Abstract

RCI (Real-time Control Interface) is a C/UNIX package which gives a programmer the ability to develop real-time control applications within a UNIX environment. It allows functions linked into a standard UNIX program to be executed as real-time “tasks”, either on the UNIX (host) CPU or on an auxiliary CPU connected to the host by shared memory. RCI has been implemented on MicroVAX-II and Sun-4 systems, and is used as a platform for the robot programming library RCCL. The paper outlines the system’s design and discusses aspects of its implementation.

1. Introduction

This paper describes RCI (Real-time Control Interface), a software package for creating simple real-time tasks in a UNIX environment. It was developed over the last few years at McGill University and the General Electric Advanced Technology Laboratory.

RCI was designed as a substrate for implementing RCCL (Robot Control C Library), a software package for doing robot control from UNIX [Lloyd, et al. 88, Hayward and Paul 86]. In RCCL, robot actions are requested from a UNIX process, and the trajectories to implement these actions are computed by a real-time control task, which communicates with the UNIX process using shared memory. The control task typically runs at 50 to 100 Hz., which is beyond the real-time capabilities of most current stock UNIX systems.

The RCI package makes it possible to create such tasks. Depending on the system, RCI tasks can be run either on the UNIX CPU itself (with some hooks introduced into UNIX to make this possible), or on auxiliary CPUs which are attached to the system backplane and utilize a real-time kernel.

Versions of RCI capable of running tasks on the UNIX CPU have been implemented on MicroVAX-II systems running 4.3 BSD, and Sun SPARC systems running SUNOS 3.2 and 4.0.3. A version capable of running tasks on auxiliary CPUs has been implemented for the MicroVAX-II system, using KA630 or KA620 CPU boards attached to the system Q-bus and running a special kernel written...
2. RCI Functionality and an Example

From a software point of view, an RCI task looks very much like a signal handler: a particular control function is designated to be executed either on a particular hardware interrupt, or off of a system clock tick. The control function runs to completion, and communicates with the parent UNIX process (known as the planning task) through shared memory. RCI tasks which run on the UNIX CPU are executed directly at interrupt level (see section 4), in order to achieve the necessary real-time response. RCI tasks running on an auxiliary CPU execute as tasks under the real-time kernel for that CPU. In either case, control functions do not have access to normal UNIX system services, although this is not a great loss since such services are typically time consuming. RCI tasks can directly access all or part of the external system bus through mapped memory (an idea also used in the /dev/bus project at the University of Pennsylvania [Holder 89]).

Space limitations prevent a detailed description of the RCI primitives [Lloyd 88]. Instead, we will give an example RCI program in which a task running at 100 Hz. computes a running average of data collected from a bus register. The averaged value is read back by the planning task.

The planning and control tasks can be compiled together as one program, although the control level portion needs to be placed in a separate module(s).

```c
/* FILE plan.c: */
#include <rci.h>               /* RCI definitions, etc. */
main()
{
    extern taskFxn();         /* function used by task */
    RCI_DESC *td;             /* task descriptor */
    float *avg;               /* pointer to shared mem */
    int t;

    td = rciCreate("task1", 0);
    avg = (float*)allocMem("avg", sizeof(float), 0);
    rciControlFxns(td, NULLF, taskFxn, NULLF, 0);
    rciScheduling(td, ON_CLOCK, NULL);
    rciRate(td, 10, 0);

    *avg = 0.0;
    rciStart("0");
    for (t=0; t<10; t++)
    {
        printf("average value is %g\n", *avg);
        sleep(1);
    }
    rciRelease("0");
}
```

RCCL/RCI Release 4.2, February 4, 1996
/* FILE control.c: */

#include <rci.h>

#define REG_ADDR 0xe000 /* bus address offset */

taskFxn(td)
{
    float *avg; /* pointer to shared mem */

    avg = (float*)getMemByName("avg");
    *avg = 0.9*(avg) + 0.1*(short*)(rciBus+REG_ADDR);
}

Compilation is done with the rcc command, a front end to cc that contains RCI extensions. To allow RCI to keep track of what the control level needs, all control level modules should be placed to the right of the special keyword CTRL on the rcc command line. The example here could be compiled with the command

crc plan.c CTRL control.c

which will produce a single executable named plan.

