MAINTAINING CONTROL OF A ROBOT'S LIMBS USING THE BAKERY ALGORITHM

Patrick McDowell, Theresa Beaubouef
Department of Computer Science and Industrial Technology
Southeastern Louisiana University
Hammond, LA 70402 USA
(985) 549-2189
{pmcdowell, tbeaubouef}@selu.edu

ABSTRACT

This paper discusses a technique for maintaining control of robotic equipment given a single control line for both actuator control and limb position sensing. In particular, the bakery algorithm is used for preventing the problem of command/response packet collisions. These techniques can benefit any robot builder using a servo controller/IO board linked to a command and control computer via single control channel where there is a chance of collisions.

INTRODUCTION

At our university we are currently pursuing a robotics research program focused on providing robots with the ability to adapt and learn as necessary in order to keep their goals in target. That being said, our program is in its early stages, and one of the immediate tasks is to create the infrastructure necessary in order to do the research. This step is a necessary one; valid research cannot be done without working hardware platforms and the low level software to control the actuators and collect the sensor data. To that end we have embarked on a handful projects, one being an automated tennis court cleaning system [1], another being a robotic air hockey opponent [2], and the project that is the subject of this paper: the large actuator test bed.

While there are many novel examples of robotic technology, we are pursuing a path that we hope will result in the development of a series of cost effective "blue collar" robots. That is, for an end result, we want our robots to be able to perform some useful work. Much as MIT's Big Dog is designed to carry heavy (upwards of 300 lbs) [3] loads across uneven terrain, our long-term goal is create systems that can assist humans in doing work, much as horses and mules have in the past, and tractors do now.

In order to do so, we must be able to control actuators whose force output is in the range of 20 to 1000 pounds of linear thrust, instead of that of typical hobby servos whose peak output is generally from 40 to 350 ounce inches of torque. At first glance it may not seem like that much of a difference, but some quick calculations show that at the low end of the spectrum there is about a 96 to 1 increase in strength, while the high end comes close to a 550 to 1 increase in strength. Given these large differences, in order to lessen equipment breakage, and avoid getting ourselves in a pinch (literally), we are taking a methodical approach to control by building a test bed to facilitate equipment and low-level algorithm development.

This paper details these activities and specifically focuses on our technique of maintaining control of the equipment given a single control line for both actuator control and limb position sensing. While we use a PC linked to an SD 84 servo controller board

[4], these techniques can benefit any robot builder who is using a servo controller/IO board linked to a command and control computer via single control channel where there is a chance of command/response packet collisions.

This paper is organized as follows: the background section provides information on typical servo operation versus large actuator control, the approach section describes the bakery algorithm and how its use solved control and sensing conflicts inherent in our application, and finally the conclusions and future work describes issues with the project and the directions in which we plan to head.

BACKGROUND

Many of the robots available in kit form from vendors or those built by researchers and hobbyists use radio control (RC) servos to provide the muscle for moving the joints and ultimately allowing the robot to accomplish a myriad of motions, such as walking, dancing, punching, and arm waving. RC servos were originally used to move the control surfaces and motor throttles of RC planes. The servo itself is a small box (see Figure 1), usually about the size of a small cell phone. On one side of it is an arm, called the horn, which typically can rotate between 0 to about 225 degrees. By attaching a rod from this horn to a control surface, such as a rudder on an RC plane, the servo can move the rudder to the position requested by the plane's pilot.

Figure 1. Typical RC servo. The circular white plate on the top of the box is the horn. The horn has a range of rotational motion from 0 to about 225 degrees. Power, control and ground signals enter the servo through the multi-colored cord as seen in the figure.

The servo's position is proportional to the duty cycle of the signal generated by the pilot's remote control. One of the beauties of the RC servo is that it has an internal system that detects the horn's current location and works through a feedback loop to move the horn to the position indicated by the input control signal. This system works independently of the remote, with the implication being that the remote can send a position signal and let the servo handle the details of moving to the position requested. So essentially the communication from the pilot's controller to the plane is one way.

This method is great, until more force than the strongest RC servo can supply is needed, or one would like to monitor the servo's progress. In the latter case, there are some specialty servos that can provide feedback, but they are expensive, and they have no more torque than the run-of-the-mill servos of the same size.

