degrees from the current goal location at which the trajectory routine reports done.

depart-pickup

(See -path-)

depart-release

(See -path-)

goal

goal [name]

Indicates a goal movetable in a path-point loc phrase. (See -ppt-)

goto

(See -ppt-)

joint

joint [acceleration] [velocity] [delta]
Indicates a joint trajectory in a path-point traj phrase.

Where:

- acceleration contains the maximum allowed acceleration of any joint given as the percentage (between Ø and 1ØØ) of the hardware maximum for each joint.

- velocity contains the maximum allowed velocity of any joint given as the percentage (between Ø and 1ØØ) of the hardware maximum for each joint.

- delta contains the difference in degrees from the location joint values at which the trajectory routine reports done. When the trajectory reports done, all joint values are within this delta.

loc

loc [name]

Indicates a location in a location phrase. (See -path- and -ppt-)

move-to

(See -path-)

nul

nul

The name of a predefined object, location, and movetable. Use nul to specify a nul value in the definition of paths and path-points.

r-tool

(See -mtb-)

t-base

(See -mtb-)

t-tool

(See -mtb-)

tool

tool [name]

Indicates a tool movetable in a path-point location phrase. (See -ppt-)

TASK Command Words

MOVE-TO

MOVE-TO [object] [grip#] [destination location type] [destination] [array sector]

Sends a move-to path to the PATH level. MOVE-TO uses the current location as the start location. The array sector specification is optional, you need to specify the array sector only if the destination location type is arr (indicating an array).

MOVE-TO updates the current location to the path goal when PATH completes the path. MOVE-TO reports an error if it cannot find a specified path. After you issue MOVE-TO, the TASK level reports its status as executing until the PATH level reports its status as done. Then the TASK level also reports its status as done.

PAUSE

PAUSE

Sends PAUSE to the PATH level and sets the current location and object to nul. After you issue PAUSE, the TASK level reports its status as executing until the PATH level reports its status as done. Then the TASK level also reports its status as done.

RESTART

RESTART

Sends the RESTART command to the PATH level and sets the current location and object to nul. After you issue RESTART, the TASK level reports its status as executing until the PATH level reports its status as done. Then the TASK level also reports its status as done.

TRANSFER

TRANSFER [object phrase] [source location phrase] [source sector list] [destination location phrase] [destination sector list]

Sends the following sequence of path commands to the PATH level:

1. move-to object source destination
2. approach-grasp object source
3. depart-grasp object source
4. move-to object source destination
5. approach-release object destination
6. depart-release object destination

RSL repeats the specified sequence of paths for each sector in the sector list if the source or the destination consists of an array. If both the source and the destination are arrays, the sector lists must have the same lengths. RSL updates the current location as it completes the execution of each path.

After you issue TRANSFER, the TASK level reports its status as executing until the PATH level indicates that all paths are complete by reporting its status as done. Then the TASK level also reports its status as done.

RSL System Parameter Words

The following lists describe parameters that are associated with the JOINT, PATH, and PRIM levels and with RSL.

JOINT parameters

These variables are set in the range of blocks 980 to 989 of the JOINT level. The auto-load block of the JOINT D>M loads block 980.

a-sysmax

a-sysmax

Contains the maximum acceleration for each joint of the robot. This 6-element array describes acceleration in inches per cycle squared or radians per cycle squared.

creep-delta

creep-delta

Contains the distance each joint is to move, relative to a joint limit. The CREEP command uses this 6-element floating-point array to control the movement of the joints.

creep-vel

creep-vel

Contains the velocity each joint is to use to reach the distance specified by creep-delta. The CREEP command uses this 6-element floating-point array to control the movement of the joints.

lj-limit

lj-limit

Contains the lower joint limits for each joint of the robot. This 6-element array describes lower joint limits in inches or radians. The JOINT-LIMIT-TEST routine uses these same limits to test for valid joint values. You must edit the JOINT-LIMIT-TEST routine if you want this routine to use other limit criteria.

next-point-scale-threshold

next-point-scale-threshold

Determines when the CARTESIAN command is to report next-point status. The next-point-scale-threshold variable has the value Ø.5 for RSL operation.

uj-limit

uj-limit

Contains the upper joint limits for each joint of the robot. This 6-element array describes upper joint limits in inches or radians. The JOINT-LIMIT-TEST routine uses these same limits to test for valid joint values. You must edit the JOINT-LIMIT-TEST routine if you want this routine to use other limit criteria.

v-sysmax

v-sysmax

Contains the maximum velocity for each joint of the robot. This 6-element array describes the velocity in inches per cycle or in radians per cycle.

PATH parameters

This variable is set in the range of blocks 980 to 989 of the PATH level. The auto-load block of the PATH D>M loads block 980.

tp-cycles

tp-cycles

Specifies the delay (in number of cycles) between the tool-pose feedback from the JOINT level and a robot's achievement of that pose. This variable has a value of 2 when you are using RSL with VAL SLAVE. The TOOL-POSE-^ routine uses this variable to set tool-pose-^.

PRIM parameters

These variables are set in the range of blocks 980 to 989 of the PRIM level. The auto-load block of the PRIM D>M loads block 980.

joint-traj-delay

joint-traj-delay

Specifies the delay (using a floating-point number of cycles) between issuing a command to, and the feedback pose from, the JOINT level.

joy-max-v

joy-max-v

Specifies the maximum translational velocity (using a floating-point number of inches per cycle) that you can obtain using the joystick. The velocity switch on the joystick controller scales velocities down from this maximum value.

joy-max-w

joy-max-w

Specifies the maximum rotational velocity (using a floating-point number of radians per cycle) that you can obtain using the joystick. The velocity switch on the joystick controller scales velocities down from this maximum value.

RSL parameters

INV-TOOL-MTB

INV-TOOL-MTB

Defines an inverse of the tool point with respect to the robot wrist. You can define INV-TOOL-MTB as the inverse of TOOL-MTB by entering the sequence -imtb- INV-TOOL-MTB TOOL-MTB.

TOOL-MTB

TOOL-MTB

Defines the tool point with respect to the robot wrist. You can redefine this movetable to change the tool point; however, you must also redefine INV-TOOL-MTB at the same time.

Control-Level Words

abort-f

abort-f

Aborts control-level execution when this flag is true. (See also ABT under "RSL Operating System Words")

cycle-time

cycle-time

Records the number of 153-KHz clock ticks required for the levels to perform the current execution cycle. You can use the PRINT-TIME routine to convert clock ticks to microseconds.

Display

Enables you to display user-defined debug information. Display is a part of the board process that runs after the level executes its routines. You can turn the printing on with PON and turn the printing off with POFF. (See also PON and POFF under "RSL Operating System Words")

error-list

error-list

Contains the list of status-arg values. You can execute this sequential variable owner in Show mode to display the names for the status-arg values.

inc-command-#-in

Indicates that a level has received a new command. You must increment this counter after setting the input command for a control level.

JS

JS [joint array]

Displays the joint angles, joint velocities, or joint accelerations contained in the specified joint array. You must edit this routine to accommodate your specific robot. For example, if you are using a PUMA 76Ø robot, enter JS servo-com-joint to display the joint values (in degrees) commanded to the robot servo level. JS is defined only for the JOINT level.

List-display

List-display

Contains user-specified variables that are useful for debugging. Executing List-display in Show mode displays the values of these variables.

max-cycle-time

max-cycle-time

Records the largest number of 153-KHz clock ticks required to complete an execution cycle since you issued RESTART. You can use the PRINT-TIME routine to convert clock ticks to microseconds.

min-cycle-time

min-cycle-time

Records the smallest number of 153-KHz clock ticks required to complete an execution cycle since you issued RESTART. You can use the PRINT-TIME routine to convert clock ticks to microseconds.

overflow-cycle

overflow-cycle

Contains the number of cycles since you issued RESTART in which the control level required longer than a cycle to complete its execution.

PDEF

PDEF

Sets the vocabulary for the PATH level. PDEF is defined only on the TASK/PATH board.

print-f

print-f

Controls debug printing. PON sets this flag true and POFF sets this flag false. (See also PON and POFF under "RSL Operating System Words")

TDEF

TDEF

Sets the vocabulary for the TASK level. TDEF is defined only on the TASK/PATH board.

WAIT-TASK-DONE

WAIT-TASK-DONE

Waits for the status-report variable to indicate executing (that is, not done) and then waits for the status-report variable to indicate done. You can use this word in a load block to issue a sequence of commands to the TASK level. For this situation, WAIT-TASK-DONE issues each successive command only after the previous command indicates done. You can use ABT TASK to abort the operation of WAIT-TASK-DONE before all commands in the sequence execute.

