An Event-Driven Operating System for Servomotor Control

Geoff Nagy[1], Andrew Winton[2], Jacky Baltes[3], and John Anderson[4]

Autonomous Agents Lab
University of Manitoba
Winnipeg, Manitoba
Canada, R3T 2N2
[1]geoffn@cs.umanitoba.ca
[2]umwintoa@cc.umanitoba.ca
[3]jacky@cs.umanitoba.ca
[4]andersj@cs.umanitoba.ca

Abstract. Control of a servomotor is a challenging real-time problem. The embedded microcontroller is responsible for fast and precise actuation of the motor shaft, and must handle communication with a master controller as well. If additional tasks such as temperature monitoring are desirable, they must take place often enough to be useful, but not so frequently that they interfere with the operation of the servo. Since microcontrollers have limited multi-tasking capabilities, it becomes difficult to perform all of these tasks at once. It was our goal to create servo firmware with high communication speeds for humanoid robots, and our solution is generalizable to non-humanoid motor control as well. In this paper, we present an event-driven operating system for the Robotis AX-12 servomotor. By using interrupts to drive functionality that would otherwise require polling, our operating system meets the real-time constraints associated with controlling a servomotor.

1 Introduction

Servomotors are used extensively in robotics to give rise to motion—arms, legs, torsos, and other appendages contain servos which, in response to commands from a master controller, must move in a coordinated, timely fashion to walk, run, or perform other tasks. Control of a servomotor is a task with many real-time constraints. Of these, the most crucial is timely response to positional feedback data, in order to actuate the servo to a target position. Failing to meet this real-time constraint could mean anything from a crashed remote-controlled airplane to the loss of more expensive equipment. Actuation commands are given by a master controller, and thus a servo is also responsible for listening to these commands and responding as quickly as possible: a servo moving in response to an actuation command that was given two seconds ago is virtually useless.

The ability of a servo to monitor its status (temperature, torque load, etc.) is also a desirable feature. Overheating, for example, is a situation that should be

avoided to prevent damage. Servos that are able to sense their own status can shut down in dangerous conditions, or provide relevant sensor data to a master controller to produce a more intelligent response and avoid the same equipment losses associated with failure to meet real time constraints.

The main challenge in meeting these requirements stems from a microcontroller's limited ability to multi-task. Analog-to-digital (A2D) conversions for reading onboard sensors take time away from the servo's primary function, which is to actuate to a specific position with minimal latency or jitter. Although temperature reads, for example, do not need to occur frequently, a servo must continually check the potentiometer connected to the motor shaft to make sure it is rotating to the correct position. Doing so requires constant A2D conversions, which take time and can cause latency in the servo response if implemented inefficiently. This becomes a challenging real-time problem, and it is the responsibility of a servo firmware designer to ensure that these constraints are met.

It was our goal to create a servo firmware implementation that would allow us to communicate with multiple servos at extremely high speeds. Our intention was to use this firmware to control the motors in humanoid robots such as the DARwIn-OP or Bioloid, but our solution has applications in other areas as well. High communication speeds with motors is desirable because latency in complex motor tasks can lead to equipment failure. (For example, a bipedal robot might fall over if it loses its balance due to servo communication latency.) As the number of motors increases, this problem is compounded since addressing more servos requires more time.

Our target motor platform was the Robotis AX-12 servo, as shown in Figure 1. Our approach applies to other programmable servos as well. In order to meet our goal of faster communication times, we needed to create firmware that would be capable of handling the servo tasks described above. To support this, we endeavoured to develop an operating system for the AX-12's embedded Atmel ATmega8 microcontroller.

It was immediately apparent to us that polling techniques for serial communications or actuation would not suffice. Simply blocking and waiting for A2D reads or bytes via serial would not be practical, since spending too much time on one task would take time away from others. Our solution was to develop an interrupt-driven operating system, and was motivated by several factors. In general, interrupt-based systems eliminate the need for polling, and are ideal for low-power microcontrollers [1] [2]. They are also ideal for memory-constrained applications [3] [1] [4]. In our case, embracing an event-driven approach allowed us to update the motor control pulse-width modulation (PWM) signals as often as A2D conversions could take place, and enabled the use of interrupts to handle serial communications and timer functions. The following section describes related work, and the section afterwards describes our operating system, Dorkeus, in detail.

Fig. 1. A Robotis AX-12 servo.

