Extending the RCCL Programming Environment to Multiple Robots and Processors

John Lloyd and Mike Parker
McGill University

Rick McClain
GE/RCA Advanced Technology Laboratories
Route 38, Moorestown, New Jersey, USA 08057

September 1987

McGill Research Centre for Intelligent Machines
McGill University
Montréal, Québec, Canada

Postal Address: 3480 University Street, Montréal, Québec, Canada H3A 2A7
Telephone: (514) 398-6319 Telex: 05 268510 FAX: (514) 283-7897
Network Address: mcrcim@larry.mcrcim.mcgill.edu

RCCL/RCI Release 4.2, April 17,1992

Extending the RCCL Programming Environment to Multiple Robots and Processors

John Lloyd and Mike Parker
McGill University

Rick McClain
GE/RCA Advanced Technology Laboratories
Route 38, Moorestown, New Jersey, USA 08057

Abstract

The Robot Control C Library (RCCL) is a system for developing robot control programs in a UNIX environment, and has proven to be particularly useful in research applications. This paper contains a description of the work presently being undertaken by the RCA Advanced Technology Laboratory and McGill University in expanding RCCL to handle multiple robots, and upgrading its implementation to a multiple-processor system. This work includes 1) reworking the RCCL primitive set to allow for the specification of multiple robot actions, 2) modifying its trajectory generation mechanism so that several robots may be controlled and coordinated at once, and 3) redesigning the system interface on top of which RCCL is built to allow the creation of multiple real-time robot control tasks interfaced to UNIX. The paper finishes by outlining the implementation of this system on a MicroVAX II configured with multiple CPUs on the same backplane.

Contents

1 Introduction **1**

2 Background: the RCCL/RCI environment **2**

3 Enhancing RCCL for Multiple Robots **5**
3.1 Multiple Robot Control Issues 5
3.2 New Function Primitives 6
3.3 Task Implementation 8
3.3.1 Dual Arm Example 9

4 RCI: Primitives for Creating Real-Time Control Tasks **11**
4.1 The single robot RCI system 11
4.2 Requirements for prototyping robot control tasks 13
4.3 Principle design concept of RCI 13
4.3.1 Task Structure 13
4.3.2 Task Images 14
4.3.3 Task Creation and Deletion 14
4.3.4 Scheduling 15
4.3.5 Shared Memory and Message Passing 16
4.3.6 Support Functions 17
4.3.7 Robot interfacing aspects 17

5 Implementation of RCI Using a set of MicroVAX II CPUs **17**
5.1 The MicroVAX II multicomputing capability 17
5.2 The UNIX interface and the auxiliary CPU minikernel 18
5.3 The RCI Implementation 19
5.3.1 Process initialization 19
5.3.2 Scheduling 19
5.3.3 Maintaining the real-time clock 20

6 Conclusion **20**

1. Introduction

This paper presents the system work currently being done at the RCA Advanced Technology Laboratories (ATL), Moorestown, New Jersey, in cooperation with the McGill University Research Center for Intelligent Machines (McRCIM) Montreal, Canada, in exploring cooperative multi-robot control and programming environments. This work began last year when ATL implemented the Robot Control C Library (RCCL) [Hayward and Paul 1986] at their site for doing research on robot force control techniques. RCCL is a portable library of robot control C routines, usually implemented under UNIX using a package of system primitives called the Robot Control Interface (RCI), which makes it possible to create a real-time robot control task and connect it to a UNIX program.

The multi-robot control work at ATL is motivated by its potential advantages in manipulating large objects, or performing complex tasks that require more than one hand [Hayward and Hayati 1987]. Many of the specific applications being studied are directed at NASA programs for space station construction and maintenance, which includes the development of a multi-armed servicer robot [Holcomb, et al. 1987].

The ATL/McGill research effort on which this paper is based is directed at

- ♢ Extending the RCCL "language" with an aim to creating a comprehensive, cohesive programming environment for coordinated control of multiple manipulators. This is discussed in section 3.

- ♢ Extending the RCI system primitives to allow the creation of multiple control tasks in a multi-processor environment. This is necessary to provide a means for implementing the multi-robot RCCL, and is described in section 4.

- ♢ Using the tools described above to develop algorithms for force control and cooperative manipulator action based on an outer force control loop wrapped around an inner position control loop, as per [Maples and Becker 1986]. This work is described in [Lee 1987].

The last section of this paper describes some of the details involved in implementing this

robot control environment on a multi-processor MicroVAX II system.

2. Background: the RCCL/RCI environment

This section summarizes a few of the key features of RCCL as a background for the discussion of our extensions.

RCCL is a library of routines for describing and controlling robot positions and actions, combined with a *trajectory generator* for realizing these actions [Hayward and Lloyd 1985, Lloyd 1985]. It is largely an implementation of the ideas presented in Richard Paul's book [Paul 1981]. An RCCL application program is written in the "C" language and uses special primitives to specify robot action requests and queue them for servicing by the trajectory generator. The trajectory generator is a high priority real-time background task, which does path interpolation, maps from Cartesian to joint space, smooths adjacent path segments, and outputs a set of joint setpoints at a (typical) rate of around 50 Hz. These setpoints are fed to joint level control modules in a particular robot (which execute a simple position interpolation and control loop at a much higher rate, typically around 1 KHz.). The control task structure is hence a 3-level affair, consisting of an asynchronous *planning* task (the main RCCL program), and synchronous *trajectory-control* and *joint-control* tasks (Figure 1).

RCCL itself implements only the top two levels, relying on the joint control modules provided by the robot manufacturer for the third level. For instance, in most present installations, the planning and trajectory modules execute on a VAX/UNIX system, which is interfaced to the joint-level robot controller through some sort of high speed parallel link. To ensure that the trajectory generator can be run in real time, it is necessary to make some adjustments to the UNIX operating system. These modifications constitute the Robot Control Interface (RCI), on top of which RCCL is written.

