



Chapter #4: Adding Multitasking To The J-Bot

Chapter #4: Adding Multitasking To The J-Bot

4

Up to now, the J-Bot control programs were single threaded. The programs only performed one action at a time. Background operations were handled by the Javelin's

virtual peripherals. The Javelin does not support Java's usual multitasking services but it is possible to get an effect that is sufficient for the J-Bot.

So why add the complexity of a multitasking system to the J-Bot? To make things simpler and more manageable actually. It is possible to control the J-Bot movements and handle its sensors in a single threaded program but this can get complex. Multitasking allows these operations to be handled by different cooperating tasks.

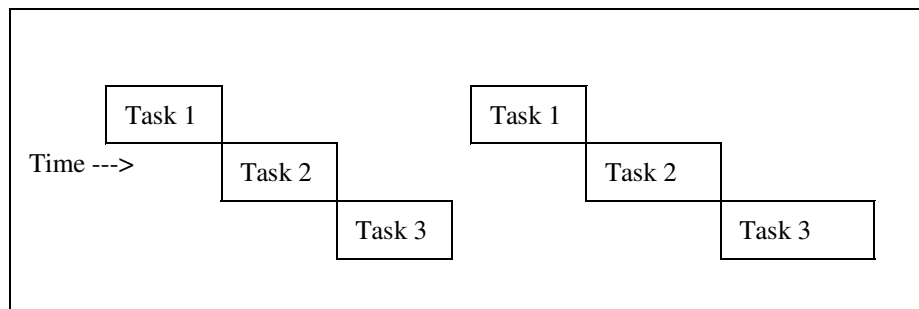
Splitting operations into different tasks provides programming independence versus a single threaded application. For example, if a series of tones need to be played while other actions are being performed then it is a simple matter to create a task that does this. Adding a task is relatively easy. Integrating this type of feature into a single threaded application is rather complex regardless of how simple the other application is.

Here's what you'll learn how to do in Chapter #4:

- Multitasking, threads, and state machines.
- How the J-Bot multitasking system works.
- Write a simple multitasking program.
- Write a multitasking version of the tone generator.
- Drive the J-Bot using the multitasking system.

Overview

Multitasking operating systems are found in most embedded applications especially robotics applications. Most desktop and server operating systems are multitasking as well. This allows different programs to be active at the same time. A closer look at a single processor system, though, reveals a CPU that only runs one task at a time. Multiple tasks get work done by sharing the processor. Each task runs for a short period of time but usually not to completion. Work not finished during this time

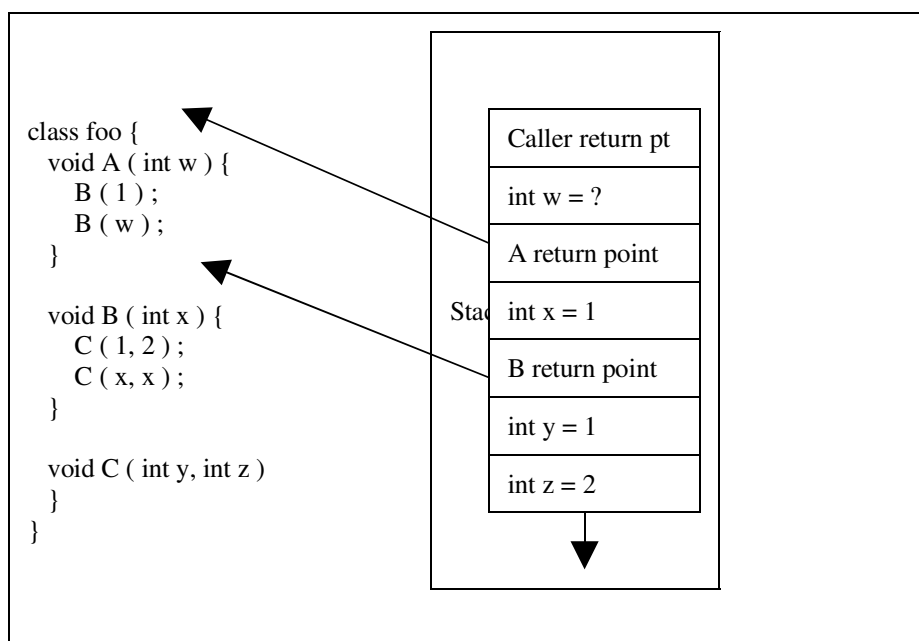


period is performed in subsequent time periods. These periods are often called time slices.

The following figure shows how this might work.

In this case, Task 1 runs for its time slice followed by Task 2 and Task 3. Task 1 then runs again and the process continues. How long each task runs and what task will run is handled by a task manager that is also known as a multitasking scheduler. We will take a look at some ways multitasking is implemented and then how the J-Bot multitasking support is implemented. More extensive coverage of multitasking systems and multitasking programming is beyond the scope of this book. Instead, we will concentrate on the services that can be run on the Javelin.

Our exploration with multitasking starts with the more complex systems most users will be used to. These employ multiple stacks; one for each active task. Java hides the stack from the programmer but it uses one to keep track of local variables for a method as well as all calling methods. For example, if method A calls method B and method B calls method C for the same object then the stack contains the local variables for method A, B and C. It also contains information that allows the calling method to continue running when a called method returns.



The figure shows how the return point and parameters for a method are stored in the stack. Not shown is the object pointer that would also be on the stack as a parameter. The stack in this example grows down. The memory at the bottom of the stack must not be used for other purposes or the data may be overwritten as method calls occur. In many operating system implementations the

heap is located below the stack with the two growing towards each other.

The key thing to note here is that each task has its own stack. Without the stack a task cannot keep track of where calling methods will continue execution when a called method is done.

In more complex systems like Microsoft Windows, tasks are more complex as well. There is the concept of a process that runs one or more threads/tasks. Each thread/task has its own stack. The process provides a way to collect together resources like open files and memory. The resources are returned to the operating system when the process terminates. The process/thread architecture allows multitasking within a process. From a task manager's point of view, there is just a big collection of tasks to run.

Switching between tasks can occur using a number of different methods. There is preemptive and non-preemptive task switching. Preemptive task switching allows an event outside of the task to cause a switch from one task to another. The time slice multitasking system mentioned earlier uses a timer to determine when a task should relinquish control. The task has no say about when this occurs and it usually does not care since it will continue running from the point that it was interrupted.

The timer event normally causes a hardware interrupt that saves the state of the currently running task, starts the task manager that then chooses the next task to run. The new task starts running until the next timer interrupt.

Other interrupts can cause a task switch as well. For example, an interrupt can occur when a serial port receives a full character. On many embedded systems there may be dozens of events and interrupts that are handled by the hardware and task manager.

There are non-preemptive actions that can cause a task switch as well. For example, a task may use an operating system service to send a message to another task. We don't care about the details here other than the semantics for this example that cause the original task to pause until the message send operation completes. Other tasks can run during this pause so the task manager performs a task switch. If this is a system that supports time slicing then the original task essentially has a shortened time slice.

It is possible to build a multitasking system that uses a cooperative, non-preemptive mode of operation exclusively. In this case, it is up to the programmer to make sure that that a task periodically yields control to let other tasks run.

Task priority is another feature supported by most multitasking systems. In this case, each task has a priority. High priority tasks run in preference to lower priority tasks. This type of feature allows a programmer to setup tasks that need to get work done as well as tasks that work on jobs that need to be done but not immediately. The lower priority tasks run when all of the higher priority tasks are waiting or have terminated.

An example where priorities can help are a communication-based embedded system. In this case, the communication task would have a high priority so incoming information can be processed immediately. Medium priority tasks would execute requests received by the communication system while low priority tasks might handle less important operations such as generating status reports.

The problem with all these neat features is their cost. There is a performance cost, a memory cost and a programming cost. Multiple tasks require enough performance to get them done plus any overhead associated with the task manager. Memory costs include support for multiple stacks and memory needed to support each task. Multitasking systems are not always efficient with memory usage. Programming is usually easier with multitasking systems but it can be more complex as well if the interaction between tasks is high. Debugging a multitasking application is more difficult than debugging a one single threaded task.

The Javelin has a limited amount of memory and its performance, although sufficient, is less than many processors that provide multitasking support. It is possible to run a multitasking system on a processor that is less capable than the Javelin but this normally uses a lower level development system that requires intimate knowledge of the hardware, the operating system and the application.

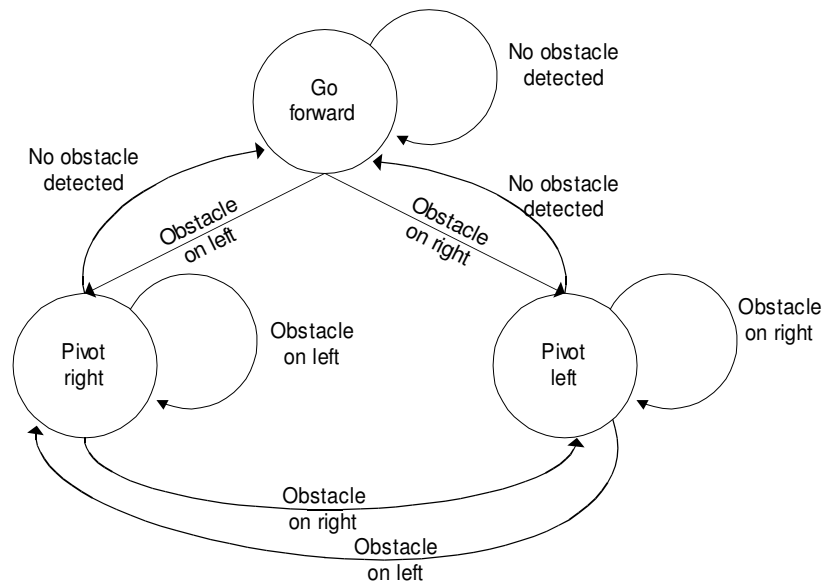
How Multitasking Works On The J-Bot

The multitasking system used here is much different than anything found in the commercial world. This is because it must operate within the limitations of the Javelin. The biggest limitation is its single stack architecture but it is still possible to build a multitasking system.

The J-Bot multitasking system can be described as a single stack, single priority, cooperative non-preemptive, state-machine multitasking system. The single stack is a limitation of the Javelin. Single priority greatly simplifies implementation and programming. The cooperative non-preemptive architecture requires programs that relinquish control periodically so they do not monopolize the processing time. The non-preemptive aspect is also a limitation of the Javelin. Virtual peripherals operate in a

preemptive fashion but virtual peripherals are limited to those built into the Javelin.

Finally there is the state machine aspect of the multitasking system. It turns out to be a useful paradigm because robotic applications are often more readily described as a state machine. For example, the following diagram shows a simple state machine for a robot that can move forward and pivot left or right. It has a sensor that detects whether there is an obstacle in front to the left or right. It is assumed that if the obstacle is directly in front then it will be detected as either an obstacle to the left or right.



There are three states shown in circles: moving forward, pivoting left and pivoting right. There are a number of transition arrows with conditions associated with each.

The program starts in a particular state such as moving forward. It then transitions to a new state based on the current conditions. It can stay in the same state as indicated by the arc that points back to the same state.

This program is relatively simple. The robot simply moves forward until an obstacle is detected at which point it turns away from the obstacle. It keeps turning until it can go forward again. This is actually a program that will be created when sensors are introduced in subsequent chapters.

State diagrams can get much more complex. For example, a maze program may keep track of wall so it can turn down a corridor when it is detected. A backtracking program may keep track of openings that were not explored to check them at a later time.