2 Related Work

Baltes et al. [5] developed a multi-threaded, real-time kernel for the Atmel ATmega128 microprocessor called *Freezer OS*. It handled scheduling of multiple tasks pre-emptively in a round robin fashion, and included support for interrupt-based A2D conversions and serial communications. The goal in its creation was to ease application development of robotic firmware. In contrast, our approach is entirely interrupt-based and was specifically designed for servomotor control applications.

Node.js is a Javascript framework that firmly embraces the event-driven paradigm [6] [7]. Using this framework, Web developers specify events that must be responded to, and provide callbacks that should execute to handle the events requiring response. This interrupt-based approach is similar to our own, whereby events are handled without the need for polling. Our work differs from Node.js in that our approach has been tailored to meet various real-time constraints associated with servo control. Additionally, our work applies the event-driven paradigm to a control application, rather than a web-based server-side framework.

TinyOS is an event-driven framework that supports the development of real-time embedded operating systems [1]. Intended for use in sensor networks, its small size and event-based design make it well-suited for running on low power, low cost microcontrollers. TinyOS programs can be written modularly without the need for a global understanding of the entire system [1], and this is made possible by the self-contained nature of the interrupt handlers that respond to various events. In this regard, our approach is quite similar. The TinyOS system, however, was designed specifically for large sensor networks, while our application is meant to address the real-time constraints associated with motor control. A similar system is *Contiki* [8], a C-implemented operating system developed for use in large sensor networks. It runs an event-driven kernel that also supports

pre-emptive multi-threading. Our approach differs from Contiki in that Dorkeus is entirely event-driven, and is meant for servo control applications.

Åarzén [9] described an event-based proportional-integral-derivative (PID) controller that only generates output responses when the input signal goes beyond a certain threshold. Improvements were made by Durand and Marchand [10], in which the controller avoids re-computation of the output signal in cases where the input signal remains unchanged. While our own approach is event-driven, our PID control is not triggered by the measured input signal, but rather by the completion of an A2D conversion. In addition to PID control, our operating system is responsible for handling serial communications and timer facilities as well.

OpenServo [11] is an online community project dedicated to developing firmware for various servomotors on different microcontrollers. It supports a large number of serial commands for servo control and is partially driven by interrupts. In contrast, our approach is entirely event-driven, and all of our processing takes place in the interrupt context. The *ActuatedCharacter* [12] project is another effort whose goals involve improving communication speeds with servos. This approach specifically addresses the AX-12, and features a small, fast communications protocol. Unlike our approach, it is not entirely interrupt-driven.

3 Dorkeus Operating System

The Dorkeus OS is a fully event-driven servo operating system—it relies entirely on interrupts to drive its functionality. Our implementation is such that virtually all of our processing occurs within the interrupt service routine (ISR) context. While simple computations taking place in an interrupt context do not pose a problem, performing expensive computation is often not desirable for a number of reasons. These include (a) the possibility that an interrupt of the same type can be missed without taking extra precautions, and (b) the fact that ISR code might need to disable interrupts for longer periods of time in order to avoid race conditions, possibly resulting in lost events. One solution to both problems is to offload time-consuming computations to the main task context. Fortunately, the computation performed in our operating system is intentionally minimal. Thus, the processing associated with each event does not require it to be handled outside of an ISR context, resulting in simpler, event-driven code that is easier to maintain and debug.

Our operating system is composed of three subsystems as shown in Figure 2: the control loop, the serial communication system, and the timer queue. The following subsections describe these systems in detail.

3.1 Control Loop

The control loop in Dorkeus is responsible for actuating the motor to a specific position—this is the highest-priority task of any servomotor firmware. In our approach, the main control algorithm is a PID controller. Positional feedback

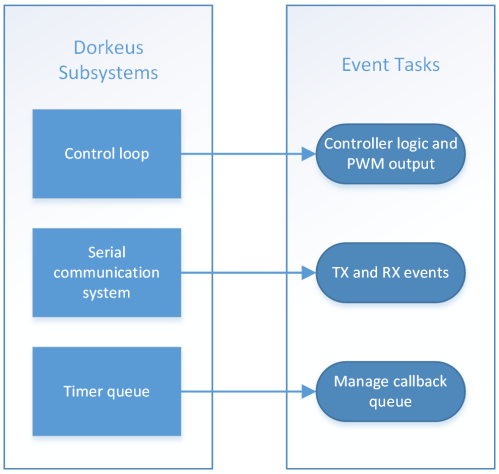

Fig. 2. The three subsystems in Dorkeus and the processing associated with each of their events.

is obtained by reading the value of the motor shaft potentiometer using the embedded ATmega8's analog-to-digital converter (ADC). An A2D conversion on an ATmega8 can take anywhere between 13μs to 260μs [13], which means that pausing to wait for a positional read would result in a delay that could otherwise be used to perform useful work. We used an interrupt-based approach to mitigate this problem.

