Multi-RCCL User's Guide

John Lloyd and Vincent Hayward

April 1992

Computer Science Department
University of British Columbia
Vancouver, B.C., Canada

Postal Address: 201-2366 Main Mall, University of British Columbia, Vancouver, B.C. V6T 1Z4
Phone: (604) 822-5109 FAX: (604) 822-5485
Bug Reports: lloyd@cs.ubc.ca

RCCL/RCI Release 5.0, June 28,1996

Multi-RCCL User's Guide

John Lloyd and Vincent Hayward

Abstract

This manual is a tutorial-style document which describes the robot programming system Multi-RCCL, Release 4.0. The primary sites for this system are McRCIM (McGill University) and the Jet Propulsion Laboratory (JPL).

Acknowledgements

Support for the development of RCCL has been provided by McGill University (Montreal), the Jet Propulsion Laboratory (Pasadena), and the General Electric Advanced Technology Laboratories (Moorestown, N.J.). The original version was written at Purdue University in 1983.

I would like to put in a word for the numerous people who contributed to RCCL over the years.

Foremost mention goes to Vincent Hayward, who wrote the first version and provided many ideas and suggestions for subsequent releases.

I would also like to thank Mike Parker (McGill University) for his continual system support, suggestions, quips, and document "debugging". Samad Hayati (JPL) has provided much of the motivation for further developments of the system. Paul Backes, Tom Lee, and Kam Tso (JPL) implemented numerous applications, and thereby provided much of the motivation for adding extra features. Rick McClain (General Electric) participated in the design of the multi-robot capabilities, and Jin Lee and Bob Russell (General Electric) implemented many of the tracking and force feedback applications. Chafye Nemri (McGill) proof read the latest version of this manual.

Thanks should also be extended to those people at McGill who helped me bring up RCCL in the "early days": to Thong for (frequent) hardware support, fixing joint 2 when it "blew" and helping us "stretch" the Q-bus, to Frank for convincing us we *could* stretch the Q-bus, as well as take other liberties with the hardware, to Roy for general and moral support, especially the day when we "upgraded" the controller cabinet, to Paul and Gregory who built the VAX-robot "FIFO" interface (which, despite darker forecasts, operated flawlessly), to Rick and David who took time to disassemble the code for the joint microprocessors, and to Don Kossman for his trepitude in porting the early system to iRMX.

Thanks is given to Larry Alexander and John Day (Digital Equipment Corp.), for their help in convincing the rest of DEC that we had, in fact, found a bug in the microVAX CPU hardware, and special thanks

Finally, special thanks is given to Isobel Mackie, for her frequent help with "last minute" chores during the dead-line crunches.

— J.L.

Contents

1. Introduction

Multi-RCCL is a C package for implementing task level robot control applications under UNIX-like operating systems. It provides data types useful for robotics applications, such as vectors and spatial coordinate transformations, in combination with routines to specify robot arm motions. Movements can be requested to target positions in either Cartesian or joint coordinates, and several arms can be operated from the same program. Arm trajectories are created by a special background task which runs at a fixed sample rate (usually 50 to 200 Hz., depending on the system). Application routines can be defined which modify the trajectories based on real-time sensor inputs.

The output from the trajectory task consists of a set of joint-level position commands, which are assumed to be implemented by some low level servo controller (such as the 1 Khz PID controllers for the PUMA). RCCL cannot be used to implement algorithms at the joint servo level, although people doing such work might find it interesting to place RCCL *on top* of such a system [1]. Depending on the control rates required, it is sometimes possible to implement servo level algorithms in RCI (the level beneath RCCL); those interested should consult the *RCI User's Guide*.

1.1 History

RCCL (which stands for Robot Control C Library) was originally written at Purdue University in 1983 by Vincent Hayward ([Hayward and Paul 1986]). This might be called Release 1.0. A slightly enhanced version (Release 2.0) was produced at McGill University in 1985 ([Lloyd 1985]). Significant changes were undertaken between 1987 and 1988, when the multi-robot and multi-CPU capabilities were added ([Lloyd, Parker, and McClain 1988]). This work was mainly undertaken at McGill University and at the General Electric Advanced Technology Laboratory (New Jersey) under contract to the Jet Propulsion Laboratory (JPL). The version delivered to JPL in the fall of 1988 was defined to be Release 3.0. The work done in creating the current Release 4.0 consisted mostly of fixing bugs, improving the documentation, and doing a general "cleanup". This was done mostly during the fall of 1989. The designation "Multi-RCCL" is sometimes applied to releases starting at 3.0.

Release 5.0 is different from previous releases primarily in the underlying implementation. First, the primary host operating systems, Sun Solaris and SGI IRIX, now come with intrinsic real-time capabilities. This means that many of the quirks and restrictions of previous releases, which were incurred by special measures necessary to support real-time tasks under UNIX, are no longer necessary. In particular, the `rcc` compilation interface, and the prohibition on calling system services from the control level, no longer exist.

[1] RCCL could then be used to drive the servo level. This has in fact been done at the Jet Propulsion Laboratory. Interfacing RCCL to a lower level system entails some code work and is beyond the scope of this manual, but is not technically difficult. Consult the *RCI User's Guide* and the *RCCL/RCI Startup and Installation Guide* for details on what the low level interface looks like.

There still are some restrictions on what can be called from the control level. For a summary, see Section 5.3.

1.2 Required Background

It is always difficult to describe the necessary background for using a technically oriented system like RCCL. One should certainly be familiar with the C programming language [Kernighan and Ritchie 1978], and also with the UNIX operating system. A thorough understanding of the robot control and programming techniques described by Paul in [Paul 1981] is also highly desirable. Paul's book is particularly important, since RCCL is in many ways just an implementation of the ideas described there (although one can probably omit chapters 6, 7, and 10 on first reading). Some concepts relating to real-time programming might also be useful; for instance, it would be good if the user were familiar with the terms "data access collision" and "re-entrant code". There are many books available on this subject; this author's favorite is [Allworth 1981].

1.3 System Architecture

An RCCL program, when it runs, sets up a real-time background task that generates the trajectories necessary to satisfy the motion requests specified by the main part of the program. The background task, which we will generally refer to as the *trajectory generator*, executes independently of the main program at a periodic sample rate (50 Hz. is common), and is mostly invisible to the RCCL programmer. However, there are ways for an RCCL program to provide application routines that are executed within the trajectory generator. Such routines are run in real-time, periodically, at the same rate used by the trajectory generator, and can be used for applications involving control or sensor feedback.

The main RCCL program, which executes like any other UNIX program, is often referred to as the *planning level*, or *planning task*. The trajectory generator, plus any application routines running within it, is usually referred to as the *control level*. Communication between the planning and control level is implemented using shared memory (see Figure 1).

The trajectory generator is implemented using RCI (Real-time Control Interface), which is a facility for spawning high-priority real-time tasks from a UNIX program (see the *RCI User's Guide*). The code that implements the trajectory task is loaded into the RCCL program along with all the other library routines. In previous releases of RCCL, the control task was not a UNIX process, and so code executing at the control level did not have access to UNIX system services. However, the UNIX systems on which Release 5.X is based now have intrinsic real-time support, with control tasks generally being implemented as light weight UNIX process. Control level code therefore has access to most UNIX system services, subject only to the caveat that calling various system services from the control level may cause unacceptable time delays. The support functions contained in earlier releases to compensate for the lack of system services have been left in place; see Section 5.3.

The trajectory generator operates one or more robots. At the start of each trajectory cycle, joint position and other sensor information is obtained from the joint-level controller associated with

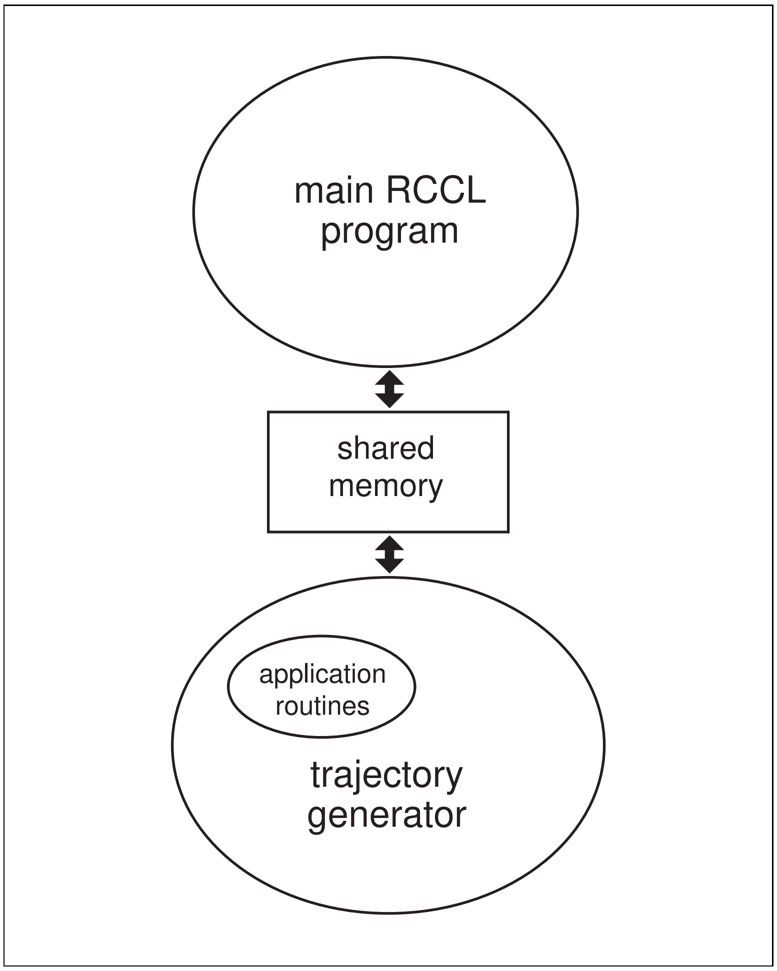

Figure 1: The main RCCL program and the trajectory generator.

each robot. Trajectory computations are then performed, with the appropriate low-level setpoint commands being sent to the joint controllers some time later in the cycle. A more detailed description of this sequence of operations is given in Section 6.3.2.

It is also possible for the trajectory generator to control *virtual* manipulators, which are not attached to any physical robot but can be used to coordinate the motions of other robots (see Section 8.2).

For simple motions, the trajectory generation is an open-loop process, with the joint position feedback serving mainly as a check against collisions and to allow the application program to compensate for servoing errors in the robot's position. However, more complex applications may use joint position and senor information in more complex ways, such as to implement guarded motions or impedance control. The trajectory generator then becomes part of a higher-level feedback loop.

Within an RCCL program, most of these details concerning trajectory generation are hidden from the user. Communication with the trajectory generator is done through functions and a few special data structures. As indicated above, application-specific functions can be called from within the trajectory generator, so as to allow on-line task modification and control (such as guarded motions or impedance control). Such functions may be of the following two types:

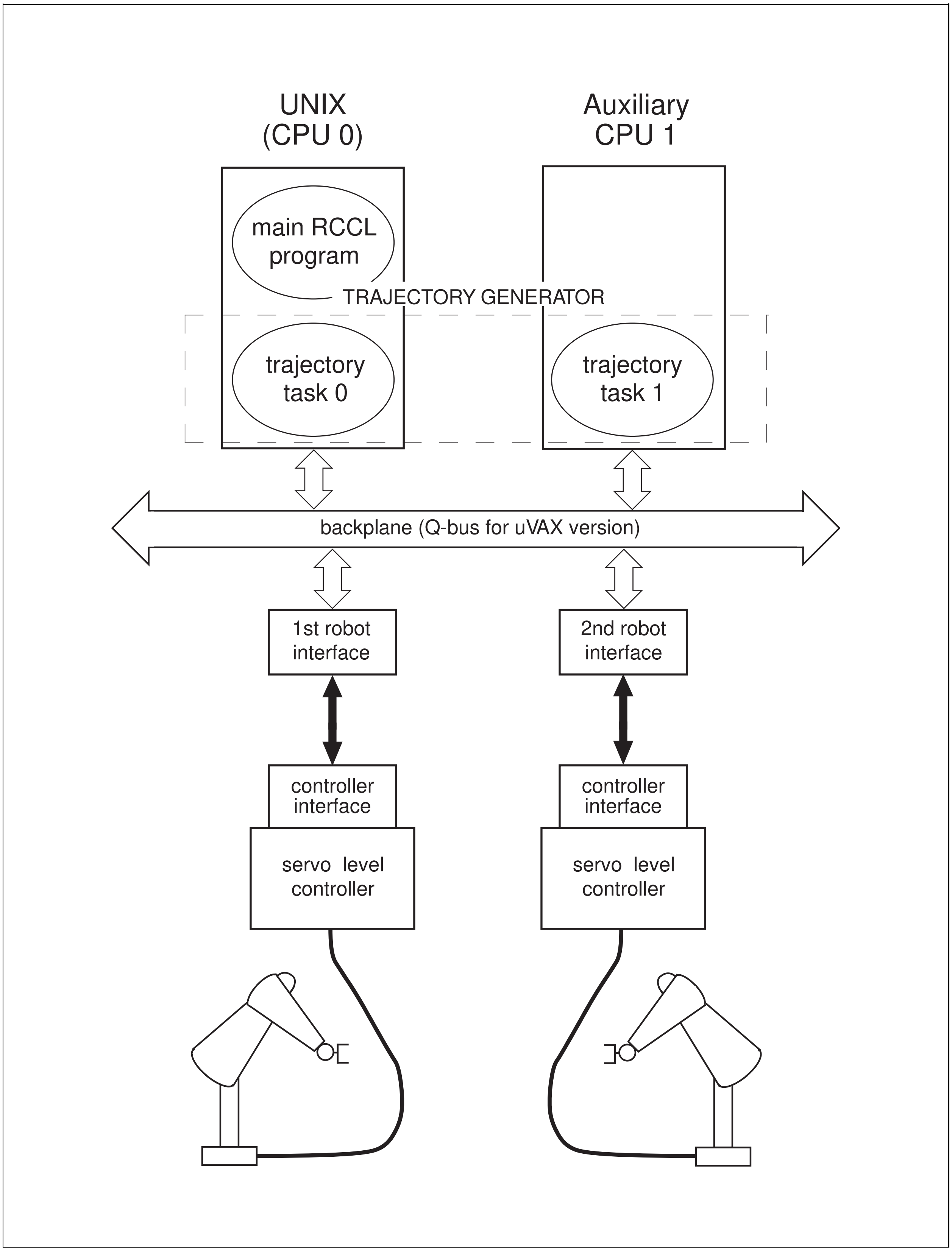

Figure 2: RCCL system, connected to two robots, with trajectory tasks running on two CPUs.

- *transform functions* – These are specifically intended to modify spatial coordinate frames. They provide an easy way to integrate sensor information into a program, and are used to implement things such as tracking (see Section 5.1).
- *Monitor functions* – these are general purpose functions that can be set up to do pretty much anything else (see Section 6.3.1).

Each such function is called by the trajectory generator, once per cycle, at certain designated times (Section 6.3.2.

Again, before writing functions to execute at the control level, the reader should check Section 5.3 to determine the restrictions that apply to such functions.

1.3.1 Support for multiple CPUs

The RCCL trajectory generator is designed to be operable on multiple CPUS. In this mode, each CPU will run a single trajectory generation task, will will in turn operate one or more robots. The computation is divided up on a per-robot basis, i.e., any given robot is still controlled by only one trajectory task, but the computation for several robots can be done concurrently by tasks on different CPUs. Figure 2 shows an example involving two robots and two CPUs. All the trajectory tasks run in sync, at the same rate, and so to the planning level they look very much like one task. The increase in available CPU power can be used in two ways: (1) to run the trajectory generator at a faster rate, and (2) to permit the execution of more complex control level applications.

Support for multiple CPUs was introduced and implemented on MicroVAX-II systems, and was motivated simply by speed requirements. Since then, host system CPU speeds have increased enormously, and the need for multiple-CPU implementations is not nearly as obvious (all the computations for a PUMA-like robot can now be done in well under one millisecond on state-of-the-art SGI or Sun systems). No multiple-CPU implementation is actually available in the current release (since the MicroVAX support has been phased out). However, the hooks for multiple-CPUs have been left in place, and are reflected in the arguments to functions such as `rcclCreate()` or `allocMem()`, which require a target CPU specification.

The multi-CPU features being introduced into Sun and SGI systems may, at some point, make it worthwhile to again consider a multi-CPU implementation.

1.4 Using the System

Using RCCL to control a robot consists of setting up a few environment variables (Section 9.4.1), starting up the robot controller (and calibrating it if necessary; Section 9.4.2), compiling an RCCL program, and running it. Depending on the specific installation, it may also be necessary to turn the robot arm power on after the program is started; the system will prompt the user to do so if this is the case.

The library and include directives necessary to compile an RCCL program are discussed in Section 9.4.3.

1.5 Name conventions used in this manual

The symbol `<RCCL>` will be used to denote the root directory of the RCCL/RCI distribution tree. Path names will often be described with reference to either `<RCCL>` or the sub-directory `<RCCL>/src`, so, for instance, `lib/generic` and `demo.rccl/simple.560.c` refer to `<RCCL>/lib/generic` and `<RCCL>/src/demo.rccl/simple.560.c`, respectively.

The symbol `<Target>` will be used to denote the name of a target execution system. This name must match one of the system-specific `.conf` files located in `<RCCL>/conf`. Typical values for `<Target>` include `Sun4sol` (for Sun/Solaris systems) and `Sgi` (for SGI/IRIX5 systems). For more information, see the *RCCL/RCI Startup and Installation Guide*.

1.6 A Simple Program Example

We will get things rolling immediately by presenting a simple RCCL program. The description of things might not be clear on first reading, but everything presented here is described in much greater detail later on. The program moves the robot to a suitable starting position, then moves in a straight line to a nearby target point, and finally returns to the starting position.

The program looks like this:

```
#include <rccl.h>
#include "manex.560.h"

main()
{
        TRSF_PTR p, t;                                          /*#1*/
        POS_PTR pos;                                            /*#2*/
        MANIP *mnp;                                             /*#3*/
        JNTS rcclpark;                                          /*#4*/
        char *robotName;                                        /*#5*/

        rcclSetOptions (RCCL_ERROR_EXIT);                       /*#6*/
        robotName = getDefaultRobot();                          /*#7*/
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }
                                                                /*#8*/
        t = allocTransXyz ("T", UNDEF, -300.0, 0.0, 75.0);
        p = allocTransRot ("P", UNDEF, P_X, P_Y, P_Z, xunit, 180.0);
        pos = makePosition ("pos", T6, EQ, p, t, NULL);         /*#9*/

        mnp = rcclCreate (robotName, 0);                        /*#10*/
        rcclStart();

        movej (mnp, &rcclpark);                                 /*#11*/

        setMod (mnp, 'c');                                      /*#12*/
```

```
        move (mnp, pos);                                    /*#13*/
        stop (mnp, 1000.0);

        movej (mnp, &rcclpark);                             /*#14*/
        stop (mnp, 1000.0);

        waitForCompleted (mnp);                             /*#15*/
        rcclRelease (YES);                                  /*#16*/
}
```

This example works for the PUMA 560 robot, and is contained in `demo.rccl/simple.560.c`. An equivalent program for the PUMA 260 is contained in `simple.260.c`. To run this program, set up your RCCL environment as described in Section 9.4.1, start up the robot and controller (Section 9.4.2), compile the program using the commands described in Section 9.4.3, and execute it.

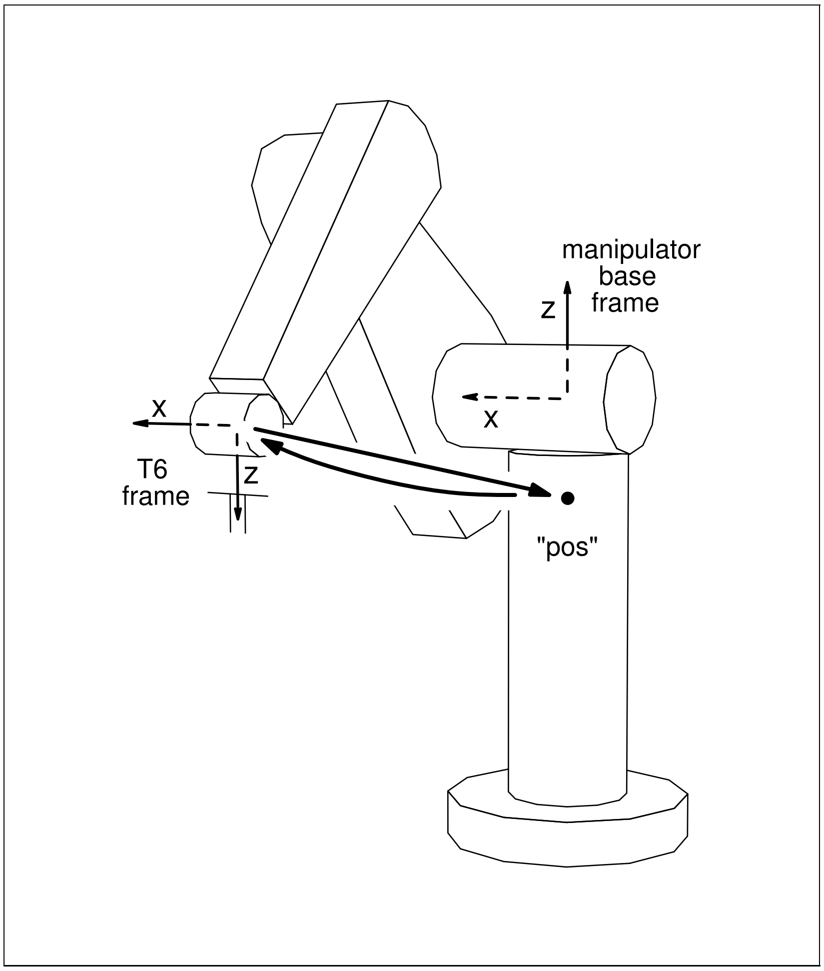

Figure 3: Moving to position "pos" using Cartesian interpolated motion, and moving back using joint interpolated motion.

Figure 3 shows the two motions which are performed after robot has moved to the starting position. Two coordinate frames are illustrated: the base frame of the robot's **T6** transform[2], situated in its shoulder, and the T6 coordinate frame itself, situated in the last link.

The include file <rccl.h> contains C structure type definitions and external entry points the same way the include file <stdio.h> does. It provides the necessary definitions for the RCCL functions, structures, and variables. The file "manex.560.h" contains declarations specific to the PUMA 560 demo programs used in this manual.

The program will make use of two 4×4 homogeneous transforms described by the TRSF data type. The variables p and t are declared as pointers to these transforms, using the pointer type TRSF_PTR (#1) (which is equivalent to TRSF*). The program will describe the target for one of its motions using a *position equation*, which is described by a data type POS. pos is declared as a pointer to the position equation using the type POS_PTR (#2) (which is equivalent to POS*). Within the program, the robot is described by a MANIP structure, pointed to by mnp (#3) (the equivalent type MANIP_PTR is also available). The first step is to move the robot to a known initial position; this position is described by a set of joint angles stored in the variable rcclpark, which is a JNTS data type (#4) (these data types are not generally allocated by system routines, so we have to provide our own; that is why rcclpark is the actual structure rather than a pointer). Finally, a string describing the name of the robot to be controlled is stored in the variable robotName (#5).

The first command is a call to rcclSetOptions() to set the option RCCL_ERROR_EXIT (#6). When this option is set, most RCCL primitives will simply print out diagnostic messages and abort the program upon encountering a run-time error. This saves the programmer from having to explicitly check each primitive's return value for an error condition (typically indicated by NULL or -1, depending on whether the primitive normally returns a pointer or not), and is useful for top level applications or code development.

The next thing the program does (at #7) is get the name of the robot it is going to control and the initial position it will move it to. getDefaultRobot() returns the name of the system default robot, which is stored in the file defaultRobot.cfg, located using the environment variable RCCL_PATH_<Target> (Section 9.4.1). With this name, the program then uses getRobotPosition() to look up the joint angles corresponding to the name "rcclpark". This routine checks the file <robotName>.pos (also located using $RCCL_PATH_<Target>) for the named set of angles and returns 1 if found and 0 otherwise (not finding the position is not considered to be an error *per se*, so the function's return value is checked explicitly). The joint angles are read into the v field of rcclpark, which is just an array of floats describing the joint values.

Next, the program allocates and instantiates the 4×4 transforms (#8). The transform pointed to by t is used to specify a target position relative to the end effector coordinates of the "rcclpark" position. It is created with a call to allocTransXyz(), which allocates a TRSF data type, gives it the name "T", sets it to have a translational component of -300.0, 0.0, and 75.0, and returns a pointer to it. The rotational component is set to the identity matrix. The routine's second argument allows the programmer to specify the memory pool from which the transform is allocated; this is normally left undefined (by specifying UNDEF), as is done here.

The transform p is set equal to the value of the manipulator's **T6** transform at the initial position. Although it is possible to determine this value directly by reading it back when the robot arrives at

[2] This is the 4×4 transformation from the robot's base frame to a coordinate frame located in its last link. It will be described in detail later.

the initial position (which will be done in later examples), we here set it explicitly. The transform is created using `allocTransRot()`, is given the name `"P"`, and is built of translational component of `P_X`, `P_Y`, and `P_Z` (defined in `manex.560.h`) along the corresponding axes and a rotational component equal to a rotation of 180.0 degrees about the x axis. (`xunit` is a built-in variable of type `VECT` whose value describes the unit vector $(1, 0, 0)$. Since transform `"P"` is referenced to the same base frame as **T6** (i.e., the robot shoulder), this rotation causes the z axis of the frame following `"P"` to point downward (see the figure).)

The two transforms are now used in a call to `makePosition()` (`#9`), which constructs a kinematic position equation (as described in [Paul 1981]) and returns a pointer to it. `makePosition()` accepts a variable number of arguments, the first being the name of the position. The remaining arguments, up to the special argument `EQ`, are transforms making up the left hand side of the position equation. The arguments after `EQ` are transforms making up the right hand side. The argument list is terminated by `NULL`. The manipulator **T6** transform is indicated by the special argument `T6`. The call to `makePosition()` in this example declares the following equation:

$$\mathbf{T6} = \mathbf{B}\,\mathbf{T}$$

If we interpret this as a target for a robot motion, we can say that the robot should be moved until its **T6** transform satisfies the equation. In this case, **T6** will be set equal to **B** (the value of **T6** at the park position) multiplied by the transform **T**.

The next group of functions initialize and turn on the trajectory generator (`#10`). `rcclCreate()` allocates the structures necessary to control a particular robot. It takes two arguments: a string describing the name of a robot to be controlled and a bit mask specifying which CPUs may be used to run that robot's trajectory generator. This last argument is relevant only for RCCL/RCI systems with multi-CPU capability and may left undefined (as is done here) by specifying 0. `rcclCreate()` returns a pointer to a `MANIP` data structure which is used by the rest of the program to reference and control the robot in question. `rcclStart()` starts the trajectory generator task running; motion requests will not have any effect unless this is done.

The program is now ready to move the robot. The first motion request is made with `movej()`, which requests that the manipulator move to the joint angles contained in `rcclpark` (`#11`).

RCCL allows motion targets to be specified using either joint coordinates (as is done here with `movej()`) or Cartesian coordinates (as will be done next, with `move()`). Furthermore, if the motion target is specified in Cartesian coordinates, the system offers a choice as to how the *path* to that target is computed. By default, RCCL computes the path to the target using *joint* interpolated motions. This can be changed to *Cartesian* straight line motion by calling `setMod (mnp, 'c')` (`#12`). In this mode, the manipulator TOOL frame (equivalent in this program to **T6**) is moved along a straight line in Cartesian coordinates to the target position. The difficulty with Cartesian mode is that sometimes a straight line path from A to B cannot be followed, typically because it would place the robot in an impossible position. What we usually do (and have done here) is to use joint mode for our initial motion; local Cartesian motions are then less likely to fail.

The next call (to `move()`) tells the robot to move to a target point defined by solving the position equation `pos` for **T6** (`#13`). The following call to `stop()` instructs the robot to stop there for 1 second (specified as 1000 milliseconds) before continuing.

Finally, we request a move back to the initial position with another call to `movej()` (`#14`) (moves requested with `movej()` are always computed in joint mode, regardless of the interpolation mode selected by `setMod()`).

Motion requests are not synchronized with the actual motion of the arm. Instead, they simply place a motion request packet on a queue, where they wait to be serviced by the trajectory generator as soon as possible. After the calls to `movej()` and `move()` in the program have returned, the robot has probably not even completed the first `movej()` request. To synchronize the main program with the motion of the arm, different methods are available. The simplest of these, used in the example here, is to wait for the manipulator to finish servicing all of its motion requests and come to rest. This is accomplished by the macro `waitForCompleted(mnp)` (`#15`).

The last call in the program, `rcclRelease()` (`#16`), turns the trajectory generator off. It takes a binary argument which, if true, specifies that the robot arm power should be turned off as well.

2. Basic Data Types

We shall now describe in detail some of the RCCL data structures, in particular vectors, transformations, differential motions, and forces. We shall also indicate how they can be used in manipulator programs.

2.1 The Data Type Functions

Each RCCL data type is generally associated with a cluster of functions for modifying instances of that type, or converting it to another type. (It is in these cases that RCCL would most greatly benefit from being implemented in a more "object oriented" language, such as C++; if anyone decides to do this, the authors would be interested in knowing.)

The data type functions follow a parameter passing convention where (1) a pointer to the type being modified is passed in as the leftmost argument, and (2) the function returns the value of this pointer (in the same style as the the string manipulation functions described in `string(3)`.) This permits calls to be nested in the following way:

```
trans2 = rotToTrsf (xyzToTrsf (trans1, 1.0,2.0,3.0), zunit, 90.0);
```

which, in one line, sets the translational components of a transform to 1.0, 2.0, and 3.0, and then sets the rotational component to describe a turn of $90°$ about z.

Structure arguments are passed by reference to save execution time.

Most of the special data types are defined in the file `<robotTypes.h>`, which is included automatically by `<rccl.h>`.

2.2 Some Basic Definitions

The include file `<rccl.h>` references `<cdefs.h>`, which defines a few very basic things used by most of the system:

```
#define NULL    0
#define NULLF   (int(*)())0

#define YES     1
#define NO      0
#define UNDEF   (-1)

#define MIN(x,y)        ((x) < (y) ? (x) : (y))
#define MAX(x,y)        ((x) > (y) ? (x) : (y))
#define ABS(x)          ((x) < 0 ? -(x) : (x))
```

```
#define NINT(x)         ((int)(((x)<0) ? ((x)-0.5)) : ((x)+0.5))

#define ARRAY_SIZE(a)   (sizeof(a)/sizeof(a[0]))
```

`NULL` and `NULLF` are used for identifying null pointers, `YES` and `NO` are "user friendly" booleans, `UNDEF` is used to indicate that some (normally non-negative) quantity is undefined, `MAX()`, `MIN()`, `ABS()`, and `NINT()` are obvious, and `ARRAY_SIZE()` is used for automatically determining the number of elements in an array.

There are some other things in `<cdefs.h>`, but they won't be referred to in this manual.

2.3 Numbers

The file `<rccl.h>` also references `<robotMath.h>`, which defines several numeric constants:

```
PI
PIB2
PIT2
DEGTORAD
RADTODEG
```

These are, respectively, π, $\pi/2$, 2π, $\pi/180$, and $180/\pi$.

2.4 Units

Distances in RCCL are expressed in millimeters. Rotational units are expressed in either degrees or radians; there is some occasional inconsistency, which is historical in origin but can be a bit bothersome. Rotations are always expressed *internally* in radians, and functions which accept (or output) degrees convert to (or from) the radian representation at the highest level.

Force is expressed in Newtons and torque is expressed in Newton-millimeters.

Time is expressed in milliseconds.

2.5 Vectors

Spatial 3-vectors are described by the type `VECT`:

```
typedef struct {
        float x, y, z;
} VECT, *VECT_PTR;
```

A pointer to a vector variable can either be defined as `VECT *pk` or `VECT_PTR pk`, depending on the programmer's taste and coding style. This paradigm will be followed with the other data types.

Vectors are assigned and manipulated by the following functions:

```
VECT_PTR crossVect (vr, v1, v2)
  VECT_PTR vr, v1, v2;

VECT_PTR unitVect (vr, v1)
  VECT_PTR vr, v1;

float dotVect (v1, v2)
  VECT_PTR v1, v2;

VECT_PTR addVect (vr, v1 ,v2)
  VECT_PTR vr, v1, v2;

VECT_PTR subtractVect (vr, v1 ,v2)
  VECT_PTR vr, v1, v2;

VECT_PTR scaleVect (vr, a, v1)
  VECT_PTR vr, v1;
  float a;

VECT_PTR combineVect (vr, a, v1, b, v2)
  VECT_PTR vr, v1, v2;
  double a, b;

float norm2Vect(v1)
  VECT_PTR v1;

float norm1Vect(v1)
  VECT_PTR v1

float normInfVect(v1)
  VECT_PTR v1;

float normVect(v1)
  VECT_PTR v1;

VECT_PTR zeroVect(vr)
  VECT_PTR vr;
```

Let $\mathbf{v}_r$, $\mathbf{v}_1$, $\mathbf{v}_2$, a, and b denote the quantities respresented by the arguments `vr`, `v1`, `v2`, `a`, and `b`. `crossVect` computes the cross product $\mathbf{v}_r = \mathbf{v}_1 \times \mathbf{v}_2$. `unitVect()` normalizes $\mathbf{v}_1$, so that $\mathbf{v}_r = \mathbf{v}_1/|\mathbf{v}_1|$. `dotVect()` returns the dot product $\mathbf{v}_1 \cdot \mathbf{v}_2$. `addVect()`, `subtractVect()`, `scaleVect()`, and `combineVect()` compute, respectively, the following arithmetic operations:

$$\mathbf{v}_r = \mathbf{v}_1 + \mathbf{v}_2$$

$$\mathbf{v}_r = \mathbf{v}_1 - \mathbf{v}_2$$

$$\mathbf{v}_r = a\mathbf{v}_1$$

$$\mathbf{v}_r = a\mathbf{v}_1 + b\mathbf{v}_2$$

`norm2Vect()` and `normVect()` both return the 2-norm $\|\mathbf{v}_1\|$, while `norm1Vect()` returns the 1-norm $\|\mathbf{v}_1\|_1$, and `normInfVect()` returns the ∞-norm $\|\mathbf{v}_1\|_\infty$. Finally, `zeroVect()` zeros a vector.

Note that these routines return a pointer to their result argument, where appropriate.

For convenience, RCCL pre-defines the three `VECT_PTR` types `xunit`, `yunit`, and `zunit`, and sets them to point to unit vectors along the x, y, and z axes.

2.6 Transformations

A 4×4 homogeneous transformation is represented by the structure `TRSF`:

```
typedef struct {
        VECT n, o, a, p;
} TRSF, *TRSF_PTR;
```

The entries in the structure are simply the columns of the matrix which comprise the *normal*, *orientation*, *approach*, and *position* vectors (**n**, **o**, **a**, and **p**). The first three of these constitute an orthogonal rotation matrix **R** $\in SO(3)$. The last row of the transform is assumed to be $[0, 0, 0, 1]$, so these types contain enough information to do only three dimensional coordinate transformations, not scaling or perspective operations.

Transforms are assigned and manipulated by numerous functions. The most basic include:

```
TRSF_PTR copyTrsfXyz (tr, t1)
  TRSF_PTR tr, t1;

TRSF_PTR copyTrsfNoa (tr, t1)
  TRSF_PTR tr, t1;

TRSF_PTR multTrsf (tr, t1, t2)
  TRSF_PTR tr, t1, t2;

TRSF_PTR multRiTrsf (tr, t1, t2)
  TRSF_PTR tr, t1, t2;

TRSF_PTR multLiTrsf (tr, t1, t2)
  TRSF_PTR tr, t1, t2;

TRSF_PTR invertTrsf (tr, t1)
  TRSF_PTR tr, t1;

TRSF_PTR identTrsf (tr)
  TRSF_PTR tr;
```

Let $\mathbf{T}_r$, $\mathbf{T}_1$, and $\mathbf{T}_2$ denote the quantities respresented by the arguments `tr`, `t1`, and `t2`. The functions `copyTrsfXyz()` and `copyTrsfNoa()` perform a selective copy of the translational (resp. rotational) part of $\mathbf{T}_1$ into $\mathbf{T}_r$, leaving untouched the rotational (resp. translational) part. The functions `multTrsf()`, `multRiTrsf()`, and `multLiTrsf()` compute, respectively,

$$\mathbf{T}_r = \mathbf{T}_1 \mathbf{T}_2$$

$$\mathbf{T}_r = \mathbf{T}_1 \mathbf{T}_2^{-1}$$

$$\mathbf{T}_r = \mathbf{T}_1^{-1} \mathbf{T}_2.$$

Finally, `invertTrsf()` computes $\mathbf{T}_r = \mathbf{T}_1^{-1}$, and `identTrsf()` converts `tr` to an identity transform.

Again, all of these routines return a pointer to their result argument.

To transform a vector between coordinate systems, the following functions are available:

```
VECT_PTR transVect (vr, v1, t1)
  VECT_PTR vr, v1;
  TRSF_PTR t1;

VECT_PTR invTransVect (vr, v1, t1)
  VECT_PTR vr, v1;
  TRSF_PTR t1;

VECT_PTR rotateVect (vr, v1, t1)
  VECT_PTR vr, v1;
  TRSF_PTR t1;

VECT_PTR invRotateVect (vr, v1, t1)
  VECT_PTR vr, v1;
  TRSF_PTR t1;
```

Let $\mathbf{v}_r$, $\mathbf{v}_1$, and $\mathbf{T}_1$ denote the quantities represented by the arguments `vr`, `v1`, and `t1`.

`transVect()` performs a spatial transformation of $\mathbf{v}_1$ into a new coordinate frame which is reached by applying the transformation $\mathbf{T}_1$ to the original frame. The resulting computation is

$$\mathbf{v}_r = \mathbf{T}_1^{-1}\mathbf{v}_1.$$

`invTransVect()` performs the inverse computation:

$$\mathbf{v}_r = \mathbf{T}_1 \mathbf{v}_1.$$

`rotateVect()` performs a rotational transformation of $\mathbf{v}_1$ into a new coordinate frame which is reached by applying the rotational component of $\mathbf{T}_1$ to the original frame. If the rotational component of $\mathbf{T}_1$ is given by the 3×3 matrix $\mathbf{R} \in SO(3)$, then the resulting computation is

$$\mathbf{v}_r = \mathbf{R}^{-1}\mathbf{v}_1.$$

`invRotateVect()` performs the inverse computation:

$$\mathbf{v}_r = \mathbf{R}\,\mathbf{v}_1.$$

To set various components of a transformation, the following functions are available:

```
TRSF_PTR xyzToTrsf (tr, px, py, pz)
  TRSF_PTR tr;
  float px, py, pz;

TRSF_PTR vaoToTrsf (tr, ax, ay, az, ox, oy, oz)
  TRSF_PTR tr;
  float ax, ay, az, ox, oy, oz;

TRSF_PTR rotToTrsf (tr, k, h)
  TRSF_PTR tr;
  VECT_PTR k;
  float h;

TRSF_PTR eulToTrsf (tr, phi, the, psi)
  TRSF_PTR tr;
  float phi, the, psi;

TRSF_PTR rpyToTrsf (tr, roll, pitch, yaw)
  TRSF_PTR tr;
  float roll, pitch, yaw;
```

All these functions use a transformation pointer as left argument; as usual, this pointer is also returned by the function.

The function `xyzToTrsf()` sets the terms of the $\mathbf{p}$ vector of `tr` and leaves the rotational part untouched.

The function `vaoToTrsf()` sets the $\mathbf{n}$, $\mathbf{o}$, and $\mathbf{a}$ vectors of `tr`. Since the vectors $\mathbf{n}$, $\mathbf{o}$ and $\mathbf{a}$ are orthogonal, `vaoToTrsf()` takes only the components of $\mathbf{o}$ and $\mathbf{a}$ and builds the vector $\mathbf{n}$ itself. The vectors whose components are input as arguments do not need to be orthogonal, since the rotational part of the transform is built as follows: take the user's supplied $\mathbf{a}$ vector, normalize it and use it as the final $\mathbf{a}$ vector; take the user's supplied $\mathbf{o}$ vector (which need not be orthogonal to $\mathbf{a}$) and build a (possibly non-unit) vector $\mathbf{n}$ that is orthogonal to both $\mathbf{o}$ and $\mathbf{a}$; reconstruct $\mathbf{o}$ to be orthogonal with $\mathbf{n}$ and $\mathbf{a}$, normalize it, and finally, compute $\mathbf{n}$ from the cross product of $\mathbf{o}$ and $\mathbf{a}$[1].

The function `rotToTrsf()` sets the rotational part of `tr` to a rotation of `h` degrees about a (possibly non-unit) vector `k`.

The function `eulToTrsf()` sets the rotational part of `tr` to a rotation expressed with the three Euler angles (in degrees).

[1] Check the reference manual for the corresponding equations.

Finally, the function `rpyToTrsf()` sets the rotational part of `tr` to a rotation expressed with the three roll-pitch-yaw angles (in degrees).

The next set of functions are exactly analogous to the previous ones, except that the transform argument is *post-multiplied* by another transform *defined by* the other parameters of the function. As usual, a pointer to the result transform is returned as the value of the function.

```
TRSF_PTR multTrsfXyz (tr, px, py, pz)
  TRSF_PTR tr;
  float px, py, pz;

TRSF_PTR multTrsfVao (tr, ax, ay, az, ox, oy, oz)
  TRSF_PTR tr;
  float ax, ay, az, ox, oy, oz;

TRSF_PTR multTrsfRot (tr, k, h)
  TRSF_PTR tr;
  VECT_PTR k;
  float h;

TRSF_PTR multTrsfEul (tr, phi, the, psi)
  TRSF_PTR tr;
  float phi, the, psi;

TRSF_PTR multTrsfRpy (tr, roll, pitch, yaw)
  TRSF_PTR tr;
  float roll, pitch, yaw;
```

It may sometimes be desirable to convert the rotational component of a `TRSF` data type back into Euler angles, roll-pitch-yaw angles, or rotation angle/axis parameters. The functions to do this are:

```
trsfToEul (phi, the, psi, t1)
  float *phi, *the, *psi;
  TRSF_PTR t1;

trsfToRpy (roll, pitch, yaw, t1)
  float *roll, *pitch, *yaw;
  TRSF_PTR t1;

trsfToRot (k, h, t1)
  VECT_PTR k;
  float *k;
  TRSF_PTR t1;
```

The Euler or roll-pitch-yaw angle representations for a rotation are not necessarily unique. The rotation angle/axis representation is unique since the returned value of `h` is defined to lie in the range $0 \leq h < 180°$. Note also that scalar values are returned through *pointers*, in a manner analogous to the UNIX routine `scanf()`.

2.7 Allocating Transformations

Structures of the `TRSF` data type described in the previous section may be defined or allocated in any of the ways normally available for creating C objects (static or automatic declaration, use of `malloc()`, etc.). However, if one wishes to create a transform which is shared between the planning and control levels, then it must not be a static or automatic variable, nor can it be `malloc()`ed; instead, it must be allocated via a set of special RCCL primitives. Transforms allocated in this way can be bound to real-time functions, and can also be used to form position equations as described later.

The basic primitives for creating and deleting transforms are

```
TRSF_PTR allocTrans (name, pool)
  char *name;
  int pool;

freeTrans (t)
  TRSF_PTR t;
```

`allocTrans()` allocates a transformation, gives it the indicated name, sets its value to the identity, and returns a pointer to it. The variable `pool` allows the programmer to specify which memory pool the transform should be allocated from. Under the current implementation, the pool number corresponds to a CPU number; i.e., specifying pool number 2 would mean that the transform is allocated from memory on CPU 2. Obviously, this is relevant only on multi-CPU versions of RCCL/RCI. Typically, the pool argument is left undefined (`UNDEF`) and the system chooses the allocation pool. It also is not necessary to give the transform a name; if no name is desired, then that argument may be specified as `NULL`. Names are useful sometimes when one wishes to obtain a pointer to the transform from the control level.

The function `freeTrans()` returns a transform to the pool from which it was allocated.

If a programmer wishes to make a compile-time declaration of a transform, as in

```
function()
{
        TRSF base;

        ...
        ...
}
```

or the declaration

```
static TRSF base;
```

then the following should be noted:

- First, transforms created this way *cannot* be accessed by the control level. They **must not be used** in a position data structure (created by the function `makePosition()`, described later).

- Structure data types are passed by reference, so literally declared transformations need to be prefixed with the `&` (address) operator when passed to a function.

- It is up to the programmer to initialize transforms declared in this way (for instance, by using `identTrsf()`).

In addition to `allocTrans()`, there is a group of other functions which allocate a new transform and at the same time initialize it to desired values:

```
TRSF_PTR allocTransXyz (name, pool, px, py, pz)
  char *name;
  int pool;
  float px, py, pz;

TRSF_PTR allocTransRot (name, pool, px, py, pz, k, h)
  char *name;
  int pool;
  float px, py, pz, h;
  VECT_PTR k;

TRSF_PTR allocTransPao (name, pool, px,py,pz,ax,ay,az,ox,oy,oz)
  char *name;
  int pool;
  float px, py, pz, ax, ay, az, ox, oy, oz;

TRSF_PTR allocTransEul (name, pool, px, py, pz, phi, the, psi)
  char *name;
  int pool;
  float px, py, pz, phi, the, psi;

TRSF_PTR allocTransRpy (name, pool, px, py, pz, roll, pitch, yaw)
  char *name;
  int pool;
  float px, py, pz, roll, pitch, yaw;
```

All of these functions return a pointer to the created transform. The first two arguments are identical to the first two arguments of `allocTrans()`. The next three arguments specify the translational component, the $\mathbf{p}$ vector. The remaining arguments, if any, define the rotational component, and are the same as those for the corresponding `xxxToTrsf()` routines (except for `allocTransPao()`, which maps onto `vaoToTrsf()` instead of `paoToTrsf()`).

For example, the code fragment

```
TRSF_PTR t1, t2, t3;    /* declare transform pointers */

...

t1 = xyzToTrsf (eulToTrsf (allocTrans ("T1", UNDEF),
                           10.0, 20.0, 30.0),
                1.0, 2.0, 3.0);

t2 = allocTransEul ("T2", UNDEF, 1.0, 2.0, 3.0, 10.0, 20.0, 30.0);

t3 = allocTransXyz ("T3", UNDEF, 1.0, 2.0, 3.0);
eulToTrsf (t3, 10.0, 20.0, 30.0);
```

produces three identical transforms.

Finally, there are functions for printing the value of a transform:

```
printTrsf (fmt, tr);
  char *fmt;
  TRSF_PTR tr;

fprintTrsf (fp, fmt, tr);
  FILE *fp;
  char *fmt;
  TRSF_PTR tr;

rciPrintTrsf (fmt, tr);
  char *fmt;
  TRSF_PTR tr;
```

All of these accept a pointer to a transform and a format string describing how the transform should be printed. Typical format strings are `"%m\n"`, which prints the transform in matrix form followed by a newline, `"%r\n%p\n"`, which prints the transform in terms of the roll-pitch-yaw angles and the translational component followed by an newline, and `"%q\n%p\n"`, which prints the transform in terms of a quaternion and the translational component (again, followed by a newline). Preceding one of the format characters with a `'L'` causes one of the labels

```
QTRN
 RPY
 XYZ
 EUL
```

to be printed in front of the quaternion, roll-pitch-yaw, translational, or euler angle representation. Non-format characters which appear in the format string are printed literally (hence the new lines in the above examples).

For example, the code fragment

```
TRSF_PTR t1, t2, t3;

t1 = allocTransEul ("T1", UNDEF, 10., 20., 30., 11., 12., 13.);
printTrsf ("T1 %m\n", t1);

t2 = allocTrans ("T2", UNDEF);
rotToTrsf(t2, yunit, 90.);
multTrsfXyz(t2, 10., 20., 30.);
putchar ('\n');
printTrsf ("T2 %Lr\n   %Lp\n", t2);

t3 = allocTrans ("T3", UNDEF);
putchar ('\n');
printTrsf ("T3 %m\n", multTrsf (t3, t1, t2));

putchar ('\n');
printTrsf ("T3 %q\n   %p\n", t3);
```

will produce the output

```
T1  0.89264 -0.40191  0.20409    10.00
    0.40267  0.91448  0.03967    20.00
   -0.20258  0.04677  0.97815    30.00

T2      RPY:    0.000   90.000   -0.000
        XYZ:    30.00    20.00   -10.00

T3 -0.20409 -0.40191  0.89264    26.70
   -0.03967  0.91448  0.40267    49.97
   -0.97815  0.04677 -0.20258    15.08

T3  0.61396 -0.14492  0.76177  0.14750
      26.70    49.97    15.08
```

Much more detailed information on the format string syntax can be found in the manual page for `printTrsf()`.

The function `fprintTrsf()` is identical to `printTrsf()` except that it allows the output stream to be specified.

`rciPrintTrsf()` was specifically designed as a version of `printTrsf()` which could be called from the control level. Since most implementations of RCCL/RCI now permit regular printing from the control level, this function is no longer as relevant.

2.8 Differential Motions and Forces

NOTE: The background to this material is described in [Paul 1981]; this section can probably be *omitted on first reading.*

A velocity or differential motion is expressed in terms of a *twist* consisting of a differential translation vector and differential rotation vector. A generalized force is expressed in terms of a *wrench* consisting of a translational force and a torque (or moment). The corresponding structures are:

```
typedef struct {
        VECT t, r;
} DIFF, *DIFF_PTR;

typedef struct {
        VECT f, m;
} FORCE, *FORCE_PTR;
```

The main functions associated with these data types are:

```
TRSF_PTR diffToTrsf (tr, d1)
  TRSF_PTR tr;
  DIFF_PTR d1;

DIFF_PTR trsfToDiff (dr, t1)
  DIFF_PTR dr;
  TRSF_PTR t1;

DIFF_PTR transDiff (dr, d1, t1)
  DIFF_PTR dr, d1;
  TRSF_PTR t1;

FORCE_PTR transForce (fr, f1, t1)
  FORCE_PTR fr, f1;
  TRSF_PTR t1;

DIFF_PTR rotateDiff (dr, d1, t1)
  DIFF_PTR dr, d1;
  TRSF_PTR t1;

FORCE_PTR rotateForce (fr, f1, t1)
  FORCE_PTR fr, f1;
  TRSF_PTR t1;
```

Let $\mathbf{d}_1$, $\mathbf{d}_r$, $\mathbf{f}_1$, $\mathbf{f}_r$, and $\mathbf{T}_1$ denote the quantities represented by the arguments `d1`, `dr`, `f1`, `fr`, and `T1`.

The function `diffToTrsf()` converts a `DIFF` type into the corresponding transform, under the assumption that the rotational displacement described by `DIFF` is fairly small. Under this assumption, we have

$$\mathbf{T}_r \approx \begin{pmatrix} 1 & -r_z & r_y & t_x \\ r_z & 1 & -r_x & t_y \\ -r_y & r_x & 1 & t_z \end{pmatrix}$$

where $(t_x, t_y, t_z)^T$ and $(r_x, r_y, r_z)^T$ are the translational and rotational vectors comprising $\mathbf{d}_1$.

The function `trsfToDiff()` converts a transform into a `DIFF` type (again, assuming that the rotational displacement is small).

The functions `transDiff()`, `rotateDiff()`, `transForce()`, and `rotateForce()` perform spatial and rotational coordinate transformations on `DIFF` and `FORCE` types.

`transDiff()` takes a twist $\mathbf{d}_1$ expressed with respect to some frame O and finds its representation with respect to a different frame T. The transformation from O to T is assumed to be given by $\mathbf{T}_1$. If $\mathbf{R}$ and $\mathbf{p}$ are the rotational and translational components of $\mathbf{T}_1$, the resulting computation is

$$\mathbf{d}_r = \begin{pmatrix} \mathbf{R}^T & (\mathbf{p} \times \mathbf{R})^T \\ 0 & \mathbf{R}^T \end{pmatrix} \mathbf{d}_1$$

where $\mathbf{p} \times \mathbf{R}$ denotes the three column vectors formed by taking the cross product of $\mathbf{p}$ with each column of $\mathbf{R}$. The 6×6 matrix associated with the computation is sometimes called the *adjoint* matrix of $\mathbf{T}_1$.

The function `transForce()` performs the analogous transformation on `FORCE` types, taking a force $\mathbf{f}_1$ expressed with respect to some frame O and finding its representation with respect to frame T. The corresponding calculation is

$$\mathbf{f}_r = \begin{pmatrix} \mathbf{R}^T & 0 \\ (\mathbf{p} \times \mathbf{R})^T & \mathbf{R}^T \end{pmatrix} \mathbf{f}_1.$$

The function `rotateDiff()` performs a rotational transformation of $\mathbf{d}_1$ into a new coordinate frame which is reached by applying the rotational component of $\mathbf{T}_1$ to the original frame. If the rotational component of $\mathbf{T}_1$ is given by the 3×3 matrix $\mathbf{R} \in SO(3)$, then the resulting computation is

$$\mathbf{d}_r = \begin{pmatrix} \mathbf{R}^T & 0 \\ 0 & \mathbf{R}^T \end{pmatrix} \mathbf{d}_1.$$

Note that this is identical to `transDiff()`, except with $\mathbf{p}$ assumed to be 0.

The function `rotateForce()` performs a rotational transformation of $\mathbf{f}_1$ into a new coordinate frame which is reached by applying the rotational component of $\mathbf{T}_1$ to the original frame. The calculation, given by

$$\mathbf{f}_r = \begin{pmatrix} \mathbf{R}^T & 0 \\ 0 & \mathbf{R}^T \end{pmatrix} \mathbf{f}_1,$$

is identical to that performed by `rotateDiff()`.

To print out the values of `DIFF` or `FORCE` structures, the RCI utility routine `printVf()` can be used. This is a general purpose function used for printing a vector (or array) of floats. It is declared as follows:

```
printVf (fmt, v, n)
  char *fmt;
  float *v;
  int n;
```

where `fmt` is a format string, `v` is a pointer to the vector, and `n` is the number of elements in the vector. The format string accepts the usual constructs used by `printf()` for outputting floating point numbers (such as `"%f"` and `"%g"`), except that the number of output fields is set equal to the number of elements in the vector. The `DIFF` and `FORCE` structures can usually be printed by this routine, if the number of elements is set to 6[2].