Moving back to those locations requires a state machine that is more complex than the one just presented.

There are programs that can convert graphical state machine drawings to program code but we will not be using those. Instead, programs will be written to match any state machine drawings in this book. It is also possible to write the programs without making state machine drawings.

The typical implementation of a state machine is a method that performs actions for the current state and returns the next state. The following is a sample state machine class for the prior state machine diagram.

```
class RobotStateMachine {
    static final int forward = 1 ;
    static final int pivotLeft = 2 ;
    static final int pivotRight = 3 ;

    int state = forward ;

    public void nextState ( int state ) {
        this.state = state ;
    }

    public void execute () {
        switch ( state ) {
            case forward:
                moveForward () ;
                if ( obstacleToLeft () )
                    nextState ( pivotRight ) ;
                else if ( obstacleToRight () )
                    nextState ( pivotLeft ) ;

                break ;

            case pivotLeft:
                pivotLeft () ;
                if ( obstacleToRight () )
                    nextState ( pivotRight ) ;
                else if ( ! obstacleToLeft () )
                    nextState ( forward ) ;

                break ;

            case pivotRight:
                pivotRight () ;
                if ( obstacleToLeft () )
                    nextState ( pivotRight ) ;
                else if ( ! obstacleToRight () )
                    nextState ( forward ) ;

                break ;
        }
        return state ;
    }

    // other methods for detection and movement would go here
}
```

Only the nextState and execute methods are shown. The other methods used by it are not shown although a comment indicates where they would be listed.

The execute method has no parameter but the state variable is part of the object so it is always available to object methods like execute. The method will perform an action and it then checks whether the state should change. The nextState method is used in this example. In theory it is possible to eliminate nextState method but it provides a debugging mechanism. It also provides a controlled way for one task to change another task's state.

Tasks used in our multitasking system will have a structure similar to this. The task manager would call the execute method for each task in a round robin fashion.

While the execute method can be rather large, each individual state should minimize the amount of time it uses. This allows other tasks to use the rest of the time available for execution.

Activity #1: JBot Class - Adding Multitasking

If this were an introduction to building a multitasking state machine system then the class definitions would be presented in an incremental fashion with features added in each step. Sorry, but we are going to jump right into the details of the full blown system otherwise this chapter and book would be significantly larger.

Don't worry if the architecture or the class definitions get too complicated. It helps to know how they work but it is not necessary to use them. Make sure you understand how to use them as illustrated in the test programs presented in this chapter. In general, the creation of task classes is relatively simple. The execute method is all that is needed.

FYI

The classes, such as the Event and Task class, that are in the stamp.util.os package will be installed with the latest version of the Javelin software. They do not need to be entered as the other application programs in this book.

Event				
	Semaphore			
	Task			
		InterruptTask		
			TimerTask	
				TaskToneGenerator
		CallableTask		
		WatchHeapTask		
ShowStatus				
	TaskStatus			
	SemaphoreStatus			

Multitasking Class Hierarchy

The following is the Event class definition. It is the basis for tasks and semaphores as noted in the class hierarchy shown above.

```

/*
 * Co-operative, multitasking state machine operating system
 * <p>
 * Tasks are a subclass of Event so a task can be resumed
 * by causing an Event.
 *
 * @author Parallax, Inc.
 *
 * @version 1.0 8/21/02
 */

package stamp.util.os;

import stamp.util.*;
import stamp.core.*;

/**
 * Interface for indicating an event occurred.
 */

public class Event {
    static final public Event nullEvent = new Event () ;

    /**
     * Create null event. Use Event.nullEvent instead.
     */
    protected Event () {
    }

    /**
     * Check if event reference is null.
     *
     * @param event event to check
     *
     * @returns nullEvent if null, otherwise event
     */
    static public Event checkEvent ( Event event ) {
        return ( event == null ) ? nullEvent : event ;
    }
}

```



```

/**
 * Cause an event to occur.
 */
public void notify() {
    notify(null);
}

/**
 * Cause an event to occur.
 */
public void notify( Object object ) {
}

/**
 * Get event name.
 *
 * @return name of event.
 */
public String name () {
    return "Event" ;
}
}

```

The Event class is a class with only two methods: `notify()` and `notify(object)`. It provides a simple interface for notification of an event. It is the superclass for the Task and Semaphore classes. Calling the notify method with a Task object will resume execution. This allows a task to setup an event and then suspend itself. For a Semaphore object, the notify method will release the semaphore. This simple Event interface will be used when controlling the J-Bot.

The Event objects can also have a name. By default this "Event" and most subclasses like Task provide a different name. Names are useful during debugging. They can be eliminated to conserve memory once an application has been debugged.

We start the definition of the Task class. The Task class is used as the superclass for all task objects. Each task object has a logical thread of execution based around the state machine architecture already presented.

The following is the Task class definition. It includes the definition of the Task object as well as class methods that implement the task manager. Don't read the program in detail if you are new to multitasking or Java. Instead, skip past the listing to the description of the methods involved. Come back to the listing after you are familiar with the methods available to the task manager and task objects.

```

/*
 * Co-operative, multitasking state machine operating system
 * <p>
 * This very basic operating system using round robin task scheduling.
 * The task list is a circular linked list for efficiency reasons.

```

```

*
* Note: If the TaskStatus class is not used and you need a few more
* bytes then it is possible to eliminate taskStatusList and
* nextTaskStatus along with references in the constructor.
*
* @author William Wong
*
* @version 3.0 8/23/02 eliminated execute() parameters and TaskManager
* @version 2.0 7/23/02 state variable to execute()
* @version 1.0 7/21/02
*/

package stamp.util.os;

import stamp.util.*;
import stamp.core.*;

/**
 * A task that is run whenever there are free cycles.
 * <p>
 * Note: The nice way for an external task to terminate a task is
 * to use taskToAbort.nextState(abort). The alternative is to
 * use taskToAbort.stop() but this does not allow any cleanup.
 */

public abstract class Task extends Event {
    final public static int initialState = 0 ;
    final public static int checkTimer   = -1 ; // for sleep() support
    final public static int interrupt    = -2 ; // see InterruptTask
    final public static int terminate    = -3 ;
    final public static int stopped      = -4 ;

    /**
     * Timer object to handle task sleep() requests
     */
    public Timer timer = new Timer () ;

    /**
     * Next active task in circular list
     */
    protected static Task taskStatusList = null ;

    /**
     * Linked list for taskStatusList
     */
    protected Task nextTaskStatus = null ;

    /**
     * Next active task in circular list
     */
    protected Task nextTask = null ;

    /**
     * State variable that can be used in execute()
     */
    protected int state = initialState ;

    /**
     * Next state after timeout occurs
     */
    protected int sleepState = checkTimer ;

    /**

```

```

    * Timeout values
    */
    protected int hi, lo ;

    /**
     * Setup and start task.
     */
    public Task() {
        addTask ( this ) ;

        // Add task to status list
        nextTaskStatus = taskStatusList ;
        taskStatusList = this ;
    }

    /**
     * Get task name. Default is an empty string.
     *
     * @return name of task.
     */
    public String name () {
        return "Task" ;
    }

    /**
     * See if the task is running
     *
     * @return true if task is running
     */
    public boolean running () {
        return Task.running ( this ) ;
    }

    /**
     * Get current task state
     *
     * @return task state
     */
    public int getState () {
        return state ;
    }

    /**
     * Remove task from task manager's task list if task is running.
     * This can be called by any task.
     *
     * @return true if task was removed from the list
     */
    public boolean suspend () {
        if ( nextTask != null ) {
            // Note: myManager should not be null if nextTask is not null
            return removeTask ( this ) ;
        }

        return false ;
    }

    /**
     * Suspend and set next state to stopped.
     *
     */
    public boolean stop() {
        if ( suspend () ) {

```

```

        state = stopped ;
        return true ;
    }

    return false ;
}

/**
 * Add task back into task manager's task list if task is not running
 * This can be called by any task but the task must have a manager.
 */
public boolean resume () {
    if ( nextTask == null ) {
        // Note: myManager should not be null if nextTask is not null
        addTask ( this ) ;
        return true;
    }
    return false ;
}

/**
 * Same as resume()
 */
public boolean start() {
    return resume () ;
}

/**
 * Allows the task to be used as an Event value.
 */
public void notify( Object object ) {
    resume () ;
}

/**
 * Set the state for the next execute() call.
 *
 * @param state next task state value
 */
public void nextState(int state) {
    this.state = state ;
}

/**
 * Set the state for the next execute() call.
 * Suspend the task. It must be resumed by another task.
 *
 * @param state next task state value
 */
public void sleep(int state) {
    this.state = state ;
    suspend();
}

/**
 * Setup to poll timer and continue with the next state after
 * the elapsed number of ticks occurs.
 *
 * @param nextState state to enter when the timeout occurs
 * @param hi hi timeout value, see Timer.timeout(hi,lo) for details
 * @param lo lo timeout value
 */

```

```

public void sleep ( int nextState, int hi, int lo ) {
    this.hi = hi ;
    this.lo = lo ;
    timer.mark () ;
    sleepState = nextState ;
    state = checkTimer ;
}

/**
 * Setup to poll timer and continue with the next state after
 * the elapsed number of ticks occurs.
 *
 * @param nextState state to enter when the timeout occurs
 * @param msec timeout value in milliseconds
 */
public void sleep ( int nextState, int timeMS ) {
    sleep ( nextState, timeMS/569, (timeMS%569)*119) ;
}

/**
 * Setup to poll timer and continue with the next state after
 * the elapsed number of ticks occurs.
 *
 * @param nextState state to enter when the timeout occurs
 * @param sec timeout value in seconds
 */
public void sleepSec ( int nextState, int timeS ) {
    sleep ( nextState, (timeS*18)/10, 0);
}

/**
 * Perform a period task. All subclasses must implement this method.
 * An idle task is executed whenever the GUI is idle. Implementations of
 * <code>execute()</code> should not take more than 100ms to ensure that the
 * interface remains responsive.
 */
protected abstract void execute() ;

// Task manager support follows

/**
 * Class variable: current task in task list
 */
static protected Task taskList = null;

/**
 * Current active task.
 */
static protected Task currentTask = null ;

/**
 * Get currently executing task object.
 *
 * @returns currently executing task
 */
static public Task getCurrentTask () {
    return currentTask ;
}

/**
 * Add a task to task list

```

```

*
* @param aNewTask task to be added to task list
*/
public static void addTask(Task aNewTask) {
    if ( taskList == null ) {
        aNewTask.nextTask = aNewTask ;
    }
    else
    {
        aNewTask.nextTask    = taskList.nextTask ;
        taskList.nextTask = aNewTask ;
    }

    taskList = aNewTask ;
}

/**
 * Check if a task is in the task list
 *
 * @param task task to located
 *
 * @return true if task in the list
 */
public static boolean running (Task task) {
    Task checkTask ;

    if ( taskList != null ) {
        checkTask = taskList ;
        do {
            if ( checkTask == task )
                return true ;

            checkTask = checkTask.nextTask ;
        } while ( checkTask != taskList ) ;
    }

    return false ;
}

/**
 * Remove a task from the task list
 *
 * @param aNewTask task to be added to task list
 *
 * @return true if task removed from the list
 */
public static boolean removeTask(Task aTaskToRemove) {
    // handle empty list first

    if ( taskList != null ) {
        // scan list from current task and remove it if found

        Task scanTask = taskList ;

        do {
            if ( scanTask.nextTask == aTaskToRemove ) {
                // found task to remove

                if ( scanTask == aTaskToRemove ) {
                    // removing the only task in the list
                    taskList = null ;
                }
            }
            else

```

```

        {
            // check if current task being removed

            if ( taskList == aTaskToRemove ) {
                taskList = aTaskToRemove.nextTask ;
            }

            scanTask.nextTask = aTaskToRemove.nextTask ;
        }

        aTaskToRemove.nextTask = null ;
        return true ;
    }

    // check next task in the list
    scanTask = scanTask.nextTask ;
}
while ( scanTask != taskList ) ; // exit if full list scanned
}

return false ;
}

/**
 * This method will run forever executing each task in the readyTask list
 * in a round robin fashion. It returns when there are no running tasks.
 */
public static void TaskManager() {
    while ( taskList != null ) {
        // change taskList so execute() can change it via removeTask
        currentTask = taskList ;
        taskList = taskList.nextTask ;

        if (currentTask.state == checkTimer) {
            // Task is waiting for a timeout
            if (currentTask.timer.timeout(currentTask.hi,currentTask.lo)) {
                // Timeout occurred. Set next state.
                currentTask.state = currentTask.sleepState ;

                // Abort if the next state was set improperly
                if ( currentTask.state == checkTimer ) {
                    currentTask.state = terminate ;
                }
            }
        } else {
            // Just call the nextstate
            currentTask.execute () ;
        }
    }
}
}
}

```

The Task.java file contains both object and class methods. The class methods implement the task manager that manages all active tasks. The task manager is initiated using Task.TaskManager(). It should be started only after one or more tasks have been created.

