2008

Developing real time applications with Java based Sun SPOT

L. Chen
University of Wollongong, chenli868@gmail.com

Phillip J. McKerrow
University of Wollongong, phillip@uow.edu.au

Q. Lu
University of Wollongong

Publication Details
This conference paper was originally published as Li, C, McKerrow, PJ and Lu, Q, Developing Real-time Applications with Java Based Sun SPOT, in Proceedings of the 2008 Australasian Conference on Robotics & Automation, Canberra, 3-5 December 2008.
Developing real time applications with Java based Sun SPOT

Abstract
Most researchers develop real-time applications with C, including programming hardware with C and using a Real Time Operating System (RTOS) that is written in C to manage the task environment. In this paper, we research a different approach by using Java to develop a real-time application. We implement an example real-time project, onboard computation of a coaxial helicopter Lama, with a Java based Sun SPOT to control the hardware and a Java RTOS (JARTOS) running on top of the Sun SPOT to manage the processes. This project enables us to answer several questions regarding to real-time system development with the Sun SPOT. At last, we conclude this paper with the experience we gather during the development.

Disciplines
Physical Sciences and Mathematics

Publication Details
This conference paper was originally published as Li, C, McKerrow, PJ and Lu, Q, Developing Real-time Applications with Java Based Sun SPOT, in Proceedings of the 2008 Australasian Conference on Robotics & Automation, Canberra, 3-5 December 2008.
Developing Real-time Applications with Java Based Sun SPOT

Li Chen  
lichen@uow.edu.au

Phillip Mc Kerrow  
phllip@uow.edu.au

Qinghua Lu  
Qinghualu83@hotmail.com

School of Computer Science and Software Engineering  
University of Wollongong  
Wollongong, NSW, 2522, Australia

Abstract

Most researchers develop real-time applications with C, including programming hardware with C and using a Real Time Operating System (RTOS) that is written in C to manage the task environment. In this paper, we research a different approach by using Java to develop a real-time application. We implement an example real-time project, onboard computation of a coaxial helicopter Lama, with a Java based Sun SPOT to control the hardware and a Java RTOS (JARTOS) running on top of the Sun SPOT to manage the processes. This project enables us to answer several questions regarding to real-time system development with the Sun SPOT. At last, we conclude this paper with the experience we gather during the development.

1 Introduction

A typical real-time application consists of two main components: controlling and interacting with embedded hardware through a micro-controller and using an RTOS to managing tasks running in the micro-controller. Most applications use C to program both components. While C can provide certain advantages in real-time application development, Java’s safe and object oriented features attract us [Chen, 2008].

1.1 The C Programming Language

C is the language used most frequently in embedded programming. Compared to assembly language, C has a more readable format and allows the compiler take control of the registers so that the programmer cannot modify the register directly.

However, C is not a safe language for several reasons. First, C uses pointers, a mechanism that allows the programmer to accidentally overwrite or corrupt crucial data. Second, C has no internal support for exceptions. All errors are handled by return values, which results in inefficient and complex error handling code. Also, C does not have a run-time environment and the programmer needs to take responsibility of memory management and watch for misbehaviour of data structures.

1.2 The Java Programming Language

Java is designed to be safe and robust. Java programs are run by a Java Virtual Machine, which handles a lot of low-level features. Hence, Java has a lot of advantages over other programming languages. However, safety has a cost in increased execution time.

First, Java uses reference instead of pointers and all memory management is handled in the JVM. The programmer can still access memory location by using object reference instead of pointers. Also, Java supports run-time data structure checking and dynamic memory allocation to data structures. Java can use assertions to verify data type consistency. Besides these, Java has full support for exception handling. Programmers can define exceptions that need to be caught by the JVM. Lastly, Java has run-time garbage collection to restore the memory that was used by objects and will never be accessed again to avoid memory leaks.

Although the above features make Java to be a safe language, there are still some problems to be handled in order to use Java in a real-time application.

Java is designed to be just-in-time compiled. This feature is important for distribution of small Java applications or applets. However, a real-time system is normally compiled once and runs on the same platform for many times. This difference means that the Java compiler will not produce optimised code for a real-time system. This difference means that the Java compiler is optimised to compile fast rather than produce code that executes fast.

Another issue with Java is its thread. Although threading mechanism in Java is easier to use and functions well, threads can result in indeterminism in concurrent programming. As the programmer does not know when a thread is running, it is may be fatal for a hard real-time task to meet time requirement.
To investigate the possibility of using Java to develop real-time applications, we start this paper by introducing the Sun SPOT and JARTOS. They are essential elements in our Java real-time development regardless of the specific applications.