Appendix C
SYSTEM MAPS

This appendix describes how RCS (with RSL version 1.6) allocates the MULTIBUS address space, I/O space, interrupt assignments, and timers.

C.1 MULTIBUS ADDRESS SPACE

The following list describes the allocation of memory for RCS and RSL version 1.6.

Address Range	Contents
ØØØØØ-1FFFF	RSL processor
2ØØØØ-3FFFF	PRIM processor
4ØØØØ-5FFFF	TASK/PATH processor
6ØØØØ-7FFFF	JOINT/COMM processor
8ØØØØ-EFFFF	Common memory
8ØØØØ-8FFFF	User files
9ØØØØ-DFFFF	System dictionary
EØØØØ-EEFFF	RCS communication buffer
EFØØØ-EFBFF	Unused address space
EFCØØ-EFC1F	Disk control
EFC2Ø-EFC7F	Remote/slave board semaphores
EFC8Ø-EFFFF	Unused address space
FØØØØ-FBFFF	Unused address space
FCØØØ-FFFFF	FORTH PROMs

The unused address space shown in this list is available for user applications. You can use 24-bit addressing to increase the address space if your MULTIBUS backplane includes the P2 connector option. This option enables you to relocate the user files and the system dictionary in memory, and then add up to three more processes in the address range 8ØØØØ-DFFFF. (See the dpage and fpage words for information on allocating memory to the system dictionary and SMACRO files.)

C.2 MULTIBUS I/O SPACE ORGANIZATION

The following list describes the allocation of the I/O space for RCS.

I/O Address	Contents
ØØØØ-ØØ4F	Unused I/O space
ØØ5Ø-ØØ51	Unused I/O space
ØØ52-ØØ53	Rimfire 45 disk and tape controller
ØØ54-ØØ5F	Unused I/O space
ØØ6Ø-ØØ7F	Unused I/O space
ØØ8Ø-ØØ9F	J4 iSBX connector on 86/3Ø
ØØAØ-ØØBF	J3 iSBX connector on 86/3Ø
ØØCØ-ØØDF	Other 86/3Ø ports
ØØEØ-ØØFF	Unused I/O space

All unused I/O address space and the J3 and J4 ports are available for user applications.

C.3 RCS INTERRUPT ASSIGNMENTS

The following list describes the assignments of the interrupts that affect the operation of RCS.

Interrupt Number	Function
Ø	6 millisecond time-out interrupt
1	8Ø87 error interrupt
2	Communication bit input (not used on the board with the communications process)
3,4	On-board serial port (not required on slave boards)
5,6,7	Unused interrupts

C.4 TIMERS

Each 86/3Ø board has three programmable timers, numbered Ø, 1, and 2. Timer Ø, on the COMM board, is used by the COMM-PROCESS. Timer Ø on other boards is not used. Timer 1 is used by the debug routines TIME-P, START-TIMER, and READ-TIMER. Timer 2 is used for the on-board serial port. (Note that this timer can be used on slave boards that do not use the serial port.)

Appendix D
RCS DISK BLOCK ORGANIZATION

RCS uses the FORTH block structure for all source code. The disk map below shows how RCS allocates blocks on the Winchester disk for RSL version 1.6. (Block numbers begin at 1000 for historical reasons.)

Absolute Block Numbers	Contents
1000-1349	MBOOT
1350-1499	CUSTOM
1350-1359	Unused blocks
1360-1363	MAP
1364	BACKUP
1365-1369	Other tape blocks
1370-1383	CUSTOM setup blocks
1384	CUSTOM? map
1385-1397	DM? maps
1398	PRESERVE-FILE map
1399	PRESERVE map (PMAP)
1400-1401	Directory blocks
1402	System addresses
1403	Init-cm block
1404	Slave board names
1405	Function-keys block
1406-1412	System vocabulary declaration
1413-1419	Additional function key blocks
1420-1459	auto-load blocks
1460-1469	BOOT-SYSTEM blocks
1470-1479	Load base system
1480-1489	Load application
1490-1499	BOOT-BASE-SYSTEM blocks
1500-1599	Screen editor
1600-5999	D>M images
6000-6999	Reserved space for tape transfer
7000-7999	RCS source code
8000-8999	RSL source code
9000-9999	TASK source code
10000-10999	PATH source code
11000-11999	PRIM source code
12000-12999	JOINT source code
13000-30999	Available to user

In addition to the RCS blocks, the disk map includes blocks 8000-12999, which are allocated for RSL. When you use RSL, block 3 (relative to the first block in the level) is a load block that loads the control-level software. Similarly, relative block 980 initializes the control level.

RCS uses a 10-block directory structure within these source-code blocks. The system does not automatically maintain this directory information; you must update the block-directory blocks when you add, delete, or move blocks. For more information on directories, see "Understanding Directory Block Conventions" in Chapter 6, "Basic RCS Operations".

Appendix E
8Ø87 OPERATION CODES FOR RCS

This appendix lists the 8Ø87 operation codes and their RCS equivalents in alphabetical order based on the Intel Operation Codes (as in Table S-19 in the Intel 8Ø86 Family User's Manual). RCS does not implement the complete 8Ø87 instruction set.

Because FORTH uses only the length and the first three characters to differentiate the names of variables, NBS has changed the Intel names of some operation codes to ensure that the name of each code is unique.

As with the Intel assembler, unless the second letter is N, RCS assembles all operation codes with an FWAIT first.

RCS provides the following words to modify operation code mnemonics:

- d sets the destination bit.
- LNG sets the temp-real format required to use the FLD, FST, and FSTP operation codes.
- P indicates that the 8Ø87 is to pop the result (the same as the Intel P suffix on the instruction mnemonic).
- rev indicates that the 8Ø87 is to perform the indicated operation with the source and destination operands reversed (the same as the Intel R suffix on the instruction mnemonic).
- INT indicates that the operation is to use 32-bit integer arithmetic.
- WINT indicates that the operation is to use 16-bit integer arithmetic.

The alphabetical list of Intel operation codes and their RCS equivalents follows.

Operation	Intel	RCS	Modifier Words
Calculate $2^{X}-1$	F2XM1	F2XM1	
Convert to absolute value	FABS	FABS	
Add real	FADD	FADD	
Add real and pop	FADDP	FADD	P
Load packed decimal	FBLD	none	
Store packed decimal and pop	FBSTP	none	
Change sign	FCHS	FCHS	
Clear exceptions	FCLEX	none	
Compare real	FCOM	FCOM	
Compare real and pop	FCOMP	FCOMP	
Compare real and pop twice	FCOMPP	FCOMPP	
Decrement stack pointer	FDECSTP	FDECSTP	
Disable interrupts	FDISI	none	

Operation	Intel	RCS	Modifier Words
Divide real	FDIV	FDIV	
Divide real and pop	FDIVP	FDIV	P
Divide real reversed	FDIVR	FDIV	rev
Divide real reversed and pop	FDIVRP	FDIV	rev P
Enable interrupts	FENI	none	
Free register	FFREE	FFREE	
Add integer	FIADD	FADD	INT or WINT
Compare integer	FICOM	FCOM	INT or WINT
Compare integer and pop	FICOMP	FCOMP	INT or WINT
Divide integer	FIDIV	FDIV	INT or WINT
Divide integer reversed	FIDIVR	FDIV	rev INT or rev WINT
Load integer	FILD	FLD	INT or WINT
Multiply integer	FIMUL	FMUL	INT or WINT
Increment stack pointer	FINCSTP	FINCSTPT	
Initialize 8Ø87	FINIT	none	
Store integer	FIST	FST	INT or WINT
Store integer and pop	FISTP	FST	P INT or P WINT
Subtract integer	FISUB	FSUB	INT or WINT
Subtract integer reversed	FISUBR	FSUB	rev INT or rev WINT
Load real	FLD	FLD	
Load +1.Ø	FLD1	1FLD	
Load control word	FLDCW	FLDCW	
Load environment	FLDENV	FLDENVIR	
Load $\log_2 e$	FLDL2E	L2EFLD	
Load $\log_2 10$	FLDL2T	L2TFLD	
Load $\log_{10} 2$	FLDLG2	LT2FLD	
Load $\log_e 2$	FLDLN2	LE2FLD	
Load pi	FLDPI	PIFLD	
Load +Ø.Ø	FLDZ	ZFLD	
Multiply real	FMUL	FMUL	
Multiply real and pop	FMULP	FMUL	P
Clear exceptions (no FWAIT)	FNCLEX	FNCLEX	
Disable interrupts (no FWAIT)	FNDISI	FNDISI	
Enable interrupts (no FWAIT)	FNENI	FNENI	
Initialize 8Ø87 (no FWAIT)	FNINIT	FNINIT	
No operation	FNOP	none	
Save state (no FWAIT)	FNSAVE	none	
Store control word (no FWAIT)	FNSTCW	none	
Store environment (no FWAIT)	FNSTENV	FNSTENVIR	
Store status word (no FWAIT)	FNSTSW	FNSTSW	
Calculate partial arctangent	FPATAN	FPATAN	
Calculate partial remainder	FPREM	FPREM	
Calculate partial tangent	FPTAN	FPTAN	
Round to integer	FRNDINT	FRNDINT	
Restore saved state	FRSTOR	FRSTOR	
Save state	FSAVE	FSAVE	
Scale	FSCALE	FSCALE	
Calculate square root	FSQRT	FSQRT	
Store real	FST	FST	
Store control word	FSTCW	FSTCW	
Store environment	FSTENV	none	