The Task class is called an *abstract class* because it contains a virtual method, execute(). This means a Task object cannot be

created but the class can be used as a superclass. This makes sense because a Task object would do nothing by itself whereas a subclass of Task would define the execute method to do something useful. If you are not familiar with abstract classes then keep reading. The first example task should make this a little clearer.

Before getting into a sample task we take a look at the methods defined in the Task class. These are divided into two categories. We take a look at the class methods first. These are used to implement and support the task manager that controls all active tasks. Then we take a look at the Task object methods. These are methods that can be used within the execute() method.

The class methods include:

```
void TaskManager ()
void addTask ( Task aTask )
boolean removeTask ( Task aTask )
boolean running ( Task aTask )
```

The object methods include:

```
void execute()
void resume()
void suspend()
void start()
void stop()
void notify(Object object)
void nextState ( int state )
void sleep ( int state, int hi, int lo )
void sleep ( int state, int timeMS )
void sleepSec ( int state, int timeSec )
```

The number of methods is not large and the functionality of the system is minimal. The Task class is the superclass to a hierarchy of more useful classes including:

```
InterruptTask
TimerTask
TaskToneGenerator
CallableTask
```

There is also one support task that provides a semaphore facility, the Semaphore class, that is useful in multitasking environments. Each of these classes will be examined in more detail later in this chapter. The InterruptTask provides an asynchronous interrupt facility but not preemptive interrupt support. The TimerTask provides a framework for simple timer oriented tasks like the TaskToneGenerator. The CallableTask class proves a framework for a state machine system where calls and

returns are possible. The basic state machine support of the Task class is limited to a single level state machine.

These more advanced task classes are available for use but the facilities they provide may not be needed by all applications. In this case, one of the simpler task classes like Task can be used. Also, The InterruptTask, TimerTask, and CallableTask are abstract classes. The TaskToneGenerator class is not an abstract class so a TaskToneGenerator task object can be created.

The basic TaskManager supports any number of active tasks. Detailed operation of the TaskManager is covered in the Task Class Methods - TaskManager section coming up next. It is possible to add and remove tasks from the active task list but all active tasks are called in a round robin fashion. A task is run for its time slice when its execute method is called. It relinquishes control to the next task in the list when the method returns. In general, the execute method should not run for more than 100usec or it will monopolize the system making services provided by other tasks unresponsive. Operations that take a long time should be spread across multiple states so a little is done each time a task's execute method is called.

Classes based on the Task class are used to create task objects that can be run by the TaskManager. By default, a task is placed in the active list when it is created. It is possible to remove the task from the list before it runs if necessary. Tasks may not be in the active list for a variety of reasons. The task may have completed its function or it may be waiting for another event to occur. For example, it may be waiting on a Semaphore object that has been acquired by another task. The waiting task will be moved to the active list when the owner of the Semaphore object releases the semaphore.

The basic facilities provided by the Task class include the ability to execute a state machine using the execute method and to start, stop and sleep for a fixed time period. The sleep methods use a single timer object that is part of the base Task class. It is included because many tasks will utilize this type of service. It also greatly simplifies the syntax of the execute method since the name of an object that would otherwise provide this service is not required.

Task Class Methods - Task Manager

The Task Class methods include:

```
void TaskManager ()  
void addTask ( Task aTask )  
boolean removeTask ( Task aTask )  
boolean running ( Task aTask )
```

The TaskManager method controls all active tasks. It should be called after the initial set of tasks is created. Additional tasks can be created or removed from the active task list as a later time. The TaskManager method returns when there are no tasks in the active list. This is done to prevent the system from being idle forever. Keep in mind that the TaskManager implements a non-preemptive multitasking system. Once the task list is empty the only thing running is the TaskManager method. There is nothing that will add a new task to the active list so the TaskManager either returns or remains idle forever. The former action is how the TaskManager works.

The TaskManager maintains the active task list as a circular list of tasks. This means that each task object has a reference to the next task in the list. If the list contains one task then this reference is to the task itself. The circular list requires careful implementation but it is more efficient for execution because the next active task is always available. Some multitasking systems implement the active list as a non-cyclic list but this means the task manager must check for the end of the list and then start back at the beginning when the end of the list is reached. This TaskManager does not have to perform such a check.

The addTask and removeTask methods are used to add or remote a task object from the active list. These methods are not usually called directly. Instead, the task object methods like start and stop are used. These methods utilize the addTask and remoteTask methods.

The same is true for the running method although this can be useful for applications that keep track of tasks explicitly. It is also possible to use the matching task method which in turn calls the class method.

Task Object Methods

The Task object methods include:

```
void execute()
void resume()
void suspend()
void start()
void stop()
void notify(Object object)
void nextState ( int state )
void sleep ( int state, int hi, int lo )
void sleep ( int state, int timeMS )
void sleepSec ( int state, int timeSec )
```

There is also a variable named *state* that is part of the Task object definition. This means that all Task-based classes will have this variable available to the object methods.

The *execute* method is called by the TaskManager each time the task object becomes the active object. The task object's *state* variable is normally used in a switch statement in the *execute* method. The *execute* method should be written to minimize the time it takes to execute. A good rule of thumb is to keep the time under 100usecs.

The *state* variable stores an integer but using numbers as state names gets very confusing. It is better to provide a symbolic name for each state. This is typically done using constant definitions at the start of a task class definition. For example:

```
class task1 extends Task {
    static final int newState = 1 ;
    static final int anotherState = 2 ;

    public void execute () {
        switch ( state ) {
            case initialState:
                // perform action for state
                break;

            case newState:
                // perform action for state
                break;

            case anotherState:
                // perform action for state
                break;

            default:
                stop ();
                break;
        }
    }
}
```

The prefix *static final int* indicates the integer definitions a constant. The state numbers should be unique. Although it is easiest to use sequential numbers there is no requirement. Also, once named constants are used, the actual value of the constant is irrelevant. State constant values should only be positive numbers. Zero and negative numbers are reserved for internal TaskManager support.

FYI It is good programming practice to stop a task if an unknown state is set. The default switch clause performs this function. It may also be useful to perform some action to indicate the problem such as printing a message using <code>System.out.println</code> or by

sounding a tone or blinking and LED. None of these actions is shown in the sample code above.

The start and stop methods are used to add or remove a task from the TaskManager's active list. A task is started when it is created so the start method is only needed when the task is inactive. It is normally called by another task. The stop method can be called by either the task itself or another task wishing to stop a task. The operating system is very limited so there are no facilities to protect one task from another. This is typical on small embedded systems like the J-Bot.

The suspend and resume methods are the same as the start and stop methods. They match the syntax and general semantics used with standard Java threads. The notify method is required because the superclass is the abstract Event class. The notify method is the same as the resume method. It is needed because the superclass explicitly named the method.

The sleep methods all require a state to enter after the sleep period has expired and the amount of time to sleep. The methods match the facilities of the Timer object including msec and second timeouts. Essentially the sleep methods save the timeout value and perform a Timer.mark method call using the task's timer object. The TaskManager then checks the task in the usual round robin fashion until the timeout occurs in which case the state variable is changed to the value passed in the sleep method call and the execute method will be called. The task actually remains in the active list so the TaskManager will not exit if all the tasks are waiting on the timer object via a sleep method call.

Finally there is the nextState method. Use this instead of setting the state variable directly. This allows debugging or trace facilities to be added to a task and it also allows subsequent task class subclassing that might need to know about state changes.

First Multitasking Program

The first multitasking program will show how the control flow operates among tasks. This example uses multiple tasks that are the same class but have different data. Typically tasks are unique requiring their own class definitions. This requires multiple class definition files.

The following MultitaskingTest1 class contains both a static main method that starts the program as well as a task definition. Task objects are created using this definition.

```
import stamp.core.*;
```

```
import stamp.util.os.*;

/**
 * Multitasking Test 1
 *
 * @version 1.0 8/15/02
 *
 * @author Parallax Inc.
 */

public class MutlitraskingTest1 extends Task {
    // execute() states
    final static int state1 = 1 ;
    final static int state2 = 2 ;

    String name ;

    MutlitraskingTest1 ( String name ) {
        this.name = name ;
    }

    /**
     * Get task name.
     *
     * @return name of task.
     */
    public String name () {
        return name ;
    }

    public void show ( String text ) {
        System.out.print ( name ) ;
        System.out.print ( ": " ) ;
        System.out.println ( text ) ;
    }

    protected void execute () {
        switch ( state ) {
            case initialState:
                show ( "Initial state" ) ;
                nextState ( state1 ) ;
                break ;

            case state1:
                show ( "State 1" ) ;
                nextState ( state2 ) ;
                break ;

            case state2:
                show ( "State 2" ) ;
                stop () ;
                break ;

            default: // terminate should be the default to catch bad states
                stop () ;
                break;
        }
    }

    public static void main() {
        new MutlitraskingTest1 ( "Task 1" ) ;
        new MutlitraskingTest1 ( "Task 2" ) ;
        new MutlitraskingTest1 ( "Task 3" ) ;
    }
}
```

```
        Task.TaskManager () ;  
        System.out.println ( "All done" ) ;  
    }  
}
```

The constructor takes a single String parameter that is the name of the task. The task prints out the current state and includes the task name as part of the output. Each task steps through three states including the initialState that the task starts in. The task terminates after printing "State 2".

The main method is where the tasks are created and the task manager runs the tasks. Three tasks are created at the start of the main method. The constructor parameters helps differentiate each one. A reference to the task is only necessary if the task will be manipulated by another task since the constructor adds the task to the task manager's active list.

The tasks in the active list are run when the Task.TaskManager method is called. The execute method of each task will be called in a round robin fashion until all the tasks terminate. In this case, each task will print a message like the following in the message window.

Task 1: Initial state

The output from each task will be interleaved because each task exits the execute method after generating a line of output.

