A MULTITHREADED IMPLEMENTATION OF A ROBOT CONTROL C LIBRARY

BY

PETER J. LEVEN

B.S., Michigan Technological University, 1993
B.S., Michigan Technological University, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

ABSTRACT

Providing a uniform interface between a host computer and a robot controller is an important problem in robotics. RCCL, a Robot Control C Library, is a system that provides such an interface. The implementation of RCCL presents three interesting challenges: how to define the robots and to communicate with them, how to implement robot control, and how to design RCCL so that it can be used on more than one type of computer. This thesis discusses these challenges and how our implementation of RCCL confronts them.

Dedicated to my parents, Peter J. and Marie E. Leven

ACKNOWLEDGMENTS

I would like to thank my parents, to whom this thesis is dedicated, for having the patience to wait for me to finish school.

I would like to thank my advisor, Professor Seth Hutchinson, for providing a rich, creative environment in which to work. His ideas and guidance have been invaluable.

I would like to thank my peers in the Computer Vision and Robotics research group for their many helpful discussions. I am especially grateful to Kevin Nickels whose comments have contributed greatly to this thesis.

I would like to thank the original developers of RCCL for providing a background for this work. I am especially grateful to John Lloyd for pointing out some of the "features" of Solaris that made porting RCCL more interesting.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

LIST OF ABBREVIATIONS

ANSI	American National Standards Institute
BSD	Berkeley Standard Distribution
CCITT	Comité Consultatif International Télégraphique et Téléphonique
CPU	Central Processing Unit
IEC	International Electrotechnical Commision
I/O	Input/Output
ISO	International Standards Organization
ISO/OSI	International Standards Organization/Open Systems Interconnect
POSIX	Portable Operating System Interface
RCCL	Robot Control C Library
SVID	System V Interface Definition
SVR4	System V Release Four UNIX
SYSV	System V UNIX, prior to release four
UI	UNIX International
UNIX	A pun on Multics

CHAPTER 1

INTRODUCTION

Robot programming is a long standing research problem in Robotics. Briefly, robot programming is concerned with the automatic generation of a sequence of control signals that, when given as input to a robot, will cause the robot to successfully execute a specified task. In early robotic systems, this sequence of controls was generated by hand for each task and was "hard-wired" into the robot controller. Later, some form of a teach pendant was used to generate the control sequence. Both of these methods are simple for the manufacturer of a robot to implement in the controller of the robot. However, these methods also require a static environment, in which the location of every object that is used in a task must be known in advance. For tasks for which this requirement holds, the hard-wired or teach pendant programming method is appropriate. However, most environments in which a robot is to be used are not static. These environments contain various forms of variability, from variability in the size and shape of parts involved in a task to variability in the tasks themselves.

To address the issue of variability in robot tasks, many robot programming systems have been developed with the goal of simplifying the process of programming the robot. These systems can be divided into two broad classes: upper level systems and lower level systems. Upper level systems are concerned with the entire task or set of tasks to be accomplished by the robots. These systems generally include a model of the world in which a task is to be executed. This world model contains a description of the robots to be controlled, the objects to be manipulated by the robots, and the obstacles surrounding the robots. This world model may be obtained from some CAD environment, or it may be constructed while the robots interact with their environment. Earlier versions of these systems tended to assume that the world model described precisely the environment of the robot [37]. In newer systems, this world model is assumed to be inexact, and sensor information is used to fine-tune the model. In many systems, the details of robot control and sensor data processing are handled by lower level systems [16], [40]. Some systems seek to provide a natural language interface to specify the task of the robot [17]. Some systems also attempt to learn tasks to be performed by the robots by watching a demonstration of that task [17], [7], [19], [20], [18]. These systems are the focus of much on-going research elsewhere and will not be discussed any further here.

Lower level robot programming systems, on the other hand, are primarily concerned with the low level details of the control of one or more robots. In this case, the world model consists primarily of the robots and the sensors, such as force or torque sensors, that may be attached to the robot. There are a large number of systems that were developed for this purpose [4]. Most of these systems define a special language to control the robot. An example of such a system is TDL [1], which uses a Pascal-like language structure to define the actions of the robot. The limitation to this approach is that it frequently restricts the type of computers that can be used to control the robots. More portable systems are based on widely available languages, such as C [22] or Ada [39]. In this case, the robot control primitives are included in a library of functions, instead of being provided by special language features.

The most useful of the low level systems provide the following features in addition to the portability requirement mentioned above: robot independence, Cartesian programming, and sensor integration. Cartesian programming is useful because it simplifies the specification of the coordinates of objects in the workspace of the robots. Cartesian programming also simplifies the specification of a task in which two or more robots co-operate. Robot independence is important because a task may require the use of more than one type of robot. Sensor integration also is important because of the uncertainty involved in most robotic tasks. For example, the exact position of the objects to be used in an assembly task may not be known in advance.

Several systems exist that achieve, to varying degrees, the above goals for a low level robot programming system. One such system is described in [9]. This system was designed to address some of the limitations of the VAL II control language. The new system is based on Euclid Version 2, a language similar to Pascal but with some extra features for systems programming. The system consists of several modules, each of which performs a set of the functions needed for robot control. For example, there is an initialization module that is used to ensure that the robot to be controlled is properly initialized before the remaining parts of the program are allowed to be executed. Another module provides an interface that replicates many of the language features of VAL II. However, this system seems to be limited to the control of one robot at a time, and is limited to PUMA robots.

A better alternative to the above system is RCCL, a Robot Control C Library [15]. RCCL was created with the above goals in mind. It has the capability to control more than one type of robot and more than one robot at a time. The motions of these robots can be specified in Cartesian coordinates, and sensor information can be incorporated into the motion of the robots.

RCCL has been the subject of much research since its creation, much of which has been to propose various extensions to the system. For example, RCCL has been extended to operate under more than one operating system [25], [24], [33]. RCCL has also been extended to control more than one robot [33]. Some of the other features of RCCL have also been the subject of research, such as trajectory generation [29], [31], [32]. In addition, RCCL has been used by some systems to implement the low level control of robots [40], [13], [12]. RCCL has also been used as the basis for several other systems. These other systems were created to add features that RCCL lacked [14], or designed to satisfy different goals [6], [11].

The implementation of RCCL presents three interesting challenges. The first challenge is to determine the kind of mechanism to be used to control the robot. In this case, there are two separate operations to perform. First, a plan for the motion of the robot must be developed. This plan depends on the objective the robot is to accomplish. For example, the plan for a robot to catch a mouse is different from a plan to assemble a mouse trap. Second, the robot must be commanded to follow this plan. These two steps, for the most part, are not sequential operations, but must occur concurrently. For example, in the case of catching a mouse, the motion plan for the robot must adapt to the motion of the mouse while the robot is in motion.

The second challenge in the implementation of RCCL is in the definition of a robot to be used by RCCL and the means used to communicate with the robot controller. In this case, the definition of a robot refers to the data structures used to describe the robot and the functions used to manipulate those structures. The definition of the robot should be sufficiently flexible to describe the robots that are to be controlled. The means used to communicate with the robot should also be sufficiently flexible to permit the use of different hardware devices to implement the communication.

The third challenge in the implementation of RCCL is that RCCL must run under multiple operating systems. This presents a challenge because the interfaces to the functions that RCCL needs typically are different under different operating systems. This requirement also extends to the programs written to use RCCL. These programs should also be able to be used under different operating systems.

This thesis presents a multithreaded implementation of RCCL. Specifically, the multithreaded approach is a solution to the first challenge proposed above, that is, it provides a mechanism by which RCCL controls robots. In pursuing the multithreaded approach, care had to be taken to ensure that the other two challenges were still met. In this case, the multithreaded approach has an effect on the portability of RCCL and on how RCCL communicates with robots. The methods by which these challenges are met are discussed in this thesis.

The remainder of this thesis is organized as follows. First, Chapter 2 discusses RCCL and the features it provides. Next, Chapter 3 discusses the threaded approach to the control of robots. This is followed by a discussion in Chapter 4 of the communication between RCCL and the robots it controls. Next, Chapter 5 discusses the important differences between the interfaces of different operating systems and how RCCL addresses these differences. Then, Chapter 6 proposes some possible extensions to the multithreaded version of RCCL presented in this thesis. Finally, some conclusions are presented in Chapter 7.

CHAPTER 2

A Robot Control C Library

2.1 Introduction

RCCL is a system for controlling robots using a set of C language functions organized into the RCCL library. RCCL consists of three levels as shown in Figure 2.1. These levels are the planning level, the control level, and the robot level. The top two levels, the planning level and the control level, execute on a host UNIX system, and the bottom level, the robot level, typically executes on hardware dedicated to each robot. Communication between the planning level and the control level is typically in the form of motion requests made by the planning level to the control level and status information about the motion requests sent from the control level to the planning level. Communication between the control level and the robot level is typically in the form of joint motor set points or torques sent from the control level to the robot level and status information such as joint positions and force values sent from the robot level to the control level.

A user program that uses RCCL is composed of two parts: one at the planning level and the other at the control level. The part at the control level incorporates information from the sensors attached to the robot into the trajectories of the robot. The part at the

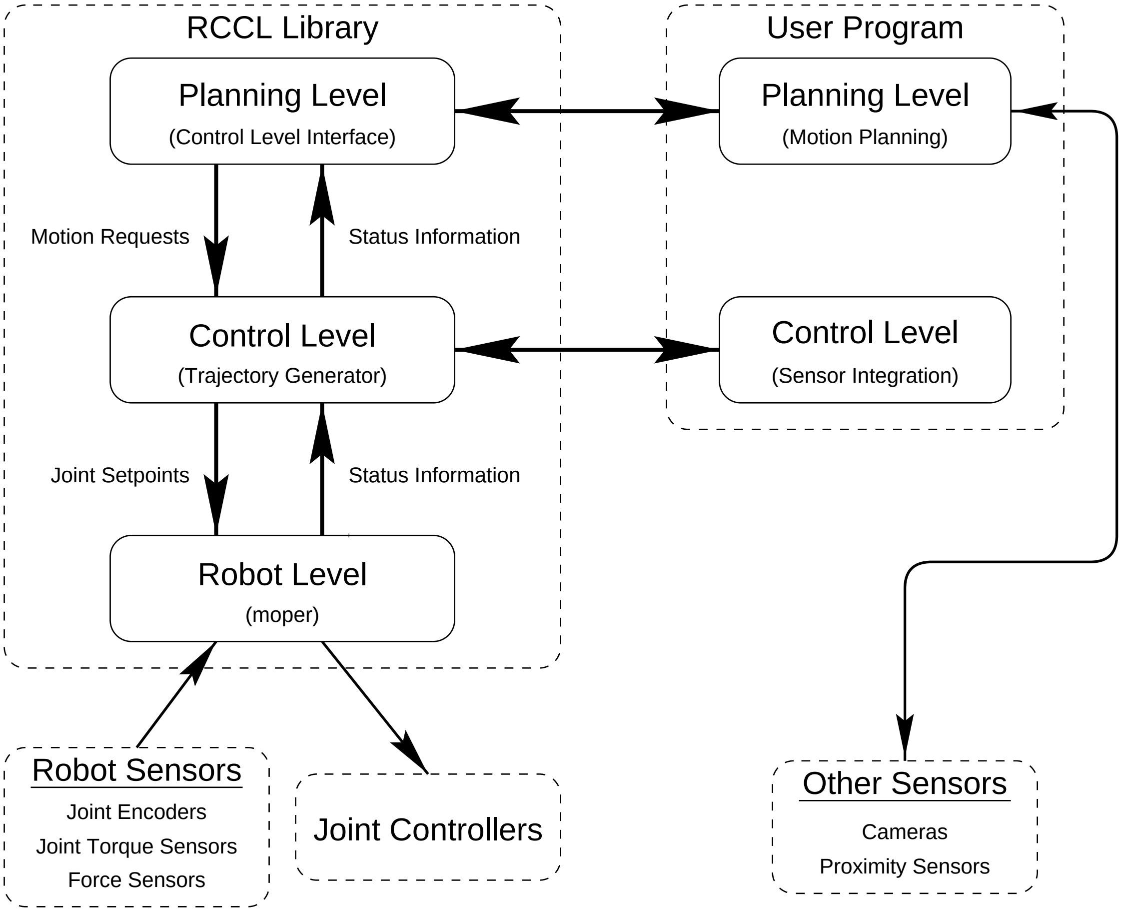

Figure 2.1 An overview of the structure of RCCL.

planning level uses data from cameras and other sensors to generate motion requests for the control level. These two parts interact with each other through the RCCL library.

The robot level of RCCL is composed of the joint controller software coupled with some interface routines to communicate with the rest of RCCL. The joint controller software is typically provided by the robot manufacturer, and controls the motors in each joint to achieve a desired result, such as moving the joint to some position or producing a force or a torque level in the joint. The interface code, called *moper* in RCCL, interfaces RCCL to the joint controllers and to some sensor hardware that may be attached to the robot such as force or torque sensors.

The control level of RCCL consists of the trajectory generator, which accepts motion requests from the planning level and converts them into joint commands to send to the robot. The trajectory generator computes joint commands that drive the robot from its current position to a desired position. Force and torque feedback can also be incorporated into the computation of the joint commands.

The planning level is responsible for determining what motions the robot is to perform. The user interface for the program is also provided at the planning level. The planning level of the RCCL library consists primarily of a set of interface functions to allow the planning level part of the user program to communicate with the control level.