The planning task accounts for most RCCL application code; the trajectory generator is provided as a library task, although one of the more useful features of RCCL allows users to pass functions to the trajectory task for real-time execution. The planning level spawns the trajectory task with the procedure `rccl_open()`. Robot motion requests can then be generated using two basic kinds of primitives:

- ♢ *Primitives used in manipulating coordinate frame relationships.*
- ♢ *Primitives which invoke and control robot motions.*

The first provide for the creation, deletion, and modification of coordinate frame transformations (represented in RCCL by the data structure type `TRSF`), and the definition of transform

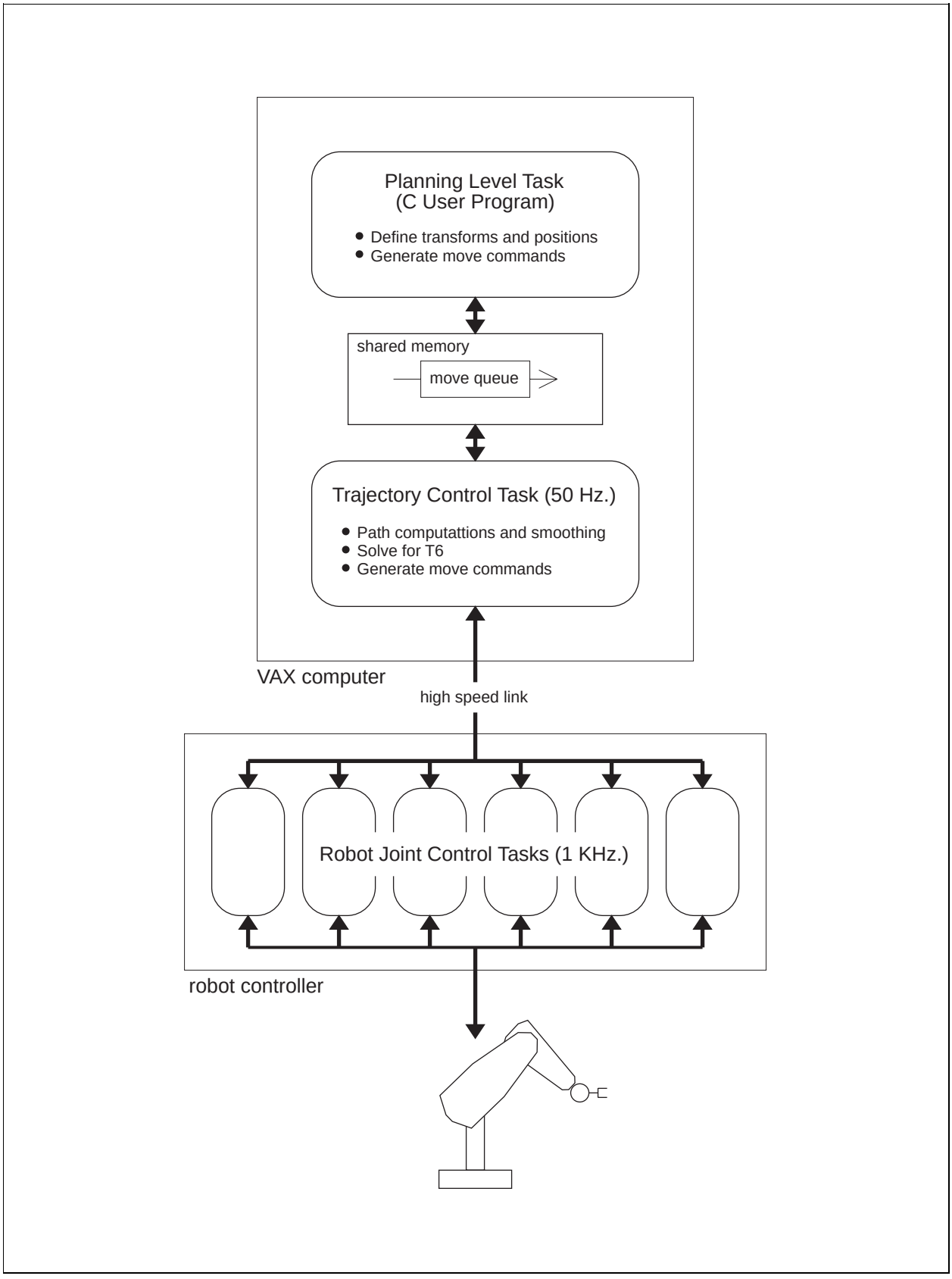

Figure 1: A typical RCCL system.

equations (type *POS*) between these transformations. Positions are defined by transform equations which relate the value of the manipulator T6 transform to the product of a set of other transforms. For example, if we have a transform WORLD from robot link 0 to some world coordinate system, a transform P locating a part with respect to the world, and a robot tool transform TOOL, then locating the tool tip at the part involves satisfying the equation

$$\text{T6 TOOL} = \text{WORLD P}$$

The RCCL primitive for specifying positions is *makePosition()*, whose argument list is a set of transforms describing a transform equation. The position equation above would be created with a piece of code that looks like:

```
TRSF *coord, *tool, *p          /* transform pointers */
POS *pos;                       /* position pointer */

...initialize transforms...

pos = makePosition( t6, tool, EQ, coord, p, TL, tool);
```

The keyword *EQ* denotes the two halves of the equation, while the keyword *TL* and the transform following it provide additional information as to where the manipulator tool frame exists. The ability to create general coordinate transform relationships is a powerful tool. Within the same equation, one transform might represent the position of a sensor, another might represent path corrections for compliant motion, and another might generate a motion path. This can be exploited within RCCL by changing the actual values of the transforms prior to or during motions. In some sense, the set of transforms and their relationships constitutes a very simple kinematic "world model" of the robot environment.

The primitive that initiates robot motion is *move()*, which takes a position data structure as an argument and queues a request to the trajectory generator to move the robot so that the position equation is satisfied. Successive *move()* requests will travel through the target point; to stop there, the command *stop()* is used. Both primitives returns immediately; if we wish to wait for the requested motion to complete, we must use one of the *wait_for* primitives. The motion can be controlled by setting various parameters such as the path generation mode (joint or Cartesian), speed, forces to exert, force limit conditions, etc. In addition, individual transforms in the target position equation can be varied by binding them to real-time functions. Since the robot trajectory will automatically track these variations, we can use this feature to create very general, dynamically changing, paths.

An RCCL application programmer typically does not work with the trajectory generation code. Often, however, developers may wish to modify the trajectory generator to incorporate new features, or replace the trajectory generator completely. Workers experimenting with new

joint-level control laws often find it convenient to prototype them directly inside the trajectory module and bypass level 3 entirely [Cohen and Daneshmend 1987, Aboussouan 1985] (although this may be constrained by computing time and the ability to achieve the desired sample rates). At this point, the programmer is no longer working with RCCL, but with RCI. Expanding the original capabilities of RCI represents a significant part of the McGill/RCA effort, and is discussed in section 4 of this paper.

RCCL was originally written at Purdue University by Vincent Hayward, in collaboration with Richard Paul, during 1982-83, using a VAX-11/780, a PUMA 560, and a Stanford manipulator. RCCL was transported to McGill University during 1984-85, where modifications were made to improve its usability [Lloyd 85]. The system was later retargeted to MicroVAX IIs at RCA, the Jet Propulsion Laboratory (California), and McGill [Lee, et al. 1986]. A MicroVAX II/VMS implementation was achieved at the NASA Robotics Laboratory, Goddard Space Flight Center, Maryland, during the spring of 1987. All of these sites still primarily use PUMA robots.