The TaskManager method returns when all the tasks have stopped which occurs in state2. Note that the order of the actions in state2 is irrelevant since the execute method always runs to completion. The stop method simply removes the calling task from the active list. This has no affect on the task until the execute method returns. Still, it is good programming practice to make the last executed statement in an execute method state to set the next state unless it remains the same.

InterruptTask Class

The basic Task class definition is sufficient for many multitasking applications but there are many ways to extend the Task class to make it easier to create new tasks. One type of operation that is useful is the ability to interrupt another task so it can handle a new request before it may have completed its current work. The InterruptTask provides this type of service and it is shown in the following listing:

```
/**
```

```

* This type of task can be interrupted using the Interrupt() method.
*
* It changes the state of the task to Interrupt and saves the current
* state. The task can then resume using lastState.
*
* public void execute (int state) {
*     switch ( state ) {
*         case interrupt:
*             // do something here with interruptValue
*
*             nextState ( lastState ) ;
*             break ;
*
* @author William Wong
*
* @version 3.0 8/23/02 execute() parameters removed
* @version 2.0 7/23/02 state variable to execute()
* @version 1.0 7/21/02
*/

package stamp.util.os;

import stamp.util.*;

public abstract class InterruptTask extends Task {
    public int interruptState = terminate ;
    public int interruptValue = 0 ;

    /**
     * Used to interrupt a task. If multiple interrupts are made before the
     * task enters the interrupt state then only the first interrupt will be
     * recognized. All others will be ignored.
     *
     * @param interruptValue value given to task to determine the cause of the
     * interrupt
     *
     * @returns true if the interrupt will be processed
     */
    public boolean interrupt ( int interruptValue ) {
        if ( state != interrupt ) {
            this.interruptValue = interruptValue ;
            interruptState = state ;
            state = interrupt ;
            return true ;
        } else
            return false ;
    }

    /**
     * Resume state prior to interrupt. This should only be called from
     * the interrupt state.
     */
    public void resumeInterrupt () {
        state = interruptState ;
    }

    /**
     * Used to interrupt a task. If multiple interrupts are made before the
     * task enters the interrupt state then only the first interrupt will be
     * recognized. All others will be ignored.
     *
     * @returns true if the interrupt will be processed
     */

```

```

    public boolean interrupt () {
        return interrupt(0);
    }
}

```

The InterruptTask adds three methods and two object variables. The methods include two interrupt methods and a resumeInterrupt method. The first interrupt method takes a single integer parameter that is saved in the interruptValue variable that can be queried by the task when it enters the interrupt state. The task's execute method needs to handle the interrupt state accordingly.

This will normally do one of two things. The first is to examine the interruptValue and perform some operation based upon this information. The second is to determine what to do next. It can call the resumeInterrupt method that will change the state back to the value it had prior to the interrupt. It can also set a new state in which case the task can be interrupted again.

The InterruptTask can only be interrupted by one source at a time. The interrupt methods will return false if the task is already interrupted.

The following is a variation on the first multitasking test program except that the task class extends InterruptTask instead of Task. State 1 is also changed so that another task is interrupted. The task to interrupt is passed as a parameter to the task constructor. Nothing occurs if the parameter is null.

```

import stamp.core.*;
import stamp.util.os.*;

/**
 * Multitasking Test 2
 *
 * @version 1.0 8/15/02
 *
 * @author Parallax Inc.
 */

public class MutlitaskingTest2 extends InterruptTask {
    // execute() states
    final static int state1 = 1 ;
    final static int state2 = 2 ;

    String name ;
    InterruptTask taskToInterrupt ;

    MutlitaskingTest2 ( String name, InterruptTask taskToInterrupt ) {
        this.name = name ;
        this.taskToInterrupt = taskToInterrupt ;
    }

    public void show ( String text ) {

```



```

        System.out.print ( name ) ;
        System.out.print ( ": " ) ;
        System.out.println ( text ) ;
    }

    protected void execute () {
        switch ( state ) {
            case initialState:
                show ( "Initial state" ) ;
                nextState ( statel ) ;
                break ;

            case statel:
                show ( "State 1" ) ;
                nextState ( state2 ) ;
                if ( taskToInterrupt != null )
                    if ( taskToInterrupt.interrupt () )
                        show ( "Task interrupted" ) ;
                    else
                        show ( "Task already interrupted" ) ;
                break ;

            case state2:
                show ( "State 2" ) ;
                stop () ;
                break ;

            case interrupt:
                show ( "Interrupted" ) ;
                resumeInterrupt () ;
                break ;

            default:    // terminate should be the default to catch bad states
                stop () ;
                break;
        }
    }

    public static void main() {
        MutlitasakingTest2 task1 = new MutlitasakingTest2 ( "Task 1", null ) ;
        new MutlitasakingTest2 ( "Task 2", task1 ) ;
        new MutlitasakingTest2 ( "Task 3", task1 ) ;

        Task.TaskManager () ;
        System.out.println ( "All done" ) ;
    }
}

```

The first task is saved in task1 so it can be passed as a parameter to the other two tasks. Each of the other two tasks will try to interrupt task1 in statel. The first task will be the only one to use the interrupt state and the interruptValue variable is not used in this exercise. The first task will simply resume its work after printing an indication that it was interrupted. The other two tasks will indicate whether they were able to interrupt task1.

The interrupt support is limited but useful. Normally this type of task will be setup to handle one or more types of interrupts. Multiple interrupt types can be handled using the parameterized version of the interrupt method. There is no indication what other task caused the interrupt but this type of facility can be added if it is needed. In general, the reason for having the interrupt support is well defined in an application and the interrupt is either from a single source or the source does not matter.

TimerTask Class

The TimerTask is an extension of the InterruptTask. We will take a look at its construction in this section but leave the example program to the next section about the TaskToneGenerator that uses the TimerTask. The following is the listing for the TimerTask class.

```
/**
 * Task for simplified timer-based operations
 * <p>
 * This type of task is designed to be started when the timer is
 * started. The <code>handleTimeout</code> method is called
 * when the timeout occurs. The timer can be restarted or the
 * task can be terminated.
 *<p>
 * A typical use is a tone generator that will be used with the
 * TaskManager. This allows other tasks to run while the timer
 * is running. The <code>handleTimeout</code> method can also
 * be defined. By default, calling the <code>interrupt</code>
 * method will terminate the timer.
 *
 * @version 1.0 7/23/02
 * @author William Wong
 */

package stamp.util.os;

public abstract class TimerTask extends InterruptTask {
    final static int timeout = 1 ;

    /**
     * Called when interrupt invoked.
     * Redefine if subclass will handle this state.
     *
     * @return <code>true</code> if resume, <code>false</code> if terminate
     */
    public boolean handleInterrupt ( int interruptValue ) {
        return false ;
    }

    /**
     * Called when timeout occurs.
     * Redefine if subclass will handle this state.
     *
     * @return <code>true</code> if resume, <code>false</code> if terminate
     */
    public abstract boolean handleTimeout () ;
}
```

```

/**
 * Check if timer is running
 *
 * @return <code>true</code> if timer is running
 */
public boolean timerRunning () {
    return state == checkTimer ;
}

/**
 * Task execution routine called by TaskManager
 *
 * @param state current task state
 */
public void execute () {
    switch ( state ) {
        case interrupt:
            if ( handleInterrupt ( interruptValue ))
                resumeInterrupt () ;
            else
                nextState ( terminate ) ;
            break ;

        case timeout:
            if ( handleTimeout ())
                break ;      // continue if timeout handled

            // otherwise fall through to stop() the task

        // case initialState:
        // case terminate:
        default:
            stop () ;
            break ;
    }
}

/**
 * Start timer and call handleTimeout() on timeout.
 *
 * @param hi hi timeout value, see Timer.timeout(hi,lo) for details
 * @param lo lo timeout value
 */
public void sleep ( int hi, int lo ) {
    sleep ( timeout, hi, lo ) ;
    resume () ;
}

/**
 * Start timer and call handleTimeout() on timeout.
 *
 * @param msec timeout value in milliseconds
 */
public void sleep ( int timeMS ) {
    // use super or you get the local sleep method
    super.sleep ( timeout, timeMS ) ;
    resume () ;
}

/**
 * Start timer and call handleTimeout() on timeout.
 *

```

```
* @param sec timeout value in seconds
*/
public void sleepSec ( int timeS ) {
    sleepSec ( timeout, timeS );
    resume () ;
}
}
```

The TimerTask is designed to support simple time-based tasks such as a tone generator. It is one way of using a task's timer object instead of using CPU.delay. Using CPU.delay is not a good idea in a multitasking environment because everything except the background virtual peripherals comes to a complete halt.

What the TimerTask does is allow an operation to start such as a tone generated using a PWM virtual peripheral and then easily determine when to turn off the PWM. This leads to some interesting source code whose use will be explained here.

First take a look at the execute method. Notice that there is a default clause in the switch statement but not one for initialState (there is a note in the comment though). In this case the default clause will be the first state that the task executes and it stops the task. That is because the normal state for a TimerTask is stopped. It is started when there is work to do. The task then does some setup and waits until some amount of time is elapsed. This type of work will be done in the TaskToneGenerator class based on the TimerTask class. The TimerTask class is an abstract class and it needs to be subclassed as in the TaskToneGenerator class.

The execute switch statement does handle two states: timeout and interrupt. This is where a TimerTask subclass will differ from a conventional task. In the TimerTask case the subclass must define the handleTimeout method, not the execute method.

The typical operation of a TimerTask is to stop and wait for it to be started. Normally a method is defined in the subclass so the task object can be started such as playTone in the TaskToneGenerator class. This method does the setup and then calls one of the TimerTask sleep methods. These methods setup a timeout value and start the task. The handleTimeout method is called when the timeout occurs. If the handleTimeout method returns true then it is assumed that the task should continue executing. This normally means another action has been setup and the task will wait for another period. If not, the task will be stopped.

There are three sleep methods that are available that match the types of methods available with the Timer class. These include timeouts based on ticks, milliseconds and seconds.

The `TimerTask` is also setup to handle interrupts. The `handleInterrupt` method is called if the task is interrupted. The result of this method should be true if the task should keep running until the timeout occurs. A value of false should be returned if the task should terminate. In the last case, the method should perform any necessary cleanup. In the `TaskToneGenerator` class, this cleanup process stops the tone output.

`TimerTask` objects can be used for a variety of purposes. For example, a watchdog timer task could require a periodic interrupt from another task. If the interrupt failed to occur then the watchdog timer task would assume that something is wrong and take action accordingly. Watchdog timer tasks are common in embedded systems.

For a real world example of the `TimerTask` we turn to the `TaskToneGenerator` class.

TaskToneGenerator Class

The `TaskToneGenerator` objects should be used in a multitasking environment instead of `FREQOUT` class used in earlier chapters. A `FREQOUT` object generates a tone for a fixed duration using the `CPU.delay` method. In a multitasking environment, the task that used the `FREQOUT` object would monopolize the Javelin until the tone stopped.

