The Real-Time Embedded System for a Humanoid: Betty

Meng Cheng Lau and Jacky Baltes

Autonomous Agent Lab,
University of Manitoba,
Winnipeg, Manitoba R3T 2N2, Canada
{laumc,jacky}@cs.umanitoba.ca
http://aalab.cs.umanitoba.ca/

Abstract. This paper investigates the efficiency of the real-time embedded system in our humanoid robot, Betty. In this paper we only discuss the upper body of Betty. Based on several experiments of different queue data structures, communication protocols and PID controller implementations, we measured and analysed the latencies and jitters of Betty's responses. The experimental results indicate the best configuration to optimise the performance of Betty's Control Program.

Keywords: Communication protocol, queue data structure, Preemptive real-time kernel, jitter and latency.

1 Introduction

Designing and implementing a real-time embedded system is one of the most challenging tasks in robot development. It is crucial to ensure that a robot responses in the expected period of time. We tested and analysed our real-time embedded system to perform in real-time by considering the latency and the jitter performances. Latency is the time between starting to send a command and receiving a response from the servos where the command has been executed. Jitter is the time variation of the latencies.

2 Hardware Configuration

Upper body of Betty consists of ten DOFs (degrees of freedom). The head has four DOFs which give pan, tilt, roll and jaw. Each of the hands provides three DOFs, a shoulder allows lateral and frontal motion, and an elbow gives lateral motion. These joints are constructed by four Dynamixel AX-12 servos to the head and three Dynamixel RX-64 servos to each of the hands as shows in Fig. 1. The main reason we choose RX-64 to construct the hands because it has higher final maximum holding torque, 64.4∼77.2 kgf.cm compare to only 12∼16.5 kgf.cm for AX-12 [1], [2]. It will allow a RX-64 servo to generate sufficient torque to support the weight of the arm.

P. Vadakkepat et al. (Eds.): FIRA 2010, CCIS 103, pp. 122–129, 2010.
© Springer-Verlag Berlin Heidelberg 2010

In order to control the servos, we use a Dynamixel's dedicated controller, CM-2+ from Robotis as the central control unit. With its `AVR ATmega128` microcontroller, we implement our real-time embedded system in C language which establishes communication with the servos. The connections on the CM-2+ is illustrated in Fig. 2 and Fig. 3 shows the overview of Betty's upper body.

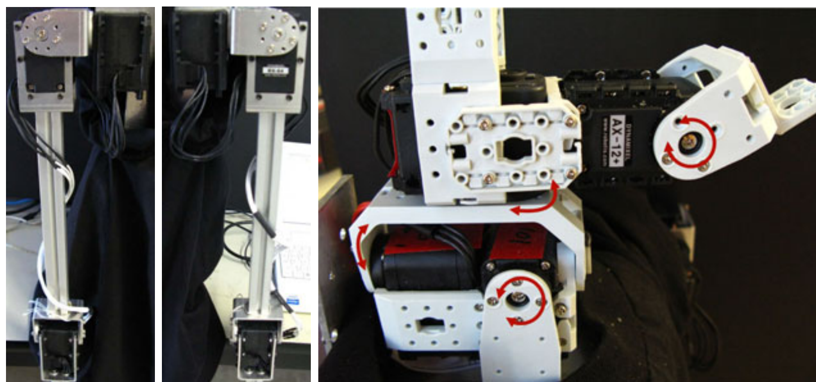

Fig. 1. Betty's upper body construction

Fig. 2. CM-2+ Connection

Fig. 3. Overview of Betty's upper body

3 Implementation

We program the CM-2+ embedded system in C language as the Control Program to control servos. The Motion Editor program which is written in C++ language is introduced to enable instruction and response communication between the CM-2+ and host computer. Fig. 4 shows the overall framework of

the system that explains the connection architecture of the robot's Motion Editor, Control Program, CM-2+ and servos. In order to develop an optimised embedded system, we employed different communication protocols, queue data structures, Preemptive multi-tasking kernel and PID controller in this project.