3. Enhancing RCCL for Multiple Robots

3.1 Multiple Robot Control Issues

Before arriving at specific extensions to the language we first considered how multiple robots might be used. This led us to establish three different modes of operation for multi-robot control:

1. *Independent Operation.* This is when several robots are performing independent tasks at the same time, which can be done by running several single-robot RCCL programs concurrently.

2. *Synchronized Operation.* This involves applications where two or more arms are required to execute a single task, such as in assembly operations: different arms might use different tools, one arm might pass objects to another, etc. A system supporting this must have the ability to synchronize individual arm motions, as in specifying that arm A must wait for arm B to move into position before continuing with its task. Synchronization primitives have recently been added to RCCL for doing this [Lloyd 85, chapter 5], but have not yet been used with multiple arms.

3. *Coordinated Operation.* This concerns instances where several arms are manipulating a common object. A system supporting this must provide 1) a convenient method of specifying relative positions amongst the arms that correspond to grasping positions of each arm on a common object, and 2) a method of specifying a coordinated motion that

maintains the relative positions while moving the object along a desired path. The new RCCL primitives described below are largely directed at this issue.

Coordinated motion implies that kinematic constraints exist between the different arms, which the trajectory computed for each arm must satisfy. Forces of interaction caused by residual position errors can then be reduced using a method such as described in [Lee 1987]. We ensure that the kinematic constraints are satisfied by computing a single path of motion for the object being manipulated, which the trajectories for each robot simply follow at some constant offset. If each arm were instead moved using independent RCCL trajectory generators, the relative displacements of the arms would vary between the motion end points.

This need to generate a single path for an object is realized by introducing into RCCL the concept of an *object frame*, which is an abstract coordinate frame which can be placed arbitrarily in the workspace of the robots. Generally, the location of the object frame is rigidly attached to some object which is to be manipulated. It is defined using a position equation like those which describe robot goal positions, and is created with similar primitives. An object frame can be "moved", in Cartesian space, using the RCCL motion primitives: a trajectory generator will then move the object frame through space. By creating robot positions which incorporate this object frame, and maintaining the robot at these positions, we can move a real object attached to this object frame. Moreover, by including functionally defined transforms in the object frame equation, we can incorporate sensor-based tracking into its motion.

3.2 New Function Primitives

One goal in specifying our RCCL extensions is to retain the character and general syntax of RCCL so that old RCCL programs will run with only minor editing, and experienced users will find the transition to multiple arms an easy one. Happily, the original RCCL primitives for controlling a single arm generalize to multiple arms quite naturally.

A multi-robot RCCL must allow us to be able to move different robots. We hence give *move()* an additional argument: a pointer to a new data structure (type *MANIP*) which denotes a particular robot and collects together in one place all the data associated with that robot such as its T6 transform (*trans*), the joint angles (*j6*), the flag *completed* which is true when all motions queued for that robot have been serviced, and other such variables.

A *MANIP* data structure is associated with a particular manipulator by *rccl_open()*, which now takes a name of a robot and sets up a trajectory generator for that robot:

```
MANIP *robot;

robot = rccl_open( "PUMA560" );
```

These simple additions, plus the recently added primitives for motion synchronization (the event flag mechanism described in Chapter 5 of [Lloyd 85]) actually gives us everything we need to do synchronized motions of multiple robots.

Coordinated robot motion and object frames require a bit more enhancement. First, we generalize *makePosition()* (section 2) slightly. Originally, this routine always required the manipulator transform *t6* in its argument list, although the purpose of *t6* is actually generic: to identify which of the transforms in the loop is unknown and must be solved for. So that position equations are not necessarily associated with robots, we defined a keyword *TRANS* that can be used in place of *t6*. The example in section 2 can hence be rewritten as

```
p0 = makePosition( TRANS, tool, EQ, coord, p, TL, tool);
```

Having generalized the position equation, we need to be able to create object frames. This is done using a special instance off the *MANIP* data structure:

```
MANIP *obj;

obj = rccl_open ("OBJECT_FRAME");
```

The object frame is essentially a virtual robot, with its own trajectory generator, and we can use it with most of the canonical motion control primitives, or examine fields such as *obj->trans* or *obj->completed*. The only differences are that the virtual manipulator has less capability than the physical manipulator: it does not make any sense to move a generic object in "joint" mode, nor does it make any sense to look at its joint angles.

The field *obj->trans* contains the instantaneous value of the object transform, and is the frame that moves in space as we request motions of the object. Using functionally defined transforms, we can force the value of *obj->trans* to be determined by sensor inputs or any other means that code can be written for.

An object frame does not have to be connected to any robot, but it can be, by incorporating its transform *obj->trans* into a position equation used for moving a robot (see the example below). Given that an object frame is not always physically attached to anything, we may wish to sometimes simply "put" an object frame at a (possibly time varying) position, instead of "moving" it there. For this, we define a primitive *maintain()*, which is a little bit like *move()*, except that 1) it immediately puts the object at the at goal position, and 2) keeps following the goal position until another action is requested on that object. The primitive can be used with robots as well as object frames, as long as we are careful to previously *move()* to the position that we plan to *maintain()*. Usage of *maintain()* is illustrated in the example below.