Consider a situation in which a walking robot gets one of its legs in a bind; how can this be detected? The position information of the actuator is not readily available. A separate sensor at the joint can be used to monitor its progress, but in order to get this information the sensor must be polled. If the sensor is attached to an analog to digital (A2D) port on the servo controller board, the channel from the computer to the controller board will have to be a bi-directional connection, opening the possibility for a collision.

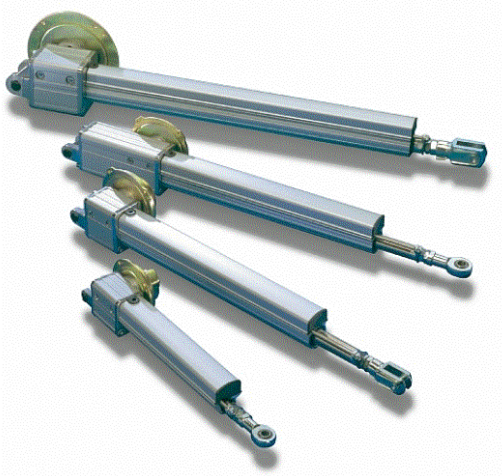

Figure 2. An assortment of linear actuators. Linear actuators work by turning a threaded rod with an electric motor. The threaded rod in turn extends the actuator arm from the actuator's case. In the figure, the arm is the narrow end of the actuator.

Our end goal is to be able control large actuations such as linear actuators or hydraulic cylinders. Unlike RC servos, these systems do not have internal feedback controllers, so they require that the computer monitor their progress by polling a sensor so the actuator can be shut down when it approaches its target position. Given these requirements, and the PC-to-SD84 control system, a reasonable way to manage the communication line between the PC and the controller board is needed. The requirements are as follows:

- *Speed*. The sensors need to be polled on a regular interval, at about 50 times per second—not blazing by any stretch, but still reasonable for getting information from the computer to the controller and reading the response on a serial line.
- *CPU usage*. The system cannot be CPU intense. Controlling the low level actuators is only one of many requirements being asked of the computer.
- *Fairness*. There cannot be a situation in which a sensor response back to the computer, or an actuator stop command, has to wait an indefinite amount of time because the control line is busy.
- *No collisions*. This point cannot be over stressed. If the control board misses an actuator shutdown command because it collides with a sensor position response, the result with even 20-lb strength actuators can be catastrophic. It was these types of events that motivated this work. Out of control linear actuators can flip chairs, move tables, bend metal work carts, and kick researchers with much more starch than ever possible with RC servos.

APPROACH

The basic system consists of a laptop PC linked via a USB cable to an SD84 84 Channel Servo Driver Module made by Devantech. Among this board's features are eighty-four configurable channels, with up to thirty-six being 10-bit resolution A2Ds, or any combination of pulsed width modulation (PWM) and digital I/O. Note that the reference voltage for the A2Ds is 5-volt TTL level, but the channel's voltage signal going to the sensor is at battery level. This means that initially, if the battery has 7.2 volts (the normal amount for a servo battery) the A2D will read maximum value (1024) at every sensor reading at 5 volts or above. The implication is that roughly one third of the sensor range is at maximum value. The solution is to use a regulated 5-volt supply for the A2D channels.

Basic communication with the board is a straightforward programming affair in which a serial communications port is opened and data packets are sent to and fro using the board's command language. For this effort a console program was written in Visual C++ 6.0.