The `TaskToneGenerator` is a subclass of the `TimerTask`. It is a bit more complex than some `TimerTasks` because it can generate a single tone or a series of tones specified in a tone list. The following is a list of the `TaskToneGenerator` class file.

```
/**
 * Tone generator for use with TaskManager
 * <p>
 * Generates tones for a specified period using the TaskTimer support.
 * It does not use CPU.delay like FREQOUT does.
 *
 * @version 1.0 7/23/02
 * @author Parallax, Inc.
 */

package stamp.util.os;

import stamp.core.*;

public class TaskToneGenerator extends TimerTask {
    protected PWM pwm ;
    protected int pin ;
    protected int tones[] ;
    protected int tonesIndex ;
```

```

protected boolean pwmRunning = false ;

final static int endTone = 1 ; // state when done done

final static int noMoreTones = -1 ;

/**
 * Setup the tone generator output.
 *
 * @param pin set output pin such as CPU.pin10
 */
public TaskToneGenerator ( int pin ) {
    this.pin = pin ;
    pwm = new PWM ( pin ) ;
}

/**
 * Get task name.
 *
 * @return name of task.
 */
public String name () {
    return "TaskToneGenerator" ;
}

/**
 * Set output frequency
 *
 * @param frequency frequency in hertz (0 - 32k)
 */
protected void setFrequency ( int frequency ) {
    if ( frequency == 0 )
        frequency = 1 ;

    int halfCycleTime = 12000 / frequency ;

    pwm.update ( halfCycleTime, halfCycleTime ) ;
}

/**
 * Play a tone for a fixed amount of time.
 *
 * @param frequency frequency in hertz (0 - 32k)
 * @param time duration in milliseconds
 */
public void playTone (int frequency, int time) {
    tonesIndex = noMoreTones ;
    playToneNow ( frequency, time ) ;
}

/**
 * Play a set of tones. The array contains an even number of items.
 * The first value in an item is the frequency.
 * The second is the duration.
 * Does nothing if there are not at least 2 elements in the array.
 *
 * @param tones tone array with (frequency, duration) pairs
 */
public void playTones ( int tones[] ) {
    if ( tones.length >= 2 )
    {
        this.tones = tones ;
        tonesIndex = 0 ;
    }
}

```

```

        startNextTone () ;
    }
}

/* Internal routine */
protected void playToneNow ( int frequency, int time ) {
    // Setup PWM
    setFrequency ( frequency ) ;

    // Check if PWM and task already running
    if ( ! pwmRunning ) {
        pwmRunning = true ;
        pwm.start () ;    // start PWM
    }

    /* Sleep for the specified amount of time.
     * Must be after timerRunning check performed above
     * because the following sets the timerRunning status
     */
    sleep(time) ;
}

/* Internal routine */
protected void startNextTone () {
    // tonesIndex will always reference a valid pair

    playToneNow ( tones[tonesIndex], tones[tonesIndex+1] ) ;
    tonesIndex += 2 ;

    // Adjust tonesIndex if this was the last tone in the list
    if ( ( tonesIndex + 2 ) > tones.length ) {
        tonesIndex = noMoreTones ;
    }
}

/**
 * Turn off tone generation if tone is being played
 */
public void stopTone () {
    interrupt ( 0 ) ;
}

/**
 * Turn off tone generation if it is on.
 *
 * @param interruptValue ignored
 *
 * @return <code>false</code>, stop task
 */
public boolean handleInterrupt ( int interruptValue ) {
    if ( pwmRunning ) {
        tonesIndex = noMoreTones ;    // disable on next timeout
        handleTimeout () ;            // force timeout
    }
    return false ;
}

/**
 * Task execution routine called by TaskManager
 *
 * @return <code>false</code> if done, <code>true</code> if more tones
 */

```

```
public boolean handleTimeout () {
    if ( tonesIndex == noMoreTones ) {
        // No more tones. Turn off PWM
        pwmRunning = false ;
        pwm.stop () ;           // turn off tones
        CPU.readPin (pin);
        return false ;         // stop task
    } else {
        // Get next tone
        startNextTone () ; // startup timer and next tone
        return true ;      // continue task
    }
}
```

The TaskToneGenerator class has a number of object variables including a PWM object and the output pin number. The other variables are used for generating the tone or for playing the tones specified in a tone list.

The TaskToneGenerator constructor takes the pin number, such as CPU.pin10, as an argument. This is used to setup the PWM object. Normally one TaskToneGenerator object is created since a system normally has only one speaker.

The setFrequency method is used to setup the PWM but it is a protected method. This means it can only be called by other methods, not by using the object. It is used by the object methods that handle tone generation.

The playTone and playTones are the methods that start the task. The playTone method sets the tonesIndex object variable to the constant noMoreTones so the task will terminate after playing one tone. This tone is setup using the playToneNow method.

The playTones method takes an integer array as a parameter. The array must have an even number of elements. The first of two elements is the frequency and the second is the duration. These are the same type of parameters passed to the playTone method. The task stops when all the tones in the list have been played. The playTones method first verifies there is at least one pair in the array. It then saves the array reference and sets up the tonesIndex to reference the first element of the array. The startNextTone method uses the referenced element to start playing a new tone.

The remaining methods are protected and used only within the class. The playToneNow method sets the frequency of the tone and it then starts up the PWM. The task then sleeps for the duration of the tone.

The `handleTimeout` method at the end of the listing is called when the task wakes up from its sleep. If there are no more tones to play the PWM is stopped and the pin used by the speaker is set to an input so the output for the next tone starts off properly. If there are more tones then the `startNextTone` method is called. The `handleTimeout` method returns false if there are no more tones. This will cause the task to stop. It is restarted by the next `playTone` or `playTones` call. The task continues running if the return value is true. The `startNextTone` method is what sets up the next tone and calls the `sleep` method.

The `stopTone` can be called to terminate any active `playTone` or `playTones` initiate output. The `stopTone` method uses an interrupt to do this. The task will be notified of the interrupt by a call to the `handleInterrupt` method. This method checks to see if the PWM is still running and shuts it down if it is.

The following program shows how tones can be generated.

```
import stamp.core.*;
import stamp.util.os.*;

/**
 * Multitasking Test 3 - Tone generation
 *
 * @version 1.0 8/15/02
 *
 * @author Parallax Inc.
 */

public class MutlitaskingTest3 extends InterruptTask {
    final static int waitForToneDone = 1 ;
    final static int waitForTonesDone = 2 ;

    // The toneList is a set of pairs of frequency and duration (msec)
    final static int toneList [] = { 120, 50, 240, 50, 360, 50 } ;

    TaskToneGenerator toneGenerator = new TaskToneGenerator ( CPU.pin10 ) ;

    protected void execute () {
        switch ( state ) {
            case initialState:
                System.out.println ( "Initial state" ) ;
                toneGenerator.playTone ( 200, 1000 ) ;
                nextState ( waitForToneDone ) ;
                break ;

            case waitForToneDone:
                if ( toneGenerator.running () ) {
                    System.out.println ( "Waiting for tone to end" ) ;
                } else {
                    toneGenerator.playTones ( toneList ) ;
                    nextState ( waitForTonesDone ) ;
                }
                break ;

            case waitForTonesDone:
                if ( toneGenerator.running () ) {
                    System.out.println ( "Waiting for tone list to end" ) ;
```

```

        } else {
            stop () ;
        }
        break ;

default:    // terminate should be the default to catch bad states
    stop () ;
    break;
    }
}

public static void main() {
    new MutlitasakingTest3 () ;

    Task.TaskManager () ;
    System.out.println ( "All done" ) ;
}
}

```

The sample program actually creates two task. One is the MultitaskingTest3 task and the other is a TaskToneGenerator. The MultitaskingTest3 task starts the toneGenerator with a single done in the initialState. It then waits for the toneGenerator to stop before starting it up again with a playTones method call.

Normally the TaskToneGenerator would handle the tone generation while other tasks did useful work. The MultitaskingTest3 task simply prints out a lot of status messages. This would not be possible if the FREQOUT object was used instead.

While the ability to print out a bunch of text may not seem important it does show how other actions can be done while sound is being generated.

CallableTask Class

Most multitasking applications can operate with single level state machines using any of the task classes as their base. The CallableTask class provides a mechanism to implement multilevel state machines. In essence, it is providing a way to call routines while remaining within the limitations of the Javelin. Remember, all calls started from within a task's execute method need to return promptly.

The need for a CallableTask is more apparent when considering more complex applications. For example, a task that sends out an arbitrary number characters via a serial port can do so without this support if the output buffer built into the serial port virtual peripheral is never filled. If it is then the call to send a character will not return until the character at the start of the buffer is sent.

The conventional state machine task can handle this nicely by having a state that checks if the buffer has room. It does not send a character until there is enough room in the buffer. The problem with a single level state machine is that this support only works for the states defined thus far. If a different part of the task must send characters too then the programmer must either setup a way to track what state should be entered after the character is sent or use the CallableTask support.

Essentially, a CallableTask uses the callState and returnState methods in a fashion similar to calling a method and returning from it. These methods are much more explicit and verbose than a typical Java call but it works within the limitations of the multitasking support. It also requires a minimal amount of overhead.

The CallableTask maintains a call stack that is actually implemented as a linked list. Each call generates a stack entry that includes the return state number and an optional parameter. The parameter reference is to an Object so it can be anything except the built-in objects such as integers. This turns out to be more flexible because the object passed as a parameter can contain an arbitrary number of object variables that can be accessed by the called state. The state can return information via the object if necessary. The parameter object class is application dependent.

The CallableTask support consists of two files. The following is the CallableTask class file. The second is the CallStackEntry. The stack entry is used by the CallableTask class to keep track of call states.

```
/**
 * This type of task supports a rudimentary state calling convention.
 *
 * The task can change states by calling other states. Those states
 * can then return to the calling state. Parameters have not been
 * implemented but a subclass could easily do so.
 *
 * @version 1.0 8/25/02
 *
 * @author Parallax, Inc.
 */

package stamp.util.os;

import stamp.util.*;

public abstract class CallableTask extends InterruptTask {
    // stack is exclusive to task
    protected CallStackEntry stack = null ;

    /**
     * Get current parameter
     *
     * @return parameter passed in latest callState
     */
}
```

```

    */
    public Object getParameter () {
        return ( stack == null ) ? null : stack.parameter ;
    }

    /**
     * Call a state.
     *
     * @param callState next state to enter
     * @param calling parameter
     * @param returnState state to enter when <code>returnState()</code> is called
     */
    public void callState ( int callState, Object parameter, int returnState ) {
        // Add a new stack entry
        stack = CallStackEntry.getEntry ( returnState, stack, parameter ) ;

        // Set new state
        nextState ( callState ) ;
    }

    /**
     * Call a state.
     *
     * @param callState next state to enter
     * @param returnState state to enter when <code>returnState()</code> is called
     */
    public void callState ( int callState, int returnState ) {
        callState ( callState, null, returnState ) ;
    }

    /**
     * Return to prior state specified in the last <code>callState</code>.
     * Terminate task if return performed without prior call.
     */
    public void returnState () {
        CallStackEntry oldEntry ;

        if ( stack == null ) {
            state = terminate ;
        } else {
            // Setup return state
            state = stack.returnState ;

            // Pop the stack
            oldEntry = stack ;
            stack = oldEntry.nextEntry ;

            // Free entry
            oldEntry.free () ;

            // Note: The following should be added if garbage collection is supported
            // stackEntry.parameter = null
        }
    }
}

```

The following is the CallableTask stack entry class, CallStackEntry.

```

/**
 * Used to implement the CallableTask stack.
 *
 * @version 1.0 8/25/02
 *
 * @author Parallax, Inc.
 */

package stamp.util.os;

import stamp.util.*;

public class CallStackEntry {
    protected CallStackEntry nextEntry ;
    protected int returnState ;
    protected Object parameter ;

    protected static CallStackEntry freeList = null ;

    protected CallStackEntry () {
    }

    /**
     * Return entry to free list
     */
    void free () {
        nextEntry = freeList ;
        freeList = this ;
    }

    /**
     * Return an unused stack entry.
     * A new entry is created if the free list is empty.
     *
     * @param returnState returnState to be saved in the new entry
     * @param nextEntry nextEntry to be saved in the new entry
     * @param parameter parameter to be saved in the new entry
     *
     * @returns unused stack entry
     */
    static CallStackEntry getEntry
        ( int returnState
        , CallStackEntry nextEntry
        , Object parameter) {
        CallStackEntry result ;

        if ( freeList == null ) {
            result = new CallStackEntry () ;
        } else {
            result = freeList ;
            freeList = result.nextEntry ;
        }

        // Add entry to stack list
        result.returnState = returnState ;
        result.nextEntry = nextEntry ;
        result.parameter = parameter ;

        return result ;
    }
}

```

We'll work backwards and start with the `CallStackEntry`. Each entry keeps track of the `returnState` and the parameter for the called state. The `nextEntry` variable does double duty. When an entry is in the `CallableTask` stack it refers to the entry of the prior call if any. It is null if this is the first entry in the stack. When a `returnState` method is called the top of stack entry is moved to the head of the `freeList` maintained by the class methods of `CallStackEntry`.