3.3 Task Implementation

Figure 2 illustrates, at the task level, how the new RCCL is

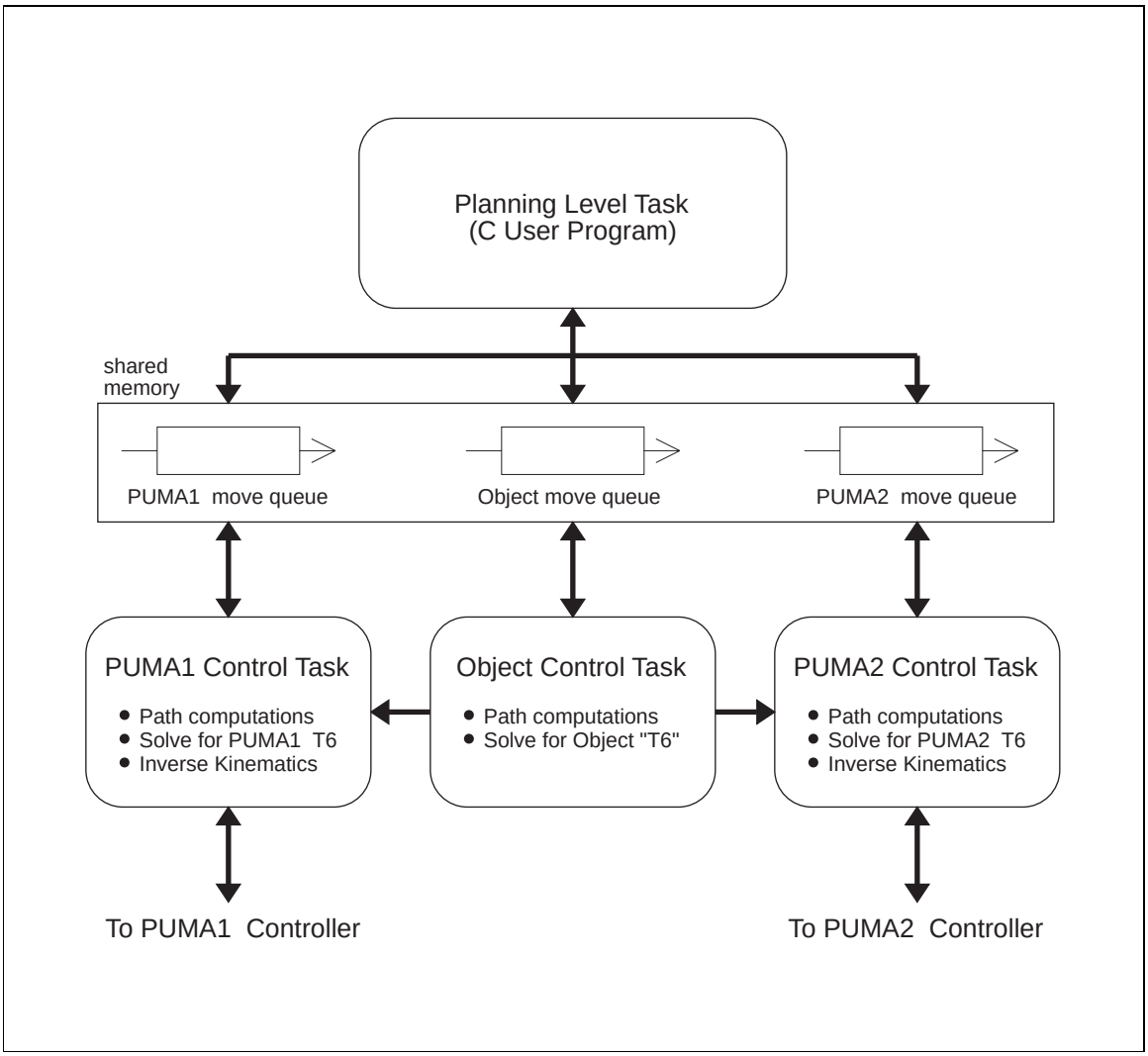

Figure 2: Task diagram of a multi-robot RCCL system.

implemented for a system controlling two arms and one object frame. Each manipulator, and the object frame, has a trajectory generator (implemented using an RCI control task) which runs at a typical rate of around 50 Hz. As usual, the planning level task is a normal C program which generates and queues motion requests for the trajectory tasks. Communication among the tasks is through shared memory. The trajectory tasks remove motion requests from their respective motion queues and then generate the intermediate setpoints that constitute a path to the specified goal position. This includes constantly reevaluating the goal position, if necessary, to take account of functionally defined transforms. Some synchronization is required here to

insure that the value of the object frame transform is evaluated first, as it may be used in the position equations of the physical manipulators. The trajectory tasks associated with the robots must perform the additional step of inverse kinematics to generate the setpoint joint angles which are transmitted to the robot controller.

This implementation uses the extended RCI primitives described in section 4.

3.3.1 Dual Arm Example

We illustrate the new features of RCCL with a brief example that arises in the context of satellite servicing. Suppose we wish to grasp a moving object with two manipulators and then follow it without exerting a force on it, an action which would be the first step in a two arm catch of a rotating satellite. Refer to the transform diagram in Figure 3.

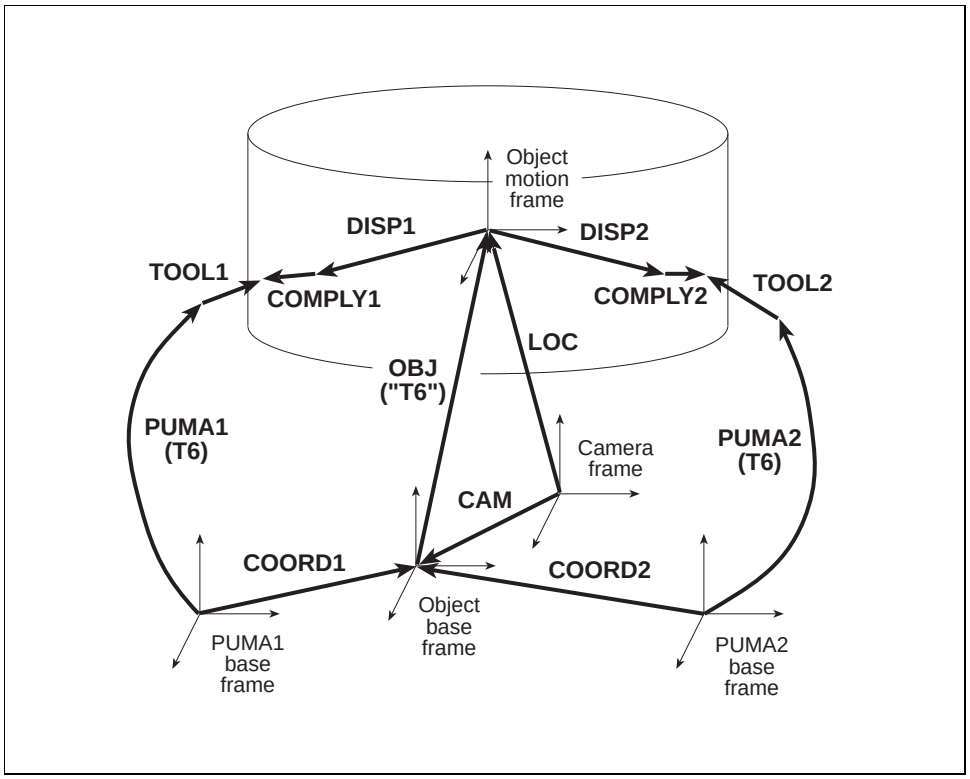

Figure 3: Transform diagram for multi-robot object manipulation task.