Operation	Intel	RCS	Modifier Words
Store real and pop	FSTP	FSTP	
Store status word	FSTSW	none	
Subtract real	FSUB	FSUB	
Subtract real and pop	FSUBP	FSUB	P
Subtract real reversed	FSUBR	FSUB	rev
Subtract real reversed and pop	FSUBRP	FSUB	rev P
Test stack top against +Ø.Ø	FTST	FTST	
Make 8Ø86 wait while 8Ø87 is busy	FWAIT	FWAIT	
Examine stack top	FXAM	FXAM	
Exchange registers	FXCH	FXCH	
Extract exponent and significand	FXTRACT	FXTRACT	
Multiply Y and $\log_2 X$	FYL2X	FYL2X	
Multiply Y and $\log_2(X+1)$	FYL2XP1	FYL2XP1	

Appendix F
JOYSTICK SCHEMATICS

This appendix contains schematic wiring diagrams describing the design of a joystick and the interface electronics housed in a separate chassis. The joystick contains 4 smaller joysticks, 15 switches, a velocity-control knob, and an indicator light. The joystick interfaces with the RSL PRIM level.

The following table lists the schematics.

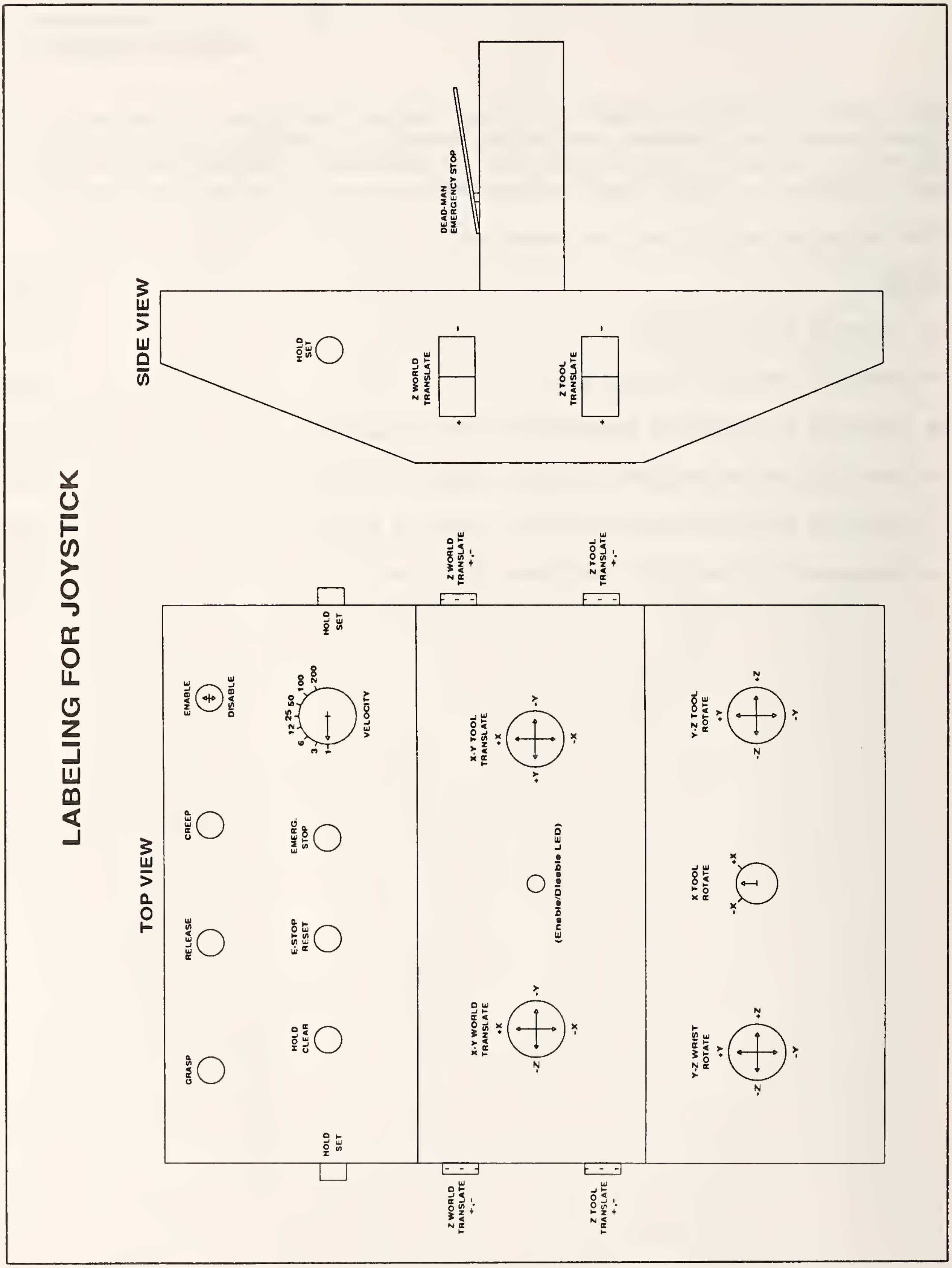


Figure F-1. Joystick switch layout.