And then we will present an example real-world project of converting a radio control helicopter to computer-based control with the Sun SPOT and JARTOS to illustrate the principle procedures of designing and developing a Java real-time application. With this project, we should be able to find out the answers to these questions: (a) can real-time systems be written in Java, (b) what is the advantages and disadvantages of writing in Java, (c) what is the performance, and (d) what are the problems.

Only taking all the above into account, can we determine whether the Sun SPOT can achieve the goal of developing real-time system in Java or the Sun SPOT has limitations that requires further development from Sun.

2 Sun SPOT and JARTOS

The purpose of this paper is to investigate whether a Java-based Sun SPOT is suitable for Java real-time application development. In a real-time application, the Sun SPOT can be used as a micro-controller. And one of the authors developed a Java RTOS running on top of the Sun SPOT [Lu, 2007], which can be used for task management.

2.1 Sun SPOT

Sun SPOT stands for Sun Small Programmable Object Technology. The Sun SPOT provides a typical set of hardware for real-time project development and aims to make hardware projects into Java software projects [SPOTManual, 2007]. A complete Sun SPOT kit contains a base station and an eSPOT (Figure 1).

The eSPOT contains the Main Board with a rechargeable LI-ION battery and an example of an eSPOT Daughter Board - the eDemo Board (Figure 2). The base station has an eSPOT Main Board without a battery and power is supplied by the USB connection to a host workstation. The base station serves as a radio gateway between a remote Sun SPOT and the host workstation during operation (using IEEE802.15.4 wireless communication).

2.2 JARTOS

For an RTOS, meeting the real-time requirement is not the only issue of concern. Maintaining safety when it is running is also a key requirement. To maintain safety, language features that might accidentally crash other programs or the operating system when used inappropriately needs to be eliminated from its programming language.

JARTOS stands for JAVA Real-Time Operating System. It is programmed using a safe high-level language (Java) and designed to be a time-sharing system with a cooperative scheduling approach [Lu, 2007]. Tasks in the JARTOS run in each time frame and the JARTOS schedules processes relative to a time step derived from a timer interrupt.

JARTOS uses a split architecture in its design, where the Operating system (OS) layer is completely isolated from the user applications layer (Figure 3). Adopting this design allows the application programmer to focus on the design and programming of the application problem, while all the other system management tasks are done by the OS.

Figure 1 Sun SPOT Kit

Figure 2 eSPOT Configuration

The eSPOT Main Board functions as a processor board, it contains a main processor, memory, power management circuit, IEEE802.15.4 radio transceiver and antenna, and connectors to battery and eDemo Board. The eDemo Board functions as a hardware interface board between the Sun SPOT and other devices [SPOTTheory, 2006].

A Squawk Virtual Machine runs on top of the Sun SPOT. The Squawk Virtual Machine (VM) is a Java VM primarily written in Java and designed for resource-constrained devices. Squawk is compliant with the Connected Limited Device Configuration (CLDC) 1.1 Java Micro Edition (Java ME) and runs on micro-controllers without the need for an underlying operating system, as if Java is running on the bare metal [Simon et al., 2006].
To implement the JARTOS design architecture into Sun SPOT, the whole system is divided into three Java files (Figure 4) [Lu et al., 2008].

The SunSpotApplication.java extends the format of a default Sun SPOT application and will be loaded on the start or reset of the Sun SPOT. This file represents the OS layer in the design of the JARTOS and should not be modified by programmers.

The Process.java consists a Process class that implements Runnable. This file manages user processes’ properties, including names of the process, execution frequency of the process, timeout of the process and other inter-process communication properties.

The Application.java represents the application layer in the design of the JARTOS and manages all user applications. If programmers want to add an application to JARTOS, they can write an application class that extends the Process class and add it in the StartApplication class.

3 Lama and Its On-board Electronics

Lama is a toy-size coaxial helicopter controlled by joysticks on a radio transmitter (Figure 5). In the off-the-shelf Lama set, there are a radio transmitter and the Lama helicopter. The transmitter has four channel inputs, which control the throttle, pitch, yaw and roll of Lama respectively.

We want to change the actuator control in the helicopter to computer-based control and add sensing capability to the helicopter. Also, we want to develop a robust and time efficient Java control system, so that we can research flight control and navigation of the helicopter with the aim of eventually using it in site assessment tasks in urban disaster search and rescue [Chen and McKerrow, 2007]. In order to perform the above modifications, we first need to understand Lama’s original electronics.