Assume that we can track the 3-D position and attitude of the object with a camera based sensor. We use the track data to drive the position of an object frame using a functionally defined transform, *Loc*, which is constantly updated by sensor processing to give the location of the object relative to the camera coordinate frame. We can set up a frame describing this object and have it maintained by the value of *Loc*:

```
TRSF *Cam, *Loc;                    /* transforms related to object */
POS *Obj_Pos;                       /* position equation for object */
MANIP *Obj;                         /* object frame data structure */

... allocate and initialize transforms and position ...

Obj = rccl_open ("OBJECT_FRAME"); /* Set up object frame, and turn */
rccl_control (Obj);                 /* on its trajectory generator. */

/* build the equation for the object frame */

Obj_Pos = makePosition (Cam, TRANS, EQ, Loc, TL, TRANS);

maintain (Obj, Obj_pos);            /* have object frame track 'Loc' */
```

Next, to move the physical manipulators to grasp points defined with respect to the object frame, we would issue move requests to positions defined with respect to *Obj->trans*:

```
TRSF *Tool1, *Coord1, *Disp1, *Comply1;  /* transforms for robot 1 */
TRSF *Tool2, *Coord2, *Disp2, *Comply2;  /* transforms for robot 2 */
POS *p1, *p2;                            /* robot position eqns. */
MANIP *robot1, *robot2;          /* data structures for robots */
extern int comply_on;            /* variable to turn on compliance */

... allocate and initialize transforms and positions ...

robot1 = rccl_open ("PUMA1"); /* Setup robot trajectory tasks */
rccl_control (robot1);           /*  and turn them on ... */

robot2 = rccl_open ("PUMA2");
rccl_control (robot2);

/* Now build the robot position equations */

p1 = makePosition (TRANS, Tool1, EQ,
        Coord1, &Obj->trans, Disp1, Comply1, TL, Tool1);
p2 = makePosition (TRANS, Tool2, EQ,
        Coord2, &Obj->trans, Disp2, Comply2, TL, Tool2);

trackingMode (robot1);           /* cause robot to follow */
move (robot1, p1);               /* Move to grasp point, */

trackingMode (robot2);
move (robot2, p2);

wait_for (robot1->completed); /* Wait for both motions to stop */
```

```
wait_for (robot2->completed); /* at the grasp position ... */

comply_on = 1;                /* tell comply functions to start */
GRASP (robot1);               /* then grasp the object at the */
GRASP (robot2);               /* same time. */
```

The primitive *trackingMode()* preceding the move requests is another small feature that we have introduced: it puts the trajectory generator into a mode where, after a move to a goal position is completed, it continues to track the position until another motion request appears on the move queue (instead of stopping firmly at the goal position's last value). The program uses the *completed* flag to determine when both robots reach their grasp positions (remember that the move requests do not block). That this will probably not happen at the same time is OK since the each arm will continue to track the grasp point. When the grippers are closed, a compliance algorithm is started, using a compliance function bound to the transforms *Comply1* and *Comply2*, which ensures the arms continue to follow the object while exerting zero force.

This program is naturally oversimplified, ignoring things like via points to control approach, transform allocation and definition, and details of the grasp process. The force control method for servoing the comply transforms is an issue dealt with in a separate paper (see [Lee, 1988]). However, the spirit of the library is illustrated.

4. RCI: Primitives for Creating Real-Time Control Tasks

In this section, we first summarize the original RCI system, and then describe how it has been enhanced to provide a multi-tasking capability. The implementation of this new system is discussed in section 5.

4.1 The single robot RCI system

RCI was originally designed as a C package that allows the creation of a real-time robot control task under UNIX. Programs written using RCI are very low level, and implement such things as path generation or control algorithms. The interface to the robot is at the joint encoder/current level. The format is very simple: The control task is a function in the user's program that is tied to a periodic interrupt from some external device (such as a robot controller interface). On every interrupt, the control function is executed asynchronously with respect to the rest of the program. The function is called by the device driver, which elevates processor priority to ensure that the control routine will not be interrupted, and temporarily restores the main program's

memory context. Because the control function and the main program use the same memory context, communication between the two is easily achieved using global variables.

For convenience, RCI automatically performs the communication with the robot controller, first reading back information values (such as joint positions, force sensor data, etc.), checking for limit conditions, and then posting them to a global "blackboard" variable, which has the somewhat historical name *how*. The user's control function is then called, which reads the robot state described in *how*, and generates low level position or current commands, which it posts to another blackboard variable called *chg*. When the control function returns, RCI examines *chg*, does some more checking, and parcels the commands off to the robot controller. To give the programmer the option of overlapping computation with the operation of the robot controller, a second user control function may then be called to precompute things for the next cycle. The whole scheme runs at some fixed control rate and is diagramed in Figure 4.

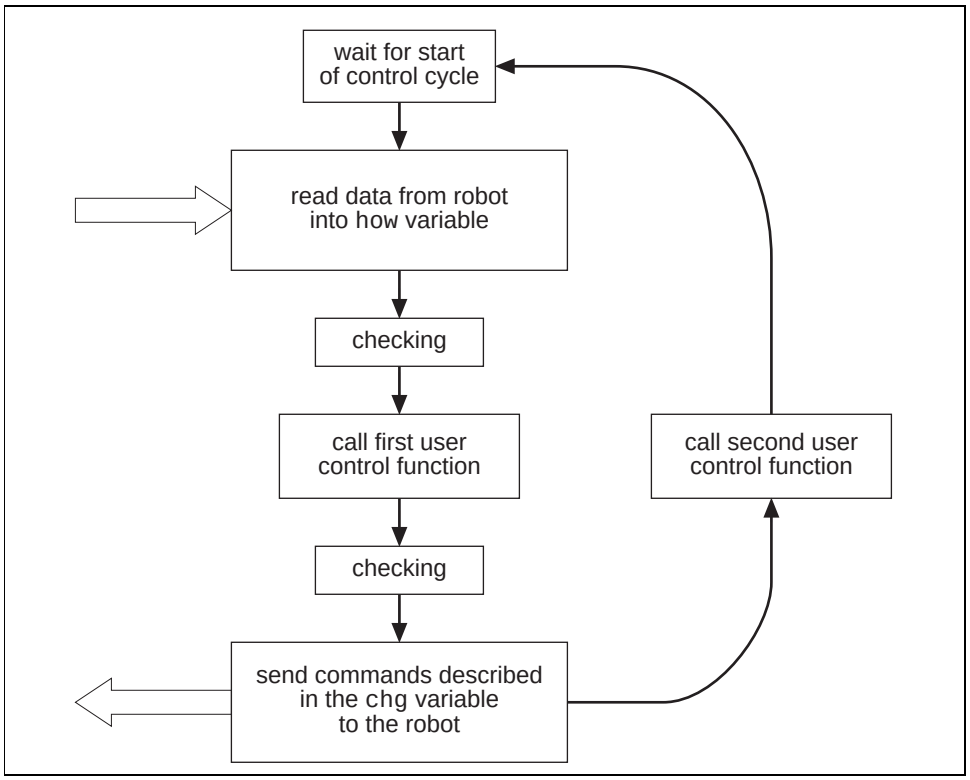

Figure 4: Execution cycle for an RCI robot control task.

When working with a Unimation PUMA robot, interfaced with a parallel port, it is possible to run simple control functions at dependable rates of up to 70 Hz. on a MicroVAX II/UNIX system.