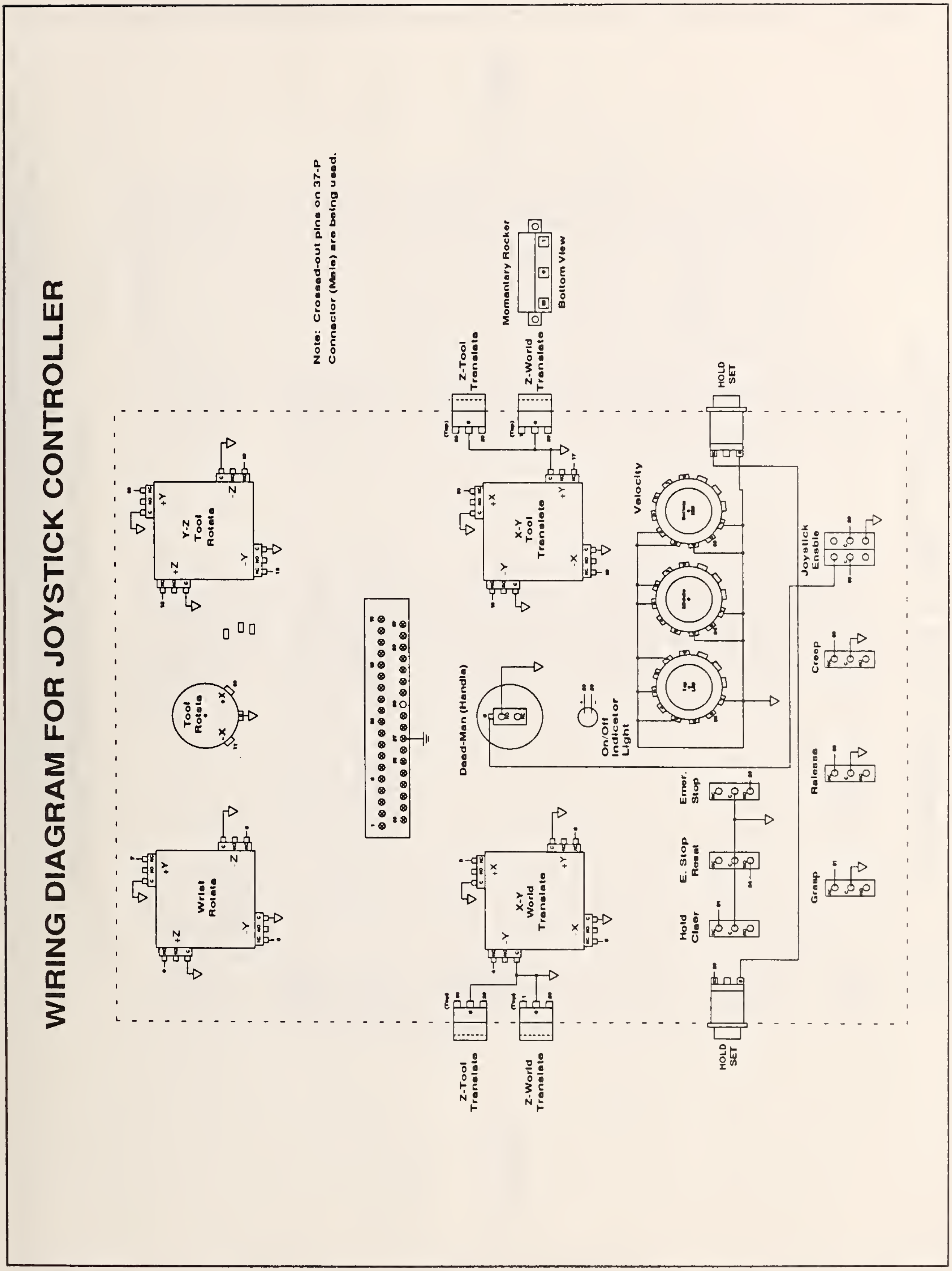


Figure F-2. Joystick switch wiring diagram.

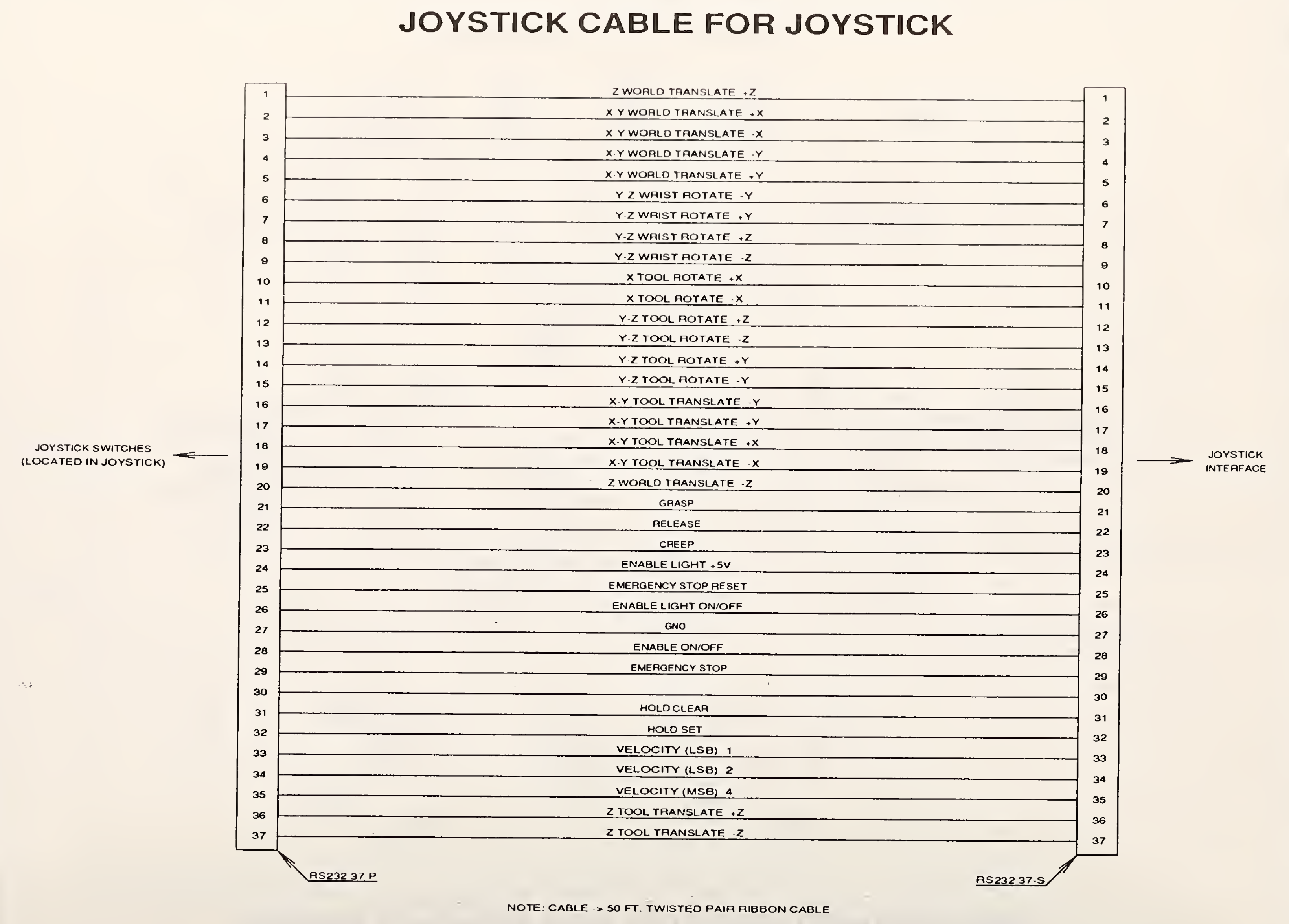


Figure F-3. Joystick to interface electronics cable diagram.

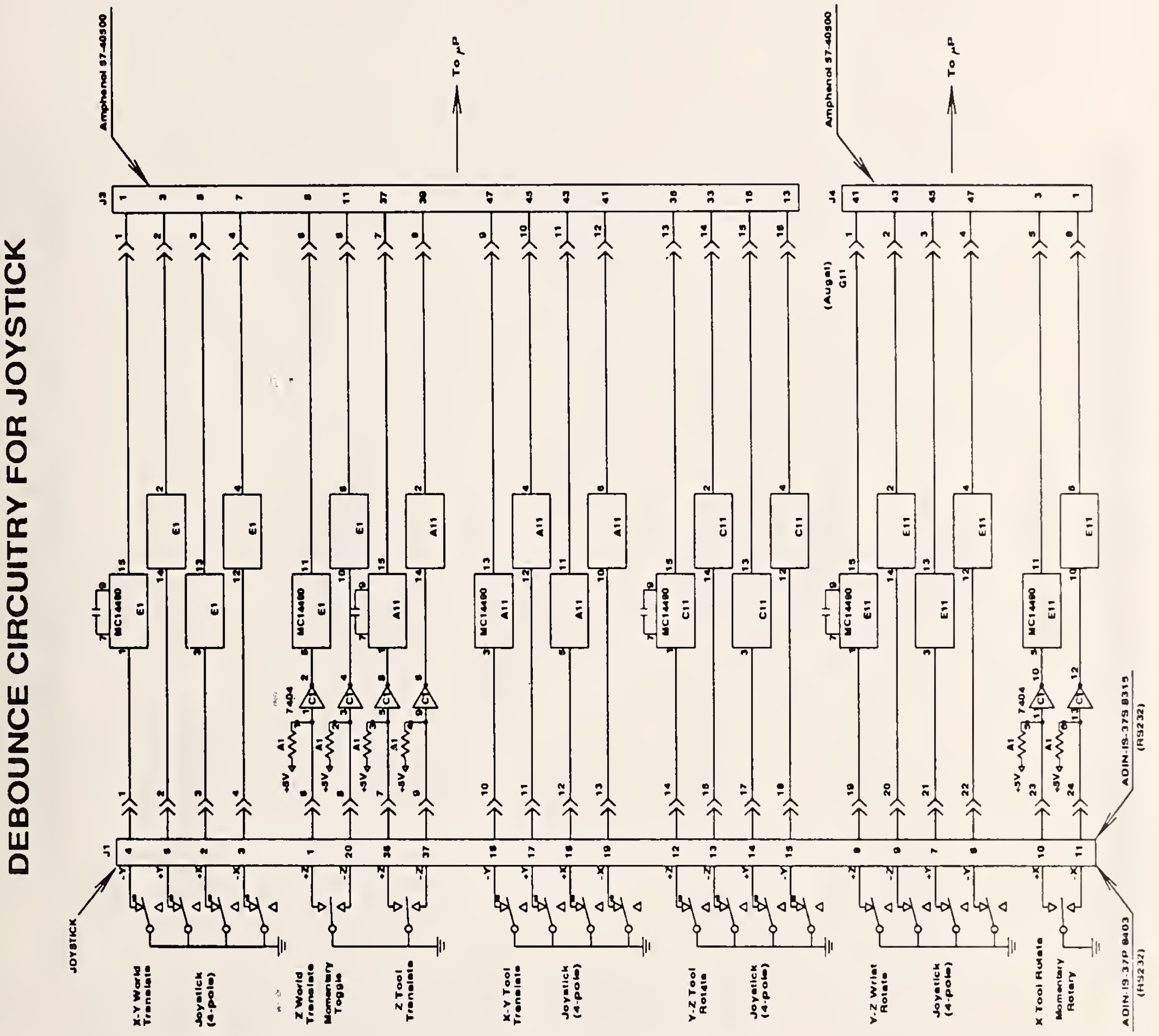


Figure F-4(a). Joystick switch debounce circuitry (page 1 of 2)