The remainder of this chapter is organized as follows. First, Section 2.2 discusses the trajectory generator, which runs at the control level in RCCL. This is followed by Section 2.3, which discusses tasks in RCCL. Then, Section 2.4 contains a discussion on the representation of robots and the mechanism used to communicate with them. Next, Section 2.5 discusses error handling in RCCL. Last, Section 2.6 finishes the discussion of RCCL by describing some additional features of RCCL. More details about all of these components can be found in [33], [31], [32], [26], [27], [28], and [30].

2.2 The Trajectory Generator

The trajectory generator is responsible for converting robot position requests into joint requests for the robot. The position requests are generated by the planning level and are placed in a motion queue for the control level. Motions are executed by the control level from this queue in a first-in, first-out manner, until the queue is empty. The current implementation of the trajectory generator is capable of controlling robots with up to six joints.

2.2.1 Positions

Robot positions can be specified in two ways in RCCL. The simplest of these is in terms of joint parameters such as angles for revolute joints or positions for prismatic joints. However, this is often not the most natural method for describing an operation to be performed by the robot.

The other method for specifying positions is through position equations. A position equation as used in RCCL is a set of two equal transformations from one coordinate frame to another. The left-hand side of the position equation consists of the set of transforms starting at some fixed coordinate frame, ending at the coordinate frame of the tool held by the robot, and including the transform that describes the position of the end-effector of the robot with respect to the base of the robot. The latter transform is called the *T6* transform in RCCL. The value of the T6 transform is maintained by RCCL, since it changes as a result of the motion of the joints of the robot. The right-hand side of the position equation starts and ends at the same coordinate frames as the left-hand side, but it includes the transform between the tool frame and the target frame. The latter transform is called the *drive* transform in RCCL. The trajectory generator generates a trajectory for the robot to follow that reduces the drive transform to the identity transform.

An example is shown in Figure 2.2. The positions shown in the figure correspond to the following equation:

$$BASE * T6 * TOOL = OBJECT * GRASP * DRIVE. \tag{2.1}$$

The objective of this equation is to move the tool of the robot from its current location to a location in which the object can be grasped. In this case, the coordinate frames are the following: W is the world coordinate frame, B is the base frame of the robot, L is the frame for the last link of the robot, T is the tool frame, G is the grasp frame, and O is the

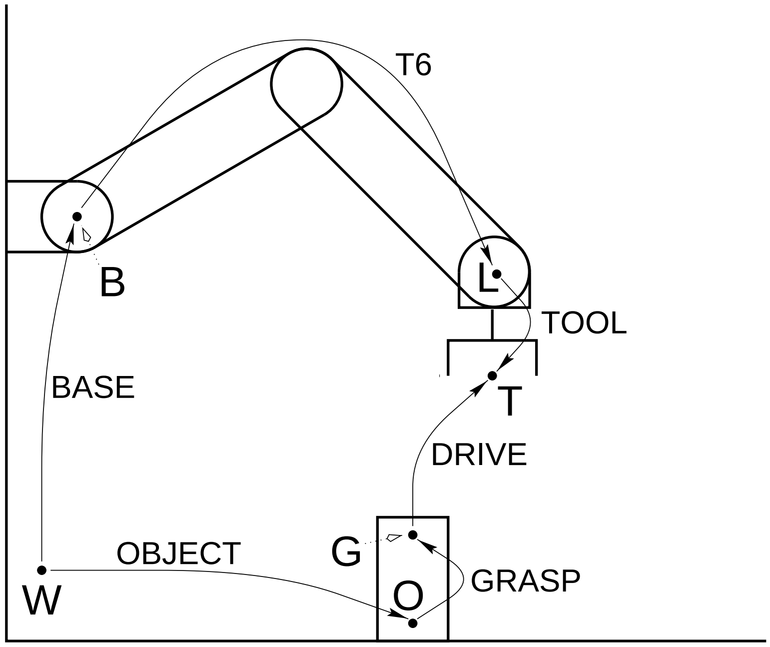

Figure 2.2 A diagram for a position equation for grasping an object.

object frame. The transformations are as follows: BASE expresses the position of frame B with respect to frame W. T6 expresses the position of frame L with respect to frame B. TOOL expresses the position of frame T with respect to frame L. OBJECT expresses the position of frame O with respect to frame W. GRASP expresses the position of frame G with respect to frame O. Last, DRIVE expresses the position of frame T with respect to frame G. As noted above, the function of the trajectory generator is to generate a trajectory for the robot to follow that will reduce the DRIVE transform to the identity transform.

The example above illustrates the case in which the target position is in a fixed location with respect to the world coordinate frame. To allow for moving targets or to incorporate additional information such as force feedback into the trajectory computation, additional transforms can be added. These additional transforms are similar to those used in the above example, except that they are identified as variable to RCCL. Identifying a transform as variable means that the trajectory generator re-evaluates its value at each step and then combines it with the other transforms, instead of combining it with the other transforms at the beginning of the motion and saving the result as a constant.

2.2.2 Parameters for the trajectory generator

The generated trajectories are controlled by several parameters. Some of these parameters are in the form of limits. For example, limits can be set for the Cartesian velocity and acceleration and the joint velocity and acceleration. Some parameters affect the timing of a motion. For example, the transition time between motions and the duration of a motion can be changed. Third, an offset to a position can be specified. This is useful for tasks that need a particular approach direction.

A fourth parameter that affects the trajectories generated for a robot is the configuration used to reach a position. In this case, the configuration of the robot refers to how the trajectory generator deals with the multiple ways a robot can reach a Cartesian position. The multiple ways a robot can reach a particular position are a result of the multiple solutions to the inverse kinematic equations for a given robot. For example, for the PUMA robot, this configuration could be shoulder right-handed or left-handed, elbow up or down, and wrist flipped or not flipped.

The last parameter for controlling the trajectories is the interpolation mode for generating the motion between positions. RCCL has two modes for interpolating between positions: joint mode and Cartesian mode. In joint mode, the interpolation is carried out in joint coordinates, and in Cartesian mode, the interpolation is carried out in Cartesian space. In Cartesian mode, the robot stays in the same configuration throughout a motion. In joint mode, the robot may change configurations. The advantage to joint mode is that a path always exists between two positions, but the path the robot takes may not be easy to describe in Cartesian coordinates. The advantage to the Cartesian mode is that obstacles may more easily be avoided since these are typically specified in terms of Cartesian coordinates rather than joint coordinates. The disadvantage to Cartesian interpolation is that the robot may not be able to perform some of the motions because some of the positions may be unreachable in the current configuration of the robot. Joint mode is the interpolation mode used when a motion request is in the form of a set of joint values and is the default mode for motions specified by position equations.

2.3 Tasks

A task is a set of functions scheduled by RCCL at the control level. These tasks are similar to the threads that will be defined in Section 3.1.1 in that each task shares the same address space, each task has a set of synchronization primitives for synchronizing between tasks, and each task executes independently from each other task. However, tasks are not threads, though each task may be executed as a separate thread. The means by which tasks are defined are also different from those for threads.

Each task is set up by the planning level to perform some operation, such as the generation of trajectories to satisfy the motion requests for a robot. Each task can generate the trajectories for one or more robots, or it can perform some other operation such as the acquisition of sensor data. Each robot may be associated with only one task at a time. Multiple tasks can be in operation simultaneously within an RCCL program. Once a task has been created, it can be started, suspended and resumed, and stopped. Each task has an associated scheduling mode, described in Section 2.3.1, and can communicate with other tasks as described in Section 2.3.2.

Associated with each task are three functions: startup, control, and release. The startup function is responsible for setting up the task and any devices with which the task communicates. Depending on the scheduling mode, the task may also set up some extra functions, as described in Section 2.3.1. The control function is the primary task function. It is this function that is called when the task executes. The release function is responsible for shutting down the task, which primarily involves shutting down external devices. All of these functions are set up automatically when a trajectory generator task is created. For other tasks, these functions must be set up by the programmer.

2.3.1 Task scheduling

There are four scheduling modes for tasks: *on clock*, *on function*, *on trigger*, and *on signal*. These scheduling modes are used to determine when the control function for a task executes. The mode chosen for a particular task depends on the robots that it is to control and other operations that it may perform.

With the *on signal* scheduling mode, a UNIX signal is used to trigger the execution of a task. This mode is primarily for use with the robot simulator. With the *on clock* mode, a task is executed at a specific rate that can be set by the program. The highest rate at which a task can be executed is limited by the rate at which the UNIX system keeps track of time or by the rate at which some special hardware can be used to signal timeouts. For modern UNIX systems, the system clock rate is typically 100 Hz. An offset with respect to the rate can also be specified. This is useful for tasks that execute at the same rate but have to be scheduled at different times with respect to one another.

With the *on function* scheduling mode, the task is executed whenever a polling function indicates that an event has occurred. This polling function is set up by the start-up function for the task. A typical event for which the polling function can poll is an indication from a robot associated with the task that data are available.

The *on trigger* scheduling mode is a combination of the *on clock* and *on function* modes. In this case, the task sets up a trigger function that activates some external device. The polling function then waits for that device to be ready, at which point the task is executed. The rate and offset defined for the *on clock* mode are also used to schedule the trigger function. Since the trigger function must be reinstalled each time the task is executed, the rate and offset are useful to prevent timing drifts.

For the *on clock* and *on trigger* scheduling modes, the control level of RCCL provides the timing for the execution of robot tasks. For the *on function* mode, the robot provides the timing. For the *on signal* mode, an external UNIX process, typically the robot simulator program, provides the timing by sending the appropriate signal at the appropriate time to the RCCL program.

In addition to the above scheduling modes, each task can also set up a function as a minitask to be executed at some rate relative to the clock. This function can be used to perform time-critical operations such as polling a sensor at a rate different from, and possibly faster than, the main task.

2.3.2 Intertask communication

Frequently, tasks must communicate with each other to perform some operations. In some cases, this involves passing data back and forth, and in other cases, it only involves signaling that some event has occurred. RCCL provides several methods to facilitate this communication. These methods include memory segments and memory objects shared among tasks, message passing, signals, and indicator functions.

The simplest of the communication methods is through the use of shared memory segments. There are three types of shared memory segments, two of which are shared among tasks. One of these latter two types is shared between the planning task and one control task, and the other type is shared among all tasks. The advantage of shared segments is that they are simple to set up and to use. The disadvantage is that their use depends on some sort of external synchronization method such as sequential task execution to prevent access conflicts.

The second method for communication among tasks is through shared memory objects. These objects are similar to shared memory segments that are shared among all tasks, except that a set of atomic access functions is defined for these objects. The atomic access operations remove the dependence on external synchronization to prevent access conflicts.

The third method for communication among tasks is through message passing. Messages can be sent from any task and the user program at the planning level, to any other single task or to the user program at the planning level. A task can receive a message sent from any other task or it can limit the set of tasks from which it will receive messages. A task can also flush all messages that are currently waiting in its message queue. Access conflicts between the receipt of messages and the transmission of messages are prevented by the interface functions that send and receive the messages.

The above methods are useful for both sending data between tasks and for indicating the occurrence of events to other tasks. There are two other methods that are used primarily for indicating the occurrence of an event from the control level to the planning level: signals and wait functions. Most events are related to the execution of motion requests. In the signal method for indicating events, the control task sends a UNIX signal to the planning level in response to some event. Problems with using UNIX signals are discussed in Chapter 3.

The second method for indicating the occurrence of events to the planning level is through the use of wait functions. In this case, the planning level calls a wait function to wait for the occurrence of the event. There are two types of wait functions: those that control the number of pending motion requests and those that wait for a particular motion. Functions that wait for a particular motion pause the execution of the planning level until a specified percentage of the motion has completed. At the percentage of zero, the planning level pauses until the motion starts, and at 100 percent the planning level pauses until the motion completes. Changing the size of the motion request queue changes the number of motion requests that may be submitted before the planning level has to wait for a motion to complete.

2.4 Robots

The purpose of RCCL is to control robots. Each robot can have as many as 12 joints, though the trajectory generator in the current implementation is limited to robots with six or fewer joints. For each robot, RCCL uses a set of data structures and functions to describe the robot and to communicate with the robot.

2.4.1 Robot data structures and functions

Associated with each robot is a set of data structures and a set of functions that operate on those structures. Some of the data structures are common to all robots, and contain information such as joint angles or positions, joint limits, encoder value to angle conversion factors, encoder offsets, and motor torque to joint torque conversion factors. Some data structures are specific to each robot and contain information such as gravity loadings and a scratch pad area for storing derived state information for computational efficiency. The functions associated with each robot perform operations such as conversion from joint angle or joint positions to encoder values, forward and inverse kinematic computation, and Jacobian computations.

2.4.2 Robot communication

In RCCL, there are three layers of functions for communicating with robots. The top two layers are part of the RCCL library, and the third layer is integrated into the host operating system. These layers correspond to different levels of abstraction for communicating with a robot.

At the top layer are the communication functions defined for each robot. The functions in this layer transform the representation that RCCL has of a robot into a packet of data that can be transmitted to the robot, and vice versa. These routines then call routines in the lower layers to perform the communication with the robot. This allows the functions that are specific to a particular robot to be independent of the type of communication device that attaches the robot controller to the computer.

The middle layer performs most of the actual communication with the robot. This isolates the robot interface code from the code needed to drive the hardware that forms the communication link with the robot. With this layer, a communication board may be used with more than one type of robot.

The bottom layer of robot communication functions is integrated into the operating system. Its primary purpose is to accept interrupt signals from the robot and to convert them into data requests for the control level. The lower two layers of these communication functions will be discussed in greater detail in Chapter 4.