4.2 Requirements for prototyping robot control tasks

RCI's capability of being able to attach a simple control task to programs executing in the rich environment of a large operating system (UNIX) has proven very useful in research and development. Driven by this success, we have proceeded to enhance the simple RCI described above to take advantage of multiple CPUs (on a common backplane) and allow the creation of several control tasks driving multiple robots.

The extended RCI system allows the creation of several control tasks, which may run at different rates or be triggered by different events. For generality, a task is not necessarily connected to a robot (as in the original system code); the robot interfacing features are optional. The mechanisms by which the RCI internally performs I/O are also available to the programmer for embedding special purpose I/O in the control functions themselves.

The design concept, described in the next section, was motivated by the following goals:

- ♢ *Ease of use* – particularly necessary since the system is meant to serve as a development and prototyping environment.
- ♢ *Extremely fast operation* – robot path generation and control requires the ability to execute tasks at rates ranging up to 100 Hz. or higher; consequently, speed is of absolute importance.
- ♢ *Synchronized scheduling* – some control tasks need to run at a fixed pace with respect to each other, so a common clock is needed that can drive tasks at synchronized rates.
- ♢ *Shared memory between tasks.* – required to ensure the fastest possible inter-task communication.
- ♢ *Message passing between tasks.* – desirable for instances where communication speed is not of too much concern and data isolation is desired.

4.3 Principle design concept of RCI

4.3.1 Task Structure

A single UNIX process (known as the *parent*, or planning task) may create several control tasks. The anticipated number of control tasks is 1 to 5; this is not a limit, but rather a number which has guided our thinking in designing the interface primitives. The parent task and the control tasks comprise an executing RCI program. Different RCI programs may run concurrently (if the required resources are available), but communication between these programs can be done only

through the normal UNIX interprocess communication facilities. While in principle it would be possible to establish communication and synchronization between control tasks with different parent processes, this was not seen to be necessary and creates resource allocation problems.

In a multi-processor configuration, one *host* processor runs UNIX, while others serve as auxiliaries which run only RCI tasks. A common backplane is assumed to facilitate shared memory. The parent task always runs on the UNIX host, while the control tasks may run on either the host or an auxiliary. As in the original RCI, each control task is simply a pair of control functions.

4.3.2 Task Images

The images for the parent task and all of the control tasks are the same, as in the original RCI system. For tasks executing on the same processor, this is implemented, as before, by using the same memory context. An auxiliary processor receives a copy of the original UNIX program image, or at least a copy of those segments of it which are accessed by the control tasks running on that processor. Although more than one control task belonging to the same program may be executed per auxiliary processor, no more than one RCI program may be associated with each auxiliary processor; this is an efficiency/implementation constraint discussed in section 5.3.1.

4.3.3 Task Creation and Deletion

Control tasks are created and deleted by primitives called from the planning task. Creating a task does not actually turn it on (*activate* it); that is done using `rci_start()`, described later.

```
td = rci_create (name, proc)
```

creates a control task with the given `name` on the processor `proc`; a default processor is selected if `proc` is unspecified. The task is connected to a robot if `name` equals the name of one of the robots supported by the system (section 4.3.7). `td` is a pointer to a *task descriptor* which is used to reference the task and contains task and robot specific information.

At the present time, all control tasks and shared memory areas (section 4.3.5) must be set up before any of them are activated. This restriction is imposed so that control tasks do not have to worry about accessing tasks or memory areas that do not exist. This is not really very restricting since most task attributes such as the control functions, the scheduling discipline, and whether it is active, may be changed dynamically.

Correspondingly, all tasks and memory areas are deleted together using the call

```
rci_close()
```

The actual control functions are specified with

```
rci_control (control1, control2, startup)
```

RCI will either call the control functions called back-to-back, or interleave them with the robot I/O if it is connected to a robot (Figure 4). The *startup* function is called once at task initialization time and can be used to perform any time consuming set up procedures.

4.3.4 Scheduling

Various planning level primitives determine how an RCI control task is scheduled.

To run a task off the common clock,

```
rci_onclock (td, interval, offset)
```

notifies the RCI scheduler to wakeup the task referenced by *td* once every *interval* clock ticks, with an offset, according to the following algorithm:

```
if ((clock_ticks - offset) % interval == 0)
 { wakeup the task;
 }
```

To run a task off some arbitrary event, instead of the clock, a wakeup function can be specified:

```
rci_onfunction (td, wakeup_function)
```

The wakeup function can examine I/O space or memory locations, and returns a non-zero value when the task should be woken up. When the task is executing on the host CPU, the wakeup function must be supplemented with an interrupt from a device specified by the primitive *rci_deviceInterrupt()* (See section 5 for an explanation). The driver for this device must have RCI support built into it.

Once a control task is invoked, it will not be interrupted by other control tasks until it completes. This permits simple round-robin scheduling for tasks on the same processor; tasks which must run concurrently have to be assigned different processors. A timeout mechanism detects rogue tasks which exceed their allocated time.

RCI tasks attached to robots are typically daisy-chained off the robot controller: a process on the controller wakes up, spends several milliseconds collecting sensor and feedback data, and then wakes up the control task and communicates with it. If we are controlling several robots in sync by running them off the RCI system clock, we really need to wakeup the *robot controller*, which will in turn wake up the control task. This can be done by requesting the scheduler to call a *trigger function* for a task on a certain clock interval/offset:

```
rci_trigger (td, interval, offset, trigger_function)
```

Setting the scheduling discipline for an RCI task does not actually activate it. Tasks are activated and released using the primitives

```
rci_start (bitmask)

rci_release (bitmask)
```

which take bit masks as arguments so that several tasks can be specified atomically (the bit corresponding to a particular task can be obtained from a field in its task descriptor).

4.3.5 Shared Memory and Message Passing

Shared memory between tasks is set up by planning level primitives before the tasks are activated:

```
memp = rci_sharedMemory (label, size, hostTd, accessMask)
```

This creates a shared memory region of *size* bytes with name *label* and returns a pointer to it. *hostTd* specifies the task whose processor will host the memory, and *accessMask* is a bit mask describing which tasks (besides the host) have access to the memory. A control task with access to this memory can get a pointer to it by calling *rci_getMemory (label)* when it first starts up.

Shared access to compile time static variables can be achieved with the call

```
rci_sharedData (accessMask, addr, size)
```

which causes the compile-time generated memory segment defined by *addr* and *size* to be shared among all tasks specified by the *accessMask*.