For example, the following sequence of program statements

```
DIFF  Dp1, Dp2;
FORCE Fp1, Fp2;
TRSF *t;

t = allocTransPao ("T", UNDEF, 10.,5.,0.,1.,0.,0.,0.,0.,1.);

printTrsf ("%m\n", t);

Dp2.t.x =   1.0;
Dp2.t.y =   0.0;
Dp2.t.z =   0.5;
Dp2.r.x =   0.0;
Dp2.r.y =   0.1;
Dp2.r.z =   0.0;
printVf ("DP2: %6.2f\n", (float*)&Dp2, 6);
printVf ("DP1: %6.2f\n", (float*)transDiff (&Dp1, &Dp2, t), 6);

Fp2.f.x =  10.0;
Fp2.f.y =   0.0;
Fp2.f.z =   0.0;
Fp2.m.x =   0.0;
Fp2.m.y = 100.0;
Fp2.m.z =   0.0;
printVf ("FP2: %6.2f\n", (float*)&Fp2, 6);
printVf ("FP1: %6.2f\n", (float*)transForce(&Fp1, &Fp2, t), 6);
```

will yield the following output:

```
 0.00000  0.00000  1.00000    10.00
 1.00000  0.00000  0.00000     5.00
 0.00000  1.00000  0.00000     0.00
DP2:   1.00   0.00   0.50   0.00   0.10   0.00
DP1:   0.00  -0.50   1.00   0.10   0.00   0.00
FP2:  10.00   0.00   0.00   0.00 100.00   0.00
FP1:   0.00   0.00  10.00 100.00  50.00   0.00
```

[2]It is possible (though unlikely) that the internal representation of these data types on some machines will not be equivalent to an array of floats, in which case this will not work. Casting the address of the structure to a float pointer can also fail, but again this is improbable on most systems likely to run RCCL.

Much more detailed information on `printVf()` can be found in the manual pages.

For printing values to a specific stream the function `fprintVf()` can be used.

Another function, `rciPrintVf()`, is provided to print vector values from the control level (although on the most recent RCCL implementations, there are no control level printing restrictions, and so `printVf()` can be used to do this as well).

2.9 Displacements

Any displacement in Cartesian coordinates can be represented as a translation followed by a single rotation about some axis. RCCL provides a data type for representing displacements this way:

```
typedef struct {
        VECT p;
        float a;
        VECT u;
} DSPL, *DSPL_PTR;
```

The `p` field describes the translation, the `a` field describes the rotation angle (in radians), and the `u` field is a unit vector parallel to the axis of rotation. The rotational representation is potentially ambiguous: if the `a` and `u` fields describe a and $\mathbf{u}$, then $-a$ and $-\mathbf{u}$ represent the same thing. To resolve this, the range of a is restricted to $0 \leq a \leq \pi$. (The rotation representation is the same as that used by `rotToTrsf()`.)

The `DSPL` data type contains all the information described by a `TRSF` data type, and vice-versa. It is sometimes the preferred representation because the associated parameters are easy to scale: all one needs to do is multiply the translation vector `p` and the rotation angle `a` by some factor.

The following functions are available for manipulating `DSPL` data types:

```
DSPL_PTR trsfToDspl (dr, t1)
  DSPL_PTR dr;
  TRSF_PTR t1;

TRSF_PTR dsplToTrsf (tr, d1)
  TRSF_PTR tr;
  DSPL_PTR d1;

DSPL_PTR scaleDspl (dr, s, d1)
  DSPL_PTR dr;
  float s;
  DSPL_PTR d1;

TRSF_PTR extrapTrsfByDspl (tr, s, d1)
  TRSF_PTR tr;
  float s;
  DSPL_PTR d1;
```

```
DSPL_PTR unitDspl (dr, d1)
  DSPL_PTR dr, d1;
```

The functions `trsfToDspl()` and `dsplToTrsf()` convert between `TRSF` and `DSPL` representations. The function `scaleDspl()` scales the values of `d1` by `s` and puts the result in `dr`. The function `extrapTrsfByDspl()` is a combination of `scaleDspl()` and `dsplToTrsf()`: it scales the values of `d1` by `s` and then converts the result into the transform `tr`. The function `unitDspl()` normalizes the values of the `a` and `u` fields of `d1` and places the result in `dr`.

Because its parameters scale easily, `DSPL` data types are useful for representing velocities. A general Cartesian velocity described by the vectors $\mathbf{v}$ and ω can be represented with a `DSPL` data type by setting the `p` field to $\mathbf{v}$, the `a` field to $|\omega|$, and the `u` field to $\omega/|\omega|$. The function `scaleDspl()` can then be used to determine the net displacement achieved by this velocity over some time interval t. The equivalent transform representation of the same displacement can be computed with `extrapTrsfByDspl()`:

```
DSPL vel;
TRSF delta;
float time;

... variables are set ...

extrapTrsfByDspl (&delta, time, &vel);
```

2.10 Quaternions

RCCL has another data type for describing quaternions which are useful for representing rotations.

```
typedef struct {
        float c;
        VECT v;
} QTRN, *QTRN_PTR;
```

A rotation can be represented as a single rotation θ about a unit vector $\mathbf{v}$. A quaternion is a four tuple; when normalized to have unit length, it represents a rotation such that the first value is equal to $\cos(\theta/2)$ and the last three values form a vector equal to $\sin(\theta/2)\mathbf{v}$. Since multiplication of quaternions is equivalent to successive rotation operations and quaternion multiplication is faster that homogeneous transform multiplication, the use of quaternions can sometimes be desirable.

Quaternions used by RCCL are assumed to be unit quaternions. They are assigned and manipulated by the following functions:

```
QTRN_PTR multQtrn (qr, q1, q2)
  QTRN_PTR qr, q1, q2;

QTRN_PTR multRiQtrn (qr, q1, q2)
```

```
  QTRN_PTR qr, q1, q2;

QTRN_PTR multLiQtrn (qr, q1, q2)
  QTRN_PTR qr, q1, q2;

QTRN_PTR invertQtrn (qr, q1)
  QTRN_PTR qr, q1;

float normQtrn (qr)
  QTRN_PTR qr;

QTRN_PTR unitQtrn (qr)
  QTRN_PTR qr;
```

The function `multQtrn()` multiplies `q1` by `q2` and leaves the result in `qr`. `multRiQtrn()` multiplies `q1` by the inverse of `q2` and leaves the result in `qr`. The function `multLiQtrn()` multiplies the inverse of `q1` by `q2` and leaves the result in `qr`. The function `invertQtrn()` takes the inverse of `q1` and leaves it in `qr`. The function `normQtrn()` returns the magnitude of the quaternion (which should normally be 1). The function `unitQtrn()` normalizes the quaternion `qr`.

The following additional functions are used to convert between quaternions and other data types:

```
TRSF_PTR qtrnToTrsf (tr, q1)
  TRSF_PTR tr;
  QTRN_PTR q1;

QTRN_PTR trsfToQtrn (qr, t1)
  QTRN_PTR qr;
  TRSF_PTR t1;

VECT_PTR qtrnToVect (vr, q1)
  VECT_PTR vr;
  QTRN_PTR q1;
```

The function `qtrnToTrsf()` sets the rotational component of the transform `tr` to the rotation defined by the quaternion `q1`. The function `trsfToQtrn()` sets the quaternion `qr` to the rotation defined by the rotational component of the transform `tr`. The function `qtrnToVect()` sets the vector `vr` to the rotational velocity indicated by the quaternion `q1` (where the magnitude of the rotation is assumed to be small).

The printing of quaternion values can be done using `printVf()`, with the number of elements in the "vector" being specified as 4.

2.11 Generic Vector Routines

There are several routines available for doing simple operations on arbitrarily sized vectors of floats (*i.e.*, arrays of floats). These routines are imported directly from RCI, and this description also appears in the RCI user's guide:

```
float dotVf (v1, v2, size)
   float *v1, *v2;
   int size;

float *addVf (vr, v1, v2, size)
   float *vr, *v1, *v2;
   int size;

float *subtractVf (vr, v1, v2, size)
   float *vr, *v1, *v2;
   int size;

float *scaleVf (vr, a, v1, size)
   float *vr, *v1;
   float a;
   int size;

float *combineVf (vr, a, v1, b, v2, size)
   float *vr, *v1, *v2;
   float a, b;
   int size;

float norm1Vf (v1, size)
   float *v1;
   int size;

float norm2Vf (v1, size)
   float *v1;
   int size;

float normInfVf (v1, size)
   float *v1;
   int size;

float* zeroVf (v1, size)
   float *v1;
   int size;

float* copyVf (vr, v1, size)
   float *vr, *v1;
   int size;
```

Let $\mathbf{v}_r$, $\mathbf{v}_1$, $\mathbf{v}_2$, a, and b denote the quantities respresented by the arguments `vr`, `v1`, `v2`, `a`, and `b`. `dotVect()` returns the dot product $\mathbf{v}_1 \cdot \mathbf{v}_2$. `addVf()`, `subtractVf()`, `scaleVf()`, and `com-`

bineVf() compute, respectively, the arithmetic operations

$$\mathbf{v}_r = \mathbf{v}_1 + \mathbf{v}_2,$$

$$\mathbf{v}_r = \mathbf{v}_1 - \mathbf{v}_2,$$

$$\mathbf{v}_r = a\mathbf{v}_1,$$

$$\mathbf{v}_r = a\mathbf{v}_1 + b\mathbf{v}_2.$$

norm2Vf() and normVf() both return the 2-norm $\|\mathbf{v}_1\|$, while norm1Vf() returns the 1-norm $\|\mathbf{v}_1\|_1$, and normInfVf() returns the ∞-norm $\|\mathbf{v}_1\|_\infty$. Finally, zeroVf() zeros a vector, and copyVf() copies $\mathbf{v}_1$ into $\mathbf{v}_r$.

Each routine returns a pointer to its result, where appropriate. The size argument denotes the number of elements in the vector. These routines are completely analogous to the addVect() routines for the 3-vectors type VECT.

There is an identical set of routines which work with doubles instead of floats; these have the names dotVd(), addVd(), etc. Another set of routines is available for integers, and takes the names dotVi(), addVi(), etc. For details on these routines, see the *RCI Reference Manual*.

Converting between vectors of floats, doubles, or integers can be accomplished using routines which have the general form

```
copyV<I><O> (vr, v1, size)
```

where <I> and <O> are one of the characters f, d, or i, and denote the input and output types, respectively. For instance, to convert a vector a floats into a vector doubles, the routine

```
double *copyVfd (vr, v1, size);
   double *vr;
   double *v1;
   int size;
```

can be used. For more details on these routines, see the entry copyVf(3) in the *RCI Reference Manual*.

Printing a vector can be done using printVf(), fprintVf(), or rciPrintVf(), described above. The functions printVd(), fprintVd(), printVi(), and fprintVi() are available for vectors of doubles or integers.

Reading a vector in can be accomplished using either of

```
scanVf (vr, size);
fscanVf (fp, vr, size);
```

which input a vector of size elements either from stdin or the stream fp, respectively. The functions scanVd(), fscanVd(), scanVi(), and fscanVi() are available for vectors of doubles or integers.

Again, details on these can be found in the *RCI Reference Manual*.

2.12 Joint Coordinates

The data type JNTS is used to describe sets of joint coordinates for a particular manipulator. The structure is presently defined as

```
typedef struct {
        float v[NUM_JNTS];
        char type[NUM_JNTS];
        int num;
} JNTS, *JNTS_PTR;
```

where NUM_JNTS is, at the time of this writing, 6. The field v contains the value of each joint coordinate, with the field type describing its type ('R' for revolute joints and 'P' for prismatic ones). The field num gives the total number of joints, where it is assumed that num is less than or equal to NUM_JNTS. The units are normally assumed to be radians for rotational joints and millimeters for translational joints.

A set of basic routines for manipulating JNTS types is:

```
JNTS_PTR addJnts (jr, j1, j2)
  JNTS_PTR jr, j1, j2;

JNTS_PTR subtractJnts (jr, j1, j2)
  JNTS_PTR jr, j1, j2;

JNTS_PTR scaleJnts (jr, a, j1)
  JNTS_PTR jr, j1;
  float a;

JNTS_PTR combineJnts (jr, a, j1, b, j2)
  JNTS_PTR jr, j1, j2;
  float a, b;

JNTS_PTR zeroJnts (jr)
  JNTS_PTR jr;
```

The function addJnts() adds the joint values j1 to j2 and places the result in jr. The function subtractJnts() subtracts the joint values j2 from j1 and places the result in jr. The function scaleJnts() multiples the joint values j1 by a and places the result in result in jr. The function combineJnts() computes

$$\mathtt{jr} = \mathtt{a}\,\mathtt{j1} + \mathtt{b}\,\mathtt{j2}$$

and places the result in jr. zeroJnts() zeros the values of jr.

The above routines do not change either the type or num field of their result arguments (although this may change). Since the contents of type and num fields are robot specific, there must be a way to initialize them for a particular manipulator. Two routines are available to do this:

```
initJntsByName (jr, name)
  JNTS_PTR jr;
  char *name;

initJntsByManip (jr, mnp)
  JNTS_PTR jr;
  MANIP_PTR mnp;
```

`initJntsByName()` instantiates `num` and `type` for the manipulator with the indicated name. This routine can be called only from the planning level. `initJntsByManip()` instantiates `num` and `type` for a robot described by a `MANIP` structure (`MANIP` is the RCCL data type used to reference and control a robot from within a program, and will be introduced in Section 4.1).

The printing of joint values can be done using `printVf()`, with the number of elements in the "vector" being specified by the `num` field. If one wants to first convert the angle values to degrees, the following piece of code will do the job, assuming that all joints are revolute:

```
JNTS jnts;      /* initialized somewhere */

{ JNTS deg;
  scaleJnts (&deg, RADTODEG, &jnts);
  printVf ("%8.3f\n", deg.v, jnts.num);
}
```

Notice that in the call the `printVf()` the joint angles are referenced by the `v` field of the structure, since this routine expects an array of floats.

A more complicated version that considers whether or not the joints are prismatic is

```
JNTS jnts;      /* initialized somewhere */
int j;

for (j=0; j<jnts.num; j++)
 { if (jnts.type[j] == 'r')
    { printf ("%8.3f ", RADTODEG*jnts.v[j]);
    }
   else
    { printf ("%8.3f ", jnts.v[j]);
    }
 }
putchar ('\n');
```

Caveat: The addition of the `type` and `num` fields to the `JNTS` data structure is new, and not all RCCL code has been updated to use them. Prior to their introduction, the number of joints was assumed to be described by the constant `NUM_JNTS`.

3. Describing Positions in Space

Describing a manipulator task typically requires specifying positions to be reached in space (the *where*) as well as specifying aspects of the trajectory (the *how*). RCCL describes target positions using either Cartesian position equations or sets of joint angles. This chapter describes the former.

3.1 Position Equations

Position equations remove the need for absolute reference coordinates; they are also one representation of the more general concept of transformation graphs. The position relationships of a set of coordinate frames F_i, $i = 1, \ldots, n$, can be expressed in terms of transformation products. Let the transformation matrix $\mathbf{T}_i$ describe the position of the frame F_{i+1} relative to the frame F_i, with $\mathbf{T}_n$ describing the transformation from frame F_n to F_1. This is represented by the equation

$$\mathbf{T}_1 \, \mathbf{T}_2 \, \ldots \, \mathbf{T}_n \; = \; \mathbf{I}$$

where **I** is the identity matrix. A closed path of transformations from frame F_1 back to frame F_1, via the frames F_i, $i = 2, \ldots, n$, describes the position of F_1 with respect to itself. The situation can be illustrated using a directed closed graph, as shown in Figure 4, where the vertices are frames and the arcs are transforms.

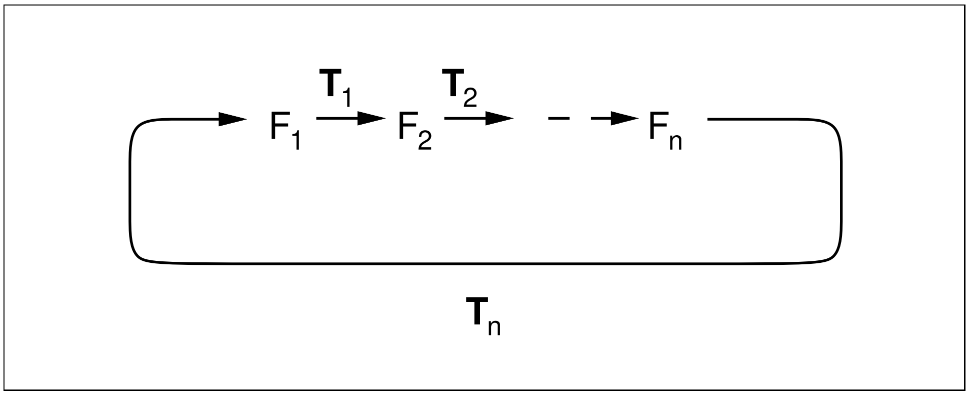

Figure 4: A directed closed transform graph.

Given a set of frames, containing two frames A and B, we can always find more than one path connecting A to B. Denoting the frames on one path by F_i, $i = 0, \ldots, n$, and the frames on the other path by G_i, $i = 0, \ldots, m$, we show such a graph in Figure 5.

The corresponding transformation equation is:

$$\mathbf{T}_0 \, \mathbf{T}_1 \, \mathbf{T}_2 \, \ldots \, \mathbf{T}_n \; = \; \mathbf{R}_0 \, \mathbf{R}_2 \, \ldots \, \mathbf{R}_m$$

Closed transformation graphs can be expressed in terms of a set of transformation equations.

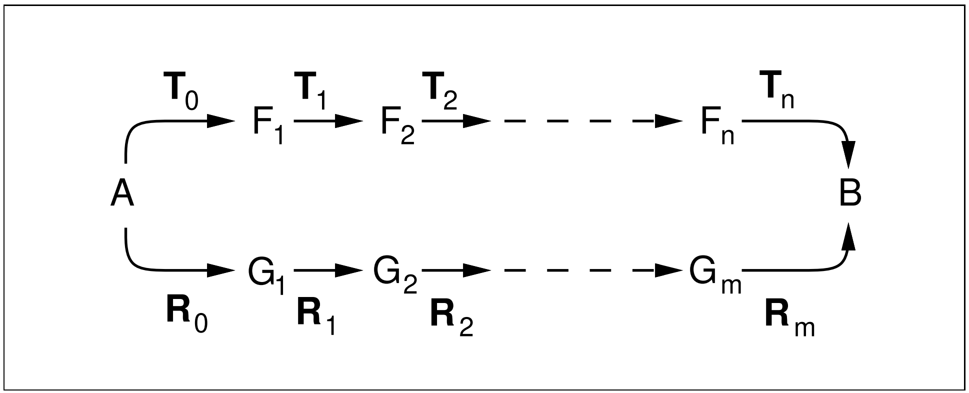

Figure 5: Transform graph containing two distinct paths between frames A and B.

RCCL uses transformation equations as one means of describing positions the manipulator has to reach. These positions always include the special manipulator transform **T6**, which we will describe next.

Assume that we have a manipulator with 6 links, labeled from 1 to 6, and a base link, labeled 0. Each of the manipulator links is assigned a frame A_i, and the transform matrix $\mathbf{A}_i$ describes its position with respect to the previous link's frame as a function of the joint variable. The position of link 1 is described with respect to the base. The transformation product

$$\mathbf{T6} \; = \; \mathbf{A}_1 \; \ldots \; \mathbf{A}_6$$

describes the position of the last link with respect to the base. Current convention is to define the transformations $\mathbf{A}_i$ according to the procedure specified by Denavit and Hartenberg [Lee 1982]. This generally will not place the last frame, A_6, directly at the end effector. Instead, to "reach" the end effector frame, it is usually necessary to provide an additional *tool* transform.

3.2 Using Position Equations to Describe Motion Targets

Let us now set up a position equation which describes a situation where the manipulator is to grasp an object lying on a table. First we must assign frames to each of the elements involved:

- Frame B is assigned to the manipulator base.
- Frame M is located in the last link of the manipulator.
- A tool is attached to link 6 and the frame T is assigned to its tip.
- Frame W describes the position of the work table.

- The position of an object lying on the table is described by frame O.

- A grasp position is described by the frame G.

Suppose that the manipulator is moving so as to grasp the object. The associated frames can be grouped together into the graph shown in Figure 6.

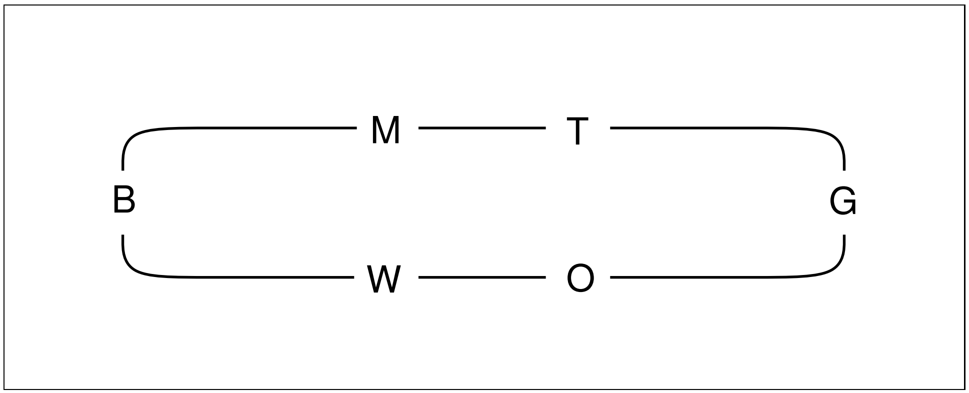

Figure 6: Example frames for a manipulator grasp task.

To obtain a transformation equation from this graph, the arcs are oriented and labeled with transforms. The choice is arbitrary but a reasonable selection is shown in Figure 7, where **END**, **BASE**, **GRASP**, and **OBJ** are predetermined transforms, and **T6** connects the manipulator base to link 6. The transform **DRIVE**, which will be discussed in more detail below, relates the current position of the tool tip to its final desired position. A diagram of where all these frames might actually be located in the workspace is given in Figure 8.

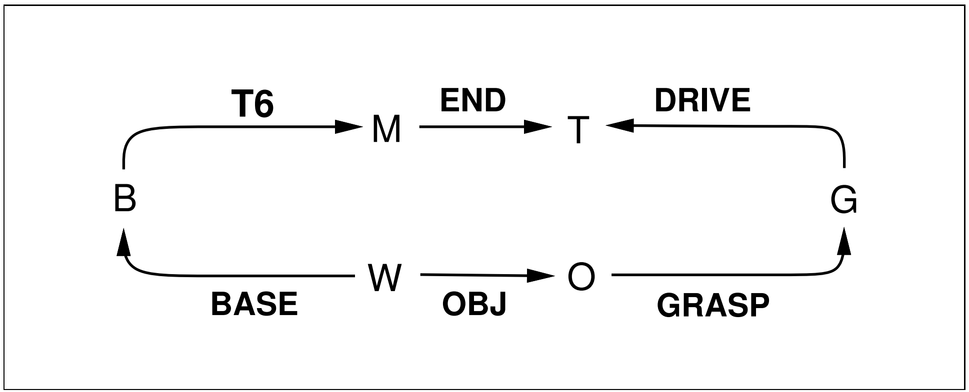

Figure 7: Example frames, plus transforms, for a manipulator grasp task.

In making the manipulator travel to its destination, it is necessary to vary **T6** from its starting value to the one defined by the position equation, which amounts to changing the value of **DRIVE** from its initial value to the identity matrix. **DRIVE** is therefore time-varying, and when appropriate will be denoted as $\mathbf{DRIVE}(t)$. If we define the time at the beginning of the motion to be 0 and the

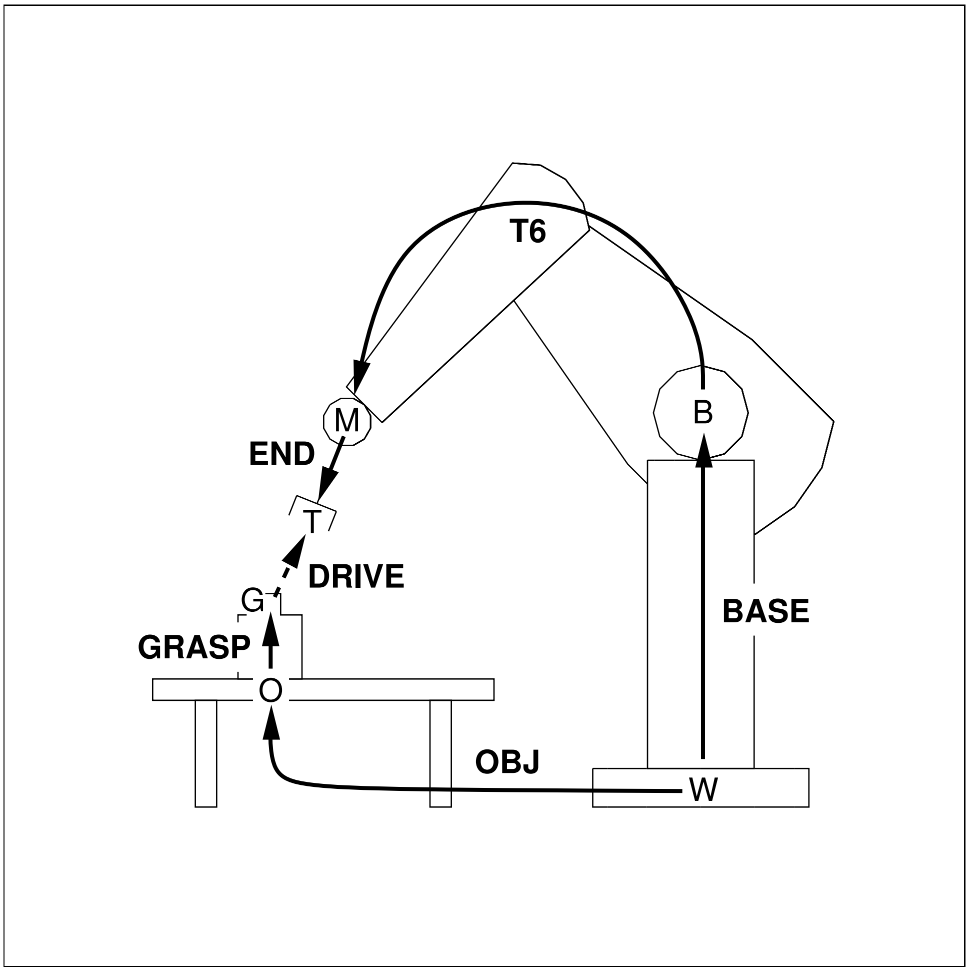

Figure 8: Example frames for grasp task superimposed on a diagram of the workspace.

time at the end of the motion to be σ, then the kinematic situation is described at the beginning of the motion by

$$\mathbf{BASE}\ \ \mathbf{T6}(0)\ \ \mathbf{END}\ =\ \mathbf{OBJ}\ \ \mathbf{GRASP}\ \ \mathbf{DRIVE}(0)$$

and at the end of the motion by

$$\mathbf{BASE}\ \ \mathbf{T6}(\sigma)\ \ \mathbf{END}\ =\ \mathbf{OBJ}\ \ \mathbf{GRASP} \tag{1}$$

By definition, $\mathbf{DRIVE}(\sigma)\ =\ \mathbf{I}$.

The two common methods of achieving this are *joint interpolation* and *Cartesian interpolation*. Joint interpolation involves solving for the target value of **T6** (*i.e.*, the value with $\mathbf{DRIVE} = \mathbf{I}$), solving for the corresponding joint values, and then interpolating to these from the initial joint values. This produces a "straight line" motion in joint coordinates. The transform **DRIVE** changes implicitly during this process, but not in a way that is necessarily obvious. On the other hand, Cartesian interpolated motion is achieved by explicitly modifying **DRIVE** so that it traces a straight line in Cartesian coordinates, and then solving for **T6** and the associated joint values during each control cycle. This produces a "straight line" in Cartesian coordinates. Rotations are handled by interpolation about one or two principal axes, as described in [Paul 1981], Chapter 5. Both forms of motion interpolation are supported by RCCL.

Transform equations can be rewritten, solved for any of the terms, or replaced by equivalent ones. For example, equation 1 can be rewritten as

$$\mathbf{BASE}\ \mathbf{T6} = \mathbf{OBJ}\ \mathbf{GRASP}\ \mathbf{END}^{-1}$$

or as

$$\mathbf{T6} = \mathbf{BASE}^{-1}\,\mathbf{OBJ}\ \mathbf{GRASP}\ \mathbf{END}^{-1}$$

Internally, when RCCL is told to move the manipulator to a given target position, it converts the corresponding equation into the canonical form

$$\mathbf{T6}\ \mathbf{TOOL} = \mathbf{COORD} \tag{2}$$

where **TOOL** and **COORD** can be composed of arbitrary numbers of sub-transforms. For equation 1, we would have

$$\begin{aligned}\mathbf{TOOL} &= \mathbf{END}\\ \mathbf{COORD} &= \mathbf{BASE}^{-1}\,\mathbf{OBJ}\ \mathbf{GRASP}\end{aligned}$$

This internal representation tells the trajectory generator where to place the **DRIVE** transform for Cartesian motions. It is inserted after the tool component:

$$\mathbf{T6}\ \mathbf{TOOL} = \mathbf{COORD}\ \mathbf{DRIVE}$$

More detailed information on how RCCL generates trajectories can be found in the paper *Trajectory Generation in Multi-RCCL*, which is part of the document set.

3.3 Frame and Transform Names

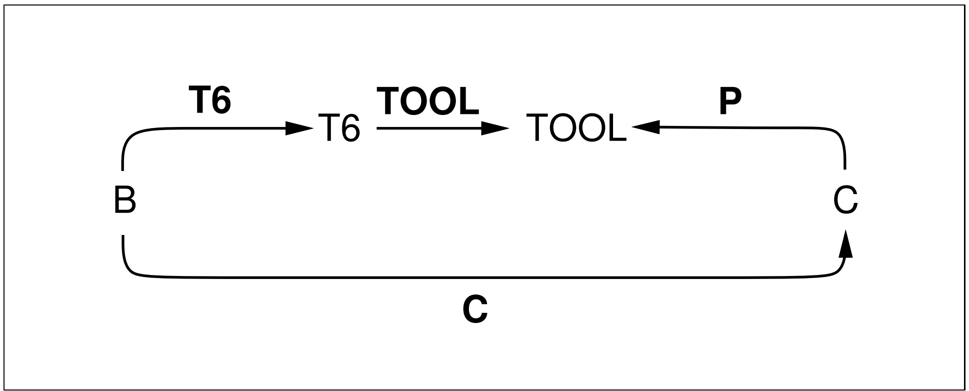

Figure 9: Transform graph where some frames have the same name as one of their incident transforms.

It is often possible to equate the name of a transform and the frame it maps to without causing confusion. For instance, in Figure 9, frames $\mathrm{T6}$, TOOL, and C are given the same names as the transforms **T6**, **TOOL**, and **C**. It is also often possible to dispense with frame names entirely.

When describing program examples, transforms may be referred to either mathematically (e.g., **T1**) or by the name of their corresponding variable within the program (e.g., `t1`). A frame will often be named after one of its incident transforms (e.g., T1).

By historical continuity, RCCL uses the name **T6** (and $\mathrm{T6}$) to denote the base-to-last-link transform product (and associated frame) for any manipulator, regardless of whether or not it actually has 6 links.

3.4 Defining Position Equations

Within an RCCL program, position equations like the ones described above can be defined with the primitive `makePosition()`. This takes a variable number of arguments which define the equation's name, left hand side, and right hand side. The function definition looks like

```
POS *makePosition (name, lhs [, lhs] ..., EQ,
                         rhs [, rhs] ..., [, TL, tl] , NULL)
  char *name;
  TRSF_PTR lhs ..., rhs ..., tl;
```

The first argument is a string naming the position. This name is optional, and can be declared as `NULL` if desired; its main purpose is to provide a key for referencing the equation from some other part of the program (section (5.5.4)). The remaining arguments are pointers to the transforms which comprise the equation. All the specified transforms must have been allocated using one of the `allocTrans()` routines; this ensures that the control level can access them when necessary. To create a position equation, simply list as arguments all the transforms forming the left-hand and right-hand sides, separate them with the predefined variable `EQ`, and terminate the argument list with `NULL`. For instance, consider the simple position equation

$$\mathbf{A}\ \mathbf{B}\ =\ \mathbf{C}\ \mathbf{D},$$

which is equivalent to the graph shown in Figure 10.

Assuming that the necessary transforms have been allocated, and pointers to them are contained in the variables `a`, `b`, `c`, `d`, this can be created with the following call:

```
POS_PTR p;

p = makePosition ("loop", a, b, EQ, c, d, NULL);
```

The function allocates a data structure representing the kinematic loop, gives it a string name `"loop"`, and returns a pointer to it. The transforms which lie on the left-hand side of the equation are assumed to be directed *clockwise*, while the transforms on the right-hand side are directed *counter-clockwise*. An arbitrary number of transform arguments may be specified to `makePosition()`, although a "safety limit" of 100 is imposed at the time of this writing.

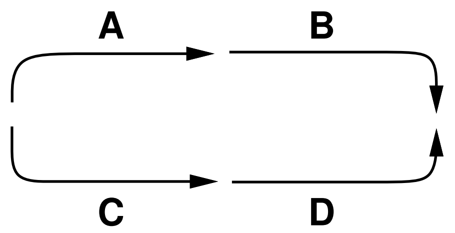

Figure 10: A sample position loop.

What can we now do with this position equation? The principal thing we can do is solve for any one of its component transforms in terms of the values of the other transforms. We can also multiply together any set of transforms which constitutes a sub-chain of the equation. The routines to do these operations are described in Section 3.5.

It should be noted that it is not necessary for the component transforms of a position equation to be set to any particular value when the position is defined. This only needs to be done before the position is actually used within the program. The equation structure itself contains only pointers to the component transforms; these pointers are used to look up the actual values "on the fly" when the equation is referenced.

The above position equation (`"loop"`) cannot be used to specify the target of a manipulator motion because it does not contain the transform **T6**. To include **T6** in an equation definition, the predefined variable `T6` should be inserted at the appropriate point into the argument list to `makePosition()`. Moreover, if we want to specify a TOOL frame that is different from the $\mathrm{T6}$ frame, then the predefined variable `TL` should be placed at the end of the argument list (before the `NULL`) and followed by a pointer `tl` to a transform denoting this TOOL frame. The transform referenced by `tl` must be one of the equation's component transforms. Sometimes, the **TOOL** component of a position equation is composed of more than one transform. If this is the case, `tl` should be the transform adjacent to the TOOL frame itself. The system will then define the total **TOOL** component to be the product of all the transforms lying between **T6** (exclusive) and `tl` (inclusive), in the same direction as **T6**.

When a TOOL frame is specified with `TL`, we can omit the `NULL` at the end of the argument list.

For example, suppose we wish to create the position equation in 1. Assume that the transforms **BASE**, **TOOL**, **OBJ**, and **GRASP** have been allocated, and pointers to them are stored in the variables `base`, `tool`, `obj`, and `grasp`. The equation can then be defined by

```
POS_PTR p;

p = makePosition ("P", base, T6, tool, EQ, obj, grasp, TL, tool);
```

If an equation definition contains `T6` but does not contain a `TL` specification, then the TOOL frame will be set equal to the $\mathrm{T6}$ frame (i.e., $\mathbf{TOOL} = \mathbf{I}$ in (2)); the same thing can be accomplished by using a `TL` specification and setting `tl` to either `T6` or `NULL`. A `TL` specification will be ignored if the equation does not contain a **T6** transform.

To illustrate the different ways of setting up the canonical components, consider the components **COORD** and **TOOL** that correspond to the following calls to `makePosition()`:

```
makePosition ("P0", T6, EQ, h, NULL); /* produces ... */
```

$$\begin{aligned} \mathbf{TOOL} &= \mathbf{I} \\ \mathbf{COORD} &= \mathbf{H} \end{aligned}$$

```
makePosition ("P1", T6, EQ, h, TL, T6); /* produces ... */
```

$$\begin{aligned} \mathbf{TOOL} &= \mathbf{I} \\ \mathbf{COORD} &= \mathbf{H} \end{aligned}$$

```
makePosition ("P2", T6, t, EQ, h, TL, t); /* produces ... */
```

$$\begin{aligned} \mathbf{TOOL} &= \mathbf{T} \\ \mathbf{COORD} &= \mathbf{H} \end{aligned}$$

```
makePosition ("P3", T6, a, EQ, h, g, TL, T6); /* produces ... */
```

$$\begin{aligned} \mathbf{TOOL} &= \mathbf{I} \\ \mathbf{COORD} &= \mathbf{H}\,\mathbf{G}\,\mathbf{A}^{-1} \end{aligned}$$

```
makePosition ("P4", T6, t1, t2, EQ, h, g, TL, t2); /* produces ... */
```

$$\begin{aligned} \mathbf{TOOL} &= \mathbf{T1}\,\mathbf{T2} \\ \mathbf{COORD} &= \mathbf{H}\,\mathbf{G} \end{aligned}$$

When a position is no longer needed, it can be deallocated with the call

```
freePosition(p)
POS_PTR p;
```

Care must be taken that the corresponding position is no longer in use. The equation's component transforms are not freed; they must be freed individually using `freeTrans()`.

3.5 Computing with Position Equations

The most basic use of position equations in RCCL is to use them directly in `move()` as the target specifications for motion requests. This will be discussed extensively in the following chapters. It is also possible to do direct computations with them. The following primitives allow a programmer to solve for a particular transform or sub-chain within an equation:

```
TRSF_PTR solveTrans (tr, p, t1, tx)
  TRSF_PTR tr, t1, tx;
  POS *p;

TRSF_PTR *solveChain (tr, p, t1, t2, tx)
  TRSF_PTR tr, t1, t2, tx;
  POS *p;

TRSF_PTR *solveInvChain (tr, p, t1, t2, tx)
  TRSF_PTR tr, t1, t2, tx;
  POS *p;
```

`solveTrans()` takes position equation `p` and solves it for the transform `t1` (which must be an element of `p`). If `p` contains a **T6** transform, but it is not the transform being solved for, then the argument `tx` is used to provide a specific value for **T6**. Otherwise, `tx` can be specified as `NULL`. `solveTrans()` returns a pointer to its result argument.

For example, suppose we have the position equation

$$\mathbf{Z}\ \mathbf{T6}\ \mathbf{E} = \mathbf{A}\ \mathbf{B}\ \mathbf{C} \tag{3}$$

representing the kinematic graph in Figure 11.

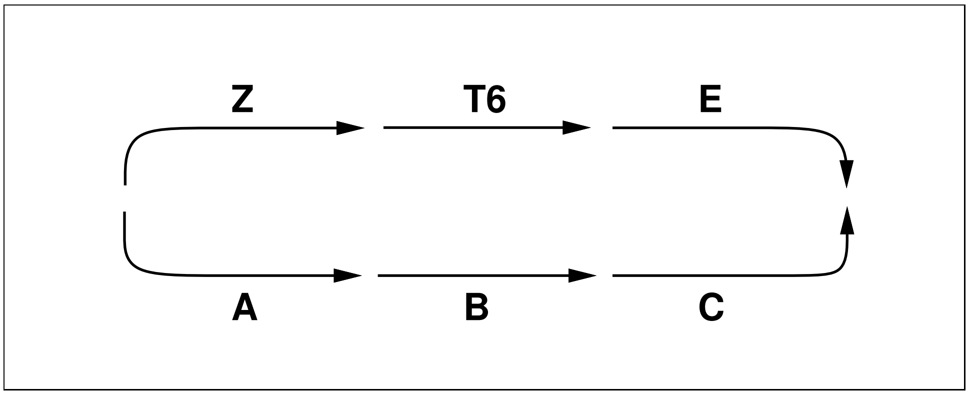

Figure 11: A sample position loop containing the manipulator transform **T6**.

This can be declared with the `makePosition()` call

```
p = makePosition ("p", z, T6, e, EQ, a, b, c, TL, e);
```

If at some later time we wish to solve for **A**, we can call

```
solveTrans (r, p, a, tx);
```

which will compute

$$\mathbf{A} = \mathbf{Z}\ \mathbf{TX}\ \mathbf{E}\ \mathbf{C}^{-1}\ \mathbf{B}^{-1}$$

and place the result into `r`. The original value of `a` is left unchanged.

`solveChain()` and `solveInvChain()` are used to evaluate sub-sections of a position equation.

`solveChain()` goes clockwise around the kinematic loop described by `p` and multiplies together all transforms found between `t1` and `t2`, inclusive (recall that the clockwise-directed transforms are those that appear on the left-hand side of the equation). If one of the transforms is directed counterclockwise, then its inverse value is used in the multiplication. Again, values for the transform **T6** are supplied, if necessary, by the argument `tx`. For example, in (3), the calls

```
solveChain (r, p, a, c, tx);

solveInvChain (r, p, T6, c, tx);
```

would compute

$$\mathbf{R} = \mathbf{A}^{-1}\ \mathbf{Z}\ \mathbf{TX}\ \mathbf{E}\ \mathbf{C}$$

and

$$\mathbf{R} = \mathbf{TX}^{-1}\ \mathbf{Z}^{-1}\ \mathbf{A}\ \mathbf{B}\ \mathbf{C}^{-1}$$

respectively.

`solveInvChain()` does the same thing as `solveChain()`, except it goes around the loop counterclockwise.

Kinematic equation loops comprise a restricted case of *transformation graphs*. Multi-branch graphs can be useful in modeling situations where several time varying coordinate frames are interrelated. They can be implemented using the primitives described above, and/or the basic transform multiplication routines.

As a simple example, consider the following: suppose two manipulators are working together on a task, with manipulator 1 occasionally putting down an object that manipulator 2 has to pick up. Assume that the relative location of the two robots is described by the transform **BASE2**, which maps from the base frame of the first robot to the base frame of the second (see Figure 12). Assume also that the first robot is being guided by real-time sensors, so it is not known in advance exactly where the object will be put down. Let the "put down" spot be described by the position equation by

$$\mathbf{T6}\ \mathbf{E}_1 = \mathbf{PLACE} \tag{4}$$

where $\mathbf{E}_1$ is a tool transform and **PLACE** locates the object's grasp point relative to the manipulator's base. When the object is actually put down, we can solve for all the components of (4) exactly. Now, let the "pick up" spot for the second robot be described by

$$\mathbf{T6}\ \mathbf{E}_2 = \mathbf{PICK} \tag{5}$$

Picking the object up requires moving the second robot's TOOL frame to the same spatial location that was occupied by the first robot's TOOL frame. This amounts to finding a value for the transform

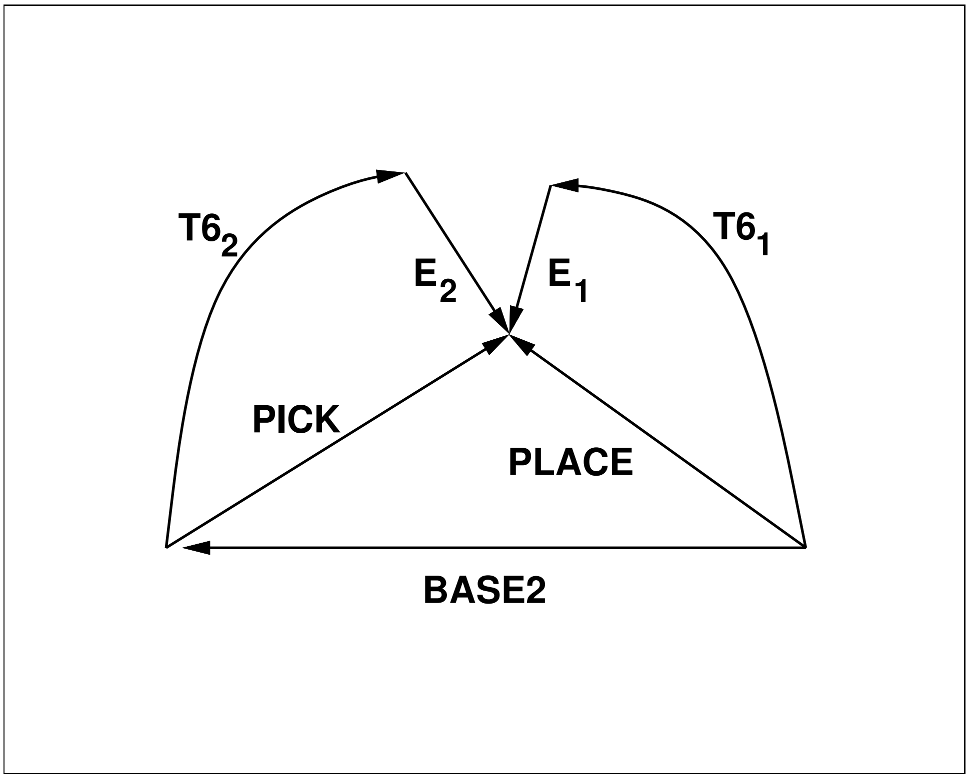

Figure 12: Kinematic diagram for a cooperative pick-and-place operation.

PICK. From Figure 12, we can see that there is a third position equation,

$$\mathbf{BASE2\ \ PICK} = \mathbf{PLACE}, \tag{6}$$

which can be used to solve for **PICK**.

Within a program, these three position equations could be defined as follows:

```
TRSF_PTR pick, place, base2, e1, e2;
POS_PTR pickPos, placePos, refPos;

placePos = makePosition ("placePos", T6, e1, EQ, place, TL, e1);
pickPos = makePosition ("pickPos", T6, e2, EQ, pick, TL, e2);
refPos = makePosition ("refPos", base2, pick, EQ, place, NULL);
```

`placePos` can be used as a target position for the first manipulator's placing motion. Assume that the value of the transform **PLACE** is modified by a sensor until the object is actually put down (ways of doing this will be presented later). When the placement is finished, we can solve for **PICK** using

```
solveTrans (pick, refPos, pick, NULL);
```

and then `pickPos` can be used as a target position for the second manipulator's pickup motion.

This particular example is simple enough that if a programmer were actually implementing it, he/she would probably dispense with `refPos` altogether and simply call

```
multLiTrsf (pick, base, place);
```

directly. Whether or not the creation of explicit position equations is advantageous depends heavily on the complexity of the task. In particular, position equations provide a compact way to describe general kinematic loops to another module (such as when they are used by the `move()` primitive to describe a motion target to the trajectory generator).

4. Controlling a Robot

This section describes the primitives available for doing actual robot control.

4.1 The Create and Delete Primitives

When an RCCL program wishes to control a robot, it must first create a MANIP structure for that robot. This is done using the call

```
MANIP_PTR rcclCreate (robotName, cpuMask)
  char *robotName;
  int cpuMask;
```

which looks up the named robot in the RCI system tables (the robot must be configured into RCI), allocates the necessary control data structures for it (including the MANIP structure), and assigns this robot to a trajectory task on one of the CPUs specified by the bitmask cpuMask, creating that task if necessary. The last argument is important only if the RCCL/RCI site does in fact support multiple CPUs. In either case, it is usually set to 0, which tells the system to automatically select a CPU from those which are available.

The MANIP pointer returned by rcclCreate() is the handle used for all subsequent control of that robot. When the robot is no longer needed, its MANIP structure may be removed with the call rcclDelete(mnp).

4.2 The MANIP structure

Information about the robot can be obtained from the MANIP structure. The fields which normally concern the programmer look like this:

```
typedef struct {

   /* Cartesian level information */

   TRSF *t6;
   TRSF *t6o;
   TRSF *here;
   TRSF *tool;
   TRSF *rest;
   TRSF *lastTC;

   POS *park;
   POS *last;
```

```
        /* Joint level information */

        JNTS *j6;
        JNTS *j6o;
        JNTS *jvel;
        int handPos;

        /* Robot information */

        RCI_RBT *rbt;
        HOW *how;
        JLS *jls;
        KYN *kyn;
        void *var;

    } MANIP, *MANIP_PTR;
```

`t6` contains the *commanded* value of the manipulator **T6** transform computed by the trajectory generator during the most recent control cycle.

`t6o` contains the *observed* value of **T6**, as read back from the robot during the last control cycle. For computational efficiency, this field is maintained only if the mode `T6O_EVAL` is selected for the manipulator (see Section 9.1.2). Differences between the values in `t6o` and `t6` are due to time delays and tracking errors associated with the low-level servo controllers.

`here` contains the value of the `t6` field at the point where the last motion ended (as in "at the end of the last motion, **T6** was *here*"). See Section 4.4.1 for a discussion of just when a motion "ends".

`tool` contains the value of the current target position's **TOOL** component, as computed by the trajectory generator during the most recent control cycle.

`rest` is a fixed read-only transform corresponding to the park position of the manipulator. The actual definition of this is a little up in the air; at the moment, the system looks up the values of the robot's "park angles" in the its `JLS` structure (see Section 6.1.1) and computes the corresponding **T6** transform value.

`lastTC` is the target value of **T6** towards which the manipulator was heading at the end of the last motion (this differs from `here` because of transitioning between path segments; **T6** usually comes close to `lastTC` but does not achieve it; see Section 4.4.1).

`park` is a built-in position equation corresponding to the manipulator's park position:

$$\mathbf{T6} = \mathbf{REST}$$

where **REST** is the transform contained in the `rest` field. This allows moves to the park position to be simply specified with the following command:

```
move (mnp, mnp->park);
```

`last` is a built-in position equation which describes the last target the manipulator was moving to:

$$\mathbf{T6} = \mathbf{LASTTC}$$

where **LASTTC** is the transform contained in the `lastTC` field. It is particularly useful in conjunction with the `distance()` primitive to create relative motions (see below).

`j6` contains the *commanded* manipulator joint values computed by the trajectory generator during the most recent control cycle (if you apply forward kinematics to these, you will get the values in the `t6` field).

`j6o` contains the actual *observed* joint values, read back from the robot during the last control cycle. For computational efficiency, this field is maintained only if the mode `T6O_EVAL` is selected for the manipulator (see Section 9.1.2). Differences between the values in `j6o` and `j6` are due to time delays and tracking errors associated with the low-level servo controllers.

`jvel` contains the current output (computed) joint velocities.

The fields `rbt`, `how`, `jls`, `kyn`, and `var` point to memory areas describing the low level RCI interface to the robot. In particular, they describe parameters and terms relating to a particular robot's physical characteristics. These fields are discussed in detail in Section 6.1.1.

Other information can be obtained from the `MANIP` structure using special macros. The macro `MANIP_CPU(mnp)` returns the CPU number of the manipulator's trajectory task. The macro `MANIP_TASK(mnp)` returns the RCI task descriptor for the manipulator's trajectory task (you won't really need this, but check the *RCI User's Guide* if you are curious).

4.3 Running the Trajectory Generator

It is necessary to turn on the trajectory generator task before the robots can actually be controlled. The routine to do this is

```
rcclStart()
```

The trajectory generator will then begin executing at a regular sample interval, the default value of which is a parameter in the `.rciparams` file (see Section 9.1.3). The sample interval can be read back or changed from within the RCCL program using the primitives

```
rcclGetInterval()

rcclSetInterval(interval)
```

It is illegal to call `rcclSetInterval()` (and `rcclCreate()` as well) while the trajectory generator is running.

Turning the trajectory generator off is done with the routine

```
rcclRelease(powerOff)
```

The parameter `powerOff` is a boolean which, if true, causes the robot arm power to be turned off as well (unless the option `RCCL_LEAVE_POWER_ON` is set).

To determine if the trajectory generator is running, the routine

```
rcclActive()
```

can be called; this returns true if the trajectory generator is running.

4.4 Specifying Motions

4.4.1 The Basic Motion Mechanism

The basic primitives to request manipulator motions are:

```
move (mnp, pos)
  MANIP *mnp;
  POS_PTR pos;

movej (mnp, jnts)
  MANIP *mnp;
  JNTS_PTR jnts;
```

`move()` tells the trajectory generator to move the manipulator `mnp` to the target specified by `pos`, which must be a position equation returned by `makePosition()`. `movej()` tells the system to move the manipulator to a target specified by a set of joint values.

These primitives do not wait until the manipulator has reached the specified target point. Instead, they simply queue up a *motion request* and return. The trajectory generator task services the motion requests in FIFO fashion (Figure 13). This decouples the activity of the planning level from the control level, but also means that explicit synchronization primitives must be provided. The simplest of these is the `waitForCompleted()` primitive described in the first example program; others will be presented later.

The planning level can set parameters controlling various motion characteristics, such as the desired velocity or the interpolation mode; these will be discussed below. Usually these parameters are lumped in and queued with the motion requests themselves, so they take effect only with the next motion request.

Each motion request serviced by the trajectory generator corresponds to a single *motion path segment*. When finishing one path segment and starting another, it is generally necessary to provide a smooth transition from the first path segment into the second. The purpose of the transition is to prevent the manipulator from undergoing large impulses in acceleration or higher derivatives. This is illustrated (with a one-dimensional example) in Figure 14, which shows a two path segment trajectory in which the manipulator is asked to move first from A to B, and then from B to C. The idealized trajectory is a dotted line, and the real trajectory is a solid line. At the start of each path segment there is a transition region which is symmetric about the nominal (or *official*) starting time for the path segment. The time on each "side" of this transition is τ control cycles, which makes the total transition time 2τ cycles; this is marked for the transition between path segments $\overline{AB}$ and $\overline{BC}$. Outside of the transition zone, the manipulator "coasts" at a constant velocity. The total time allocated for the path segment (from the midpoint of one transition to the midpoint of the next) is σ

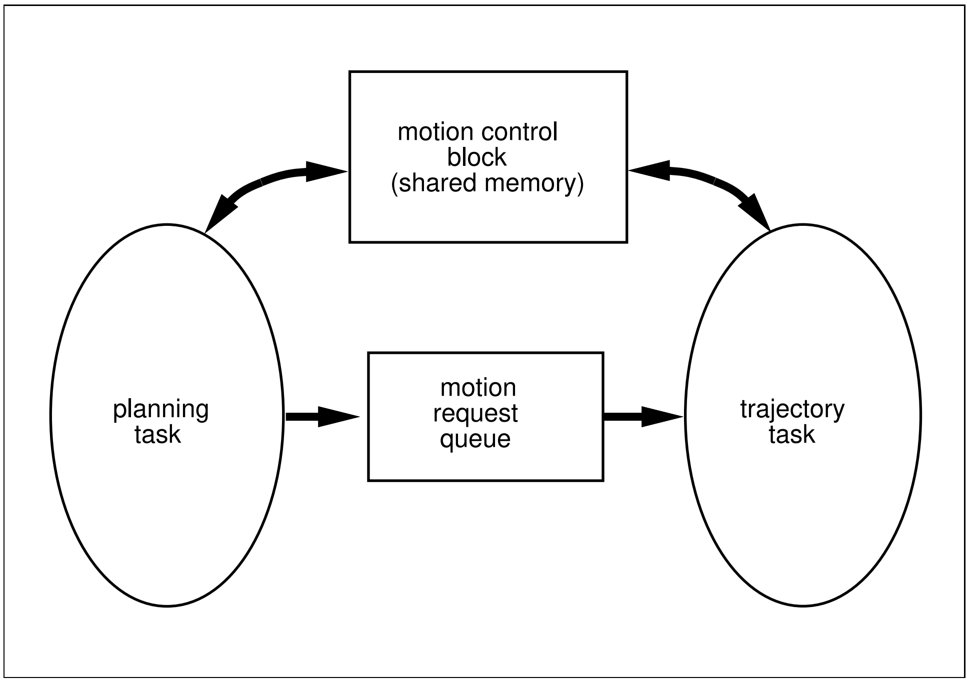

Figure 13: Motion queue and shared memory block linking the planning and control levels internally. The queue is for storing motion request packets, while the shared memory block is for information that is transferred immediately.

motion counts. The trajectory generator normally computes the values of τ and σ for each motion "on the fly", using the velocity and acceleration limits currently specified for the manipulator.