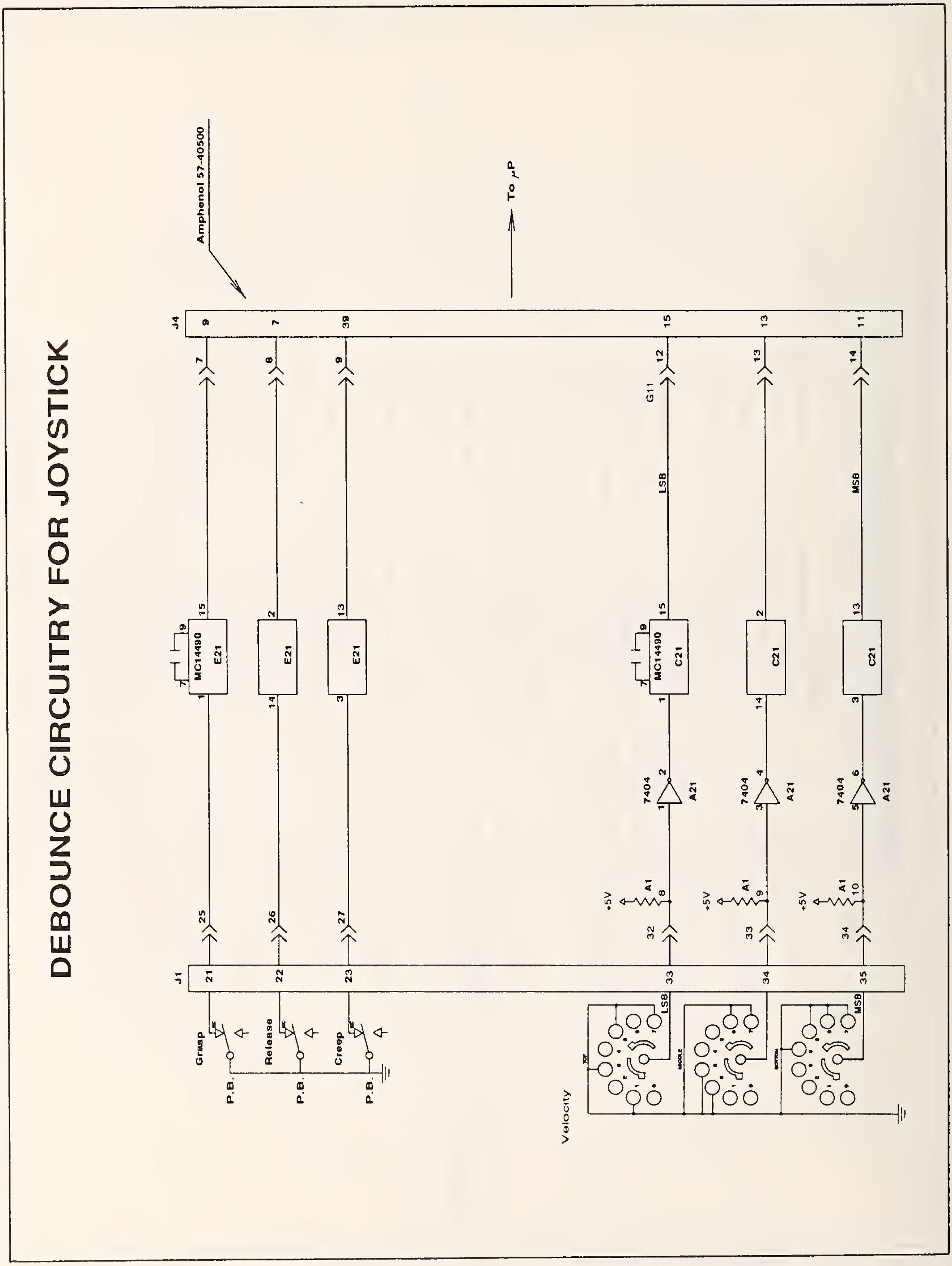


Figure F-4(b). Joystick switch debounce circuitry (page 2 of 2)

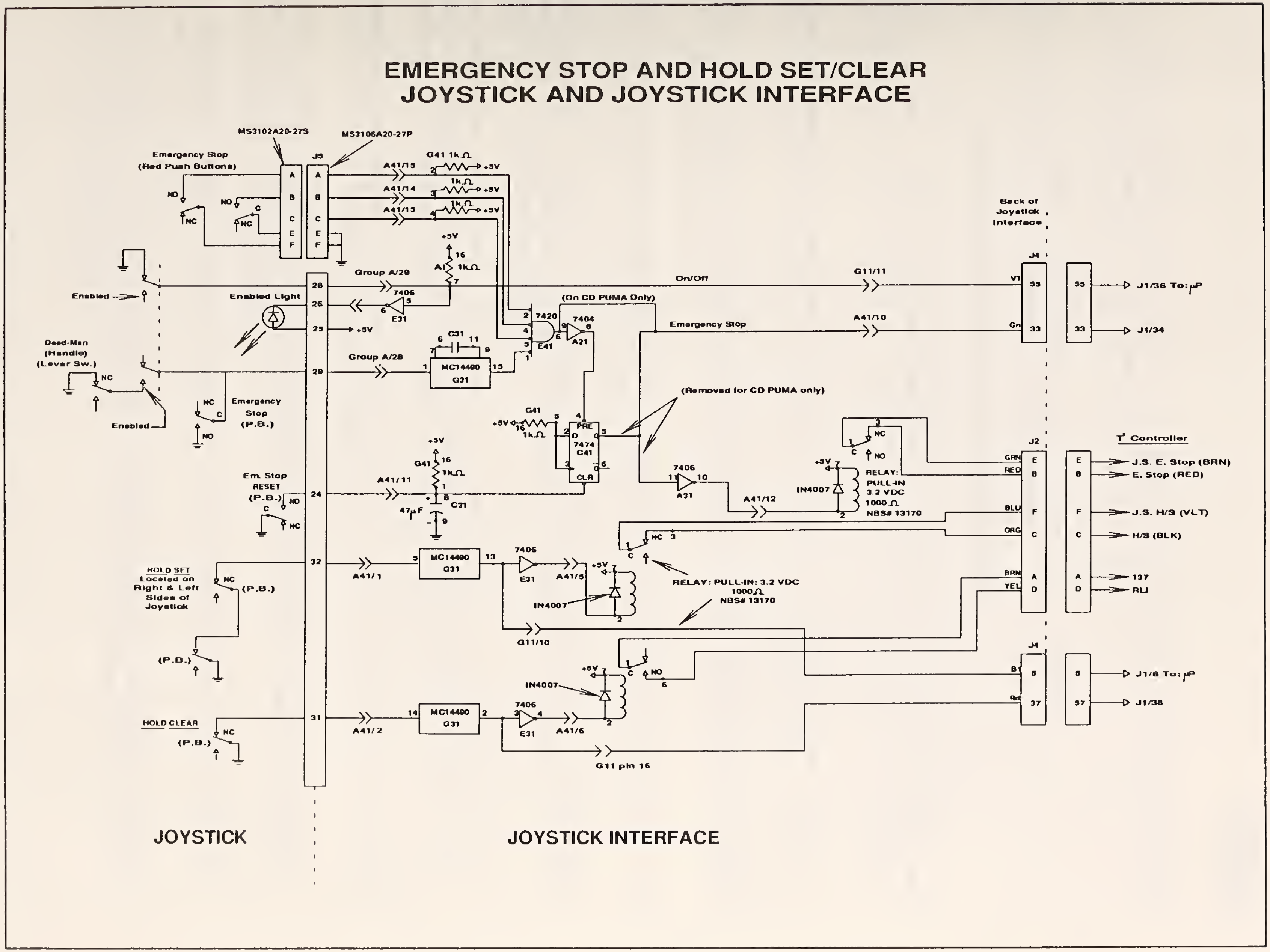

EMERGENCY STOP AND HOLD SET/CLEAR
JOYSTICK AND JOYSTICK INTERFACE

Figure F-5. Emergency stop and hold set/clear circuitry.