In our approach, we trigger an A2D conversion inside an infinite loop. Whenever an A2D conversion completes, the corresponding interrupt is executed. Inside this interrupt, the PID controller is used to compute the response of the motor. Then, this response is used to adjust the output PWM signal. The hardware configuration of the AX-12 dictates that this signal is output using the ATmega8's only 16-bit timer. Once this process is complete, another A2D conversion begins.

This method has the advantage of not continually polling and waiting for the ADC result: when the conversion result is ready, it is fed into the controller logic. This ensures that the servo responds with minimal latency, since it responds to positional errors as quickly as the A2D conversions can finish. Using interrupts in this fashion means that while an A2D conversion is taking place, the ATmega8 is free to perform other useful work.

We chose to use a PID controller in our approach due to its generalizability and ease of implementation. It is also used in the original AX-12 firmware provided by Robotis [14]. Our firmware implementation does not depend on any particular control algorithm, and so it would be a simple matter to substitute our PID controller for another control algorithm.

3.2 Serial Communication System

The serial communication system is responsible for listening to serial commands from the master controller and responding as necessary. We chose to embrace an event-driven approach for our serial communications. This eliminated the need to poll for serial data.

Supported commands include actuation instructions, pings, and servo position read requests. A key element in our implementation is the communication protocol's support of a *synchronous read* command that does not exist in the original Robotis firmware. This command enables each servo to send positional data one after the other, after an initial synchronous read request, without further management from the master controller. This was inspired by the *chain reply* that is used by the ActuatedCharacter firmware [12].

The AX-12 hardware is configured to use a half-duplex communication protocol with a main controller (i.e., it can either send or receive data, but it cannot do both at the same time). Initially, the servo is in receive mode. Every time a byte is received by the ATmega8 hardware, a *receive-complete* interrupt executes. Within the interrupt, the byte is fed through a simple state machine that interprets the serial commands. Once the machine reaches a terminal state (i.e., the whole message is received), the complete command is executed. If the command requires it, the ATmega8 will transmit a response.

Message transmission works in a similar manner. The master-slave architecture of the communications and the half-duplex hardware dictate that the servo should only transmit messages in response to commands from a main controller. Therefore, once a complete message is received, Dorkeus can switch from receive to transmit mode, and begin the process of generating a response. With *transmission-complete* interrupts enabled, the servo sends the first byte of the response. The transmission-complete interrupt then fires as a consequence of the completed byte transmission. The interrupt logic then determines if another byte needs to be sent. If so, the next byte is transmitted, triggering another interrupt, and so on. This cascading effect allows all response bytes to be sent from the ATmega8 as quickly as possible. Eventually, when all bytes have been sent, the last transmission-complete interrupt finishes without sending anything, thus ending the chain of byte transmissions. The AX-12 then switches back to receive mode to await the next serial command.

3.3 Timer Queue

Certain low-priority events, such as measuring slow-changing sensor data (e.g., temperature), do not have to occur frequently, but must happen regularly enough to be useful. For example, it is only necessary to check once approximately every second if a servo is overheating. To facilitate this, we have developed an interrupt-based timer system with which events can be scheduled for execution in the future. Events are registered by the user specifying (a) the time delay in milliseconds, and (b) the callback function that should be executed when that time expires. Listing 1 shows the pseudocode of our implementation.

```
void timerExpire()
{
  TimerEvent e = queue.getFront()
  e.elapseTime(getTimerElapsed())
  if e.isTimeExpired
    queue.dequeue.fire()
    foreach i in queue
      i.elapseTime(front.getFullDelay())
      if i.isTimeExpired
        queue.dequeue().fire()
  else setTimer(e)
  if !queue.isEmpty() setTimer(queue.getFront())
}
void setTimer(TimerEvent t)
{
  if t.getDelay() > MAX_TIME
    setTimerInterrupt(MAX_TIME)
  else
    setTimerInterrupt(t.getDelay())
}
void registerTimerEvent(Callback c, int ms)
{
  TimerEvent e = TimerEvent(c, ms)
  Time t = getTimerElapsed()
  if queue.isEmpty() setTimer(e)
  else
    int delay = queue.getFront().getDelay()
    if ms < (delay - t)
      setTimer(e)
      queue.getFront().elapseTime(t)
  queue.orderedEnqueue(c)
}
```

Listing 1. Pseudocode for our interrupt-based timer implementation.