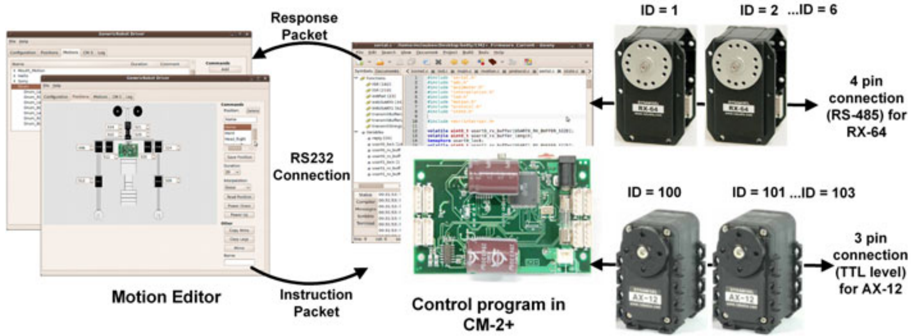

Fig. 4. General program framework of Betty

3.1 Preemptive Multi-tasking Kernel

We use Preemptive multi-tasking kernel in the Control Program to handle task switching. Currently our kernel supports four tasks. First task is an idle task to toggles a LED which is always ready to run to ensure that at least one thread is running and the Control Program is working properly. The second task is the PC thread that handle serial port communication between the Motion Editor program and CM-2+. Third task is the servo thread which will prepare commands to send from CM-2+ to the RX-64 and AX-12 servos. Finally, the last task is the torque thread that reads the current torque of each servo and sends it through the PC thread. The Preemptive multi-tasking kernel uses a timer interrupt to switch between different threads at an initial timer frequency of 10Hz. To execute task switching, the kernel will save the complete task state on the stack and then store the stack pointer in the task control block (TCB).

In our Preemptive multi-tasking kernel, we use *Timer3* ISR (interrupt service routine) to trigger the task switching together with the *Timer3* overflow interrupt vector `TIMER3_OVF_vect`. Then `ISR_NAKED` is added to the program which is responsible to preserve the machine state including the `SREG` register, as well as placing a *reti()* at the end of the interrupt routine and `ISR_BLOCK` where global interrupts are initially disabled by the AVR hardware when entering ISR. So by using the `ISR_NAKED`, *inline asm* can be embedded in our C program to perform context switching in ISR. The correct task ids and stack sizes should be allocated correctly in the TCB. The main advantage of Preemptive scheduling is real-time response on the task level, because the time to tick a task is mainly depends on the interrupt latency [5].

3.2 Communication Protocols

The Control Program uses three different types of communication protocols which are Absolute Position, Sliding Resolution and Difference protocols. Based on these protocols, the Motion Editor program from host computer will send instructions to the CM-2+'s Control Program. For instants, we use `SyncWrite` command to move the servos to specific positions. Table. 1 shows the structures and the differences between these protocols where `SyncWrite` commands were sent to move all servos to move from position 512 to 520 at the speed of 100. We divided the positions and speed by 4 so each of them can be wrapped into one byte in hexadecimal.

According to Table 1, each `SyncWrite` command of the Absolute Position protocol will send 14 bytes of instruction in one packet. On the other hand, Sliding Resolution protocol will send its command in two separate packets which are high and low packets. The high packet consists the most significant four bits of every position whereas the low packet contains the least significant four bits. The advantage of this protocol is to start the servos as soon as a command is received. So while the Control Program receives the high packet, all servos will

Table 1. Communication Protocols

Protocol	Command Structure
Absolute Position	FF01198282828282828282828282E5 FF: Header 01: SyncWrite command 19: Speed 82: Position E5: Checksum = lower byte of ~(01+19+82+82+82+82+82+82+82+82+82+82)
Sliding Resolution	FF0101198888888888883C (First frame) FF010219222222222239 (Second frame) FF: Header 01: SyncWrite command 01: Highbytes or 02: Lowbytes 19: Speed 88: combination of 2 Highbytes 22: Combination of 2 Lowbytes 3C: Checksum = lower byte of ~(01+01+19+88+88+88+88+88)
Difference	FF011922222222223B FF: Header 01: SyncWrite command 19: Speed 22: Each 4-bit represents the difference between the desired and the previous positions. 1 to 7 represent increment and 8 to F represent decrement 3B: Checksum = lower byte of ~(01+19+22+22+22+22+22)

start moving towards their target positions then the final target positions will be determined once the low packet has been received.