This glossary defines terms that are unique to or have unique meanings in RCS or the RSL application. RSL terms are differentiated from RCS terms by placing RSL in parentheses after the term.

Absolute block number. The number used to specify the storage location of a block on the disk. See Relative block number.

AMRF. See Automated Manufacturing Research Facility.

Array. (RSL) An RSL data structure that contains a group of locations. An array is one-, two-, or three-dimensional. Movetables define the displacement between sectors of the array. Arrays are stored in an array file in common memory.

Automated Manufacturing Research Facility (AMRF). An experimental factory developed by NBS to operate as a small, totally automated, batch machine shop.

Block. A division of the disk containing 1,Ø24 bytes, arranged in 16 lines of 64 characters each. The entire disk is organized into blocks, each of which has a unique block number.

COMM. An RCS utility that provides a communication protocol for passing commands and status information between control levels.

Common memory. Random-access memory in a central location that all processor boards in the system use to store the system dictionary, SMACRO files, and command and status information.

Communications dead time. The communications interval during which the contents of command, status, and data buffers and flags in common memory are changed. During this time, no other process may execute.

Communications table. A table used by the RCS utility COMM to define desired buffer transfers between processor boards.

Compile mode. One of the four RCS operating modes. RCS uses Compile mode when you load blocks of code containing compiling words. See also Locate mode, Run mode, and Show mode.

Configuration flags. Variables used to resolve ambiguities in the inverse transform for a robot.

Control level. A group of functionally bounded modules that handle inputs, decision processing, and outputs for one of the levels generated by the hierarchical task decomposition.

D>M. On the system disk, an area of 12Ø consecutive blocks, used to save a copy of local memory from the currently selected processor board. RCS enables you to save up to five D>Ms for each of seven processor boards. The RCS word [d>m#] D>M, where [d>m#] is a number from 1 through 5, enables you to move the specified D>M region from the disk to the local memory of the current board. The RCS word [d>m#] MEM>DISK moves code from the local memory of the current board to the specified D>M area on the disk. A D>M is also called a disk image, and is similar to an object file in a conventional system.

Directory block. A disk block that contains a description of the contents of other disk blocks. RCS uses a hierarchical directory block structure, in which directory blocks describe the contents of each 1ØØØ-, 1ØØ-, and 1Ø-block section of the disk.

Disk image. See D>M.

Hierarchical task decomposition. An approach to process control that defines tasks in terms of successively more primitive modularized subtasks.

JOINT level. (RSL) The fourth, or lowest, control level in RSL's hierarchical task decomposition. The JOINT level generates specific commands that are transmitted to the robot to control its movement in real time. If the robot has attached sensors, the robot also sends status information back to the JOINT level. See also PATH level, PRIM level, and TASK level.

List owner. A SMACRO construct that enables you to group any type of variable under a specified name (the list owner). See also sequential variable owner and variable owner.

Locate mode. An operational mode that enables you to search for the source block of any SMACRO word. One of the four RCS operating modes. See also Compile mode, Run mode, and Show mode.

Location. (RSL) An RSL data structure that consists of a location name and a specified pose and movetable combination. A location defines a new robot position relative to a reference pose, using the movetable to specify the coordinate transformations required to move to the new position from the reference pose. RSL stores location definitions in a location file in common memory.

Master board. The single-board processor that handles user communication with the system through the user terminal. The RCS editor is on the master board. See also slave board.

Movetable. (RSL) An RSL data structure that consists of a movetable name and up to eight separate rotations or translations that define the coordinate transformations required to move from one robot position to another. RSL stores movetable definitions in a movetable file in common memory.

Object. (RSL) An RSL data structure that defines objects in terms of an object name and a list of grip numbers and associated movetables. The movetables define the orientation of each grip relative to the object frame. RSL stores object definitions in an object file in common memory.

Offset. The value of the FORTH variable OFFSET. The offset is usually a multiple of 1000 associated with the currently selected board. By subtracting the offset from the absolute block number, the system calculates the relative block number. See also absolute block number and relative block number.

Path. (RSL) An algorithm that specifies the sequence of steps required to perform a simple task, such as moving between locations or grasping an object. Paths are stored as linked lists of path-points in a path-point file in common memory. See also path-point.

PATH level. (RSL) The second control level in RSL's hierarchical task decomposition. The PATH level decomposes paths into a sequence of path-points and then into trajectories. See also JOINT level, PRIM level, and TASK level. (RSL)

Path-point. (RSL) An RSL data structure that consists of a path-point command and a pointer to the associated command parameters file. A path-point consists of a single sensor-based motion. A linked list of path-points defines an RSL path. See also path.

Pose. (RSL) An RSL data structure that defines the physical location of a robot. The pose definition consists of a pose name and its associated translation vector and rotation quaternion. RSL stores the pose definition in a pose file in common memory.

PRIM level. (RSL) The third control level in RSL's hierarchical task decomposition. The PRIM level decomposes trajectories into poses along a straight line. See also JOINT level, PATH level, TASK level, and trajectory.

Quaternion. A four-number representation of a rotation by angle about an axis $\hat{n}$. These numbers are expressed as: cos /2, sin /2 $\hat{n}_x$, sin /2 $\hat{n}_y$, sin /2 $\hat{n}_z$.

RCS. See Real-time Control System.

Real-time Control System (RCS). A software system that uses hierarchical task decomposition, multiple processors, and cyclic execution to control real-time applications.

Relative block number. The absolute block number minus the offset; the offset is stored in the FORTH variable OFFSET: See also absolute block number and offset.

Robot Sensor Language (RSL). A real-time control application written by NBS. RSL implements a hierarchical task decomposition that uses four processors to run four hierarchical control levels.

Round-robin. (RSL) An RSL data structure that defines a circular, singly linked list. RSL stores round-robin definitions in a round-robin file in common memory.

Routine. A SMACRO program. All SMACRO programs start with the word routine, which assigns a name to the following SMACRO code, and end with the word end-routine. SMACRO routines can be longer than one block.

RSL. See Robot Sensor Language.

RSL algorithm. (RSL) The RSL code a user writes to accomplish a task. RSL compiles algorithms into a linked-list representation. Then RSL control levels interpret the algorithm, using environmental data to command the robot to execute a task.

RSL data. (RSL) Information about the robot environment, such as the name, size, or location of an object. This information is stored in common memory and is accessible to all control levels. Robot sensors can supply data to RSL, or you can enter data directly from the keyboard. See also RSL algorithm.

Run mode. The normal operating mode for executing the commands that control robotic equipment. Run mode is one of the four RCS operating modes. See also Compile mode, Locate mode, and Show mode.

Sector. (RSL) An element of an array. Sector numbers start at Ø. See also array.

Sequential variable. A SMACRO variable type. The value of the first sequential variable is Ø, and succeeding sequential variables contain successive integers. Sequential variables belong to sequential variable owners.

Sequential variable owner. A SMACRO construct that enables you to group sequential variables as members under a specified name (the owner). Members must be sequential variables. See also list owner and variable owner.

Show mode. An operating mode that enables you to display SMACRO variables and owners. Show mode is one of the four RCS operating modes. See also Compile mode, Locate mode, and Run mode.

Slave board. A single-board processor that implements one or more of the process control levels within the system. Direct user communication with a slave board is possible only with an ABCD switch box. Normally, communications between user and slave board occur through the master board. See also master board.

SMACRO. A set of word extensions to the FORTH programming language and operating system written by NBS for use in the RCS environment. The extensions include structured programming constructs such as case, if-then-else, repeat-until, and while-do statements. The extensions also include words and constructs to facilitate programming real-time control applications, such as state table statements and vector, quaternion, and matrix operator words.

SMACRO file. A linked list of records that resides in common memory. SMACRO files provide the primary method of communication between control levels.

State table. A construct that enables you to describe the possible states of its inputs and outputs. The state table lists the set of relevant input variables, the possible combinations of values the input variables may assume, and the required outputs for each combination of input values. The SMACRO language includes a state table statement to facilitate programming state tables.

System dictionary. A block of common memory that contains records with names and pointers for every variable and procedure in the system.