The existence of transition regions creates some ambiguity about when one motion segment ends and another begins. By default, RCCL defines the *official* time at which this happens to be the **midpoint** of the transition region[1] (see Figure 15). This is when the `here` transform and various motion status flags are updated. On the other hand, the actual computations associated with a path segment (including the execution of monitor and transform functions, which will be discussed later) are started at the **beginning** of the transition interval (and the corresponding computations for the previous path segment are halted). This is an artifact of the way RCCL computes trajectories. If this slight inconsistency causes trouble, then the official motion change over time can be set to the beginning of the transition interval as well (see Section 9.1.2).

4.4.2 Stopping at Target Points

Because of the motion transitions, a manipulator will generally not pass directly through its target point unless it is made to stop there; instead, it will tend to "undercut" the target point. This is the difference between the `lastTC` and `here` fields (in the `MANIP` structure) which are updated at the end of each motion: `lastTC` corresponds to the ideal target position (B, in Figure 15), while `here` corresponds to the point actually reached by **T6**.

[1] Except for "stop" motions, described in the next section, which are considered to end at the beginning of the transition to the next motion.

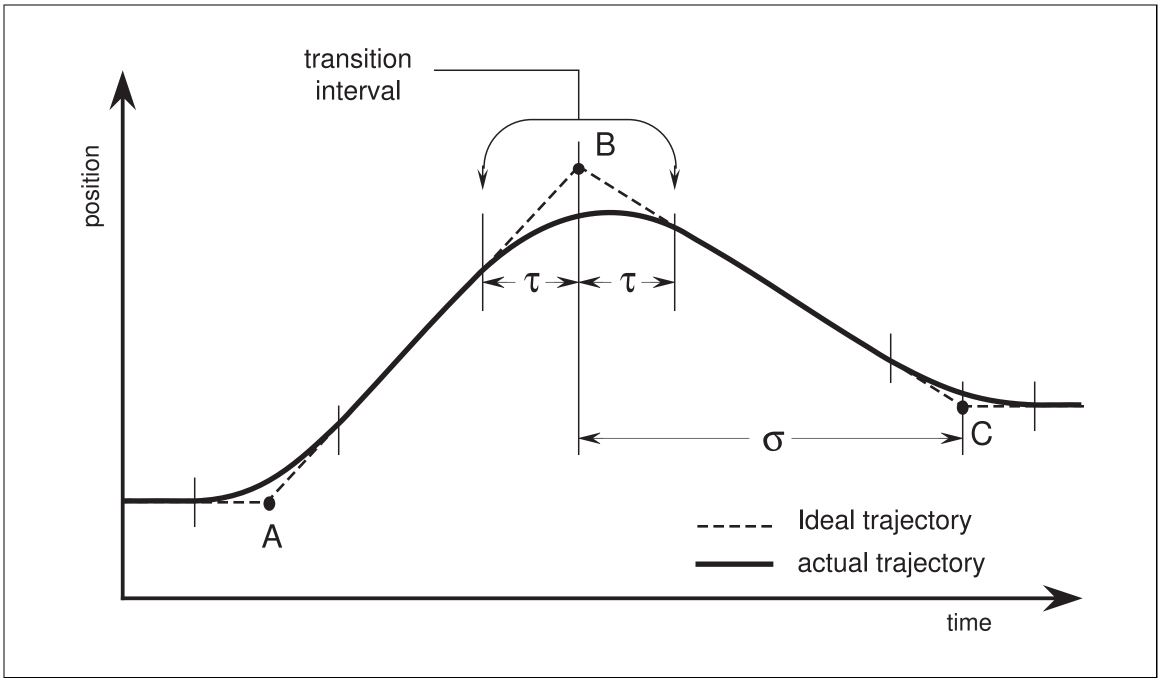

Figure 14: Manipulator trajectories are created by blending together individual path segment, as illustrated here in one dimension.

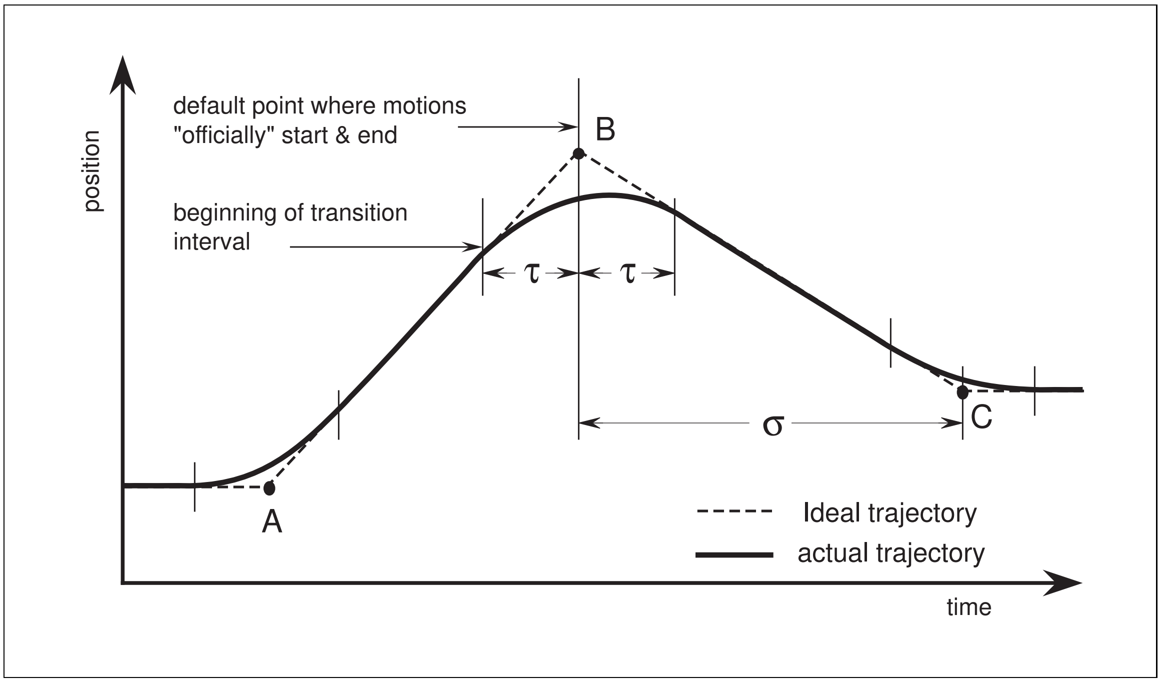

Figure 15: The “switchover” point for motions is generally taken to be the midpoint of the transition region.

In RCCL, stopping a robot is accomplished by essentially moving to the same target position *twice*. The second "motion" ensures that the manipulator will actually come to rest at the target, as illustrated in Figure 16, which shows the manipulator stopping at B for a short time. This is the same technique as that described in Richard Paul's book [Paul 1981].

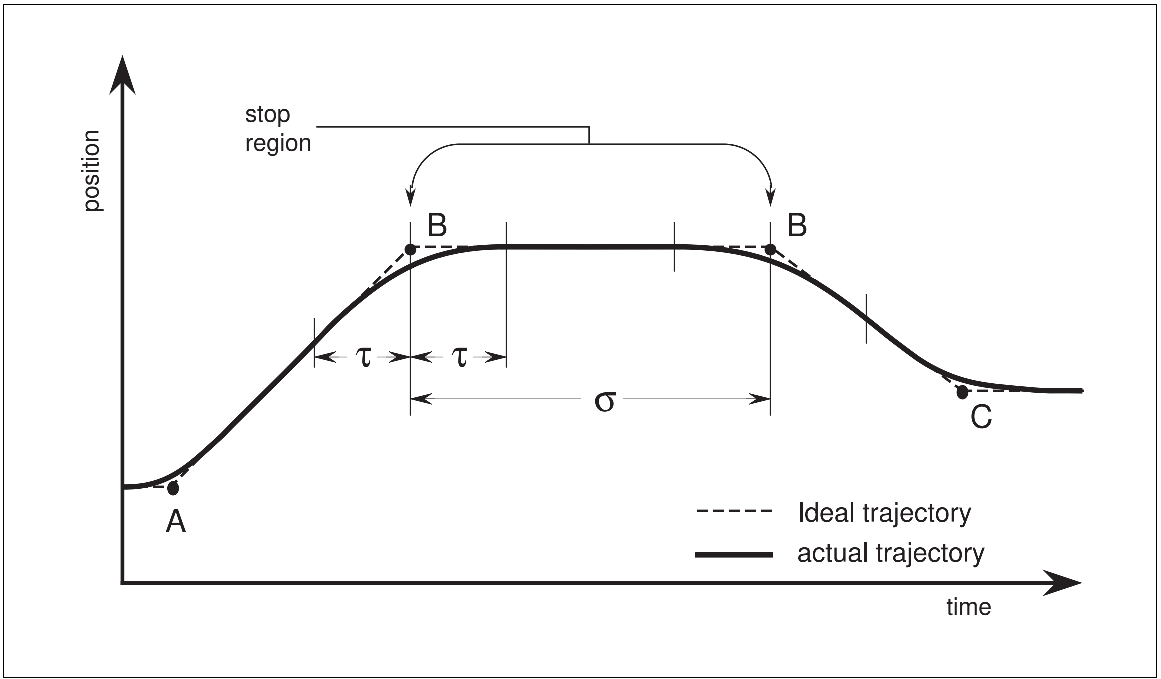

Figure 16: Stopping at a point is done using two path segments which have the same target.

Motion requests to stop the manipulator are invoked by the `stop()` primitive,

```
stop (mnp, time)
  MANIP *mnp;
  float time;
```

which stops the manipulator at the last target position for `time` milliseconds.

Note that stopping at a target position does not necessarily mean the manipulator will physically come to rest, since if the target itself is moving in time, then the manipulator will continue to track it. One the other hand, when all the motion requests for a manipulator have been executed, the trajectory generator will normally bring it to an absolute stop at wherever it happens to be. In other words, when a program waits for all requested motions to finish with the macro

```
waitForCompleted (mnp);
```

the manipulator will usually be brought to an absolute stop. This an intuitively reasonable thing to do: "no motion has been specified, so don't do anything". On the other hand, it is sometimes desirable in these circumstances to have the robot continue to track the last target position[2]. This behavior can be selected by setting `TRACKING_MODE` (Section 9.1.2) for the manipulator in question;

[2]This is what the original version of RCCL did.

if this is done, `waitForCompleted()` will still block until all explicit motions have finished, but the manipulator might not be brought to an absolute stop.

Although similar in effect to issuing another move to the last target position, *stop* requests are slightly different from *move* requests in several ways. First, they ignore the setting of certain motion parameters. In particular, the effect of calls to `setMod()`, `setConf()`, `distance()`, and `setDistance()` (see below) is held over until the next *move* request. Second, the trajectory generator treats stop requests differently in that no "drive" computation is performed for them. Last, a stop motion is considered to be officially "over" when the transition to the next motion segment begins (instead of at the transition region midpoint).

4.5 Setting Motion Parameters

4.5.1 Interpolation Mode

As described in Section 3.2, RCCL implements two ways of moving between target positions: joint interpolation, also known as *joint mode*, and Cartesian interpolation, or *Cartesian mode*. Because a joint interpolated path always exists between two well defined end points, joint mode is usually preferred for large motions. Cartesian mode, on the other hand, generates trajectories which are easy to visualize and which are often suited to particular types of tasks.

Motions specified with `move()` are performed in either joint or Cartesian mode, depending on which is currently selected. Selection of the interpolation mode is done on a per-manipulator basis with the primitive

```
setMod (mnp, c)
  MANIP *mnp;
  char c;
```

`c` should be set equal to `'j'` for joint mode and `'c'` for Cartesian mode; Joint mode is the default. A change introduced by `setMod()` takes effect with the next requested motion, and remains in effect for all subsequent motions. As an illustration of its usage, consider the following:

```
MANIP *mnp;
POS_PTR p, p1, p2;
int i, m;

p1 = makePosition(...);
p2 = makePosition(...);

for (move(mnp, p2), i = 0; i < 10; ++i)
 { if (i % 2 != 0)
    { m = 'c';
      p = p1;
    }
   else
```

```
        { m = 'j';
          p = p2;
        }
       setMod (mnp, m);
       move (mnp, p);
    }
```

This will cause the arm to move from `p2` to `p1` (`i` odd) in *Cartesian* mode and from `p1` to `p2` (`i` even) in *joint* mode. Of course, C experts will insist on coding the same thing like this:

```
    for (move(mnp, p2), i = 10; i--;)
     { setMod (mnp, (m = i % 2) ? 'c' : 'j');
       move (mnp, m ? p1 : p2);
    }
```

To read back the mode setting for a manipulator, the routine

```
    getMod (mnp)
```

may be used. Since the mode setting has to propagate through the motion queue before actually taking effect, the mode setting read back with `getMod()` may not be the one currently in use by the trajectory generator. The routine

```
    getActiveMod (mnp)
```

returns the interpolation mode in effect for the current motion.

Motions specified with `movej()` are always joint interpolated.

4.5.2 Setting Velocities and Motion Times

There are many ways to specify the desired manipulator velocity.

The simplest way to change the robot speed is to call the routine

```
    setSpeed (mnp, scale)
      MANIP *mnp;
      float scale;
```

which simply scales the base velocity up or down (the default value for `scale` is, predictably, 1.0). The same effect can be achieved by setting the `speed` parameter in the `.rciparams` file (see Section 9.1.3); this speed parameter is combined with whatever the programmable speed scale happens to be. The new speed setting will take effect with the next requested motion. The speed setting can be read back using the routine `getSpeed(mnp)`.

Setting a manipulator base velocity is slightly more complicated since it depends on the interpolation mode: when doing the straight line motion associated with Cartesian mode, one wishes to specify velocities in terms of something like mm. per sec. for translation and degrees per sec. for rotation. When moving in joint interpolated mode, the velocities need to be specified in units particular to the joints themselves.

For controlling the Cartesian mode velocity, the following primitives are available:

```
setCartVel (mnp, transVel, rotVel)
  MANIP *mnp;
  float transVel, rotVel;

getCartVel (mnp, transVel, rotVel)
  MANIP *mnp;
  float *transVel, *rotVel;

getDefaultCartVel (mnp, transVel, rotVel)
  MANIP *mnp;
  float *transVel, *rotVel;
```

`setCartVel()` sets the base Cartesian velocity to `transVel` mm. per second and `rotVel` degrees per second. These represent *limits*: the arm will accelerate to the lowest of the two bounds and then maintain that speed. For example, for a motion involving a 30 millimeter translation and a 30 degree rotation, the call

```
setCartVel (mnp, 30.0, 300.0);
```

will result in a 1 second motion caused by the translation constraint, overriding the $1/10$ of a second time needed to perform the rotation. If either of the arguments is specified as `F_UNDEF`, then it is left unchanged; if either is specified as `F_DEFAULT`, then it is set to the system default value. `getCartVel()` and `getDefaultCartVel()` allow the currently selected values and the system default values to be read back. There is no `getActiveCartVel()` per se, since the base velocities and scale factors are combined into a single net velocity before being sent to the trajectory generator; for the equivalent function, see Section 4.5.4.

For controlling the velocity of joint interpolated motions, the following primitives are available:

```
setJointVel (mnp, jvel)
  MANIP *mnp;
  JNTS *jvel;

getJointVel (mnp, jvel)
  MANIP *mnp;
  JNTS *jvel;

getDefaultJointVel (mnp, jvel)
  MANIP *mnp;
  JNTS *jvel;

setJvelScale (mnp, scale)
  MANIP *mnp;
  float scale;

getJvelScale (mnp)
  MANIP *mnp;
```

`setJointVel()` sets the base velocities for each joint to the values given by `jvel`. These will be millimeters per second for prismatic joints and degrees per second for rotational joints. Again, these represent *limits*: the arm will accelerate to the lowest of the bounds and then maintain that speed. If any of the velocities are specified as `F_UNDEF`, then they are left unchanged; if any are specified as `F_DEFAULT`, then they are set to the system default value. `getJointVel()` and `getDefaultJointVel()` allow the currently selected values and the system default values to be read back. The overall joint velocities may be scaled using `setJvelScale()` and `getJvelScale()`; the joint velocity scale factor acts in addition to the scale factor specified with `setSpeed()`. There is no `getActiveJointVel()` per se, since the base velocities and scale factors are combined into a single net velocity before being sent to the trajectory generator; for the equivalent function, see Section 4.5.4.

4.5.3 Setting Acceleration

The acceleration limits for a manipulator control the length of the path segment transition times. The higher the allowed accelerations, the shorter the transition times. Acceleration limits can be specified with the following primitives:

```
setAccelScale (mnp, scale)
  MANIP *mnp;
  float scale;

float getAccelScale (mnp)
  MANIP *mnp;

setCartAccel (mnp, transAcc, rotAcc)
  MANIP *mnp;
  float transAcc, rotAcc;

getCartAccel (mnp, transAcc, rotAcc)
  MANIP *mnp;
  float *transAcc, *rotAcc;

setJointAccel (mnp, jacc)
  MANIP *mnp;
  JNTS *jacc;

getJointAccel (mnp, jacc)
  MANIP *mnp;
  JNTS *jacc;
```

These routines are analogous to `setSpeed()`, `getSpeed()`, `setCartVel()`, `getCartVel()`, `setJointVel()`, and `getJointVel()`. `setAccelScale()` scales the acceleration limits for both Cartesian and joint mode motions; the default scale value is 1.0. `getSpeed()` reads back the acceleration scale. `setCartAccel()` sets the acceleration limits for Cartesian motions to `transAcc` $\mathrm{mm/sec}^2$ and `rotAcc`

$\mathrm{deg/sec}^2$, while `setJointAccel()` sets the limits for joint interpolated motions to the values given by `jacc` ($\mathrm{mm/sec}^2$ for prismatic joints and $\mathrm{deg/sec}^2$ for rotary joints). Specifying any of the accelerations as `F_UNDEF` causes the corresponding value to be unchanged; specifying `F_DEFAULT` causes the system default value to be used. The routines `getCartAccel()` and `getJointAccel()` allow the current values to be read back. Changes created by any of these routines take effect with the next requested motion.

4.5.4 More on Velocity and Acceleration Limits

Assume that s is the speed scale factor controlled by `setSpeed()`, s_g is the "program global" speed factor set in the `.rciparams` file (Section 9.1.3), v_{bt} and v_{br} are the "base" translational and rotational velocities controlled by `setCartVel()`, $\mathbf{v}_{bj}$ is the "base" joint velocity vector controlled with `setJointVel()`, and s_j is the joint velocity scale controlled by `setJvelScale()`. These terms are combined into "net" velocities, described by v_t, v_r, and $\mathbf{v}_j$, as follows:

$$v_t = s\, s_g\, v_{bt}$$

$$v_r = s\, s_g\, v_{br}$$

$$\mathbf{v}_t = s\, s_g\, s_j\, \mathbf{v}_{bj}$$

These net velocities are bundled into each motion request. The trajectory generator uses them as required to determine an appropriate value of σ (Section 4.4.1) for the motion.

The values of v_t, v_r, and $\mathbf{v}_j$ which are in effect for the current manipulator motion can be read back using the routines

```
getActiveCartVel (mnp, transVel, rotVel)
  MANIP *mnp;
  float *transVel, *rotVel;

getActiveJointVel (mnp, jvel)
  MANIP *mnp;
  JNTS *jvel;
```

Acceleration limits are handled similarly. Let α be the acceleration scale controlled by `setAccelScale()`, a_{bt} and a_{br} be the "base" translational and rotational accelerations controlled by `setCartAccel()`, and $\mathbf{a}_{bj}$ be the "base" joint acceleration vector controlled with `setJointAccel()`. The "net" acceleration limits a_t, a_r, and $\mathbf{a}_j$ are computed as follows:

$$a_t = \alpha\, a_{bt}$$

$$a_r = \alpha\, a_{br}$$

$$\mathbf{a}_j = \alpha\, \mathbf{a}_{bj}$$

These are bundled into each motion request and used by the trajectory generator to estimate an appropriate transition time τ. The values of a_t, a_r, and $\mathbf{a}_j$ in effect for the current manipulator motion can be read back using the routines

```
getActiveCartAccel (mnp, transAcc, rotAcc)
  MANIP *mnp;
  float *transAcc, *rotAcc;

getActiveJointAccel (mnp, jacc)
  MANIP *mnp;
  JNTS *jacc;
```

Finally, there is another routine, `getActiveMotionCounts()`, which permits the program to read back the values of τ and σ (the times for the incoming transition and motion duration, respectively, in units of control cycles; see Section 4.4.1), which are in effect for the current motion:

```
getActiveMotionCounts (mnp, tau, sigma)
  MANIP *mnp;
  int *tau, *sigma;
```

4.6 Program Example: "box"

This is a simple program which repeatedly moves the robot around the edges of a square "box" while demonstrating the effect of changing the speed and acceleration specifications.

```
#include <rccl.h>                                               /*#1*/
#include "manex.560.h"

#define BOXSIZE         150.0

main()
{
        TRSF_PTR e, b, c[4];                                    /*#2*/
        POS_PTR corners[4], start;
        MANIP *mnp;
        char *robotName;
        int k, ia, iv;
                                                                /*#3*/
        static float accelValues[] = { F_DEFAULT, 4000.0, 20.0 };
        static char *accelDesc[] = { "default", "high", "low" };
        static float velValues[] = { F_DEFAULT, 400.0, 40.0 };
        static char *velDesc[] = { "default", "fast", "slow" };

        rcclSetOptions (RCCL_ERROR_EXIT);                       /*#4*/
        robotName = getDefaultRobot();                          /*#5*/

        e = allocTransXyz ("E", UNDEF, 0.0, 0.0, TOOLZ);        /*#6*/
        b = allocTransRot ("B", UNDEF, B_X, B_Y, B_Z, xunit, 180.0);

        start = makePosition ("start", T6, e, EQ, b, TL, e);    /*#7*/

        c[0] = allocTransXyz (NULL, UNDEF,  BOXSIZE/2, 0.0, -BOXSIZE/2);
        c[1] = allocTransXyz (NULL, UNDEF, -BOXSIZE/2, 0.0, -BOXSIZE/2);
```

```
        c[2] = allocTransXyz (NULL, UNDEF, -BOXSIZE/2, 0.0,  BOXSIZE/2);
        c[3] = allocTransXyz (NULL, UNDEF,  BOXSIZE/2, 0.0,  BOXSIZE/2);
        for (k=0; k<4; k++)                                     /*#8*/
         { corners[k] = makePosition (NULL, T6, e, EQ, b, c[k], TL, e);
         }

        mnp = rcclCreate (robotName, 0);                        /*#9*/
        rcclStart();

        move (mnp, start);                                      /*#10*/
        stop (mnp, 1000.0);
        waitForCompleted (mnp);                                 /*#11*/

        setMod (mnp, 'c');                                      /*#12*/

        for (ia=0; ia<ARRAY_SIZE(accelValues); ia++)            /*#13*/
         { for (iv=0; iv<ARRAY_SIZE(velValues); iv++)
            { float vel, accel;

              setCartAccel (mnp, accelValues[ia], F_UNDEF);
              setCartVel (mnp, velValues[iv], F_UNDEF);
              getCartAccel (mnp, &accel, (float*)NULL);
              getCartVel (mnp, &vel, (float*)NULL);

              printf ("%s vel (%g), %s acceleration (%g):\n",
                      velDesc[iv], vel, accelDesc[ia], accel);

              moveSet (mnp, corners, ARRAY_SIZE(corners));      /*#14*/
              waitForCompleted (mnp);                           /*#15*/
            }
         }

        setMod (mnp, 'j');                                      /*#16*/
        move (mnp, start);
        stop (mnp, 1000.0);
        waitForCompleted (mnp);                                 /*#17*/

        rcclRelease (YES);
}

moveSet (mnp, p, size)
MANIP *mnp;
POS_PTR *p;
int size;
{
        int k;

        for (k=0; k<size; k++)
         { move (mnp, p[k]);
         }
}
```

NOTE – this example has been coded for the PUMA 560 robot, and lives at `demo.rccl/box.560.c`. *An equivalent program for the PUMA 260 is contained in* `box.260.c`

As with `simple.560`, the program begins by including the standard file `<rccl.h>` (#1) and the manual example file `"manex.560.h"`, and declaring pointers for the various transform and position structures (#2).

The arrays `accelValues` and `velValues` (#3) store the different Cartesian acceleration and velocity values that will be used. Note the explicit use of the floating point quantity `F_DEFAULT`. The arrays `accelDesc` and `velDesc` contain string information that the program will print as it executes.

As with the first example, the program sets the `RCCL_ERROR_EXIT` option to enable an automatic program abort if the RCCL routines detect an error condition (#4) (*most* RCCL and RCI primitives respond to this option; the reference manual should be consulted to check whether or not a particular one does). We repeat that while this option does not have to be set, it can be very convenient. The name of the robot to be controlled is again taken to be the system default robot (#5).

Several transformations are created (#6). `e` is a transform from the robot $\mathrm{T6}$ frame to the tool tip; in this case (as with most of the other examples) it is simply a translation along the z axis of the T6 frame. Transform `b` locates the robot's initial starting position in terms of a displacement from the manipulator base frame to the tool frame. It is used in the position equation `start`, which is defined as

$$\mathbf{T6}\ \ \mathbf{TOOL}\ =\ \mathbf{B}$$

and is created with the call to `makePosition()` at (#7). Four separate transforms `c[k]` are created to locate each of the four corners of the box. Each corner is reached by a translation in the xz plane of the frame defined by `b` and is described by an equation of the form

$$\mathbf{T6}\ \ \mathbf{TOOL}\ =\ \mathbf{B}\ \ \mathbf{C}(k).$$

The corresponding position equations `corners[k]` are created at (#8).

Note that the transforms `b` and `e` are named but all the other transforms and position equations are unnamed. The naming of the first two transforms was done for illustrative purposes only; the names will not actually be used in this program. Note also that four separate transforms and position equations were created to represent each of the box's corners. Again, this was mainly done for illustrative purposes; for fixed targets, one has the option of creating only one position equation and then moving to different points by changing the values of its component transforms between move requests. Later examples will illustrate this.

After the transforms and position equations have been allocated, the program creates a `MANIP` structure for the robot using `rcclCreate()` and turns on the trajectory generator with `rcclStart()` (#9). It should be mentioned that while the examples in this manual will tend to create their transform and position objects before the first call to `rcclStart()`, this is not necessary; these can be created at any time, whether the trajectory generator is running or not.

The first action is to move the robot to the starting position (#10). By default, this will be done in joint mode, using the default speed settings. The robot stops at the start position for 1 second before proceeding, and the program waits for this pause to finish before proceeding (#11).

The program next switches to Cartesian interpolation mode (using `setMod()` (`#12`)) and goes into a loop (`#13`) which causes the robot to trace out the edges of a box using several different velocity and acceleration values. The sizes of the arrays containing the values are computed automatically by the macro `ARRAY_SIZE()`. On each pass through the inner loop, new acceleration and velocity parameters are set using `setCartAccel()` and `setCartVel()`. The rotational parameters are left unchanged by specifying `F_UNDEF` or `NULL` in the appropriate slot of each primitive. To illustrate the usage of the "get" routines, the settings are read back using `getCartAccel()` and `getCartVel()` and printed.

The move commands to trace around the outside of the box are generated by the subroutine `moveSet()` (`#14`); the program waits (at `#15`) for the drawing of each box to finish before doing it again with another set of acceleration and velocity values.

`moveSet()` works by simply queuing up motion requests to a list of positions given as an input array. An alternative way of doing this would be to use a single position equation and an input array of target transform values. For instance, we could define a routine like this:

```
moveTarget (mnp, tool, points, size)
MANIP *mnp;
TRSF_PTR tool, points[];
int size;
{
        POS *p;
        TRSF *target, *tl;
        int k;

        target = allocTrans (NULL, UNDEF);
        tl = allocTrans (NULL, UNDEF);
        *tl = *tool;
        p = makePosition (mnp, T6, tl, EQ, target, TL, tl);
        for (k=0; k<size; k++)
         { *target = *points[k];
           move (mnp, p);
         }
        freePosition (p);
        freeTrans (tl);
        freeTrans (target);
}
```

All that this routine requires the user do is specify a tool transform and an array of target points. The input transforms do not need to be allocated with the `allocTrans()` primitives since they are not actually part of the position equation which is used.

The program example ends by changing back to joint interpolated motions (using `setMod('j')`) and moving the robot back to the start position (`#16`). We do not bother to reset the acceleration or velocity parameters, since `setCartVel()` and `setCartAccel()` affect only Cartesian interpolated motions (the equivalent primitives for joint interpolated motions are `setJointVel()` and setJointAccel()). If one wants to simply scale the robot's speed regardless of the interpolation mode, then setSpeed() can be used.

Note that before the program exits, it waits for the last motion to finish using waitForCompleted(mnp) (#17); otherwise, the program would probably exit, and the robot would be stopped (abruptly!), before it actually reached its last position. The final call to rcclRelease() turns off the trajectory generator, the argument YES indicating that the robot arm power should also be shut of.

4.7 More Motion Parameters

4.7.1 Explicitly Setting Motion Times

The program can override the automatic computation of a motion's transition and duration times (τ and σ) with the primitive

```
setTime (mnp, accelTime, travelTime)
  MANIP *mnp;
  float accelTime, travelTime;
```

accelTime and travelTime are both in milliseconds. accelTime is actually one half the total acceleration time; i.e., it is equal to τ multiplied by the control interval time (see Figure 14). travelTime is equal to σ multiplied by the control interval time. If either argument is specified as F_DEFAULT, then the corresponding time parameter is computed by the trajectory generator, as usual. setTime() affects only the next motion requested.

This function is very useful for motions involving time varying targets (which will be discussed later). Depending on how the target is time varying, the system may not be able to compute the correct segment time since this computation is based on the distance to the target position at the *beginning* of the motion. In such cases the motion time can be specified explicitly. For example, suppose we have a target position spiral which contains a functionally defined transform causing it to track a spiral path. Then the code fragment

```
setMod (mnp, 'c');
setTime (mnp, 150.0, 2000.0);
move (mnp, spiral);
```

would cause the manipulator to perform a spiraling motion for 2 seconds, beginning with a 0.15 second transition time.

If we want to specify the transition time explicitly, then a call like

```
setTime (mnp, 200.0, F_DEFAULT);
```

will do the trick; this forces the transition time to be 0.2 seconds but lets the trajectory generator compute the segment duration as normal, using the currently defined velocities. It is sometimes necessary to specify a transition time of zero. This can be useful for motions terminated on condition where the reaction time is of primary importance.

A common thing to do is set the overall motion time and let the system determine the transition time, as in

```
setTime (mnp, F_DEFAULT, 1000.0);
```

which specifies a total motion duration of 1 second.

Finally, we can specify a motion of indefinite duration by setting the `travelTime` to `F_UNDEF`:

```
setTime (mnp, F_DEFAULT, F_UNDEF);
```

This is useful in cases where the motion of the arm is created by variable transforms within the target position (Section 5.1), and we don't care about reaching the target per se.

4.7.2 Offsetting the Motion Target

It is possible to specify an offset to a motion's target position. This is useful if several poses are being planned in the vicinity of a particular target but we do not wish to create a separate position equation for each pose. The primitives

```
distance (mnp, fmt, v[, v] ...)
  MANIP *mnp;
  char *fmt;
  float v;

setDistance (mnp, offset)
  MANIP *mnp;
  TRSF *offset;
```

both apply an offset to the target position of the next motion request (and the next motion request only). `distance()` allows the offset to be specified as a set of translation values and roll-pitch-yaw angles. It takes a format string listing the offset's components, followed by a variable number of arguments giving the associated values. In the format string, each component is described with two letters. The first letter can be either `d` or `r` (for *distance* or *rotation*) and the second letter can be either `x`, `y`, or `z` (for each of the principal axes). For instance, the specification string `"dx rz"` indicates a two-component offset consisting of a translation along x and a rotation around z. This string would be followed by two arguments giving the corresponding values. When computing the total offset, translation values are incorporated first, followed by the rotations about the z, y, and x axes (the same order used for roll-pitch-yaw angles).

As an example, the code fragment

```
distance (mnp, "dz", -30.);
move (mnp, p);
move (mnp, p);
distance (mnp, "dz", -30.);
move (mnp, p);
```

implements a simple "approach, reach, depart" sort of sequence along the z direction of the tool.

The `distance()` function can be used in conjunction with the `last` position equation in the `MANIP` structure to execute successive *relative* motions:

```
move (mnp, p);
distance (mnp, "dx", 100.0);
move (mnp, mnp->last);
distance (mnp, "dy", 100.0);
move (mnp, mnp->last);
distance (mnp, "dx", -100.0);
move (mnp, mnp->last);
distance (mnp, "dy", -100.0);
move (mnp, mnp->last);
```

This code fragment first moves the robot to the target specified by `p`, and then moves about a square in the xy plane.

The components given to `distance()` are turned into a transformation which is applied to the target position in the TOOL coordinate frame. For instance, if a target position is described by

$$\mathbf{T6}\ \ \mathbf{E}\ =\ \mathbf{C}\ \ \mathbf{P}$$

with `E` defining the TOOL frame, then a call to `distance()` would implicitly insert an offset transform `SHIFT` as follows:

$$\mathbf{T6}\ \ \mathbf{E}\ =\ \mathbf{C}\ \ \mathbf{P}\ \ \mathbf{SHIFT}$$

`setDistance()` behaves the same as `distance()`, but allows the program to specify the offset transform directly; this provides a more "program oriented" interface.

Both `setDistance()` and `distance()` are canceled by a `movej()` request.

4.7.3 Changing the Robot Configuration

Most robot arms are capable of reaching a designated Cartesian position in several ways. A particular instance of these redundancies is known as the robot *configuration.* In particular, for PUMA robots, there are three separate redundancies (right-handed/left-handed, elbow down/up, and wrist flip/noflip), making a total of eight possible ways to achieve any particular position.

When a robot is moving about in Cartesian coordinates, it usually retains its configuration from one pose to the next. The primitive

```
setConf (mnp, conf)
  MANIP *mnp;
  char *conf;
```

sets up an arm configuration change during the subsequent motion. This motion **has** to be performed in *joint* mode, since a configuration change always involves moving through a degenerate arm position unreachable in *Cartesian* mode. Once the configuration change is obtained, the motions can again be performed in *Cartesian* mode. `setConf()` takes a string argument indicating the configuration change. Different configurations are indicated by unique characters within the string. For the PUMA arm, the configurations can be: shoulder right-handed/left-handed (`r`/`l`); elbow down/up (`d`/`u`); wrist flip/noflip (`f`/`n`). For example, if the arm is in a left-handed, up, and noflip configuration (`lun`), then to change the wrist configuration to flip, we can write

```
/* the arm is currently "lun" */

setMod (mnp, 'j');    /* go in joint mode if it wasn't */
setConf (mnp, "luf"); /* specify flip */
move (mnp, new);      /* go "luf" */
```

The configuration string does not have to be complete; it is permissible to specify only those aspects of the configuration which are to desired to be changed. For instance, in the example above, the call `setConf (mnp, "f")` could have been used instead.

`setConf()` is ignored by stop requests and handled only by the next move request. Also, if `setConf()` is followed by a `movej()` request, then the `setConf()` request will be "over-ruled" by whatever configuration is implicit in the `movej()`.

The current manipulator configuration can be read back with

```
getActiveConf (mnp, conf)
  MANIP *mnp;
  char *conf;
```

which reads the robot's present configuration into the string `conf`.

Robot configurations can also be described by a bitmask; for instance, the kinematic routines (Section 6.2.1) use this representation. The configuration codes for the PUMA are defined in the file `<puma_kynvar.h>`:

```
PUMA_RIGHT_CONF -- right handed ('r')
PUMA_DOWN_CONF  -- elbow down ('d')
PUMA_FLIP_CONF --  wrist flip ('f')
```

`setConf()` returns a bitmask indicating which configuration bits were specified. `getActiveConf()` returns a bitmask equivalent to the configuration string.

For a given robot, the routines

```
configToStr (str, mask, kyn)
  char *str;
  int mask;
  KYN *kyn;

strToConfig (maskp, str, kyn)
  int *maskp;
  char *str;
  KYN *kyn;
```

convert between the string and bit representations for configuration. `kyn` is a pointer to the robot's `KYN` structure, which is described in Section 6.1.1. Details on these routines can be found in the reference manual.

The configuration representation used by RCCL should probably be improved. For instance, there should be ways of indicating joint redundancies and singularities (configurations are undefined at singularities). This may be done when RCCL is upgraded to handle redundant manipulators.

4.8 Synchronization

As has been mentioned, motions are requested asynchronously to their execution; a motion request is simply placed in a queue for servicing by the trajectory generator at the next possible opportunity. This makes it necessary to provide ways to explicitly coordinate the planning level with the trajectory generator.

The most straightforward thing we can do is simply wait for the trajectory generator to finish handling all the motion requests in its queue. This can be done using the macro `waitForCompleted()` that was discussed earlier. The code construction

```
move (mnp, p1);
move (mnp, p2);
move (mnp, p3);

waitForCompleted (mnp);
```

will quickly queue three motion requests and then wait for them all to complete. `waitForCompleted()` actually blocks until the transition *out of* the last motion has finished, so the manipulator is guaranteed to be stationary when it returns (unless `TRACKING_MODE` has been selected).

Alternatively, we might want to wait for an individual motion request to finish. To do this, we need to have a handle on the motion request. Such a handle is returned by the motion primitives: all calls to `move()`, `movej()`, or `stop()` return a *motion ID*, an integer, that can be used to query information about the motion. In particular, the following primitives are useful:

```
motionStatus (mid)
  int mid;

float motionScale (mid)
  int mid;

motionStartCode (mid)
  int mid;

motionStopCode (mid)
  int mid;
```

`motionStatus()` returns either `M_PENDING`, `M_RUNNING`, `M_PAUSED`, or `M_FINISHED` depending on whether the indicated motion is queued, is executing, has been paused (see section pause), or has completed.

`motionScale()` returns a scalar value indicating how far along the motion has progressed in its execution. The scale value is 0.0 at the midpoint of the transition **into** the motion and 1.0 at the midpoint of the transition **out of** the motion, unless the transition was canceled prematurely, in which case the final scale value will be less than 1.0. Because the system starts computing the scale value at the beginning of the transition into the motion, it will be slightly negative during the period between the beginning and midpoint of the initial transition.

RCCL associates a code value with the start and completion of a motion. The default code value is `ON_NORMAL`, which is defined in `<rccl.h>`. Other code values can be specified with the functions `startMotion()` and `stopMotion()` (see below).

The primitives

```
waitForStart (mid)
  int mid;

waitForStop (mid)
  int mid;
```

respectively poll the indicated motion's status and block until the motion has started executing or has finished executing. For instance, the following code patterns are equivalent:

```
id = move (mnp, p0);
waitForStop (id);
id = move (mnp, p1);
waitForStop (id);
```

or

```
move (mnp, p0);
waitForCompleted (mnp);
move (mnp, p1);
waitForCompleted (mnp);
```

To further demonstrate the use of these primitives, consider the following (somewhat contrived) example:

```
MANIP *mnp;
FILE *fp;
POS_PTR posA, posB, posC;
int id0, id1;

id0 = move (mnp, posA);
id1 = move (mnp, posB);
move (mnp, posC);
```

```
waitForStop (id0);

while (motionStatus (id1) == M_RUNNING)
 { fprintf (fp, "scale = %f\n", motionScale (id1));
   fprintTrsf (fp, "%m\n", mnp->t6);
 }
```

This program queues three motion requests, waits for the first one to complete, and then begins to record the successive **T6** values (along with the motion scale value) for the second motion in the file `fp`.

As another example, consider the "approach, reach, depart" sequence described earlier. If we want to coordinate the planning level of the program so that it does something (such as open a gripper at the approach point and close it at the reach point), then we can use a code segment that looks something like this:

```
int approachId, reachId;

distance (mnp, "dz", -30.);
approachId = move (mnp, p);
reachId = move (mnp, p);
distance (mnp, "dz", -30.);
move(p);

waitForStop (approachId);
OPEN_HAND (mnp);
waitForStop (reachId);
CLOSE_HAND (mnp);
```

The motion IDs for the moves to the approach and reach positions are recorded in `approachID` and `reachId`. The program waits until the manipulator arrives at the *approach* position, opens the gripper, waits again for the manipulator to arrive at the reach position, and closes the gripper.

The above example introduces the macros

```
OPEN_HAND(mnp)
  MANIP *mnp;

CLOSE_HAND(mnp)
  MANIP *mnp;
```

which open and close the gripper for the specified manipulator (this hand control interface is quite rough and could use some improvement).

To wait for an arbitrary condition or delay for a specific period of time, the following primitives are available:

```
waitFor(exp)
```

```
waitWhile(exp)

delay(msec)
  float msec;
```

The first two are macros: `waitFor(exp)` waits for the indicated expression to return true, and `waitWhile(exp)` waits for the indicated expression to return false. `delay()` puts the program to sleep for the indicated number of milliseconds; it can be called only from the planning level.

4.8.1 Canceling and Controlling Motions

The functions

```
stopMotion (mid, code)
  int mid, code;

startMotion (mid, code)
  int mid, code;

setMotionHold (mnp)
  MANIP *mnp;
```

may be used to control individual motion requests.

`stopMotion()` cancels the indicated motion and sets its motion stop value to `code`.

`startMotion()` gives the indicated motion permission to begin and gives it a start value equal to `code`. This may seem a bit obscure, since normally all motions are queued with permission to begin and are given a start value of `ON_NORMAL`. However, calling `setMotionHold(mnp)` before a motion request is issued will remove its permission to begin, so that when it reaches the head of motion queue, and thus would otherwise be ready to run, it will not do so until permission is granted by a call to `startMotion()`. This is mainly useful for starting motions from the control level (which is not permitted to issue actual motion requests).

We now consider the motion ID values. While a motion ID which is automatically allocated is always unique, only its low order bits are used to point to the internal cell containing the motion's status. Because memory space is limited, the number of these cells is also limited, and the status cell associated with a motion ID will be re-used every `NUM_SYSMFLAGS` motions. This number is large enough to accommodate most applications, but in case the programmer desires to keep motion status information around indefinitely, the function

```
setMotionFlag (mnp, mid)
  MANIP *mnp;
  int mid;
```

can be used to explicitly set the motion ID of the next motion to be `mid`. This value must be between 0 and `NUM_USRMFLAGS`-1. The use of these motion IDs is strictly under application control; RCCL will never automatically assign a motion ID in this range.

To illustrate some of this, consider another example involving two robots:

```
#define FLAG1   1
#define FLAG2   2

MANIP_PTR mnp1, mnp2;
POS_PTR posA, posB;

setMotionFlag (FLAG1);
move (mnp1, posA);

setMotionFlag (FLAG2);
move (mnp2, posB);

while (motionScale (FLAG1) < 0.5)
   ;

stopMotion (FLAG2, 0);
```

The idea here is that we have two robots, respectively controlled via `MANIP` structures pointed to by `mnp1` and `mnp2`. We start the first robot moving toward `posA` with `FLAG1` explicitly set as the motion ID (note that we do not store the ID returned by the motion request since we know what it is). The second robot is then started moving towards `posB` with the motion ID `FLAG2`. We then monitor the scale value of the first motion, and when it is more than half completed, cancel the motion on the second robot with a code value of 0.

4.8.2 Controlling the Current Motion and Motion Queue

The primitive

```
stopCurrentMotion (mnp, code)
  MANIP *mnp;
  int code;
```

may be used to cancel whatever motion happens to be executing currently, regardless of its motion ID.

A few functions are available for controlling the motion queue:

```
flushMotionQueue (mnp)
  MANIP *mnp;

numMotionsQueued (mnp)
  MANIP *mnp;

checkMotionQueue (mnp, pos)
  MANIP *mnp;
  POS *pos;
```

```
setMotionQueueSize (mnp, size)
  MANIP *mnp;
  int size;

getMotionQueueSize (mnp)
  MANIP *mnp;
```

`flushMotionQueue()` cancels all pending motions on the motion queue. It will not, however, abort the current motion, so the proper way to "kill everything" is with a code fragment like this:

```
flushMotionQueue (mnp);
stopCurrentMotion (mnp, STOPCODE);
```

The motion queue is flushed first so that when the current motion is aborted, the system will not take another off the queue and start executing it.

`numMotionsQueued()` returns the number of motions pending on the motion queue, and `checkMotionQueue()` returns true if there is enough space on the queue to handle a motion request to the target position `pos`.

The maximum size of the motion queue can be set with `setMotionQueueSize()` and read back with `getMotionQueueSize()`. Setting the maximum queue size can be a useful synchronization tool in that different queue sizes will produce different blocking behaviors for the motion requests. Setting the motion queue size to 0 will cause every motion request to block until the associated motion has been completed (equivalent to doing a `waitForCompleted()` after every motion request). Setting the motion queue size to 1 will cause every motion request to block until its motion is being executed. A general motion queue size of n will cause the move primitives to block until there are only $n - 1$ motions still pending on the queue. Consider the following example:

```
MANIP *mnp;
POS_PTR posA, posB, posC, posD, posE;

setMotionQueueSize (mnp, 0);
move (mnp, posA);

/* nothing on queue; move to posA is now complete */

setMotionQueueSize (mnp, 1);
move (mnp, posB);

/* nothing on queue; move to posB has at least started */

setMotionQueueSize (mnp, 3);
move (mnp, posC);
move (mnp, posD);
move (mnp, posE);

/* no more than 2 motions on queue; */
```

```
/* move to posC has at least started */
```

We first set the motion queue size to 0, causing `move(mnp, posA)` to block until completion. The queue size is then set to 1, which means that the call `move (mnp, posB)` will block until the motion is at least being executed. Setting the queue size to 3 means that subsequent move requests will block until there are no more than 2 motions on the queue; this means that `move (mnp, posE)` must block until the move to position `posC` is at least being executed.

The motion queue can be set as large as `MAX_MQ_ENTRIES`. The default queue size can be obtained by calling `getMotionQueueSize()` immediately after the `MANIP` structure has been created.

As another example of how the motion request queue interacts with the planning level, consider the following program:

```
TRSF_PTR z, e , b;
POS_PTR p;
float iz;

z = allocTransXyz ("Z",  0., 0., 864.);
e = allocTransXyz ("E" , 0., 0., 170.);
b = allocTransRot ("B", 600., 128., 800., yunit, 180.);

p = makePosition ("P", z, t6, e, EQ, b, TL, e);

while (1)
 { printf ("Enter Z increment> ");
   scanf ("%f", &iz);
   b->p.z += iz;
   move (mnp, p);
 }
```

This allows the user to continuously adjust the z position of the robot. Every time an increment for z is entered, a new motion request to handle that increment will be queued, using the current value of the `b` transform.

If the user enters data faster than the manipulator can move to the goal positions, several motion requests will be queued up. If the user stops entering data, all the requests will eventually be executed, the manipulator will be brought to rest, and the program will block at the `scanf()` call. If the data is provided by continuous stream `fp`, then the same program could look something like:

```
while (fread (b, sizeof(TRSF), 1, fp) != NULL);
 { move (mnp, p);
 }
```

4.8.3 Getting UNIX signals on motion completion

A useful feature of RCCL is the ability to request a UNIX signal to be delivered whenever (1) all requested motions complete or (2) a particular motion completes (these are the same conditions which cause `waitForCompleted()` or `waitForStop()` to return true). The routines to set up the signals are

```
setCompletedSig (mnp, signum)
  MANIP *mnp;
  int signum;

setStopSig (mnp, signum, mid)
  MANIP *mnp;
  int signum, mid;
```

`setCompletedSig()` requests that the signal `signum` be sent to the planning task the next time the trajectory generator finishes executing all the motions on the queue. `setStopSig()` requests that the signal `signum` be sent to the planning level task whenever the motion specified by the ID `mid` finishes.

Consider the following example:

```
#include <rccl.h>
#include <signal.h>

handler()
{
  printf ("All motions completed\n");
}

robottask()
{
  ... initializations and other things ...

  signal (SIGUSR1, handler);

  setCompletedSig (mnp, SIGUSR1);

  move (mnp, p0);
  move (mnp, p1);
  move (mnp, p2);

  printf ("Hit <CR> to quit:\n");
  getchar();
}
```

This program will quickly queue three motion requests, and then block waiting for a character. When all three motions have completed, the message `All motions completed` will be asynchronously printed to the screen.

4.9 Program Example: "hex"

This program has the manipulator trace out a small hexagonal grid, or "honeycomb" pattern (see Figure 17). The robot does one column in the grid at a time, and before doing each one goes to a "pickup" position, as though it were reloading a tool (see Figure 18). When each column is completed, a signal is sent to the planning level that causes a message to be printed.

This program is a bit complex for example purposes, but it does illustrate a wide variety of capabilities.

```
#include <rccl.h>
#include <signal.h>                                                 /*#1*/
#include <math.h>
#include "manex.560.h"

#define FLAG    2

onSignal()                                                          /*#2*/
{
        printf ("Column done\n");
}

main()
{
        TRSF_PTR e, park, pickup, top, down;
        POS_PTR home, reload, center;
        MANIP *mnp;
        JNTS rcclpark;
        char *robotName;

        float hexsize = 15.0;
        int numRows = 5;
        int numCols = 5;
        int i, j;

        float sin_60;

        int mid;

        rcclSetOptions (RCCL_ERROR_EXIT);
        robotName = getDefaultRobot();                              /*#3*/
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }

        e = allocTransXyz ("E", UNDEF, 0.0, 0.0, TOOLZ);
        park = allocTrans ("PARK", UNDEF);                          /*#4*/
        pickup = allocTrans ("PICKUP", UNDEF);
        top = allocTrans ("TOP", UNDEF);
        down = allocTrans ("DOWN", UNDEF);
```

```
home = makePosition ("home", T6, e, EQ, park, TL, e);    /*#5*/
reload = makePosition ("reload", T6, e, EQ, pickup, TL, e);
center = makePosition ("center", T6, e, EQ, park, top, down, TL, e);

mnp = rcclCreate (robotName, 0);                         /*#6*/
rcclStart();

movej (mnp, &rcclpark);                                  /*#7*/
waitForCompleted (mnp);

solveTrans (park, home, park, mnp->here);                /*#8*/

rotToTrsf (pickup, zunit, 90.0);                         /*#9*/
multTrsf (pickup, pickup, park);
multTrsfXyz (pickup, 0.0, 0.0, -125.0);
multTrsfRot (pickup, xunit, 30.0);

xyzToTrsf (top, 100.0, 0.0, 50.0);                       /*#10*/
multTrsfRot (top, xunit, 90.0);

setJvelScale (mnp, 2.0);                                 /*#11*/
setCartVel (mnp, 100.0, 50.0);
signal (SIGUSR1, onSignal);

sin_60 = sin(60.0*DEGTORAD);                             /*#12*/

for (j=0; j<numCols; j++)                                /*#13*/
 { setConf (mnp, "n");                                   /*#14*/

   distance (mnp, "dz", -50.0);                          /*#15*/
   move (mnp, reload);
   mid = move (mnp, reload);
   stop (mnp, 2000.0);
   distance (mnp, "dz", -50.0);
   move (mnp, reload);

   waitForStop (mid);                                    /*#16*/
   OPEN_HAND (mnp);
   delay (500.0);
   CLOSE_HAND (mnp);

   setConf (mnp, "f");                                   /*#17*/

   top->p.x -= 2*hexsize*sin_60;
   identTrsf (down);                                     /*#18*/
   for (i=0; i<numRows-1; i++)                           /*#19*/
    { traceHex (mnp, center, hexsize, 50.0, UNDEF);
      if (i%2 == 0)                                      /*#20*/
       { multTrsfXyz (down, hexsize*sin_60, 3/2.*hexsize, 0.0);
       }
      else
       { multTrsfXyz (down, -hexsize*sin_60, 3/2.*hexsize, 0.0);
       }
```

```
            }
           traceHex (mnp, center, hexsize, 50.0, FLAG);
           setStopSig (mnp, SIGUSR1, FLAG);                      /*#21*/
         }

        setConf (mnp, "n");                                      /*#22*/
        move (mnp, home);
        stop (mnp, 1000.0);
        waitForCompleted (mnp);
        rcclRelease (YES);
}

traceHex (mnp, p, hexsize, approach, flag)
MANIP *mnp;
POS *p;
float hexsize, approach;
int flag;
{
        TRSF offset;
        int saveMod;
        int mid;
        int theta;

        xyzToTrsf (identTrsf(&offset), 0.0, 0.0, -approach);
        setDistance (mnp, &offset);
        move (mnp, p);                                           /*#23*/

        saveMod = getMod(mnp);                                   /*#24*/
        setMod (mnp, 'c');                                       /*#25*/

        offset.p.z = 0.0;                                        /*#26*/
        for (theta=30; theta<=390; theta+=60)
         { offset.p.x = hexsize * cos(DEGTORAD*theta);
           offset.p.y = hexsize * sin(DEGTORAD*theta);
           setDistance (mnp, &offset);
           move (mnp, p);                                        /*#27*/
         }
        xyzToTrsf (identTrsf(&offset), 0.0, 0.0, -approach);
        setDistance (mnp, &offset);
        if (flag != UNDEF)
         { setMotionFlag (mnp, flag);                            /*#28*/
         }
        mid = move (mnp, p);                                     /*#29*/
        setMod (mnp, saveMod);                                   /*#30*/
        return (mid);
}
```

NOTE – this example has been coded for the PUMA 560 robot, and lives at `demo.rccl/hex.560.c`. *An equivalent program for the PUMA 260 is contained in* `hex.260.c`

The file `<rccl.h>` is included as usual. `<signal.h>` and `<math.h>` are UNIX files included to get the definition of the signal `SIGUSR1` and define the types for routines `sin()` and `cos()` (`#1`).