In Difference protocol, the Control Program receives two types of instruction packets, Absolute and Relative. The Absolute packet is similar to the one described in the Absolute Position protocol. Then the Relative packet will be sent with only four bits per position which encodes the differences between current and new target positions.

3.3 Data Structure

We tested two different queue data structures in our embedded system which are circular queue and regular queue. For a circular queue, it provides few advantages over regular queue. It only uses a single fixed-size data structure [3] and its array can be accessed at both end. It avoids memory wastage and let the program handles data streaming efficiently [4]. But data in circular queue can be replaced unintentionally if the algorithms in the program is not designed properly. On the other hand, regular queue is easier to implement compare to circular queue because the memory only can be accessed linearly but its downside is more memory has to be allocated if large queue size is needed.

In the Control Program, these interrupt driven queue data structures are adapted into IRS for both transmission and receiving of USART0 and USART1. Data Register Empty Interrupt, UDREI0 and UDREI1 are selected in the Control Program to trigger ISRs on USART0 and USART1 [6],[7]. On the other hand, the Control Program measures the latencies and jitters in every send-receive cycle of CM-2+. Because the UDREIs are implemented in the circular solution, it will introduce latency when ISRs occur. The experimental results are discussed in section 4.

3.4 PID Controller and Optimization

PID (proportional-integral-derivative) controller is the most popular industrial feedback control algorithm. Implementation of the PID controller in our Control Program is based on feedback about the latencies and jitters of the torque responses which will modify the context switching frequency. We need to ensure that our Preemptive kernel is robust enough so that the host program from its Motion Editor can modify its context switching frequency in real-time. To change the context switching frequency, all ISRs are disabled with *cli()* before the assignment of new frequency and they are enabled with *sei()* after its new frequency is updated. Fig. 5 shows the theory of the PID controller.

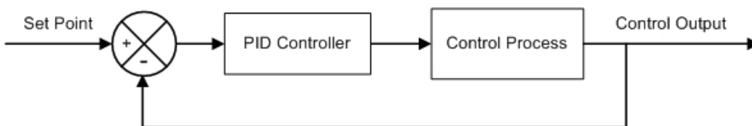

Fig. 5. Block diagram of PID controller

In the PID control loop, the control output (co) is calculated based on the following equations:

$$e(t) = sp - pv \tag{1}$$

$$P_{out} = K_P e(t) \tag{2}$$

$$I_{out} = K_I[e(t) - e(t-1)] \tag{3}$$

$$D_{out} = K_D \sum_{t=0}^{n} (e(t)) \tag{4}$$

$$co = P_{out} + I_{out} + D_{out} \tag{5}$$

Firstly, the error, $e(t)$ is calculated in equation (1) by subtraction of the set point, sp and the process variable, pv which is the latency of current run, t where $t = 0, 1, 2...n$. Then the proportional, integral, and derivative terms of output are determined by equations (2), (3) and (4). Finally, all PID outputs are summed to calculate co as equation (5). K_P, K_I and K_D are the constants gain of proportional, integral and derivative respectively.

Then we extended the PID controller to minimize the latency with the Optimisation Quality Function which returns its control output in the factor of 75% latency and 25% jitter are implemented to control the context switching frequency. The optimization phase is similar to the general PID controller, except it will find the context switching time which has the lowest control output from the quality function. It will be adjusted to reach the desired set point based on the latencies and jitters measurements.

4 Experimental Results

In order to optimise our embedded system, we measured the latencies and jitters of 100 runs to analyse the efficiency of those different implementation discussed in section 3. Baud rate of 1 Mbps is used in USART0 for serial communication. In these experiments, we sent 58 bytes of instruction packet to the Control Program; then we compare the averages of latencies and jitters in different queue data structures.