The timer system is implemented as a priority queue, sorted by time in non-descending (i.e., increasingly distant) order. When an event is added to the front of the timer queue, the ATmega8's 8-bit timer should be set to expire exactly when the first event should occur. Unfortunately, the maximum amount of time allowed by an 8-bit timer in our implementation is 16.32 milliseconds, which is generally too small an amount to be useful. This limitation is due to the speed of the external clock used (16Mhz), and even with a timer prescaler of 1024, the maximum timer value is rather short. We are unable to use the 16-bit

timer on the ATmega8 since it is already in use by the PWM output logic. Our calculations are shown below in Equation 1.

$$maxtime = \frac{255}{\left(\frac{16MHz}{1024}\right)} = 16.32ms \tag{1}$$

To bypass this limitation of the 8-bit timer, events that are scheduled to occur sufficiently far into the future (i.e., later than 16ms from when the event was scheduled) set the 8-bit timer to expire in 16ms. When the expiration occurs, the timer logic examines the event at the head of the queue and determines if there is any time remaining before the event fires. If there is time remaining, the timer is reset again for either (a) the remaining amount, if the delay is less than 16ms, or (b) the maximum time if there are still 16ms or more remaining.

If the timer expires and an event is ready to fire, the function associated with that event is invoked. Figure 3 illustrates the execution of two events. The total amount of time in milliseconds that it took the front event to fire is subtracted from the respective times of the remaining events to allow for their consideration of the passage of time. The timer queue is then checked for any additional events whose time may have expired. Any events that are ready to execute then do so in the proper order.

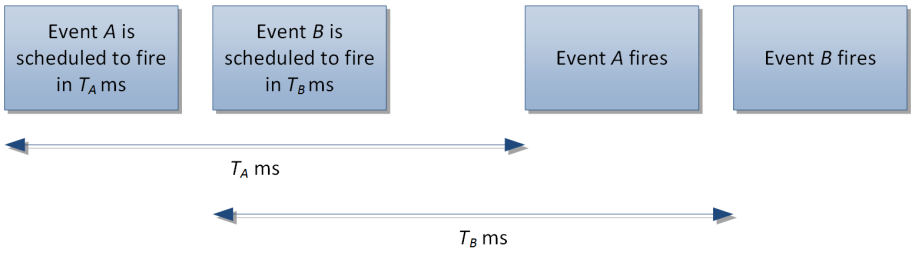

Fig. 3. Shows the scheduling and execution of two timer events.

It should be noted that there is some imprecision associated with our timer implementation. Our timer delays are expressed in whole millseconds, and the maximum delay that can be implemented using an 8-bit timer in our approach is 16.32ms. When setting the 8-bit timer counter register, we multiply the delay in millseconds by 15.625 (since $\frac{16MHz}{1024} = 15.625$ ticks per millisecond) and then round the resulting number to the nearest integer. We were not concerned with sub-millisecond accuracy, so imprecisions due to rounding were not an issue. If these imprecisions were problematic, we could have solved this problem by using a different prescaler (such as 64, which would give $\frac{16MHz}{64} = 250$ ticks per millisecond) that would remove the need to round. An alternative method would be to use a prescaler of 256, which would give $\frac{16MHz}{256} = 62.5$ ticks per

millisecond, and would allow us to specify delays in terms of half-milliseconds. A third option would be to simply alter our timer queue to work in microseconds instead of milliseconds, but this would necessitate the use of **unsigned long**s and would use more memory. We chose to use the highest prescaler available to ensure that the system spent as little time handling the timer queue as possible.

4 Performance

Our operating system achieves our goal of faster communication speeds with a master controller. Using our interrupt-based approach and minimal communication protocol, we have achieved highly reliable communication at speeds as high as 1Mbps with up to 10 servos. Both the original Robotis firmware and Dorkeus support the ability to send and receive positional data to and from servos, and additional tests have revealed that our protocol enables us to continually send and receive positional data at a rate nearly ten times faster than the original firmware on the AX-12 for an equal number of servos. The use of our synchronous read command contributes greatly to these reduced communication times when compared to the original Robotis firmware, which requires positional data to be requested individually from each servo, one at a time. Since our communication protocol can address up to 32 unique IDs, the system as it stands should be able to perform with similar success with a larger number of servos. This would enable much faster control of the servos in a robot, allowing faster motion updates and helping to reduce the probability of falling or becoming unbalanced. We intend to discuss our protocol in more depth in a future paper.