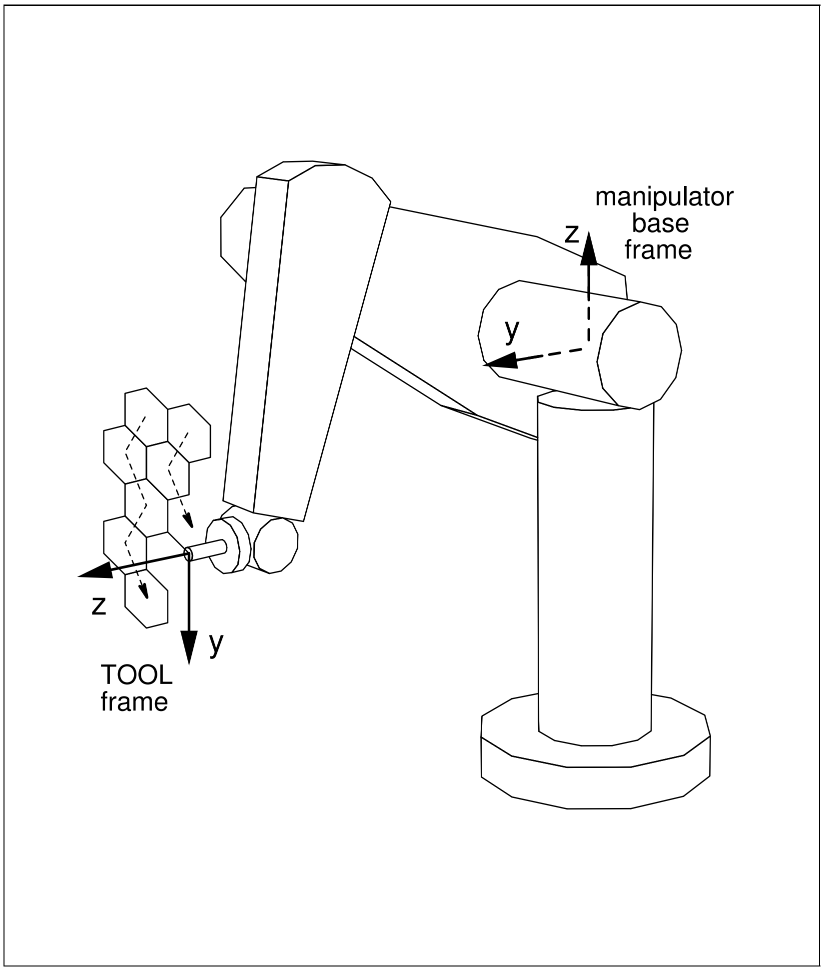

Figure 17: Robot drawing out a hexagonal “honeycomb” pattern.

The signal handler which will be used to print the column completion message is defined at (`#2`).

The program declares a few variables and pointers, gets the name of the robot to be controlled and its starting position (`#3`), then allocates several transforms and position equations (`#4` and `#5`). Except for the end-effector transform `e`, all of the transforms will be initialized later in the program. The position `start` describes the position reached after moving to the joint angles `rcclpark`, and is defined simply by

$$\mathbf{T6} = \mathbf{PARK}$$

`reload` is the place the robot moves to before tracing out each column, and is defined by the equation

$$\mathbf{T6\ \ E} = \mathbf{PICKUP}$$

where **E** describes the manipulator’s end-effector transform. The most complicated position is `center`, defined by

$$\mathbf{T6\ \ E} = \mathbf{PARK\ \ TOP\ \ DOWN}$$

which specifies the center point of each hexagon the robot traces out. **TOP** locates the top of each column with respect to the initial frame defined by **PARK**. The transform **DOWN** describes, with

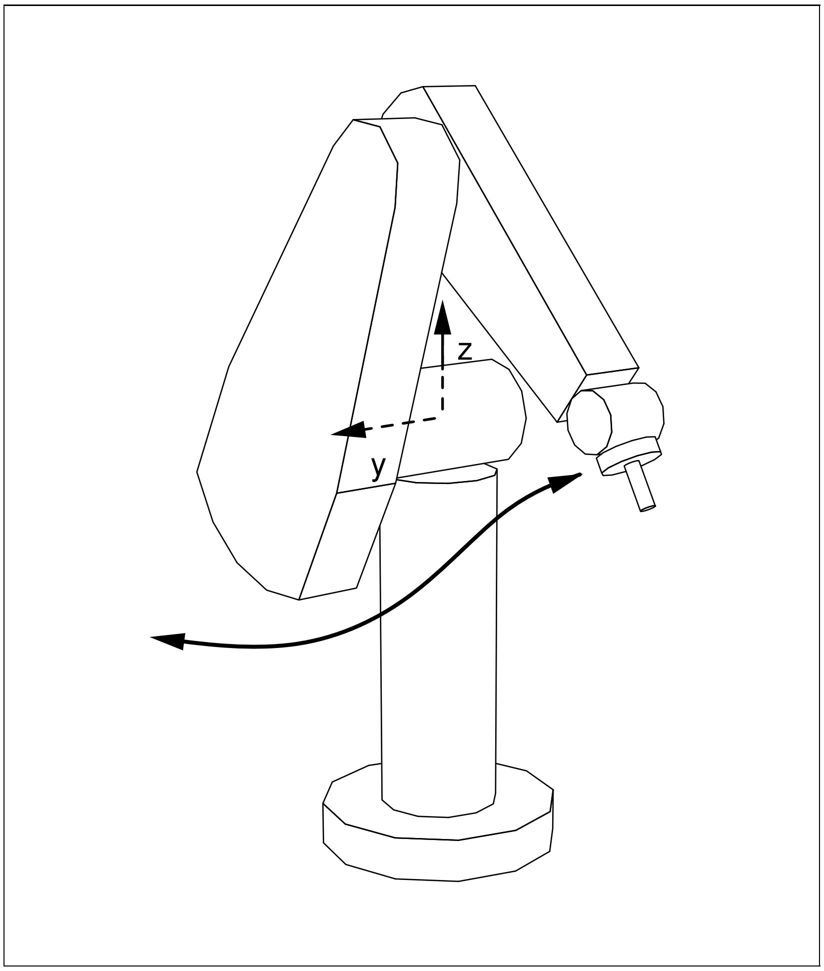

Figure 18: Position the robot moves to in between drawing each column of the honeycomb.

respect to the frame TOP, a translation in the xy plane from the top of the column down to the center of a particular hexagon. As the program traces out the different hexagons, it locates the center of each one by changing the values for `top` and `down` appropriately.

The program begins in the usual fashion by setting up the manipulator, starting the trajectory generator, and moving the robot to the initial angles described by `rcclpark` (`#6` and `#7`). At `#8` the program initializes `park` to the current value of **T6** by reading the `here` field of the `MANIP` structure. `pickup` is then defined to be a position "off to the side" of `park`; specifically, `park` rotated by $90°$ in world coordinates about the z axis of the robot's base, followed by a $30°$ rotation about the local x axis (`#9`) (see Figure 18). `top` is initialized to describe the top left-hand corner of the hex grid. Specifically, this involves a translation in the xz plane followed by a rotation of $90°$ about the x axis so that the robot's tool tip points horizontally at the grid (`#10`) (see Figure 17). Because of this last rotation, the xy plane of the frame TOP is parallel to the xz plane of the initial frame PARK. At `#11` the program sets the speed for joint interpolated motions to be double that of Cartesian interpolated motions, sets the Cartesian velocities explicitly, and sets up the signal handler for the end-of-column signal. The variable `sin_60` initialized at `#12` is introduced to make the program more readable.

A loop begins at `#13` that causes the robot to trace out each of the `numCols` columns in the grid. The robot first moves to the `reload` position, where it does an approach-move-stop-depart sequence `#15`. The program synchronizes with the arrival of the robot at reload by waiting on the motion ID `rid` (`#16`) after which it opens the robot's hand, waits half a second, and then closes the hand.

The robot then moves to the top of the next column in the grid. Since it is better able to reach the grid with its wrist in the *flip* configuration, this configuration is explicitly requested (and later unrequested) with calls to `setConf()` and `getConf()` (`#14` and `#17`). Each column is set adjacent to the previous one by shifting the x coordinate of `top` by $2\sin(60°)$`hexsize`, where `hexsize` is the length of each hexagon's side. `down`, which is used to specify the center of each hexagon relative to the top of the column, is initially cleared (`#18`). The subloop at `#19` uses the routine `traceHex()` (discussed below) to create the hexagons. For each loop iteration, the hexagon's center is located by increasing the y coordinate of `down` by $3/2$`hexsize` and shifting the x coordinate either to the right or left by $\sin(60°)$`hexsize` (`#19`). The tracing of a single column is illustrated by Figure 19.

The routine `traceHex()` generates the motion requests necessary to trace out a hexagon in the xy plane of a position `p`. All the motions are specified relative to the center `p` using an offset transform (`offset`) and the routine `setDistance()`. It begins by issuing a move to an approach position located `approach` millimeters away from the center point along the z axis (`#23`). `setMod()` is then used to ensure that the remaining motions are done in Cartesian mode (`#25`). To improve the modularity of `traceHex()`, no assumptions are made about the previous interpolation mode. This means that the mode should be "saved and restored", which is done with calls to `getMod()` and `setMod()` (`#24`, `#30`). After the approach move is requested, the z offset is removed (`#26`) and a set of moves are generated to take the robot around the corners of the hexagon. Each corner point is determined by computing an appropriate X, Y offset from the hexagon center as a function of an angle `theta` (`#27`). The actual trajectory will be slightly "rounded out" because of path transitions, and the corner points will be correspondingly undercut, unless the robot is slowed down enough to permit a zero transition time to be specified. Lastly, the robot is moved back to the approach position (`#29`). The ID returned by this last motion is saved and returned to the caller. If the routine argument `flag` has been set to a value other than `UNDEF`, then it is used as the motion ID for the last motion. This is useful in case the caller needs to know the motion ID `before` `traceHex()` is called.

4.10 Program Example: "jmove"

This example shows how to create a simple program to move the robot around in joint coordinates and change its configuration. For simplicity, it does not attempt to deal with prismatic joints. The program is interactive and illustrates the use of the "C-tree matcher", an automatic command completion keyboard interface used by many programs in the RCCL/RCI system.

```
#include <rccl.h>
#include <ctree.h>                                              /*#1*/

char key_buf[256];
```

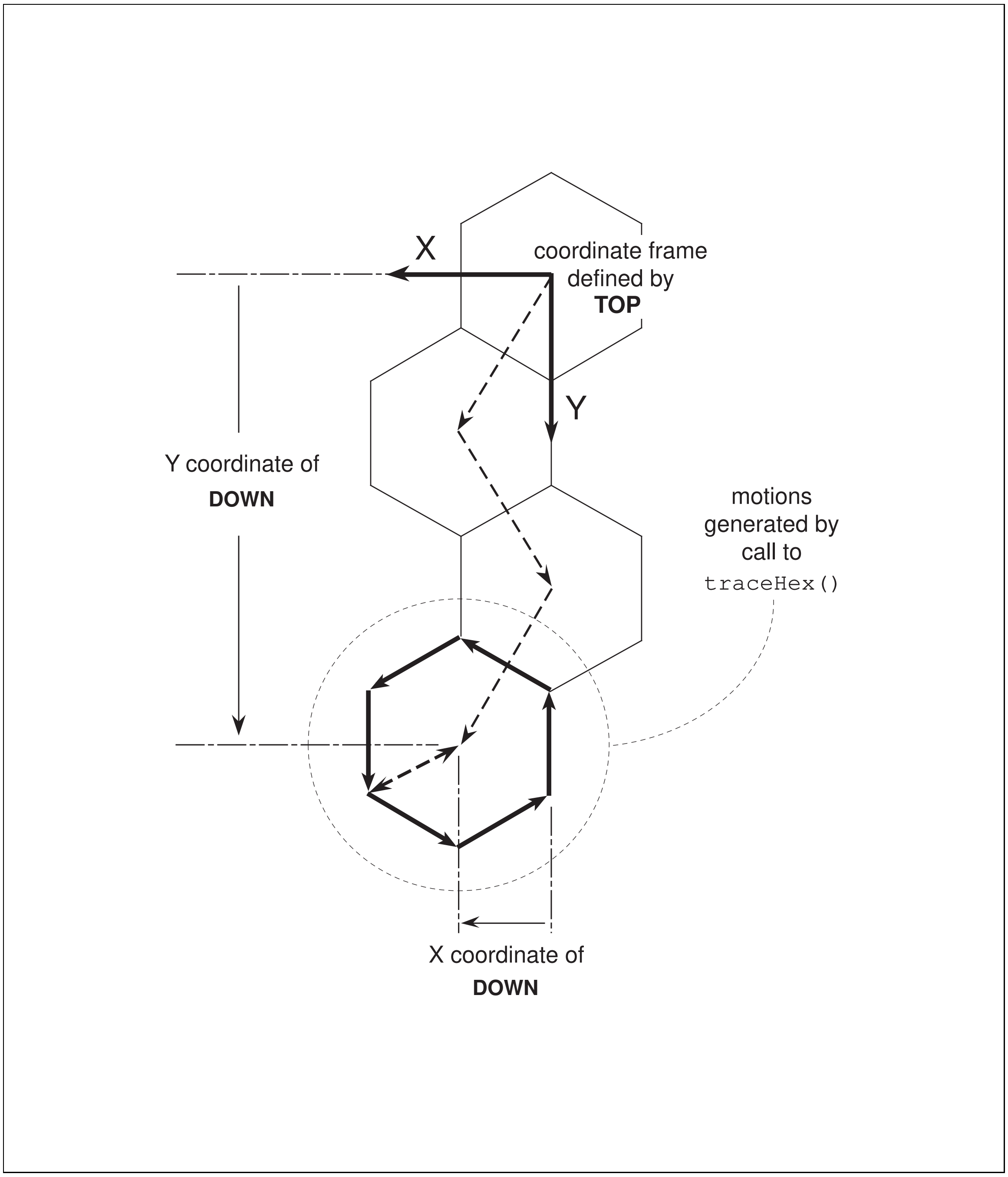

Figure 19: Path followed by the robot as it traces out one column in the hex grid, projected onto the xy plane of the frame TOP. The rounding due to transitions is ignored.

```
main(argc, argv)
int argc; char **argv;
{
        MANIP *mnp;
        JNTS angles;
        char *robotName;
        char *keytree;

        int movejRequest = 0;
        int jnt;
        int mid = UNDEF;
        int newMid = UNDEF;

        float angval;
        float speed;
        char conf[4];

        rcclSetOptions (RCCL_ERROR_EXIT);
        if (argc != 2)                                          /*#2*/
         { fprintf (stderr, "USAGE: %s <robotName>\n", argv[0]);
           exit (-1);
         }
        robotName = *++argv;
                                                                /*#3*/
        keytree = tree_match_parse("                                    \
(       quit                                                            \
        show                                                            \
        speed                                                           \
         (%f", &speed, ",0,,            \"speed scale\")                \
        move                                                            \
         (all                                                           \
          %d", &jnt, ",1,6,             \"joint number\"                \
           ([by to]                                                     \
             (%f", &angval, ",,,        \"angle value (degrees)\"))     \
          )                                                             \
        setConf                                                         \
         (%s", conf, ",                 \"configuaration string\")      \
)");
        if (keytree == 0)                                       /*#4*/
         { printf ("ERROR/Parse error at %.30s\n", tree_match_err_at());
           exit (-1);
         }

        mnp = rcclCreate (robotName, 0);                        /*#5*/
        rcclStart();

        angles = *mnp->j6;                                      /*#6*/

        do
         { tree_match (keytree, "JMOVE> ", key_buf);            /*#7*/

           if (COMMAND("show"))                                 /*#8*/
            { float deg[6];
```

```
        getActiveConf (mnp, conf);
        printf ("speed: %g\n", getSpeed(mnp));
        printf ("conf: %s\n\n", conf);
        printTrsf ("T6:    %m\n\n", mnp->t6);
        scaleVf (deg, RADTODEG, mnp->j6->v, 6);
        printVf ("J6:    %8.3f\n", deg, 6);
      }
    else if (COMMAND("speed"))                          /*#9*/
      { setSpeed (mnp, speed);
      }
    else if (COMMAND("move all"))                       /*#10*/
      { float deg[6];
        printf ("Joint values> ");
        if (scanVf (deg, 6) != 1)
         { printf ("Error reading values\n");
           continue;
         }
        scaleVf (angles.v, DEGTORAD, deg, 6);
        movejRequest = 1;                               /*#11*/
      }
    else if (COMMAND("move by"))                        /*#12*/
      { angles.v[jnt-1] = DEGTORAD*angval + mnp->j6->v[jnt-1];
        movejRequest = 1;
      }
    else if (COMMAND("move to"))                        /*#13*/
      { angles.v[jnt-1] = DEGTORAD*angval;
        movejRequest = 1;
      }
    else if (COMMAND("setConf"))                        /*#14*/
      { stopCurrentMotion (mnp, 0);
        waitForCompleted (mnp);                         /*#15*/

        setConf (mnp, conf);                            /*#16*/
        move (mnp, mnp->last);                          /*#17*/
        waitForCompleted (mnp);
        angles = *(mnp->j6);                            /*#18*/
      }
    if (movejRequest)                                   /*#19*/
      {
        if ((newMid = movej (mnp, &angles)) < 0)
         { printErrors();                               /*#20*/
           clearErrors();
         }
        else
         { if (mid != UNDEF && motionStatus(mid) != M_FINISHED)
            { stopMotion(mid, 0);                       /*#21*/
            }
           mid = newMid;
         }
        movejRequest = 0;
      }
}
```

```
        while (!(COMMAND("quit")));                                 /*#22*/

        stopCurrentMotion (mnp, 0);
        waitForCompleted (mnp);
        rcclRelease (YES);
}
```

NOTE – this example has been coded for the PUMA robots, and lives at `demo.rccl/jmove.c`.

The most complicated thing to explain about this program is the interface code for the "C-tree matcher". We feel this is justified because this interface is ubiquitous throughout the RCCL/RCI system. To avoid learning about the C-tree matcher, just skip over the next few paragraphs.

The C-tree matcher is a command completion input routine. When called, it presents the program operator with a prompt and allows him/her to type in only a selected set of keywords or values. Since the matcher knows what inputs are valid, it can assist the user by providing a menu of what is legal to type (if the user hits '`?`') or by expanding input to the next branch point (if the user hits `<space>`). The matcher returns when a valid sequence has been entered and the user hits `<return>`. It returns to the program the string of keywords which the user typed, plus any input values it collected. While this sounds cumbersome, in practice it is not. Full documentation on the C-tree matcher, for both the end user and the programmer, is available in `doc/CtreeMatch.doc`.

The program begins as usual by including `<rccl.h>` and declaring miscellaneous variables and pointers. Definitions relevant to the C-tree matcher are contained in the include file `<ctree.h>` (`#1`). The name of the robot to be controlled is obtained, in this program, from the command line rather than from the system routine `getDefaultRobot()` (`#2`).

The first main thing the program does is describe to the C-tree matcher the set of permitted inputs. This is done using a tree of keywords and input values, specified by a long string argument given to the routine `tree_match_parse()` (`#3`). The inputs allowed by this program are

```
quit
show
speed <speedValue>
move all
move <jointNumber> by <angleValue>
move <jointNumber> to <angleValue>
setConf <configString>
```

Successive values or keywords are indicated within the specification string by parentheses:

```
<keyword or value> '(' <following keywords or values> ')'
```

Whenever the input requires actual values, the C-tree matcher will parse the value according to a prescribed format and place it in a variable whose address has been given to `tree_match_parse()`. Allowed formats are `%d` (integer), `%f` (float), `%s` (string), and `%x` (hex integer). The address through which the value should be stored is placed in the argument list to `tree_match_parse()` by breaking the specification string, inserting the address, and restarting the specification string:

```
string spec ... %d", &integerVal, " ... more string spec
```

Integer and float specifications are followed by (optional) lower and upper bounds, and all value specifications may also be followed by a help prompt (contained within `\" \"`) that appears when the user hits '`?`'.

`tree_match_parse()` digests the input specification and returns a handle (`keytree`) that is later used by `tree_match()`, which does the actual reading of input. If an error was found in the input specification, `tree_match_parse()` returns 0, and a string showing the neighborhood where the error occurred can be obtained with the routine `tree_match_err_at()`. (`#4`).

Now back to RCCL. After the program has initialized the C-tree matcher, it calls `rcclCreate()` and `rcclStart()` to set up the trajectory generator and start it (`#5`). It then initializes the variable `angles` to the current values of manipulator joint angles, which are stored in the `j6` field of the `MANIP` structure (`#6`). A loop is then entered which calls `tree_match()` (`#7`) to solicit commands from the user's terminal, until the final `quit` command is received (`#22`).

Commands returned by `tree_match()` consist of sequences of keywords and are placed in the buffer `key_buf`. The macro `COMMAND(str)` does a `strcmp()` between `str` and the contents of `key_buf` to determine what command was entered. The commands that may be returned to this program are

```
quit
show
speed
move all
move by
move to
setConf
```

The `show` command (`#8`) causes the program to print out the robot's current speed setting, configuration, **T6** transform value, and joint angles. Note that the joint angles are converted to degrees before being printed out; all joints are assumed to be revolute as mentioned earlier.

The `speed` command (`#9`) sets the robot's speed parameter, scaling the speed at which the robot responds to further motion commands. The variable `speed` is set during command input by `tree_match()`.

The `move all` (`#10`) command is used to move the robot to a particular set of joint values. It prompts the user for a set of joint angles, reads these in from the keyboard, converts them to radians, and requests a call to `movej()` by setting the variable `movejRequest` (`#11`).

The `move by` command (`#12`) moves one joint by a specified number of degrees. The joint number and the amount to move it by are read into the variables `jnt` and `angval` by `tree_match()` when the command is entered. The appropriate field in the joint target variable `angles` is changed, and a call to `movej()` is requested by setting `movejRequest`.

The `move to` command (`#13`) is similar to the `move by` command except that it moves a joint **to** a specified value.

All of the `move` commands change the joint target variable `angles` and bump the variable `movejRequest`, which is examined at the bottom of the loop (`#19`). If set, the program uses `movej()` to queue a motion request to `angles`. The value returned by `movej()` is stored in `newMid`. If this value is negative, then `movej()` is known to have failed and the appropriate error message(s) are printed and the error stack is cleared with a call to `printErrors()` and `clearErrors()` (#20). (see Section 9.2.1 for a discussion of the error stack). Otherwise, the request succeeded and `newMid` contains its synchronization flag. The program next checks to see if there has been any previous call to `movej()` (`mid != UNDEF`), and if so, if the resulting motion has completed yet. If it has not, the previous motion is aborted with a call to `stopMotion()` using the old motion's ID value (#21). Doing this **after** the call the `movej()` allows for a smooth transition from any previous motion to the new motion. It also provides a nice illustration of the use of the synchronization primitives.

The last command in `jmove` is `setConf` (#14), which is used to change the robot's current kinematic configuration. It first uses `stopCurrentMotion()` and `waitForCompleted()` to stop any current manipulator motion (#15). A string describing the new desired configuration is read into `conf` by `tree_match()` when the command is entered, and is used directly as an argument to `setConf()` (#16). A motion is then requested to the robot's current position (#17), using the the position equation `mnp->last` built in to the `MANIP` structure. Internally, this will cause the trajectory generator to compute the target joint angles for the motion using the new configuration and generate a joint interpolated motion to these new angles. The program waits for the motion to finish with another call to `waitForCompleted()` and then updates the value of `angles` (#18).

5. Moving to Variable Targets

5.1 Transform Bindings

The transforms which make up a position equation are objects which describe positional relationships in the environment. Since these relationships may change, it should be possible to provide ways for changing the values of the associated transforms.

By default, an RCCL transform allocated using one of the `allocTrans()` routines is assumed to be *constant*. That means that whenever that transform is used in a motion request (as part of a position equation) the trajectory generator is free to assume that the value of that transform will not change for the duration of the motion. When `move()` is called, a private copy of the transform's current value is made and sent to the trajectory generator along with the motion request packet. If the transform's value is changed *after* the call to `move()`, the change will affect only subsequent motion requests.

While this is a reasonable paradigm, one may not always want a transform to be assumed constant. It may be desirable for the trajectory generator to consider a particular transform to be *variable* and constantly read its value and adjust the target position accordingly. In other words, a motion to a target that contains a variable transform will *track* any changes that variable transform makes. The value of a transform can be changed by either the planning task or by functions being executed through the trajectory generator.

The primitive to declare a transform variable is

```
setTransVarb (tr)
  TRSF_PTR tr;
```

After this call is made, subsequent motions which reference the transform will track any changes to it. These changes will be tracked literally, without any of the acceleration limiting that is normally applied between adjacent motions. One should therefore be careful to ensure that changes to such transforms are smooth and clean. If a variable transform contained in a target position changes suddenly by a large amount, the robot will try to "leap" across the work space[1]. One reason that variable transforms should usually be changed from the control level is to ensure that the changes to it are made at a constant rate.

A transform may be set back to being a constant transform with the primitive

```
setTransConst (tr)
  TRSF_PTR tr;
```

One special way of making a transform variable is to bind it to to a function executed at the control level:

[1] Which is a bad thing, because if the robot is large, it might succeed.

```
transEval (tr, fxn, arg)
  TRSF_PTR tr;
  int (*fxn)();
  int arg;
```

This instructs the trajectory generator to call the function `fxn` once every control cycle to adjust the value of `tr`. The function will be called with the following arguments:

```
fxn (tr, arg)
TRSF_PTR tr;
int arg;
{
        ... modify the value of the transform ...
}
```

`tr` points to the transform being updated (or actually a copy of it; the real transform is updated atomically as soon as the updating function returns), and `arg` is the application-defined argument specified by the third parameter to `transEval()`. The trajectory generator begins executing the transform function immediately, and continues to do so until the binding is removed (such as with a call to `setTransVarb()` or `setTransConst()`).

A transform may also be bound to a function that is executed only when the transform is contained within the manipulator's current target position. This is done with the call

```
transMotionEval (tr, fxn, arg, mnp)
  TRANS_PTR tr;
  int (*fxn)();
  int arg;
  MANIP *mnp;
```

The function `fxn` will be called to update `tr` only when `mnp` is executing a motion that contains `tr` as part of the target specification (this is actually the way function binding in the old RCCL system worked). For this sort of function binding, the trajectory generator calls `fxn` with a pointer to the associated `MANIP` structure as an additional argument:

```
fxn (tr, arg, mnp)
TRSF_PTR tr;
int arg;
MANIP *mnp;
{
        ... modify the value of the transform ...
}
```

An alternative way of setting the transform bindings discussed above is to use the pair of functions

```
setTransBinding (tr, binding)
  TRSF_PTR tr;
  TRANS_BINDING *binding;
```

```
getTransBinding (tr, binding)
  TRSF_PTR tr;
  TRANS_BINDING *binding;
```

`setTransBinding()` sets the binding for a transform and `getTransBinding()` reads the current binding back. Binding information is specified or read back through the data type `TRANS_BINDING`, which is defined as follows:

```
typedef struct {
        int code;
        int (*fxn)();
        int arg;
        MANIP *mnp;
} TRANS_BINDING;
```

The fields `fxn`, `arg`, and `mnp` contain the same information that is provided by the arguments to the functions `transEval()` and `transMotionEval()`. `code` describes the type of binding and is set to one the four values

```
TRANS_CONST
TRANS_VARB
TRANS_EVAL
TRANS_MOTION_EVAL
```

depending on the binding.

Function bindings may be applied to any transform in a position equation. When a manipulator is asked to move to such a time varying target, it will track the changes as it moves toward the target (and continue to track the changes if it is asked to stop there).

When setting up functionally defined positions, as with all position equations, it is important to realize that the effect of varying a particular transform depends heavily on where it is located within the position equation: we have to consider the coordinate frame in which the transform is itself defined.

As an illustration of this, consider the position equations defined in Figure 20. Each of these contains a manipulator $\mathbf{T6}$ transform, a fixed transform **BASE** which maps to the manipulator base frame from a "world" coordinate frame W, a fixed **TOOL** transform that maps into a tool frame T, and a single variable transform (either $\mathbf{T}(t)$ or $\mathbf{R}(t)$). The manipulator transform is indicated by $\mathbf{T6}(t)$ as a reminder that it is time varying as well. $\mathbf{T}(t)$ performs pure translation, and $\mathbf{R}(t)$ performs pure rotation.

The first two position equations create pure translational motions, the first with respect to the *world* frame W and the second with respect to the *tool* frame T. The difference is obtained by changing the direction sense of $\mathbf{T}(t)$. (Pure translation always produces a pure translation, no matter what coordinate frame it is induced in.) The difference between the two motions would simply be difference in the orientation of the line along which the motion was taking place.

The difference is more severe for the last two position equations, which depict two rotations applied in two different frames. A rotation in one frame may result in both a rotation and a translation

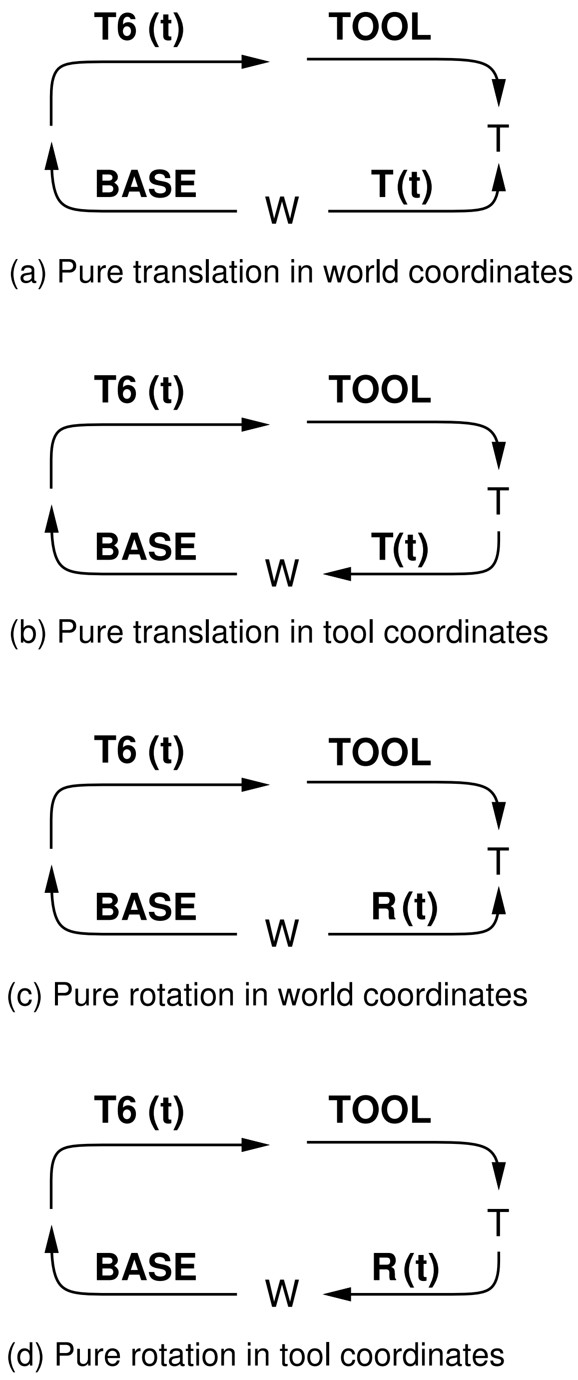

Figure 20: Graphs for four different time-varying positions.

when observed from another frame. In position (c), the center of rotation is fixed with respect to W (and will move with respect to T), while in position (d), the center of rotation will be fixed with respect to T (and will move with respect to W). Rotations in tool coordinates are always hard to visualize because the tool frame itself is moving.

5.2 Program Example: "zigzag"

In this simple example, the robot is programmed to move back and forth between two positions which are moving as though they are attached to a conveyor belt. The net result is a sort of "zigzag" motion (see Figure 21).

planning level module

```
#include <rccl.h>
#include "manex.560.h"

main ()
{
        TRSF_PTR z, e, conv, t1, t2;
        POS_PTR pt1, pt2;
        MANIP *mnp;
        JNTS rcclpark;
        char *robotName;

        int convfn();
        int i;

        rcclSetOptions (RCCL_ERROR_EXIT);
        robotName = getDefaultRobot();
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }

        z = allocTransXyz ("Z", UNDEF, 0.0, 0.0, ZBASE);
        e = allocTransXyz ("E", UNDEF, 0.0, 0.0, TOOLZ);
        conv = allocTransRot ("CONV", UNDEF, C_X, C_Y, C_Z, xunit, 180.0);
        t1 = allocTransXyz ("T1", UNDEF, -300.0,  50.0, 0.0);
        t2 = allocTransXyz ("T2", UNDEF, -300.0, -50.0, 0.0);
                                                                /*#1*/
        pt1 = makePosition ("pt1", z, T6, e, EQ, conv, t1, TL, e);
        pt2 = makePosition ("pt2", z, T6, e, EQ, conv, t2, TL, e);

        mnp = rcclCreate (robotName, 0);
        rcclStart();

        movej (mnp, &rcclpark);                                 /*#2*/
        waitForCompleted (mnp);
```

```
        move (mnp, pt1);
        stop (mnp, 0.0);
        waitForCompleted (mnp);
        setMod (mnp, 'c');                                      /*#3*/

        transEval (conv, convfn, /*speed=*/50);                 /*#4*/

        for (i=0; i<8; ++i)                                     /*#5*/
         { move (mnp, pt1);
           stop (mnp, 1000.0);
           move (mnp, pt2);
           stop (mnp, 1000.0);
           if (i == 3)                                          /*#6*/
            { waitForCompleted (mnp);
              transEval (conv, convfn, -50);
            }
         }
        movej (mnp, &rcclpark);                                 /*#7*/
        stop (mnp, 1000.0);
        waitForCompleted (mnp);
        rcclRelease (1);
}
```

control level module

```
#include <rccl.h>

convfn (t, vel)
TRSF_PTR t;
int vel;
{
        t->p.x += vel * rcclGetInterval() / 1000.0;             /*#8*/
}
```

NOTE – this example has been coded for the PUMA 560 robot, and lives at `zigzag.560.c` *and* `zigzagCtrl.560.c` *in* `demo.rccl`. *An equivalent program for the PUMA 260 is contained in* `zigzag.260.c` *and* `zigzagCtrl.260.c`. *For details on how to compile it, see Section 9.4.3.*

The target position motion is created by binding a transform to a real-time function. Historically, functions which execute at the control level needed to be defined in modules separte from those executing at the planning level. Although this is no longer a strict requirement, the examples in this manual will continue this tradition, and control level functions will be placed in separate modules with a names that end with "`Ctrl`".

The program creates two manipulator target positions, both of which are attached to a fictitious conveyor belt (`#1`). These are described by the following equations:

$$\mathbf{Z}\ \ \mathbf{T6}\ \ \mathbf{E}\ =\ \mathbf{CONV}\ \ \mathbf{T1}$$

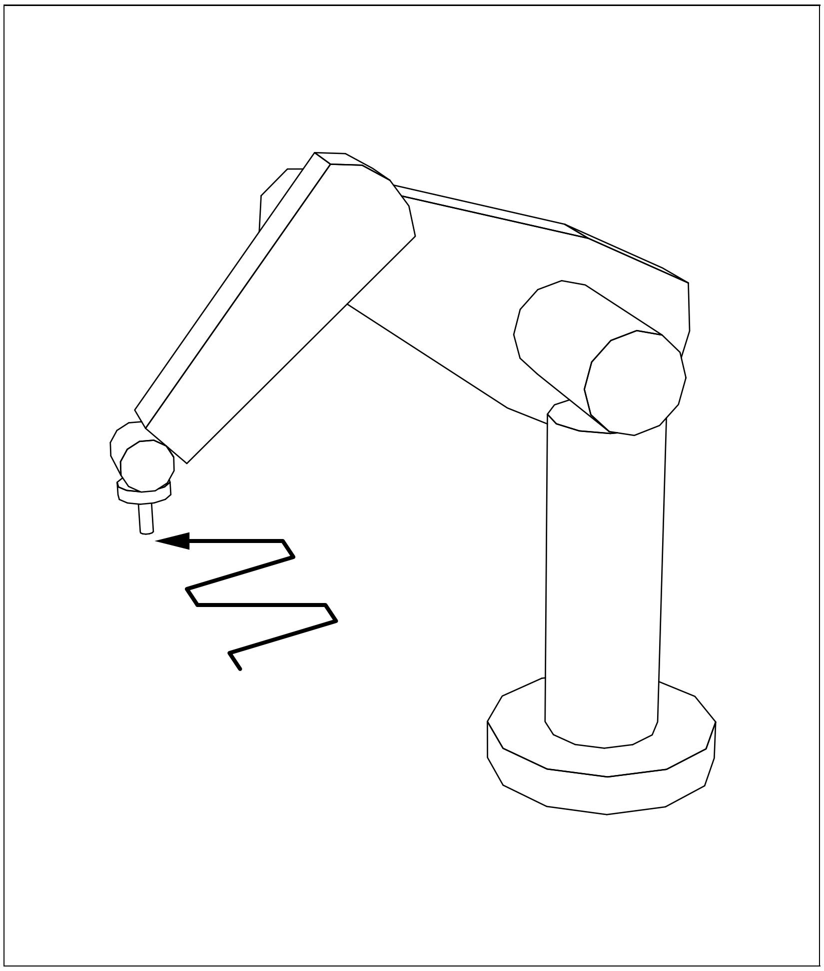

Figure 21: Motion created in the program "zigzag" by moving back and forth between two moving target points.

and

$$\mathbf{Z}\ \ \mathbf{T6}\ \ \mathbf{E}\ =\ \mathbf{CONV}\ \ \mathbf{T2}$$

These positions are a little more complicated than those in previous examples. To begin with, they both use a transform **Z** which maps to the base of the manipulator from some more convenient frame (in this case located at the bottom of the robot's pedestal). This is useful sometimes because the pedestal base may make a more reasonable reference frame than the manipulator base (which is often located in some obscure spot like the middle of the robot's shoulder). The location of the conveyor belt itself, relative to the pedestal base, is then given by **CONV**. The transforms **T1** and **T2** represent the location of each target with respect to the conveyor frame.

The program starts up and brings the robot to the usual starting position (`#2`). It then moves the robot to the first target point, stops it there, waits for completion, and then requests Cartesian interpolated motions (`#3`). The conveyor is not yet "in motion"; this is done next by using `transEval()` to bind `conv` to the function `convfn` (`#4`). The application argument is used (in this case) to indicate the conveyor velocity, in millimeters per second. As soon as the function binding takes place, `convfn()` will begin to "move" **CONV** in a positive direction along its x axis. The program then issues a series of requests to move the robot back and forth between the two target points attached to the conveyor, stopping at each point for one second. Since the points are separated along the y axis of the conveyor, perpendicular to the direction of motion, this results in a "zigzag" like motion. Half way through the move-stop sequence, the direction of the conveyor is reversed with another call to `transEval()` (#6), which leaves the function binding unchanged, but negates the velocity argument. When this is finished, the program moves the robot back to the starting point and terminates in the usual way (#7).

As mentioned above, the function `convfn()` is defined in a separate load module since it will be executed at the control level. It is defined to receive two arguments: a pointer to the transform it is to modify and the application argument. The application argument, which specifies the conveyor velocity in millimeters per second, is multiplied by the RCCL sample interval (obtained with `rcclGetInterval()`) to obtain a per-cycle increment which is added to the x axis of the transform (#8).

5.3 Restrictions for Control Level Functions

Before proceeding further, we should describe in detail the restrictions that apply to functions being executed at the control level.

Control level application functions are executed by the trajectory generator, once per cycle, within the context of whichever process is used to implement the trajectory generation task. For the most recent version of RCCL, this will likely be a lightweight UNIX process running at real-time priority.

In the case of a multi-CPU implementation of the trajectory generator, each application function will be executed only on *one particular* CPU. In some cases, this CPU is chosen automatically by the system. In other cases, the function may be associated with a particular manipulator, and if so, then the function is executed on the manipulator's CPU (recall from the description of `rcclCreate()` that manipulators are tied to CPUs). Functions which are associated with a manipulator include monitor functions (to be described later) and transform motion functions.

Because the control level must run in real-time, UNIX system calls and other computations which are time consuming (relative to the control sample interval) should be avoided. One should also be cautious about calling process control functions (such as `fork()`).

Again, for time efficiency reasons, system calls which block (i.e., suspend process execution) should probably be avoided at the control level, unless one is very confident that the blocking time will be small. If the control level accesses a routine which *does* block, then this needs to be handled specially; see Section 5.3.1. Routines such as `malloc()` and `printf()` are *likely* to be safe from blocking, but RCI provides surrogate routines to perform these functions, just in case; see Section 5.6.2.

Some routines in the RCCL/RCI library cannot be called from the control level, mainly because of concurrency issues and because of the way RCI control tasks are allocated and managed. Routines which allocate structures, or operate control tasks, are usually in this category. Whether or not an RCCL/RCI routine can be called from the control level will generally be indicated in its reference manual page under the heading **Restrictions**.

Another possible problem with control tasks is infinite loops. If control code enters an infinite loop, then this has a very good chance of hanging the machine (since the high priority of the control task will prevent any interrupt signals from getting through). Care should be taken to prevent infinite loops.

5.3.1 Calling functions which block

If control code calls a function which blocks (i.e., suspends process execution), then this function needs to be encased within calls to `rciSaveContext()` and `rciRestoreContext()`. For example, suppose the application wishes to make a call to `read()`. Then this should be set up as follows:

```
rciSaveContext();
read (fd, buf, size);
rciRestoreContext();
```

The need to call these two special RCI routines stems from earlier versions of RCCL/RCI, in which control level tasks were *unable* to block.

The requirement to call `rciSaveContext()` and `rciRestoreContext()` may be eliminated in the future.

5.4 Ways of Modifying Transforms With Functions

There are several ways that a function bound to a transform may change its value. We will mention a few of these here, although the discussion is by no means exhaustive.

Absolute Changes. The transform's value is set directly each control cycle. For instance, we might have quantities `px`, `py`, and `pz` which are read in from some device such as a joystick. These could be used to set the transform's **p** vector, as in

```
fxnAbs (tr, arg)
TRSF *tr;
int arg;
{
        float px, py, pz;

        ... values are read in from somewhere ...

        xyzToTrsf (tr, px, py, pz);
}
```

Relative Changes. The transform's value is changed incrementally. A common way of accomplishing this is to multiply the transform's existing value by a perturbation transform. Assume that the quantities `dx`, `dy`, and `dz` are read in from some sensor device. The following sample function uses these to apply a relative translational increment to its transform:

```
fxnRel (tr, arg)
TRSF *tr;
int arg;
{
        TRSF offset;
        float dx, dy, dz;

        ... values are read in from somewhere ...

        xyzToTrsf (&offset, dx, dy, dz);
        multTrsf (tr, tr, &offset);
}
```

The same thing could be accomplished by a call to `multTrsfXyz()` instead of two calls to `xyzToTrsf()` and `multTrsf()`. The difficulty with relative changes is that they can cause errors in the transform structure. This problem is more severe when modifying the rotational component, which can lose orthogonality quite easily. This situation can be improved somewhat by renormalizing the transform regularly, such as with the routine `unitTrsf()`. Another option is to use a representation for the net rotation that has fewer parameters (this is particularly easy if the rotation is about a single axis), change the reduced **that**, and then convert it into a transform once every cycle. This will in fact be done in the example below.

Time Dependent Changes. The changes to a transform may depend explicitly on time. For instance, suppose that we desire to make a transform rotate slowly about the z axis with a rotational velocity ω. A function that will do this is

```
rotZ (tr, omega)
TRSF *tr;
int omega;      /* omega, in deg/sec */
{
        float angle;
        static float t0 = F_UNDEF;

        if (t0 == F_UNDEF)
         { t0 = rcclSysTime();
         }
        angle = omega*(rcclSysTime() - t0)/1000.0;
        rotToTrsf (tr, zunit, angle);
}
```

It obtains ω through the application argument and uses `rcclSysTime()` to get the current system time in milliseconds. Because absolute time is being used, it is necessary to keep track of the initial time; this is done using the static variable `t0`. An equivalent function that does the same thing with less fuss, using incremental changes, is

```
rotZrel (tr, omega)
```

```
TRSF *tr;
int omega;      /* omega, in deg/sec */
{
        multTrsfRot(tr,zunit,omega*rcclGetInterval()/1000.0);
}
```

This simply uses `rcclGetInterval()` to obtain the system sample interval (in milliseconds) and converts this into a per-cycle displacement.

Motion Relative Changes. The variation of a transform can also be tied directly to the scale value of a motion, that is, the value that changes from 0.0 to 1.0 as the motion goes from start to finish. For example, suppose we wish to add a translational displacement `dp` to a transform over the course of a motion. Assuming that `dp` has been defined somewhere, a function that would do this is

```
dispFxn (tr, arg, mnp)
TRSF *tr;
int arg;
MANIP *mnp;
{
        float s;

        s = motionScale (getActiveMotionId (mnp);
        tr->p.x = dp.x * s;
        tr->p.y = dp.y * s;
        tr->p.z = dp.z * s;
}
```

The motion scale is obtained using the `motionScale()` function called with the ID of the current motion for `mnp`. Several things about the motion scale value should be noted. First, if the motion has no explicit time limit, then the scale will always be 0.0. Second, if the motion starts with a non-zero transition time, then the motion scale will be negative prior to the transition midpoint, since scale is defined to be 0.0 only at the transition midpoint.

5.5 Communicating with the Control Functions

5.5.1 Memory objects

The function `convfn()` in the program `zigzag` obtained information from the planning level using the application argument. This works well, provided that no more than one integer's worth of information needs to be passed. To pass more information between the planning and control levels, shared memory facilities are available.

These basically work as follows: the planning task allocates a piece of shared memory, the control level routines obtain a pointer to it, and then everybody can access it. The basic shared memory mechanism used by RCCL is the RCI `allocMem()` package, which is also described in the *RCI*

User's Guide. Its manual pages appear in the *RCI Reference Manual*, instead of the *RCCL Reference Manual*.

Let us assume that the programmer has defined a structure `foo`:

```
typedef struct foo {
        int value1;
        int value2;
} FOO;
```

and would like to create an object of this type which is shared between the planning and control levels.

The object is allocated by the planning level using the call

```
FOO *fp;

fp = (FOO*)allocMem ("foo", sizeof(FOO), UNDEF);
```

`allocMem()` takes three arguments: a string name for the memory object, the size of the memory object in bytes, and a specifier for the memory allocation pool to be used. This last argument serves the same purpose as the `pool` argument in `allocTrans()`; it is typically not too important and is set to `UNDEF`. Also, as with transforms, the name argument is optional. `allocMem()` allocates the memory, zeros it, and returns a pointer to it. Memory can be deallocated with the call `freeMem()`.

Because the memory returned by `allocMem()` is almost invariably cast to some structure, there is a macro available to make this easier:

```
#define ALLOC_MEM(n,t,p) (t*)allocMem(n,sizeof(t),p)
```

Using this, the allocation of `"foo"` shown above would be written as

```
fp = ALLOC_MEM ("foo", FOO, UNDEF);
```

For another part of the program (such as a control level routine) to access a shared memory object, it must first obtain a valid pointer to it. Doing this generally requires using a reference function. This is because the control level may reside on a separate CPU, and a memory address which is valid at the planning level may not be valid on a different CPU. To get a valid pointer, we may use either of the functions

```
void *getMemByName (name)
  char *name;

void *getMemByAddr (addr)
  void *addr;
```

`getMemByName()` locates a pointer to the memory using its `name` as a key (the same name that was specified in the `allocMem()` call), while `getMemByAddr()` locates the pointer using the planning level address (the same one returned by `allocMem()`) as a key. In practice, the routine `getMemByAddr()` is used more commonly, because (1) it is faster, and (2) it relieves us of the need to name the memory object, thereby avoiding name space collision problems. The planning level address can be passed to the control routine using either another object or the routine's application argument[2].

5.5.2 Access Collision and Atomic Access

When using shared memory to communicate between different tasks, one frequently runs into the program of access collision: one task may try to read an object at the same time another task is writing it. Often enough this is not a problem, since the primary data types which comprise the object (`float`, `int`, etc.) are usually written atomically on most systems and bus architectures for which RCCL/RCI is implemented. The biggest problem is that an aggregate which is read back can be "out of sync", *i.e.*, while the contents of the individual fields are OK, some of them contain "old" information and some contain "new" information. For instance, suppose the planning level reads back the output joint angles in the `j6` field of the `MANIP` structure, which could be done simply with the code fragment

```
JNTS jbuf;

jbuf = *mnp->j6;
```

While the structure is being copied, it is possible for the trajectory generator to write new values to it. This means that the first few joint values may be valid for one control cycle while the remaining values are valid for another control cycle. This sort of thing may not be a problem, depending on what the information in the object is being used for. However, to avoid the situation, there are three primitives available which may be used to atomically read and write memory objects which have been created using `allocMem()`:

```
readMem (mem, buf)
  void *mem, buf;

writeMem (mem, buf)
  void *mem, buf;

accessMem (mem, src, dst, size)
  void *mem, src, dst;
  int size;
```

`readMem()` reads the contents of memory object `mem` into `buf` (which is assumed to be the same size as the memory object) and `writeMem()` writes the contents of `buf` into the memory object `mem`. `buf` is not assumed to be a memory object and access to it is **not** atomic. The last routine, `accessMem()`, is the most general: it reads `size` bytes from `src` into `dst` while locking the memory object `mem` (which may be either `src` or `dst`, or neither). It is intended for accessing subfields of an object.

[2]Time for a confession: we frequently cast pointers to integers and back for purposes of passing them through the application argument. The clean way around this would be to either make the application argument itself a generic pointer, or to make it a union of, say, a generic pointer and an integer. A change like this may be implemented if it becomes important, so stay tuned ...

Needless to say, in order for the atomic access primitives to work, all write operations (and read operations, usually) on the object in question must use them. All of the aggregate fields in the `MANIP` structure are updated by the trajectory generator using calls to `writeMem()`. They may be read atomically using either `readMem()` or `writeMem()`, as in

```
JNTS jbuf;
readMem (mnp->j6, &jbuf);
```

or

```
float angles[NUM_JNTS];
accessMem (mnp->j6, mnp->j6->v,
           angles, mnp->j6->num*sizeof(float));
```

The first example copies the entire structure. The second example is a little faster because it copies from `j6` only the `v` field entries corresponding to actual joints.

5.5.3 Double Buffering

We should mention briefly another common technique that can be used to avoid data consistency problems. This is called double buffering, and can be used whenever there are several tasks that read an object but only one task that writes it. It is conceptually simple, and generally faster than using the atomic access primitives. Suppose that the data we are interested in is described by a type `DATA`. We create an object that contains two versions of `DATA` and a version number:

```
struct dataObj {
        DATA data[2];
        int version;
}
```

The data buffer which is currently in use is indicated by the low bit of `versionNumber`. When the writing process updates the data, it changes the buffer that is **not** in use, and then increases the version number:

```
writeData (mem, buf)
struct dataObj *mem;
DATA *buf;
{
        int new = (mem->version&1)^1;

        mem->data[new] = *buf;
        mem->version++;
}
```

The process doing the read uses the data buffer that is in use, and then checks to see if the version number has increased by more than one. If it has, it tries again:

```
readData (mem, buf)
struct dataObj *mem;
DATA *buf;
{
        int idx;
        do
         { idx = mem->version;
           *buf = mem->data[idx&1];
         }
        while ((mem->version&~1) != (idx&~1))
}
```

This will work quite nicely in practice, as long as the data is not updated so frequently as to saturate the system (an update once every RCCL control cycle would be considered quite infrequent).

5.5.4 Locating and Using Other Objects

Now to divulge what the alert reader was probably beginning to suspect: Transforms allocated with `allocTrans()` and position equations allocated with `makePosition()` are also memory objects, completely analogous to the ones described in the section (5.5.1). They each have their own access functions:

```
TRSF *getTransByName (name)
  char *name;

TRSF *getTransByAddr (addr)
  TRSF *addr;

POS *getPositionByName (name)
  char *name;

POS *getPositionByAddr (addr)
  POS *addr;
```

which can obtain pointers to them given either names or planning level addresses. One reason for giving these objects separate access routines is to make it possible to separate their name spaces. For instance, it is possible to create a memory object using `allocMem()` and a transform using `allocTrans()` which both have the same string name.

It is also necessary to use a reference routine to get a pointer to a `MANIP` structure from the control level, even though these items are not actually implemented as formal objects. The routines

```
MANIP *getManipByName (name)
  char *name;

MANIP *getManipByAddr (addr)
  MANIP *addr;
```

are provided to do this. (NOTE: a valid `MANIP` pointer is frequently given as an argument to the control level application functions. It is not necessary to re-reference this pointer. However, sometimes we want to access the `MANIP` structure for *another* manipulator from within one of these routines, and for this the above routines can be useful.)

Transforms also have their own atomic access routines, analogous to the ones described in section (5.5.2):

```
TRSF *readTrans (tr, buf)
  TRSF *tr, *buf;

TRSF *writeTrans (tr, buf)
  TRSF *tr, *buf;
```

To find the name of a transform given a pointer to it, the routine

```
char *getTransName (tr)
  TRSF *tr;
```

may be used.

Using position equations from the control level is slightly more complicated. Although one may access a position equation from anywhere within the RCCL program, the equation returned will generally not be usable at the control level because the data structures within the equation itself use pointers whose values are valid for the planning level. To get around this problem, a routine called `copyPosition()` is available which makes a local copy of a position object which is valid for the current memory context. When the local copy of the position is no longer needed, it can be deleted using `freePosition()`. A typical usage paradigm for `copyPosition()` looks something like this:

```
-- planning level --

   POS *gpos;

   gpos = makePosition (NULL, T6, tool, EQ, b, TL, tool);

-- control level --

   POS lpos;

   ... gpos is passed down from the planning level ...

   lpos = copyPosition (getPositionByAddr (gpos));

    ... use 'lpos' locally ...

   freePosition (lpos);
```

The planning level declares the position equation in the usual way using `makePosition()`. Later, the control level uses `getPositionByAddr()` to get a local reference to equation using the planning level's original address, and then translates this into valid local equation using `copyPosition()`. This local copy may then be used with the routines which do position equation computations, such as `solveTrans()` or `solveChain()`. Note that `copyPosition()` copies only the equation itself, not the transforms that it references.

The ability to access positions from the control level is new and has not been used much. More information on this feature can be found in the manual pages.