The message passing primitives are quite simple and are callable from all tasks. Since the scheduling of RCI tasks is handled by a separate mechanism, messages do not have to fulfill a synchronization/rendezvous function and so do not block.

```
n = rci_send (sendcode, buffer, size)

n = rci_receive (getcode, buffer, size, sender)
```

rci_send() takes the message of *size* bytes contained in *buffer* and queues it for delivery to the task indicated by the bit mask *sendcode*. The function blocks only until the queuing is complete and returns the number of bytes it was able to queue; if this number is less than *size* it indicates that the queuing area is out of space. *rci_receive()* probes the queue of deliverable messages and returns the first from any of the senders specified by the bit mask *getcode* into the supplied buffer. The task descriptor of the sending task is returned in *sender*; a value of 0 means no message was received.

An RCI control task can interrupt the parent task by sending it a UNIX signal with the call *rci_signal()*. Generalizing this to allow software interrupts to be sent to other tasks could be useful but was considered overly elaborate for the present implementation.

4.3.6 Support Functions

A few support functions are available for control tasks, which do not have access to the usual UNIX system calls. These include functions for local memory allocation (*rci_malloc()* and *rci_free()*), the function *rci_printf()* which prints diagnostic messages to the task processor console, *rci_descriptor()* which returns the task descriptor of a named task, and a few test-and-set primitives for shared memory interfacing.

4.3.7 Robot interfacing aspects

RCI maintains a database of robots. When a task is created which has the name of one of these robots, that task is connected to the robot. This means that RCI will 1) instantiate parameter blocks in the task descriptor with the joint level data for that robot and 2) automatically perform I/O with the robot and maintain the *how* and *chg* structures for use by the control functions. The fields in the robot parameter blocks, as well as the *how* and *chg* data structures, are defined to represent a general robot at the joint level. This includes features such as joint angles, velocities, and torques, the limits on these values, lower level representations such as joint encoder counts and DAC values, conversion routines between such representations, and calibration information. Fields are also available for generic sensor I/O. RCI does not, at the moment, maintain kinematic or dynamic robot parameters since this information tends to depend on the RCI application.

5. Implementation of RCI Using a set of MicroVAX II CPUs

5.1 The MicroVAX II multicomputing capability

The usual processor elements in a MicroVAX II system are (presently) either KA630 or KA620 CPU boards, each of these implements a VAX instruction set with floating point support and contains one megabyte of local memory and a Q-bus interface [DEC 1986]. The local memory of each processor can be expanded through a private interconnect. One CPU must be configured as an *arbiter* which controls the Q-bus; it is then possible to add up to three auxiliary CPUs. The CPUs communicate with each other through the Q-bus (Figure 5).

A processor can attach a region of Q-bus space to a section of its physical memory. Another processer can then access this memory by mapping a region of its virtual memory to the Q-bus region, instead of physical memory. Processors signal each other through an *interprocessor doorbell* mechanism: processor A can cause an internal interrupt in processor B by setting a bit in processor B's *interprocessor communication register* which appears in Q-bus space. Unfortunately,

the Q-bus architecture prevents auxiliary CPUs from being able to receive device interrupts; all device interrupts must be handled by the arbiter CPU.

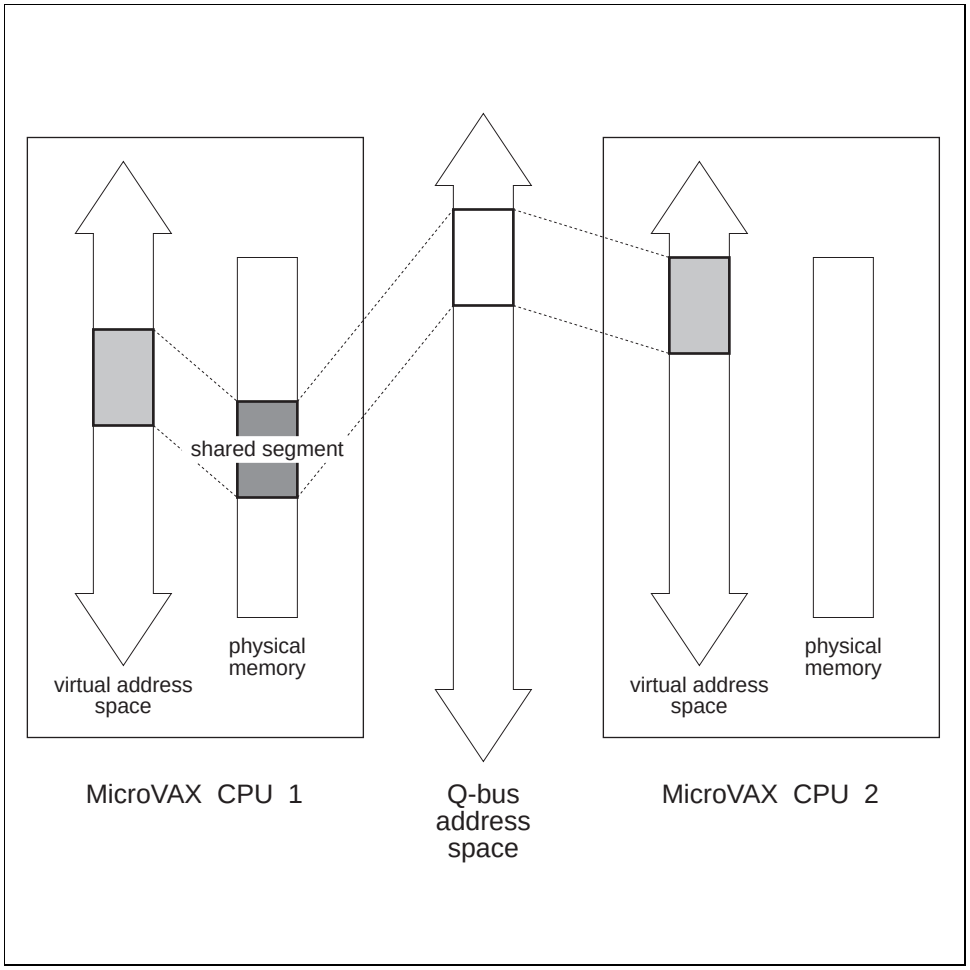

Figure 5: Shared memory between MicroVAX processors.

5.2 The UNIX interface and the auxiliary CPU minikernel

UNIX (which runs on the arbiter CPU) can be made to communicate with the auxiliary CPUs by means of a "KA device" driver, which keeps track of their status and is responsible for booting them. Booting is accomplished by having the arbiter CPU set up a "boot block" in Q-bus space which the auxiliary CPU boots from on request by the arbiter.

Normally, the KA driver is used to boot a special "minikernel" on the auxiliary which is responsible for downloading, running, and maintaining one user "process". These actions are facilitated by passing simple messages between the CPUs using the doorbell interrupt and some shared Q-bus memory. The messages pass through the KA driver and include directives for

downloading code and data segments to the auxiliary, allocating memory on the auxiliary, creating and mapping to shared regions of Q-bus memory, starting or stopping execution of code on the auxiliary, and sending signals back to the arbiter CPU.