With information we obtained from Lama’s user manual and experiments on Lama, we can determine the block diagram of Lama as illustrated in Figure 6. When a person issues a flight command through a transmitter joystick, the transmitter will encode the analogue input into a pulse width and insert it into a pulse train. A pulse train contains eight pulses, one for each command channel. The first four pulses are used to represent the throttle, pitch, yaw, and roll input respectively, while the last four pulses are not used. The width of each pulse shows the input value for each channel and is proportional to the position of the respective joystick.

When Lama picks up these pulse trains through its radio receiver circuit, a detector demodulates the radio signal commands to a pulse train. This pulse train is passed to an embedded micro-controller, which extracts the 4 commands from the pulse train. Then the micro-controller interprets the pulse width to actual power applied on the actuators, and sends commands to rotors and motors to drive the helicopter as instructed. At the same time, the yaw gyro will provide feedback of the helicopter’s yaw velocity. The micro-controller calculates the error between the commands measured and yaw rates. It then uses a control law to calculate the velocity difference between the two rotors that is required to correct the error in the yaw [Chen, 2008].

4 Real Time System Design

Converting Lama to computer-based control involves design of six components: Graphic User Interface (GUI), wireless communication, actuator control, sensor reading, flight control and an OS to integrate them in real-time. Our Focus is on the low-level components on the embedded system - wireless, actuator, sensor, and OS - these have to be achieved before flight control is possible.
4.1 Wireless Communication

The Sun SPOT has internal support for IEEE 802.15.4 (Zigbee) wireless protocol, which provides us with realizable wireless communication between the host machine and the remote helicopter platform. Communication can be easily established by open datastream or datagram connections in both end (Figure 7).

We have design a demo application to test the wireless function of the Sun SPOT. This application enable the user to turn on LEDs in the eDemo Board by selecting checkbox in the host GUI and display the states of the switches in the eDemo Board in the GUI (Figure 8).

4.2 Actuator Control

We have two options for controlling Lama’s servos and motors. One option is for the Sun SPOT to generate a pulse train carrying control information and insert that pulse train into Lama’s original circuit, which results in minimum modification to the circuit. The other option is to control each motor and servo directly from the Sun SPOT with an R/C power amplifier, which results in full replacement of the Lama original circuit.

Pulses can be generated by the high output pin H0 in the SunSPOT. In order to generate a pulse train, we have to generate pulses and intervals between pulses. We use setHigh() function to generate pulses and setLow() function to generate intervals. And we use AT91 internal timer to generate interrupts to measure the width of the pulses and intervals. With this method, we can generate the pulse train with very good quality, of which the pulse width can be controlled in the GUI (Figure 10).

We inject this pulse train into the Lama electronics by setting (Figure 11), to control Lama’s actuators from the host.

We have measured the pulse train generated by the R/C transmitter for all joysticks with oscilloscope. In previous research [Asthana, 2007] we have found that motor controllers and servo controllers reject commands when any of pulse width, interval width, synchronization pulse width, and pulse train frame time vary more than a certain amount from the normal. For example, when frame times (synchronization pulse) are too short servos chatter. Therefore, we need to use the Sun SPOT software to generate a pulse train to match these times (Table 1) without consuming too much CPU time.

<table>
<thead>
<tr>
<th>Unit: uS</th>
<th>MIN</th>
<th>Max</th>
<th>Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Pulse Width</td>
<td>550</td>
<td>1320</td>
<td></td>
</tr>
<tr>
<td>Pulse Interval</td>
<td></td>
<td></td>
<td>550</td>
</tr>
<tr>
<td>Sync Pulse Width</td>
<td>10600</td>
<td>11400</td>
<td>11000</td>
</tr>
<tr>
<td>Time Frame Width</td>
<td>22500</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 Time Constrain for Pulse Train
4.3 Sensor Control

It is important for both the human operator and the embedded control algorithm to know the current position and attitude of the helicopter. To measure the present state of the helicopter, a nIMU (Figure 12) from MEMSENSE is used. The nIMU is a typical Inertial Measurement Unit (IMU) that contains a 3D accelerometer, a 3D gyro and a 3D magnetometer. Therefore, we can measure the linear acceleration and angular velocity vectors of the helicopter in 6 Degrees Of Freedom (DOF), and then calculate the current position and attitude of the helicopter. The outputs of nIMU are internally compensated by a temperature sensor.

