Real-Time Control of Multiple Servo Motors

Rocky Ratia

***Abstract* — Moving a servo motor at a set speed which is not the maximum speed can be achieved with relative ease. However, many ways involve stepping the servo, which in turn will force the program to halt until the desired position is reached. With the short limit of 2 minutes, waiting for each servo to move to the goal position can take up a lot of time. This paper explores various methods of simultaneous servo control. Through a series of experiments, the author compares the effectiveness of different methods and shows that utilizing the Wombat's multithreading ability can greatly improve the flexibility and control of multiple servo motors.**

I. Introduction

Autonomous robots are programmed and designed to complete various tasks accurately and repeatedly. Because most designs and strategies account for one robot performing multiple tasks, and because of the nature of some tasks, teams tend to use some form of arm mechanism.

II. Types of Arms

Simpler arm mechanisms are usually a form of rotating arm. The most common and user inclusive design is a 1-DOF revolute joint manipulator [1]. However, some tasks require a grabbing mechanism to stay upright. The two common ways to do this are to add a joint to the arm near the claw, or to use a shifting parallelogram. In the case of adding a joint, the arm now becomes a 2-DOF planar articulated manipulator in RR configuration. 'R' means revolute joint, therefore an arm in RR configuration would have 2 revolute joints. In a more advanced robot, the arm pivots in place. To achieve this, the arm can be mounted on a servo motor. The arm is now a 3-DOF articulated robotic arm in RRR configuration [2].

To control an arm that has 2 or more articulatory servo motors, the program now needs to control two servos for the arm with a joint, one being the joint and the other being the base. In the case of the pivoting arm, the program will need to be able to control 3 servos simultaneously. The bearing, (or pivot), the base, and the joint.

III. The Environment

The controller given is the Wombat, which uses a Raspberry Pi 3B+ for its main computing. Powered by a quad-core ARM Cortex-A53 processor at 1.4 GHz, the Pi offers a decent balance between performance and efficiency [3]. With a 4 core processor, the Pi is capable of modest multithreading. Using the given OS thread scheduling ability, the Pi is more than capable of running parallel tasks.

IV. Mechanical Structure

The arm needs to be operable by any of the servos in use. If even one of them fails, the stepping program will have no way to determine the failure without complex use of sensors. This greatly limits the possible design; as the torque range of the standard issue servo at 6 volts is ~11 kg · cm.

$$T_{total} = g \cdot \left[\left(\frac{L_1}{2} \cdot m_1 \right) + (L_1 \cdot m_2) + \left(\left(L_1 + \frac{L_2}{2} \right) \cdot m_2 \right) + \left((L_1 + L_2) \cdot m_c \right) \right] \cdot \sin\theta$$

Fig. 1. The equation for finding torque for a given two linkage arm [4].

Using the measurements of 20 cm for the first arm linkage and 24 cm for the second arm linkage, the maximum torque put on the servo is about 10 kg·cm with rounding. Because this torque is within the servo's operating range, and that the load decreases along upper linkages of the arm, the base and joint servos can safely operate within their limits.

V. Servo Operation

In addition to a stable, operable hardware structure, advanced programmatic logic is required to prevent undesirable, sudden movements that can add to the servo's workload and possibly damage the arm. The provided function for moving a servo, `set_servo_position`, on its own, is undesirable for this use case as it forces the servo to jump to the position. This is not a slow, graceful movement, more like a jerky sudden shift. To slow down the servo, a widely used method is to "step" the motor, or loop through positions and move the servo by smaller increments. There are two ways to do this using loops, a 'while' loop or a 'for' loop. Either of these is a valid way to step the servo.

For a revolute joint manipulator, without joints separating the base from the claw mechanism, this method would be well suited for the purpose of not damaging the arm. However, for arms with more complex joints, such as the 2-DOF and 3-DOF articulated arms, this method can add a lot of timing issues, particularly if the game plan the robot is being designed for requires completion of many tasks, with little room for error or time flexibility. Moving each joint one at a time would not only be cumbersome, but would also add extra stress on some motors as the joint(s) above them are actuated.

VI. Programmatic Synchronisation

To be able to step two joints at any given moment, without forcing the program to halt in a loop, there are several options available, namely timer based state machines, fire and forget threading, and multi threading. Each of these methods has both advantages and disadvantages.

A timer based state machine is a type of state machine where transitions between states are driven by time. In the case of a more complex arm, this method could be to set multiple positions of servos at regular intervals, though this would require precise calculation of the step rate of the motor. The advantage to this would be that this is a more simpler, failproof way of controlling the arm. A key disadvantage would be the calculation required, and the careful tuning needed to properly synchronise the arm.