2.5 Error Handling

There are many types of errors that can occur in an RCCL program. The simplest type of error is an invalid parameter passed to a function. This typically occurs due to programming errors. Another type of error is the occurrence of an out-of-resource condition. A typical resource that may be exhausted is computer memory. A third type of error is a resource conflict, which will occur when two programs attempt to control the same robot at the same time. A fourth type of error is the absence of a robot to which a program may be attempting to connect. This can occur when a robot is not turned on, for example. A fifth type of error is an abort condition indicated by the robot. This will occur when the emergency stop button is pressed on the robot. These errors vary in severity, and the respective responses from RCCL also vary. There are three methods used by RCCL to handle errors: special return values, messages on an error stack, and the abort function.

The first error handling method is to return a value from a function that indicates that an error has occurred. This method is primarily used for RCCL functions that are called directly from user code at the planning level and perform relatively simple processing on input data passed as parameters. In this case, the errors are typically related to invalid operands.

The second method for handling errors is to push an error message and a corresponding error code onto an error stack. This method is used in RCCL for functions that are called at many nested levels. Using this method, a message for an error that occurs at a low level will make it back to the top, where the error can be handled. Each of the levels in between can add a message to the stack, providing a more detailed description of the error. This method for reporting errors is also used to report error conditions from the control level to the planning level. This method and the first method are used in cases in which some sort of option is available for recovering from the error.

The third method for error handling combines the error stack with a call to the RCCL abort function. This is used for errors for which the only recovery option is to shut down all tasks and report the problem to the planning level. There are two options for what action the planning level takes as the result of an abort. The first option is for the current contents of the error stack to be printed and for the program to exit. This is the default option. The other option is for the program to set up a handler function to catch the aborts, and for this handler to execute some recovery operation.

RCCL also has a special mode for handling errors that is primarily useful for program development, in which the errors provide an aid to debugging a program. This mode causes an RCCL program to exit when certain RCCL functions detect error conditions. The RCCL functions that use this mode are primarily those that also report error conditions using the error stack. The messages on the error stack are printed before the program exits.

2.6 Other Components of RCCL

This section discusses some additional parts of RCCL. These parts include the teach mode of RCCL, the robot simulator, the C-Tree input parser, the demonstration programs, and the utility programs.

2.6.1 Teach mode

RCCL contains a teach mode, in which the user can move the robot and record its positions. While in the teach mode, the robot can be moved either by using the teach pendant connected to the robot or by commands typed in from the keyboard. Each time a "record" command is entered, the teach mode exits and returns the last position to the program that called it. This can be used in a loop to learn a sequence of motions for tasks that do not need sensor feedback.

2.6.2 Simulator

RCCL also contains a robot simulator, which is a separate program that an RCCL program can connect to in the place of a real robot. The simulator provides a graphical view of the world of the robot and shows the motions that the robot would take. In the interactive mode of the simulator, the user can input sensor values which are then sent to the RCCL program controlling the robot. The user can also single-step through control cycles, which is useful for debugging RCCL programs.

2.6.3 C-Tree input parser

The C-Tree input parser is a simple command completion interface used by RCCL demonstration programs, utility programs, and the teach mode. As an initialization step, it accepts a command template, which consists of a null-terminated list of strings alternating with pointers to variables. This template is stored as a parse tree. Each time a program waits for another command from the user, this tree is used to guide the user to select a command from the set of available commands.

For example, consider the demonstration program **jmove**, whose parse tree is shown in Figure 2.3. At the top level, **jmove** has the following four possible commands: *quit*, to leave the program; *setconf*, to set a configuration for the robot; *move*, to move one or all of the joints to some location; and *show*, to display the current position and configuration of the robot. Two of these commands, *show* and *quit*, are single word commands that require no other modifiers or input values. The other two commands require more input from the user. The command *setconf* requires the input of a string value that indicates a new configuration into which the robot should be placed. For this program, the possible configurations are right or left shoulder, elbow up or down, and wrist flip or no-flip. The command *move* has more possibilities. The modifier *all* can be selected, or an integer

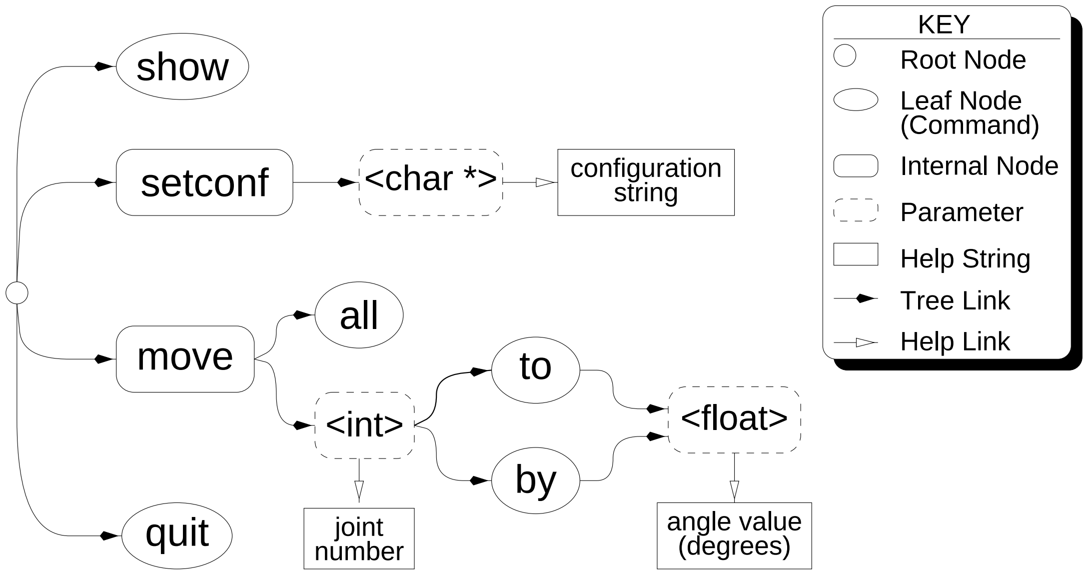

Figure 2.3 The parse tree for the program **jmove**.

can be entered to indicate the number of the joint to be moved. If a joint number was selected, then either *to* or *by* can be selected, and a floating-point value is entered to specify the angle *to* which the joint should be moved or the amount *by* which the joint should be moved.

A user of this program will see a prompt message when the program is waiting for a command. The user can type one of the commands or one of several special characters. The special characters are the following: the "?" character prints a list of all commands that can be completed from characters already entered, or a list of all commands if no characters have yet been entered; the "*" character prints a list of the tree; the "!" character allows a shell command to be executed; and the "@" character tells the matcher to read commands from a file. A space character or an end-of-line character completes the current word if a unique match is available and either returns the command to the program if the match is at a leaf node in the tree or waits for the next part of the command.

2.6.4 Demonstration programs

RCCL includes a number of demonstration programs to illustrate the use of some of its capabilities. A list of these programs is shown in Table 2.1. These programs demonstrate various capabilities of RCCL, such as different interpolation modes, the effect of velocity and acceleration limits, and the integration of external sensor information such as that from force or vision sensors into the trajectory of the robot.

2.6.5 Utility programs

RCCL provides two types of utility programs. The first is used for setting up the robot so that it can be controlled by RCCL, or for moving the joints of the robot. The programs used for setting up the robots are **down**, **free**, **makejls**, **potcal**, **pumacal**, **rciProbe**, and **termlink**. The program **down** downloads moper to the robot controller and **termlink** can be used to monitor moper once it is downloaded. The program **free** allows one to move the joints of the robot by hand to a position at which the robot can be calibrated by the **pumacal**, **potcal**, or **primecal**. For these programs, calibration means verifying that RCCL's view of the robot corresponds to its actual configuration. The **rciProbe** program is used to check the existence of the communication boards used by RCCL to control the robot.

The second type of utility program is for moving the joints of the robot. These programs are **move**, **power**, **teachdemo**, and **zerograv**. The program **move** moves a joint to a new position. The program **power** turns on the arm power of the robot. The program **teachdemo** allows use of the teach pendant to move the robot and the program **zerograv** puts the robot in a gravity-compensated mode in which the torque in each motor is just enough to balance the weight of the robot.

Table 2.1 Demonstration programs available in RCCL.

Program	Description
box	Traces the outline of a square. Demonstrates the effect of changing the speed and acceleration settings.
comply	Demonstrates the integration of a force sensor attached to the wrist of the robot to allow the robot to be "pushed" in any translational direction.
cylin	Demonstrates the combination of guarded motions and force sensing to trace the outside of a cylinder.
gravityDemo	Demonstrates friction and gravity compensation.
guarded	Demonstrates guarded motions.
hex	Traces the outline of a hexagonal grid. Demonstrates signaling the planning task on completion of a motion.
jmove	Demonstrates the use of the C-Tree parser to allow interactive motion of the joints of the robot. Assumes the robot has no prismatic joints.
logger	Demonstrates a double-buffered method for logging data from the control level.
memDemo	Demonstrates the memory allocation mechanism.
msgDemo	Demonstrates the message passing mechanism.
onSigDemo	Demonstrates the *on signal* scheduling mode using timers.
pickAndDrop	Simulates the use of a camera to determine the position and orientation of an object that is to be picked up by the robot.
pivot	Demonstrates the use of a functional transform to achieve the effect of rotating a lever 60°.
playback	Demonstrates the use of the teach mode to record and play back positions.
rotate	Demonstrates the tracking of a wheel rotating on a second rotating wheel using variable transforms and the table-lookup routines for sine and cosine.
shmemDemo	Demonstrates the shared memory mechanism.
simple	Performs a simple movement. Demonstrates joint interpolation versus Cartesian interpolation
track	Demonstrates position tracking of a sensor value in which the sensor value indicates a position on the X-Y plane of the robot's end-effector.
trackII	Demonstrates two robots tracking the same path.
zerogDemo	Demonstrates gravity compensation.
zigzag	Demonstrates the use of variable transforms in a position equation to move between two points on a conveyer belt.

CHAPTER 3

MULTITHREADING RCCL

3.1 Introduction

As described in Chapter 2, RCCL is composed of three levels. The lowest level is the robot level, and it typically operates on the hardware provided with the robot. This level will not be discussed here. The other two levels, the control and planning levels, execute on a host computer which has a UNIX variant for its operating system.

The control level is concerned with the control of one or more robots. At this level, the control of a robot refers to satisfying motion requests generated by the planning level. This motion request may involve moving the joints of the robot to reach a position, to follow a path, to achieve some level of torque or force response at the end-effector, or to perform some combination of these three. The algorithms used to achieve this typically must execute at a specific rate or respond within a bounded period of time to an external stimulus. An external stimulus may be a robot indicating that it has data to process, such as for the *on function* scheduling mode described in Section 2.3.1.

The planning level is concerned with the generation of motion requests for the control level to execute. These motion requests are generated to control a robot or set of robots to achieve some goal, such as the welding of a joint between metal parts, catching a mouse, or painting a wall. This level will typically process a large amount of information, such as that received by a camera or set of cameras, to determine what these motion requests should be.

3.1.1 Definitions

To facilitate a discussion on the implementation of the control and planning levels of RCCL, some definitions are needed. These definitions are similar to those found in references on operating systems [38], real-time operating systems [8], and programming with threads [23]. A *program* is a set of instructions for a computer that, when executed, will perform some desired operation. A *process* is a program being executed by a computer. The *address space* of a process is the set of memory locations to which the process can refer. A *real-time process* is one whose correctness of operation depends on the timing of its operation, in addition to the results it produces. A *hard real-time process* is one for which late results indicate a failure in the program. The control level of RCCL is an example of a hard real-time process in which a late computation could, for example, result in a collision between a robot and its workspace. A *soft real-time process* is one for which late results are still useable but are not as useful as on-time results. The planning level of RCCL is an example of a soft real-time process, since a late motion request usually means that the robot stops between motions, perhaps resulting in a non-smooth path. A process is *preempted* when the operating system stops the process to run some other process before the first process has completed its computation. From the point of view of the operating system, a process has completed its computation if it either terminates or stops to wait for some event.

A *signal* is a notification of an event to a process. There are two types of events that generate signals that can be received by a process: *synchronous* and *asynchronous*.

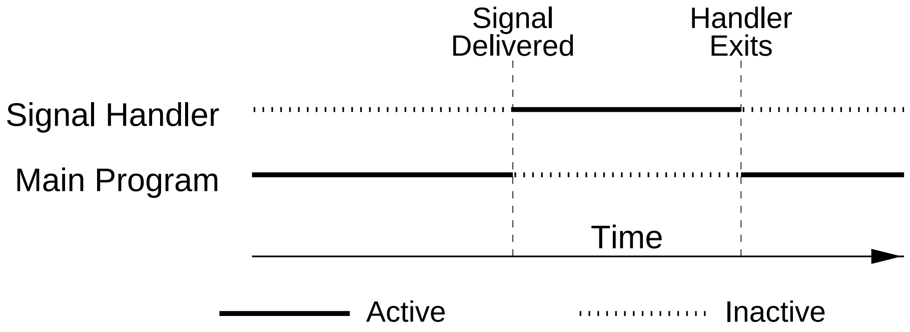

Figure 3.1 Signal delivery in a process.

A synchronous event is one caused by the execution of a program, such as a divide by zero or a reference to an invalid memory location. Most of these events occur due to programming errors, although they can be used in other cases as well. For example, RCCL uses the signal that indicates a reference to an invalid memory location to test for the presence of the hardware used to communicate with the robot. An asynchronous event is one that is caused by something external to the process, such as an interrupt or alarm.