5.5.5 Synchronizing with the control level

It is sometimes necessary to synchronize activity between the planning level and an application-defined control level function. The basic way of doing this is with flags implemented using shared memory. Suppose the planning level wants to send a command to a monitor function, and then wait for the monitor function to respond. The command can be implemented as a field (which we will call `com`) in a shared memory object (which we will assume is called `mem`). Then the following simple construction may be used:

```
-- planning level --

   mem->com = COMMAND;
   waitWhile (mem->com)

-- control level function --

   if (mem->com)
    { ... do action ...
      mem->com = 0;
    }
```

The planning level just sets the command field and waits for the control level function to clear it. If it is known that control level will respond during the next control cycle, then the planning level can use the primitive `rcclBlock()` instead:

```
mem->com = COMMAND;
rcclBlock();
```

`rcclBlock()` waits for one entire control cycle to elapse between the time it is called and the time it returns, and so can be used to effect single cycle synchronization.

5.6 Control Level Support Routines

5.6.1 RCCL system routines

The following RCCL functions may be called from anywhere with an RCCL program, but are particularly useful for control level applications.

The system time, in milliseconds, may be obtained with

```
float rcclSysTime();
```

This is not a “real time”, but rather is the product of the trajectory generator sample interval and the number of elapsed control cycles. In other words, it can be thought of an an “ideal” time associated with the beginning of the most recent control cycle. The value returned by `rcclSysTime()` does not increase when the trajectory generator is not running or is paused (as with `rcclPause()`), and is reset to 0 every time `rcclStart()` is called. Applications which change the value of a transform explicitly with time are encouraged to use `rcclSysTime()`, because it “stops ticking” when the system is paused, and so prevents discontinuities in target positions when the system is unpaused.

To obtain a “real” time value, applications may use

```
struct timeval rciWallClock();
```

which returns a `timeval` structure containing the most accurate real time information available to the system (up to the one micro-second resolution of the `timeval` structure). Of course, applications may also call whatever time functions, such as `gettimeofday()` or `gethrtime()`, are available on a specific system.

The routine

```
getActiveCount(mnp)
  MANIP *mnp;
```

returns the *count* associated with a manipulator’s current motion. The count starts at 0 (at the beginning of the transition into the motion)and is increased by one for every control cycle that the motion is executed. It can be used to initialize transform functions:

```
function (tr, arg, mnp)
TRSF *tr;
int arg;
MANIP *mnp;
{
        if (getActiveCount(mnp) == 0)
         {  ... motion is just starting, do
                some special stuff ...
         }
}
```

5.6.2 RCI support routines

In earlier releases of RCCL/RCI, functions executing at the control level did not have access to UNIX system services. Therefore, they could not call library routines such as `printf()` or `malloc()` which use system services. To compensate for this, RCI provided a few routines which could be called from the control level to do diagnostic printing, allocate memory, and send signals to the main UNIX program.

Although the restriction on using system services has disappeared, care should still be taken to not call services that are likely to block or cause excessive delays. The original RCI support routines are still available, and should probably be used when the user has doubts about using other system routines directly. These support routines include:

```
rciPrintf (format, values ... )
  char *format;

void *rciMalloc (size)
  int size;

rciFree (ptr)
  void *ptr;

rciSignal (signum)
  int signum;
```

`rciPrintf()` uses the same format as `printf()` to print diagnostic messages to the program's terminal. Care should be used when doing printing from the control level, since it can cause large delays in the execution of the control level code.

`rciMalloc()` and `rciFree()` are equivalent to the conventional UNIX routines `malloc()` and `free()`. There is generally a limit to the total amount of memory that can be allocated, and this may be as low as 256K.

`rciSignal()` sends the indicated UNIX signal `signum` to the main program.

For details on all of these functions, the programmer should consult their manual pages in the *RCI Reference Manual*.

5.6.3 The Fast Math Library

Robotics applications make frequent use of the trigonometric functions `sin()`, `cos()`, `atan2()`, and the function `sqrt()`. The execution times for these routines can be lengthy (by real-time standards), particularly if the host machine does not have hardware support for these functions or the native math library does not make use of it.

To help with this problem, RCCL provides a set of routines that do the same computations using table lookup and interpolation. Execution times are significantly faster (by as much as 5 or 6 times), and the accuracy is typically around 10^{-6}, which is quite tolerable for many applications.

The available routines are

```
float SIN (x)
  float x;

float COS (x)
  float x;

float ATAN2 (x, y)
  float x, y;

SQRT (x)
  float x;

SINCOS (x, s, c)
  float x;
  float *s, *c;
```

`SIN()`, `COS()`, `ATAN2()` and `SQRT()` do the same computations as their math library counterparts (except the arguments and results are in single precision). All angle values are passed in radians. `SINCOS()` computes both the sine and cosine of `x`, and returns the values through the pointers `s` and `c`, respectively. This routine exists because it is faster to compute both these quantities together than to compute them separately.

Definitions for these routines are contained in the file `<fastmath.h>`. The user is encouraged to make use of them.

5.7 Program Example: “rotate”

This demonstration program uses two transforms bound to circular motions to create a “wheels within wheels” effect. The robot is located at the center of fictitious circular table which can be made to rotate about the robot’s base. An initial motion is made to a point tracking a small wheel whose axis is fixed at a point on the larger turntable (Figure 22). Another motion is then specified for which the table itself begins to turn and the robot starts tracing out a “circular cycloid” (Figure 23). This last motion automatically aborts when the table has traveled through a certain distance.

planning level module

```
#include <rccl.h>
#include "manex.560.h"

main ()
{
        TRSF_PTR table0, table, p, wheel;
        POS_PTR p0, pt;
        MANIP *mnp;
        ROTATE_CTRL_BLK *rcb;                                   /*#1*/
```

```
        JNTS rcclpark;
        char *robotName;

        int tableFxn();
        int wheelFxn();
        int mid;

        rcclSetOptions (RCCL_ERROR_EXIT);
        robotName = getDefaultRobot();
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }

        p = allocTransRot ("P", UNDEF, P_X, P_Y, P_Z, xunit, 180.0);
        table0 = allocTransRot ("TABLE0", UNDEF, 0.0, 0.0, 0.0, zunit, -45.0);
        wheel = allocTrans ("WHEEL", UNDEF);
        table = allocTrans ("TABLE", UNDEF);

        rcb = ALLOC_MEM ("rcb", ROTATE_CTRL_BLK, UNDEF);        /*#2*/

        rcb->wheelRadius = 70.0;                                /*#3*/
        rcb->wheelRPS = .50;
        rcb->tableRPS = 0.02;

        p0 = makePosition ("p0", T6, EQ, table0, p, wheel, TL, T6);
        pt = makePosition ("pt", T6, EQ, table0, table, p, wheel, TL, T6);

        mnp = rcclCreate (robotName, 0);
        rcclStart();

        transEval (wheel, wheelFxn, 0);                         /*#4*/

        movej (mnp, &rcclpark);

        setMod (mnp, 'c');                                      /*#5*/
        move (mnp, p0);
        mid = stop (mnp, F_UNDEF);                              /*#6*/

        transMotionEval (table, tableFxn, 0, mnp);              /*#7*/
        setTime (mnp, F_DEFAULT, F_UNDEF);
        move (mnp, pt);                                         /*#8*/
        movej (mnp, &rcclpark);                                 /*#9*/
        stop (mnp, 1000.0);

        waitFor (motionStatus(mid) == M_RUNNING);               /*#10*/
        delay (10000.0);                                        /*#11*/
        stopMotion (mid, 0);

        waitForCompleted (mnp);
        rcclRelease (1);
}
```

control level module

```
#include <rccl.h>
#include <fastmath.h>
#include "manex.560.h"

wheelFxn (t, arg)
TRSF_PTR t;
int arg;
{
        ROTATE_CTRL_BLK *rcb;
        static float alpha = 0.0;
                                                                /*#12*/
        if ((rcb = (ROTATE_CTRL_BLK*)getMemByName("rcb")) == NULL)
         { rciAbort (0, "wheelFxn(): can't find argument object\n");
           return;
         }
        alpha += rcb->wheelRPS * rcclGetInterval() / 1000.0;    /*#13*/

        t->p.x = rcb->wheelRadius * COS(alpha * PIT2);
        t->p.y = rcb->wheelRadius * SIN(alpha * PIT2);
}

tableFxn (t, arg, mnp)
TRSF_PTR t;
int arg;
MANIP *mnp;
{
        ROTATE_CTRL_BLK *rcb;
        static float alpha = 0.0;
        static done = 0;
                                                                /*#14*/
        if ((rcb = (ROTATE_CTRL_BLK*)getMemByName("rcb")) == NULL)
         { rciAbort (0, "tableFxn(): can't find argument object\n");
           return;
         }
        alpha += 360.0 * rcb->tableRPS * rcclGetInterval() / 1000.0;
        rotToTrsf (t, zunit, alpha);

        if (alpha >= 90.0 && !done)                             /*#16*/
         { stopCurrentMotion (mnp, 1);
           done = 1;
         }
}
```

NOTE – this example has been coded for the PUMA 560 robot and lives at `rotate.560.c` *and* `rotateCtrl.560.c` *in* `demo.rccl`. *An equivalent program for the PUMA 260 is contained in* `rotate.260.c` *and* `rotateCtrl.260.c`.

The first new thing the reader may notice in this example is the pointer declaration for the type `ROTATE_CTRL_BLK` (#1), which is defined in `"manex.560.h"`. A memory object of this type is allocated using `allocMem()` (#2) and is used to communicate information to the control level functions. The fields of the structure contain parameters for the function computations: `wheelRadius` is the radius of the smaller wheel; `wheelRPS` is the speed the smaller wheel should spin at (in revolutions per second), and `tableRPS` is the speed the turntable should spin at (also in revolutions per second) when it starts to move (#3).

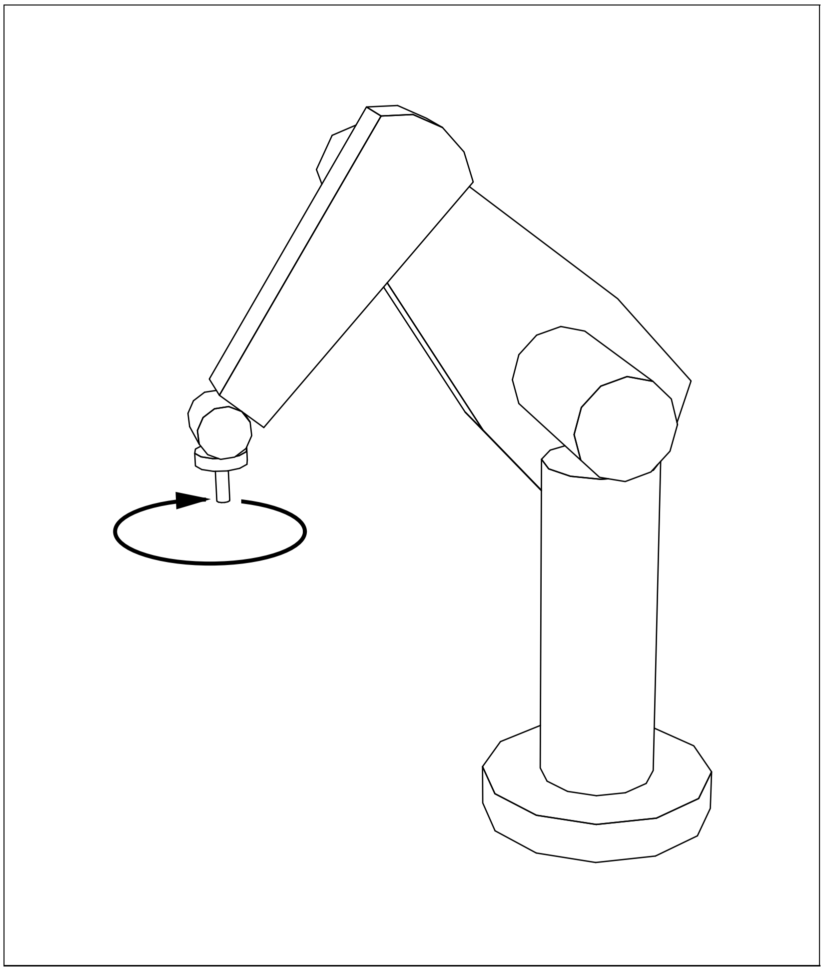

Figure 22: Manipulator tracking a rotating "wheel", in the example program `rotate`.

The program uses two position equations: `p0`, defined as

$$\mathbf{T6} = \mathbf{TABLE0\ \ P\ \ WHEEL}$$

describes a point on the spinning wheel attached to the larger turntable. Notice that for this program we have omitted the manipulator TOOL transform. **TABLE0** describes a rotation about the z axis of the manipulator base frame. **P** defines a translational offset and flip about the x axis relative to **TABLE0**. Both of these transforms are fixed. **WHEEL** is a moving transform whose translation values trace out a circle. Position `p0` assumes that the turntable is fixed at the initial rotation angle of **TABLE0**. Position `p1`, defined as

$$\mathbf{T6} = \mathbf{TABLE0\ \ TABLE\ \ P\ \ WHEEL}$$

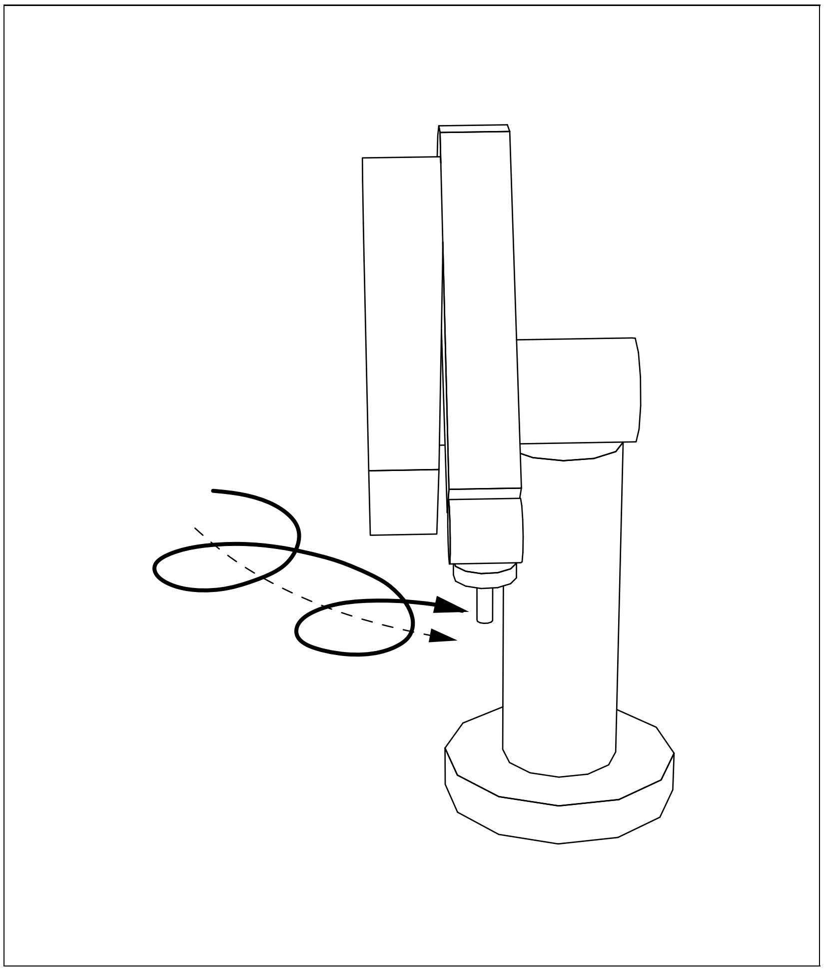

Figure 23: Tracking the "wheel" when its axis of rotation is fixed to a "turntable" rotating about the manipulator's base.

is similar, except that the turntable is now set in motion by an additional (moving) transform **TABLE**.

After the program has turned on the trajectory generator, it sets the wheel transform in motion with a call to `transEval()` (`#4`). The application argument is specified as 0 because it will not be used. After moving to the canonical initial position, the robot is told to do a Cartesian interpolated motion to `p0` and stop there indefinitely (`#5`), during which time it will continue to track the motion of the wheel (Figure 22). Without waiting for this to complete, the program then sets up the motion requests for the rest of the program. A call to `transMotionEval()` (`#7`) binds **TABLE** to the function `tableFxn()`, which will be executed whenever **TABLE** is contained in the target of the current motion. A request for a motion of indefinite time to position `p1` is then queued (`#8`), followed by a final motion request which moves the robot back to the starting position (`#9`).

Both the stop at `p0` (`#6`) and the motion to `p1` (`#8`) are specified to last indefinitely, which means that they must be explicitly aborted by the program. The stop is canceled by the planning level, which first waits for the stop request to start executing (`#10`), waits another 10.0 seconds (`#11`), and then cancels it with a call to `stopMotion()`. The motion to `p1` is canceled from within the function that produces that turntable motion when the turntable offset angle exceeds $90°$ (#16).

The control level functions are quite simple. Each first obtains a pointer to the control structure using `getMemByName()` and the name `"rcb"` (#12, #14). If the memory object is not found, the program is aborted with a call to `rciAbort()` (see Section 9.2.2). `wheelFxn()` causes the translational component of its transform to trace out a circle in the x–y plane. It increments the current angle of rotation (#13) and uses this to compute appropriate values for `t->p.x` and `t->p.y`. `tableFxn()` causes its transform to rotate about the z axis. To do this, it increments a rotation angle and feeds this into the function `rotToTrsf()`.

5.8 Program Example: "pivot"

This next program makes the robot move through a large arc, with its tool pointing along the radius of the arc, as though it were turning a lever (Figure 24). The robot will line itself up with the lever's initial position and then use a functional transform to move it through $60°$. Circular arc motions are interesting in that they can sometimes be accomplished by locating the TOOL frame in a particular place, rather than by using moving frames. As an illustration, the program does this to move the "lever" back.

planning level module

```
#include <rccl.h>
#include "manex.560.h"

extern pivotFxn();

main()
{
        TRSF_PTR e, b, handle, rot0, rot1, roty;
        POS_PTR p0, p1, pt, pd;
        MANIP *mnp;
        JNTS rcclpark;
        char *robotName;

        rcclSetOptions (RCCL_ERROR_EXIT);
        robotName = getDefaultRobot();
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }

        e = allocTransXyz ("E", UNDEF, 0.0, 0.0, TOOLZ);
        b = allocTransRot ("B", UNDEF, B_X, B_Y, B_Z, xunit, 180.0);
        handle = allocTransXyz ("HANDLE", UNDEF, 0.0, 0.0, -300.0);
        rot0 = allocTransRpy ("ROT0", UNDEF, 0.0, 0.0, 0.0, 0.0, -30.0, 0.0);
        rot1 = allocTransRpy ("ROT1", UNDEF, 0.0, 0.0, 0.0, 0.0,  30.0, 0.0);
        roty = allocTrans ("ROTY", UNDEF);
```

```
        p0 = makePosition ("p0", T6, e, EQ, b, rot0, handle, TL, e);
        pt = makePosition ("pt", T6, e, EQ, b, roty, handle, TL, e);
        p1 = makePosition ("p1", T6, e, EQ, b, rot1, handle, TL, e);
        pd = makePosition ("pd", T6, e, EQ, b, rot0, handle, TL, handle);

        mnp = rcclCreate (robotName, 0);
        rcclStart();

        setMod (mnp, 'c');
        distance (mnp, "dz", -75.0);                            /*#1*/
        move (mnp, p0);
        move (mnp, p0);

        setTime (mnp, 200.0, 12000.0);                          /*#2*/

        *roty = *rot0;
        transMotionEval (roty, pivotFxn, 0, mnp);               /*#3*/
        move (mnp, pt);
        distance (mnp, "dz", -75.0);                            /*#4*/
        move (mnp, p1);

        move (mnp, p1);                                         /*#5*/
        setTime (mnp, 200.0, 12000.0);
        move (mnp, pd);
        distance (mnp, "dz", -75.0);
        move (mnp, p0);

        setMod (mnp, 'j');
        movej (mnp, &rcclpark);
        stop (mnp, 1000.0);

        waitForCompleted (mnp);                                 /*#6*/
        rcclRelease (1);
}
```

control level module

```
#include <rccl.h>

pivotFxn (tr, arg, mnp)
TRSF_PTR tr;
int arg;
MANIP *mnp;
{
        static TRSF tr0;
        TRSF trr;
        float scale;

        if (getActiveCount(mnp) == 0)                           /*#7*/
```

```
          { tr0 = *tr;
          }
         scale = motionScale(getActiveMotionId(mnp));             /*#8*/
         identTrsf (&trr);
         rotToTrsf (&trr, yunit, 60.0*scale);
         multTrsf (tr, &tr0, &trr);
}
```

NOTE – this example has been coded for the PUMA 560 robot and lives at `pivot.560.c` *and* `pivotCtrl.560.c` *in* `demo.rccl`. *An equivalent program for the PUMA 260 is contained in* `pivot.260.c` *and* `pivotCtrl.260.c`

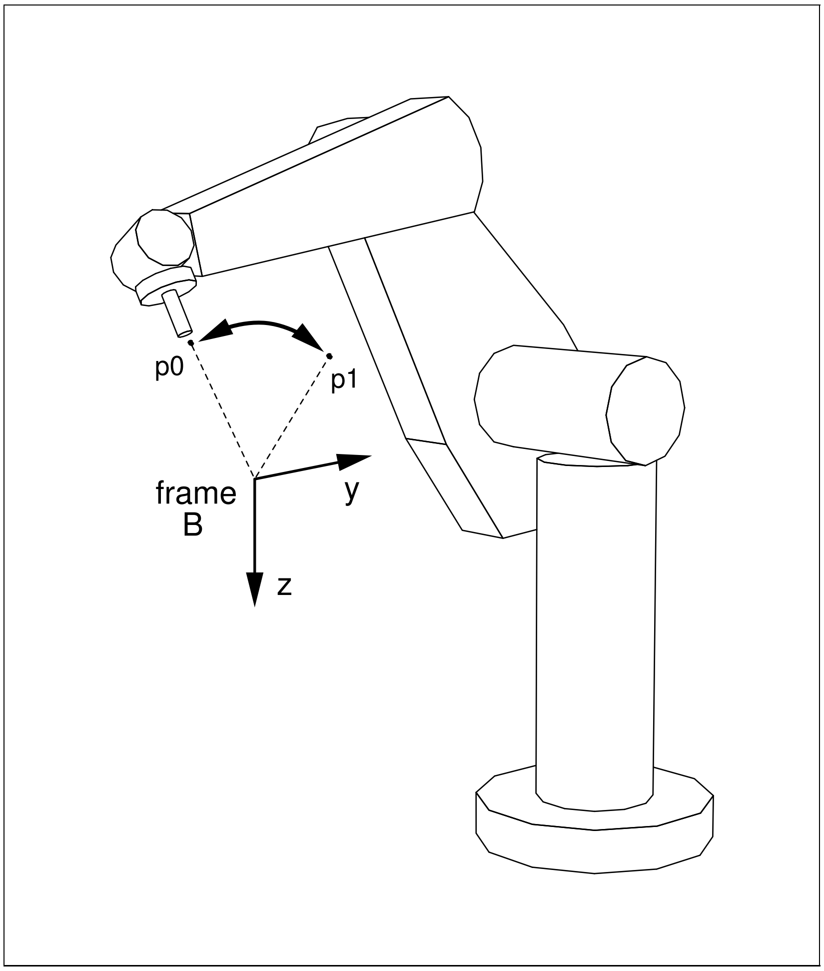

Figure 24: Robot "turning a lever" about a pivot point, in the program example `pivot`.

The program sets up four position equations `p0`, `pt`, `p1`, and `pd`, defined respectively as

$$\mathbf{T6\ \ E\ = B\ \ ROT0\ \ HANDLE}$$

$$\mathbf{T6\ \ E\ = B\ \ ROTY\ \ HANDLE}$$

$$\mathbf{T6\ \ E\ =\ B\ \ ROT1\ \ HANDLE}$$

$$\mathbf{T6\ \ E\ \ HANDLE^{-1}\ =\ B\ \ ROT0}$$

These describe the various positions of the tip of the lever. The lever's center of rotation is located at frame B. To get from there to the tip, we rotate by some amount about the y axis (which is done using either **ROT0**, **ROTY**, or **ROT1**), and then use **HANDLE** to translate along the z axis. Position `p0` defines the initial position of the lever, with **ROT0** describing a rotation of $-30.0°$. The place we want to move the lever to is described by `p1`, which is the same as `p0` except that its rotation angle (given by **ROT1**) is $30°$. To get from `p0` to `p1` along an arc, the transform **ROTY** is used. **ROTY** is bound to a function which varies its rotation angle from $-30.0°$ to $30.0°$. Position `pd` is used to move back from `p1` to `p0`. It describes the same target position as `p0`, except that the TOOL frame is located at the "pivot", the base of the handle, coincident with frame B. In this frame, the displacement from `p1` to `p0` is purely rotational, and so when the trajectory generator interpolates between the two positions, it effectively follows the same path (in reverse) as was created by **ROTY** in the first motion.

The robot first approaches and then positions itself at `p0` (`#1`). An explicit motion time of 12 seconds is established for the motion to `pt` (`#2`), since the trajectory generator would otherwise compute this based only on the displacement between `p0` and `pt`, which is initially zero. **ROTY** is initialized to the value of **ROT0**, and then bound to a transform motion function (`pivotFxn()`) that will change the rotation angle (`#3`). The motion to `pt` is then requested, which brings the robot to the final position described by `p1`. This is followed by a small depart motion away from `p1` (`#4`).

To return to `p0`, the robot is first moved back to `p1`, and then a motion to `p0` is specified using `pd` (`#5`) as the target. As with the initial motion, the time is explicitly specified; this is done to ensure that both motions will be completely symmetrical.

Notice that the entire motion sequence for the program is specified without waiting for any motions to complete; this is done only before the program exits (`#6`).

The function `pivotFxn()` rotates its transform by $60°$ about the y axis. It records the initial transform value during the first cycle of the motion; it senses whether the current cycle is the first cycle by calling `getActiveCount()` (`#7`), which returns 0 only during the first cycle of the motion. (We could also have used a static variable in this case, since `pivotFxn()` is used for only one motion.) The amount of rotation to apply is determined from the motion scale, which is found by using `motionScale()` and the current motion ID (`#8`).

5.9 The Joint Bias Functions

A recent addition to the RCCL tool kit is *joint biasing*, which is an extension of the concept of variable transforms to joint coordinates.

Each manipulator has a built in set of joint bias values, described by the `jbias` field of the `MANIP` structure. If the user calls the routine

```
jbiasSelect (mnp, on)
  MANIP *mnp;
  int on;
```

with the argument `on` set to true, then, for each subsequent motion request, the trajectory generator will add the contents of `jbias` to the joint values of the motion target. Joint biasing works only for joint interpolated motions. It may be deselected by calling `jbiasSelect()` with `on` set to 0.

Like a variable transform, the joint biases may be declared constant, variable, or bound to an evaluation function, through the following routines:

```
setJbiasConst (mnp)
   MANIP *mnp;

setJbiasVarb (mnp)
   MANIP *mnp;

jbiasMotionEval (fxn, arg, mnp)
   int (*fxn)();
   int arg;
   MANIP *mnp;
```

`setJbiasConst()` sets the joint biases to be *constant*, which means the trajectory generator uses a private copy of them, created at the time the motion request was issued. This is the default setting. `setJbiasVarb()` instructs the trajectory generator not to use a local copy of the joint biases, but rather to constantly track the values in `mnp->jbias`. `jbiasMotionEval()` binds `mnp->jbias` to the function `fxn` for the duration of the next motion request. The evaluation function is called by the trajectory generator using the call format

```
fxn (jbias, arg, mnp)
JNTS *jbias;
int arg;
MANIP *mnp;
{
        ...
}
```

`jbias` is the joint bias values, `arg` is the application argument provided through `jbiasMotionEval()`, and `mnp` is the manipulator containing the joint biases.

Joint biasing is particularly useful in applications involving control of the robot using joysticks or hand controllers. In these situations, it is often desirable to map the controller inputs into joint coordinates rather than into some Cartesian coordinates. Although one can sometimes accomplish this using functional transforms, the necessary kinematic computations are quite time consuming, and are not necessarily well defined.

5.10 Pausing the System

A couple of routines are provided to suspend all motions currently being executed:

```
rcclPause (accelTime, pauseTime)
  float accelTime, pauseTime;

rcclResume()
```

`rcclPause()` brings all the program's manipulators to rest, for a duration of `pauseTime` milliseconds; if `pauseTime` is equal to `F_UNDEF`, then the pause state will continue indefinitely until `rcclResume()` is called. The transition time used to bring the manipulators to rest can be specified in milliseconds by `accelTime`; otherwise, if this argument is given as `F_DEFAULT`, the system will compute the transition times automatically. As with the function `setTime()`, the specified transition time is actually one half the total acceleration time; i.e., it is equal to τ multiplied by the control cycle time (see Figure 14).

When the system is paused, all motion monitors and transform motion functions are suspended (i.e., those which are set up with `runMotionFxn()` and `transMotionEval()`. Permanent monitor and transform functions continue to execute (monitor functions will be discussed in Section 6.3.1).

> **WARNING: One should be careful when pausing the system with motions underway that depend on real-time sensor inputs. If large changes in the sensor inputs accumulate while the system is paused, then the dependent motion(s) might glitch when they are resumed.**

To help the programmer handle the problem stated above, there are two predicate functions available:

```
rcclIsPaused()

rcclWasPaused()
```

`rcclIsPaused()` returns true if the system is currently in a paused state. Monitor functions can use this and act accordingly. Similarly, `rcclWasPaused()` returns true if the system was paused during the *last* control cycle; this helps motion monitors or transform motion functions look out for discontinuities.

Another problem with "pause glitches" can occur with functions that depend explicitly on time. This can be overcome by using `rcclSysTime()`, which returns a time value which is frozen when the system is paused.

6. Interacting with the Environment

6.1 The Low Level RCI Robot Interface

The present version of RCCL is implemented on top of RCI (Real-time Control Interface), which is a software system that allows real time control tasks to be run in a UNIX environment. RCI also provides a simple joint-level robot and sensor I/O interface, which RCCL uses internally. Since parts of this interface may be useful to the RCCL programmer, it is described briefly here. A comprehensive description of everything in this section may be found in the *RCI User's Guide* and the *RCI Reference Manual*.

One reason a programmer may wish access to the RCI interface is to get at the sensor I/O interface. At the present time, RCCL does not specifically provide very much in the way of sensor I/O. The disadvantage of having not done this, however, is that the sensor I/O features that do exist (using RCI directly) are somewhat ad-hoc. Certain types of sensor inputs which are clearly robot-specific (such as force/torque measurements from joint or wrist sensors) may be formally incorporated into RCCL at a later time.

6.1.1 The `RCI_RBT` Structure

The RCI robot interface is centered around the `RCI_RBT` structure. Every robot being controlled by an RCI control task[1] is referenced through one of these. A pointer to the `RCI_RBT` structure is provided by the `rbt` field of the `MANIP` structure.

The `RCI_RBT` structure contains several fields, the most important of which are:

```
JLS *jls;
KYN *kyn;
void *var;
HOW *how
```

For convenience, pointers to each of these are also provided in the `MANIP` structure.

The `jls` field ("Joint Level Stuff") points to a data type which defines an extensive set of fixed joint level parameters for the robot. It is a grab-bag of all sorts of things such as the number of joints, the type of each joint, the gear ratios, joint value limits, maximum velocities, calibration information and vectors, desired nominal velocities, etc. This constitutes a database of joint level robot parameters.

The `kyn` field points to the robot's `KYN` structure, which contains various *kinematic* and *dynamic* level parameters for a particular robot.

The `KYN` structure is defined as follows:

[1] Recall that the trajectory generator is implemented using RCI control tasks.

```
typedef struct _KYN {
   GEN_KYN *gen;
   void *ext;
} KYN;
```

The gen field points to a substructure of type GEN_KYN, and contains parameters generic to most serial link robots. This includes such things as the "A" matrix parameters, link inertias and masses (currently being added), and joint friction coefficients. It also contains pointers to routines for doing robot specific kinematic and dynamic computations.

The ext field points to a data area which defines extra parameters specific to the robot's class. This includes precomputed terms for enhancing the efficiency of the kinematic computation functions. The type associated with this data area varies from one robot class to another, and takes a name of the form <CLASS>_KYN. The one for the PUMA robots, defined in puma_kynvar.h, is called PUMA_KYN. Notice that the ext field is declared to be void*, so that it may be handled independantly of the robot class.

The var field points to the robot's VAR structure, which is a "scratch pad" in which terms which are generated during a particular computation may be remembered for use in another computation. Like the ext component of the KYN structure, the actual definition depends on the robot class, and so it is generally referenced using a generic pointer. Typical things that are placed in the VAR structure include the sines and cosines of the joint angles and coefficients of the Jacobian. The VAR structure for PUMA robots is defined by the type PUMA_VAR. It is unlikely that this structure will be of much initial interest to an RCCL programmer.

The HOW structure is a blackboard containing robot state and sensor I/O information which is updated once every control cycle by the RCI interface. It defines a large number of fields, not all of which are necessarily implemented on a particular system. More information on HOW is given in Section 6.1.3.

More information on the JLS, KYN, and VAR structures can be found in the "RCI Reference Manual" and the "RCI User's Guide".

6.1.2 Other ways to get JLS, KYN, and VAR

It may be necessary to obtain pointers to the JLS, KYN, or VAR structures even in cases where the robot is not under RCCL control and so no MANIP structure has been allocated. This might happen, for instance, if one is using the kinematic and dynamic functions described in Section 6.2.1.

Pointers to the JLS and KYN structures can be obtained by calling the routines getJls() and getKyn() with the name of the robot in question:

```
#include <rci.h>

JLS *jls;
KYN *kyn;
int kynExtSize;

jls = getJls ("nameOfRobot");
```

```
kyn = getKyn ("nameOfRobot", &kynExtSize);
```

The second argument to `getKyn` is an optional integer pointer for returning the size of the structure associated with the `ext` field. Both of these routines, and the pointers they return, can only be used from the RCCL planning level.

To access `JLS` and `KYN` structures from the control level as well as the planning levels, the routines

```
rciGetJls (name)
rciGetKyn (name, sizep)
```

should be used instead. Before using these routines, the function

```
rciJlsKynInit (name)
```

must first be called from the planning level.

`VAR` structures must be allocated by the application program. The necessary size can be be determined using the routine `getVarSize()`:

```
void *var;

var = malloc (getVarSize("nameOfRobot"));
```

If called from the RCCL control level, it may be better to use `rciMalloc()` in place of `malloc()`. If the structure is to be shared by both the planning and control levels, then `allocMem()`, called from the planning level, should be used instead.

Once allocated, it is often necessary to instantiate the `VAR` structure with values for a particular robot position; see Section 6.2.1.

More information on these routines can be found in the "RCI Reference Manual" and the "RCI User's Guide".

6.1.3 The `HOW` Blackboard

A few of the `HOW` structure fields likely to be of immediate interest to an RCCL programmer are described here. Some which relate directly to the robot are

```
  int enc[MAXJNTS];
float encVel[MAXJNTS];
float jtorque[MAXJNTS];

 char torqueOK;
```

`enc` and `encVel` give the robot's current position and velocity, as read back from the robot controller, in terms of encoder counts. Angle level information is also available, but this can generally be obtained from the fields of the `MANIP` structure. `jtorque` gives the current force/torque values on each joint. The system must be explicitly asked to update the `jtorque` field (see the next section), but some systems (or robots!) do not support this capability. If `jtorque` is being updated, and the information in it is valid for the current cycle, the `torqueOK` field is set true.

Some fields related to sensor I/O include:

```
  long options;
  char numAdc;
  char numPioIn;
  char numPioOut;

  long pendantInput[HOW_PENDANT];
 float forceInput[6];
   int jstickInput[6];
   int adcInput[HOW_MAXADC];

ushort pioInput[HOW_MAXPIO];

  char pendantOK;
  char forceOK;
  char jstickOK;
  long adc;
```

`options` is a bit mask describing which of the various sensor I/O features are implemented. Codes of interest here include `TORQUE_AVAIL` (joint torque information is available), `PENDANT_AVAIL` (teach pendant information is available), `FORCE_AVAIL` (wrist force sensor information is available), `JSTICK_AVAIL` (input from a multi-DOF joystick is available). `numAdc` is the number of analog-to-digital converter channels available, `numPioIn` is the number of parallel input ports, and `numPioOut` is the number of parallel output ports. Input from the teach pendant, force sensor, joystick, and ADC channels can be turned on and off using special commands (see the next section). When valid data is read back from each of these, it is placed in the appropriate field `pendantInput`, `forceInput`, `jstickInput`, or `adcInput`, and the corresponding field `pendantOK`, `forceOK`, `jstickOK`, or `adc` is set valid. Parallel port input information is always available through the field `pioInput`.

`HOW` structure information may be read back from any part of an RCCL program, although there is currently no way to do this atomically (Section 5.5.2), unless the read is done from the control level on the same CPU that is running the trajectory generator for the manipulator.

The reason the `HOW` structure was implemented using shared memory was to make access to it very fast. Historically, this was very important because the original RCCL systems were implemented on VAXen, where the overhead associated with a subroutine call can be particularly gruesome. The inability to read information from it atomically could be resolved by double buffering, since the structure is updated at a regular interval. Both issues could, and probably should, be fixed by wrapping access to the `HOW` structure fields in macros. For more detailed information on the `HOW` structure, the reader should consult either the manual page or the *RCI User's Guide*.

6.1.4 Robot and I/O Commands

The "other half" of the RCI robot interface is a large set of command macros that control the robot and attached sensor devices. Like the HOW structure, these macros can be called from anywhere within the RCCL program. What each macro actually does is store the command information away in a private buffer that is flushed by the RCI interface once every control cycle. Each macro is called with a pointer to the RCI_RBT structure, along with whatever other arguments are appropriate. Since RCCL uses the command interface itself, care should be taken to avoid conflicts. Some of the commands likely to be of interest to an RCCL programmer and unlikely to conflict with the trajectory generator are:

```
SET_TORQUE_READ (rbt, on)
SET_FORCE_READ (rbt, on)
SET_JSTICK_READ (rbt, on)
SET_ADC_CHAN (rbt, mask)
PUT_PIO (rbt, num, value)
```

SET_TORQUE_READ() enables reading of the joint torque values into the jtorque field of the HOW structure. Likewise, SET_FORCE_READ() and SET_JSTICK_READ enable reading of the wrist/force torque sensor and the multi-DOF joystick into the forceInput and jstickInput fields of the HOW structure. SET_ADC_CHAN() sets a mask describing which ADC channels should be activated for reading into the adcInput field of the HOW structure. Finally, PUT_PIO places the bit pattern value into parallel output port num. The value which is set may be read back from the pioOutput field of the HOW structure.

6.2 Kinematic Computation Functions

6.2.1 Routine Descriptions

RCI provides routines for doing various types of kinematic and dynamic computations, such as forward and inverse kinematics, forward and inverse Jacobian mappings, etc. Each routine takes as one of its arguments a pointer to the robot's KYN structure. Some routines require pointers to the VAR and/or JLS structures as well. The following functions are currently defined:

```
fwdKinematics (t6,c,j6,kyn,var)
invKinematics (j6,t6,c,refj,kyn,jls,var)
solveConf (c,j6,kyn)

fwdJacob (dc,dj,j6,kyn)
invJacob (dj,dc,j6,kyn,epsilon)
fwdJacobT (t,f,j6,kyn)
invJacobT (f,t,j6,kyn,epsilon)

gravload (t,j6,kyn)
```

```
strToConfig (c,s,kyn)
configToStr (s,c,kyn)
```

`fwdKinematics()` computes `t6` and a configuration bit-code `c` from the joint values `j6`. It uses information provided by the `KYN` structure `kyn` and updates the sine/cosine information in the `VAR` structure `var` if that argument is not `NULL`. `invKinematics()` is the most complicated routine: it computes `j6` given `t6`, a configuration bit-code `c`, and a set of reference joint values (`refj`) that it uses to help resolve singularities and joint angle redundancies. It uses information in the `kyn` and `jls` structures and, if `var` is not `NULL`, updates the sine and cosine terms in that. `solveConf()` computes only the robot configuration `c` from the joint values `j6`.

The Jacobian routines each apply one of the four principal Jacobian calculations to an input vector. Each uses information in the `kyn` structure as well as the current robot joint values `j6`. Ecah routine computes, respectively,

$$\mathbf{dc} = \mathbf{J}\,\mathbf{dj}$$

$$\mathbf{dj} = \mathbf{J}^{-1}\,\mathbf{dc}$$

$$\mathbf{t} = \mathbf{J}^{T}\,\mathbf{f}$$

$$\mathbf{f} = \mathbf{J}^{-T}\,\mathbf{t}$$

The variable `dc` is of type `DIFF` and `f` is of type `FORCE`. `dj` and `t` are simply vectors of floats.

The routine `gravload()` computes the gravity loading torques `t` given a `kyn` structure and the robot's joint position `j6`. No other dynamics routines are implemented at this time.

`strToConfig()` takes a string representation `s` of the robot's configuration and returns the corresponding bit code `c`. `configToStr()` performs the inverse operation.

Arguments for the above routines which correspond to sets of joint values are defined to be of type `float*`. This is because the "level" at which these routines are defined is below that of RCCL and the definition of the `JNTS` data type. When using a `JNTS` data type with these routines, the `v` field should be used for the argument.

More detailed information about these routines, and what they do in the event of errors, can be found by consulting their respective entries in the "RCI Reference Manual".

Some of these routines make use of terms, such as the sines and cosines of the joint angles, or entries in the Jacobian matrix, which are constant for a particular robot position. It may be computationally more efficient to compute these terms only once if several routines are called for a specific robot position. Because of this, some routines have equivalent routines, whose names end in `Var`, which take the robot `VAR` structure as an argument in place of the joint values. These routines currently are:

```
fwdJacobVar (dc,dj,var,kyn)
invJacobVar (dj,dc,var,kyn,epsilon)
fwdJacobTVar (t,f,var,kyn)
invJacobTVar (f,t,var,kyn,epsilon)
gravloadVar (t,var,kyn)
```

When these routines are called, the `VAR` structure must be valid for the current robot position. The `VAR` structure associated with the robot's `mnp` structure is automatically maintained by the RCCL trajectory generator, and so will be valid whenever the trajectory generator is running. Programs should not modify the contents of this particular `VAR` structure.

Otherwise, if the `VAR` structure is allocated by the user, then it is necessary to explicitly update it for a given joint position. There are two routines for doing this:

```
updateVar (var,j6,kyn)
updateVarXsincos (var,j6,kyn)
```

`updateVar()` computes all the information in the `VAR` structure based on a given set of joint values `j6` and information in the `KYN` structure. `updateVarXsincos()` does the same thing, but does not compute the sine and cosine terms that may optionally be set by the routines `fwdKinematics()` and `invKinematics()`. It is only necessary to update the `VAR` structure when the robot position changes (which typically happens once per control cycle). The following call sequence would be valid:

```
updateVar (var,j6,kyn);
fwdJacobVar (dc,dj,var,kyn);
gravloadVar (load,var,kyn);
```

All of these functions are implemented internally by routines specific to each particular robot class. Pointers to the implementation routines are contained in the `KYN` structure itself. The internal implementation routine for a particular robot class takes the same name as the external routine, only with the first letter capitalized and the robot class name added in front. For example, the routine to implement `fwdKinematics()` for the PUMA is called `pumaFwdKinematics()`.

These routines are meant to be somewhat "robot independent". In fact, there is still some robot dependency with regard to the inverse routines, since inverses are only well defined for robots with six degrees of freedom or less. For robots with more degrees of freedom, the interface functions would probably need to be generalized to allow the specification of strategies for resolving the redundancy.

6.2.2 Example Program

The following stand alone program illustrates how the above routines can be used directly to compute inverse kinematics for robots supported by RCI.

```
#include <rccl.h>
#include <rci.h>

char **argvec;

main(argc, argv)
int argc;
char *argv[];
{
```

```
        float ang[MAXJNTS];
        float roll, pitch, yaw;
        unsigned long confcode;
        char confstr[9];
        int status;
        TRSF t6;
        JLS *jls;
        KYN *kyn;

        argvec = argv;
        argc--; argv++;
        if (argc != 1)
         { usage (-1, "%s <robot-name>\n", argvec);
         }
        if ((jls = getJls (*argv)) == NULL)
         { printErrors ();
           exit (-1);
         }
        if ((kyn = getKyn (*argv, (int*)0)) == NULL)
         { printErrors ();
           exit (-1);
         }
        while (1)
         { if (scanf ("%f%f%f%f%f%f%s", &t6.p.x, &t6.p.y, &t6.p.z,
                       &roll, &pitch, &yaw, confstr) != 7)
            { exit (0);
            }
           rpyToTrsf (&t6, roll, pitch, yaw);
           strToConfig (&confcode, confstr, kyn);
           status = invKinematics (ang, &t6, confcode, (float*)0,
                                   kyn, jls, (void*)0);
           if (status & (KYN_JOINT_LIMIT | KYN_CANT_REACH))
            { printf ("Out of range\n");
            }
           else
            { radianToDegree (ang, ang, jls);
              printVf ("joints: %9.3f\n", ang, jls->numJ);
            }
         }
}
```

The program gets the name of the robot from the argument list. It then calls `getJls()` and `getKyn()` to obtain pointers for the robot `JLS` and `KYN` structures. The rest of the program is a loop which reads in a Cartesian position and a desired robot configuration, and applies the inverse kinematic routine. The Cartesian position is described by x, y, and z translations and by roll, pitch, and yaw rotations, which are converted to the `TRSF` structure `t6`. The desired robot configuration is read as a string and is converted to a bit code using `strToConfig()`. Both `t6` and the configuration code are handed to `invKinematics()`, which tries to solve for the joint values and return them in `ang`. The fourth argument to `invKinematics()` is optional and is omitted here. If specified, it gives a set of joint values that are used to resolve joint angle redundancies (caused by the fact that a joint can have more than $360°$ of range). If omitted, the joint mid-range values are used instead (these are obtained from the `JLS` structure).

`invKinematics()` returns a status value which is used to see if the routine succeeded. Although this example does not do so, a status value that indicates failure can be analysed to yield detailed information about what went wrong. If the inverse kinematics routine succeeds, then the revolute joint values are converted to degrees using `radianToDegree()` and the result is printed out.

6.3 Control Level Routines and the Trajectory Generator

6.3.1 Monitor Functions

In addition to transform functions, an RCCL program can establish *monitor* functions, which are also executed by the trajectory generator, and can do things ranging from general sensor computations to the monitoring of cancellation conditions for motions.

There are two types of monitor functions, similar in concept to the two types of transform functions: *permanent* monitor functions, which, once created, are executed until they are explicitly deleted, and *motion* monitor functions, which are executed only for the duration of a particular motion.

Permanent monitor functions are created and deleted with the routines

```
runMonitorFxn (mnp, fxn, arg)
  MANIP *mnp;
  int (*fxn)();
  int arg;

deleteMonitorFxn (mnp, id)
  MANIP *mnp;
  int id;
```

`runMonitorFxn()` instructs the trajectory generator to call `fxn` once every control cycle, with the application argument `arg`, in the following format:

```
fxn (arg, mnp)
int arg;
MANIP *mnp;
{
   ...
}
```

Function execution begins immediately (provided the trajectory generator is running). `runMonitorFxn()` returns an ID value which can be used to cancel the monitor function with `deleteMonitorFxn()`. Both `runMonitorFxn()` and `deleteMonitorFxn()` require a pointer to a manipulator; this specifies which trajectory task should be used to run the monitor (recall that for multiple CPU systems, one trajectory task is created per CPU, and each `MANIP` is controlled by a particular task). A program may run up to `MAX_MONITOR_FXNS` monitors at once.

A motion monitor function is run for the duration of a particular motion. It may be established with the call

```
runMotionFxn (mnp, fxn, arg)
  MANIP *mnp;
  int (*fxn)();
  int arg;
```

The trajectory generator will then execute `fxn` once every control cycle for the time span of the next requested motion on `mnp`. The monitor is called using the same arguments as for permanent monitors.

```
fxn (arg, mnp)
int arg;
MANIP *mnp;
{
   ...
}
```

Execution starts at the beginning of the transition into the motion, and ends at the beginning of the transition out of the motion. Up to `MAX_MONITOR_FXNS` motion monitors may be specified for any given motion.

Monitor functions may communicate with other parts of the program using the same shared memory primitives that are available to the rest of the system (section (5.5.1)).

6.3.2 Trajectory Generator Computation Sequence

When an application is created where several different transform and monitor functions are executing together, it may be important to know the order in which the various executions take place. This section describes the computation sequence used by the trajectory generator for each control cycle.

We will describe the default behavior first (see Figure 25). Each trajectory task begins by exchanging data with all the robots which it is controlling: for each robot, state and sensor information is read in and the trajectory setpoints and commands which were computed and accumulated during the *previous* control cycle are written out. The rendezvous operation which follows the robot communication will be described momentarily. Next, all the control computations are performed: for each of the task's *manipulators*[2] all the permanent monitors are called, followed by all the mo-

[2]We say manipulator instead of robot because it is possible to create virtual manipulators which are not attached to any real robot. This is described in Section 8.2.

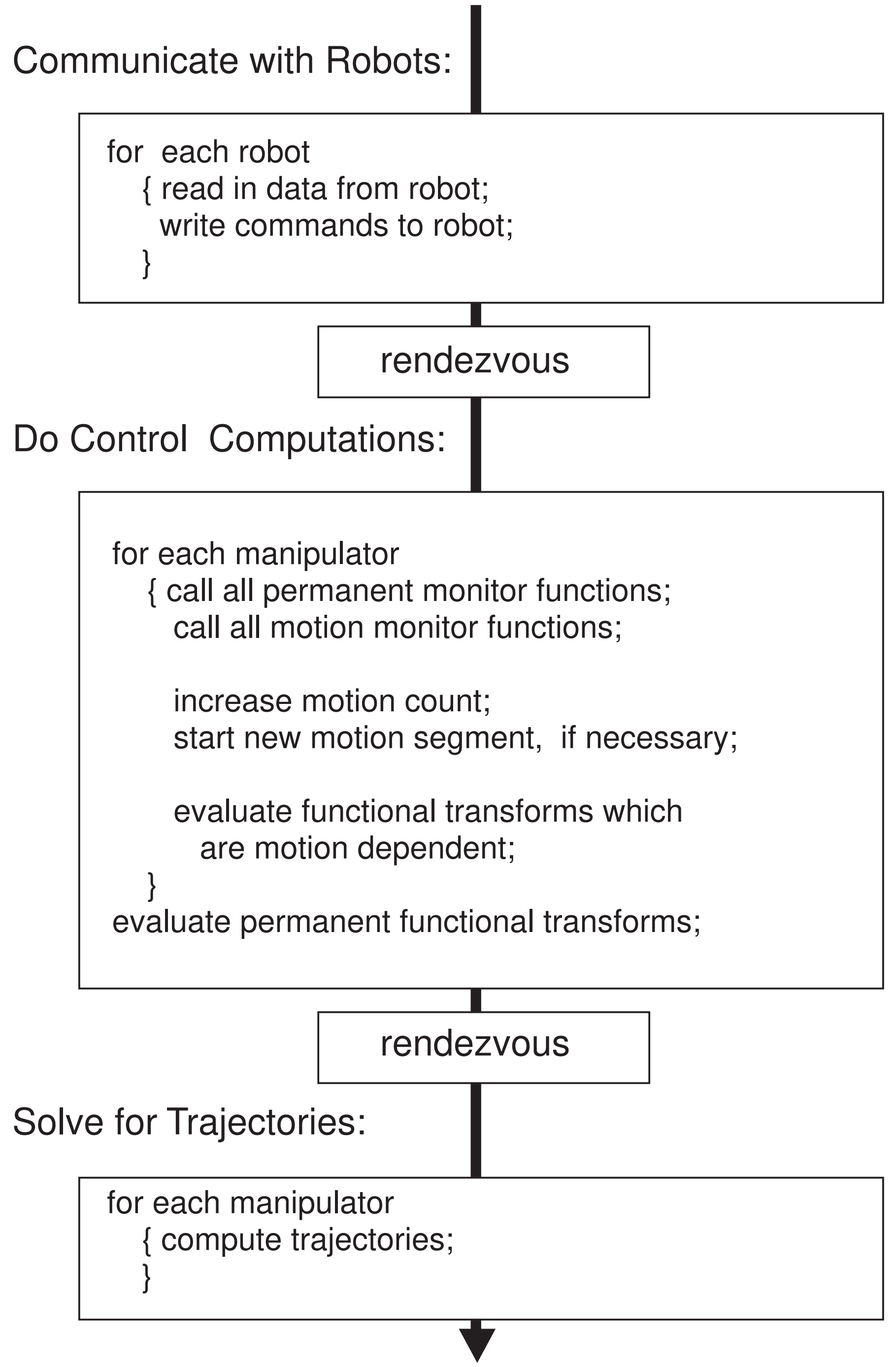

Figure 25: Default sequence of operations carried out by each trajectory task, once per control cycle.