The program uses rciCreate() to set up a task on CPU 0. allocMem() is used to allocate a piece of shared memory to store the averaged data value. controlFxn() sets the control function to be taskFxn. To arrange for the task to be run at 100 Hz., its scheduling discipline is set to ON_CLOCK using rciScheduling(), and its execution interval is set to 10 msec. using rciRate().

After initialization, the RCI task is set running with a call to rciStart() (~0 is a bitmask specifying all control tasks). The planning task then reads out the value computed by the control task for a few seconds before shutting it down with rciRelease() and exiting.

The control task consists of the function taskFxn(). It uses getMemByName() to locate the shared memory. The quantity being averaged is read from location REG_ADDR on the system bus, which is available through mapped memory starting at the base address rciBus.

3. Scheduling, Communication, and Utilities

An RCI task may be run according to one of four principal scheduling disciplines, established using rciScheduling() and a few auxiliary routines:

ON_CLOCK – The task is run off the system clock at a periodic rate.

ON_FUNCTION – The task is run whenever a polling function, driven by a device interrupt, returns true.

ON_TRIGGER – A combination of ON_CLOCK and ON_FUNCTION: a trigger function, executed off of the system clock, wakes up some external entity, and this in turn wakes up the task with an interrupt.

RCCL/RCI Release 4.2, February 4, 1996
The task is run off of a UNIX signal, in user mode. This is used for simulation and debugging applications.

Inter-task communication is done primarily through shared memory. The functions

```c
mem = allocMem (name, size, pool)
freeMem (mem)
```

may be used to allocate or free a piece of shared memory that is accessible to any of the program's tasks. The allocation routines can be called from the planning level only. Since RCI tasks may run on different CPUs, shared memory addresses may differ from one CPU to another. Because of this, shared memory addresses should be looked up using a reference function, such as `getMemByName(name)`, which uses the memory object's name as a reference, or `getMemByAddr(addr)`, which uses its planning level address. To avoid name space collisions, it is common to use `getMemByAddr()` and not name the memory objects at all.

Shared memory objects, or portions of them, may be atomically read or written using the primitives

```c
readMem (mem, buf)
writeMem (mem, buf)
copyMem (mem, src, dst, size)
```

While one task is referencing an object through one of these primitives, all other references through these primitives to the same object are blocked.

The `allocMem()` facility is optimized for handling small chunks of memory (typically 1 Kbyte or less). To allocate larger amounts of memory, it may be better to use the primitives

```c
rciGlobalMemory (name, size, pool)
rciSharedMemory (name, size, pool)
```

The memory allocated by these routines cannot be recycled. However, the memory allocated by `rciGlobalMemory()` can be accessed by other UNIX processes or RCI programs, which is useful for data logging or monitoring applications.

A simple message passing mechanism has also been built into RCI, but has not been used very much.

A primitive which has proven to be extremely useful is `rciSignal(num)`, which sends a UNIX signal to the planning task.

For communicating with the outside world, portions of the system bus are mapped into memory, with base locations at `rciBus` and `rciIOpage`. What part of the bus is mapped depends on the system: on MicroVAX-II systems, `rciBus` is the entire Q-bus and `rciIOpage` is the Q-bus I/O space. On Sun systems, `rciBus` is a portion of VME24d16 space, and `rciIOpage` is the VME16d16 space. Direct bus access removes the need to go through the system in order to interface with peripheral devices.