System vocabulary. A predefined vocabulary that contains system variables. Each processor board has a separate system vocabulary called SDEF. See also user-defined vocabulary and vocabulary.

TASK level. (RSL) The first, or highest, control level in RSL's hierarchical task decomposition. The TASK level decomposes all tasks into a sequence of path types. See also JOINT level, PATH level, and PRIM level.

Trajectory. A goal point and parameters describing the path to the goal point. RSL supports two trajectory types: Cartesian straight-line and joint interpolated.

User-defined vocabulary. A subset of the system dictionary defined by the user. You can add up to five vocabularies per processor board. See also system vocabulary and vocabulary.

User file. See SMACRO file.

Variable owner. A SMACRO construct that enables you to group variables as members under a specified name (the owner). Members can include the following SMACRO variable types: integer, byte, floating point, string, segment, one-dimensional integer array, two-dimensional integer array, one-dimensional floating-point array, and two-dimensional floating-point array. See also list owner and sequential variable owner.

Vocabulary. An independently linked subset of RCS words in the system dictionary. See also system vocabulary and user-defined vocabulary.

Appendix H
BIBLIOGRAPHY

For additional information on robotics, robot interface specifications, the Real-time Control System, FORTH, and specific hardware components, see the documents listed in this bibliography.

ROBOTICS

The following books discuss robotics in general:

Asada, H., and Slotine, J.-J.E. Robot Analysis and Control. New York, NY: Wiley-Interscience, 1986.

Brady, M.; Hollerback, J.M.; Johnson, T.L.; Lozano-Pérez, T.; and Mason, M.T. Robot Motion: Planning and Control. Cambridge, MA: The MIT Press, 1982.

Coiffet, P., and Chirouze, M. An Introduction to Robot Technology. New York, NY: McGraw-Hill Book Company, 1982.

Craig, J.J. Introduction to Robotics--Mechanics and Control. Reading, MA: Addison-Wesley Publishing Company, 1986.

Engelberger, J.F. Robotics in Practice--Management and Applications of Industrial Robots. London, England: Kogan Page Limited, 198Ø.

Kane, T.; Likine, P.; and Levinson, D. Spacecraft Dynamics. New York, NY: McGraw-Hill Book Company, 1983.

Nof, S.Y. (editor) Handbook of Industrial Robotics. New York, NY: John Wiley & Sons, 1985.

Paul, R.P. Robot Manipulators: Mathematics, Programming, and Control. Cambridge, MA: The MIT Press, 1981.

Rehg, J.A. Introduction to Robotics--A Systems Approach. Englewood Cliffs, NJ: Prentice-Hall Inc., 1985.

Snyder, W.E. Industrial Robots: Computer Interfacing and Control. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

ROBOT INTERFACE SPECIFICATIONS

For information on robot interface specifications, see the following publications:

Anon. "SLAVE Interface Specification for External Computer Path Control Using VAL II", Publication Number 397T1; Unimation Incorporated; Danbury, CT: January 1986.

Nashman, M. "A Low Level Robot Interface: The High Speed Host Interface", NBSIR 86-3393; National Bureau of Standards; Gaithersburg, MD: June 1986.

REAL-TIME CONTROL SYSTEM (RCS)

The following documents provide additional information on the NBS Real-time Control System:

Albus, J.S.; Barbera, A.J.; Fitzgerald, M.L.; and Nashman, M. "Sensory Interactive Robots", Presented at the 31st General Assembly of the International Institution for Production Engineering Research; September 1, 1981; Toronto, Canada.

Albus, J.S.; Barbera, A.J.; and Nagel, R.N. "Theory and Practice of Hierarchical Control", Proceedings of the Twenty-Third IEEE Computer Society International Conference; September 1981.

Albus, J.S.; Barbera, A.J.; and Fitzgerald, M.L. "Programming a Hierarchical Robot Control System", Proceedings of the 12th International Symposium on Industrial Robots/6th International Conference on Industrial Robot Technology; June 9-11, 1982; Paris, France.

Albus, J.S.; McLean, C.R.; Barbera, A.J.; and Fitzgerald, M.L. "An Architecture for Real-Time Sensory-Interactive Control of Robots in a Manufacturing Facility", Proceedings of the Fourth IFAC/IFIP Symposium--Information Control Problems in Manufacturing Technology; October 26-28, 1982; Gaithersburg, MD.

Albus, J.S.; McLean, C.R.; Barbera, A.J.; and Fitzgerald, M.L. "Hierarchical Control for Robots in an Automated Factory", Proceedings of the 13th International Symposium on Industrial Robots/Robots 7 Symposium; April 17-21, 1982; Chicago, IL.

Barbera, A.J.; Albus, J.S.; and Fitzgerald, M.L. "Hierarchical Control of Robots Using Microcomputers", Proceedings of the 9th International Symposium on Industrial Robots; March 13-15, 1979; Washington, DC.

Barbera, A.J.; Fitzgerald, M.L.; and Albus, J.S. "Concepts for a Real-Time Sensory-Interactive Control System Architecture", Proceedings of the Fourteenth Southeastern Symposium on System Theory; April 1982; Blacksburg, VA.

Barbera, A.J.; Fitzgerald, M.L.; Albus, J.S.; and Haynes, L.S. "A Language Independent Superstructure for Implementing Real-Time Control Systems", Proceedings of the International Workshop on High-Level Computer Architecture; May 21-25, 1984; Los Angeles, CA.

Barbera, A.J.; Fitzgerald, M.L.; Albus, J.S.; and Haynes, L.S. "RCS: The NBS Real-Time Control System", Proceedings of the Robots 8 Conference and Exposition, Volume 2 - Future Considerations; June 4-7, 1984; Detroit, MI.

Fitzgerald, M.L., and Barbera, A.J. "A Low-Level Control Interface for Robot Manipulators", Proceedings of the Workshop on Robot Standards (Sponsored by the National Bureau of Standards and the Navy Manufacturing Technology Program); June 6-7, 1985; Detroit, MI.

Fitzgerald, M.L.; Barbera, A.J.; and Albus, J.A. "Real-Time Control Systems for Robots", presented at the SPI National Plastics Exposition Conference; June 1985; Chicago, IL.

Haynes, L.S.; Barbera, A.J.; Albus, J.S.; Fitzgerald, M.L.; and McCain, H.G. "An Application Example of the NBS Robot Control System", Robotics & Computer-Integrated Manufacturing, 1984; Washington, DC.

McCain, H.G. "A Hierarchically Controlled, Sensory-Interactive Robot in the Automated Manufacturing Research Facility", Proceedings of the IEEE International Conference on Robotics and Automation; March 25-28, 1985; St. Louis, MO.

FORTH

For more information on FORTH, see the following books that discuss FORTH in general, and the following FORTH reference manuals.

General FORTH Documents

The following books discuss FORTH in general:

Brodie, L. Starting FORTH. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

Brodie, L. Thinking FORTH - A Language and Philosophy for Solving Problems. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Hogan, T. Discover FORTH - Learning and Programming the FORTH Language. Berkeley, CA: Osborne/McGraw-Hill, Inc., 1982.

Katzan, H. Invitation to FORTH. Princeton, NJ: Petrocelli Books, Inc., 1981.

Toppen, D.L. FORTH - An Applications Approach. New York, NY: McGraw-Hill, Inc., 1985.

Winfield, A.F.T. The Complete FORTH. New York, NY: Sigma/Wiley Press, 1983.

FORTH Reference Manuals

The following manuals provide reference information on FORTH. Note that the source of these reference manuals is also listed for your convenience.

Source: FORTH, Inc., 2309 Pacific Coast Highway, Hermosa Beach, CA 90254; (213) 372-8493.

Manuals: *FORTH Reference Manual*, Fourth Edition, April 1979.

Using FORTH, Second, Revised, Edition, March 1980.

Intel 8086 User's Supplement to the PolyFORTH Reference Manual, March 1980.

HARDWARE REFERENCE MANUALS

The following reference manuals discuss processor and I/O boards, memory boards, the disk and tape controller board, the tape drive, the Winchester disk drive, the MULTIBUS chassis, terminals, and printers. The source of each manual or group of manuals is also listed for your convenience.

Processor and I/O Boards

For information on processor and I/O boards, see the following manuals:

Source: Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051; (408) 987-8080. Publications available from Intel Literature Sales, P.O. Box 58130, Santa Clara, CA 95052-8130; (800) 548-4725.

Manuals: *OEM Systems Handbook*, Publication Number 210941-004, 1986.