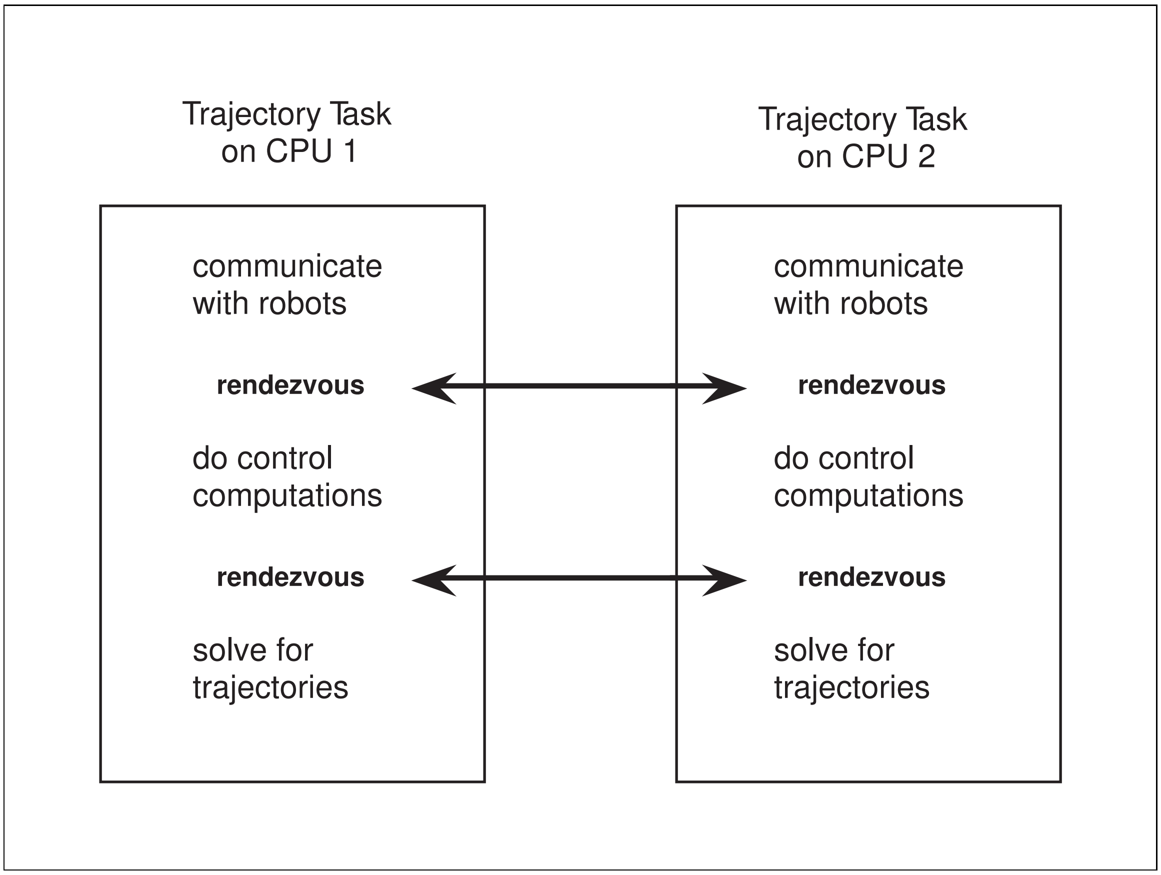

Figure 26: Rendezvous points between trajectory tasks executing on separate CPUs.

tion monitors. The current motion count is then increased, and if it is time to start a new motion request, that is done and the motion count is set to zero. All the functional transforms associated with the current (and perhaps new) motion segment (the `transMotionEval()` bindings) are then evaluated. The permanent functional transforms are evaluated afterwards. Finally, the trajectory generator combines all this information and solves for the output setpoints of each manipulator.

On multi-CPU implementations of the trajectory generator, one trajectory task executes on each CPU, and the associated computations run in parallel. This may cause some ambiguities about what was computed when. To offset this problem, all the trajectory tasks rendezvous with each other at two prescribed points during each control cycle (Figure 26). The first rendezvous takes place after each robot's state and sensor data has been read in, and ensures that each task will see the same state and sensor information for each robot. The second rendezvous takes place after all the application computations have been performed. At this point, all the information associated with moving motion targets has been computed, and the rendezvous ensures that the same values will be seen by each task.

In cases where several monitor or transform functions are defined simultaneously, they are evaluated in the same order that they were set up in. In other words, for the following example,

```
extern foo();
extern bar();
```

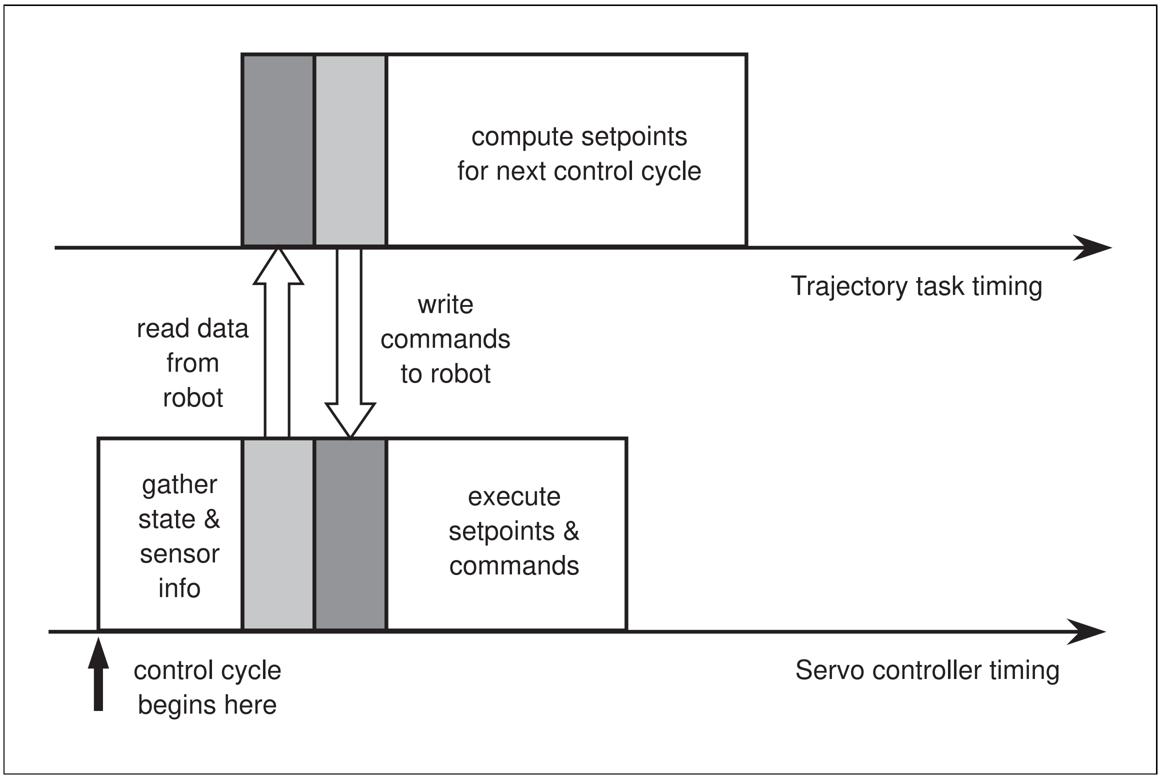

Figure 27: Default sequence used by the trajectory task for communicating with the robots.

```
runMonitorFxn (mnp, foo, 0);
runMonitorFxn (mnp, bar, 0);
```

monitor function `foo` will be executed before function `bar`. For the transform motion functions set up with `transMotionEval()`, the order of evaluation is based on the transform's counterclockwise location within the position equation, starting with the transform closest to **T6**.

The reader will notice that by default, the trajectory task reads data in from the robot, writes out setpoints and commands from the previous cycle, and then computes the setpoints for the next control cycle. This is done to allow parallel operation with the low level robot controller. Typically, the robot controller collects the robot state and sensor information, sends this to the trajectory generator, and receives back a set of setpoints and commands which it then executes (Figure 27). The computation of the next setpoint values by the trajectory task can then overlap with the execution of the current setpoint values by the robot controller. This can save time, particularly if the robot controller is slow (which was true historically). However, it also introduces a one-cycle delay in the trajectory computation process, which can be detrimental in feedback situations where the setpoints depend on the state and sensor information.

It is possible to have the robot setpoints sent to the controller at the **end** of each control cycle, rather than at the beginning of the next control cycle. To do this, the routine

```
rcclTightControlLoop (on)
  int on;
```

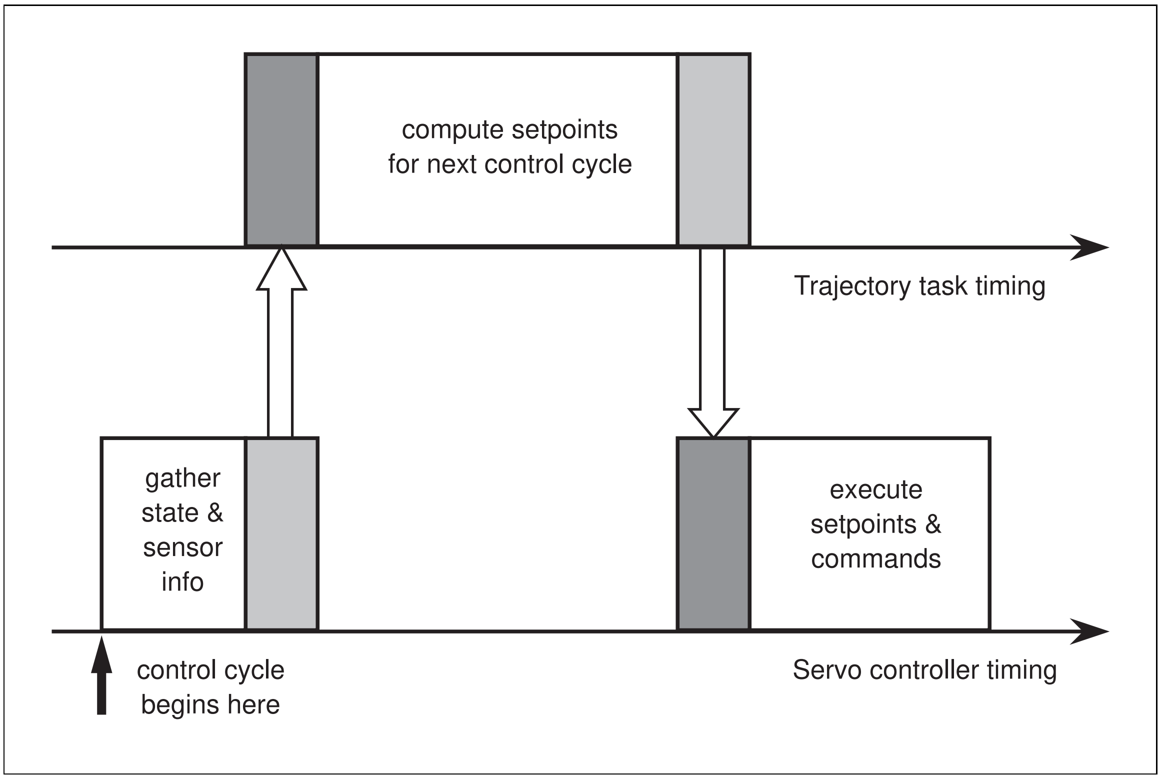

Figure 28: Robot communication sequence with no control lag.

should be called, with `on` set true, at some point when the trajectory generator is switched off. Setpoints will then be sent to the robots during the same cycle that they are solved for, as shown in Figure 28.

6.4 Sensor Integration

An important feature of RCCL is its ability to easily combine control of the robot with a variety of sensor inputs. The robot behavior can be influenced by modifying a motion's target positions or canceling the motion altogether. If the target position is modified in advance of the motion, this is called "presetting the world model"; such sensor integration occurs at the task level. Initializing a transform is a special case of this. If the target position is modified while the motion is in progress, this is called "tracking". If sensor monitors are set up to cancel a motion on certain conditions, this is called a "guarded motion". Executing a guarded motion usually implies some uncertainty about the world model; correspondingly, if the guarded motion makes "contact", then the environment is known with greater certainty than before, and it may be desirable to update the world model accordingly.

6.4.1 Task Level Integration

This section will present a demonstration program that simulates the integration of RCCL with a computer vision system. Assume that a camera has been attached to link 4 of a PUMA robot. The computer vision system is simulated by a function `snapshot()`, which causes a picture to be taken of the scene and stored, and by a function `getobj()`, which extracts the position and orientation of an object in the camera coordinate frame. The task is programmed in such a way that the "processing" done by `getobj()` overlaps with the motion of the arm. The strategy used consists of moving the manipulator toward a position where the object is expected to be in the camera's field of view. The program is synchronized so that the picture is taken at a given point in the trajectory. At the instant the picture is taken, the manipulator's **T6** value is recorded, and from this the camera coordinate frame can be computed. Knowledge of the camera coordinate frame, plus the object position returned in camera coordinates, provides all the information necessary to approach the object, grasp it, and take it somewhere else.

```
#include <rccl.h>
#include <fastmath.h>
#include "manex.560.h"

TRSF *computeU5();

int numberOfObjects;

main()
{
        TRSF_PTR z, e, cam, o, coord, expect, drop;
        POS_PTR look, get, put;
        MANIP *mnp;
        TRSF t6r, u5;
        JNTS rcclpark, j6r;
        char *robotName;
        int putId, lookId, getId;

        numberOfObjects = 5;

        rcclSetOptions (RCCL_ERROR_EXIT);

        robotName = getDefaultRobot();
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }

        z = allocTransXyz (NULL, UNDEF, 0.0, 0.0, ZBASE);
        e = allocTransXyz (NULL, UNDEF, 0.0, 0.0, TOOLZ);
        cam = allocTransRot (NULL, UNDEF, 50.0, 0.0, 0.0, xunit, 90.0);
        expect = allocTransRot (NULL, UNDEF, 400.0, 400.0, 700.0,
                                      xunit, 180.0);
        drop = allocTransRot (NULL, UNDEF, -400.0, 400.0, 700.0,
                                      xunit, 180.0);
        o = allocTrans (NULL, UNDEF);
        coord = allocTrans (NULL, UNDEF);

        look = makePosition (NULL, z, T6, e, EQ, expect, TL, e);
        get = makePosition (NULL, T6, e, EQ, coord, cam, o, TL, e);
        put = makePosition (NULL, z, T6, e, EQ, drop, TL, e);

        mnp = rcclCreate (robotName, 0);
        rcclStart();

        setSpeed (mnp, 2.0);                                    /*#1*/

        OPEN_HAND(mnp);

        for ( ; ; )                                             /*#2*/
         { lookId = move (mnp, look);
           waitFor (motionScale(lookId) > 0.7)                  /*#3*/
```

```
            snapshot();                                                    /*#4*/
            readTrans (mnp->t6, &t6r);                                     /*#5*/
            readMem (mnp->j6, &j6r);
            computeU5 (&u5, j6r.v[4], j6r.v[5]);                           /*#6*/
            multRiTrsf (coord, &t6r, &u5);

            if (!getobj(o))
             { break;
             }

            distance (mnp, "dz", -50.0);                                   /*#7*/
            move (mnp, get);

            getId = move (mnp, get);
            stop (mnp, 500.0);

            distance (mnp, "dz", -50.0);
            move (mnp, get);

            waitForStop (getId);                                           /*#8*/

            CLOSE_HAND (mnp);

            move (mnp, put);
            putId = stop (mnp, 500.0);
            stop (mnp, 500.0);
            waitForStop (putId);

            OPEN_HAND (mnp);                                               /*#9*/
          }

        movej (mnp, &rcclpark);
        stop (mnp, 1000.0);
        waitForCompleted (mnp);

        rcclRelease (1);
}

snapshot()
{
        printf ("\nTAKING PICTURE\n");
}

getobj(t)
TRSF_PTR t;
{
        extern double frandom();
        double rand;

        /* fake a camera coordinate reading */
        rand = frandom (-20.0, 20.0);
        xyzToTrsf (t, rand, rand, 150.0+rand);
```

```
        rpyToTrsf (t, 90.0, -25.0+rand, 0.0);
        printTrsf ("Object is at %m\n", t);

        return (numberOfObjects--);
}

TRSF *computeU5 (u5, ang5, ang6)
TRSF *u5;
float ang5, ang6;
{
        /* Compute U5 for a PUMA 560 (old coordinates) */

        float s5, c5, s6, c6;

        SINCOS (ang5, &s5, &c5);
        SINCOS (ang6, &s6, &c6);

        u5->n.x =  c5 * c6;
        u5->n.y =  s5 * c6;
        u5->n.z =  s6;
        u5->o.x = -c5 * s6;
        u5->o.y = -s5 * s6;
        u5->o.z =  c6;
        u5->a.x =  s5;
        u5->a.y = -c5;
        u5->a.z =  0.0;
        u5->p.x =  0.0;
        u5->p.y =  0.0;
        u5->p.z =  0.0;

        return (u5);
}
```

NOTE – this example has been coded for the PUMA 560 robot, and lives at `pickAndDrop.560.c` *in* `demo.rccl`*. An equivalent program for the PUMA 260 is contained in* `pickAndDrop.260.c`

The program makes use of three position equations: the position `look`, where the object is expected to be seen by the camera, the position `get`, at which the object should be grasped, and the position `put`, where the object should be placed.

The position equation `look` expresses the ordinary transform graph shown in Figure 29. **Z** is a reference frame at the base of the manipulator pedestal, **E** is the usual end effector frame, and **EXPECT** is the location, with respect to the pedestal base, where we expect to find the object in the camera's field of view.

The position equation `put` also expresses an ordinary transform graph (Figure 30). **DROP** is the location, relative to the manipulator base, where the object is to be dropped.

The position equation `get` expresses a slightly more complicated transform graph (Figure 31). It uses the additional transforms **COORD**, which locates the base frame to which the camera is mounted (in this case link 4 of the arm), **CAM**, which describes the camera view frame relative to

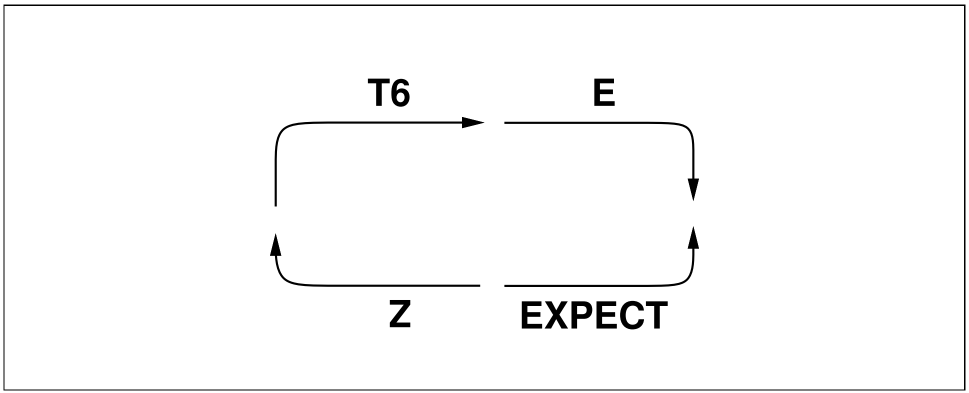

Figure 29: Transform graph for position "look".

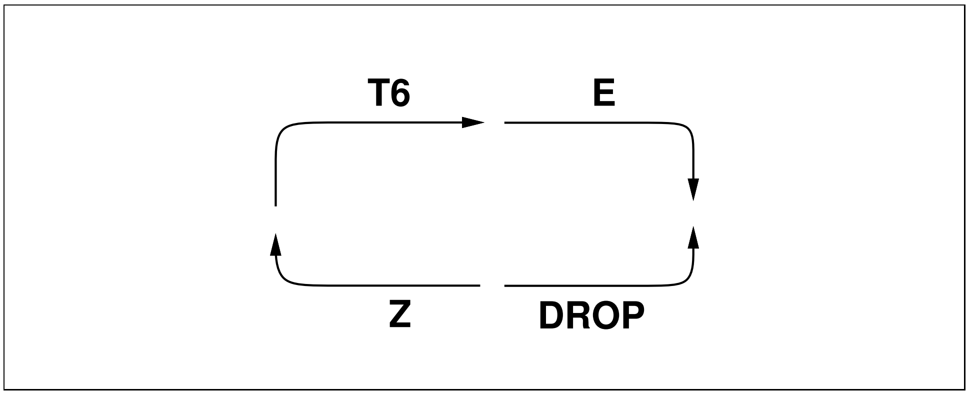

Figure 30: Transform graph for position "put".

the camera base frame, and **O**, which describes the position of the object in relative to the camera view frame.

Once all the component transforms of `get` are known, the manipulator may be moved to the pickup location. **CAM** and **E** are given, and **O** is returned by the "vision software" inside `getobj()`. How do we find **COORD**? Note that `get` is really just a subgraph of the (nicely complicated) graph shown in Figure 32.

The camera is mounted on link 4 of the robot, and so **COORD** is equivalent to the manipulator's $\mathbf{T4}$ transform, which is the product of the robot's first 4 "A" matrices (see Section 3.1). $\mathbf{T4}$ may be cumbersome to compute directly, but this is not necessary: instead, we can compute the transform $\mathbf{U5}$ which maps from link 4 to link 6 (and is the product of the robot's last two "A" matrices. From the graph it is obvious that

$$\mathbf{COORD} = \mathbf{T6}\ \mathbf{U5}^{-1}$$

$\mathbf{U5}$ can be computed quite easily given joint angles 5 and 6; the program does this using the routine `computeU5()`. Notice that the sines and cosines of the two angles are computed using `SINCOS()`, which is part of the RCI *fastmath* library.

Looking now at the program itself, we see that it increases the manipulator speed for dramatic effect (`#1`), opens the hand, and then enters into a loop where it will "pick and drop" objects until

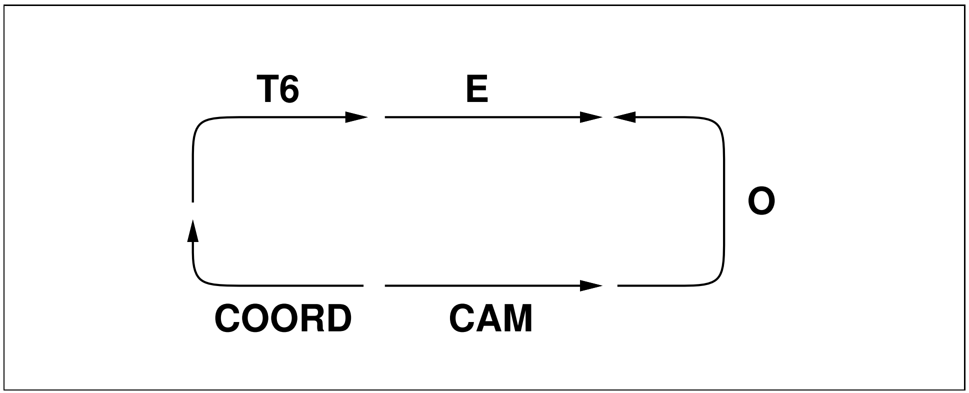

Figure 31: Transform graph for position "get".

`getobj()` says that it can't find any more.

Once the robot has been told to move to `look`, the program waits on the corresponding motion ID until the robot is 70% of the way there (`#3`). At this point the camera is told to take a picture (`#4`), and the current **T6** and joint angles values are read back from the manipulator structure (`#5`). `readTrans()` and `readMem()` are used to guarantee that the data is read atomically [3]. **U5** is computed using the read back angle values and the recorded **T6** value is post multiplied by its inverse (`#6`) to get **COORD**. Finally, `getobj()` is called to either report the object position or indicate that it couldn't find anything. Notice that `getobj()` has the remainder of the travel time to the target `look` in which to compute the value of **O**. While it is unlikely that a complicated vision function could profit much from this extra time, this does illustrate how the asynchronous nature of the motion requests can be put to use.

The rest of the loop (`#7`) issues the commands for the robot to approach the object, grasp it, and take it away and drop it. Notice how the opening and closing of the hand is coordinated with the motion requests using motion IDs (`#8`, `#9`).

The function `getobj()` constructs a simulation of **O** using a random number generator and a few well chosen transform manipulator routines.

As a side note, the reader might notice that we have not assigned names to any of the position equations or transforms in the program, and this will be true of the remaining examples as well. Names for these are only necessary when they are used by calling either `getPositionByName()` or `getTransByName()`. Memory objects will also be unnamed for the same reason. The use of names in the previous examples was for illustrative purposes only.

6.4.2 Tracking

This next example illustrates *tracking*, which is what happens when functionally defined transforms are updated based on sensor readings. The program itself is very simple: the xy translation values of a particular transform are made to follow the readings from a dual-channel potentiometer.

[3] although it will **not** guarantee that **T6** and the joint angles are consistent with respect to one another; however, this is unlikely to be a problem because the robot is not moving very fast. If it did present a problem, we could grab only the joint angle values, and compute **T6** using the forward kinematics routines.

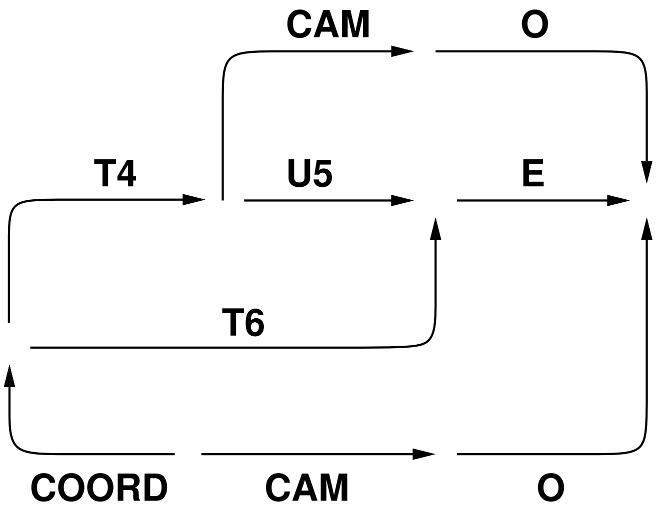

Figure 32: More complex description of position "get".

When connecting sensor inputs directly to transform values, one typically uses either position tracking, in which the transform coordinates are constructed directly from the input values, or velocity tracking, in which the inputs define an incremental offset which is applied to the existing transform value. Position tracking has the advantage that no errors are accumulated, but the resulting motion can be very rough unless some sort of smoothing is applied. Velocity tracking is much smoother but can result in drifting unless the input device has a well defined dead band. The demo program here uses absolute position tracking.

planning level module

```
#include <rccl.h>
#include "manex.560.h"

main ()
{
        TRSF_PTR p, track;
        POS_PTR p1;
        MANIP *mnp;
        TRACK_CTRL_BLK *tcb;

        extern trackFxn();
```

```
        rcclSetOptions (RCCL_ERROR_EXIT);

        tcb = ALLOC_MEM (NULL, TRACK_CTRL_BLK, UNDEF);          /*#1*/

        p = allocTransRot (NULL, UNDEF, P_X, P_Y, P_Z, xunit, 180.0);
        track = allocTrans (NULL, UNDEF);

        p1 = makePosition (NULL, T6, EQ, p, track, TL, T6);

        mnp = rcclCreate (getDefaultRobot(), 0);
        rcclStart();

        tcb->compute = YES;
        tcb->gain = 1.0;
        tcb->vlimit = 400.0;
        tcb->mnpAddr = (void*)mnp;

        transEval (track, trackFxn, (int)tcb);                  /*#2*/
        rcclBlock();

        move (mnp, p1);                                         /*#3*/
        stop (mnp, F_UNDEF);

        while (1)                                               /*#4*/
         { printf ("%7.4f %7.4f\n", tcb->xval, tcb->yval);
           delay (500.0);
         }
}
```

control level module

```
#include <rccl.h>
#include <fastmath.h>
#include "manex.560.h"

float clip (delta, maxDelta)                                    /*#5*/
float delta;
float maxDelta;
{
        return (delta >= 0.0 ? MIN(delta, maxDelta) : -MIN(-delta, maxDelta));
}

trackFxn (t, arg, mnp)
TRSF *t;
int arg;
MANIP *mnp;
{
        char *fxnName = "trackFxn()";

        TRACK_CTRL_BLK *tcb;
```

```
        RCI_RBT *rbt;
        float maxDelta;
        float delta;
        float x, y;
                                                                /*#6*/
        if ((tcb = (TRACK_CTRL_BLK*)getMemByAddr (arg)) == NULL)
         { rciAbort (0, "%s -- Can't find paramter block\n", fxnName);
           return;
         }
        rbt = (getManipByAddr(tcb->mnpAddr))->rbt;             /*#7*/

        if (tcb->tracking)
         { getInput (rbt, &x, &y);
           tcb->xval = x*tcb->gain;
           tcb->yval = y*tcb->gain;                             /*#8*/

           maxDelta = FABS(tcb->vlimit) * rcclGetInterval() / 1000.0;
           t->p.x += clip (tcb->xval - t->p.x, maxDelta);
           t->p.y += clip (tcb->yval - t->p.y, maxDelta);       /*#9*/

           if (!tcb->compute)                                   /*#10*/
            { turnInputOff (rbt);
              tcb->tracking = 0;
            }
         }
        else
         { if (tcb->compute)                                    /*#11*/
            { turnInputOn (rbt);
              tcb->tracking = 1;
            }
         }
}

static float faket;

turnInputOn(rbt)
RCI_RBT *rbt;
{
        if (rbt->how->numAdc > 0)
         { SET_ADC_CHAN (rbt, 0x3);                             /*#12*/
         }
        else
         { rciPrintf ("will fake the input\n");                 /*#13*/
           faket = 0.0;
         }
}

turnInputOff(rbt)
RCI_RBT *rbt;
{
        if (rbt->how->numAdc > 0)
         { SET_ADC_CHAN (rbt, 0x0);                             /*#14*/
         }
```

```
}

getInput(rbt, x, y)
RCI_RBT *rbt;
float *x, *y;
{
        if (rbt->how->numAdc > 0)
         { if (rbt->how->adc & 0x3)
            { /* input bias of 1024 is specific to McGill site */
              *x = rbt->how->adcInput[0] + 1024;
              *y = rbt->how->adcInput[1] + 1024;                /*#15*/
            }
         }
        else
         { /* fake the input ... draw an ellipse */
           faket += rcclGetInterval()/1000.0;
           *x = 100*COS(PI*faket);
           *y = -50*SIN(PI*faket);
         }
}
```

NOTE – this example has been coded for the PUMA 560 robot and lives at `track.560.c` *and* `trackCtrl.560.c` *in* `demo.rccl`. *An equivalent program for the PUMA 260 is contained in* `track.260.c` *and* `trackCtrl.260.c`. *If the program discovers that the system does not have the correct sensor inputs, the tracking function just punts and simulates them.*

The program uses only a very simple position equation `p1`, containing a base transform **P**, which is set to correspond to the robot's starting position, and a tracking transform **TRACK**. The structure `TRACK_CTRL_BLK` (#1) is used to communicate between the planning level and the tracking function. It contains the following control fields: `compute`, which "turns the function on", `gain`, a constant by which the sensor input values are multiplied, `vlimit`, a tracking velocity threshold (in millimeters per second), `mnpAddr`, the planning level address of the `MANIP` structure which is used to reference the `MANIP` structure from the control level, and `tracking`, a field which is used internally by the tracking function. The fields `xval` and `yval` are set by the tracking function to the current x and y target values being tracked. The tracking function is set running with a call to `transEval()` (#2). This is followed by a call to `rcclBlock()`, so that we know that the tracking function has executed at least once. The robot is then requested to move to the tracking position and stay there indefinitely while the planning task reads and prints out the x and y coordinates as determined by the tracking function (#4). There is no explicit quit command for this program; the usual interrupt character will do quite nicely[4].

Most of this program's action takes place in the tracking function. Because absolute position tracking is implemented, the function `clip` is used to limit the maximum displacement that can occur during any particular cycle (#5).

The first thing `trackFxn()` does is use `getMemByAddr()` to get a pointer to the control memory structure. It then obtains a pointer to the `MANIP` structure using `getManipByAddr()`, and uses this

[4]Notice that this means we do not call `rcclRelease()` at the end of the program. While this is certainly "cleaner" to do if possible, it is not mandatory.

get a pointer to the RCI_RBT structure (#7) (Section 6.1.1), which will be used to obtain and control sensor inputs.

The state variable tcb->tracking indicates whether the tracking function is active or not. If tracking is active, then getInput() is called to get the raw x and y values from the sensor, which are then scaled by tcb->gain and written into the xval and yval fields of the control block (#8). The maximum permitted per-cycle displacement is computed from the velocity threshold and the RCCL sample interval, and is used to clip the final x and y values which are set in the transform. If the tcb->compute has been cleared, then input sensor values are turned off with a call the turnInputOff() (#10). If the function is not tracking, then it waits for tcb->compute to be set by the planning level, then turns sensor inputs on with turnInputOn().

The sensor I/O is encapsulated in separate routines. This was done mainly for show, but also because the sensor input code is a bit of a hack and we didn't want to clutter up trackFxn() with it. turnInputOn() instructs the RCI robot interface to start reading analog-to-digital converter channels 0 and 1 (#12), or, if there are no ADC channels, announces that the sensor input will be faked (#13). turnInputOff() closes the channels (#14). Finally, getInput() either reads back the ADC values from the HOW structure (#15) or fakes them with an ellipse pattern (#16).

The velocity limiting algorithm used in this example is kept very crude for simplicity. Of course, more elaborate smoothing techniques, or even general control laws, can be used as required by the application.

The original version of RCCL contained an example program in which a linear potentiometer was attached to the robot tool tip and was used to do depth tracking along surfaces. To accomplish this, a very simple tracking function fingerFxn() was defined which adjusted the z coordinate of its transform using the following code:

```
fingerFxn (tr, arg, mnp)
TRSF *tr;
int arg;
MANIP *mnp;
{
        HOW *how = mnp->how;

        tr->p.z + 100.0 * how->adcInput[CHAN] * GAIN;
}
```

If the sensor inputs are very slow or the corresponding computations are very lengthy, one has the choice of either spreading the tracking computation over several control cycles, or doing the tracking from the planning level. Planning level tracking is tricky mainly because the planning level does not have the real-time priority of the control level (and can even be suspended for short periods of time by UNIX). However, the reader might find one of the following paradigms interesting:

1. Make the tracking transform variable and set it from the planning level at some reasonable rate. Presumably the changes are not large.

```
TRSF change;
TRSF tmp;
```

```
...

setTransVarb (track);
readTrans (track, &tmp);        /* initialize */

...

setTime (mnp, F_DEFAULT, F_UNDEF);
move (mnp, pt);                 /* semi-permanent motion */
while (!quit)
 { getSensorInput (&change);
   multTrsf (&tmp, &tmp, &change);
   writeTrans (track, &tmp);
   delay (REASONABLE_INTERVAL);
 }
```

2. A more reasonable method than the one above would be to issue many short motion requests from the planning level. This is preferable because then the trajectory generator will interpolate and provide smoothing between successive data points.

```
setTransConst (track);

...

setMotionQueueSize (mnp, 1);
while (!quit)
 { getSensorInput (&change);
   multTrsf (track, track, &change);
   move (mnp, pt);
 }
```

`track` is set constant since each move request will then make a private copy of it. Setting the motion queue size to 1 prevents queue saturation and ensures that each call to `move()` will block until the requested motion is being executed. One problem with this method is that the first few cycles of any motion incur a large amount of overhead (to compute the drive and transitioning parameters). This effect will be greatly reduced if the path is naturally smooth enough that the motion transition times can be set to zero. A small fixed delay within the loop would also help reduce the frequency of the motions.

6.4.3 Guarded Motions

The next example uses a monitor function to implement guarded motions. The basic guarded motion paradigm is very simple: a monitor function waits until some particular condition is raised, and then cancels the motion. The planning level can examine the *stop code* associated with the motion to see whether or not the cancellation actually occurred[5].

When doing guarded motions, it is often desirable to be able to determine the value of a particular transform (within the context of a specified position equation) at the instant of contact (or whatever condition is being tested for). This computation has to be done by the trajectory generator, because an unacceptable length of time may elapse between when the guarded motion ends and when the planning task learns of it and tries to record the appropriate data. Assume that we want to know the value of a transform `t0` in the context of some position equation `p0`. What we want to be able to do is ask the trajectory generator to call `solveTrans()` for `t0`, using position `p0`, at the end of a particular motion. The function

```
updateTrans (mnp, tr, p0, t0, tx)
  MANIP *mnp;
  TRSF_PTR tr, t0, tx;
  POS_PTR p0;
```

will do just this. The evaluation is done at the *end* of the next requested motion on the manipulator `mnp`. The last four arguments to `updateTrans()` are equivalent to the four arguments of `solveTrans()`. Usually, the position `p0` will contain a **T6** transform, which we will want to be instantiated with the current value of the manipulator's `t6` field. The corresponding call looks like this:

```
updateTrans (mnp, tr, p0, t0, mnp->t6);
```

All of the transform arguments to `updateTrans()` must be allocated using the `allocTrans()` routines.

The sample program below simulates a situation in which the manipulator supposedly has a contact sensor with which it tries to find the locations of two points on a planar surface. When each motion finishes, the current manipulator position at the contact point is remembered and used afterwards (if both motions achieved contact) to trace out a line along the surface between the two points.

planning level module

```
#include <rccl.h>
#include "manex.560.h"

extern int touchFxn();

main()
{
        TRSF_PTR target, e, contact[2];
```

[5] This information could also be returned by the application using shared memory, but motion stop codes are explicitly incorporated into RCCL, so they may as well be used. Stop codes exist mainly to make it possible to (in the future) implement a feature where different *reflex* motions are executed depending on the stop code returned by the previous motion.

```
TRSF start[2];                                              /*#1*/
POS_PTR p0;
MANIP *mnp;
JNTS rcclpark;
char *robotName;
int mid;
int bothTouched;
int i;

rcclSetOptions (RCCL_ERROR_EXIT);
robotName = getDefaultRobot();
if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
 { printf ("position 'rcclpark' not defined for robot\n");
   exit(-1);
 }

target = allocTrans (NULL, UNDEF);
e = allocTrans (NULL, UNDEF);
contact[0] = allocTrans (NULL, UNDEF);
contact[1] = allocTrans (NULL, UNDEF);

p0 = makePosition (NULL, T6, e, EQ, target, TL, e);      /*#2*/

mnp = rcclCreate (robotName, 0);
rcclStart();

setJvelScale (mnp, 2.0);
movej (mnp, &rcclpark);
waitForCompleted (mnp);

xyzToTrsf (e, 0.0, 0.0, TOOLZ);                             /*#3*/
solveTrans (&start[0], p0, target, mnp->t6);                /*#4*/
solveTrans (&start[1], p0, target, mnp->t6);
multTrsfXyz (&start[0], -750.0, 150.0, 0.0);
multTrsfXyz (&start[1], -750.0, 350.0, 0.0);

/* turn on 1st ADC channel, *if* we have one */

if (mnp->how->numAdc > 0)
 { SET_ADC_CHAN (mnp->rbt, 0x1);                            /*#5*/
 }
else
 { printf ("no real ADC channel; going to fake it\n");
 }

bothTouched = 1;
mid = 0;
for (i=0; i<2; i++)                                         /*#6*/
 { *target = start[i];
   move (mnp, p0);

   setMod (mnp, 'c');
   setCartVel (mnp, 10.0, F_DEFAULT);                       /*#7*/
```

```
        setMotionFlag (mnp, ++mid);                           /*#8*/
        runMotionFxn (mnp, touchFxn, mid);
        updateTrans (mnp, contact[i], p0, target, mnp->t6);  /*#9*/
        distance (mnp, "dz", 200.0);
        move (mnp, p0);                                       /*#10*/

        setTime (mnp, 0.0, F_DEFAULT);
        distance (mnp, "dz", -20.0);
        move (mnp, mnp->last);                                /*#11*/

        setMod (mnp, 'j');
        *target = start[i];
        move (mnp, p0);                                       /*#12*/

        waitForStop (mid);                                    /*#13*/
        if (motionStopCode(mid) == 1)                         /*#14*/
         { printf ("Motion %d touched\n", i+1);
         }
        else
         { printf ("Motion %d did not touch\n", i+1);
           bothTouched = 0;
         }
     }
    waitForCompleted (mnp);

    if (bothTouched)                                          /*#15*/
     { printf ("playing back ...\n");

        setMod (mnp, 'c');
        setCartVel (mnp, F_DEFAULT, F_DEFAULT);
        *target = start[0];
        move (mnp, p0);
        *target = *contact[0];                                /*#16*/
        target->p.z += 5.0;
        move (mnp, p0);
        *target = *contact[1];
        target->p.z += 5.0;
        move (mnp, p0);
        *target = start[1];
        move (mnp, p0);
        waitForCompleted (mnp);
        setMod (mnp, 'j');
     }
    movej (mnp, &rcclpark);
    stop (mnp, 1000.0);
    waitForCompleted (mnp);

    rcclRelease (YES);
}
```

control level module

```
#include <rccl.h>
#include "manex.560.h"

#define THRESHOLD 500

touchFxn(mid, mnp)
int mid;
MANIP *mnp;
{
        HOW *how;
        static int count = 0;
        how = mnp->how;                                        /*#17*/

        /* *if* we have an ADC channel, use it for sensor readings.
           Otherwise, fake the touch function using the motion
           scale value */

        if (how->numAdc > 0)
         { if (how->adc & 0x1 && how->adcInput[0] > THRESHOLD)
             { stopMotion (mid, 1);                            /*#18*/
             }
         }
        else
         { if (motionScale(mid) > (count == 0 ? 0.2 : 0.6))    /*#19*/
             { stopMotion (mid, 1);
               count++;
             }
         }
}
```

NOTE – this example has been coded for the PUMA 560 and lives at `guarded.560.c` *and* `guardedCtrl.560.c` *in* `demo.rccl`*. An equivalent program for the PUMA 260 is contained in* `guarded.260.c` *and* `guardedCtrl.260.c`*. If the program discovers that the system does not have right sensor inputs, then the monitor function just punts and aborts the guarded motions at different points along their trajectories.*

The program is a bit verbose, but fairly straightforward. To illustrate a different way of specifying motion targets, it only defines one position equation `p0` (`#2`):

$$\mathbf{T6\ \ E} \ = \ \mathbf{TARGET}$$

E is the usual end-effector transform and **TARGET** is a transform which is set to a specific target location for each motion. This can be done since all motion requests are to fixed locations. Some transforms can then be declared as regular variables, since they will be used only within the planning task (`#1`). The transforms pointed to by `contact` are allocated using `allocTrans()` since they will be used as arguments to `updateTrans()`. All of the transform values are instantiated starting at (`#3`), after the robot has been moved to its initial position. **E** is set to the canonical end-effector values. The locations in `start` describe the points at which the guarded motions will begin; they are defined simply by a translational offset from the initial value of **TARGET** in the position `p0` (`#4`).

The touch sensor input is assumed to come from channel 0 of the analog-to-digital converters supplied by the RCI robot/sensor interface. If ADC channels are available, the program enables this channel at (`#5`). It will be read by the monitor function.

The guarded motions are set up by the loop at (`#6`). Moving the manipulator to a particular location is done by setting **TARGET** to the desired location and then queueing a move request to `p0`. The first move is to the start position. The next move is the guarded motion, which involves setting up a few things. Cartesian interpolation is specified. A slow velocity is set in anticipation of contact (`#7`). An explicit motion ID is requested, using `setMotionFlag()` (`#8`); this is so that the motion ID will be known in advance and can therefore be passed as an argument to the monitor function `touchFxn()`, which is established with a call to `runMotionFxn()`. To store the location at the end of the guarded motion, `updateTrans()` asks the trajectory generator to solve `p0` for **TARGET** at the end of the motion and place the result in one of the transforms pointed to be `contact` (`#9`). Finally, the guarded motion itself is implemented by requesting a move to a target offset from `start` by 200.0 millimeters along the z axis (`#10`). Since the guarded motion presumably makes contact with a hard surface, we wish to follow it with a quick "jump" back up the z axis by a small amount. This is accomplished using a relative motion, created with another z offset applied to the position `mnp->last` (`#11`). The transition time is set to 0 so that the manipulator will pull away from the surface immediately; otherwise, the deceleration associated with the transition would carry the robot farther into the surface. A transition time of 0 is tolerable since the approach velocity is low. After the pullback motion, the robot switches to joint mode and moves back to the starting point (`#12`). The program itself uses a motion ID to wait for the guarded move to complete (`#13`), and then examines the stop code (which was set by the monitor function) to determine if contact was actually achieved (`#14`). A stop code of 1 indicates contact; otherwise, the system will have set the stop code to `ON_NORMAL` when the nominal endpoint was reached.

If both guarded moves make contact with the surface, the program has the robot draw a line, at normal velocity, 5.0 millimeters above and between the two contact positions (`#15`). The robot is moved to the first start point, the first contact point, the second contact point, and back to the first start point. The 5.0 millimeter vertical displacement is achieved by simply adding an offset to the contact target positions (`#16`).

The monitor function `touchFxn()` is quite simple. It begins by fetching a pointer to the robot's `HOW` structure (`#17`; see Section 6.1.3) from which it reads ADC channel 0. The motion is canceled if this channel displays a value greater than `THRESHOLD` (`#18`). If the system does not have any ADC channels, the monitor function fakes it by canceling the motion when it is either 20% or 60% done (`#19`).

6.5 Teaching Positions

6.5.1 The Teach Routine

RCCL provides a teach routine, `rcclTeach()`, which allows a user to move the arm around using a combination of keyboard and teach pendant commands:

```
rcclTeach (mnp, prompt, tool)
  MANIP *mnp;
  char *prompt;
  TRSF *tool;
```

`rcclTeach()` assumes that the trajectory generator is currently active (*i.e.*, `rcclStart()` has been called. When it returns (at the command of the operator), the robot's new position can be read from the appropriate fields in the `mnp` structure (such as `t6`, `here`, `lastTC`, or `j6`; the first three fields will all be equal). If an error occurs such that the teach routine was unable to successfully complete, it returns -1 and an appropriate error code is placed on the error stack (Section 9.2.1). The argument `prompt` defines the prompt used for keyboard input. `tool` is an optional argument that specifies a transform applied to the T6 frame to define a TOOL frame. If `tool` is given as `NULL`, then the T6 frame is used as the TOOL frame.

`rcclTeach()` allows the operator to specify motions in three principal coordinate systems: *joint*, *tool*, and *world*. These are completely analogous to their counterparts in the Unimation language VAL. In *joint* coordinates the robot is moved by specifying individual joint angles. In *tool* coordinates the robot is moved by specifying Cartesian translations and rotations relative to the TOOL frame. In *world* coordinates, the robot is moved by specifying Cartesian translations relative to the manipulator base frame and rotations relative to the manipulator base frame translated so that its origin is coincident with that of the TOOL frame.

6.5.2 Keyboard Commands

The keyboard commands are implemented using the C-tree matcher (see Section 4.10): the terminal displays a prompt and the user enters commands. Typing `?` gives either a menu of valid commands or instructions as to what should be entered next. Hitting `<space>` automatically matches what the user has typed against the offered commands and expands the input line accordingly.

`return` – Causes `rcclTeach()` to return with a value of 0.

`record` – Causes `rcclTeach()` to return with a value of 1. This can be useful when the teach routine is executing in a loop; a return value of 1 can be used to signal that we want to record the current point (and perhaps call `rcclTeach()` again to record more points).

`stop` – Stops the arm "dead in its tracks".

`show position` –
Prints the manipulator's current position in both Cartesian and joint coordinates.

`show tool` – Prints the specified value of the *tool* transform.

`show t6*tool` –
Prints the location of the TOOL frame relative to the manipulator base frame.

`world <dof> by <value>` –
Moves the indicated degree of freedom by a certain value in *world* coordinates. Translations are specified by a `<dof>` of `x`, `y`, or `z`, and a `value` in millimeters. Rotations are specified by a `<dof>` of `rx`, `ry`, or `rz`, and a `value` in degrees.

`tool <dof> by <value>` –
Moves the indicated degree of freedom by a certain value in *tool* coordinates. Values are specified the same way as for world coordinates.

`joint <num> by <value>` –
Moves the indicated joint **by** a specified value (degrees for rotational joints and millimeters for prismatic joints).

`joint <num> to <value>` –
Moves the indicated joint **to** a specified value (degrees for rotational joints and millimeters for prismatic joints).

`goto joints` –
Moves all the joints of the robot to a specified set of values.

`set speed` – Sets a scale value for the motion speed (the default value is 1.0). The default velocities used by `rcclTeach()` are the current values set by `setCartVel()` and `setJointVel()`. Setting the speed within the teach routine will not change the speed settings in the program when the routine returns.

`set translation <x> <y> <z>` –
Moves the manipulator so that the x, y, and z translational coordinates of the TOOL frame (relative to the manipulator base frame) are set to a particular value (in millimeters).

`set rotation euler <phi> <the> <psi>` –
Moves the manipulator so that the rotational component of the TOOL frame (relative to the manipulator base frame) is described by a particular set of Euler angles (degrees).

`set rotation rpy <roll> <pitch> <yaw>` –
Moves the manipulator so that the rotational component of the TOOL frame (relative to the manipulator base frame) is described by a particular set of roll-pitch-yaw angles (degrees).

`set tool <x> <y> <z> <roll> <pitch> <yaw>` –
Explicitly sets the *tool* transform to the specified translational values (millimeters) and roll-pitch-yaw angles (degrees).

`set integration [on | off]` –
Enables or disables servo level integration; when integration is enabled, joint servo errors are reduced to zero, but the manipulator may become more "frisky". This command is (at the moment) specific to Unimation/PUMA robots.

`hand <value>` –
Sets the manipulator hand to a particular value (for open/close grippers, 1 usually means "open" and 0 usually means "closed").

6.5.3 Pendant Commands

The commands from the (PUMA) teach pendant are now described.

- The buttons *joint*, *tool*, or *world* set the default coordinate system for subsequent teach pendant commands.

- The button *free* places the manipulator into a zero-gravity mode, with all joints limp and their gravity loadings compensated for; the operator is then free to move the arm around into any position desired.

  WARNING: this command requires that the robot's gravity model is known and properly calibrated. Otherwise, the the robot might fall under its own weight. Proceed with caution at first.

- The switch on the side of the pendant controls the speed of motion; it has three settings: fast, medium, and slow. The fastest setting is the momentary one.

- The *tick* button enables "ultra-slow" pendant motions; this is for fine position adjustment.

- The *clamp1* and *clamp2* buttons (usually) open and close the gripper.

- The *REC* button causes the teach routine to return with a value of 1, as though the keyboard command `record` had been entered. This is useful if the operator is teaching a whole set of points and does not wish to return to the terminal.

- The joint control buttons (1 to 6) do different things depending on the mode. In *tool* or *world* mode, they cause relative translations or displacements: the first three buttons create $\pm$ motions along the x, y, and z axes, while the last three buttons create $\pm$ rotations about the x, y, and z axes. In *joint* mode, each button controls its corresponding joint. In *free* mode, the $(+)$ side of each button "locks" its corresponding joint, so that it is no longer limp, while the $(-)$ side limps it again.

6.5.4 Programming with the Teach Routine

`rcclTeach()` may be embedded inside applications and called whenever it is desirable for an operator to move the arm around explicitly, either for teaching new positions or just for the fun of it.

This is best illustrated by the following sample program, in which the teach routine is used to record a sequence of positions (and hand settings), and then "play them back".

```
#include <rccl.h>
#include <ctree.h>                                                  /*#1*/

char *key_buf[256];

#define MAXPOINTS       256

int numPoints = 0;
JNTS points[MAXPOINTS];                                             /*#2*/
int handSet[MAXPOINTS];

main()
{
        char *keytree, *tree_match_parse(), *tree_match_err_at();
        float speed = 1.0;
        MANIP *mnp;

        numPoints = 0;
                                                                    /*#3*/
        keytree = tree_match_parse("                                        \
(       quit                                                                \
        set                                                                 \
         (speed                                                             \
           (%f", &speed, ",0,10,        \"speed scale (1.0 nominal)\")  \
         )                                                                  \
        show                                                                \
        record                                                              \
        playback                                                            \
)",0);
        if (keytree == 0)
         { printf ("ERROR/Parse error at %.30s\n", tree_match_err_at());
           exit (-1);
         }

        if ((mnp = rcclCreate (getDefaultRobot(), 0)) == NULL)  /*#4*/
         { printErrors();
           exit (-1);
         }
        if (rcclStart() < 0)
         { printErrors();
           exit (-1);
         }
```

```
        do
         { tree_match (keytree, "PLAYBACK> ", key_buf);          /*#5*/

           if (COMMAND ("set speed"))                            /*#6*/
            { setSpeed (mnp, speed);
            }
           else if (COMMAND("show"))                             /*#7*/
            { printf ("speed = %g\n", speed);
              printf ("%d points stored\n", numPoints);
            }
           else if (COMMAND ("record"))                          /*#8*/
            { int status;

              numPoints = 0;
              printf ("use command \"return\" to stop recording points\n");
              while ((status = rcclTeach(mnp, "RECORDING> ", NULL)) == 1)
               { points[numPoints] = *mnp->j6;                   /*#9*/
                 handSet[numPoints] = mnp->handPos;
                 if (++numPoints >= MAXPOINTS)
                  { break;
                  }
               }
              if (status < 0)
               { printErrors();                                  /*#10*/
                 exit(-1);
               }
            }
           else if (COMMAND ("playback"))                        /*#11*/
            { int i;

              for (i=0; i<numPoints; i++)
               { movej (mnp, &points[i]);
                 if (handSet[i] != mnp->handPos)
                  { waitForCompleted (mnp);
                    if (handSet[i])
                     { OPEN_HAND(mnp);
                     }
                    else
                     { CLOSE_HAND(mnp);
                     }
                  }
               }
            }
         }
        while (!(COMMAND("quit")));

        rcclRelease(1);
}
```

NOTE – this example has been coded for the PUMA robot and lives at `playback.c` *in* `demo.rccl`.

Like the teach routine itself, this program uses the C-tree matcher. A brief description of the C-tree matcher is given for the example program in section (4.10). Detailed information is provided by the document `doc/CtreeMatch.doc`.

Definitions relevant to the C-tree matcher are included with the file `<ctree.h>` (`#1`). The taught positions are stored (in joint coordinates) in the array `points`, with the hand position settings in the array `handSet` (`#2`). Binary (open/close) hand settings are assumed.

The call to `tree_match_parse()` (`#3`) sets up the keyboard commands that will be accepted by the Ctree-matcher. These commands include

```
quit
set speed <value>
show
record
playback
```

Since the program does not set the option `RCCL_ERROR_EXIT`, it checks the routines `rcclCreate()` and `rcclStart()` explicitly for error conditions; if errors are detected, the error information is printed by `printErrors()` (see Section 9.2.1) and the program exits (`#4`).

Commands are read in by the routine `tree_match()` (`#5`), in a loop, until the command `quit` is entered.

The `set speed` command (`#6`) calls `setSpeed()` to set the speed with which the program will play back the recorded positions

The `show` command (`#7`) prints out the current playback speed setting and the number of playback points which are currently stored.

The `record` command (`#8`) allows the operator to teach a set of points, which is done by calling `rcclTeach()` in a loop. The teach routine will return each time the operator enters the `record` command from the keyboard, hits the `REC` button on the teach pendant, or enters the command `return`. In the first two cases, the routine will return 1, which the program uses as a signal to record the current point and call the teach routine again (`#9`). The joint values of each recorded point are read from the `j6` field of the `MANIP` structure. The hand settings are read from the `handPos` field of the `MANIP` structure. If `rcclTeach()` returns a negative value, this indicates an error has occurred, and the program prints out information about the error and exits (`#10`).

The `playback` command (`#11`) plays back all points which are currently recorded. This is done simply by calling `movej()` for each recorded point, and, if the hand position for that point is different from the current hand position, waiting for the move to finish and setting the new hand value appropriately.

6.6 Logging Data

In RCCL programs, it is often necessary to log data created or read in at the control level. For instance, an experimenter working with force control algorithms might want to record, for every control cycle, the force sensor values and the current position of the robot.

The convenient thing to do in these cases would be to write the data directly to a file. Unfortunately, the disk accesses associated with file I/O can be quite time consuming, and may cause unacceptable delays if invoked from the control level. This can be mitigated somewhat by using the buffered file I/O routines of `<stdio.h>`, but even then disk accesses will have to occur whenever the buffers become full.