5.3 The RCI Implementation

5.3.1 Process initialization

The RCI primitives either specify processors explicitly or allow the system to choose an auxiliary processor from among those available. The KA driver is used to check the requested processors for availability; a processor is not available if it has been allocated to another RCI program. This restriction is for both simplicity and efficiency: to be able to share an auxiliary CPU between different RCI programs would require giving the CPU a full multi-processing capability, which is more complex to implement and less efficient to run. It is possible to run multiple control tasks from the same RCI program on one CPU since they are just routines which can be executed sequentially and do not interrupt each other.

The required processors are then booted (if necessary), the real-time code and data segments of the RCI program are then downloaded into them, and a small RCI monitor is then started on each auxiliary CPU which helps finish the task set-up and which will later call the various control tasks assigned to that CPU, round-robin style, when they are active and ready to run. Shared memory is set up according to the description passed to the RCI system by the RCI user program. Space is allocated for each of the task descriptors and robot-specific parameter blocks associated with each control task. The task descriptors are made global to the whole system; each control task receives its own descriptor as a parameter, and can get the descriptor for another task with the *rci_descriptor()* call.

5.3.2 Scheduling

When the first call is made to *rci_start()* for a control task, the RCI monitor on the appropriate CPU is notified, and this then calls the task's *startup()* function. If the task is driven off the system clock, the scheduler is also notified (see below). The task is then active and remains so until a call is made to *rci_release()* or an abort condition is raised.

The RCI monitor on each processor polls to see when any of the control tasks on that processor are runnable. For tasks bound to a wakeup function (using *rci_onfunction()*), this is done by simply calling that function repeatedly. For tasks attached to the system clock, the monitor checks a run count in the task descriptor which is incremented by the scheduler. When a task is runnable, the monitor calls the two control functions; if the task is associated with a robot, it

also calls robot I/O and checking routines and maintains the *how* and *chg* blackboard variables. For scheduler driven tasks, the monitor decrements the run count in the task descriptor. Polling is then resumed.

For RCI tasks running on the arbiter CPU, there is no monitor per se since polling is not workable. Instead, the invocation of each task must be tied to an device interrupt (as specified by the *rci_deviceInterrupt()* call. Arbiter tasks invoked by the scheduler are tied to a specific (default) device interrupt associated with the scheduler.

The scheduler is a special asynchronous "task" bound to a real-time clock. One auxiliary processor per RCI program is elected to run the scheduler, which works in the following way: the scheduler maintains a private list of all control tasks, whether they are active and bound to the system clock, and if so, at what time they should next execute. Every time the scheduler routine is called, it scans the task list and wakes up each task whose indicated run time is less than or equal to the current time. The wakeup is done (as described above) by incrementing a count in the task's descriptor (and sending an interrupt if the task is executing on the arbiter). The next wakeup time for the task is then computed. Before the scheduler routine returns, it posts a request to the minikernel to reinvoke it at the next time that a task wakeup will be required. Because of the tight time constraints the scheduler must run under, we assume that it is run only on an auxiliary processor.

5.3.3 Maintaining the real-time clock

The scheduler paradigm described above assumes that the auxiliary minikernel can maintain a real-time clock, and can execute functions (specifically, the scheduler routine) at specific times described by that real-time clock. The easiest clock to use is the 100 Hz. interval timer built into each VAX CPU board. This may, however, be too slow for some applications. In that case, an external timer is required. A small difficulty is encountered here in that it is not possible to interrupt an auxiliary processor using the usual Q-bus interrupt mechanism. To overcome this, we have implemented a small clock device which sits on the Q-bus, and at some selectable interval obtains control of the bus and sets the doorbell interrupt bit on the auxiliary CPU. The clock device maintains a count which the auxiliary can then read to determine that the interrupt is in fact a clock interrupt.

6. Conclusion

We have described work which is being done to create a useful multi-robot programming and development environment by extending the toolkit already provided by the RCCL/RCI system.

Our principle motivation has been the proven usefulness of having a full C/UNIX environment for creating and testing robot control algorithms at many different levels. The present status of our work is as follows: the RCI extensions are nearly complete and will be running on the MicroVAX system by the end of October, 1987. Since the multi-rccl and force control algorithm work is proceeding in parallel with this, the full system should be in operation with preliminary results available by the end of 1987.

References

[Aboussouan 1985] Patrick Aboussouan, "Frequency Response Estimation of Manipulator Dynamic Parameters" (M. Eng. Thesis). Dept. of Electrical Engineering, McGill University, Montreal, Canada, December 1985.

[Cohen and Daneshmend 1987] Moshe Cohen and Laeeque K. Daneshmend, "Evaluation of an Acceleration Feedback Position Control Algorithm on a Commercial Manipulator". Submitted to the 1988 IEEE Conference on Robotics and Automation.

[DEC 1986] Digital Equipment Corporation, *KA630-AA CPU Module User's Guide*, DEC Educational Services Development and Publishing, Marlboro, Mass. DEC Order Number EK-KA630-UG-001.

[Hayward and Hayati 1987] Vincent Hayward and Samad Hayati, "Design Principles of a Cooperative Robot Controller". To appear in Space Station Automation III, *Proceedings of SPIE*, November 1987, Cambridge, Mass.

[Hayward and Lloyd 1985] Vincent Hayward and John Lloyd, "RCCL User's Guide". Technical Report, Dept. of Electrical Engineering, McGill University, Montreal, Canada, December 1985.

[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)

[Lee, et al. 1986] Jin S. Lee, Samad Hayati, Vincent Hayward, and John Lloyd, "Implementation of RCCL, a robot control C library on a MicroVAX II". *Advances in Intelligent Robotics Systems*, SPIE's Cambridge Symposium on Optical and Optoelectronic Engineering, Cambridge, Massachusetts, October 26-31, 1986 pp. 472 – 480.

[Lee 1987] Jin Lee, "Multi-Arm Force Control Using Position Accommodation", submitted to the 1988 IEEE International Conference on Robotics and Automation.

[Lloyd 1985] John Lloyd, *Implementation of a Robot Control Development Environment* (M. Eng. Thesis). Dept. of Electrical Engineering, McGill University, Montreal, Canada, December 1985.

[Holcomb, et al. 1987] L. B. Holcomb and M. D. Montemerlo, "NASA Automation and Robotics Technology Program". IEEE AES Magazine, April, 1987, pg. 19 – 26.

[Maples and Becker 1986] J. A. Maples and J. J. Becker, "Experiments in Force Control of Robotic Manipulators". *Proceedings of IEEE Conference on Robotics and Automation*, San Francisco, CA., April 7-10, 1986, pp. 695 – 702. (Vol. 2)

[Paul 1981] Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control.* MIT Press, Cambridge, Mass., 1981