Sensing the surrounding environment is also necessary to enable the helicopter to achieve autonomous flying. For instance, when hovering, we need to measure helicopter’s height above the floor. Also, we need to measure the position of potential obstacles that might cause danger to the helicopter. To measure the environment, we use Ultrasonic Range Finder SRF08 pulse wave ultrasonic sensors (Figure 13). The ultrasonic sensor can measure the distance between the helicopter and the obstacles. We can use these range measurement for calculating the current position of the helicopter and correcting the drift of the IMU.

Both nIMU and Ultrasonic sensors have built-in support for the I2C protocol. And the D2 and the D3 pins on the eDemo Board are described in the Sun SPOT documentation as supporting the I2C protocol, by acting as the SDA and SCL line. However, we found that there is no supplied software for implementing such setting. Therefore, we have to write our own I2C software.

To design a software I2C protocol, we implemented the I2C interface by realizing the following five functions that are provided in the interface.

- `isIdle()` checks if the connection is idle;
- `start()` generates I2C START bit-sequence;
- `stop()` generates I2C STOP bit-sequence;
- `receive(boolean ack)` receives data from the bus, ack is to decide whether receive further byte from slave or not;
- `transmit(byte d)` transmits data d to the bus, this function may return a boolean that indicates an acknowledgement from slave device that receives the data.

Sun SPOT D2 and D3 pins use 3V logic. So if we want to connect the ultrasonic sensor to the Sun SPOT, we need a level converter to convert the I2C bus voltage as the ultrasonic sensor only works on a 5V I2C bus. To convert I2C bus voltage, we choose the 2-channel I2C bus switch PCA9543A. This chip can connect up to three I2C buses with different voltages and the programmer can select which bus to be enabled by writing to its internal register.

To test our design and our software, we soldered a circuit board and connected a nIMU and an ultrasonic to it (Figure 14). The Li-Po battery is providing power for the nIMU.

In Figure 15, we can see that the sensor feedback has been displayed in the GUI. To simplify the GUI, we only select the latest echo from the ultrasonic sensor and the accelerometer and gyro value from the nIMU to be displayed.

4.4 RTOS Integration

The Sun SPOT is required to perform a lot of functionalities, including controlling actuators, initialising and managing sensors, and communicating with the host machine. These functionalities can be divided into five main tasks. They are radio communication, motor control, nIMU control, ultrasonic sensor control and other
supporting functions. In order to manage such a multi-task environment, we use JARTOS, a real time operating system programmed with Java that sits on top of the Squawk JVM and the Sun SPOT libraries (Figure 16).

To design a task to be managed by the RTOS, we decompose it into a set of communication processes, with each process performing part of the whole task. Then we design process scheduling to allow each process to function well under scheduler management.

**Process Design**

<table>
<thead>
<tr>
<th>Function</th>
<th>Process Number</th>
<th>Description</th>
<th>Average Time Consumed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>startRanging()</td>
<td>5</td>
<td>start and transmit address for write</td>
<td>10.80</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>transmit register number</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>ranging command and stop</td>
<td>10.80</td>
</tr>
<tr>
<td>readRangingData()</td>
<td>8</td>
<td>start and transmit address for write</td>
<td>10.80</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>transmit register number</td>
<td>9.45</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>start and transmit address for read</td>
<td>10.80</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>receive data and receive more</td>
<td>10.13</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>receive no more and stop</td>
<td>11.48</td>
</tr>
<tr>
<td>readnMUData()</td>
<td>13</td>
<td>start and transmit address for read</td>
<td>10.80</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>receive data and receive more</td>
<td>10.13</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>receive no more and stop</td>
<td>11.48</td>
</tr>
<tr>
<td>receiveData()</td>
<td>16</td>
<td>receive data from host</td>
<td>1.37 - 9.93</td>
</tr>
<tr>
<td>transmitData()</td>
<td>17</td>
<td>transmit data to host</td>
<td>12.54</td>
</tr>
<tr>
<td>generatePulseTrain()</td>
<td>18</td>
<td>generate pulse train</td>
<td>9.8 - 13.8</td>
</tr>
<tr>
<td>updatePulseTrain()</td>
<td>19</td>
<td>update pulse train variables</td>
<td>0.10</td>
</tr>
<tr>
<td>updateLED()</td>
<td>20</td>
<td>update connection LED</td>
<td>0.05</td>
</tr>
</tbody>
</table>

*This function does not generate the synchronization pulse.*

Table 2 Application Processes Table

We divide a task into several small processes by studying the timing requirements of each function that it performs. We aim to find break points where it can be separated into different processes. Previously we developed stand-alone programs that work for the individual tasks. In order to integrate the functions used in these programs into processes that can be managed by the JARTOS scheduler, we need to measure the time consumed by each program and function. With timers inserted into the code, we are able to design 16 processes to perform the five main tasks mentioned before (Table 2).