Table 2. Results of circular and regular queue data structures

	Queue Data Structures	
	Regular	Circular
Latency (μs)	581	1103
Jitter (μs)	2.0	1.4

Based on the experimental results in Table 2, generally both queue data structures produce acceptable latency and jitter. Circular queue needs 1103μs compare to 581μs in regular queue which perform better in this implementation where it takes 10μs to send a byte at 1 Mbps baud rate. However, circular queue provide an advantage over regular queue where it uses less memory because in an embedded system, using minimum amount of memory is as important as real-time transmission.

Performance of the PID controller and Optimisation Quality Function are tested based on three difference protocols introduced in subsection 3.2. Fig. 6 shows the test results of PID controller with 15ms as the set point (SP) of latency. Fig. 7 shows the optimization result of 100 trials at 5000Hz with $\pm$ 50 of context switching frequency in different protocols. The averages of latencies are 15320, 15105 and 15411 for Absolute Position, Difference and Sliding Resolution respectively.

Although the Difference protocol has shown better performance in both experiments but the differences are not significant. Moreover, its advantage only holds if the differences between new and current positions are less then 28 for

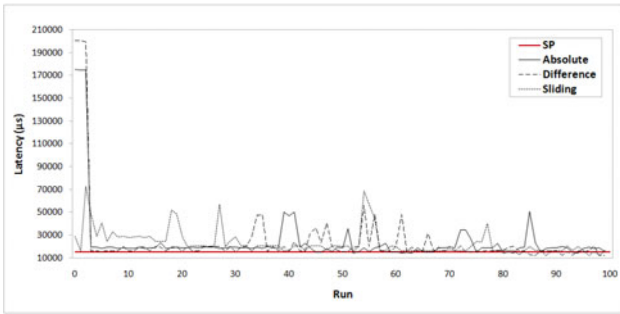

Fig. 6. Performance of PID controller in different communication protocols with $K_P = 0.1$, $K_I = 0$ and $K_D = 0.02$

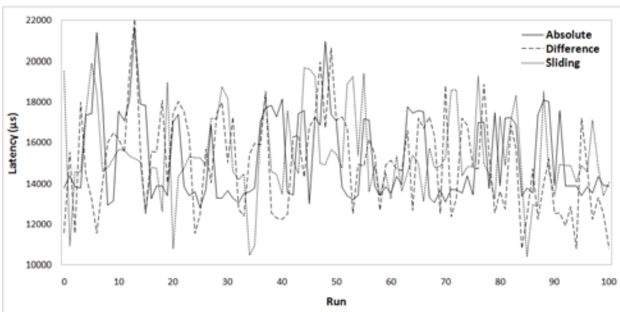

Fig. 7. Performance of Optimisation Quality Function in different communication protocol

each servo movement. In contrast, the Absolute Position protocol is simpler and easier to implement and it has most reliable performance compare to other protocols. Because the latency is fluctuated along the desired set point, therefore we implement control limit approach to acknowledge the steady state after few occurrences of control outputs fall between the control limits are perceived.

5 Conclusion

In this paper we have described the advantages and disadvantages of different implementation of queue data structures, communication protocols and PID controller. Based on the experimental results, we decided to choose the regular queue for lower average latency. On the other hand by choosing the Absolute Position protocol, the communication between the Motion Editor and Control Program becomes more reliable and easier to handle. Generally, Control Program performance has been improved by implementing the Optimisation Quality Function in program PID Controller with our selected configuration.

References

1. Robotis: User's Manual Dynamixel RX-64. Ver 1.10, p. 6 (2007)
2. Robotis: User's Manual Dynamixel AX-12, p. 3 (2006)
3. Circular Buffer, `http://en.wikipedia.org/wiki/Circular_buffer`
4. Wolf, W.H.: Computers as Components: Principles of Embedded Computing System Design. Academic Press, San Diego (2001)
5. On Time RTOS-32 Documentation, `http://www.on-time.com/rtos-32-docs/rtkernel-32/programming-manual/advanced-topics/preemptive-or-cooperative-multitasking.htm`
6. 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash, `http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf`
7. Barnett, R.H., O'Cull, L., Cox, S.: Embedded C Programming ant the Atmel AVR. Cengage Learning, Florence (2007)