Another way to control this arm, would be to use fire and forget threading. Fire and forget threading is when a program starts a thread, and leaves it to run in the background without acknowledging it for future reference. A key advantage to this would be that the program can create several threads, synchronously controlling the arm, without blocking the main program allowing it to control other important aspects of the program, such as driving or sensor reading. A disadvantage would be that because the thread has been created and "forgotten", hence the name fire and forget, concurrent modification of servo positions can result in serious damage to the servo and to the arm mechanism. Essentially, the servo will be stepped to two different positions, each at increasingly large distances from each other.

Multithreading is quite similar to fire and forget, however it has more refined control. By keeping track of created threads, the program can halt one thread in favor of the most recent position goal to prevent the concurrent position modification seen in fire and forget. This refined control allows for better timing, synchronisation, and speed control of the arm.

```
void step_arm(int target_base_position, int target_joint_position) {
    int current_base_position = get_servo_position(2);
    int current_joint_position = get_servo_position(3);
    while (current_base_position != target_base_position || current_joint_position != target_joint_position) {
        if (current_base_position < target_base_position) current_base_position ++;
        else if (current_base_position > target_base_position) current_base_position --;
        if (current_joint_position < target_joint_position) current_joint_position ++;
        else if (current_joint_position > target_joint_position) current_joint_position --;
        set_servo_position(2, current_base_position);
        set_servo_position(3, current_joint_position);
        msleep(1);
    }
}
```

Fig. 1. An outline of a function for control of multiple servos, using a timer based state machine.

VII. Implementation

The implementation of each of these methods ranged from relatively simple to quite complex. The timer based program was by far the simplest, using a basic while loop, to simultaneously set the positions of two or more servos at the same time. As shown in Fig. 1, transitions in states, namely changes in servo position are driven by time, in this case the sleep function call at the end. This will still halt the program, however, until the while loop finishes. Another disadvantage is that because the movement for both motors is inside the same while loop, the function is less flexible with individual motor speed control.

For fire and forget threading, the implementation is more complex. To even start controlling the arm using this method, it is imperative to first understand KIPR's threading system.

```
void do_something(){
    printf("Something has been done\n");
}

thread new_thread = thread_create(do_something);
thread_start(new_thread);
```

Fig. 3. A basic example of threading use is shown.

In Fig. 3, a key observation is that the threading system runs a thread from a function. After `thread_start` is called, everything in that function will run separately from the main program. For example, if from Fig. 3 the contents of the void function `do_something` is replaced with `while(1) msleep(1000)` or another form of forever loop, the program would not halt. Instead, it would run as normal. Threads run parallel to the main program. The thread still has access to servo control.

```
void operate_servo(int port, int target_position){
    int current_position = get_servo_position(port);
    while (current_position != target_position) {
        if (current_position < target_position) current_position ++;
        else if (current_position > target_position) current_position --;
        set_servo_position(port, current_position);
        msleep(1);
    }
}
void thread_function_to_be_executed(){
    operate_servo(2, 1155);
}
void move_servo(){
    thread servo_thread = thread_create(thread_function_to_be_executed);
    thread_start(servo_thread);
}
```

Fig. 4. A basic implementation of moving a servo using threading

If the code from the timer based state machine, namely the while loop, is adapted to run a single servo motor, and this new loop is packed into a thread function, the thread can now step the servo without blocking the program. This can be seen in Fig. 4, which is a basic outline to move a servo to a position in a nonblocking manner. However, threads do not support passing in function parameters. This makes sending the goal position of the servo much more complex.

```
void move_servo(int port, int pos){
    thread servo_thread = thread_create(operate_servo(port, pos));
    thread_start(servo_thread);
}
```

Fig. 5. An incorrect implementation of moving a servo. This will not work.

Code from Fig. 5 will result in a compiler error. To get around the lack of passing function arguments, variables can be used. Because the threads execute functions, global variables can pass values into the threads. Implementing this, where the function that is called to move the servo sets these variables the calls the threads,

```
int base_port = 2;
int joint_port = 3;

int base_target_position = 680;
int joint_target_position = 1623;

void operate_servo(int port, int target_position){
    int current_position = get_servo_position(port);
    while (current_position != target_position) {
        if (current_position < target_position) current_position ++;
        else if (current_position > target_position) current_position --;
        set_servo_position(port, current_position);
        msleep(1);
    }
}
void base_thread_function(){
    operate_servo(base_port, base_target_position);
}
void joint_thread_function(){
    operate_servo(joint_port, joint_target_position);
}

void step_arm(int joint_pos, int base_pos){
    joint_target_position = joint_pos;
    base_target_position = base_pos;

    thread base_thread = thread_create(base_thread_function);
    thread joint_thread = thread_create(joint_thread_function);

    thread_start(base_thread);
    thread_start(joint_thread);
}
```

Fig. 6. A basic implementation of fire and forget threading.