A signal is defined as *delivered* when an appropriate action has been performed for the process that is the target of the signal. This action can be defined by the process, or it can be defined by the operating system. When the action for a signal is executed, the process is interrupted. This execution is asynchronous with respect to the process because the process can be interrupted at any time to handle the signal. An example of a process receiving a signal is shown in Figure 3.1. The *disposition* of a signal refers to the action that will be performed when a signal is delivered to a process. A *signal mask* is a bit vector in which each bit controls whether the process will receive a signal associated with that bit. If a bit is *set* in the signal mask, then the signal is said to be *blocked* from delivery to the process.

A *thread* is an independent sequence of execution in a process. A process can be single-threaded, in which case there is only one instruction sequence, or it can be multithreaded,

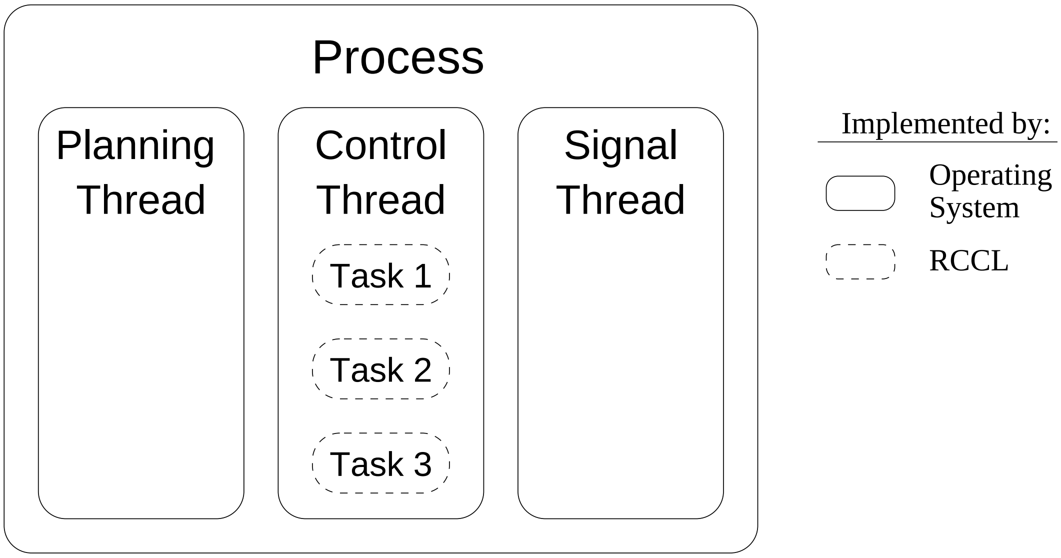

Figure 3.2 The relationship among processes, threads, and tasks.

in which case there is more than one instruction sequence. Each thread in a process shares most of the resources used by the process, such as the address space and all open files. Each thread has its own execution stack, signal mask, and scheduling parameters. Thread scheduling can be handled by the operating system, by library functions, or by some combination of the two.

Figure 3.2 illustrates the relationships among processes, threads, and tasks for a typical RCCL program. The process is the whole program as it executes on a computer. Inside the process, there are three threads. Each of these threads will be described in more detail in Section 3.2. Inside the control thread, there are three tasks. Processes and threads are created, maintained, and scheduled by an operating system and its associated libraries. Tasks are created, maintained, and scheduled by RCCL.

Threads communicate with each other using a variety of mechanisms. One mechanism is shared global variables. If each thread operates on separate data from the other threads, then no synchronization is needed since the operation of each thread does not conflict with the others. If some or all threads need to access the same variable, some

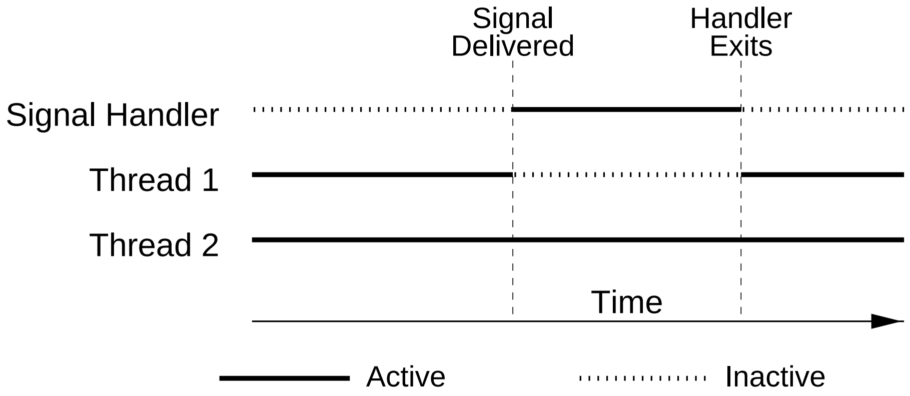

Figure 3.3 Signal delivery in a multithreaded process.

kind of synchronization is required to prevent an access conflict between threads. To synchronize access to these variables, threads can use mutual exclusion locks, in which only one thread can access a variable at a time, or readers and writers locks, in which one writer or multiple readers can access a shared variable at the same time. Mutual exclusion locks are also known as *mutexes*.

A second mechanism for communication among threads is through the use of condition variables. These are used in situations in which a thread needs to wait for another thread to complete some activity. Condition variables are used in conjunction with mutexes and another variable. The mutex is used to ensure serial access among threads to the condition variable, and the extra variable is used to indicate what event occurred. A thread indicating the occurrence of an event can then use the condition variable to signal one or all threads that are waiting for the event.

A third mechanism for communication is through the use of signals. A thread can send a signal to any other thread. A signal delivered to a multithreaded process interrupts only one thread in the process. Shown in Figure 3.3 is an example in which a program with two threads receives a signal. In this example, only Thread 1 is interrupted by the signal.

A function is defined as *asynchronous safe* if it can be called by a signal handler and by the main program simultaneously. A function is defined as *thread safe* if it can be called by many threads simultaneously, and execute correctly in each case. A function is defined as *reentrant* when it can be entered multiple times by a single thread or by multiple threads. This is similar to a thread safe function except that multiple threads can execute a reentrant function without conflict as long as the data passed to the reentrant function are unique in each thread. A recursive function, such as a function to search a binary tree, is an example of a reentrant function. If each thread maintains its own tree, the tree search function can be called by all threads simultaneously without conflict.

3.1.2 Implementation of RCCL

For a typical operating system, RCCL has two conflicting goals. On the one hand, the control level of RCCL requires execution at a fixed rate, a bounded response time, and the ability to execute without preemption until it has completed its computation. On the other hand, the planning level requires the ability to perform a large amount of computation and to possibly communicate with other programs that may also perform a large amount of computation. In a traditional operating system, the scheduling of processes is based on a priority system in which processes that have higher priorities are executed before those with lower priorities. These priorities are calculated dynamically to balance access to the processor between programs that perform a large amount of computation with little input or output and programs that perform little computation but spend most of their time waiting for some input or output. This approach satisfies the computational requirements for the planning level, but typically does not satisfy the fixed execution rate or the bounded response time required by the control level.

Real-time operating systems, on the other hand, do provide the facilities required by RCCL for both the planning and control levels. In this case, the control level can be guaranteed to have some bound on its response time, to be executed at a fixed rate, and to execute until completion without preemption. This is typically achieved by giving the control level a scheduling priority that is higher than that of other processes executing on the system. The planning level is operated as a lower priority process to avoid interfering with the control task.

In previous implementations of RCCL, the planning and control levels were implemented using one of two methods. The first of these methods is used on operating systems that do not have sufficient intrinsic support for real-time programs. In this case, the control level is integrated into the operating system along with some additional functionality to make the operating system more supportive of RCCL. The planning level executes as a normal process and uses system calls to interface to the control level in the operating system. The two types of systems in this category are MicroVAX running a variant of BSD UNIX and SUN SPARCstation running SunOS 4.1. The second method for implementing RCCL applies to operating systems that include sufficient support for real-time programs. In this case, the planning and control levels operate as two or more independent special processes that share a common address space. The two levels communicate with each other through their shared address space. The two types of systems in this category are Silicon Graphics computers running IRIX 4.0 and computers running the VxWorks operating system.

Section 3.2 describes our threaded implementation of the planning and control levels of RCCL. The threaded version is a variant of the special process method for implementing RCCL. Threads were chosen rather than the special process method because the interface to threads is more portable between operating systems and the interface to threads provides a larger set of capabilities.

3.2 Threads in RCCL

The implementation of RCCL described here is composed of three threads: control, signal, and planning. Each of these will be described below.

3.2.1 Control thread

The control thread, which implements the control level of RCCL, is implemented as an event loop in which the thread waits for three types of events. The first type of event consists of requests from the planning level to perform some operation. Some of these operations are provided to support some older architectures on which RCCL is implemented. On these older architectures, certain types of operations could only be performed at the control level. Other operations are used to provide bookkeeping functions for the control level. The attachment of robot handler functions to robot interfaces is one example. Another example is the registration of functions to be executed at intervals specified by the planning level.

The second type of event for which the control thread waits consists of robot interface events. A robot interface event typically occurs when the robot level, executing on some robot, is ready for a new set of commands, or is ready to report the occurrence of some event to the control level. A handler function is then called to process the event from the robot. This handler is typically different for each robot connected to RCCL. These handlers are used to implement the *on function* scheduling discipline and the polling part of the *on trigger* scheduling discipline, but other operations can also be defined.

The third type of event for which the control level waits consists of timing events. There are two types of timing events: alarm and periodic. Alarm events are one-time events that are set up to execute a function at some time in the future. The trigger function for the *on trigger* scheduling mode is set up this way. Periodic events are those that occur at some fixed rate. These are used to implement the *on clock* scheduling discipline.

The timing events and the robot events are the reasons that RCCL requires a real-time operating system and the control thread must be executed as a real-time thread. For these two types of events, the control level must respond quickly, on the order of one millisecond, to keep the robot moving on a smooth trajectory. The clock events also require a high degree of determinism to ensure a constant rate. This is necessary for most control algorithms for robots, especially the trajectory generator used by RCCL. These algorithms must also be able to complete their computation during each control cycle. These requirements are satisfied by creating the control thread as a special real-time thread.

3.2.2 Signal thread

In a multithreaded program, the problem of signal handling is worse than for a single threaded program. In a single threaded program, only one sequence of instructions is interrupted to process a signal. In this case, it is clear that either the main part of the program or one of its signal handlers is being executed at any given time during the execution of the program. This imposes some restrictions on the type of operations that can be performed by the signal handler. For example, the process may be interrupted while it is in the middle of updating a global variable. The signal handler cannot then use this variable, since its state is incorrect. This induces a restriction on the functions that can be called by the signal handler. The functions that can be called by a signal handler are those functions that do not use global variables.

In a multithreaded process, on the other hand, many more problems can arise. The first of these is that only one thread is interrupted to handle the signal. The other threads continue executing concurrently with the signal handler. This further restricts the types of accesses to global variables that the signal handler can make. Another problem is that mechanisms for synchronizing with other threads, such as mutexes, cannot be used by a signal handler because their use can cause a deadlock to occur. For example, a deadlock occurs when a signal handler tries to lock a mutex held by the thread that was interrupted to run the handler. In this case, the thread cannot release the lock because the signal handler has not exited, and the signal handler will not exit until after it has acquired the lock.

A third problem is that the signal handlers are shared among all of the threads in the process. For many types of signals, this is not a problem, since the response to the signal is independent of which thread handles the signals. For some signals, however, the response to the signal depends on which thread caused the signal and that response may be valid in only that thread. A related problem is that, for most signals, it is not possible to determine in advance which thread will handle the signal. The signals that do not suffer from this problem are those that are caused by the execution of a particular thread, such as a signal resulting from a divide by zero operation.

Fortunately, a new paradigm for handling signals has been developed for threaded programs. In this paradigm, one or more of the threads is set up to wait for the arrival of one or more signals, and all other threads block these signals from delivery. In this case, the signals are handled synchronously with respect to a thread, which permits a handler to use any function that other threads are allowed to use, including the synchronization primitives available to the other threads.

This is the purpose of the signal thread in RCCL. It is used to wait for the arrival of all signals, and all signals are blocked in the other threads. When a signal arrives, the signal thread responds in one of two ways. In one case, RCCL has an operation defined for that particular signal, and this operation is performed on receipt of the signal. In the other case, RCCL does not have an operation defined for the signal. In this case, the process-wide signal handler is invoked by unblocking the signal in the signal thread and resending the signal. This allows the process-wide signal handler to be executed synchronously by the signal thread.

The signal thread operates as part of the planning level of RCCL. This is done for three reasons. First, RCCL provides a means for the control level to send a signal to the planning level, typically as the result of the completion of some motion. Having the signal thread as part of the planning level preserves this semantic. The functions that send a signal to the planning level are then converted into functions that send the signal to the signal thread. The second reason for implementing the signal thread in the planning level is that signals are not used for the time-critical operations that control the robots. Signals are not used for time-critical operations because signal delivery is usually too slow or unreliable on UNIX operating systems to be used for robot control. The third reason for a signal thread at the planning level is that signals are used to indicate abort conditions in the control level. These abort conditions then need to be reported to the planning level so that some appropriate action can be performed.

3.2.3 Planning thread

The planning thread implements the planning level of RCCL with the exception of signal handling, which is performed by the signal thread as described above. As described earlier, the planning level is responsible for generating motion requests and for providing the user interface for the program.