A more robust approach is to write the data into an intermediate buffer (implemented using shared memory) which the *planning level* flushes to a file when it gets full. This is not very difficult to do, and once written, the actual code can be applied to most similar situations. We have written a short program demonstrating this.

planning level module

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

#include <rccl.h>
#include <stdio.h>
#include <signal.h>
#include "logger.h"

int logfd;
DATABUF *dataBuf;
extern logJ6Values();

logHandler()                                                    /*#1*/
{
        writeLogFile (logfd, dataBuf->bufnum^1);
}

writeLogFile (fd, num)                                          /*#2*/
FILE *fd;
int num;
{
        if (write (fd, dataBuf->buf[num], dataBuf->count[num]) < 0)
         { perror ("can't write to log file");
           exit (-1);
         }
}

main()
{
        MANIP *mnp;
        int fxnId;

        rcclSetOptions (RCCL_ERROR_EXIT);
        mnp = rcclCreate (getDefaultRobot(), 0);
        rcclStart();

        signal (SIGUSR1, logHandler);                           /*#3*/
        if ((logfd = open ("log", O_RDWR|O_CREAT, 0666)) < 0)   /*#4*/
```

```
          { perror ("can't open log file");
            exit (-1);
          }
        dataBuf = ALLOC_MEM (NULL, DATABUF, UNDEF);             /*#5*/
        dataBuf->bufnum = UNDEF;
        fxnId = runMonitorFxn (mnp, logJ6Values, (int)dataBuf); /*#6*/

        delay (10000.0);

        deleteMonitorFxn (mnp, fxnId);                          /*#7*/
        rcclBlock();
        writeLogFile (logfd, dataBuf->bufnum);                  /*#8*/
        close (logfd);
        rcclRelease(YES);
}
```

control level module

```
#include <rccl.h>
#include <signal.h>
#include "logger.h"

logJ6Values (arg, mnp)
int arg;
MANIP *mnp;
{
        DATABUF *db;
        char *str;
        int num;
        int length;
        static int count = 0;

        if ((db = (DATABUF*)getMemByAddr((void*)arg)) == NULL)
          { rciAbort (0, "can't get data buffer memory\n");
          }
        if (db->bufnum == UNDEF)                                /*#9*/
          { db->bufnum = 0;
            db->count[0] = 0;
            count = 0;
          }
        num = db->bufnum;
        str = &db->buf[num][db->count[num]];                    /*#10*/
        sprintf (str, "%d %8.3f %8.3f %8.3f %8.3f %8.3f %8.3f\n",
                       count++,
                       RADTODEG*mnp->j6->v[0], RADTODEG*mnp->j6->v[1],
                       RADTODEG*mnp->j6->v[2], RADTODEG*mnp->j6->v[3],
                       RADTODEG*mnp->j6->v[4], RADTODEG*mnp->j6->v[5]);
        db->count[num] += (length = strlen(str));               /*#11*/
        if ((db->count[num] + length) >= BUFSIZE)               /*#12*/
          { num ^= 1;
```

```
            db->count[num] = 0;
            db->bufnum = num;
            rciSignal (SIGUSR1);                                /*#13*/
          }
}
```

NOTE – this example has been coded for all systems, and lives at logger.c and `loggerCtrl.c` *in* `demo.rccl`.

All this program does is start the trajectory generator, and then turn on a monitor function which logs the current joint setpoints. The monitor writes the log data into a buffer, and when the buffer is full, it sends a signal to the planning task, which then writes the contents out to the log file. A double buffer system is used, as described in Section 5.5.3. The buffer itself is implemented using a shared memory object defined as follows:

```
#define BUFSIZE 4096

typedef struct {
          int bufnum;
          char buf[2][BUFSIZE];
          int count[2];
} DATABUF;
```

`bufnum` is the index of the buffer currently being written to by the monitor function. The double buffer itself is the `buf` element. The field `count` describes how much data is in each of the two buffers. In general, the size of the buffer (`BUFSIZE` in this case) should be selected so that UNIX has time to catch the "buffer full" signal and write all the data out to a file before the second buffer fills up. What this number is depends on how much data is being logged, the control sample rate, and the UNIX system itself. To be safe, one should probably make the buffer big enough to accommodate 1 second worth of data, and also should try to avoid writing so much as to cause the system to choke. Logging about 10 Kbytes per second should certainly cause no difficulties, and more is certainly possible.

When the planning task receives the "buffer full" signal, it calls the signal handler `logHandler()` (`#1`), which in turn calls `writeLogFile()` (`#2`) to write out the buffer. The second argument to `writeLogFile()` is the number of the buffer which should be written. This is set to the buffer opposite from `bufnum`, that is, the buffer not currently being written to by the control level.

We will now look at the program itself. `SIGUSR1` is used to indicate "buffer full"; `logHandler()` is bound to this signal at (`#3`). The log file is opened at (`#4`) and the data buffer structure is allocated at (`#5`). `bufnum` is set to `UNDEF` to tell the monitor function it has to initialize the data structure. Once everything has been set up, the monitor function is started at (`#6`), the program naps for 10 seconds, and then the monitor function is deleted (`#7`). We delay for a control cycle to make sure that we have caught the last "buffer full" signal. Then, the rest of the buffer that was being filled at the time the monitor was turned off is written out (`#8`).

The logging function itself is fairly straightforward. It does some initialization if the `bufnum` field is set to `UNDEF` (`#9`). It then sets the variable `str` to the point in the buffer where it left off the last time (`#10`), and prints information there using `sprintf()`. The buffer counter is increased by the size of the string that was created (`#11`), and the string length is also used to decide if there is enough buffer space for another cycle (`#12`)[6]. If there is not, we switch buffers and signal the planning task (`#13`).

If space is very tight, one does not have to use an ASCII data format as we have done here. Binary formats are several times more compact (and usually much faster), although they are less portable and generally require a program interface to read.

A programmer doing a lot of work involving data logging would probably want to encapsulate the logging code presented here into a small set of routines.

[6]This particular code might break if the string is unexpectedly longer during the next cycle; the necessary checks were omitted from this example for simplicity.

7. Force Control and Motion Limit Detection

In assembly tasks, objects are required to be brought into contact and motions have to be stopped when the collision occurs. Once objects are in contact the task is said to be *constrained* because arbitrary motions are no longer possible in every direction. When the task is constrained, the arm must exert forces on objects and can no longer be purely position servoed for all six degrees of freedom. Instead, the degrees of freedom (DOFs) along or about which the constraints occur must *comply* to the interaction forces. When moving to and from contact situations, guarded motions are usually performed. The limit conditions for these guarded motions are typically force thresholds (when contact is anticipated), and possibly trajectory error thresholds (when the robot is complying but there is a possibility it might "slip off" the contact surface).

Multi-RCCL does not have an explicit built-in capability for performing compliant motions or monitoring specific motion limit conditions. The original version of RCCL did provide a simple force algorithm based on the old Paul and Shimano method ([Paul and Shimano 1976]), but this does not work well on robots which lack joint level force sensors, and even when such sensors are available, it is approximate and unstable. Old RCCL also provided built-in checking for force and displacement limits, which allowed automatic cancellation of a motion when either observed forces or displacements from the desired trajectory exceeded specified threshold values.

Explicit support of this capability was dropped because it required making too many assumptions about what sensors were available and what algorithms might be suitable for different applications. This does not restrict the overall functionality of the system, however, since both compliant motion and the limit checking can be easily implemented by application code using monitor functions and functionally defined transforms. In fact, the ease with which such functionality can be embedded within RCCL has been a principal reason for its popularity. The RCCL sites at GE/ATL and JPL have both implemented their own force control strategies within RCCL, based, in both cases, on position accommodation techniques such as those described by [Maples and Becker 1986]. Example force control programs written at JPL are included at the end of this section.

As a convenience for the programmer (and for possible future extensions) Multi-RCCL still provides *interface* routines which allow force control and limit specifications to be described for particular motions, and read back when necessary by application code responsible for their implementation. These routines will be discussed below and their use illustrated by example programs.

7.1 Limit Specification Routines

Force or displacement limits for the next motion can be specified by the routines

```
limit (mnp, format, value1, value2, ... )
  MANIP *mnp;
  char *format;
  float value1, value2, ...
```

```
setForceLimit (mnp, mask, flimit)
  MANIP *mnp;
  unsigned long mask;
  FORCE *flimit;

setDispLimit (mnp, mask, dlimit)
  MANIP *mnp;
  unsigned long mask;
  DIFF *dlimit;
```

`limit()` uses a format string, containing a set of two letter codes separated by white space, to state what types of limits to check for. Translational force limits (along the principal axes) are indicated by the codes `fx`, `fy`, and `fz`. Torque limits (about the principal axes) are indicated by the codes `tx`, `ty`, and `tz`. Likewise, displacement limits along or about the principal axes are indicated by `dx`, `dy`, `dz`, `rx`, `ry`, or `rz`. For each given limit, the routine takes an additional argument describing the associated threshold value.

For instance, a force limit of 10.0 Newtons along the x and y axes, and a displacement limit of $30°$ about the z axis, could be specified as

```
limit (mnp, "fx fy rz", 10.0, 10.0, 30.0);
```

A more "machine oriented" way of describing the same thing is provided by the routines `setForceLimit()` and `setDispLimit()`. `setForceLimit()` uses a bit mask to specify the degrees of freedom along (or about) which force limits should be monitored; the limit value for each selected DOF is obtained from the corresponding field in the argument `flimit`. The bit mask should be assembled from the following codes:

```
ALONG_X
ALONG_Y
ALONG_Z
ABOUT_X
ABOUT_Y
ABOUT_Z
```

Displacement limits are set in the same way using `setDispLimit()`. The sample usage of `limit()` given above could be coded using these routines as:

```
FORCE flimit;
DIFF dlimit;

flimit.f.x = 10.0;
flimit.f.y = 10.0;
setForceLimit (mnp, (ALONG_X|ALONG_Y), &flimit);
dlimit.m.z = 30.0;
setDispLimit (mnp, ABOUT_Z, &dlimit);
```

Limit specifications are made by the planning level and are valid only for the duration of the next requested motion.

Since RCCL does not actually implement the limit detection, some part of the application program (a monitor function most likely) will have to read back the limit specifications. This is done with the routines

```
getActiveForceLimit (mnp, mask, flimit)
  MANIP *mnp;
  unsigned long *mask;
  FORCE *flimit;

getActiveDispLimit (mnp, mask, dlimit)
  MANIP *mnp;
  unsigned long *mask;
  DIFF *dlimit;
```

`getActiveForceLimit()` returns (through `mask`) a bit mask describing the force limits for the motion currently active on `mnp`. The associated thresholds are returned in the corresponding fields of `flimit`. Similarly, `getActiveDispLimit()` returns the currently active displacement limits.

A simple monitor function that checks for displacement limits could be written as follows:

```
dispMonitor (arg, mnp)
int arg;
MANIP *mnp;
{
        unsigned long mask;
        DIFF limit;
        TRSF disp;
        float rz, ry, rx;

        getActiveDispLimit (mnp, &mask, &limit);

        if (mask)
         { multRiTrsf (&disp, mnp->t6o, mnp->t6);
           trsfToRpy (&rz, &ry, &rx, &tmp);

           if (((mask & ALONG_X) && FABS(disp.p.x) > limit.t.x) ||
               ((mask & ALONG_Y) && FABS(disp.p.y) > limit.t.y) ||
               ((mask & ALONG_Z) && FABS(disp.p.z) > limit.t.z) ||
               ((mask & ABOUT_X) && FABS(rx) > limit.r.x) ||
               ((mask & ABOUT_Y) && FABS(ry) > limit.r.y) ||
               ((mask & ABOUT_Z) && FABS(rz) > limit.r.z))
            { stopCurrentMotion (mnp, ON_LIMIT);
            }
         }
}
```

This is a permanent monitor function that runs all the time. It checks to find out what limit specifications are currently in effect for its associated manipulator. If there aren't any (`mask == 0`), it saves computation by simply returning. Otherwise, it computes the difference between the desired trajectory and the real trajectory by multiplying the output value of **T6** (given by the `t6` field of the `MANIP` structure) by the inverse of the observed value of **T6** (given by the `t6o` field; the mode `T6O_EVAL` (Section 9.1.2) has presumably been set so that the trajectory generator will in fact maintain `mnp->t6o`). The rotational difference is converted into yaw, pitch, and roll angles so that it may be easily compared with the displacement thresholds along each axis. For each selected DOF, the trajectory error is then compared with the threshold and, if it is exceeded, the motion is canceled with the code `ON_LIMIT`.

Because only the application software uses the values set by the limit routines, they can be used to implement things different from their original defined purpose. For instance, `setDispLimit()` could be used to implement a velocity limit rather than a trajectory error limit. A monitor function which did this would look the same as the one above, except that it would have to compute a velocity transform instead of a displacement transform. It could do this by keeping the old value of **T6** around in a static variable, multiplying it by the new value of **T6** every cycle, and then scaling the outputs to units-per-cycle rather than units-per-second:

```
TRSF oldT6;
float scale;

...

scale = rcclGetInterval() / 1000.0;

multRiTrsf (&disp, &oldT6, mnp->t6);
trsfToRpy (&rz, &ry, &rx, &tmp);
rx *= scale;
ry *= scale;
rz *= scale;
disp.p.x *= scale;
disp.p.y *= scale;
disp.p.z *= scale;

oldT6 = *mnp->t6;
```

7.2 Compliance Specification Routines

Compliant motions can be specified with routines similar to the limit routines:

```
comply (mnp, format, value1, value2, ...)
  MANIP *mnp;
  char *format;
  float value1, value2, ...
```

```
lock (mnp, format)
  MANIP *mnp;
  char *format;

setComply (mnp, mask, bias)
  MANIP *mnp;
  unsigned long mask;
  FORCE *bias;
```

`comply()` takes a format string indicating which DOFs should be put into compliant mode. The codes are the same as those used to describe force limits with the `limit()` routine: `fx`, `fy`, `fz`, `tx`, `ty`, and `tz`. For each selected DOF, an additional argument is provided indicating the bias force to be associated with the compliance.

A compliance specification for a particular DOF takes effect with the next requested motion and stays in effect for all subsequent motions until explicitly canceled. To take a particular DOF out of comply mode, the primitive `lock()` can be called, where the DOFs to be locked are specified using the same format as for the `comply()` routine. For example, suppose the program wishes to establish a compliance along the z axis with a bias force 10.0 (Newtons) for the next three moves, and in addition, specify a zero-bias compliance about the x and y axes for the second motion. This could be specified as follows:

```
comply (mnp, "fz", 10.0);
move (mnp, p1);
comply (mnp, "tx ty", 0.0, 0.0);
move (mnp, p1);
lock (mnp, "tx ty");
move (mnp, p3);
lock (mnp, "fz");
```

Compliance can also be described in a more "machine oriented" way using the primitive `setComply()`, which takes a mask selecting the desired comply DOFs and a `FORCE` type argument containing the corresponding biases. Using this, the above example could be coded as:

```
FORCE bias;

bias.f.z = 10.0;
setComply (mnp, ALONG_Z, &bias);
move (mnp, p1);
bias.m.x = bias.m.y = 0.0;
setComply (mnp, (ALONG_Z|ABOUT_X|ABOUT_Y), &bias);
move (mnp, p2);
setComply (mnp, ALONG_Z, &bias);
move (mnp, p3);
setComply (mnp, 0, &bias);
```

Since compliance specifications are "sticky" (*i.e.*, they remain in effect for more than one motion request), it is useful to have a routine available that reads back the current settings so that intermediate specifications can be made without disturbing the overall context. The routine

```
getComply (mnp, mask, bias)
  MANIP *mnp;
  unsigned long *mask;
  FORCE *bias;
```

returns the current compliance settings. In the following example, we specify compliance along the x and y axes for two motions, without any knowledge of the surrounding context:

```
FORCE bias;
FORCE oldbias;
unsigned long oldmask;

...

getComply (mnp, &oldmask, &oldbias);
bias = oldbias;
bias.t.x = bias.t.y = 0.0;
setComply (mnp, (oldmask|ALONG_X|ALONG_Y), &bias);
move (mnp, p1);
move (mnp, p2);
setComply (mnp, oldmask, &oldbias);

...
```

`getComply()` returns the comply state currently set at the planning level. On the other hand, a function responsible for implementing compliance at the control level will need to know the specification in effect for the current motion. This is obtained with the routine `getActiveComply()`:

```
getActiveComply (mnp, mask, bias)
  MANIP *mnp;
  unsigned long *mask;
  FORCE *bias;
```

To implement compliant motion in RCCL, the applications generally use some variation of the position accommodation technique described in [Maples and Becker 1986]. With this paradigm, correctional displacements responding to the sensed forces are added to some coordinate frame in the manipulator's target position. In effect, the manipulator is made to "track" the observed force errors. Because the lowest level of control is still a position servo (responding to positional setpoints output by the trajectory generator) the response of such a system to changes in force values is somewhat damped. This, however, is usually desirable, as force control systems can otherwise be quite unstable.

It is easy to see that accommodation-based compliant control can be implemented by a functional transform that has access to force sensor data. Let the compliance specification be given by a selection matrix $\mathbf{S}$ and a set of bias forces $\mathbf{f}_c$, let the observed forces (in the comply frame) be $\mathbf{f}_o$, let the corresponding error be $\mathbf{f}_e$, let the value of the comply transform be $\mathbf{C}$, and let K be a gain associated with the force error. Then the computation done by the compliance function is effectively

$$\mathbf{f}_e = \mathbf{S}\,(\mathbf{f}_c - \mathbf{f}_o)$$

$$\mathbf{C} = \mathbf{C}\ \texttt{diffToTrsf}(K\ \mathbf{f}_e)$$

The term $K\ \mathbf{f}_e$ represents a small Cartesian displacement, which is converted into a transform using `diffToTrsf()`, and then accumulated in $\mathbf{C}$. This computation will be performed in the following example program.

The force control law presented above is a simple proportional one; more elaborate control laws can be developed with characteristics suitable for different sorts of tasks.

7.3 Program Example: "Comply"

This program uses RCCL primitives to implement a very simple compliance package, which is then used to put the manipulator into a "free" mode created by zero-bias compliance along each of the three coordinate axes. In this mode, the arm will allow itself to be "pushed around" in any translational direction. The force sensor connected to the robot is the usual 6 DOF wrist model manufactured by the Lord Corporation. The values read back from it are available in the `forceInput` field of the `HOW` structure.

planning level module

```
#include <rccl.h>
#include <errorCodes.h>
#include "manex.560.h"

COMPLY_CTRL_BLK *complyInit (mnp, tgain, rgain)
MANIP *mnp;
float tgain, rgain;
{
        COMPLY_CTRL_BLK *cbk;
        TRSF *comply;

        extern computeForce();
        extern complyFxn();

        comply = allocTrans (NULL, UNDEF);                      /*#1*/
        cbk = ALLOC_MEM (NULL, COMPLY_CTRL_BLK, UNDEF);
        cbk->tgain = tgain;                                     /*#2*/
        cbk->rgain = rgain;
        cbk->comply = comply;
        cbk->mnp = mnp;
        transMotionEval (comply, complyFxn, (int)cbk, mnp);     /*#3*/
```

```
        return (cbk);
}

complyRun (cbk)
COMPLY_CTRL_BLK *cbk;
{
        cbk->init = YES;
        runMonitorFxn (cbk->mnp, computeForce, (int)cbk);       /*#4*/
        rcclBlock();
}

main()
{
        TRSF_PTR b, e;
        POS_PTR p1;
        MANIP *mnp;
        JNTS rcclpark;
        COMPLY_CTRL_BLK *cbk;

        rcclSetOptions (RCCL_ERROR_EXIT);

        e = allocTransXyz (NULL, UNDEF, 0.0, 0.0, TOOLZ);
        b = allocTransRot (NULL, UNDEF, B_X, B_Y, B_Z, xunit, 180.0);

        mnp = rcclCreate (getDefaultRobot(), 0);
        cbk = complyInit (mnp, 0.01, 0.0);                      /*#5*/

        p1 = makePosition (NULL, T6, e, cbk->comply, EQ, b, TL, cbk->comply);

        rcclStart();
        complyRun (cbk);                                        /*#6*/

        setMod (mnp, 'c');
        setTime (mnp, F_DEFAULT, F_UNDEF);
        comply (mnp, "fx fy fz", 0.0, 0.0, 0.0);                /*#7*/
        move (mnp, p1);

        while (1)                                               /*#8*/
         { FORCE f;

           accessMem (cbk, &cbk->force, &f, 6*sizeof(float));   /*#9*/
           printVf ("%10.4f\n", (float*)&f, 6);
           delay (500.0);
         }
}
```

control level module

```
#include <rccl.h>
#include <puma_kynvar.h>
```

```
#include <errorCodes.h>

#include "manex.260.h"

#define SENSOR_Z_OFFSET         90.0

#define NEWTONS_PER_POUND       4.448
#define LB_PER_UF               (1.0/40.0)
#define NEWTONS_PER_UF          (4.448/40.0)
#define MM_PER_INCH             25.4

computeForce (arg, mnp)
int arg;
MANIP *mnp;
{
        COMPLY_CTRL_BLK *cbk;
        FORCE force1;
        FORCE force2;
        char *fxnName = "computeForce()";
                                                                /*#10*/
        if ((cbk = (COMPLY_CTRL_BLK*)getMemByAddr ((void*)arg)) == NULL)
         { rciAbort (EFatal, "%s -- can't find control block\n", fxnName);
           return;
         }

        if (cbk->init)
         { SET_FORCE_READ (mnp->rbt, 1);                        /*#12*/
           cbk->init = NO;
           cbk->valid = 0;
         }
        else
         { if (mnp->how->forceOK)                               /*#13*/
            { TRSF stoc;

              force1.f.x = mnp->how->forceInput[0]*NEWTONS_PER_UF;
              force1.f.y = mnp->how->forceInput[1]*NEWTONS_PER_UF;
              force1.f.z = mnp->how->forceInput[2]*NEWTONS_PER_UF;

              force1.m.x = mnp->how->forceInput[3]*NEWTONS_PER_UF*MM_PER_INCH;
              force1.m.y = mnp->how->forceInput[4]*NEWTONS_PER_UF*MM_PER_INCH;
              force1.m.z = mnp->how->forceInput[5]*NEWTONS_PER_UF*MM_PER_INCH;

              multRiTrsf (&stoc, mnp->tool, getTransByAddr(cbk->comply));
              stoc.p.z -= SENSOR_Z_OFFSET;                      /*#14*/
              transForce (&force2, &force1, &stoc);
              accessMem (cbk, &force2, &cbk->force, 6*sizeof(float));
              cbk->valid = 1;                                   /*#15*/
            }
           else
            { cbk->valid = 0;                                   /*#16*/
            }
         }
}
```

```
complyFxn (t, arg, mnp)
TRSF *t;
int arg;
MANIP *mnp;
{
        char *fxnName = "complyFxn()";

        COMPLY_CTRL_BLK *cbk;
        FORCE complyValues;
        unsigned long complyMask;
        FORCE forceError;
        DIFF accomodate;
        TRSF accomTrsf;

        if ((cbk = (COMPLY_CTRL_BLK*)getMemByAddr ((void*)arg)) == NULL)
         { rciAbort (EFatal, "%s -- can't find control block\n", fxnName);
           return;
         }

        if (!cbk->valid)
         { if (cbk->badDataCount++ > 4)                          /*#17*/
            { rciAbort (0, "bad force data for 4 consecutive cycles\n");
            }
           return;
         }
        cbk->badDataCount = 0;

        bzero ((char*)&forceError, sizeof(FORCE));

        getActiveComply (mnp, &complyMask, &complyValues);      /*#18*/

        if (complyMask & ALONG_X)
         { forceError.f.x = (complyValues.f.x - cbk->force.f.x);
         }
        if (complyMask & ALONG_Y)
         { forceError.f.y = (complyValues.f.y - cbk->force.f.y);
         }
        if (complyMask & ALONG_Z)
         { forceError.f.z = (complyValues.f.z - cbk->force.f.z);
         }
        if (complyMask & ABOUT_X)
         { forceError.m.x = (complyValues.m.x - cbk->force.m.x);
         }
        if (complyMask & ABOUT_Y)
         { forceError.m.y = (complyValues.m.y - cbk->force.m.y);
         }
        if (complyMask & ABOUT_Z)
         { forceError.m.z = (complyValues.m.z - cbk->force.m.z);
         }                                                       /*#19*/

        accomodate.t.x = forceError.f.x * cbk->tgain;
        accomodate.t.y = forceError.f.y * cbk->tgain;
```

```
        accomodate.t.z = forceError.f.z * cbk->tgain;

        accomodate.r.x = forceError.m.x * cbk->rgain;
        accomodate.r.y = forceError.m.y * cbk->rgain;
        accomodate.r.z = forceError.m.z * cbk->rgain;

        diffToTrsf (&accomTrsf, &accomodate);                     /*#20*/

        multTrsf (t, &accomTrsf, t);                              /*#21*/
}
```

NOTE – this example has been coded for the PUMA 560 robot and lives at `comply.560.c` *and* `complyCtrl.560.c` *in* `demo.rccl`. *No equivalent program has been written for the PUMA 260 because no force sensor is currently available (force sensors tend to be too big for the PUMA 260).*

The comply package implemented in this program consists of two planning level routines, `complyInit()` and `complyRun()`, and two control level routines, `computeForce()` and `complyFxn()`.

`complyInit()` should be called after the `mnp` structure for the robot has been created (#5). It allocates a comply transform and sets up an internal data structure used to communicate with the control level functions. It also sets the translational and rotational compliance gains, which it takes as arguments. At (#1), the comply transform (which will be used to accumulate the necessary force accommodations) is allocated. Communication with the control level routines will be done using a memory object of the type `COMPLY_CTRL_BLK` (defined in `"manex.560.h"`). This type is defined as follows:

```
        typedef struct {
                float tgain;
                float rgain;
                TRSF *comply;
                MANIP *mnp;
                FORCE force;
                int valid;
                int init;
          } COMPLY_CTRL_BLK;
```

The fields include the translational and rotational accommodation gains `tgain` and `rgain`; `comply`, which is the planning level address of the comply transform and is used by the control level function to get a valid pointer to it; *force*, which contains force sensor values transformed into the comply frame; *valid*, a boolean value stating whether or not the force values are valid for the current control cycle; and *init*, another boolean value used to initialize the force computation function. `complyInit()` sets the fields `tgain` and `rgain` to the gain values passed in as arguments (#2), sets the `comply` field to the comply transform address, and uses `transMotionEval()` to bind the comply transform to the function `complyFxn()` on a per-motion basis (#3). The routine returns a pointer to the comply control block, which will be passed back as an argument to `complyRun()`.

The gains which are passed to `complyInit()` are very application specific. They depend, in general, on the characteristics of the robot and its position servos, the task being performed, and the trajectory control rate.

Having allocated a comply transform by calling `complyInit()`, the main program needs to incorporate it into the target position of any motion which is to respond to comply requests. In particular, the comply transform should be the last transform in the target position's TOOL frame. In the current program example, this is done for the position `p1`, defined as follows:

$$\mathbf{T6\ \ E\ \ COMPLY\ =\ B}$$

where **E** is the end-effector transform, **B** is a convenient starting location in space, and **COMPLY** is the comply transform.

The program starts the trajectory generator and calls `complyRun()` (`#6`), which starts up a permanent monitor function to compute the force values in the current manipulator TOOL frame (`#4`). `complyRun()` sets the `init` field of the comply control block to tell the monitor to turn on the force torque sensor.

Compliant motions may now be requested using the system `comply()` primitive and motions to target positions which contain the comply transform. The desired "free manipulator" effect is achieved by requesting a zero-force compliance along all three translational degrees of freedom (`#7`). The compliant motion itself is set to have an indefinite time limit. While it is in progress, the program occupies itself by printing out the force sensor values computed by the monitor function. The monitor writes the force values into the `force` field of the comply control block for the planning level to read out. To guarantee consistency of the force data, the field is written and read using `accessMem()` (`#9`), although (in this case) this is done more for show than from necessity.

Two control level functions are used to implement the compliance: `computeForce()`, which computes the forces as seen in the compliance frame, and `complyFxn()`, which uses these to determine the necessary changes to the **COMPLY** transform. Partitioning the computation in this way is reasonable (though not essential).

`computeForce()` runs all the time, computing force values for anyone who cares to read them. It begins by using its application argument to get a pointer to the comply control block (`#10`). The force sensor readings themselves are obtained from the manipulator's `HOW` structure (see Section 6.1.3).

If `cbk->init` has been set by the planning level, this indicates that things are just starting up and the RCI interface must be told to start reading the force torque sensor. This is done with a call to the macro `SET_FORCE_READ()` (`#12`), after which `init` is cleared and `computeForce()` returns, since valid force torque data will not be ready for at least another control cycle.

To determine whether the `HOW` structure contains valid force/torque values for the current control cycle, the field `how->forceOK` is examined (`#13`), and if true, the force values are read out of the `forceInput` field and converted from sensor units into Newtons and Newton-millimeters. (The conversion values in this program are specific to the 6 DOF Lord sensor.) Once the force values have been read in, they still must be converted to the coordinate frame in which the **COMPLY** transform is being computed. The transform **STOC** is defined to map from the wrist sensor frame to the COMPLY frame. Recall that the application code was requested to make **COMPLY** the last transform in the **TOOL** part of the target position. If the rest of **TOOL** is described by **T**, then we have

$$\mathbf{T}\ \mathbf{COMPLY}\ =\ \mathbf{TOOL}$$

If we have another transform **F** which maps from the $\mathrm{T6}$ frame to the sensor, then **STOC** is given by

$$\mathbf{STOC}\ =\ \mathbf{F}^{-1}\ \mathbf{T}$$

which can be computed as

$$\mathbf{STOC}\ =\ \mathbf{F}^{-1}\ \mathbf{TOOL}\ \ \mathbf{COMPLY}^{-1}$$

since **TOOL** is constantly maintained by the trajectory generator and is available as the `tool` field in the `MANIP` structure. **STOC** is computed at location (`#14`). Since **F** is simply an offset along z with no rotational component, some computation can be saved by simply subtracting this offset from $\mathbf{TOOL}\ \ \mathbf{COMPLY}^{-1}$ rather than premultiplying by $\mathbf{F}^{-1}$. Once we have **STOC**, it is used by `transForce()` to map forces from the sensor frame into the comply frame. These values are then written into the comply control block and the `valid` field is set (`#15`). For any control cycle in which no valid force values are available, the `valid` field is cleared (`#16`).

The function `complyFxn()` executes only when the target motion position contains the **COMPLY** transform. Its job is to use the force values computed by `computeForce()` and perturb **COMPLY** to accommodate the error between these values and desired force bias for whatever degrees of freedom happen to be in compliance mode. Again, the function uses its application argument to get a pointer to the comply control block. It immediately checks the `valid` field to see if there is proper force data for this cycle, and if not, then the function bumps a counter and returns. When the counter exceeds a certain value, it means that force data values have been unavailable for several consecutive cycles, and the program is aborted (`#17`).

The next thing the routine does is use `getActiveComply()` to obtain the current comply mask and bias forces. The force error is initialized to zero, and then for each compliant DOF given by the mask, the corresponding field is set to the difference between the desired bias force and the observed force (`#19`). The required accommodations are determined by multiplying the force errors by the appropriate gain to get a displacment. This displacement is small enough to treat as a differential[1], so it can be converted to a transform using `diffToTrsf()` (`#20`). Successive perturbations are accumulated into **COMPLY** by post multiplication (`#21`).

7.4 Program Example: "Cylin"

This demo program was written at the Jet Propulsion Laboratory on a PUMA 560 robot. It uses guarded force-limit motions and compliance to locate a small cylinder and then comply around the outside of it. It assumes that the robot is fitted with a force/torque sensor and a peg-like end-effector, and that there is a small cylinder in the work space which the operator can position the robot near using the teach routine.

[1] if it is not, then the gains are too high and there will be stability problems.

planning level module

```
#include <rccl.h>
#include <math.h>
#include "manex.560.h"

#define PEG_LENGTH      230.0   /* length of comply peg */
#define PEG_RADIUS        9.5   /* radius of comply peg */
#define DROP_DEPTH       35.0   /* vertical dist. to probe points */
#define MAX_RADIUS       70.0   /* max. likely radius of cylinder */
#define NUM_TURNS           2   /* no. of times to go around cylinder */
#define MSEC_PER_REV  12000.0   /* speed to go around cylinder at */

extern forceMonitor();
extern rotzFxn();
extern COMPLY_CTRL_BLK *complyInit();

float square(x)
float x;
{
        return (x*x);
}

main ()
{
        MANIP *mnp;
        JNTS rcclpark;
        char *robotName;
        COMPLY_CTRL_BLK *cbk;

        TRSF_PTR start, rotz, offset, peg, touch;
        float xval[3], yval[3];
        POS_PTR idle, seek, turn;
        int i;

        float centerX;
        float centerY;
        float radius;

        rcclSetOptions (RCCL_ERROR_EXIT);
        robotName = getDefaultRobot();
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }

        start = allocTrans (NULL, UNDEF);
        offset = allocTrans (NULL, UNDEF);
        rotz = allocTrans (NULL, UNDEF);
        peg = allocTransXyz (NULL, UNDEF, 0.0, 0.0, PEG_LENGTH);
        touch = allocTrans (NULL, UNDEF);
```

```
mnp = rcclCreate (robotName, 0);
cbk = complyInit (mnp, 0.005, 0.00);                        /*#1*/

idle = makePosition (NULL, T6, peg, EQ, start, TL, peg);
seek = makePosition (NULL, T6, peg, rotz, EQ,
                           start, rotz, offset, TL, rotz);
turn = makePosition (NULL, T6, peg, rotz, cbk->comply, EQ,
                           start, rotz, offset, TL, cbk->comply);

rcclStart ();
complyRun (cbk);                                            /*#2*/

printf ("Position robot about %g mm. above cylinder, with peg down\n",
         DROP_DEPTH-10.0);
rcclTeach (mnp, "TEACH> ", NULL);                           /*#3*/
solveTrans (start, seek, start, mnp->here);                 /*#4*/

/* locate the points around the cylinder */

runMonitorFxn (mnp, forceMonitor, (int)cbk);                /*#5*/

setMod (mnp, 'c');
offset->p.x = MAX_RADIUS;                                   /*#6*/
for (i=0; i<3; i++)
 { int mid;
   TRSF delta;

   rotToTrsf (rotz, zunit, 120.0*i);                        /*#7*/
   move (mnp, seek);                                        /*#8*/
   offset->p.z = DROP_DEPTH;
   move (mnp, seek);                                        /*#9*/

   limit (mnp, "fx", 5.0);
   setCartVel (mnp, 10.0, F_DEFAULT);                       /*#10*/
   distance (mnp, "dx", -MAX_RADIUS);
   updateTrans (mnp, touch, idle, start, mnp->t6);          /*#11*/
   mid = move (mnp, seek);                                  /*#12*/
   setCartVel (mnp, F_DEFAULT, F_DEFAULT);
   setTime (mnp, 0.0, F_DEFAULT);
   move (mnp, seek);                                        /*#13*/
   offset->p.z = 0.0;
   move (mnp, seek);                                        /*#14*/
   waitForStop (mid);
   if (motionStopCode(mid) != ON_FORCE)
    { printf ("Did not touch cylinder\n");
      rcclRelease (1);
      exit (-1);
    }
   multLiTrsf (&delta, start, touch);
   xval[i] = delta.p.x;
   yval[i] = delta.p.y;                                     /*#15*/
   printf ("point %d: x = %g, y = %g\n", i+1, xval[i], yval[i]);
 }
```

```
        offset->p.x = 0.0;
        move (mnp, seek);
        waitForCompleted (mnp);

        computeCenter (xval[0], yval[0], xval[1], yval[1],      /*#16*/
                       xval[2], yval[2], &centerX, &centerY);

        printf ("Cylinder center is at (%g, %g)\n", centerX, centerY);

        multTrsfXyz (start, centerX, centerY, 0.0);             /*#17*/

        radius =  sqrt (square(xval[0] - centerX) + square(yval[0] - centerY))
                - PEG_RADIUS;

        printf ("radius = %g\n", radius);

        rcclSetModes (mnp, TRACKING_MODE);                      /*#18*/

        setCartVel (mnp, 10.0, F_DEFAULT);

        identTrsf (rotz);
        move (mnp, turn);
        offset->p.x = radius + PEG_RADIUS + 5.0;
        move (mnp, turn);
        offset->p.z = DROP_DEPTH;
        move (mnp, turn);
        comply (mnp, "fx", 2.0);                                /*#19*/
        offset->p.x = radius;
        move (mnp, turn);
        waitForCompleted (mnp);

        transMotionEval (rotz, rotzFxn, 0, mnp);                /*#20*/
        comply (mnp, "fx", 10.0);                               /*#21*/
        setTime (mnp, F_DEFAULT, NUM_TURNS*MSEC_PER_REV);
        move (mnp, turn);

        waitForCompleted (mnp);
        delay (1000.0);                                         /*#22*/

        setCartVel (mnp, F_DEFAULT, F_DEFAULT);
        lock (mnp, "fx");
        setTransConst (rotz);
        offset->p.x = radius + PEG_RADIUS + 5.0;
        move (mnp, seek);
        offset->p.z = 0.0;
        move (mnp, seek);
        offset->p.x = 0.0;
        move (mnp, seek);                                       /*#23*/
        stop (mnp, 1000.0);

        waitForCompleted (mnp);
        rcclRelease (1);
}
```

control level module

```
#include <rccl.h>
#include "manex.560.h"

#define MSEC_PER_MINUTE (60.0*1000.0);
#define RPM             5

rotzFxn (t, arg)
TRSF *t;
int arg;
{
        /* produce a rotation about z at 5 rpm */

        static float ang = 0.0;
        ang += 360.0*RPM*rcclGetInterval()/MSEC_PER_MINUTE;
        rotToTrsf (t, zunit, ang);
}

forceMonitor (arg, mnp)
int arg;
MANIP *mnp;
{
        char *fxnName = "forceMonitor()";

        COMPLY_CTRL_BLK *cbk;
        int limitMask;
        FORCE limitValues;

        if ((cbk = (COMPLY_CTRL_BLK*)getMemByAddr((void*)arg)) == NULL)
         { rciAbort (0, "%s -- can't find control block\n", fxnName);
           return;
         }
        getActiveForceLimit (mnp, &limitMask, &limitValues);    /*#24*/

        if (limitMask && cbk->valid)                            /*#25*/
         {
           if (   (limitMask & ALONG_X &&
                   FABS(cbk->force.f.x) > limitValues.f.x)
                || (limitMask & ALONG_Y &&
                   FABS(cbk->force.f.y) > limitValues.f.y)
                || (limitMask & ALONG_Z &&
                   FABS(cbk->force.f.z) > limitValues.f.z)
                || (limitMask & ABOUT_X &&
                   FABS(cbk->force.m.x) > limitValues.m.x)
                || (limitMask & ABOUT_Y &&
                   FABS(cbk->force.m.y) > limitValues.m.y)
                || (limitMask & ABOUT_Z &&
                   FABS(cbk->force.m.z) > limitValues.m.z))
```

```
            { stopCurrentMotion (mnp, ON_FORCE);                /*#26*/
              rciPrintVf ("LIMIT X %11.4f\n", &cbk->force, 6);
            }
        }
}
```

NOTE – this example has been coded for the PUMA 560 robot and lives at `comply.560.c` *and* `complyCtrl.560.c` *in the directory* `demo.rccl`. *An equivalent program for the PUMA 260 lives at* `cylin.260.c` *and* `cylinCtrl.260.c`*, except that it only simulates the presence of the cylinder and does not do any real compliance.*

This program makes use of the compliance package defined in the previous example. Compliance is enabled by calling `complyInit()` and `complyRun()` after the calls to `rcclCreate()` and `rcclStart()`, respectively (`#1`, `#2`). The operator is instructed to move the robot so that the peg tip is positioned several millimeters above the cylinder, with the peg facing down. To facilitate this, the program enters the teach routine (`#3`). When the teach routine returns, the transform **START** is instantiated so that the position equation `seek` defines the current robot location (`#4`).

The program is now ready to begin the guarded motions to locate the cylinder exactly. A permanent monitor function, `forceMonitor()`, is set up to detect force limits (`#5`). This monitor will use the force values calculated by the monitor `computeForce()` (see the previous example), which was set running by the routine `complyRun()`. It is assumed that the peg tip is located roughly above the center of the cylinder. The exact location of the center is determined using three guarded moves, in which the robot moves away from cylinder, down, and then back towards it in hope of making contact. Each move is spaced radially $120°$ apart. The different via points used to do the moves are created by making local modifications to the transform **OFFSET** in `seek`.

`forceMonitor()` checks force limits independently along or about each specified axis. For best accuracy, therefore, the force limit for each guarded move should lie along a single axis (the x axis is used in this case)[2]. How may this be done when the motions themselves approach the cylinder from three different directions? The answer is to rotate the tool frame itself between moves so that its x axis is always parallel to the line of approach. This is the purpose of the transform **ROTZ**. The reason this works is that the force values themselves are computed by the function `computeForce()` and hence are described with respect to the current tool frame (offset by the **COMPLY** transform, but **COMPLY** is the identity at this point in the program, and so we do not worry about it).

To keep from having to actually rotate the end-effector when the TOOL frame is rotated, another **ROTZ** is placed on the other side of the position equation to cancel out the net rotation (the transform **OFFSET**, which lies in between, does not interfere because it contains only translations).

Each guarded motion starts at a position located `MAX_RADIUS` millimeters away from the presumed cylinder center. This location is specified by setting the x coordinate of `offset` to `MAX_RADIUS` (`#6`). This works for all the guarded motions because the OFFSET frame is constantly rotated so as to be parallel to the approach axis (`#7`). The motion sequence consists of moving out along the radius (`#8`), dropping down (`#9`), and coming back in with a force limit set along the x axis (`#12`)

[2] Alternatively, the monitor function could be implemented to interpret its limit information differently.

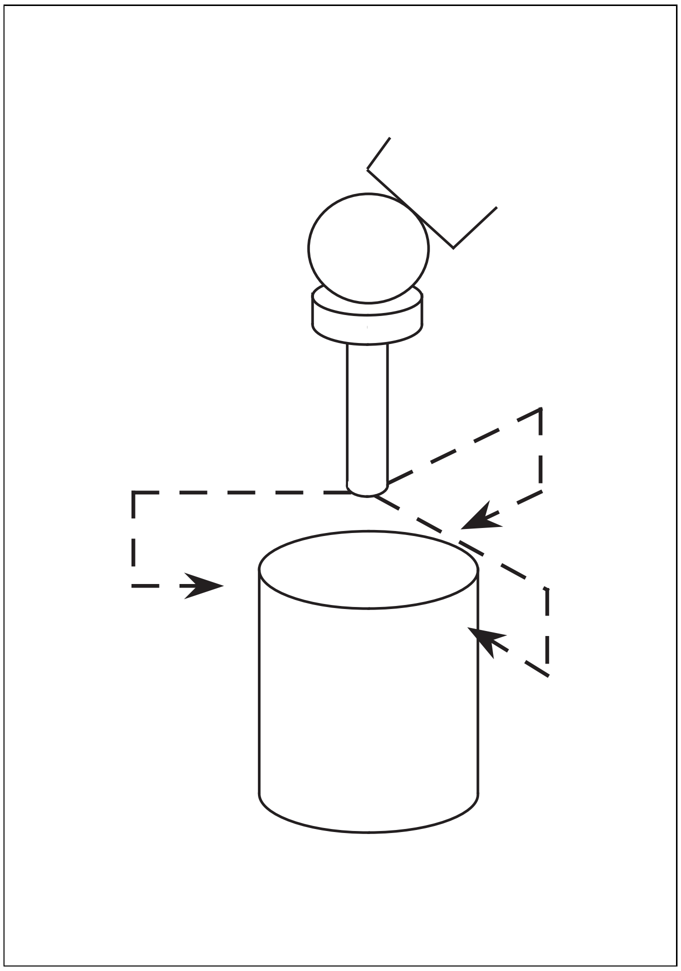

Figure 33: The three guarded motions used to locate the cylinder.

(see Figure 33). The velocity is set low to prevent damage on contact (`#10`), and `updateTrans()` is used to request that the transform `touch` be evaluated when the guarded move terminates (`#11`). Upon contact, the robot moves back out (`#13`) and up (`#14`). The program waits for each guarded move to complete, checks the associated motion stop code to make sure that contact was actually achieved, and then uses the transform `touch` to determine the location of the contact point in the xy plane of the frame START.

When the guarded moves are complete, the collected x and y coordinates of the three contact points are used to compute the real center of the cylinder (`#16`) (the function `computeCenter()` is defined in a companion file). The **START** transform is then updated correspondingly (`#17`).

The last part of the program involves moving the manipulator back to the cylinder and complying around it. The comply motions are done with the manipulator in "tracking mode" (`#18`) (Section 9.1.2). This ensures that if a delay occurs between adjacent compliant motions, such that the following one is not on the motion queue when the previous one ends, the robot will continue to track the last target (and hence remain in comply mode), rather than entering the idle state and "freezing" where it is. As with the force limits, the compliance is specified along the x axis only. For the approach move, during which contact is made, a small bias force is set (`#19`); for the next move, which moves the peg around the cylinder (`#21`), a larger bias force is used. The action of moving the peg around the cylinder is achieved by binding **ROTZ** to a rotation function (`#20`). When the circular motion has finished, the program waits for another 10 seconds (`#22`) to illustrate the effect of tracking mode: the rotation around the cylinder should continue during this time. Finally, the arm is moved back up to its starting position (`#23`).

The control level module for this program contains the rotation function `rotzFxn()` used for the compliant move and the force monitor function `forceMonitor()`. It also loads in the routines `complyFxn()` and `computeForce()`, which belong to the comply package. The force limit monitor uses the force data computed by the comply routine `computeForce()`. It gets the force limit specification for the current motion (`#24`), checks to see that the force data is valid (`#25`), and cancels the motion if the observed forces exceed any of the specified limit values (`#26`).

8. Multi-Robot Capabilities

8.1 Controlling Multiple Robots

Multi-RCCL allows several robots to be controlled from within one program. All the programmer needs to do is make separate calls to `rcclCreate()` for each robot, and then refer to the different `MANIP` structures as required. At the Jet Propulsion Laboratory, there are two robots called `"Right"` and `"Left"`. A piece of code which moves both of them to their starting positions looks like this:

```
MANIP *right, *left;
JNTS parkR, parkL;

...

getRobotPosition (parkL.v, "rcclpark", "Right");
getRobotPosition (parkL.v, "rcclpark", "Left");

right = rcclCreate ("Right", 0);
left  = rcclCreate ("Left", 0);

rcclStart();

movej (right, &parkR);
movej (left, &parkR);

waitForCompleted (right);
waitForCompleted (left);

rcclRelease (1);
```

On possible limitation to this capability is the CPU power available to do all the necessary trajectory computations. This was the motivation for developing a multi-CPU version of the trajectory generator, in which the trajectory generation for each robot can be relegated to a different CPU. On a multi-CPU implementation, `rcclCreate()` will assign robots to different CPUs by default. If for some reason the application wants to have the trajectory task for a robot run on a *particular* CPU, then this can be specified explicitly using the CPU selection mask (the second argument to `rcclCreate()`). In the example above,

```
right = rcclCreate ("Right", 0x2);
left  = rcclCreate ("Left", 0x2);
```

would explicitly assign both robots to CPU 1 (the CPUs are numbered starting at 0, where 0 is the arbiter CPU).

As mentioned in the introduction, Release 5.0 of RCCL does not contain a multi-CPU implementation (since the MicroVAX support has been phased out). However, the enormous increase in CPU speed in recent years has made it quite feasible to do the trajectory computations for several robots a single CPU.

8.2 Virtual Manipulators

Another particularly useful capability of Multi-RCCL is the ability to create "virtual manipulators". These are basically T6 frames without the attached robot. They are controlled by the same data structures and routines as normal RCCL robots, and can be moved with the usual motion primitives. Their only restriction is that joint level features are not defined for them; *i.e.*, calling `setJointVel()` or doing `setMod('j')` on a virtual manipulator will cause an error.

A virtual manipulator is created with a call to `rcclCreate()`. The system will set up a virtual manipulator if the name of the robot (given by the first argument) is not equal to the name of any real robot configured into the system. Naturally, the name `foo` is a common one:

```
vmnp = rcclCreate ("foo", 0);
```

The `t6` (and related fields) of the `MANIP` structure are initialized to the identity. The joint angle structures, such as `j6` and the like, are simply zeroed and left that way. The only interpolation mode allowed for a virtual manipulator is Cartesian (which of course means that unlike manipulators corresponding to real robots, the default interpolation mode for virtual manipulators is Cartesian).

Because a virtual manipulator has no physical constraints, it can be assigned any acceleration or velocity limit desired. When created, it is given the same default Cartesian velocity limits as the rest of the manipulators in the system, along with the acceleration limits `DEFAULT_TRANS_ACCEL` and `DEFAULT_ROT_ACCEL`.

Sometimes, when moving a virtual manipulator to a position, it is nice to simply "put it there", rather than having to move it there with a `move()` command. Again, this is possible because there are no physical constraints on the manipulator. The primitive

```
maintain (vmnp, pos)
  MANIP *vmnp;
  POS *pos;
```

is designed to do this: it "puts" `vmnp` at the target position `pos` and keeps it there indefinitely, until another move or maintain request is received.

One place where virtual manipulators are useful is in system testing, since there is no robot to worry about or even to start up. As a simple example, we present the following program based on one of the internal RCCL testing routines. The program tests to see that the system is behaving correctly with regard to the `setCartVel()` primitive. To do so, it reads back the number of control cycles assigned by the system to a particular motion (the σ value; see Section 4.4.1) and checks that this corresponds to the specified velocity. The check count is much easier to determine if the transition time is set to 0, which is risky on a real robot but completely natural on a virtual one.

```
MANIP *vmnp;
TRSF *target;
POS *p0;
int sigmaCheck;
int sigma;

target = allocTrans (NULL, UNDEF);
p0 = makePosition (NULL, T6, EQ, target, TL, T6);

vmnp = rcclCreate("foo", 0);
rcclStart();

target->p.z = 123.0;
setCartVel (vmnp, 123.0, F_UNDEF);
setTime (vmnp, 0.0, F_DEFAULT);
move (vmnp, p0);
rcclBlock();
getActiveMotionCounts (vmnp, NULL, &sigma);
sigmaCheck = NINT(1000.0/rcclGetInterval());
if (sigma != sigmaCheck)
 { printf ("sigma = %d vs. %d\n", sigma, sigmaCheck);
 }
```

The virtual manipulator is told to move from its initial (identity) position to a target position that has a displacement of 123 along the z axis. Since the translational velocity is also set explicitly to 123, and the transition time is set to 0, the time required for this motion should be 1 second. This is verified by reading back the actual motion segment count σ with the routine `getActiveMotionCounts()`. The preceding call to `rcclBlock()` ensures that at least one control cycle has elapsed since the move request and that σ has in fact been computed.

Another, more application oriented use of the virtual manipulator is in controlling the trajectory of a robot arm by making it track a remote frame. Suppose that a robot target position `p0` is composed of the usual $\mathrm{T6}$, E, and Z (base) frames, plus another frame OBJ defined with respect to Z:

$$\mathbf{Z}\ \ \mathbf{T6}\ \ \mathbf{E}\ =\ \mathbf{OBJ}$$

Now suppose that OBJ happens to be the $\mathrm{T6}$ frame for a virtual manipulator. If this virtual manipulator is moved to, or made to follow, some arbitrary target position `pv`, defined (generally) as as

$$\mathbf{OBJ}\ =\ \mathbf{TARGET},$$

then we will have the kinematic situation seen in Figure 34.

Any motion of the virtual manipulator will be tracked by the real manipulator. If the motion of the real robot to `p0` is set for indefinite duration, then an entire set of motions can be specified for it by moving only the virtual manipulator.

```
setTime (mnp, F_DEFAULT, F_UNDEF);
```

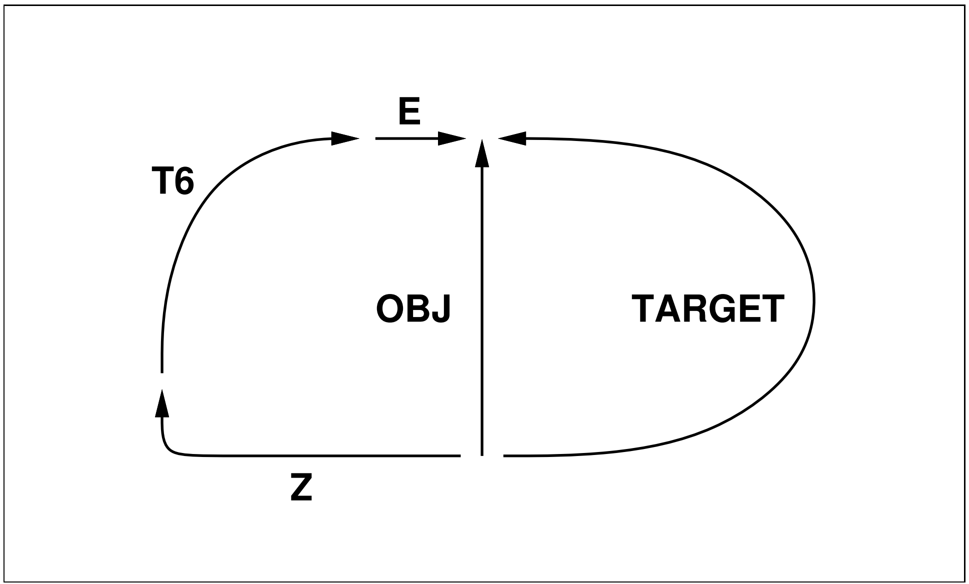

Figure 34: Kinematic graph in which a robot's target position is bound to a virtual manipulator.