UNIX system calls, as well as most of the RCI allocation and task control primitives, are not available to the control level. However, the control level is provided with a simple set of utility functions, such as:

```c
rciPrintf(fmt, ... ) -- diagnostic print routine
ciMalloc(size) -- basic memory allocation
ciFree(mem)
```
4. Implementation of RCI on the UNIX CPU

A good way to implement RCI tasks on the UNIX CPU would be to invoke the control function from interrupt level using UNIX signals. Unfortunately, this is too slow ([Holder 89]). Instead, the RCI task is executed directly at interrupt level, in kernel mode. This requires attending to the following issues:

1. **Context switching.** The memory context must be set to that of the RCI process (since the system may be inside any context when the interrupt arrives). The RCI interrupt code masks out all other system interrupts, switches to the RCI process context, executes the control function, restores the original context, and then unmasks interrupts. This switching can be done quickly since we are only concerned with the address space context. On the MicroVAX-II system, the page table base and length registers ($p_{0br}$, $p_{0lr}$, $p_{1br}$, and $p_{1lr}$) are switched. On Sun/SPARC systems, the memory management unit (MMU) context number is switched (it is also necessary to lock the context number to the process so that it won’t be given to another process).

2. **Memory locking.** All memory referenced by an RCI task must be physically resident. Those portions of the program’s text and data used by the control level must be locked down. To do this, a pair of locking primitives, `memlock()` and `memunlock()`, were added to the MicroVAX-II and Sun kernels.

3. **Exception Handling.** When an exception (such as a divide by zero or a memory access fault) occurs in kernel mode, the normal response of the system is to crash, since it indicates a kernel bug. Since this is unacceptable when running RCI applications, a hook was put in the system trap code to catch exceptions occurring within an RCI task and terminate the program more gracefully.

Shared memory between tasks running on the arbiter CPU is trivial to implement because the tasks and the UNIX process share the same address space. Creating shared memory is simply a matter of allocating it and locking it down. Shared memory between an RCI program and other UNIX processes is done by allocating memory out of a pool of kernel memory whose pages have been made user-accessible. The bus interface ($rciBus$ and $rciIOpage$) is created by mapping the appropriate bus region into system virtual space and making it user accessible.

5. Implementing RCI on auxiliary CPUs

Implementing an RCI capability on auxiliary CPUs is much “cleaner” than piggy-backing RCI onto UNIX, since the auxiliary CPUs can be booted with a real-time kernel that provides the necessary performance characteristics. Communication between the auxiliary CPUs and the arbiter is done using shared memory over the system bus.
To simplify the implementation, each auxiliary CPU may be attached to only one RCI process at a time. When a call to \texttt{rciCreate()} first references a particular CPU, that CPU is checked for availability and booted with the real-time kernel if necessary. The CPU is then loaded with those portions of the RCI process text and data segments used by the control level\(^1\). A monitor is then started which accepts instructions from the UNIX CPU to set up RCI tasks and run them when the appropriate scheduling conditions arise.

Communication between auxiliary CPUs and the arbiter CPU is done through shared memory. Each CPU exports a region of memory onto the external bus for access by all the other CPUs. The other CPUs import this memory into a specific region of their virtual address space. It would also be possible to use separate memory modules, although this has not yet been done.

6. Discussion

RCI provides a connect-to-interrupt facility for UNIX, allowing processes to create high-priority tasks for doing data acquisition or process control. Its principal use has been to provide the trajectory generation task for RCCL. The system concept is very simple, but this has allowed it to be implemented on a variety of hardware systems.

The main reason the system has been useful is probably because of the well-defined scheduling structure of the tasks it runs: RCI control tasks are assumed (more or less) to execute periodically at a known sample rate. Such tasks are easy to synchronize with, and they are also conducive to shared memory communication (a certain frequency of data access can be assumed, double buffering is feasible, etc.). This is also the reason that the RCI message passing primitives have not been used much.

References


\(^{1}\)This assumes that the auxiliary CPU can execute from the same image as the RCI process itself, which is true for implementations to date. In cases where this was not true, the \texttt{rcc} command could produce a separate image for the auxiliary CPU.