A `CallStackEntry` is not allocated by the `CallableTask` using `new`. Instead it calls the `getEntry` method that may generate a new object. It does so only if the `freeList` is empty. When the entry is no longer needed it is returned to the `freeList` using the `free` method. This is done with the Javelin because there is no garbage collection. With normal Java, the object could simply be dereferenced and it would eventually be returned to free memory via the garbage collector.

The one thing you may have noticed is there is a constructor for the `CallStackEntry` that does nothing but it is protected. This prevents another class from creating a new object. A new object can only be created using `getEntry`.

Now back to the `CallableTask` class. There are just two methods: `callState` and `returnState`. They work as a pair. The `callState` takes two, or three, parameters. The two parameters are the state to call and the state to return to. The third parameter on one version of `callState` is an argument. More on this later.

The reason `callState` needs both a call and return state is that the current state is usually not the state that the call should return from. Typically the return state will be the case statement following the case statement where the `callState` method is called.

The `callState` method works like the `nextState` method. The new state is not entered until the task's `execute` method is called again.

The `returnState` method takes no parameters and changes the state to the return state in the matching `callState`.

When the called state is entered, the parameter passed by the `callState` is available using the `getParameter` method. As mentioned earlier, the parameter that is passed must be an object of class `Object`. In theory, this should be any object but in practice the built-in values like `int` and `byte` are not suitable as a parameter. In general, a parameter class will be defined with one or more arguments. It can also be used to return values.

For example, the following could be used as a parameter to a serial routine.

```
class aCallStateParameter extends Object {
    public int bufferSize ;
    public byte buffer [ 20 ] ;
    public boolean success ;
}
```

If this parameter were used for output then the bufferSize would be set to the number of bytes in the buffer array. The success variable would be set by the serial output routine before the returnState is called. It would indicate whether the data was sent successfully.

We take a look at a sample program that actually uses a String parameter. Unlike an int or byte, the String class has the Object class as a superclass.

```
import stamp.core.*;
import stamp.util.os.*;

/**
 * Multitasking Test 4 - Callable Task test
 *
 * @version 1.0 7/25/02
 * @author Parallax, Inc.
 */

public class MutlitasakingTest4 extends CallableTask {
    final static int state1 = 1 ;
    final static int state2 = 2 ;
    final static int call1 = 10 ;
    final static int call2 = 20 ;
    final static int call2a = 21 ;

    protected void execute () {
        switch ( state ) {
            case initialState:
                System.out.println ( "Starting" ) ;

                callState ( call1, "test base", state1 ) ;
                break ;

            case state1:
                System.out.println ( "State 1" ) ;
                callState ( call2, state2 );
                break ;

            case state2:
                System.out.println ( "State 2" ) ;
                nextState ( terminate ) ;
                break ;

            // --- Call 1 -----

            case call1:
                System.out.println ( "Enter call 1" ) ;
```

```
        System.out.println ( (String) getParameter () ) ;

        System.out.println ( "Exit call 1" ) ;
        returnState () ;
        break ;

// --- Call 2 -----

case call2:
    System.out.println ( "Entering call 2" ) ;
    callState ( call1, "test 2", call2a ) ;
    break ;

case call2a:
    System.out.println ( "Done with call 2" ) ;
    returnState () ;
    break ;

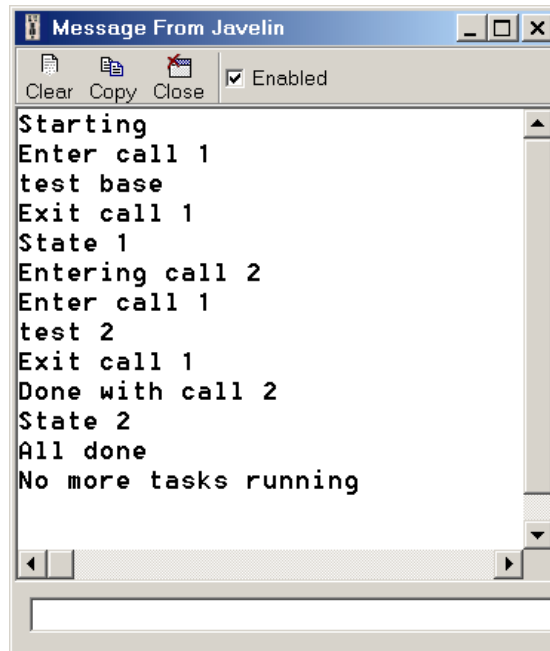
// ---- End of program ---

default:    // terminate should be the default to catch bad states
case terminate:
    System.out.println ( "All done" ) ;
    stop () ;
    break;
}
}

public static void main() {
    new MultitaskingTest4 () ;

    Task.TaskManager () ;
    System.out.println ( "No more tasks running" ) ;
}
}
```

The program creates a single `CallableTask`, actually `MultitaskingTest4`. It is similar to earlier test programs that print out the states being entered. The output from the program is shown below.



The CallableTask is not something that is always needed but it is very handy when the need arises.

Semaphore Class

Tasks can communicate with each other using shared variables. These may be static or object variables where a task has access to the object. The problem is, this type of access is unrestricted. In some applications there needs to be more control intertask (also referred to as interprocess) communication.

It is possible to implement a full range of interprocess communication methods but, given the limited memory of the Javelin, something like the Semaphore class is all that is needed.

Semaphores are useful for controlling access to resources. For example, control of the J-Bot's wheel servos may be managed by a semaphore. There could be multiple tasks vying for control. For example, a task may need to maintain control until a particular action is done such as ramping.

FYI

As with Java threads, Java has its own interprocess communication services that are not supported by the Javelin. The Semaphore class can be implemented in standard Java but other methods will normally be used.

The following is the definition for the Semaphore class.

```

/*
 * Semaphore class
 * <p>
 * Used by a Task to control a resource. It can be acquired
 * by one Task at a time. Subsequent tasks will be suspended
 * and added to the Semaphore list
 *
 * @author Parallax, Inc.
 *
 * @version 2.0 8/30/02   extends Event. Added semaphoreList
 * @version 1.1 7/24/02   acquire() uses TaskManager.currentTask
 * @version 1.0 7/24/02
 */

package stamp.util.os;

import stamp.util.*;

/**
 * Standard semaphore support for Task object.
 * It starts in the ready state.
 */

public class Semaphore extends Event {
    protected static Semaphore semaphoreList ;

    protected Semaphore nextSemaphore ;
    protected Task acquiredTask ;
    protected Task waitingTaskList ;

    /**
     * Create semaphore object.
     */
    public Semaphore () {
        nextSemaphore = semaphoreList ;
        semaphoreList = this ;
        acquiredTask = null ;
        waitingTaskList = null ;
    }

    /**
     * See if semaphore is ready
     *
     * @return true if semaphore is ready to be acquired
     */
    public boolean ready () {
        return ( acquiredTask == null ) ;
    }

    /**
     * Acquire semaphore. Sets task's next state.
     *
     * The task should exit execute() for this to take affect.
     * Use ready() if the task must conditionally acquire the semaphore.
     * Does a task.suspend() if semaphore is not ready.
     * If the semaphore is acquired then execution can continue. This allows
     * the task to release the semaphore immediately after performing
     * some action if possible. Set the nextState appropriately.
     * Remember not to do more than one matching release().
     * <p>
     * Example:
     * <code><pre>

```

```

* case 9: // this always yields control
*     semaphore.acquire(11);
*     break;
* case 10: // yields control if not acquired
*     if ( semaphore.acquire(11))
*         break;
* case 11:
*     // do something
*     semaphore.release() ;
*     nextState(12);
*     break ;
* </pre></code>
*
* @return <code>>false</code> if semaphore acquired, <code>>true</code> if
suspended
*/
public boolean acquire ( int nextState ) {
    // force next state
    Task.currentTask.nextState ( nextState ) ;

    if ( ready () ) {
        acquiredTask = Task.currentTask ;
        return false ;
    }

    // remove task from run list
    if ( ! Task.currentTask.suspend () ) {
        return true ; // task was already suspended
    }

    // add task to waiting list
    if ( waitingTaskList == null ) {
        // list was empty
        waitingTaskList = Task.currentTask ;
    } else {
        // add to end of list
        // Note: task.nextTask should be null
        Task lastTask = waitingTaskList ;

        // walk to end of list
        while ( lastTask.nextTask != null )
            lastTask = lastTask.nextTask ;

        // append task to end of list
        lastTask.nextTask = Task.currentTask ;
        Task.currentTask.nextTask = null ; // just in case
    }

    return true ;
}

/**
 * Release semaphore. Start next task if one is waiting.
 * Assumes the task that is doing the release is allowed to do so.
 */
public void release () {
    if ( waitingTaskList == null ) {
        acquiredTask = null ;
    }
    else {
        // Tasks are suspended with the nextState set.
        // Resume the first waiting task.
        acquiredTask = waitingTaskList ;
    }
}

```

```

        waitingTaskList = waitingTaskList.nextTask ;
        acquiredTask.nextTask = null ;
        acquiredTask.resume () ;
    }
}
/**
 * Release semaphore when an event occurs.
 * The Semaphore class is based on the Event class
 * that has only the notify method.
 */
public void notify (Object object) {
    release () ;
}

/**
 * Get name.
 *
 * @return name of semaphore.
 */
public String name () {
    return "Semaphore" ;
}
}

```

The a Semaphore object can be acquired by one task. If a second task tries to acquire the semaphore object while a task already has acquired the object then the second task will be added to the wait list. The first task in the wait list will acquire the Semaphore object when the task that has acquired the Semaphore object releases it.

The acquire method has a state as a parameter. The reason is that the acquire method may add the task to the wait list of the Semaphore object. The method will return true if the semaphore is acquired. This means the task can continue and use the resources protected by the semaphore. If the method returns false then the task should return from the execute method. The state parameter to the acquire method will be the state when the execute method is called next. The acquire method saves the task reference of the task that acquires the Semaphore in the acquiredTask variable.

FYI

If the task is an InterruptTask and an interrupt occurs then the next state value when the task's execute method is called will be the interrupt state. The state specified in the semaphore acquire method call will be in the next execute call. Also, the interrupt state will not be entered until the semaphore is acquired.