CHAPTER 4

DEVICE DRIVERS FOR RCCL

4.1 Introduction

In order to control a robot, RCCL needs some means by which to communicate with the robot. This communication is implemented using some type of hardware I/O device, such as the Xycom digital I/O board, attached to the computer on which RCCL runs. In addition to this I/O hardware, RCCL also requires hardware to provide the timing used at the control level. In a UNIX operating system, access to these devices from a user program is performed using special files in the file system of the computer. An access to one of these files is handled by a device driver installed in the operating system.

RCCL uses two layers of device drivers. One of these layers is added to the operating system and includes the drivers for the devices described above. These device drivers are responsible for verifying the existence of the hardware devices and for catching hardware interrupts generated by the devices. The clock device for RCCL is also implemented at this level, either using some special clock hardware, or using a timeout function provided by the operating system to perform the timing using the UNIX system clock. The other layer of device drivers used by RCCL is integrated into the RCCL library. This layer performs the communication between the robot and the control level of RCCL.

The remainder of this chapter discusses these two layers of drivers in RCCL. The lower level drivers are discussed in Section 4.2, and the upper level drivers are discussed in Section 4.3.

4.2 Drivers in the Operating System

A computer consists of many different hardware components. The simplest components for a program to access are the processor and its associated memory. These two components require no special software to access them. However, consider the addition of a monitor, a keyboard, a mouse, a disk drive, and a network connection. These additional components can vary widely from computer to computer and, therefore, require some special software to isolate the differences in the hardware components from the programs that use them. In the language of operating systems, these components are called *devices*. In modern UNIX systems, there are three major classes of devices [36], [10].

The first device class is the *character* device. There are two types of character devices. One of these types controls a set of input and output lines, such as a serial port or a parallel port. These devices are used to communicate between the host computer and some external hardware such as a printer or a robot controller. The other type consists of an array of memory locations, such as a frame buffer. For these devices, the writing of a value into one of these locations may cause some operation to be performed. For example, writing to a location in a frame buffer may cause the color of a pixel on the screen to change.

The second device class defined in a modern UNIX system is a *block* device. These are used for hardware, such as a disk drive, that can be accessed as an array of fixed-size blocks of bytes. A file system uses this interface to access disks, and the files are stored in one or more of the blocks.

The third device class is a *stream* device, which is a special type of character device that is used to support communication between computers. Stream devices are packet oriented, in which groups of bytes are considered as a single message to be transmitted through the stream. Streams also support the addition of modules between the input and output ends of the stream. Such modules can be used to convert the messages from the format used by a program into the format used by the communication medium. These modules can also perform other services, such as retransmitting a message in case it is lost.

Each of these device classes requires a software driver to provide an interface between the hardware and the operating system. Programs executing under the operating system may also use the driver to access the hardware. A separate driver is required for each hardware device, since typically each device is unique.

Drivers that do not actually control any hardware, but provide some useful function, may also be included in an operating system. These drivers are called *pseudo device drivers*. A driver that provides terminal emulation is one such driver.

In some operating systems, such as Solaris 2.X, drivers are loaded dynamically when they are used, and unloaded if not used for some set period of time. This allows the operating system to use less memory by not having unused drivers loaded.

RCCL uses two or more device drivers. One of these is used to provide timing for RCCL and is called *clock*. The others are used to interface to the hardware that communicates with the robot or robots. These other drivers are typically named after the hardware device they control, such as the *xyd* driver used to control the Xycom digital I/O board.

Table 4.1 Loadable driver kernel interface functions.

Function	Description
`_init()`	Called when a driver is loaded. Sets up and initializes resources used on a per-driver basis. Does not interact with hardware at this level.
`_fini()`	Called when a driver is unloaded. Releases resources used on a per-driver basis.
`_info()`	Identifies the driver functions to the operating system.

4.2.1 Generic components

Some components of device drivers are found in all drivers used on a particular operating system. These components include a small set of data structures and a set of interface functions. Some of the data structures are used to identify the functions provided by the driver and others are used to support the operation of the driver. Some of the implementation details of the data structures and functions are common to all drivers, but for most functions only the name of the function and the type of its operation are consistent among drivers for different hardware.

For an operating system that supports dynamic loading of device drivers, there are three types of standard functions that are implemented in all drivers. The names of the functions described here are patterned after those used on the Solaris 2.X operating system, though other operating systems use similar names. The first set of standard functions for device drivers is related to the loading and unloading of the driver. These functions, listed in Table 4.1, are used to identify the driver functions to the operating system. The second set of functions includes device configuration functions. These functions, listed in Table 4.2, are used to identify the hardware attached to the system, initialize the device, and initialize the driver for each copy, or *instance*, of the device attached to the system. The third set of functions includes the driver interface functions

Table 4.2 Device configuration functions.

Function	Description
`identify()`	Called to verify the identity of the driver.
`probe()`	Called to verify the existence of the hardware.
`attach()`	Called to attach the driver to an instance of its associated hardware. Sets up driver data structures and an interrupt handler, if necessary.
`detach()`	Performs the inverse operation to attach.
`getinfo()`	Called by the operating system to obtain configuration information known by the driver.

Table 4.3 Device operation functions.

Function	Description
`open()`	Sets up a device for use by a program.
`close()`	Inverse of open.
`read()`	Read data from the device.
`write()`	Write data to the device.
`ioctl()`	Perform a device driver specific operation.
`chpoll()`	Poll a device for events.
`memmap()`	Permits a program to map the registers of a device into the program's address space.

used by programs to access the device. These functions, listed in Table 4.3, support operations such as setting up the device to enable it to be used by the program, transmitting data through the device, polling for events that have occurred and that are supported by the device, mapping the device registers into the address space of the program, and performing driver specific operations. The latter two operations are used by RCCL to support direct access to the communication device by functions in the control level.

Some other functions may also be defined for the driver, depending on the device. These functions may perform operations such as interrupt handling or transfers of data between the device and memory.

4.2.2 The *clock* driver

The *clock* driver provides the timing for RCCL. It can be implemented as a pseudo-device driver, in which a timeout function provided by the operating system serves to implement the timing operation, or it can be implemented using a hardware clock device. Two types of timing operations are used by RCCL. The first of these is the periodic "heartbeat," which drives the clock-based functions of RCCL, such as the *on clock* scheduling mode discussed in Section 2.3.1. The other type of timing operation is an alarm function, which is used for single-shot events such as those used for the *on trigger* scheduling mode.

To support the timing operations, the *clock* driver maintains a monotonically increasing timer variable which is updated at every clock timeout. The clock events are defined with respect to this timer. The driver also implements some control functions that are used by RCCL. These functions perform operations such as setting the clock rate, retrieving the current clock rate, setting up an alarm event, obtaining the current "time," and setting the current time. The timer events are signaled to the program by indicating that an event structure is available to be read from the device. This event structure describes what event has taken place and whether it was a clock event or an alarm event.

4.2.3 The *xyd* driver

The *xyd* driver is an example of a driver used to control a hardware device that allows RCCL to communicate with a robot. The *xyd* driver controls an Xycom digital I/O board, which can be used to communicate with one or two robots. Each robot is connected to the Xycom board by a 16-bit parallel cable. The primary purpose of this driver is to catch interrupt events generated by the Xycom board, which indicate that the robot is requesting attention. These interrupts are signaled to an RCCL program by indicating that a byte of data is available to be read from the device. The actual transmission of data using the device is performed using a different driver included in the RCCL library. This second driver is described in Section 4.3. Another purpose for the *xyd* device driver is to provide exclusive access to the device and therefore to the robot connected to the device. This prevents more than one program from using the robot at the same time.

4.3 Drivers in the RCCL Library

As stated above, RCCL uses two levels of device drivers to communicate with robots. The lower-level drivers are described above. These drivers are used to transmit the interrupt events from the robot communication hardware to the RCCL program. On the other hand, the upper-level drivers are used to perform the communication between the RCCL program and the robot controller.

RCCL uses a two-level driver structure for two reasons. First, the upper-level driver in the RCCL library does not have access to interrupt events from the communication hardware. These interrupts are needed for some of the scheduling modes of RCCL programs. Second, the upper-level driver has less overhead for transmitting data to the robot controller. The extra overhead for the lower-level is due to its integration into the operating system: to use the lower-level driver, the program makes a system call, an operation that takes significantly more time than a function call. In addition, the amount of data to be transmitted on each control cycle is small, on the order of 100 bytes, and can be more effectively transmitted by the upper-level driver.

4.3.1 Structure of an RCCL library driver

The structure of a device driver in the RCCL library is patterned after the structure of device drivers in the operating system. Each library driver implements the following functions: `read()`, `write()`, `test_read()`, `test_write()`, `probe()`, `attach()`, `detach()`, `signal()`, and `control()`. The `read()` and `write()` functions implement the data transfer operation between the RCCL program and the robot. The `test_read()` and `test_write()` functions determine if the robot is ready to send or receive data, respectively. The `probe()` function determines if the communication hardware device is present. The `attach()` function establishes a connection to the robot controller through the communication device. The `detach()` function releases the connection to the robot controller. The `signal()` function sends a one-word command to the robot controller. The `control()` function sets parameters that control the behavior of the driver.

4.3.2 Robot communication

A diagram illustrating how an RCCL program communicates with a puma robot is shown in Figure 4.1. Communication with other robots works similarly, though specific components may change. Each robot has an associated set of communication functions that are specific to the type of robot. These functions maintain the robot specific data structures. When a new set of joint commands is to be sent to the robot, an output function written for the robot takes that set of commands and creates a packet of data to send to the robot. This function then calls the write function of the device driver in the RCCL library to send the data to the robot. When the current state of the robot is to be retrieved from the robot, the robot read function calls the read function of the device driver to retrieve a packet of state data from the robot. Upon receipt of that data, the robot function updates the robot data structures with the new information.

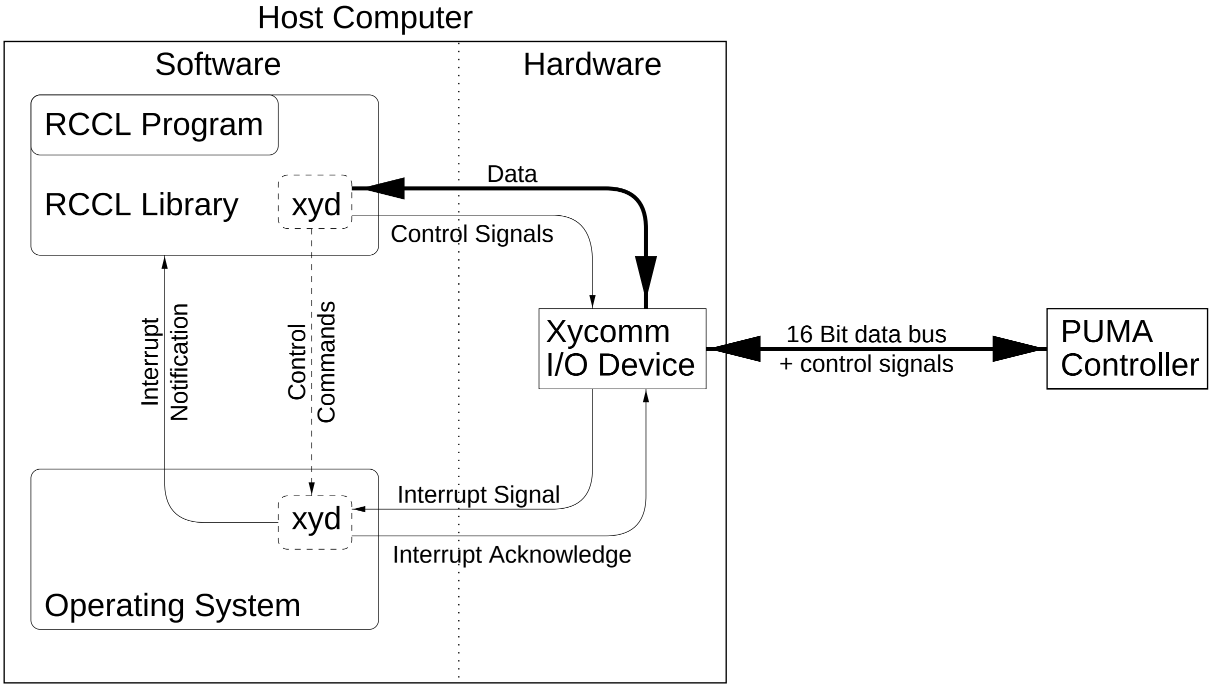

Figure 4.1 A diagram of the robot communication structure of RCCL

During each control cycle, the following two events occur over the communication link between the computer and the robot controller. First, the RCCL program reads the current state of the robot. This is initiated in one of two ways, depending on the scheduling mode for the robot (see Section 2.3.1). For the *on clock* and *on trigger* scheduling modes, the RCCL program initiates the transaction by sending the robot controller a command through the upper-level driver to initiate the transfer of the data. For the *on function* scheduling mode, the robot controller sends a signal to the computer indicating that it has data available. This signal is converted into an interrupt in the I/O device attached to the computer. The interrupt is acknowledged by the lower-level driver and is then sent to the RCCL program. The status data from the robot are then received by the upper-level driver. Second, the RCCL program sends a new set of commands to the robot controller. These commands are also sent through the upper-level driver.