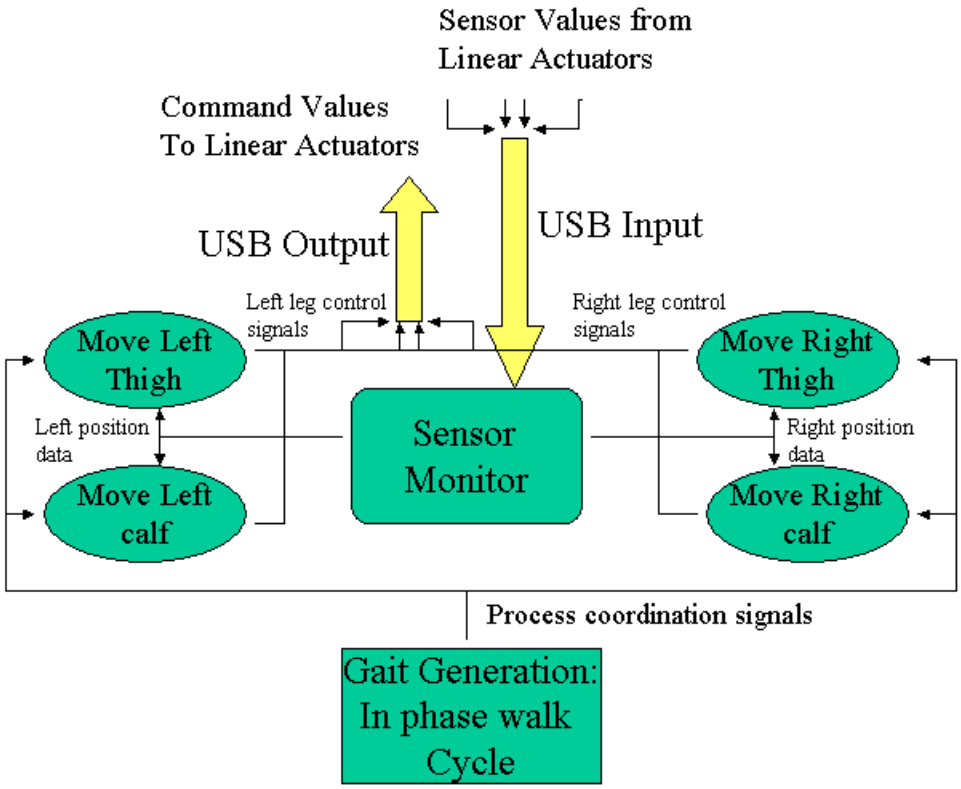

Figure 3. Block-level view of the control software. The gait generator resides in the main program, while the sensor monitor and all leg movement routines run as separate threads.

The control software is set up as follows. The main program starts a sensor monitoring thread whose job is to read the potentiometers in the linear actuators so that the degree of extension is always known to the leg part (calf, thigh, as seen in Figure 3) controller threads. The sensor monitor thread operates in a continuous loop, alternating between reading the sensors and sleeping. The sleeping is necessary in order to let the other processes have CPU slices.

A linear actuator provides the muscle for each leg part's movement. To control a linear actuator, a PWM port on the SD84 board is connected to a Victor motor controller that in turn provides voltage to the actuator's motor. The Victor controller is basically an amplifier whose input is the same as that of an RC servo (it is a PWM input) and whose output is a voltage between –12 and +12 volts. The leg part controllers (left thigh, right calf etc.) manage movements directed from the gait generation module in the main program. In order to move a part to a particular position, the leg controller sends a signal to the Victor amplifier, which in turn provides the linear actuator with the voltage to get it moving. Then the leg part controller enters a monitor/sleep loop and stops the leg when it reaches its ending position.

All actuator control and sensor reading commands go through the USB line to the SD84 board. Since the parts on a leg are moving in phase with one another, and directly out of phase with their counterparts on other legs, if left unchecked, conflicts between sensor values coming from the control board and actuator commands going from the computer to the control board will occur. In order to assure that no conflicts occur, Lamport's bakery algorithm [5] was used to manage the communications.

The bakery algorithm is usually presented as a tool for managing critical sections in an operating system setting, such as a piece of memory that producer and consumer processes share. The algorithm is extensible to N processes and fulfills the critical section requirements of mutual exclusion (only one process can use the resource at a time), progress (processes that wish to enter a critical section can participate in the decision about which process enters next), and bounded waiting (the wait time is not indefinite and a process can only be deferred a set number of times). The pseudo-code, along with clarifying comments, for a two-process bakery algorithm is shown below.

```
Process P(i)                                  /* The variable i = (0, 1) */
Entry: flag[i] = TRUE;                        /* This process wants to enter the critical section. */
Turn = 1-i;                                   /* This process defers to the other process. */
While(flag[i-1] && (turn = 1 – i)) {          /* Wait while the other process is in the critical
        Do nothing;                              critical section (flag[i – 1] is TRUE) and the
}                                                other process has not deferred to this process. */

/* Here is the body of                        /* In our case, we use the USB serial line to
   code to be executed in                     get a set of sensor values, or send a command. */
   the critical section.
*/                                            /* When complete, we indicate to the other
Exit: flag[i] = FALSE;                        process that we no longer need the critical section, thus
                                              allowing it to enter. */
```

The purpose of the flag variable is to indicate that a process wants to enter a critical section, and that once it is in the critical section the other process cannot enter until it is reset. The Turn variable breaks any ties, that is, if both processes hit the entry point at the same time (they set their flags to TRUE), then (assuming a one CPU situation) the last process to set the Turn variable will wait for the other process (process 0 sets Turn to 1, process 1 sets Turn to 0, thus allowing process 0 to enter the critical section).