In Fig. 6, notice how the `step_servo` function never waits for the servos to reach their position. This can be a goal but also not desired, as in some cases the program is required to wait for the arm to finish moving. An implementation of a basic wait mechanism is through more use of global variables.

```
int joint_port = 3;

int base_target_position = 680;
int joint_target_position = 1623;

void operate_servo(int port, int target_position) {
    int current_position = get_servo_position(port);
    while (current_position != target_position) {
        if (current_position < target_position)
            current_position++;
        else if (current_position > target_position)
            current_position--;

        set_servo_position(port, current_position);
        msleep(1);
    }
}

void base_thread_function() {
    operate_servo(base_port, base_target_position);
}

void joint_thread_function() {
    operate_servo(joint_port, joint_target_position);
}

void step_arm(int joint_pos, int base_pos, bool wait_for_completion) {
    joint_target_position = joint_pos;
    base_target_position = base_pos;

    thread base_thread = thread_create(base_thread_function);
    thread joint_thread = thread_create(joint_thread_function);

    thread_start(base_thread);
    thread_start(joint_thread);

    if (wait_for_completion) {
        thread_wait(base_thread);
        thread_wait(joint_thread);
    }
```

Fig. 7. Implementation of a completion mechanism.

Fig. 7 shows how such a completion mechanism can be implemented. The builtin function `thread_wait` halts the program until the thread has finished. This, in and of itself, is a form of completion mechanism. Another function parameter can be added to declare whether the completion delay is desired, but this is a basic outline. Using the code from Fig. 7, the thread variables are declared inside of the function. For fire and forget threading, this is acceptable. However, for multithreading, this is unwanted. That is because multithreading requires prolonged access for the same thread. By declaring threads outside of any function, their persistence is ensured and access is universal. Speed adjustments and arm position interrupts can also be used, as well as moving the arm while driving the bot. Fig. 8 gives

an example of this implementation.

```
int base_port = 2;
int joint_port = 3;

int base_target_position = 680;
int joint_target_position = 1623;


void operate_servo(int port, int target_position){
    int current_position = get_servo_position(port);
    while (current_position != target_position) {
        if (current_position < target_position) current_position ++;
        else if (current_position > target_position) current_position --;
        set_servo_position(port, current_position);
        msleep(1);
    }
}
void base_thread_function(){
    operate_servo(base_port, base_target_position);
}
void joint_thread_function(){
    operate_servo(joint_port, joint_target_position);
}

thread base_thread = thread_create(base_thread_function);
thread joint_thread = thread_create(joint_thread_function);

void step_arm(int joint_pos, int base_pos, bool wait_for_completion){
    joint_target_position = joint_pos;
    base_target_position = base_pos;

    thread_start(base_thread);
    thread_start(joint_thread);

    if(wait_for_completion){
        thread_wait(base_thread);
        thread_wait(joint_thread);
    }

}

void interrupt_arm_movement(){
    thread_destroy(base_thread);
    thread_destroy(base_thread);

    //Recreate threads to ensure future use
    base_thread = thread_create(base_thread_function);
    joint_thread = thread_create(joint_thread_function);
}
```

Fig. 8. Implementation of advanced thread control, motion thread interrupts, and universal threading access. Note that the threads need to be declared after functions like `operate_servo` that they rely on.

VIII. Testing

Through testing the arm's response to various methods, the most effective, safest, and fastest method was the use of multithreading. The timings for each of the motors is a big issue. With the timer based method, 2 of the servo motors operating the arm can reach their goal position, with one still moving. This would halt the program for one motor. Implementing further logic into a structure similar to that of Fig. 8 can resume the program when one servo is close enough to its goal position. This could even be done for fire and forget, though persistent thread control logic would make that some version of multithreading. The testing method used to determine efficiency was having a robot with a 3 DOF arm attached, and timing each of the 3 methods.