CHAPTER 5

RCCL ON DIFFERENT OPERATING SYSTEMS

5.1 Introduction

Porting programs to a new computer was once a difficult task. Most different types of computers had different operating systems that supported only one set of hardware, and there was little compatibility between systems. The establishment of variants of UNIX as the operating system on a variety of platforms increased the portability of programs by providing a more consistent interface for programs that ran under UNIX. Here, portability is defined as the ability of a program to be compiled without change and to execute with the same results on different systems. Portability in this sense is a quantity to be maximized and is not a binary relation.

There were, however, major differences between early versions of UNIX, especially in terms of process management and in support for multiple processors. For example, most early versions of UNIX operating systems did not have the concept of real-time processing. Therefore, these operating systems required some modifications in order to support RCCL programs. There were also major differences in the interfaces between programs and the operating system in these early versions of UNIX.

In more recent years, operating systems have been evolving to include more of the real-time capabilities that RCCL needs. For example, both Silicon Graphics and Sun Microsystems now have real-time support built into their operating systems. However, there still are differences between the operating systems that require special effort to support. For example, on a Silicon Graphics system, RCCL uses the special shared process group to implement the planning and control levels. But, on a Sun Microsystems system, threads, as described in Chapter 3, are used instead.

The arrival of the standardization process to operating systems and to the C language has increased the ability of systems such as RCCL to be portable. One such standard is the ANSI C standard, which was developed to allow conforming programs to be compiled and executed on any platform, including non-UNIX platforms, provided that the C compiler used on the platform conformed to the standard [3]. However, the ANSI standard is missing some important features, particularly those related to a multiprogrammed environment, such as UNIX, in which multiple programs can execute concurrently. For example, the ANSI standard does not contain the concept of one process sending a signal to another process. Nor does the standard address other forms of interprocess communication.

The POSIX standards are being developed to address the issue of portability among UNIX systems. Each of the POSIX standards applies to one particular area of the operating system. For example, the POSIX.2 standard applies to the set of utility programs that a UNIX system should have. POSIX.5 applies to the interaction between Ada programs and a UNIX operating system. POSIX.9 applies to the interaction between Fortran programs and a UNIX operating system. POSIX.3 applies to methods to test whether a UNIX system complies with a standard. Each of these standards discusses the interface that the operating system should have between itself and either a program executing under it or a user using it. The details of implementing the interface are left to the developers of the operating system, and the details of programming languages using the interface are left to other standards. In this sense, the POSIX standards can be considered as a generalization of UNIX.

Each of the POSIX standards consists of a set of required components and a set of optional components. For an operating system vendor to claim compliance with one of the standards, their operating system must support all of the required components. Initially, most of each standard is listed as required. As a standard evolves to include other features, these new features are listed as optional. However, each optional feature has associated with it an indicator to indicate whether the feature is present in the operating system. For the C programming language, an example of an indicator is a macro defined in a header file included with the operating system. These indicators are required, although the feature they indicate is not.

The most important of the POSIX standards for RCCL is the POSIX.1 standard that addresses the interface between the operating system and a C program. This standard includes much of the ANSI C library as a basis and adds features relevant to a UNIX system. However, POSIX.1 is not a superset of the ANSI C library since the ANSI C library contains some functionality that is not defined in POSIX.1. A discussion of the differences between the ANSI C library and POSIX.1 is given in [41] and [35].

The POSIX.1 standard has been expanded twice since it was first adopted. The first extension, POSIX.1b, includes primitives to support real-time programs such as memory locking, real-time scheduling, and memory-mapped files. The second extension, POSIX.1c, adds support for threaded programs. Since these are extensions to the initial standard, they are optional for operating system implementers. Fortunately, some operating systems have now implemented enough of the optional features to be useful to RCCL.

Two other standards are also important for UNIX, BSD and SVID. The early editions of these two standards predate the POSIX.1 standard and have contributed to its content. The BSD standard was developed at the University of California at Berkeley by the Computer Systems Research Group, and is the basis for the BSD versions of UNIX. The current version of this standard is 4.4BSD [5]. The SVID standard was originally developed by AT&T and is now maintained by Novell, Inc. This standard is the basis for System V UNIX, current variants of which are System V Release Three, commonly referred to as SYSV, and System V Release Four, commonly referred to as SVR4. The current version of SVID is the fourth edition. Other standards also exist for UNIX, though these other standards are derived mostly from those described above. The latest versions of the standards generally include the ANSI standard C library and the required parts of the POSIX.1 standard. Some of the optional parts of the POSIX standard are included in some of the standards and in some operating systems.

The remainder of this chapter is devoted to a discussion of how the ANSI, POSIX, SVID and BSD standards affect the portability of RCCL. Section 5.2 discusses the real-time scheduling requirements of RCCL. Section 5.3 discusses some important aspects of I/O. Section 5.4 completes the discussion with some additional functionality that is required for RCCL but does not fall neatly into separate general categories.

5.2 Real-Time Control Functions

This section describes the three real-time capabilities that RCCL needs in an operating system. The first of these is the ability to create separate threads of control in a process. This is used to split the real-time portion of RCCL, which interacts directly with the robots, from the planning part of RCCL, which may need to cooperate with other processes to plan future motions of the robots. The second capability is the ability to lock a process into memory to prevent it from being paged or swapped to disk. This is one requirement for providing a bounded response time to requests from a robot. The third capability is the ability to change the scheduling of a process to a real-time class. This real-time class needs to prevent lower-priority tasks from interrupting a higher priority task. This is necessary for the control level of RCCL to provide a bounded response time to robot requests. The planning level of RCCL is scheduled as a normal process to allow it to cooperate more effectively with other processes that may be involved in motion planning. These capabilities are described in more detail below.

5.2.1 Threads

As described in Section 3.2, our version of RCCL uses three threads to accomplish different functions. One thread is for signal handling, one is for the control level, and the third is for the planning level. At least two methods for threading programs have become available recently. RCCL contains support for two of them: UI threads as defined in the SVID Fourth Edition, and POSIX threads as defined in POSIX.1c. The UI version is a predecessor to the POSIX version, and the two versions have much in common. The primary difference between the two versions is that the POSIX version allows threads to be canceled, whereas the UI version allows threads to be suspended.

RCCL can also be used on systems that do not support threads of either type but do provide a function similar to `fork()` such as the `sproc()` function of SGI. The important feature of such a function is that two separate processes share the same address space. This is needed because the two processes communicate with each other through this shared memory.

Table 5.1 A list of memory locking functions available on various systems.

System	Lock process	Lock address range
POSIX	`mlockall()`	`mlock()`
	`munlockall()`	`munlock()`
SVID	`plock()`	`memcntl()`
SGI	`plock()`	`mpin()`
		`munpin()`

5.2.2 Memory locking

In order to provide a deterministic response to external events, it is necessary to prevent the operating system from paging out certain parts of a program. If such paging were not prevented, then the response time would depend on whether the page of the program were in memory or not. If the page is not in memory, then the access time for the storage device that contains the page is added to the response time. For applications such as robot control, this is not desirable, since the robot could crash into a wall while the computer waits for the instructions that could have avoided the collision. The memory locking functions found in many UNIX platforms, especially real-time UNIX systems, are designed to avoid this problem. Table 5.1 lists some of the memory locking functions available from different standards and operating systems.

5.2.3 Scheduling control

Another feature required in an operating system to provide a deterministic response to external events is the ability to control the scheduling of the process. In a typical UNIX operating system, most user processes are controlled by a time-share scheduler that attempts to provide fair access to the processor for most processes. It does this by giving short, I/O bound jobs a higher priority than long, processor intensive jobs. It also adjusts the priority of a process as its nature changes. When a process becomes more I/O

Table 5.2 A list of priority control functions available on various systems.

System	Priority control related functions
POSIX	`sched_getscheduler()`, `sched_setscheduler()`
POSIX threads	`pthread_attr_setschedpolicy()`, `pthread_attr_getschedpolicy()`
SVID	`priocntl()`
SVID threads	`thr_getscheduler()`, `thr_setscheduler()`
SGI	`schedctl()`

bound, its priority is increased, and when a process starts to require more processing time, its priority decreases. The `nice()` function allows a process to change its relative priority within this scheme. However, `nice()` is not adequate for RCCL for two reasons. First, parts of an RCCL program must run to completion without interruption, and `nice()` cannot guarantee this. Second, other parts of an RCCL program require a large amount of computation that causes the priority of the process to be reduced. This increases the response time of the parts of RCCL that have to respond quickly to external events. What is needed is a system to give a process a fixed priority that is higher than that of other, lower priority processes to allow that process to run until completion without being interrupted by less important processes. Table 5.2 lists the scheduling control functions available from different standards and operating systems that provide this capability.

5.3 Input and Output Control Functions

For the most part, the standard I/O library for C provides all of the functionality that RCCL needs for processing data files, and the memory mapping interface used for communicating with robot devices is fairly standard for most operating systems. However, there are two areas in which operating systems differ. The first of these is the function used to wait for events to occur on input devices, which are represented as files in UNIX. The types of events for which RCCL waits include timing events, interrupt events from the robots, and requests from the planning thread to the control thread. The other area in which operating systems differ is in the functions and data structures used to control terminal devices such as a serial line or the terminal from which an RCCL program is being executed. These two areas are discussed in the next two subsections.

5.3.1 Waiting for input from files

When RCCL is used to control a robot, it has to wait for events from multiple devices. One of these devices is the *clock* device that provides the clock for the execution of RCCL functions. The other devices are connections to the robots that RCCL is controlling. The RCCL control thread uses a function call that takes a list of these devices and returns when an event has occurred on one or more of them. Unfortunately, neither ANSI nor POSIX have defined a standard function to perform this task. Both BSD and SVID have such functions, but each defines a different function.

BSD defines the function `select()` to poll devices for events. This function takes five arguments. The first specifies the maximum size of the set of file descriptors with which the function should be concerned. The next three arguments specify the set of files for which read events are of interest, the set of files for which write events are of interest, and the set of files for which error events are of interest. The last argument is a timeout parameter that specifies how long the function should wait for something to happen before returning. The return value of the function is the number of files that satisfied the event criteria.

SVID defines the function `poll()` to do a similar task. This function takes three arguments. The first argument is an array of structures in which each structure describes the following: a file to check, the set of events that are of interest for the file, and the set of events that have occurred. The set of events that have occurred is set when the `poll()` function returns. The second parameter specifies the number of elements in the array passed as the first argument. The third parameter specifies a timeout period, similar to the timeout for the `select()` function. The value returned from this function indicates the number of files for which events have occurred.

RCCL can use either of these two interfaces. The type of interface RCCL uses on a given system depends on the value of options in effect at the time RCCL is compiled.

5.3.2 Terminal handling

Terminal control functions are used in two types of interfaces in RCCL. The first type is for communicating with the robot controller via a serial port. In this case, parameters that control the type of serial connection, such as the baud rate, are important. The programs **down** and **termlink** are examples. The second type of interface is for communicating with the user of a program via a terminal interface. In this case, parameters that control the interface mode are important. Examples of programs that use this type of interface are programs such as **primecal** and **free**, which use the C-Tree input parser described in Section 2.6.3 to handle their user interface.

There are at least three different mechanisms defined in different operating systems that allow a program to change the terminal interface. Two of them, *sgtty* and *termio*, operate by passing special parameters to the `ioctl()` function call. The third, *termios*, uses several different functions to perform the same operation. A comparison of these three methods is shown in Table 5.3. This table shows how some of the modes used by RCCL programs can be set in each of the different mechanisms. All three mechanisms use a similar style of interface, in which two function calls are provided to retrieve and set all of the attributes at once. The attributes are stored in a structure that has different

Table 5.3 A comparison of three interfaces for changing some of the characteristics of a communication line.

<table>
<tr><th colspan="2">Get the attributes of a terminal</th></tr>
<tr><td>sgtty</td><td><code>ioctl(fd, TCGETA, &data)</code></td></tr>
<tr><td>termio</td><td><code>ioctl(fd, TIOCGETP, &data)</code></td></tr>
<tr><td>termios</td><td><code>tcgetattr(fd, &data)</code></td></tr>
</table>

<table>
<tr><th colspan="2">Set the attributes of a terminal</th></tr>
<tr><td>sgtty</td><td><code>ioctl(fd, TCSETA, &data)</code></td></tr>
<tr><td>termio</td><td><code>ioctl(fd, TIOCSETP, &data)</code></td></tr>
<tr><td>termios</td><td><code>tsgetattr(fd, &data)</code></td></tr>
</table>

<table>
<tr><th colspan="2">Set the non-canonical mode attribute</th></tr>
<tr><td>sgtty</td><td><code>data.sg_flags |= RAW</code></td></tr>
<tr><td>termio</td><td><code>data.c_lflag &= ~(ISIG | ICANON)</code></td></tr>
<tr><td>termios</td><td><code>data.c_lflag &= ~(ISIG | ICANON)</code></td></tr>
</table>

<table>
<tr><th colspan="2">Set the canonical mode attribute</th></tr>
<tr><td>sgtty</td><td><code>data.sg_flags |= CBREAK</code></td></tr>
<tr><td>termio</td><td><code>data.c_lflag |= ISIG | ICANON</code></td></tr>
<tr><td>termios</td><td><code>data.c_lflag |= ISIG | ICANON</code></td></tr>
</table>

<table>
<tr><th colspan="2">Turn off the character echo attribute</th></tr>
<tr><td>sgtty</td><td><code>data.sg_flags &= ~(ECHO)</code></td></tr>
<tr><td>termio</td><td><code>data.c_lflag &= ~(ECHO)</code></td></tr>
<tr><td>termios</td><td><code>data.c_lflag &= ~(ECHO)</code></td></tr>
</table>

<table>
<tr><th colspan="2">Turn on the character echo attribute</th></tr>
<tr><td>sgtty</td><td><code>data.sg_flags |= ECHO</code></td></tr>
<tr><td>termio</td><td><code>data.c_lflag |= ECHO</code></td></tr>
<tr><td>termios</td><td><code>data.c_lflag |= ECHO</code></td></tr>
</table>

<table>
<tr><th colspan="2">Set the speed attribute</th></tr>
<tr><td>sgtty</td><td><code>data.sg_ispeed = speedcode</code>, <code>data.sg_ospeed = speedcode</code></td></tr>
<tr><td>termio</td><td><code>data.c_cflag &= ~(CBAUD)</code>; <code>data.c_cflag |= speedcode</code></td></tr>
<tr><td>termios</td><td><code>cfsetospeed(&data, speedcode)</code>, <code>cfsetispeed(&data, speedcode)</code></td></tr>
</table>

fields for different terminal settings. These attributes are changed in a similar manner in all three mechanisms, primarily by setting or clearing the bits in the appropriate fields. More details about the meaning of the modes can be found in other sources [35], [5], [2], [34].