The release method should only be called when a task is done with the resources associated with the Semaphore object. The notify

method is included because the Event class is the superclass of the Semaphore class. The cause method releases the semaphore so programs should only be setup so this will occur after the semaphore is acquired.

The acquire method adds a task to the wait list if the Semaphore object is already acquired. This may not always be desirable. For example, a task may have other things to do and need the controlled resources if they happen to be available. In this case, the ready method can be used to query the Semaphore object's status. It returns true if the object can be acquired. This type of procedure works because the multitasking system is non-preemptive. It would not work if this was a preemptive multitasking system since a task switch might occur between the time the ready method is called and a subsequent acquired method call.

The Semaphore class also keeps a list of Semaphore objects using the semaphoreList class variable and the nextSemaphore object variables. The list is used for debugging purposes only. It is not required for general operation.

The following is a sample application that shows how the Semaphore object can be used.

```
import stamp.core.*;
import stamp.util.os.*;

/**
 * Multitasking Test 5 - Semaphore demonstration
 *
 * @version 1.0 7/15/02
 * @author William Wong
 */

public class MutlitasakingTest5 extends Task {
    static public Semaphore semaphore = new Semaphore ( ) ;

    String name ;

    MutlitasakingTest5 ( String name ) {
        this.name = name ;
    }

    public void show ( String text ) {
        System.out.print ( name ) ;
        System.out.print ( ": " ) ;
        System.out.println ( text ) ;
    }

    // execute() states
    final static int state1 = 1 ;
    final static int state2 = 2 ;
    final static int state3 = 3 ;

    protected void execute ( ) {
        switch ( state ) {
```

```

    case initialState:
        show ( "Initial state - acquire semaphore" ) ;
        semaphore.acquire ( state1 ) ;
        break ;

    case state1:
        show ( "State 1 - semaphore acquired" ) ;
        sleep(state2,1000) ;          // wait and then do state 2
        break ;

    case state2:
        show ( "State 2 - done sleeping" ) ;
        show ( "Releasing semaphore" ) ;
        semaphore.release ( ) ;
        stop ( ) ;
        break ;

    default:      // terminate should be the default to catch bad states
        stop ( ) ;
        break;
}
}

public static void main() {
    new MutlitranskingTest5 ( "Task 1" ) ;
    new MutlitranskingTest5 ( "Task 2" ) ;
    new MutlitranskingTest5 ( "Task 3" ) ;

    // Each task tries to acquire the semaphore.
    // The task then waits for a short period.
    // It then releases the semaphore and terminates.
    Task.TaskManager ( ) ;
    System.out.println ( "All done" ) ;
}
}

```

The tasks try to access the semaphore and they release it. A task terminates after it releases the Semaphore object. Eventually all tasks will return.

A Semaphore object can be used in isolation or it can be used as a superclass for a class of objects that will be controlled by the Semaphore. Either approach is valid but the latter tends to be more restrictive since it prevents the class from using another, possibly more useful, superclass.

Activity #2: Multitasking J-Bot

Activity #1 was very long but it introduced the multitasking support that will be used throughout much of the book. The sample programs were very simple and designed to highlight specific features of the multitasking system. Now we put these tools to good use and drive the J-Bot. The source code looks a bit more complex than the code used in prior single tasking examples but

keep in mind the trade off. This new approach allows other tasks to operate in the background. These might include obstacle detection, wireless communication with other J-Bots or computers, and more advanced recording and planning programs. Such features would be very difficult to build and manage without a multitasking system.

As with many of the examples in this book, we start with a class definition for an object that will do most of the work followed by a class definition with a main method that starts things off. The following BasicMultitaskingJBot class file utilizes the BasicJBot class defined earlier. It allows movements to occur without using CPU.delay method calls that would monopolize the time spent handling the movements.

```
package JBot ;

import stamp.core.*;
import stamp.util.os.*;

/**
 * MultitaskingJBot class
 * <p>
 * The program runs the J-Bot using as a task using the
 * multitasking system.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class MultitaskingJBot extends Task {
    protected JBotInterface jbot ;

    static final int checkMovement = 1 ;

    /**
     * Setup to poll jbot object
     */
    public void notify ( Object jbotInterface ) {
        jbot = (JBotInterface) jbotInterface ;
        nextState(checkMovement);
        resume () ;
    }

    /**
     * Task execute method
     */
    protected void execute () {
        switch ( state ) {
            case checkMovement:
                if ( jbot.movementDone () ) {
                    // stop task since movement is done
                    stop () ;

                    // event may restart this task with a new movement
                    jbot.causeNextEvent () ;
                }
                break;

            case initialState:
```

```

    default:          // default catches bad states
        stop () ;
        break;
    }
}
}

```

The `MultitaskingJBot` class is designed to work with a `JBotInterface` object, like `BasicJBot`, plus another task that initiates movements using a set of public methods including `move`, `pivot`, and `turn`. These methods do essentially the same thing. The `JBotInterface` object is setup so its `startEvent` refers to a `MultitaskingJBot` object. The `JBotInterface` object calls the `MultitaskingJBot`'s `notify` method with a reference to itself as the argument. This is stored in the `MultitaskingJBot`'s `jbot` object variable so it can be used while the task is running. The task is started via the `resume` method call in the `notify` method.

The `MultitaskingJBot`'s `execute` method will be called periodically by the `Task.TaskManager` that controls all tasks. The `execute` method then calls `jbot`'s `movementDone` method until it returns `true`. The task then stops itself and calls the `jbot`'s `causeNextEvent` method. This will notify the events maintained by the `jbot` that should then initiate the next movement, if necessary. This may cause the `MultitaskingJBot` object to be restarted.

The following sample application shows how this task can be used.

```

import stamp.core.*;
import stamp.util.os.* ;
import JBot.* ;

/**
 * Test MultitaskingJBot class
 * <p>
 * The program runs the J-Bot using BasicMultitaskingJBot class methods.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class MultitaskingJBotTest1 extends Task {
    JBotInterface jbot = new RampingJBot ( new MultitaskingJBot () ) ;

    protected void execute () {
        switch ( state ) {
            case initialState:
                jbot.move ( 2 ) ;
                nextState ( 1 ) ;          // need to poll for completion
                break;

            case 1:                        // polled version
                if ( jbot.movementDone () ) {
                    // Non-pollled version.
                    // This task will be suspended until the movement is done.

```



```

        jbot.pivot ( -2 ) ;
        jbot.wait ( 2 ) ;          // suspend this task until done
    }
    break;

    case 2:
        jbot.turn ( 1 ) ;          // make a short short turn
        jbot.wait ( 3 ) ;          // suspend this task until done
        break;

    case 3:
        jbot.stop();               // don't wait, background task will run
    default:                       // default catches bad states
        stop ( ) ;
        break;
    }
}

public static void main () {
    new MultitaskingJBotTest1 ( ) ;

    Task.TaskManager ( ) ;
    System.out.println ( "All done" ) ;
}
}

```

The MultitaskingJBotTest1 program creates its own task object and starts it running using the familiar Task.TaskManager call. We cheated and use numbers instead of defined constants since there are only a few states used in the execute method.

A MultitaskingJBot task object is created when the MultitaskingJBotTest1 task object is created so there are really two tasks running initially. The MultitaskingJBot object reference is stored and used only by the JBotInterface-based object. In this case it is a RampingJBot object that is a subclass of the BasicJBot class.

The first state, initialState, of the main task starts the J-Bot by calling jbot.move that in turn starts the J-Bot moving forward. The main task then sets the next state to 1 where it polls the jbot.movementDone method until it returns true. This is also occurring with the MultitaskingJBot object but the example shows how both can use the method without causing problems.

When the movementDone method returns true, the jbot.pivot method is called followed by a call to jbot.wait. The latter causes the next state to be set to 2 and the task is put to sleep. The task will be resumed when the MultitaskingJBot's execute method notices that the movementDone method returns true and the causeNextEvent is called. This in turn calls the notify method for the MultitaskingJBot object that subsequently resumes itself allowing its execute method to be called by the Task.TaskManager at a later point in time. The state value used by execute method

will be the 2 set earlier. The MultitaskingJBot is stopped at this time.

The main task repeats its process to call the jbot.turn in state 2 and stopping the servos in state 3. The jbot.wait in state 3 is needed because the jbot.stop in state 3 must

Activity #3: Multitasking J-Bot With Obstacle Checks

The multitasking J-Bot support in the prior activity handles movements without feedback. In this activity we add support for obstacle checks although we will not actually use sensors for in the sample program. That will be add in future chapters.

Now that we have multitasking support is should be relatively easy to have a task that monitors sensors to determine whether the J-Bot is close to or in contact with an obstacle. The first step is to define an abstract class that will be basis for sensor objects. Using objects with this superclass will allow a movement algorithm to be implemented for something like a random walk or maze exploration and then use this implementation with different types of sensors.

In this activity we will also come up with a simulation of a sensor. It will be based on a random number generator. This will be used with a simple collision avoidance algorithm.

The following is the abstract class definition for a sensor.

```
package JBot;

import stamp.util.os.* ;

/**
 * Basic abstract sensor class
 * <p>
 * This provides a standard method to access sensor information.
 * Normally the sensor will notify an event when a change occurs.
 * Often the event is a task that is waiting for an obstacle to be
 * detected.
 *
 * Detection is assumed to be through the front 180 degrees.
 * The forward position is 90 degrees.
 * The far left is 0 degrees. Left is 45 degrees.
 *
 * @version 1.0 8/23/02
 * @author Parallax Inc.
 */
public abstract class BaseSensor {
    protected Event event = Event.nullEvent ;

    /**
     * range: No obstacle detected
     */
}
```

```

    */
    static final int none = -1 ;

    /**
     * direction: obstacle to left
     */
    static final int left = 45 ;

    /**
     * direction: obstacle to right
     */
    static final int right = 135 ;

    /**
     * direction: obstacle in front
     */
    static final int front = 90 ;

    /**
     * Indicate whether an obstacle has been detected.
     * Normally used when polling versus using an event.
     *
     * @returns obstacle detected
     */
    public abstract boolean obstacleDetected () ;

    /**
     * Indicate initial obstacle position.
     * For simple detection systems the detection of an object
     * on the right and left will return front.
     *
     * @returns obstacle's relative direction (left, right, etc.)
     */
    public abstract int obstacleDirection () ;

    /**
     * Get the distance to an obstacle in the specified direction.
     * A value of <code>none</code> indicates no object detected.
     *
     * @param direction to get range for
     *
     * @returns distance to an obstacle for the specified direction
     */
    public abstract int obstacleDistance ( int direction ) ;

    /**
     * Set minimum event notification distance.
     * Notification will not occur until an obstacle is
     * outside of this distance. The minimum value is 0.
     *
     * @param minimumDistance minimum number of inches to detect an obstacle
     */
    public void setMinimumEventDistance () {
        /* Default case is to ignore the minimum distance
         * For example, contact oriented sensors can only detect objects
         * when they are in contact with them.
         */
    }

    // Protected classes for use by this class or subclasses

    /**
     * Set notification event

```

```

    *
    * @param event Event object to notify when a change occurs
    */
    public void setEvent ( Event event ) {
        this.event = Event.checkEvent(event) ;
    }

    /**
    * Cause event when obstacle status has changed.
    * May be called by subclass methods.
    */
    protected void notify () {
        notify (null) ;
    }

    /**
    * Cause event when obstacle status has changed.
    * May be called by subclass methods.
    */
    protected void notify (Object object) {
        event.notify (this) ;
    }
}

```

We assume the sensor can do a number of things. It can detect an obstacle. It can determine its direction relative to the J-Bot and determine the distance to the obstacle. Although the abstract class supports a fine resolution, an implementation may not. For example, the contact whiskers used in the next chapter are limited to detecting an object to the left and the right and the distance to the J-Bot is always 0 since the object must be in contact with the J-Bot to be detected. Infrared range finders provide a more accurate reading at a much farther range.