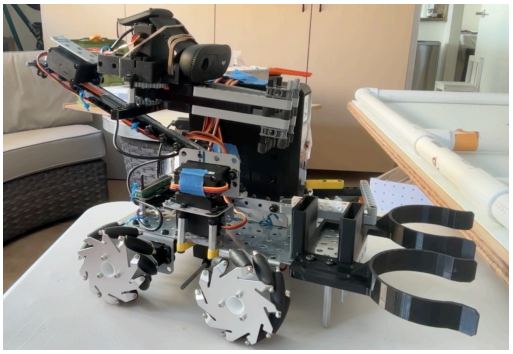

Fig. 9. The robot used in the tests.

Each method was run a total of 5 times. A higher speed camera was used to record every run, and each run the goal arm position stayed consistent. Each servo on the arm was required to turn to a different position.

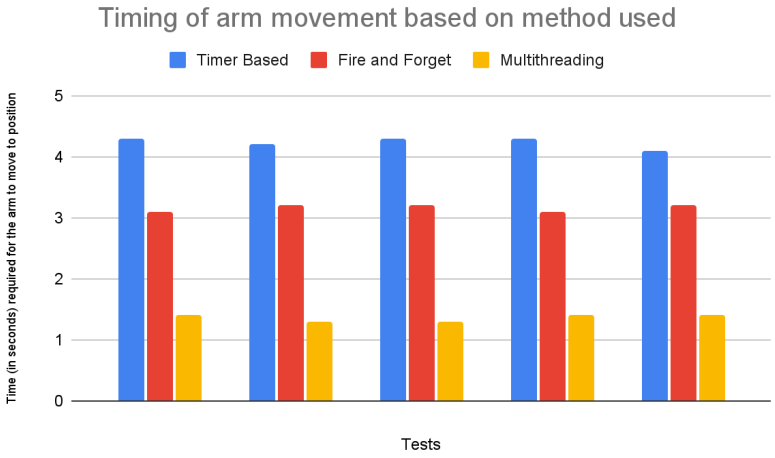

Fig. 10. Results from the five tests covering each method. The vertical axis is the time (in seconds) required for the arm to move to the goal position. The goal position and starting stayed constant over these tests.

As seen in Fig. 10, the timer based control system was more inefficient. The reason for this was that because every servo had a different position, the program would need to wait for the last servo. Fire and forget featured better servo speed management, but the issue of needing to wait for the arm was still prevalent. Multithreading had the feature of only waiting for the more dominant servos to finish their goal, saving larger amounts of time. It also allowed for even better speed control, and acceleration.

Acceleration was not possible with fire-and-forget, because it required persistent thread access to adjust speeds.

IX. Conclusion

Real time control of multiple servo motors is essential for operating a variety of arm designs. There are many methods of synchronous control, ranging from timer based methods to implementations of threading. Timer mechanisms use timing to transition between arm states. One method of threading, fire and forget, can use the Wombat's threading functionality to control the arm, and finally multithreading uses more advanced threading to control speed and handle multiple calls to the arm. Every method has advantages and disadvantages. Timer based methods are often simpler, easier to implement, and straightforward to debug. This method can be more rigid in arm speed and timing. Fire and forget features more complex control schemes, and also a more nonblocking manner than a timer based method. A key disadvantage is that attempts to step the servo at the same time to different positions can lead to sudden, and possibly damaging movements. Multithreading offers ways to interrupt the threads, preventing this issue. It also offers acceleration of servo, more options for complex movement patterns, and more flexibility in speed of individual motors. The biggest disadvantage is its complexity, and difficulty to implement. All in all, synchronous control of multiple motors is required for complex autonomous robots, such as in Botball, that feature more compound mechanisms.

Acknowledgements

The author would like to thank Coach Harsukh, the members of Botball Team 328, and the members of Etheris for hosting this publication.

References

[1] M. W. Spong, S. Hutchinson, and M. Vidyasagar, Robot Dynamics and Control. New York, NY, USA: Wiley, 2005.

[2] J. J. Craig, Introduction to Robotics: Mechanics and Control, 3rd ed. Upper Saddle River, NJ, USA: Pearson, 2004.

[3] Raspberry Pi Ltd., "Raspberry Pi 3 Model B+," *Raspberry Pi*, [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/. [Accessed: Jun. 11, 2025].

[4] "7.5: Torque," *Physics LibreTexts*, University of California Davis, Aug. 13, 2020. [Online]. Available: https://phys.libretexts.org/Courses/University_of_California_Davis/UCD%3A_Physics_7B_-_General_Physics/7%3A_Momentum/7.5%3A_Torque. [Accessed: Jun. 11, 2025].