5.4 Miscellaneous Functions

There are a number of other areas in which operating systems differ that have an impact on RCCL. These areas include the following: signals, string functions, variable numbers of function parameters, random numbers, and the size of a page of memory. These are described in the following sections.

5.4.1 Signals

Signals and signal handling comprise the area in which there is perhaps the greatest difference between operating systems. The mechanism described by the ANSI C standard is perhaps the lowest common denominator among all operating systems. It provides two functions, one to establish a handler for a signal and the second to allow a program to send a signal to itself. ANSI does not provide a way for the program to block signals from delivery. The types of signals that a program may receive are also limited to those that indicate an exceptional condition in the program, such as an invalid address or an invalid operand to a math function. A program may also receive an interrupt signal from the user of the program. The result is that a program that complies with the restrictions of ANSI C is portable to any operating system for which an ANSI C compiler exists, including non-UNIX systems.

The ANSI system is more suited to environments in which only a single program executes at a time. In a UNIX environment, however, a number of programs execute

Table 5.4 A comparison of signal handling methods in different standards.

Function	BSD	SVID	ANSI	POSIX

Send signal	`kill()`	`kill()`
		`sigsend()`
		`sigsendset()`
			`raise()`	`kill()`
				`sigqueue()`

Install handler	`signal()`
	`sigvec()`
		`signal()`
		`sigset()`
		`sigignore()`
			`signal()`	`sigaction()`

Mask type	`int`	—	—	`sigset_t`
Mask manipulation	`sigsetmask()` `sigblock()`	`sighold()` `sigrelse()`		`sigprocmask()`

Wait for signal	`sigpause()`	`sigpause()`		`sigsuspend()`
				`sigwait()`
				`sigwaitinfo()`
				`sigtimedwait()`

Non-local goto	`setjmp()`
	`longjmp()`
	`_setjmp()`
	`_longjmp()`
		`setjmp()`
		`longjmp()`
			`setjmp()`
			`longjmp()`
				`sigsetjmp()`
				`siglongjmp()`

Other	`siginterrupt()`
	`sigstack()`
	`sigreturn()`
		`sigaltstack()`		`sigpending()`

simultaneously, and some of these programs communicate with other programs by using signals. UNIX extends the set of signals for this purpose, including signals for job control and other forms of communication. In addition, UNIX adds more functions for handling signals to allow a program to block signals or to affect what happens to a program when a signal is delivered. A listing of the relevant functions provided by ANSI, BSD, SVID, and POSIX is shown in Table 5.4. This table has been arranged to illustrate the differences between the standards. The POSIX method of signal handling has been integrated into the later versions of the BSD [5] and SVID [34] standards.

There are differences between the way BSD handles signals and the way SVID does. See Section 3.1.1 for the definitions used in this discussion. When the function `signal()` is used to install a handler for a process, the results are different for SVID and BSD. In

SVID, when the signal is delivered to the process and a handler has been defined for the signal, the disposition is reset to the default and then the handler is executed. Because of this, the first operation performed by most signal handlers is to reset the disposition of the signal to call themselves. In BSD, the disposition for the signal remains the same. In addition, in BSD, the signal that caused the handler to be executed is blocked from delivery until the handler exits, unless explicitly unblocked by the handler. In SVID, this is not the case and the signal remains unblocked. This difference in SVID creates a timing window between the operating system setting the disposition of the signal to the default and the signal handler resetting the disposition to call itself. A second occurrence of a signal during this window may cause the process to exit as a result of the default action for the signal. Because of this, SVID signal handlers set by the `signal()` function are considered unreliable.

The difference for handling signal blocking upon entrance to a signal handler is reflected in `setjmp()` and `longjmp()` functions. These functions provide a way to handle certain types of exceptions when a program knows such an exception is possible. In BSD, the `setjmp()` saves the signal mask at the time the function was called, and the `longjmp()` function restores this mask. In SVID, since a signal is not blocked by default upon entrance to a signal handler, the `setjmp()` function does not save the signal mask and the `longjmp()` function does not restore the mask.

Another difference between BSD and SVID signal handlers is in the number and types of parameters passed to the signal handling function. The signal handler type for SVID is the same as that specified by the ANSI standard: the return type of the function is `void` and the function accepts a single argument of type `int` that contains the number of the signal that caused the function to be executed. BSD extends the interface to this function to include two more parameters, one to specify a code that explains the reason for the signal and one that describes what the program was doing when the signal was delivered.

A third difference between BSD and SVID is in how the signal masks are maintained. In SVID, only the functions `sighold()`, which blocks a signal from delivery, and `sigrelse()`, which unblocks a signal, are available. No direct method for accessing the signal mask is provided. BSD, on the other hand, provides the function `sigsetmask()` to manipulate the signal mask directly. A second function, `sigblock()`, is provided to add a set of signals to the mask.

In developing a standard method for signal handling, POSIX defined a new interface instead of attempting to reconcile the differences found in the BSD and SVID methods. The original POSIX.1 functions draw heavily on the BSD version of signal handling, especially the reliability of signal delivery, with several differences. First, the data type for specifying the signal mask was changed from an `int` to a structure type to allow a UNIX implementation to have more than 32 signals. Second, POSIX does not require some features of signal handling that BSD provides, such as alternate stacks for the signal handlers. Third, POSIX added a function, `sigpending()`, to determine if a blocked signal is pending.

The first extension to POSIX.1, POSIX.1b adds several more functions to provide some capabilities to signals that are desired for real-time systems, such as the ability to queue signals, the ability to send more information along with the signal to the handler function, and the ability to wait for these signals.

The second extension, POSIX.1c, adds the `sigwait()` function to allow individual threads to wait for signals. The new functions for waiting for signals have a different response from the `sigsuspend()` function. When the `sigsuspend()` function is called and a signal arrives for which the program is waiting, the handler for that signal is called before the `sigsuspend()` function returns. For the new functions, the handler is not

Table 5.5 A comparison of character string handling functions between the pre-ANSI versions and the ANSI versions.

Pre-ANSI C	ANSI C	RCCL
`<strings.h>`	`<string.h>`	`<RCCLstring.h>`
`bcopy()`	`memmove()`	`RCCL_memmove()`
	`memcopy()`	`RCCL_memcopy()`
`bzero()`	`memset()`	`RCCL_memset()`
`bcmp()`	`memcmp()`	`RCCL_memcmp()`
`index()`	`strchr()`	`RCCL_strchr()`
`rindex()`	`strrchr()`	`RCCL_strrchr()`
`sys_errlist[]`	`strerror()`	`RCCL_strerror()`

called and the waiting function returns with an indication of which signal has arrived. A complete rational for the POSIX system can be found in [35].

Because of the differences between all of these standards, RCCL does not support user applications using signal handlers which expect more than a single argument specifying the number of signals. One other limitation of RCCL is that some signals have been historically used by RCCL for its own communication and should not be used by programs. Otherwise, user programs are free to install their own signal handlers.

5.4.2 String functions

Shown in Table 5.5 is a set of string manipulation functions that are available on many systems. Before the ANSI C standard, some systems supported the functions shown in the pre-ANSI column of the table, while other systems supported the functions in the ANSI column. Some of these functions provide operations on blocks of memory that are optimized for a given computer architecture. For example, the `bcopy()`, `memcopy()`, and `memmove()` functions copy a block of memory from one location to another, the `bcmp()` and `memcmp()` compare blocks of memory, and `bzero()` and `memset()` functions initialize a block of memory. Others provide a means of locating particular characters in a sequence of characters. The `index()` and `strchr()` functions find the first occurrence of a character from the beginning of a string, while the `rindex()` and `strrchr()` functions find the first occurrence from the end of the string. The last function returns a string that describes error numbers that are returned from system function calls. The ANSI function is `strerror()`, which returns a pointer to a string describing the error, while the `sys_errlist` is an array of pointers to strings indexed by the error number.

There are some small differences between some of these functions. For example, the `bcopy()` and `memmove()` functions do operate correctly on overlapping regions of memory, but the `memcopy()` function does not. The order of the parameters to these three functions differs as well. The `bcopy()` takes the source operand as its first parameter followed by the destination operand as its second parameter. The `memmove()` and `memcopy()` take these parameters in the opposite order. There is also a difference between the `bzero()` and `memset()` functions. The `bzero()` function will set the value to zero of each byte in a block of bytes, whereas the `memset()` allows this value to be a number other than zero.

All of these string functions perform simple operations that can be easily implemented by any programmer. The purpose for including them in a standard C library is that the library versions can be designed for maximum efficiency on each given computer. Prior to the ANSI standard, some systems provided the functions shown in the left column of Table 5.5, some systems provided those in the center column, and some provided neither. With the ANSI standard, the functions listed in the center column are provided by any operating system conforming to the standard.

Since not all systems provide the ANSI standard functions, RCCL provides macros that give access to the correct functions that are available on the system. These macros are expanded either to a call to the appropriate function that is available on the system or to a call to a function in the RCCL library that provides the appropriate function. On a system that provides neither version of these functions, the macros become the names of functions, and code to implement these functions becomes part of the RCCL library. These macros are shown in the right-hand column of Table 5.5.

5.4.3 Variable number of parameters to functions

Before the development of the ANSI C standard, there was no truly portable method to define a function that accepted a variable number of parameters. However, a *de facto* method emerged that was compatible with previous C style for function definition [21]. With the ANSI standard came a new method for defining these functions. This new method is compatible with the new style of function declaration that is defined in the ANSI standard [22], [3].

A comparison between these two methods is shown in Figure 5.1, which highlights their differences. This example shows the `makePosition()` function of RCCL that creates a position equation from a variable number of transform matrices. In the old style of C, a function prototype is used only to declare the return type of the function. Because of this, no parameters are listed inside the parentheses. In ANSI C, the function prototype lists the type of each parameter accepted by the function. Each parameter whose type is listed in this manner is called *fixed*. For ANSI C, an ellipsis is used at the end of the list of fixed parameters to indicate that the function accepts more parameters than are listed. As shown in Figure 5.1(b), the `makePosition()` function takes a character pointer as its fixed parameter.

At the function definition, only the names of the variables are listed inside the parentheses in the old style of C. The types of these variables are listed after the closing parenthesis and before the opening curly brace of the function. In this case, the variable name `va_alist` is used as the only variable for the function and `va_dcl` is used as type

```c
#include <varargs.h>
...
/* Function Prototype */
int makePosition();
...
/* Function Definition */
int makePosition(va_alist)
va_dcl
{
   char *name;
   TRSF *t1;
   va_list ap;
   va_start(ap);
   ...
   /* Parameter 1 */
   name = va_arg(ap, char *);
   ...
   /* Parameter 2 */
   t1 = va_arg(ap, TRSF *);
   ...
   va_end(ap);
   return 1;
}
```

(a) Old style C

```c
#include <stdarg.h>
...
/* Function Prototype */
int makePosition(char *, ...);

...

/* Function Definition */
int makePosition(char *name, ...)
{
   TRSF *t1;
   va_list ap;
   va_start(ap, name);
   ...
   /* Parameter 2 */
   t1 = va_arg(ap, TRSF *);
   ...
   va_end(ap);
   return 1;
}
```

(b) ANSI C

Figure 5.1 A comparison between the old style C and the ANSI C interfaces for specifying a variable number of parameters using the `makePosition()` function of RCCL.

of the variable. In ANSI C, the function definition is the same as the prototype, except that the variable names are listed for each fixed type.

Inside the function, though, the interfaces are similar. For example, the same variable type (`va_list`) is used to declare a variable that is used to step through the list of parameters, and the same macros are used to step through the list of parameters (`va_arg`) and to finish parameter processing (`va_end`). The primary difference is that the macro that is used to set up the parameter list variable (`va_start`) uses a different number

```
#include <RCCLvararg.h>
...
int makePosition PROTO((char *, ...));
...
int makePosition ONE_ARG_VARARG(char *, name)
{
   TRSF *t1;
   VARARG_START(ap, name);
   ...
   t1 = va_arg(ap, TRSF *);
   ...
   va_end(ap);
   return 1;
}
```

Figure 5.2 The `makePosition()` function using the macros provided by RCCL

of parameters in the two versions. In the old style of C, this macro accepts only one parameter: the name of the parameter list variable. In ANSI C, this macro takes two parameters: the name of the parameter list variable, and the name of the last fixed parameter of the function.