Our AX-12 operating system is entirely event-based; all actions that occur do so as a result of either an internal or external event. This has certain implications and presents challenges to which traditional time-based serial implementations are not susceptible. Consider the Watchdog Timer (WDT) present in the ATmega8.

The WDT's primary function is to detect nonprogress in the operation of a microcontroller. At regular intervals, onboard firmware should reset the WDT counter to indicate that progress is being made. Should the WDT expire (perhaps due to an infinite loop), a system reset command will be issued if the proper configurations are made. This ensures that the system is restarted if it fails to make progress. While this method works with conventional approaches, it will not succeed in an event-driven environment where there is no guarantee that progress will be made in finite time. It is conceivable that the WDT could expire in between events, undesirably resetting the system.

While disabling the WDT reset functionality (by, for example, enabling the WDT interrupt) can prevent undesirable resets, this defeats the main purpose of the WDT, which is to detect nonprogress. Since inter-event progress is impossible to determine in an interrupt-based implementation, progress must be considered at the event level itself. In other words, if a single event fails to make progress on its own, the system should reset. A detailed examination of possible approaches is beyond the scope of this paper, but one solution might be to implement an

idle task that continually runs in the background, resetting the WDT counter at regular intervals. Should progress be halted in an event handler, the idle task will not run, causing the WDT to expire, which will in turn effect a system reset.

5 Future Work

The minimal nature of the computations performed by the ATmega8 in our implementation means that all of our events can be handled in an ISR context. More complex events and processing—such as inverse kinematic computations—would require much more intense calculations and should be run in the main task (i.e., non-interrupt) context. We intend to investigate the possibility of including such processing into our operating system to better support the inclusion of more advanced computational tasks.

Our minimal serial communication protocol, while beyond the scope of this paper, has helped contribute to the fast response times we have observed. We intend to investigate possible extensions to it in the future. Another one of our goals is to more strongly evaluate the performance of our operating system and its communication protocol by using it to control the servos on a humanoid robot—such as the Bioloid—while walking or performing other complex tasks.

6 Conclusion

In this paper, we have presented an event-driven operating system for the Robotis AX-12 servomotor for use in humanoid robotics. This approach is valuable because it allows efficient management of the many tasks that a servo must perform in real time, and is not limited to any particular servo application. Although our implementation utilizes an AX-12 motor, our approach is applicable to virtually any servo with a microcontroller. Combined with an efficient communication protocol, our operating system greatly reduces a servo's response time. In this case, polling techniques simply cannot offer the efficiency granted by the use of interrupt-based approaches. Our firmware embraces this paradigm and offers a more modern alternative in the face of multi-tasking and real-time constraints.

References

1. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al.: Tinyos: An operating system for sensor networks. In: Ambient intelligence. Springer (2005) 115–148
2. Shih, E., Bahl, P., Sinclair, M.J.: Wake on wireless: an event driven energy saving strategy for battery operated devices. In: Proceedings of the 8th annual international conference on Mobile computing and networking, ACM (2002) 160–171
3. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: ACM SIGOPS operating systems review. Volume 34., ACM (2000) 93–104

4. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In: Proceedings of the 4th international conference on Embedded networked sensor systems, Acm (2006) 29–42
5. Baltes, J., Iverach-Brereton, C., Cheng, C.T., Anderson, J.: Threaded c and freezer os. In: Next Wave in Robotics. Springer (2011) 170–177
6. Deitcher, A.: Simplicity and performance: Javascript on the server. Linux Journal **2011**(204) (2011) 3
7. Tilkov, S., Vinoski, S.: Node. js: Using javascript to build high-performance network programs. Internet Computing, IEEE **14**(6) (2010) 80–83
8. Dunkels, A., Gronvall, B., Voigt, T.: Contiki-a lightweight and flexible operating system for tiny networked sensors. In: Local Computer Networks, 2004. 29th Annual IEEE International Conference on, IEEE (2004) 455–462
9. Årzén, K.E.: A simple event-based pid controller. In: Proc. 14th IFAC World Congress. Volume 18. (1999) 423–428
10. Durand, S., Marchand, N., et al.: Further results on event-based pid controller. In: Proceedings of the European Control Conference 2009. (2009) 1979–1984
11. OpenServo Community: OpenServo. http://www.openservo.com Accessed: 2014-06-10.
12. ActuatedCharacter: AX12 Firmware | Actuatedcharacter's Blog. http://actuated.wordpress.com/ax12firmware/ Accessed: 2014-06-13.
13. Atmel Corporation: ATmega8/L Datasheet. (February 2013)
14. Robotis: Dynamixel AX-12. (June 2006)