```
move (mnp, p0);

... now control 'mnp' using 'vmnp' ...

move (vmnp, p1);
stop (vmnp, 1000.0);
move (vmnp, p2);

... and now release 'mnp' ...

waitForCompleted (vmnp);
stopCurrentMotion (mnp);
```

These induced motions will differ from direct motions on `mnp` because the coordinate frame in which the drive parameters are computed will be the TOOL frame of `vmnp`, not `mnp`. This can in fact be useful if it is necessary to move a robot with the drive transform computed somewhere other than adjacent to the TOOL frame.

The transition and path segment times for induced motions, unless explicitly stated, will be based on the default acceleration and velocity limits for the virtual manipulator rather than the real manipulator. Because of this, it may be desirable to set these limits on the virtual manipulator to be equal to the limits for the real manipulator. A piece of code that does this is

```
float tval, rval;

getCartVel (mnp, &tval, &rval);
setCartVel (vmnp, tval, rval);
getCartAccel (mnp, &tval, &rval);
```

```
setCartAccel (vmnp, tval, rval);
```

where the velocity and acceleration scale factors are assumed to be 1; if not, they can be "brought over" as well:

```
setSpeed (vmnp, getSpeed (mnp));
setAccelScale (vmnp, getAccelScale (mnp));
```

It is possible not only to drive a real manipulator with a virtual one, but to connect two real manipulators together in a "master-slave" configuration. All we need to do is to create a "slave position" for the slave robot which explicitly contains the **T6** transform for the master robot. For example, consider the following:

```
MANIP *master, *slave;
POS *ps;

master = rcclCreate ("Right", 0);
slave  = rcclCreate ("Left", 0);

... set up appropriate transforms 'a' and 'b'
    connecting the slave to the master at
    the base and tool tips ...

ps = makePosition (NULL, a, T6, b, EQ, master->t6);

setTime (slave, F_DEFAULT, F_UNDEF);
move (slave, ps);

... slave is now bound to master ...
```

After this code is executed, the robot "Left" is slaved to the robot "Right" for all subsequent motions (until the move to `ps` is canceled). The kinematic situation is essentially the same as in Figure 34, except that **Z** is replaced by **A** and **E** by **B**.

When manipulators are connected together and made to move around, a problem is usually created by the residual forces which build up between them: they start doing "isometric exercises" with one another. In general, this can be solved by inserting a **COMPLY** transform at some point in the kinematic chain between the two robots. The **COMPLY** transform is attached to a real-time function that adjusts its values to cancel out the residual forces. While this transform can be placed anywhere between the two robots, it is desirable in practice to place it as close as possible to the place where the forces are actually sensed. If they are sensed in or near the **T6** frame of the slave manipulator, then we will want to construct a slave kinematic loop like the one in Figure 35.

It is also possible to slave two or more real robots to a single master manipulator, which may be either a virtual manipulator or another real robot. There should be at least one **COMPLY** transform for every extra real manipulator in the graph. If the master manipulator is a virtual manipulator, then we have a kinematic situation which is useful if we want two or more real robots to manipulate a common physical object. This situation is shown in Figure 36.

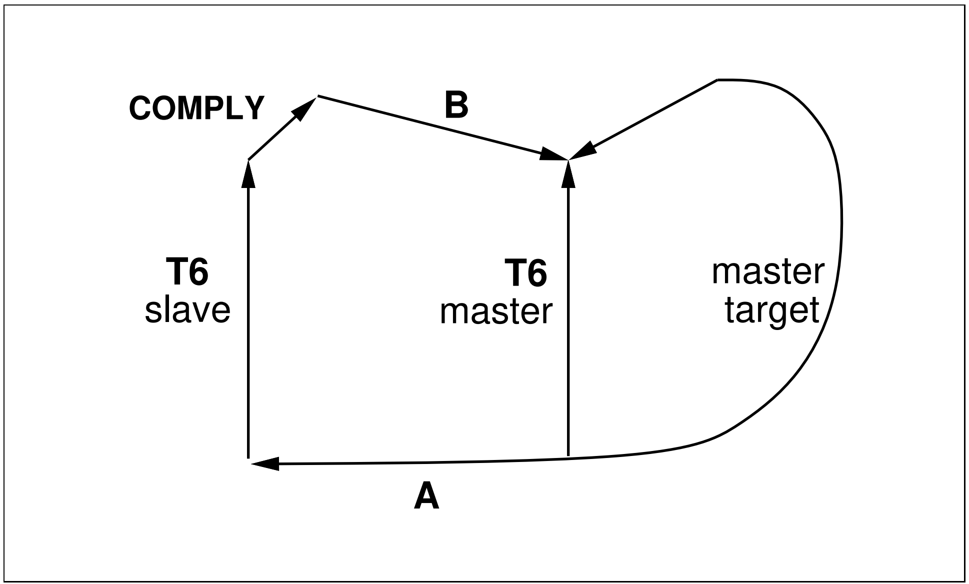

Figure 35: Kinematic graph for a simple master-slave robot configuration.

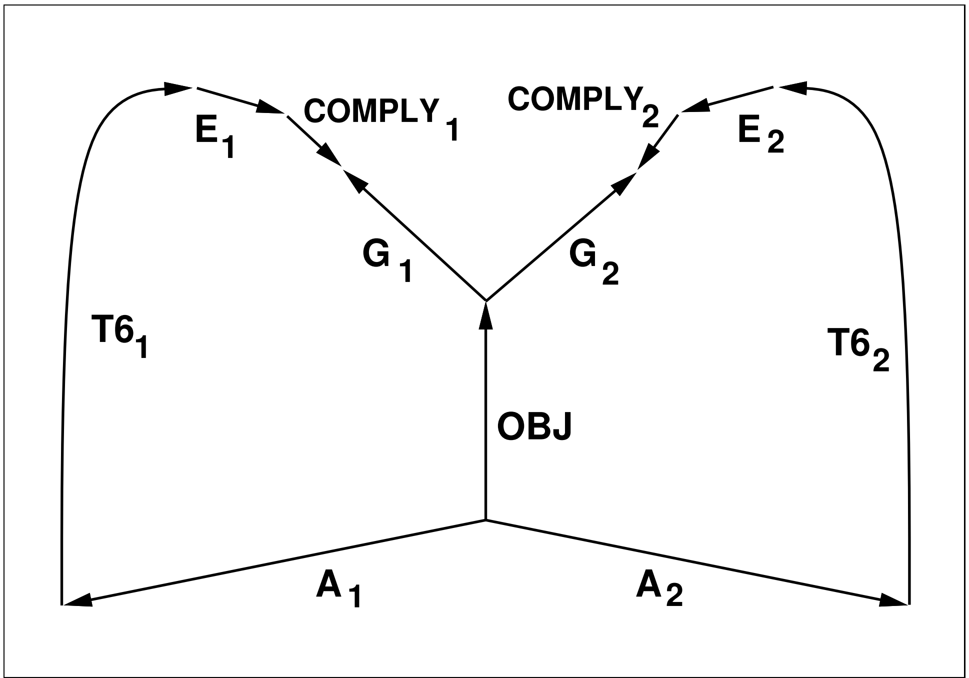

Figure 36: Kinematic graph for two manipulators holding a common object.

Each manipulator is “connected” to the physical object using a position equation that looks like

$$\mathbf{A}\ \ \mathbf{T6}\ \ \mathbf{E}\ \ \mathbf{COMPLY}\ =\ \mathbf{OBJ}\ \ \mathbf{G}$$

Transform **A** maps from the object base frame to the manipulator base frame; **E** is the transform from **T6** to the point at which the object is grasped; **OBJ** is location of the object (equal to the “**T6**” of the virtual manipulator controlling the object); and **G** transforms from the object location to the grasp point. On the assumption that the force sensors are located near the grasp points, the **COMPLY** transforms are placed there as well. This example uses two comply transforms, which is not strictly necessary since there are only two real manipulators. However, using two transforms potentially allows us to “balance” the interactions between the two robots and achieve true distributed load sharing.

No attempt will be made to discuss here the ways in which the comply transforms can be computed to achieve cooperative control between manipulators, as this is still an active area of research. In particular, the RCCL site at the Jet Propulsion Laboratory is studying problems of this sort.

8.3 Program Example: “TrackII”

A virtual manipulator which is used to drive a slave manipulator is sometimes referred to as an *object* frame. This program demonstrates two manipulators tracking a single object frame. Because the program runs in free space with no kinematic connection between the two arms, there is no need for any of the **COMPLY** transforms discussed above. A diagram of the kinematic relationships is given in Figure 37. The object frame is made to trace sequences of boxes and circles, with both attached robots duplicating these motions.

planning level module

```
#include <rccl.h>
#include "manex.560.h"

extern int circleFxn();

#define CIRCLE_RADIUS   50

main()
{
        JNTS rcclpark1;
        JNTS rcclpark2;
        TRSF_PTR offset, e, track;
        POS_PTR mnpPos, objPos;
        MANIP_PTR mnp1, mnp2, obj;
        float tval, rval;

        char *robotName1 = "Rsun";                          /*#1*/
        char *robotName2 = "Lsun";
```

```
rcclSetOptions (RCCL_ERROR_EXIT);

if (!getRobotPosition (rcclpark1.v, "rcclpark", robotName1))
 { printf ("position 'rcclpark' not defined for robot\n");
   exit(-1);
 }
if (!getRobotPosition (rcclpark2.v, "rcclpark", robotName2))
 { printf ("position 'rcclpark' not defined for robot\n");
   exit(-1);
 }

e = allocTransXyz (NULL, UNDEF, 0.0, 0.0, TOOLZ);
track = allocTrans (NULL, UNDEF);
offset = allocTrans (NULL, UNDEF);

mnp1 = rcclCreate (robotName1, 0);
mnp2 = rcclCreate (robotName2, 0);
obj = rcclCreate ("foo", 0);                          /*#2*/

mnpPos = makePosition (NULL, T6, e, EQ, offset, obj->t6, TL, e);
objPos = makePosition (NULL, T6, EQ, track, TL, T6);

rcclStart();

movej (mnp1, &rcclpark1);
movej (mnp2, &rcclpark2);
waitForCompleted (mnp1);                              /*#3*/
waitForCompleted (mnp2);

setMod (mnp1, 'c');
setTime (mnp1, F_DEFAULT, F_UNDEF);
solveTrans (offset, mnpPos, offset, mnp1->t6);
move (mnp1, mnpPos);
setMod (mnp2, 'c');
setTime (mnp2, F_DEFAULT, F_UNDEF);
solveTrans (offset, mnpPos, offset, mnp2->t6);
move (mnp2, mnpPos);                                  /*#4*/

getCartVel (mnp1, &tval, &rval);
setCartVel (obj, tval, rval);
getCartAccel (mnp1, &tval, &rval);
setCartAccel (obj, tval, rval);                       /*#5*/

while (1)
 { distance (obj, "dx", 50.0);
   move (obj, obj->last);
   distance (obj, "dz", -100.0);
   move (obj, obj->last);
   distance (obj, "dx", -100.0);
   move (obj, obj->last);
   distance (obj, "dz", 100.0);
   move (obj, obj->last);
```

```
            distance (obj, "dx", 50.0);
            move (obj, obj->last);                              /*#6*/

            waitForCompleted (obj);

            setTime (obj, F_DEFAULT, 2000.0);
            transMotionEval (track, circleFxn, CIRCLE_RADIUS, obj);
            move (obj, objPos);                                 /*#7*/
            move (obj, obj->park);                              /*#8*/
        }
}
```

control level module

```
#include <rccl.h>
#include <fastmath.h>
#include "manex.560.h"

circleFxn (t, radius, mnp)
TRSF *t;
int radius;
MANIP *mnp;
{
        float scale;

        scale = motionScale(getActiveMotionId(mnp));

        t->p.y = radius * (COS (PIT2*scale) - 1.0);
        t->p.x = radius * (SIN (PIT2*scale));
}
```

NOTE – this example has been coded for the PUMA 560 and lives at `trackII.560.c` *and* `trackIICtrl.560.c` *in* `demo.rccl`. *An equivalent program for the PUMA 260 lives at* `trackII.260.c` *and* `trackIICtrl.260.c`.

The names of the two robots are “Rsun” and “Lsun” (`#1`). The name of the object frame, in keeping with tradition, is “foo” (`#2`). The position `mnpPos`, defined as

$$\mathbf{T6\ \ E\ =\ OFFSET\ \ OBJ}$$

connects each manipulator to the object frame (`#3`). **OFFSET** is set to the initial value of $\mathrm{T6\ \ E}$. The object frame itself is moved either with relative motions specified with respect to `obj->last`, or by changing the value of **TRACK** in the simple target position `objPos`:

$$\mathbf{T6\ =\ TRACK}$$

After the trajectory generator is started, both robots are moved to their usual starting positions. The `OFFSET` transform in `mnpPos` is then updated for each of them, and both are bound to position

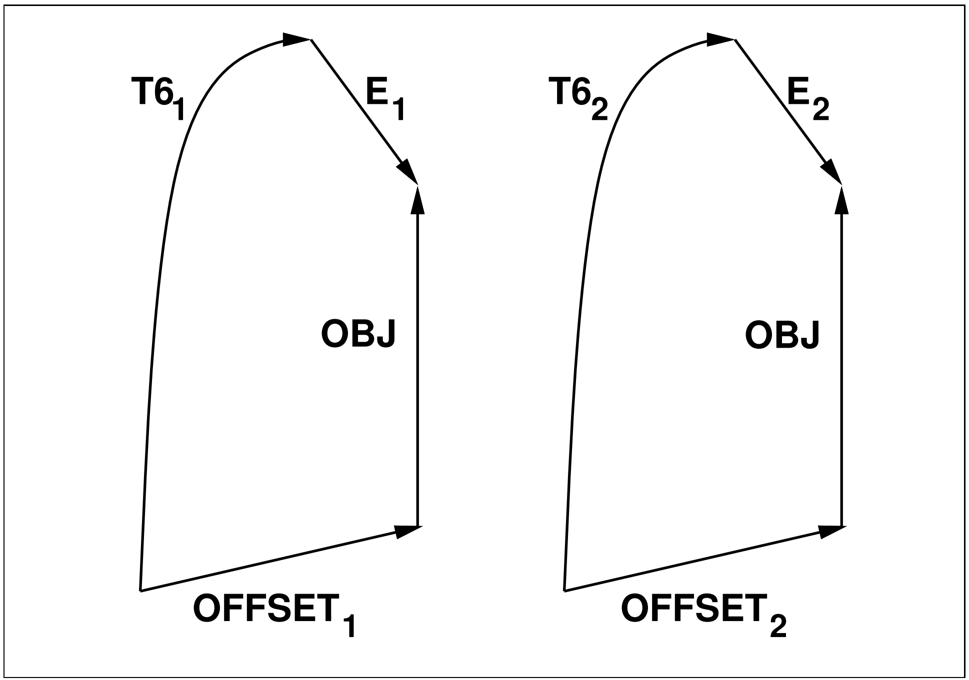

Figure 37: Kinematic diagram for the program “trackII”.

`mnpPos` indefinitely (`#4`). Before each motion request is issued, the transform `offset` is instantiated for each robot so that `mnpPos` defines its current location. The velocity and acceleration limits for the object frame are then set equal to those for the first manipulator (`#5`) (both robots are assumed to be of the same type).

The program next executes a loop in which the object frame is made to first trace out a box, and then a circle. The “box” motion is accomplished with a series of relative motions (`#6`). The “circle” motion is done by binding the transform **TRACK** to a circle tracing function (`#7`). The last motion in the loop is a move to `obj->park`, which ensures that the next position of the object frame does not drift from one iteration of the loop to the next (`#8`) (the park location for virtual manipulators is the identity transform).

9. Other Features

9.1 Program Modes and Options

Multi-RCCL provides several program-selectable options which will cause the program and the manipulators to behave in different ways.

9.1.1 Options

Options are used to control behavior associated with the entire program. Each option has an associated bit code, and may be set or read back with the routines

```
unsigned long rcclSetOptions (mask)
  unsigned long mask;

unsigned long rcclClearOptions (mask)
  unsigned long mask;

unsigned long rcclPutOptions (mask)
  unsigned long mask;

unsigned long rcclGetOptions (mask)
  unsigned long mask;
```

`rcclSetOptions()` and `rcclClearOptions` set or clear, respectively, the options indicated by `mask`. `rcclPutOptions()` loads all the options at once. `rcclGetOptions()` returns the settings of the options specified by `mask`.

The program options currently implemented include

	Default Value
RCCL_ERROR_EXIT	not set
RCCL_SIMULATE	not set
RCCL_LEAVE_POWER_ON	not set

`RCCL_ERROR_EXIT`, if set, causes most RCCL and RCI functions to print diagnostic information and exit the program in the event of an error. This is useful in top level applications and development work, because it saves the programmer from having to perform explicit error checks on the routines. If a particular function responds to this option, it will be documented as doing so in the reference manual. Setting the option also causes the RCI option `EXIT_ON_ERROR` to be set.

`RCCL_SIMULATE`, **if set before** the first call to `rcclCreate()`, will cause the program to run in simulator mode (see Section 9.3). (If it is set after the first call to `rcclCreate()`, it will have no effect.)

RCCL_LEAVE_POWER_ON causes the robot arm power to be left on when rcclRelease() is called, regardless of the setting of that function's powerOn argument. The power will still be switched off if the control task is terminated by an RCI abort.

9.1.2 Modes

Modes control activity specific to a particular manipulator. They are set and read with routines similar to those for the RCCL options:

```
unsigned long rcclSetModes (mnp, mask)
  MANIP *mnp;
  unsigned long mask;

unsigned long rcclClearModes (mnp, mask)
  MANIP *mnp;
  unsigned long mask;

unsigned long rcclPutModes (mnp, mask)
  MANIP *mnp;
  unsigned long mask;

unsigned long rcclGetModes (mnp, mask)
  MANIP *mnp;
  unsigned long mask;

unsigned long rcclGetActiveModes (mnp, mask)
  MANIP *mnp;
  unsigned long mask;
```

mnp is the manipulator with which the modes are associated. rcclSetModes() and rcclClearModes set or clear, respectively, the modes indicated by mask. rcclPutModes() loads all the modes at once. rcclGetModes() returns the settings of the modes specified by mask. Not all mode settings actually take effect immediately; some take effect only with the next requested motion. Because of this, rcclGetModes() may not necessarily return the mode settings that are currently active; this information should instead be obtained with rcclGetActiveModes().

The following modes are currently implemented:

	Takes Effect	Default Value
TRACKING_MODE	immediately	not set
INTEGRATE_MODE	immediately	not set
T6O_EVAL	immediately	not set
COMPLETE_AT_TB	with next motion	set
GRAVITY_FEEDFORWARD	immediately	not set

TRACKING_MODE causes the trajectory generator to continue tracking the last target position when it runs out of motion requests to execute. It is useful when we don't want a "break in the action" to occur in the event of a time gap between successive motion requests.

INTEGRATE_MODE enables integration in the PID servo loop controlling the robot's actuators. This option is defined only for real manipulators running on systems where the attached servo controller box does in fact allow integration to be enabled or disabled. It is principally intended for PUMA robots connected to the Unimation controller. Enabling integration reduces the steady-state servo error but can also reduce stability, particularly in cases where a higher level feedback loop is implemented within the trajectory generator.

T6O_EVAL causes the trajectory generator to maintain the *observed* values of **T6** and the joint values in the t6o and j6o fields of the MANIP structure. This is normally not done because the information is frequently not needed and is expensive to compute.

COMPLETE_AT_TB tells the trajectory generator that the time at which one motion officially ends (and another begins) is the *midpoint* of the transition to the next motion (except for "stop" requests; see below). At this time, the MANIP structure's here field is updated, the motions' status flags are set correctly, computations which have been requested with updateTrans() are carried out, and any end-of-motion signal that has been requested with setMotionSig() is delivered. If this option is **not** set, then the time at which a motion officially ends is the *beginning* of the transition to the next motion. "stop" requests form an exception to this option: stop motions are always considered to end at the beginning of the transition to the next motion.

GRAVITY_FEEDFORWARD causes the trajectory generator to compute the gravity loading for the manipulator and feed it forward to the servo level controllers. This option is very site specific and was implemented specifically for the Jet Propulsion Laboratory.

9.1.3 The Parameter File

Some program control parameters are defined in an ASCII file that is read by the system each time an RCCL program starts up. The file is named .rciparams, and the system looks for the first instance of it in the directories listed in the RCCL_PATH_<Target> environment variable (Section 9.4.1).

A typical .rciparams file looks like this:

```
# system parameter defs for rci
name = RCISYS   ARBCLOCK=5.0
#
# parameter set used by utility programs
name=sysprogs   cpus=0x01 scheduling=ON_TRIGGER interval=20
#
# parameter set used by RCCL
name=RCCL       cpus=0x3 simulate=0 interval=30 timeout=10
```

Not all of the things in this file are relevant to RCCL; the parameter sets named RCISYS and sysprogs are used by other parts of the RCCL/RCI system. In the example here, the RCCL parameters are at the bottom and are denoted by the construction name=RCCL. All of the parameters have the form <field>=<value>, where <field> is a field name and <value> is either an integer, a float, or a string (delimited by either white space or by quotes ("")). In the above example, the parameters `cpus`, `simulate`, `interval`, and `timeout` are defined.

In general, RCCL recognizes the following parameters:

```
cpus         (hex integer)
interval     (integer)
speed        (float)
tvel         (float)
rvel         (float)
simulate     (boolean integer OR string)
timeout      (integer)
scheduling   (string)
leavePowerOn (boolean integer)
```

`cpus` is a bitmask describing which CPUs are available for running the trajectory generator tasks. On single CPU implementations, this parameter must have the value `0x1`.

`interval` defines the sample interval (in milliseconds) at which the trajectory generator is run. It is the same quantity which can be set or obtained within the program by `rcclSetInterval()` or `rcclGetInterval()`

`speed` is an optional velocity scale factor; if specified, all the velocity settings in the program are multiplied by it. For instance, setting `speed=0.1` will cause manipulator motions to run roughly 1/10 as fast (except for cases where the motion time is set explicitly with `setTime()`). This scaling factor is applied on top of the program scaling factor controlled by `setSpeed()`.

`tvel` and `rvel` are optional parameters which define the default translational and rotational velocities for Cartesian interpolated motions. These are the same quantities controlled by the primitive `setCartVel()`. If these parameters are not set, the program uses the defined constants `DEFAULT_TRANS_VEL` and `DEFAULT_ROT_VEL`.

`simulate` is an optional parameter, which, if set to a non-zero integer, causes the program to run in simulator mode (which has the same effect as setting the option `RCCL_SIMULATE` at the beginning of the program). The program will attempt to connect its robots to a simulator program which is running on the current machine and has the name of the current user. To explicitly specify either the name of the simulator, or the machine on which it is running, the `simulate` parameter can be given as a string instead. The format used by this string is described in `simAddress(5)`.

`timeout` is an optional parameter which defines the length of time the trajectory generator task(s) will wait for a response from the robots before deciding something is seriously wrong and aborting the program. The units of `timeout` are control cycles, so if the control level is being run at 50 msec. (*i.e.*, `interval` is 50), and `timeout` is set to 10, then the system will wait 500 msec. before issuing a timeout abort. The default value of `timeout` is 2. If the system real-time response is poor and this value needs to be boosted, the highest reasonable value is probably around 10.

`scheduling` defines the scheduling discipline used by the trajectory generator task(s). The default scheduling discipline is `ON_TRIGGER`. Other scheduling disciplines are `ON_CLOCK` and `ON_FUNCTION`. It is also possible to set the scheduling discipline from within an RCCL program by calling `rcclScheduling()` before the first call to `rcclCreate()`. Consult the manual page `RCCL_params(5)` for details on what these scheduling disciplines are all about.

leavePowerOn causes the RCCL_LEAVE_POWER_ON option (see above) to be set during the first call to rcclCreate(). This will cause the robot power to be left on after the program finishes, providing the program does not terminate with an RCI abort.

More information about the RCCL parameters is given in the manual page RCCL_params(5). Information about the format of the .rciparams file may be found in the manual page rciParameters(3).

9.2 Error Handling and Recovery

9.2.1 The Error Stack

Most RCCL and RCI routines return either NULL or -1 if they fail, depending on whether the routine normally returns a pointer or an integer. Also, they will usually place diagnostic information on the task's *error stack*. This is a special buffer used for storing error codes and messages. The idea is that when some internal routine fails, it places a code and a message on the error stack; when higher level routines discover the error, they too can place information on the error stack. When the error is discovered at the top level, the error stack contains a fairly complete trace of what went wrong and where.

A typical thing for a top level application to do when it finds that a routine has returned an error is to print the contents of the error stack and exit. This is done with the routine printErrors(), and is commonly coded like this:

```
if (routine() == -1)  /* error indicated by -1 in this case */
 { printErrors();
   exit(-1);
 }
```

NOTE: At the control level, it is best to use rciPrintErrors() in place of printErrors(). See below.

One should verify that a routine does in fact use the error stack before calling printErrors(); otherwise, the error stack is likely to be empty and nothing interesting will be printed. When in doubt, consult the reference manual entry for the routine in question.

Application software can also add its own error codes and messages to the error stack. The common way of doing this is to use the routine errorMsg():

```
errorMsg (code, format, values ...)
  int code;
  char *format;
```

The error code is the first argument to the routine. The message is specified by a printf() style format string (second argument) followed by a variable number of arguments containing values referred to by the format string. As an example of using the error stack, consider the following program:

```
main()
{
        if (routine1() == -1)
         { printErrors();
           exit (-1);
         }
}

routine1()
{
        char *fxnName = "routine1()";

        if (routine2() == -1)
         { errorMsg (1, "%s -- function call failed\n", fxnName);
           return (-1);
         }
        return (1);
}

routine2()
{
        char *fxnName = "routine2()";

        errorMsg (1, "%s -- not written yet!\n", fxnName);
        return (-1);
}
```

Running this will produce the following output:

```
routine1() -- function call failed
routine2() -- not written yet!
```

If, instead of exiting, the application software wishes to continue running, it should clear the error stack by calling `clearErrors()`. Interactive programs will frequently print the error stack, clear the errors, and continue running, as in

```
if (routine() == -1)
 { printErrors();
   clearErrors();
 }
```

It is also possible to reference the error codes on the error stack. These are primarily intended for use by software (versus messages, which are mainly for users to look at). In cases where software continues to run after detecting an error, error codes may be useful in helping the software decide what sort of action to take. The routine `getErrorCode(n)` returns the error code from the n-th position on the stack, where 0 corresponds to the information most recently added. Application code which uses the error codes typically looks something like this:

```
if (routine() == -1)
 { if (getErrorCode(0) == ENotSetup)
    { setUpData();
    }
 }
```

Error codes are also sometimes "passed on" when a calling routine adds its own information to the error stack:

```
if (routine() == -1)
 { errorMsg (getErrorCode(0), "routine() failed\n");
   return (-1);
 }
```

The error codes used by RCCL/RCI are defined in the file `<errorCodes.h>`. Error codes aren't used as much as the error messages.

Each RCI task has its own private error stack, which becomes the current stack when the task's context is invoked. If you call `errorMsg()` from the planning level and then from the control level, the first message will go to the standard error stack, and the second will go to the control task's error stack.

To explicitly reference the error stack for an RCI task from *outside* that task's context, one may use the routines

```
rciPrintErrors (td)
rciErrorMsg (td, code, fmt, ...)
rciClearErrors (td);
rciGetErrorCode (td, n)
```

These behave identically to the error stack routines described above, except that they use the error stack of the RCI task whose descriptor is specified by the additional first argument. If the supplied task descriptor is `NULL`, then the current error stack is used. Recall that to get the descriptor for the RCI task associated with a particular manipulator `mnp`, one may use the macro `MANIP_TASK(mnp)`.

When an error occurs at the control level, and one wishes to exit the program, it is necessary to call `rciAbort()` instead of `exit()` (see Section 9.2.2). There is a special form of `rciAbort()`, called `rciAbortErrors()`, which transfers the messages on the error stack into the abort message buffer and then calls `rciAbort()` itself. A piece of code that invokes this looks like:

```
if (routine() == -1)
 { rciAbortErrors();
   return;
 }
```

Care should be taken to not let errors from routines that use the error stack go undetected, or information will collect on the stack. The error stacks for the control tasks are cleared automatically at the start of every control cycle. For more information on the error stack, see the manual pages `errorStack(3)` and `rciErrorStack(3)`, or the *RCI User's Guide*.

9.2.2 Aborting

RCI provides an abort utility which may be used to force an orderly exit of the program in the event of some abnormal condition. The routine is called

```
rciAbort (code, format, values ...)
  int code;
  char *format;
```

and it may be called from either planning or control level. The first argument specifies a code associated with the abort, while the second and following arguments specify an abort message using a `printf()` style format string and a variable number of value arguments.

Internally, `rciAbort()` causes a `SIGHUP` signal to be sent to the planning level, where it is caught by RCI system software. All control tasks are released, which (for RCCL) means the trajectory generator is switched off. The default action is then to print the abort message, along with some other information (which RCI task called the abort, what CPU was it running on, how many control cycles had elapsed) and exit the program.

As an example, consider the following simple program:

```
#include <rccl.h>

abortAtOnce (arg, mnp)
int arg;
MANIP *mnp;
{
        rciAbort (10, "Aborting for fun, arg=%d\n", arg);
}

main()
{
        MANIP *mnp;

        mnp = rcclCreate ("foo", 0);
        rcclStart();
        printf ("Here we go ...\n");
        runMonitorFxn (mnp, abortAtOnce, 123);
        while(1);
}
```

This will produce the following output and exit:

```
Here we go ...

RCI ABORT: task 'RCCL1' on cpu 1, cycle 6
Aborting for fun, arg=123
```

Aborts may be issued by the application code at any time. They may also be issued by the trajectory generator if the manipulator hits a joint limit or encounters a singularity, or some other aspect of the control process fails. Aborts will also be issued sometimes in the case of internal software failure.

More information on `rciAbort()` may be found in the manual pages, or in the *RCI User's Guide*.

9.2.3 Catching Aborts Yourself

As seen in the previous section, the default action for an abort handler to is to print information about the abort and exit the program. This is often satisfactory, but for some applications the user may wish to catch aborts and process them within the program.

The RCI routine

```
(*rciAbortHandler)(handlerFxn)
  int (*handlerFxn)();
```

specifies an application defined function to be called when an abort occurs and returns a pointer to the previous handler. The system will still catch the abort signal and shut down the trajectory generator and other RCI tasks, but it will call the handler function instead of printing the abort information and exiting. The handler is called with the following arguments:

```
handler (code, msg, td)
int code;
char *msg;
RCI_DESC *td;
{
        ...
}
```

`code` and `msg` are the code and message specified by the call to `rciAbort()`. `td` points to the descriptor of the RCI task which initiated the abort.

The handler can do various things. For example, it can print out the abort information and exit the program, similar to what the default handler does (although establishing the handler might then be somewhat pointless). It might be used to print out extra debugging information before exiting; this is a common application when systems are under development. Or it might just return, in which case the program goes on as before, except the trajectory generator has been shut off and must be started again by calling `rcclStart()`.

Usually, if it desirable to keep executing the program after an abort, the best way to do this is to `longjmp()` out of the abort handler to some defined checkpoint and restart the trajectory generator from there. If you don't know what `longjmp()` is, then you should find out (check either a C programming guide or the UNIX manual page for `longjmp(3)`, or ask whichever programmer at your site has the longest hair).

An example of using an abort handler with `longjmp` is given by the following program:

```
#include <rccl.h>
#include <setjmp.h>

jmp_buf env;

handleAbort (code, msg, td)
int code;
char *msg;
RCI_DESC *td;
{
        printf ("\nABORT called\n");
        rcclAbortReset();
        longjmp (env, 1);
}

doAbort (arg, mnp)
int arg;
MANIP *mnp;
{
        rciAbort (10, "");                          /*1*/
}

main()
{
        MANIP *mnp;
        int first = YES;

        rciAbortHandler (handleAbort);
        mnp = rcclCreate ("foo", 0);
        if (setjmp (env))
         { first = NO;
         }
        printf ("Starting ...\n");
        rcclStart();
        if (first == YES)
         { runMonitorFxn (mnp, doAbort, 123);
         }
        while(1);
}
```

The program starts, aborts itself, catches the abort with its own handler, and then starts up again. Notice that the call to `rciAbort()` gives a null message, since no use is made of the message by the handler (#1). Another discussion of using `longjmp()` with `rciAbort()` is found in the *RCI User's Guide*.

There are several caveats that should be observed when doing things with abort handlers. The first is that many of the RCCL routines are not re-entrant (if you don't know what re-entrant is, consult your local system guru). Basically, this means that things might fail if you call an RCCL or RCI routine from a handler while another (or perhaps the same) RCCL or RCI function is in progress in the main program. For instance, consider what might happen if the main program is inside a call to `allocTrans()` when an abort fires. If the handler "long jumps" out to a catch point, the original call to `allocTrans()` will never return properly and some of its internal data structures could be left in an undefined state. Technically, this is a problem with most UNIX packages as well; for instance, the `stdio` library is not guaranteed to function correctly if its routines are exited in this way. Similar problems can occur if one tries to call library routines inside a signal handler.

The RCCL routine `rcclAbortReset()` should be called before doing a `longjmp()` out of a handler. This does a few things to try and fix up the system state, although it is not guaranteed to be perfect. In general, care should be taken not to `longjmp()` out of routines that allocate memory or manipulator objects, such as `allocTrans()`, `allocMem()`, `rcclCreate()`, etc. Motion request primitives are generallysafer because the motion queue is reset when `rcclStart()` is called.

9.2.4 Program Crashes

When you run a regular UNIX program and an exception occurs like a divide by zero or a memory fault, you generally get a message like

```
Floating exception (core dumped)
%
```

or

```
Segmentation fault (core dumped)
%
```

The same thing will happen with an RCCL program, if the exception occurs *within the planning task*. However, what happens if the exception occurs in the *control* task depends on the system.

On Sun Solaris systems (where control tasks are implemented as real-time threads), exceptions at the control level cause the whole program to crash in the same way as an exception at the planning level.

On SGI systems (where control tasks are implemented as light weight processes), exceptions at the control level cause only the control level to crash. However, the RCI system is set up so that a *SIGXCPU* will get delivered to the planning level, which then prints a message similar to

```
Program crash in RCI control level. Check core file.
```

and forces a program exit.

In earlier releases of RCCL/RCI, exceptions at the control level had to be caught be special RCI code embedded in the system kernel. In particular, no core dump was created. Instead, a rather elaborate mechanism was put in place to make a stack trace of the control level and relay this information to the planning task. This is no longer necessary.

9.3 The Simulator

RCCL/RCI supplies a program, named `robotsim`, that provides 3D graphic simulation for one or more robots. These robots can be "connected to" and controlled by an RCCL program. The simulator provides a way to graphically preview programs, single step individual control cycles, and provide simulated sensor inputs.

To use `robotsim`, the user needs to do two things:

1. Start the simulator program running, using a command like

   ```
   % robotsim <robotName>
   ```

   It may be preferable to do this in a separate window, since the simulator normally prints messages while it operates (although this can be suppressed with the `-s` option).

2. Set the RCCL program itself to run in simulator mode. There are several ways of doing this, as discussed in the following section.

9.3.1 Making RCCL Use the Simulator

An RCCL program will run in simulator mode if

```
rcclSetOptions (RCCL_SIMULATE)
```

is called at the very beginning of the program (before the first call to `rcclCreate()`), or if the `RCCL` parameter `simulate` is set to a non-zero value in the `.rciparams` file (see Section 9.1.3). RCCL will then connect its robots to a simulator program instead of to a real robot controller.

Simulator programs have "names" so as to distinquish them from one another and allow different ones to be run on the same system at the same time. The program `robotsim`, for example, normally sets its name to the user name under which it is invoked. Unless told otherwise, RCCL will try to connect its robots to a simulator program which is running on the current machine and has the name of the current user. To specify a simulator running on a different machine or with a different name, the `simulate` parameter can be set to a simulator *address string* instead of a non-zero integer. Similarly, the routine

```
rcclSetSimulator("<address string>")
```

can be called in place of `rcclSetOptions(RCCL_SIMULATE)`. In both cases the address string can specify the desired simulator name, and/or the host machine on which it is running. The format of the address string is described in the manual page `simAddress(5)`.

When running the simulator, particularly on the same machine, it may be desirable to "turn down" the RCCL sample interval, because of the computational load introduced by the simulator. When doing simulation runs, this author usually sets the trajectory generator to run at 100 msec., either by using the `interval` parameter in the `.rciparams` file, or by calling `rcclSetInterval()`.

In simulation mode, the trajectory generator is run off of `SIGALRM` instead of a kernel level interrupt. The advantage of this is that the "control level" is now run in UNIX user mode. In particular, it is possible to set breakpoints in the control level functions and step through them with a debugger. This capability proved invaluable when developing the RCCL system code, particularly the trajectory generator.

One problem is that the simulator's use of `SIGALRM` prevents application software from using this signal. Specifically, it means that application software must not call `sleep()` (which uses `SIGALRM`), but must use `nap()` instead.

9.3.2 Simulator Features

By default, the simulator will create a graphics window containing a representation of the robot(s) being simulated. Graphics are available on systems with Xwindows, SunView, or GL (the SGI graphics library). Xwindows and SunView use wireframe graphics (with backface removal), while GL uses full shaded graphics.

Specific features of the simulator program presently include:

- ◊ the ability to "simulate" more than one robot;
- ◊ a configuration file that can be used to define other graphic objects to be displayed in addition to the robot;
- ◊ an interactive mode which allows the user to single step through control cycles and set various "inputs" to the simulated robot controller;
- ◊ the ability to control the graphics view point by moving a mouse cursor across the window;
- ◊ an optional graphic "teach pendant", attached to the graphics window, which can be used to simulate teach pendant inputs to the RCCL or RCI application.

For detailed information about the simulator program, the manual page `robotsim(1)` should be consulted. The simulator is still in a state of development, and new features are frequently appearing.

If the simulator is run interactively, then there are commands which allow the operator to set sensors to specific values. The interactive mode also allows the operator to single-step through different control cycles, setting different sensor values at different points. This can be useful, but is also tedious, and many applications involving real-world input can be debugged more easily the old fashioned way, using `printf()` (or `rciPrintf()`, as appropriate).

The limitation of the simulator is that it simulates only arm movements: it has no knowledge of the workspace around it or real-world events. Dynamics are only roughly simulated in the form of gravity loadings. However, adding modules to do dynamic modeling or collision detection should not be difficult.

9.3.3 A Sample Program Running in Simulation

The best way to learn about the simulator is to run it, so to this end the reader is encouraged to run the following example program.

planning level module

```
#include <rccl.h>
#include <stdio.h>

extern simMonitor();

extern FILE *setpClf;
extern FILE *setpJlf;
FILE *clf;
FILE *jlf;

main()
{
        TRSF_PTR t;
        POS_PTR p0;
        MANIP *mnp;
        JNTS rcclpark;
        char *robotName;

        rcclSetOptions (RCCL_ERROR_EXIT | RCCL_SIMULATE);       /*#1*/
        rcclSetInterval (100);                                  /*#2*/

        robotName = getDefaultRobot();
        if (!getRobotPosition (rcclpark.v, "rcclpark", robotName))
         { printf ("position 'rcclpark' not defined for robot\n");
           exit(-1);
         }
        t = allocTrans (NULL, UNDEF);
        p0 = makePosition (NULL, T6, EQ, t, TL, T6);

        mnp = rcclCreate (robotName, 0);
        rcclStart();

        runMonitorFxn (mnp, simMonitor, 0);                     /*#3*/

        movej (mnp, &rcclpark);
        waitForCompleted (mnp);
        readTrans (mnp->here, t);                               /*#4*/

        if ((clf = fopen ("clf", "w")) == NULL)                 /*#5*/
         { printf ("Can't open Cartesian log file 'clf'\n");
           exit (-1);
         }
        if ((jlf = fopen ("jlf", "w")) == NULL)
         { printf ("Can't open joint log file 'clf'\n");
           exit (-1);
         }
        setpJlf = jlf;
        setpClf = clf;
        rcclBlock();                                            /*#6*/
```

```
        setMod (mnp, 'c');
        t->p.x += 100.0;
        move (mnp, p0);
        stop (mnp, 1000.0);
        movej (mnp, &rcclpark);
        stop (mnp, 1000.0);
        waitForCompleted (mnp);                                 /*#7*/

        setpJlf = NULL;                                         /*#8*/
        setpClf = NULL;
        fclose (clf);
        fclose (jlf);

        rcclRelease (YES);
}
```

control level module

```
#include <rccl.h>

simMonitor (arg, mnp)
int arg;
MANIP *mnp;
{
        int sigma, tau;

        getActiveMotionCounts (mnp, &tau, &sigma);
}
```

NOTE – this example should run on all PUMA robots. It lives at `sim.c` *and* `simCtrl.c` *in* `demo.rccl`.

This program is essentially the same as the program "simple", except that a few changes have been made to demonstrate the simulator features.

Before running the program, the user needs to set the simulator program running. This is best done with the command

```
% robotsim <robotName> -p rcclpark
```

`<robotName>` should be the name of the default system robot. The option `-p rcclpark` tells the simulator to start with the robot in the `rcclpark` position. This is not essential, but it saves the program from having to first move the robot there. As was indicated above, it is desirable to run the simulator on a separate terminal or in another window, since it will try to print things. To some extent, the simulator emulates the `moper` program that is normally loaded into Unimatation controllers, and so when it starts up it prints the same messages as `moper`:

```
<robotName> Moper started
<robotName> listening
```

Although the simulator handles robots other than PUMAs, it acts as though they all have a Unimate controller.

If running the simulator in another window is impossible, then you probably want to run it in the background, in silent mode so it won't print anything. Do this with the command

```
robotsim <robotName> -p rcclpark -s &
```

The simulator is now running and waiting for a connection from an RCCL (or RCI) program. To put itself into simulation mode, the example program here explicitly sets the `RCCL_SIMULATE` option and sets the control interval to 100 msec. (`#1`, `#2`), so there is no need to make any changes in the `.rciparams` file. When the program starts up, the simulator will come alive by printing

```
<robotName> active (slave)
```

The program moves the manipulator to an initial position, does a straight line motion to a point 100 mm. along the x axis, stops, and returns to the initial position. When the program calls `rcclRelease()`, the simulator will print

```
<robotName> released
<robotName> listening
```

and go back to waiting for another program to connect to it. In particular, the above program can be run again without having to restart the simulator – the simulator program, as a separate process, acts like a running robot controller and responds to whatever RCCL or RCI program tries to connect to it. To exit from the simulator, simply break out of it using the interrupt character.

The simulator allows the RCCL system to create trajectory log files in either joint or Cartesian coordinates. This is the purpose of the variables `setpJlf` (for "setpoint joint log file") and `setpClf` ("setpoint Cartesian log file"). Whenever either of these is set to a valid file pointer, and the trajectory generator is running in simulation mode, then the output setpoints for each control cycle are printed into the respective file. Information is printed to `setpJlf` in joint coordinates, using the format

```
<mcount>[T] <jval1> <jval2> <jval3> ...
```

where `mcount` is the motion count for that cycle, `'T'` is printed if the motion is in transition, and the `<jvaln>` are the joint values (degrees for rotational joints and millimeters for prismatic joints). Information is printed to `setpClf` in Cartesian coordinates, for which the format is similar:

```
<mcount>[B] <px> <py> <pz> <ang> <ux> <uy> <uz>
```

`px`, `py`, and `pz` are the translational coordinates, in millimeters, `ang` is the equivalent angle of rotation, in degrees, and `ux`, `uy`, and `uz` are the normalized coordinates of the equivalent axis of rotation. This representation is used for rotations because it is less ambiguous than Euler angles. A special line is also printed into each log file every time a new motion segment begins, giving information about what sort of motion segment it is and what some of its parameters are. Trajectory logging continues until the file pointers are cleared by the planning level.

The program opens the files "jlf" and "clf" at (#5); the setpJlf and setpClf variables are set at (#6). The call to rcclBlock() makes sure that the trajectory generator has noticed that they are set and has started recording before the program proceeds further. When the program has finished executing, the files should be in the current directory waiting for examination.

9.4 User Environment and Compilation

This section gives a brief description on how to compile and run RCCL programs. ‘¡Some of this description is implementation specific. All of the information given here is supplied in greater depth in the *RCCL/RCI Startup and Installation Guide*, and in case of conflict, the latter should take priority.

9.4.1 The User's UNIX Environment

An example of the required user environment will be given here. For this example, it will be assumed that the RCCL distribution is located at /local/rccl, and the binaries and libraries have not been moved away from the distribution tree.

As described in section 1.5, <Target> denotes the name of your executing target system; e.g., Sun4sol for Sun/Solaris, Sgi for SGI/IRIX 5.X, etc. More detailed information is given in the RCCL/RCI Startup and Installation Guide.

Given the above assumptions, the necessary environment can be set up with the following entries in a .cshrc file:

```
setenv  RCCL /local/rccl   # set this for convenience ...

set path=($RCCL/bin/<Target> $RCCL/bin/generic $path)
setenv  MANPATH "$MANPATH":"$RCCL/man"

setenv  RCCL_PATH_<Target> .:$RCCL/lib/<Target>:$RCCL/lib/generic
setenv  RCCL_LIB_GENERIC $RCCL/lib/generic
```

This accomplishes the following:

The path variable contains a list of directories in which your shell looks for executable programs. Adding <RCCL>/bin/<Target> and <RCCL>/bin/generic to this variable allows your shell to locate standard RCCL/RCI programs.

Adding <RCCL>/man to your MANPATH environment variable allows the man command to locate the RCCL/RCI online manual pages (MANPATH may have a different name on some UNIX systems; if necessary, check the documentation for your man command).

The RCCL_PATH_<Target> environment variable specifies a path in which executing RCCL/RCI applications search for the .rciparams file, the .robotsim file (used by the simulator), and the files defaultRobot.cfg and robotIO.cfg. Also, the system checks this path for subdirectories named robotData in which to locate the .jls, .kyn, and .pos files for specfic robots.

The RCCL_LIB_GENERIC environment variable is used by a few RCCL utility programs.

9.4.2 Starting Up the Robot

This section summaries the initialization sequence for PUMA robots (with Mark I,II, or III controllers) attached to your RCCL/RCI system. If your system has been configured to interface to another kind of robot, the initialization sequence will be different.

Again, much more detailed information on this is given in the *RCCL/RCI Startup and Installation Guide*.

1. *Turn on the controller.*

2. *Get out of VAL and into ODT.* ODT is the LSI11 prom monitor. The quicket way to get into ODT from VAL is to toggle the controller's "run/halt" switch.

3. *Load the RCCL/RCI communication software into the robot controller.* This software is generically called `moper`, and for any particular robot, it's binary has the same name as the robot and is usually located in `lib/generic/moper` relative to the root directory of the RCCL/RCI distribution tree. So for instance, if your robot is called "Jack", then the moper program will be located in `lib/generic/moper/Jack`. Generic `moper` programs exist also, with names like `puma560` or `puma760`. The `moper` program is downloaded from the host machine through the controller's serial console line, using a program called `down`. The controller is assumed to be initially be in ODT. `down` takes several arguments and its invocation is usually bundled into an alias, such as

   ```
   alias   loadJack   'down -d /dev/ttya -e 10000 -a 10000 \
                       <RCCL>/lib/generic/moper/Jack -t'
   ```

   The `-d` argument specifies the name of the serial port connected to the controller console line (assumed here to be `/dev/ttya`); `-e` and `-a` specify, in octal, the entry and load address points, with `10000` being the standard value for both; there is an unoptioned argument specifying the file containing the `moper` executable; and `-t` tells moper to exit after it has started `moper` up.

3. *Calibrate the robot.* This is done using the standard RCCL/RCI utility `pumacal`, which is described in the manual pages.

It should be noted that it is generally unecessary to repeat the above sequence unless the robot controller is switched off. In particular, `moper` will continue to run and the arm will remain in calibration regardless of the state of the UNIX host system. If for some reason `moper` is not running on the controller, then any attempt to execute an RCCL/RCI program will result in an abort with a set of messages that look something like:

```
RCI ABORT: task 'RCCL1' on cpu 1, cycle 0
rbtStartup() -- cannot connect to robot
priAttach() -- other end not listening
```

9.4.3 Program Compilation

This section describes how to compile and load RCCL/RCI programs. The root directory will be denoted

9.4.3.1 Using non-ansi compilers

If you are compiling RCCL/RCI applications using a non-ansi C compiler, you may need to provide the following definitions on the command line:

`-DNO_PROTOTYPES`
: Define this if your compiler does not understand prototypes (which applies to some compilers on SunOS and BSD systems).

`-Dconst=`
: Define this if your compiler does not understand `const` (which applies to some compilers on SunOS and BSD systems). Note that the = sign is necessary.

`-Dvoid=int`
: Define this if your compiler does not understand `void` (which applies to some compilers on BSD systems).

Of course, these definitions can be made directly in your code, using `#define` directives. However, if you do so, make sure such definitions *precede* any RCCL/RCI include files.

Also, if you are using a cross-compiler to build RCCL code for a different target, the command line arguments for defining preprocessor symbols *may* be different; consult the manual for your cross-compiler.

9.4.3.2 The `rcc` command

Earlier versions of the systems required that programs be compiled using a special interface to the C compiler, called the `rcc` command. This command is no longer necessary and is not supported.

Earlier versions of the system also required that modules containing variables and code accessed by the control level needed to be compiled into modules separate from those containing planning level code. This restriction no longer exists, but the convention of separating control and planning level code has been retained.

9.4.3.3 Include files

The include files used by RCCL are located in `<RCCL>/include`. To reference them, the compiler directive `-I<RCCL>/include` should be used.

9.4.4 Other compiler options

On Sun4/Solaris systems, `cc` should be given the following option:

`-D_REENTRANT`

9.4.4.1 Libraries

RCCL programs need to be linked with `librciUser.a` and `librciCtrl.a`, in that order. These are located in `<RCCL>/lib/<Target>` (or wherever `lib/<Target>` has been installed). See Section 1.5 for a description of `<Target>`. The math library `-lm` is also required. On Sun Solaris targets, `-lsocket`, `-lnsl`, and `-lthread` are also required, with `-lthread` being specified after all other libraries. If the program uses the C-tree matcher supplied with RCCL, then `libCtree.a` is needed as well, before the references to `librciUser.a` and `librciCtrl.a`.

Commamd line library specification is made easier by specifying `<RCCL>/lib/<Target>` using either the `-L` option or the `LD_LIBRARY_PATH` environment variable (assuming that these are supported by your compiler/loader). RCCL libaries can then be specified using `-l`, with the leading `lib` and trailing `.a` suffix dropped.

On an `Sgi` or `Sun4` target, the following library sequence should be used:

```
-lCtree -lrciUser -lrciCtrl -lm
```

On a `Sun4sol` target, the following additional libraries should be appended to the above sequence:

```
-lsocket -lnsl -lthread
```

Again, if you are using a cross-compiler/loader, the loading conventions and options may be different.

9.4.5 The Utility Programs

There are a set of programs available for moving the robot arm around (`move`), calibrating it (`pumacal`), putting its joints in limp, or "free" mode (`free`), putting the joints in weightless mode (`zerograv`), and measuring calibration parameters (`primecal` and `potcal`). These programs are **not** written using RCCL; instead, they are written using a separate, joint-level-only, trajectory generator called CHASE, which lives in the directory `src/chase`. Each of these programs is described in the *RCI User's Guide*, as well as in individual manual pages.

In general, the utility programs take the name of the robot to be controlled as an argument. If no name is given, then the name of the system default robot, defined in the file `defaultRobot.cfg`, is assumed.

There is also a general purpose calculator program, called `arc` (which stands rather lamely for "A Robotics Calculator"). It is something of a cross between a basic infix calculator, RCCL, and "Matlab". It permits evaluation of arithmetic expressions (including matrices), variable assignment, and various functions, including many of the RCCL functions for manipulating transforms and vectors, as well as the kinematic functions for specific robots.

9.4.6 The Utility Routines

RCI provides some utility routines which may be used by RCCL applications. Only a couple of the more useful ones will be described here. More information may be obtained from either the *RCI Reference Manual* or the *RCI User's Guide*.

The routine

```
char *getDefaultRobot()
```

reads the file `defaultRobot.cfg` (located in the path specified by the environment variable `RCCL_PATH_<Target>`) to obtain the name of the *default* system robot. This is mainly useful for sites which make predominant use of one robot.

The routine

```
getRobotPosition (jvalues, posName, robotName)
  float *jvalues;
  char *posName;
  char *robotName;
```

reads a file named `robotData/<robotName>.pos` (located relative to the path specified by the environment variable `RCCL_PATH_<Target>`) and looks for a set of joint values named `posName`. If found, they are returned in the array `jvalues`.

9.4.7 Limitations

There are a few things RCCL users should avoid, or at least be wary of:

control level code –
: Because control level code runs at high priority and in real-time, one should be careful about performing time consuming operations there (particularly I/O). Also, infinite loops at the control level may hang the machine.

keeping the machine unloaded –
: The new release of RCCL relies on intrinsic real-time support provided by Sun Solaris and SGI IRIX. This support is reasonable for tasks executing as fast as 100 Hz. However, the real-time performance can degrade if the machine is loaded with other processes (particularly those which consume memory). For best results, try to avoid memory intensive operations (including compilation) while running RCCL/RCI applications.

unsafe routines –
: A number of RCCL/RCI routines can only be called from the planning level, for various reasons concerning implementation and possible data collisions. If a routine cannot be called from the control level, then this is indicated in the "Limitations" section of that routine's manual page.

signals – Do not use the UNIX signals `SIGHUP` or `SIGXCPU`. These have been expropriated by RCI for signaling abort and crash conditions. Also, when running with the simulator, do not use the signal `SIGARLM`. The simulator uses this to control the trajectory task. This also means that one cannot use `sleep()` when running an RCCL program under the simulator, since this uses `SIGALRM`. Instead, the routines `delay()` or `nap()` may be used.

References

[Allworth 81] S.T. Allworth, *Introduction to Real-Time Software Design*. MacMillan Press, Ltd., London, 1981.

[Hayward and Paul 1986] Vincent Hayward and Richard Paul, "Robot Manipulator Control Under UNIX: RCCL, a Robot Control C Library". *International Journal of Robotics Research*, Winter, pp. 94 – 111. (Vol. 5, No. 4)

[Kernighan and Ritchie 1978] Brian K. Kernighan and Dennis Ritchie, *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1978.

[Lee 1982] C.S.G. Lee, "Robot Arm Kinematics, Dynamics, and Control". *Computer*, December 1982, pp. 62 – 80. (Vol. 15, No. 12)

[Lloyd, et al 1988] John Lloyd, Mike Parker, and Rick McClain, "Extending the RCCL Programming Environment to Multiple Robots and Processors". *IEEE Conference on Robotics and Automation*, Philadelphia, Pa., April 24-29, 1988, pp. 465 – 469

[Lloyd 1985] John Lloyd, "Implementation of a Robot Control Development Environment", (M. Eng. Thesis). Dept. of Electrical Engineering, McGill University, Montreal, Canada, December 1985.

[Paul 1981] Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press, Cambridge, Mass., 1981.

[Paul and Shimano 1976] Richard P. Paul and Bruce Shimano, "Compliance and Control". *Proceedings of the Joint Automatic Control Conference*, West Lafayette, Indiana, July 27 – 30, 1976, pp. 694 – 699.