RCCL provides a set of macros to allow functions to be defined using either version. These macros take advantage of the fact that the interface used in the old version of C does not preclude other fixed parameters to be listed in the function definition. The `makePosition()` function is shown again in Figure 5.2 to illustrate the use of these macros. Under an ANSI C compiler, these macros reduce to the version shown in Figure 5.1(b), otherwise the result is similar to the version shown in Figure 5.1(a). A list of these macros and their expansions under the two versions of C is shown in Table 5.6.

Table 5.6 A list of the macros defined by RCCL used for functions that accept a variable number of parameters.

RCCL Macro	Evaluates to: Old style C	ANSI C
`PROTO((type1, type2, ...))`	`()`	`(type1, type2, ...)`

`ONE_ARG_VARARG(type1, var1)`	`(var1, va_alist)`
	`type1 var1;`
	`va_dcl`
		`(type1 var1, ...)`

`VARARG_START(ap, varn)`	`va_list ap;` `va_start(ap);`	`va_list ap;` `va_start(ap, varn);`
`RESTART_START(ap, varn)`	`va_start(ap);`	`va_start(ap, varn);`

5.4.4 Other functions

Two other sets of functions that are not standard among UNIX operating systems are used in RCCL. The first of these obtains the size of a memory page and the second provides a means to obtain a stream of pseudo-random numbers.

The system function `getpagesize()` returns the number of bytes in a page of memory. A page of memory is the smallest unit of memory that the operating system manages. This number is used in RCCL to provide memory management using a more efficient size of memory block and to provide page-aligned addresses to some system calls that require them. In some UNIX operating systems and in POSIX.1, this function has been replaced by a call to the 'sysconf' function that allows a program to obtain a variety of configuration information from the running operating system in addition to the page size. In the VXWorks environment, neither of these functions is available, so the page size is assumed to be 1024. RCCL defines a macro, RCCL_getpagesize(), to simplify access to this number.

Random numbers are used by various test programs and demo programs to provide more interesting test cases. The function `rand()` has been historically provided on most operating systems, but its 16 bit resolution is too poor to be of practical use by most programs. Fortunately, both BSD and SVID specify functions to provide a better stream of random numbers, though the set of functions provided is different. BSD defines the function `random()` that can generate a 31 bit random number in a long integer format. SVID defines a set of functions for random numbers that can provide up to 48 bits of resolution in different formats and using different parameters. These functions are `drand48()` and `erand48()`, which provide random doubles in the range $[0.0, 1.0)$; `lrand48()` and `nrand48()`, which provide random long integers in the range $[0, 2^{31})$; and `mrand48()` and `jrand48()`, which provide random long integers in the range $[-2^{31}, 2^{31})$. This set of functions also demonstrates how standards can change over time. According to the third edition of SVID, these functions are obsolete, but in the fourth edition of SVID, these functions are listed as standard requirements for a library. RCCL will use either of these functions if they are present on a system, or if neither is provided, RCCL has its own implementation of a function to produce a random float in the range $[0.0, 1.0)$.

CHAPTER 6

FUTURE DIRECTIONS

There are three areas in which the implementation of RCCL presented in this thesis can be improved. These areas are signal handling, thread safety, and the use of additional threads in RCCL.

6.1 Signal Handling

There are two areas in which signal handling can be improved in RCCL. The first of these is to allow user programs to register their signal handlers with the signal thread in RCCL. This would then remove the need to recast signals, which is the current practice of the signal thread. This would improve the reliability of the receipt of signals in the planning level since the recasting of the signal may cause two closely spaced signals to be seen as one signal.

The second area in which signal handling can be improved is to allow signal handlers to receive extra information along with the signal number, as is the case for some UNIX operating systems. This extra information would then be registered with RCCL and would be passed to the function when the corresponding signal arrives. This option would then support the newer signal delivery systems which pass extra information to the handler functions.

6.2 Thread Safety

The implementation of RCCL presented in this thesis is a multithreaded program. Unfortunately, this implementation also assumes that the three threads it defines are the only threads in running in the program. This assumption is reflected in the way in which many RCCL functions are written. These functions assume one thread for the planning level and a limited number of threads for the control level. The number of threads at the control level is one for most operating systems, except for the multiprocessor MicroVAX system [33]. These functions, therefore, cannot be considered thread safe themselves. A future implementation would change these assumptions and make these functions thread safe. This would allow a user of RCCL to build a multithreaded program on top of the RCCL library without having to be concerned about which threads access RCCL functions.

6.3 Additional Threads

There is no particular reason that RCCL should be restricted to three threads. In particular, there are two ways in which the number of threads can be increased that would increase the amount of parallelism that can be obtained from RCCL. This is more important in the case in which RCCL is used to control more than one robot and is running on a host computer which has more than one processor. The first of these methods to increase the number of threads is to use multiple threads in the control level in place of the event loop. In this case, there would be one thread to handle requests from the planning level, one thread to process timer events, and one thread for each of the robot interfaces. This approach would also remove the need for the functions described in Section 5.3.1.

The second way to increase the potential for parallelism in RCCL is to have one thread per control task. This would be combined with the above multithreaded implementation of the control level. These additional threads can use condition variables to allow the timing thread to indicate timing events, or the task functions can be executed by the robot interface threads, depending on the scheduling mode selected for the robot.

CHAPTER 7

CONCLUSION

Presented in this thesis was RCCL, a system for controlling robots. First, the features of RCCL were discussed. These features include the following: position-based control using position equations, sensor integration capability, tunable trajectory generation, tunable error handling, a teach mode, a simulator, a simple command completion interface, and a set of demonstration and utility programs. A robot that is to be controlled by RCCL can have up to six joints if its motions are to be controlled by the trajectory generator, or up to 12 joints otherwise. In addition, control level tasks can be created by the user. These tasks perform operations such as trajectory generation for one or more robots. Associated with each task is a scheduling mode, and tasks may communicate with other tasks.

Next, the threaded implementation of RCCL was discussed. In this implementation, three threads are used: one for the control level, one for the planning level, and one for handling signals. The control thread runs as a real-time thread, but the planning and signal threads do not. The control thread is implemented as an event loop, waiting for events that trigger the execution of a task.

There are two types of device drivers used by RCCL, one for timing, and the other for communicating with robot controllers. The timing driver is integrated into the operating system, and supports the timing operations required by RCCL. Two layers of drivers are used to communicate with the robot controllers. One layer is integrated into the operating system and is used to process interrupt events from the hardware used to communicate with the robot. The second layer is incorporated into the RCCL library and is used to perform most of the communication between an RCCL program and the robot controller.

RCCL must address the differences in the interfaces presented by different operating systems. The important interfaces for RCCL are those for real-time control functions, input and output control functions, signals, and string functions. The real-time functions include those that manipulate threads, those that perform memory locking, and those that perform scheduling control. The input and output functions include those that wait for input from files and those that control the characteristics of a terminal.

Several possible improvements can be made to RCCL. These improvements include modifications to how signals are processed by an RCCL program, modifications to allow the RCCL library to be used by multithreaded programs, and modifications to use more threads. These are the subjects of on-going research.

REFERENCES

[1] A. Adler. TDL: A task description language for programming automated robotic workcells. In K. Rathmill, editor, *Control and Programming in Advanced Manufacturing*, International Trends in Manufacturing Technology, pages 321–359. Springer-Verlag, Berlin, 1988.

[2] American Telephone and Telegraph Company. *System V Interface Definition (SVID).* UNIX Press, Morristown, NJ, third edition, 1989.

[3] ANSI/ISO 9899:1990. *American National Standard for Programming Languages — C*, 1990.

[4] J. R. Blaha, J. P. Lamoureux, and K. E. McKee. Currently available robot programming languages. In K. Rathmill, editor, *Control and Programming in Advanced Manufacturing*, International Trends in Manufacturing Technology, pages 67–83. Springer-Verlag, Berlin, 1988.

[5] Computer Systems Research Group, University of California at Berkeley. *4.4BSD Programmer's Reference Manual.* O'Reilly & Associates, Inc., Sebastopol, CA, April 1994.

[6] P. I. Corke and R. Kirkham. The ARCL robot programming system. In *Proceedings of International Conference of the Australian Robot Association*, pages 484–493, 1993.

[7] N. Delson and H. West. Robot programming by human demonstration: Adaptation and inconsistency in constrained motion. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 30–36, 1996.

[8] B. O. Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

[9] A. A. Goldenberg and L. Chan. An approach to real-time control of robots in task space. application to control of a PUMA 560 without VAL-II. *IEEE Transactions on Industrial Electronics*, 35(2):231–238, May 1988.

[10] M. A. Goodman, M. Goyal, and R. A. Massoudi. *Solaris Porting Guide.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[11] R. Guptill and P. Stahura. Multiple robotic devices: Position specification and coordination. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 1655–1659, 1987.

[12] S. Hayati, T. Lee, K. Tso, P. Backes, and J. Lloyd. A testbed for a unified teleoperated-autonomous dual-arm robotic system. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 1090–1095, 1990.

[13] S. Hayati, T. Lee, K. Tso, P. Backes, and J. Lloyd. A unified teleoperated-autonomous dual-arm robotic system. *IEEE Control Systems Magazine*, 11(2):3–8, 1991.

[14] V. Hayward and S. Hayati. KALI: an environment for the programming and control of cooperative manipulators. In *Proceedings of the American Control Conference*, pages 473–478, 1988.

[15] V. Hayward and R. P. Paul. Robot manipulator control under UNIX - RCCL: A robot control "C" library. *International Journal of Robotics Research*, 5(4):94–111, 1986.

[16] S. A. Hutchinson and A. C. Kak. FProlog: A language to integrate logic and functional programming for automated assembly. In K. Rathmill, editor, *Control and Programming in Advanced Manufacturing*, International Trends in Manufacturing Technology, pages 361–371. Springer-Verlag, Berlin, 1988.

[17] Y. K. Hwang, P. C. Chen, and P. A. Watterberg. Interactive task planning through natural language. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 24–29, 1996.

[18] K. Ikeuchi. Assembly plan from observation. In *Proceedings of International Electronic Manufacturing Symposium*, pages 9–12, 1995.

[19] S. B. Kang and K. Ikeuchi. Robot task programming by human demonstration: Mapping human grasps to manipulator grasps. In *International Conference on Intelligent Robots and Systems*, pages 97–104, 1994.

[20] S. B. Kang and K. Ikeuchi. Robot system that observes and replicates grasping tasks. In *IEEE International Conference on Computer Vision*, pages 1093–1099, 1995.

[21] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, 1978.

[22] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[23] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads.* Prentice Hall, Upper Saddle River, NJ, 1996.

[24] D. Kossman and A. Malowany. Multi-processor robot control system for RCCL under IRMX. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 1298–1306, 1987.

[25] J. S. Lin, S. Hayati, V. Hayward, and J. Lloyd. Implementation of RCCL, a robot control C library, on a MicroVAX II. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 726, pages 472–480, 1986.

[26] J. Lloyd. *RCCL/RCI Hardware Installation Notes.* McGill University, Montréal, Québec, Canada, 1992.

[27] J. Lloyd. *RCCL/RCI Startup and Installation Guide.* McGill University, Montréal, Québec, Canada, 1992.

[28] J. Lloyd. *RCI User's Guide.* McGill University, Montréal, Québec, Canada, 1992.

[29] J. Lloyd and V. Hayward. Real-time trajectory generation using blend functions. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 784–789, 1991.

[30] J. Lloyd and V. Hayward. *Multi-RCCL User's Guide.* McGill University, Montréal, Québec, Canada, April 1992.

[31] J. Lloyd and V. Hayward. Real-time trajectory generation in Multi-RCCL. *Journal of Robotic Systems*, 10(3):369–390, April 1993.

[32] J. Lloyd and V. Hayward. Trajectory generation for sensor-driven and time-varying tasks. *International Journal of Robotics Research*, 12(4):380–393, August 1993.

[33] J. Lloyd, M. Parker, and R. McClain. Extending the RCCL programming environment to multiple robots and processors. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 465–469, 1988.

[34] Novell, Inc., Provo, UT. *System V Interface Definition (SVID)*, fourth edition, 1995.

[35] POSIX 1003.1b-1993. *Portable Operating System Interface (POSIX) Part 1: System Application Program Interface Amendment 1: Realtime Extension [C Language]*, 1993.

[36] Prentice Hall, Englewood Cliffs, NJ. *SunOS 5.3 Writing Device Drivers*, 1994.

[37] B. E. Shimano, C. C. Geschke, C. H. Spalding, III, R. Goldman, and D. W. Scarborough. AIM: A task-level control system for assembly. In K. Rathmill, editor, *Control and Programming in Advanced Manufacturing*, International Trends in Manufacturing Technology, pages 305–320. Springer-Verlag, Berlin, 1988.

[38] U. Vahalia. *UNIX Internals: The New Frontiers.* Prentice Hall, Upper Saddle River, NJ, 1996.

[39] R. A. Volz, T. N. Mudge, and D. A. Gal. Using Ada as a programming language for robot-based manufacturing cells. In K. Rathmill, editor, *Control and Programming in Advanced Manufacturing*, International Trends in Manufacturing Technology, pages 329–359. Springer-Verlag, Berlin, 1988.

[40] F. Zanichelli, S. Caselli, A. Natali, and A. Omicini. A multi-agent framework and programming environment for autonomous robotics. In *Proceedings of IEEE Conference on Robotics and Automation*, pages 3501–3507, 1994.

[41] F. Zlotnick. *The POSIX.1 Standard: A Programmer's Guide.* The Benjamin/Cumming Publishing Company, Inc., Redwood City, CA, 1991.