Process numbering starts from 5 in JARTOS because the first four process are system process. Each of these process are implement as user applications in JARTOS. When a process finishes, it stops its own thread and then returns to the scheduler to call the next process to run (Figure 17). (process 5 is user application 1)

**Scheduling Design**

We have to make compromise between performance and hardware limitation to integrate all processes into the scheduler in JARTOS. We decide to use 100mS time tick and make pulse train generation run as a background thread. Also, we read only the first 5 echoes from the ultrasonic sensor as they already contain the information for nearer obstacles. And because a nIMU sample contains a fixed number of 38 bytes, we still need to read all the nIMU data for a complete sample. Radio communication and update function processes can be added after the start ranging process, as there is a 65mS interval to wait for the ranging to complete (Figure 17).

The above figure illustrates all processes scheduling design except pulse train generation. Pulse train generation is not available in a 100mS time tick, as it requires running nearly every 22mS. (synchronization pulse takes 11mS, pulse train is nearly 11mS) Therefore, we add it as a background thread scheduled by JARTOS (Figure 18).
4.5 Problems with Current Design and Possible Solutions

Problems in Sensor Control
As can be seen in Figure 17, scheduling all the processes once takes 7 time ticks, which is equivalent to 700mS. That means sensing information are updated 1.43 times per second in the helicopter, which is very slow for controlling an aerial vehicle.

The main problem is the speed of the software I2C protocol. To understand the performance of our software I2C protocol, we use internal timer to measure each function (transmit(), receive(), start() and stop() ). To minimise the error, we measure the total time for executing 100 times for each function. And we found out that it takes approximately 9.82mS for transmitting a byte and 10.15mS for receiving a byte. This is really slow considering an I2C bus running in standard mode can operate at 100kHz (about 100uS for one byte).

Figure 19 Timing for WRITE and READ I2C Sequence

With further investigation into the code, we found out that the setHigh() and setLow() functions that are used to set the output pins are actually the source of the poor performance. Executing one of them takes nearly 0.3mS. The reason for them consuming so much time is that the ARM chip AT91 needs to send SPI commands to the eDemo Board every time to change the states of the pins in the eDemo Board.

However, if Sun can develop eDemo Board firmware support for the I2C bus, the performance can be improved significantly by adding a new command to send a byte or small array of bytes to reduce the frequency of sending SPI commands.

A readByte() command and a writeByte() command would require the use of the Java setHigh() and setLow() functions to handle I2C protocols. So while the transmission of a byte can be reduced from 27 (=data(8*3) + ack(1*3)) SPI commands to 1, there is no saving in the number of SPI commands for the protocol bits (3 for stop, 4 for start). Therefore, the performance improvement would be about one order of magnitude.

A read(count, buffer[]) command and a write(count, buffer[]) command require the firmware to handle the I2C protocol bits and clock. However, this has the advantage of up to 2 orders of magnitude improvement in performance. For example a read of 10 bytes would require 240 (= 10*8*3) for the data and 37 (= start(4) + stop(3) + ack(10*3 = 30)) for the I2C protocol. This is a total of 277 bit operations requiring 277 SPI commands. Thus the improvement could be as high as 35 times (277/8) and the improvement is even greater when more bytes are transferred in one transaction. This assumes that most of the time goes in the SPI command. As the I2C bus can be driven at up to 100kHz, the above data (10*8 + 10*1 + 7 = 97 cycles) could take as little as 1mS to transfer on the I2C bus. In contrast, at present each SPI command is taking 0.3 mS.

So either way there is potential for considerable improvement in performance with I2C implemented in firmware in the future release of the Sun SPOT SDK.

Problems in Actuator Control
Currently we are using a pulse train to control the on-board actuators in Lama for its simplicity of implementation. However, we notice some exceptions in the pulse train. As illustrated in Figure 20, occasionally the width of channels increases randomly. The possible reason for the exception is that the interrupt signal is sometimes blocked because a background process, such as garbage collection, takes priority for CPU time. This implies that most, if not all, the CPU time is used in generating the pulse train. R/C controller pulse train generation is a CPU intensive problem [Asthana, 2007]. Another possible reason is that the AT91 internal timer has problem with accuracy.