Microsystem Components Handbook, Volumes I and II, Publication Number 230843-002, 1985.

iSBC 86/14 and iSBC 86/30 Single Board Computer Hardware Reference Manual, Publication Number 144044-002, 1982.

iSBC 337A Multimodule Numeric Data Processor Hardware Reference Manual, Publication Number 142887-001, 1980.

iSBC 519 Programmable I/O Expansion Board Hardware Reference Manual, Publication Number 9800385B, 1979.

iSBC 534 Four Channel Communications Expansion Board Hardware Reference Manual, Publication Number 9800450-02, 1979.

iSBX 311 Analog Input Multimodule Board Hardware Reference Manual, Publication Number 142913-001, 1981.

iSBX 328 Analog Output Multimodule Board Hardware Reference Manual, Publication Number 142914-002, 1982.

iSBX 350 Parallel Multimodule Board Hardware Reference Manual, Publication Number 9803191-02, 1980.

iSBX 351 Serial Multimodule Board Hardware Reference Manual, Publication Number 9803190-03, 1984.

Memory Boards

For information on memory boards, see the following manual:

Source: Plessey Microsystems, P.O. Box 154, 1 Bluehill Plaza,
Pearl River, NY 10965; (914) 735-4661.

Manual: PSM 512A MULTIBUS Error-Correcting Dram--User's Guide, Document Number PMUS/HB/10225, Issue 2, January 1984.

Disk and Tape Controller Board

For information on the disk and tape controller board, see the following document:

Source: Ciprico Inc., 2955 Xenium Lane, Plymouth, MN 55441;
(612) 559-2034.

Document: "Ciprico Product Specification--Rimfire 45A, MULTIBUS Disk Controller," Publication Number 21010084, Revision A, March 25, 1985.

Tape Drive

The following manual provides additional information on the tape drive:

Source: Cipher Data Products, Inc., 10225 Willow Creek Road, San Diego, CA 92131; (619) 578-9100. Publications available from Cipher Data Products, Inc., Technical Publications Department, P.O. Box 85170, San Diego, CA 92138; (800) 982-8808.

Manual: Model F880 Magnetic Tape Transport, Volume 1--Operation Maintenance, Technical Manual Number 799816-003, Revision Q, April 1986.

Winchester Disk Drive

For information on the Winchester disk drive, see the following manual:

Source: Priam Corporation, 20 West Montague Expressway, San Jose, CA 95134; (408) 946-0293.

Manual: Fourteen-Inch Winchester Disc Drives-OEM Manual (Model 3350, Model 6650, and Model 15450), Publication Number 308002, Revision B, January 1984.

MULTIBUS Chassis

For information on the MULTIBUS Chassis, see the following manual:

Source: ETI Micro, 6918 Sierra Court, Dublin, CA 94568; (415) 829-66ØØ.

Manual: 8223 MULTIBUS Chassis User's Manual, Publication Number 8323Ø58223133Ø1, 1983.

Terminals

The following manuals provide information on terminals:

Source: TeleVideo Systems Inc., 117Ø Morse Avenue, Sunnyvale, CA 94Ø86; (4Ø8) 745-776Ø.

Manual: TeleVideo Model 95Ø CRT Terminal Installation and User's Guide, Document Number B3ØØØØ2-ØØ1, Revision B, April 1982.

Source: Digital Equipment Corporation, 129 Parker Street, Maynard, MA Ø1754; (8ØØ) 258-171Ø. Publications available from DEC Accessories and Supplies Group, P.O. Box CS2ØØ8, Nashua, NH Ø3Ø61; (8ØØ) 258-171Ø.

Manual: VT1ØØ User Guide, 3rd Edition, Publication Number EK-VT1ØØ-UG-ØØ3, 1981.

Source: Micro-Term, Inc., 512 Rudder Road, Fenton, MO 63Ø26; (314) 343-6515.

Manuals: Operating Manual--Model ERGO 3Ø1, Publication Number 1-562ØØØØ4-ØD, March 1983. [This terminal is VT1ØØ compatible.]

ERGO 3Ø1FK Addendum--Programmable Function Keys, Publication Number 7985ØØ14-ØA, June 1984.

Source: Datamedia Corporation, 74Ø1 Central Highway, Pennsauken, NY Ø81Ø9; (6Ø9) 665-54ØØ.

Manual: Elite 1521A Video Terminal Operator's Handbook, Publication Number Ø81ØØØ1-ØØØ-B, April 1978.

Printers

For information on printers, see the following manuals:

Source: Apple Computer, 2Ø525 Mariani Avenue, Cupertino, CA 95Ø41; (4Ø8) 996-1Ø1Ø.

Manual: Imagewriter User's Manual, Part I: Reference, Publication Number Ø3Ø-Ø73Ø-A, 1983.

Source: Digital Equipment Corporation, 129 Parker Street, Maynard, MA 01754; (800) 258-1710. Publications available from DEC Accessories and Supplies Group, P.O. Box CS2008, Nashua, NH 03061; (800) 258-1710.

Manual: LA120 DECwriter III User Guide, Publication Number EK-LA120-UG-001, 1978.

Source: Integral Data Systems, Inc., Route 13 South, Milford, NH 03055; (603) 673-9100.

Manual: The Paper Tiger, IDS-460 Impact Printer--Owner's Manual, Publication Number 9000-000-728, Second Edition, April 1981.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NIST/TN-1250	2. Performing Organ. Report No.	3. Publication Date September 1988

4. TITLE AND SUBTITLE

The NBS Real-Time Control System User's Reference Manual.

5. AUTHOR(S)

Stephen A. Leake and Roger D. Kilmer

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
(formerly NATIONAL BUREAU OF STANDARDS)
U.S. DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

SAME AS ITEM #6 ABOVE.

10. SUPPLEMENTARY NOTES

□ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

The NBS-developed Real-Time Control System--RCS-- is a hierarchically-structured controller designed to use sensory feedback for real-time control of automated systems. This manual describes the basic structure of RCS and the programming features available to develop application software. In addition to a detailed description of the structure of RCS, examples illustrating the use of RCS for the control of robotic systems are presented.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

Control architecture; FORTH; hierarchical; RCS; real-time control; robot control; robot programming; SMACRO: sensory feedback.

13. AVAILABILITY

☒ Unlimited

□ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

381

15. Price

☆U.S.GOVERNMENT PRINTING OFFICE:1988-201-597:92573

DEPARTMENT OF COMMERCE • UNITED STATES •

75

METRIC SYSTEM WITH U.S. EQUIVALENTS

METRIC UNIT (I)		U.S. EQUIVALENT
LENGTH		
millimeter	(mm)	.04 inches
centimeter	(cm)	0.39 inches
decimeter	(dm)	3.94 inches
meter	(m)	39.37 inches
decameter	(Dm)	393.7 inches
hectometer	(hm)	109.36 yards
kilometer	(km)	0.62 miles
myriameter		6.2 miles
AREA		
square centimeter	(cm²)	0.155 sq. inches
square meter	(m²)	1.196 sq. yards
are	(a)	119.60 sq. yards
hectare	(ha)	2.47 acres
sq. kilometer	(km²)	0.3861 sq. miles
WEIGHT		
milligram	(mg)	0.015 grains
centigram	(cg)	0.154 grains
decigram	(dg)	1.543 grains
gram	(g)	0.035 ounces
decagram	(Dg)	0.353 ounces
hectogram	(hg)	3.527 ounces
kilogram	(kg)	2.2046 pounds
quintal		220.46 pounds
metric ton		1.1 tons

METRIC UNIT (I)		U.S. EQUIVALENT
VOLUME		
cubic centimeter	(cm³)	0.061 cubic inches
cubic decimeter	(dm³)	61.023 cubic inches
cubic meter	(m³)	[illegible] cubic yards
cubic decameter	(Dm³)	[illegible] cubic yards
CAPACITY, CUBIC		
milliliter	(ml)	.06 cubic inch
centiliter	(cl)	.6 cubic inch
deciliter	(dl)	6.1 cubic inches
liter	(l)	61.02 cubic inches
decaliter	(Dl)	[illegible] cubic foot
hectoliter	(hl)	3.5 cubic feet
kiloliter		[illegible]
CAPACITY, DRY		
deciliter	(dl)	0.18 pint
liter	(l)	0.908 quart
decaliter	(Dl)	1.14 pecks
hectoliter	(hl)	2.84 bushels
CAPACITY, LIQUID		
milliliter		0.27 fluid dram
centiliter		0.338 fluid ounces
deciliter		0.21 pints
liter		1.057 quarts
decaliter		2.64 gallons

OORUM & PEASE
B&P
ELTE
·2532