Figure 4 shows our implementation of a 3-process bakery algorithm. The algorithm protects the 3 critical sections (for process numbers i = 0, 1, and 2) in which data is moved between the PC and the servo controller/A2D board.

3 process Bakery Algorithm

```
/* SEND ACTUATOR COMMANDS */
i = 0; /* Set process number. */

/* Entry into critical section. */
turn0 = 1; /* Defer to the other processes.*/
turn1 = 2;
flag[i] = TRUE;  /* I want to go. */

/* While any other process is in the critical
      section,  and I am deferring to them, I
      wait. */
while((flag[1] || flag[2]) &&
     ((turn0 == 1)&&(turn1 == 2))) {
     Sleep(20);
}
/* START CRITICAL SECTION. */
/* SEND ACTUATOR COMMANDS */
/* END CRITICAL SECTION */

/* Exit Critical section. */
flag[i] = FALSE;
```

```
/* GET A2D SENSOR VALUES */
i = 1;  /* Set process number. */

/* Entry into critical section. */
turn0 = 0; /* Defer to the other processes. */
turn1 = 2;
flag[i] = TRUE;/* I want to go. */

/* While any other process is in the critical
      section, and I am deferring to them, I
      wait. */
while((flag[0] || flag[2]) &&
     ((turn0 == 0)&&(turn1 == 2))) {
     Sleep(20);
}

/* START CRITICAL SECTION. */
/* GET A2D SENSOR VALUES */
/* END CRITICAL SECTION */

/* Exit Critical section. */
flag[i] = FALSE;
```

```
/* GET DIGITAL INPUTS */
i = 2;/* Set process number. */

/* Entry into critical section. */
turn0 = 0; /* Defer to the other processes. */
turn1 = 1;
flag[i] = TRUE; /* I want to go. */

/* While any other process is in the critical
      section, and I am deferring to them, I
      wait. */
while((flag[0] || flag[1]) &&
     ((turn0 == 0)&&(turn1 == 1))) {
     Sleep(20);
}

/* START CRITICAL SECTION. */
/* GET DIGITAL INPUTS */
/* END CRITICAL SECTION */

/* Exit Critical section. */
flag[i] = FALSE;
```

Figure 4. This figure shows the bakery algorithm/critical section code for 3 processes in C. The code sections are nearly identical, except for the process numbers and references to the turn0 and turn1 variables.

CONCLUSION

The bakery algorithm provides a quick and easy-to-implement solution for problems that need a fair way to administer mutual exclusion for a group of processes and/or resources. By using this algorithm versus an OS dependent solution such as a semaphore, execution speed is preserved. The "Sleep" command works to prevent busy waiting, resulting in a system whose CPU slice usage is minimal. While the algorithm is usually discussed in reference to operating systems, it has provided an ideal solution for low-level communication management for our robot's hardware/software interface.

ACKNOWLEDGEMENT

This research is supported by Louisiana Board of Regents LEQSF (2007-10)-RD-A-27.

REFERENCES

1. Beaubouef, T., McDowell, P., Ice [Air] Hockey and Tennis Balls: Playing at Computer Science Research with Robotics, *SIGCSE Bulletin*, December 2007
2. Lakin, M., A Robotic Air Hockey Opponent; *CCSC-SC* Hammond La., April 2009
3. Boston Dynamics, "BigDog - The Most Advanced Rough-Terrain Robot on Earth," http://www.bostondynamics.com/robot_bigdog.html, October, 2009.
4. SD84 Technical Specification; http://www.robot-electronics.co.uk/htm/sd84tech.htm
5. Silberchatz and Galvin, *Operating Systems Concepts*, 5th Edition, Addison Wesley, 1998.