A read(count, buffer[]) command and a write(count, buffer[]) command require the firmware to handle the I2C protocol bits and clock. However, this has the advantage of up to 2 orders of magnitude improvement in performance. For example a read of 10 bytes would require 240 (= 10*8*3) for the data and 37 (= start(4) + stop(3) + ack(10*3 = 30)) for the I2C protocol. This is a total of 277 bit operations requiring 277 SPI commands. Thus the improvement could be as high as 35 times (277/8) and the improvement is even greater when more bytes are transferred in one transaction. This assumes that most of the time goes in the SPI command. As the I2C bus can be driven at up to 100kHz, the above data (10*8 + 10*1 + 7 = 97 cycles) could take as little as 1mS to transfer on the I2C bus. In contrast, at present each SPI command is taking 0.3 mS.

So either way there is potential for considerable improvement in performance with I2C implemented in firmware in the future release of the Sun SPOT SDK.

Problems in Actuator Control
Currently we are using a pulse train to control the on-board actuators in Lama for its simplicity of implementation. However, we notice some exceptions in the pulse train. As illustrated in Figure 20, occasionally the width of channels increases randomly. The possible reason for the exception is that the interrupt signal is sometimes blocked because a background process, such as garbage collection, takes priority for CPU time. This implies that most, if not all, the CPU time is used in generating the pulse train. R/C controller pulse train generation is a CPU intensive problem [Asthana, 2007]. Another possible reason is that the AT91 internal timer has problem with accuracy.

Therefore, we have considered changing actuator control by pulse train to parallel pulses to control the actuators directly with the Sun SPOT by each pulse controlling one motor controller or servo controller.

Parallel pulses are simple to generate using the startPulse() method and the pulse width can be accurately specified in the method. A typical actuator control pulse is 1.5mS and starting four pulses sequentially takes 2.55mS on average.
Future Potential Performance Improvement
If we can improve the performance of the I2C as we discussed before, we can reduce the time consumed by sensor control and data transfer significantly.

<table>
<thead>
<tr>
<th>Task</th>
<th>Function</th>
<th>Time Consumed (mS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Old</td>
</tr>
<tr>
<td>ultrasonic sensor control</td>
<td>startRange(n)</td>
<td>31.44</td>
</tr>
<tr>
<td></td>
<td>readRange(nulData)</td>
<td>380.71</td>
</tr>
<tr>
<td>nMU control</td>
<td>readnMU(Data)</td>
<td>390.11</td>
</tr>
</tbody>
</table>

Figure 22 Possible Improvements in the Average Time Consumed for Sensor Control

Now we look at integrating the proposed new actuator control process and new sensor control processes into the process design. In order to make sure the new actuator control process runs in every time tick, we divide the radio transmit function into transmit nMU data and transmit ultrasonic sensor data (Figure 23). With this design, we manage to execute all processes within 100mS.

Figure 23 Helicopter Control Timing in 25mS Time Tick

5 Conclusion
With the experience that we gained during the development, we are able to identify that a real-time system can be developed with Java. Java’s safe and object-oriented features provide us with convenience in both hardware programming and process design. However, uncertain behaviours of the Java Virtual Machine brought us a series of problems, mainly performance problem.

We also found that the Sun SPOT is not suitable for this real-time application with current version of the software. It does not contain firmware support for I2C, which is a frequently used protocol in real-time system design. Therefore, we have to implement I2C protocol with software. However, the multiple layer structure of the Sun SPOT wastes time doing inter-board communication, which causes the transfer rate of our software I2C protocol to be much worse than standard I2C protocol and results in serious timing issue in processes scheduling under JARTOS. Also, the generation of pulse train is influenced by interrupt handling mechanism of the JVM or timing inaccuracy of the pulse generation software in the Sun SPOT.

We have provided evidences of possible performance improvement with a proposed new firmware release and a change of control structure. Therefore, our future work is to implement a new system with solutions we proposed in Section 4.5 and evaluate its performance.

Developing real-time applications with C has been recognised as an industry standard. We wish that the work we performed during this paper could justify a more effective and safer way of real-time application development - with Java.

6 Acknowledgements
We wish to thank Sun Microsystems for providing us with Sun SPOT kit, and the Apple University Consortium (AUC) for providing us with a Macintosh Host for development and a Sun SPOT kit for this research.

References
[Simon et al., 2006] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels and Derek White, Java on the Bare Metal of Wireless Sensor Devices, Second International Conference on Virtual Execution Environments, Ottawa, Canada, June 2006.