The following is a sample subclass of BaseSensor. As noted earlier, it uses a random number generator to indicate when an obstacle is found.

```

package JBot;

import stamp.util.os.* ;
import java.util.* ; // for Random class

/**
 * Random sensor class
 * <p>
 * This sensor generates random obstacle information.
 * It only works changes obstacle information when polled.
 * Direction and distance are random.
 *
 * @version 1.0 8/23/02
 * @author Parallax Inc.
 */

public class RandomSensor extends BaseSensor {
    protected int direction ;
    protected int distance ;
    protected Random generator = new Random () ;
}

```

```

/**
 * Generate a random number between 0 and limit
 *
 * @param limit maximum random value
 *
 * @returns obstacle detected
 */
protected int random ( int limit ) {
    int result = generator.next () / (Random.MAX RAND/(limit+1)) ;

    return ( result > limit ) ? ( result - 1 ) : result ;
}

/**
 * Indicate whether an obstacle has been detected.
 * Normally used when polling versus using an event.
 *
 * @returns obstacle detected
 */
public boolean obstacleDetected () {
    if ( random ( 9 ) == 0 ) {
        // Obstacle seen 1:10 times

        direction = random(4)*45 ; // integral 45 degree values from 0-180
        distance = random(10) ;
        notify () ;
        return true ;
    } else {
        // Nothing detected. Reset stored values.

        direction = 0 ;
        distance = none ;
        return false ;
    }
}

/**
 * Indicate initial obstacle position.
 * For simple detection systems the detection of an object
 * on the right and left will return front.
 *
 * @returns obstacle's relative direction (left, right, etc.)
 */
public int obstacleDirection () {
    return direction ;
}

/**
 * Get the distance to an obstacle in the specified direction.
 * A value of <code>none</code> indicates no object detected.
 *
 * @param direction to get range for
 *
 * @returns distance to an obstacle for the specified direction
 */
public int obstacleDistance ( int direction ) {
    return distance ;
}

/**
 * Set minimum event notification distance.
 * Notification will not occur until an obstacle is

```

```

    * outside of this distance. The minimum value is 0.
    *
    * @param minimumDistance minimum number of inches to detect an obstacle
    */
    public void setMinimumEventDistance () {
        /* Default case is to ignore the minimum distance
        * For example, contact oriented sensors can only detect objects
        * when they are in contact with them.
        */
    }
}

```

The random generation of an obstacle occurs about once every tenth time the random sensor object is polled using the `obstacleDetected` method. It stores a random direction and distance so the controlling task can obtain values that will not change until the object is polled again. Although the event object is not set in the test programs in this chapter they may in later applications which is why the `notify` method is called when an object is detected. If the sensor object is operated independently of the object that will use the obstacle information then a task can be setup to do the polling and the change in status can be signaled through the event.

Instead of putting the J-Bot support class in the test class we create a standalone support task class for avoiding obstacles. This will allow the task to be used in other applications in later chapters. The following is the `AvoidObstacleTask` class source code.

```

package JBot;

import stamp.util.os.* ;

/**
 * Simple obstacle avoidance task
 * <p>
 * This tries to stay away from obstacles using a sensor object.
 * The J-Bot will be moved in fixed increments.
 *
 * @version 1.0 8/23/02
 * @author Parallax Inc.
 */

public class AvoidObstacleTask extends Task {
    JBotInterface jbot ;
    BaseSensor sensor ;

    public AvoidObstacleTask ( BaseSensor sensor, JBotInterface jbot ) {
        this.sensor = sensor ;
        this.jbot = jbot ;
    }

    protected void execute () {
        final int turnAround = 1 ;

        switch ( state ) {

```

```

case initialState:
    if ( sensor.obstacleDetected () ) {
        int direction = sensor.obstacleDirection () ;

        if ( sensor.obstacleDistance ( direction ) < 2 ) {
            // Too close, back up 2 inches, then turn around

            jbot.move ( -2 ) ;
            jbot.wait ( turnAround ) ;
        } else {
            // Enough room to pivot away from object

            if ( direction < 75 ) {
                // Something to the left
                jbot.pivot ( -2 ) ;
            } else {
                // Something in front or to the right
                jbot.pivot ( 2 ) ;
            }
            jbot.wait ( turnAround ) ;
        }
    } else {
        // Nothing detected. Move forward 1 inch

        jbot.move ( 1 ) ;
        jbot.wait ( initialState ) ;
    }
    break;

case turnAround:
    // J-Bot has backed up. Time to pivot 180 degrees

    jbot.pivot ( 4 ) ;
    jbot.wait ( initialState ) ;
    break;

default:
    // default catches bad states
    stop () ;
    break;
}
}
}

```

The AvoidObstacleTask requires a sensor and a JBotInterface in its constructor. This allows any combination to be used from a single tasking sensor and JBotInterface to a multitasking version.

The AvoidObstacleTask uses a simple algorithm to keep moving. It checks the sensors to see if an object is in the way. If the object is in contact or very close to the J-Bot then the program backs the J-Bot away from the obstacle and turns around 180° which is 4 pivot steps. If the obstacle is farther away then the J-Bot will try pivoting away from the obstacle. The J-Bot moves forward one inch if no obstacle is detected.

The program makes two assumptions. First, it is able to move an inch forward if no obstacle is detected. Second, there is no

obstacle behind it when it starts. The first assumption may not be valid with a contact sensor but it may work in a practical sense because either the obstacle will be pushed for up to an inch or the J-Bot will spin its wheels for a short period of time if it runs into a fixed object that it cannot move. In either case, the J-Bot should be able to make adjustments and proceed on forever.

The test program is now significantly shorter since most of the work is done in the AvoidObstacleTask class just presented.

```
import stamp.core.*;
import stamp.util.os.* ;
import JBot.* ;

/**
 * Test AvoidObstacleTask class
 * <p>
 * Tun the J-Bot so it avoids obstacles.
 *
 * @version 1.0 7/23/02
 * @author Parallax Inc.
 */

public class AvoidObstacleTaskTest1 {
    public static void main () {
        new AvoidObstacleTask
            ( new RandomSensor ()
              , new RampingJBot ( new MultitaskingJBot ())) ;

        Task.TaskManager () ;
        System.out.println ( "All done" ) ;
    }
}
```

The test program is now simply a main method that creates the AvoidObstacleTask object. The parameter to the task is the RandomSensor and a RampingJBot object. The J-Bot will now wander around as if it were detecting obstacles using built-in sensors. We will be able to use the classes just presented with other sensors defined in later chapters by simply changing the test program so the appropriate sensor is used. If the sensor requires a task then it can be created as well, either by the sensor object or in the main method.

Now that the test program is available, the J-Bot can be programmed and it will then run forever. Of course, movement could be limited by adding a timer task that stopped the other tasks after a fixed amount of time. This is one of the exercises presented at the end of the chapter.

Activity #4: Task Status And Living Without Garbage Collection

The multitasking programs presented up to this point have been relatively simple and well tested. Debugging can get more complex as the number of tasks increases and as the tasks become more complex. Luckily there are ways to make debugging easier.

At this point it is useful to take a quick look at memory allocation using the Javelin. In particular, the Javelin does not support garbage collection. This can be a problem if an application is not properly designed because it is possible to allocated all available memory at which point the Javelin will terminate the application. The Javelin will notify the debugger of the error if the Javelin is connected to the PC with the IDE running.

It is preferable to catch problems before they happen. We present some status classes and tasks that will provide more insight into what the multitasking system is doing. These can be used with any set of tasks. They will not be used elsewhere in this book but they should be added if problems are encountered building new applications or modifying ones presented in this book.

The first step is the ShowStatus class. This is the basis for the TaskStatus and SemaphoreStatus classes. Here is the ShowStatus class.

```
package stamp.util.os;

import java.io.* ;
import stamp.core.* ;

/**
 * Multitasking status support class
 * <p>
 * This is a class object only.
 *
 * @version 1.0 8/23/02
 * @author Parallax Inc.
 */

public class ShowStatus {
    static public PrintStream stream = System.out ;

    protected static void print ( Event event ) {
        stream.print ( "<" ) ;
        stream.print ( event.name () ) ;
        stream.print ( "> " ) ;
    }

    protected static void print ( int value ) {
        stream.print ( value ) ;
    }

    protected static void print ( String string ) {
        stream.print ( string ) ;
    }

    protected static void println ( String string ) {
```

```

    stream.println ( string ) ;
}

protected static void println () {
    stream.println ( " " ) ;
}
}

```

This class, like the TaskStatus and SemaphoreStatus classes provide only class methods. The constructor is protected to prevent creation of class objects that would do nothing. This class provides basic print support that can be redirected by changing the stream variable. Note that the Event class is the basis for the Task and Semaphore classes.

Next is the TaskStatus class file.

```

package stamp.util.os;

import java.io.* ;
import stamp.core.* ;

/**
 * Display multitasking status
 * <p>
 * Shows active tasks and free memory.
 * This is a class object only.
 *
 * @version 1.0 8/23/02
 * @author Parallax Inc.
 */

public class TaskStatus extends ShowStatus {
    protected TaskStatus () {
        // Prevents creation of object
    }

    protected static void printStateName ( int state ) {
        switch ( state ) {
            case Task.initialState:
                print ( "initial " ) ;
                break ;

            case Task.checkTimer:
                print ( "sleeping" ) ;
                break ;

            case Task.interrupt:
                print ( "interrupt" ) ;
                break ;

            case Task.terminate:
                print ( "terminate" ) ;
                break ;

            case Task.stopped:
                print ( "stopped " ) ;
                break ;
        }
    }
}

```

```

        default:
            print ( state ) ;
            break ;
        }
    }

/**
 * Print memory status
 *
 * @param stream output stream
 */
public static void printMemoryStatus () {
    print ( Memory.freeMemory () ) ;
    println ( " bytes free" ) ;
}

/**
 * Print task status
 */
public static void printTaskStatus ( Task task ) {
    print ( task ) ;
    printStateName ( task.state ) ;
    println () ;
}

/**
 * Print list of tasks
 *
 * @param prefix print before list of tasks
 * @param running true if running tasks should be listed
 */
public static void printTasks ( String prefix, boolean running ) {
    println ( prefix ) ;

    for ( Task task = Task.taskStatusList
        ; task != null
        ; task = task.nextTaskStatus
        ) {
        if ( task.running () == running ) {
            print ( " " ) ;
            printTaskStatus ( task ) ;
        }
    }
}

/**
 * Generate a random number between 0 and limit
 *
 * @param stream output stream
 */
public static void show () {
    Task task = Task.taskList ;

    println ( "== Task Status ==" ) ;
    printMemoryStatus () ;

    printTasks ( "Tasks running", true ) ;
    printTasks ( "Tasks not running", false ) ;

    if ( task == null ) {
        println ( "-- Empty task list --" ) ;
    }
}

```

```

    } else {
        // Check taskList integrity
        do {
            task = task.nextTask ;
            if ( task == null ) {
                println ( "== Error: Task list not circular ==" ) ;
                break ;
            }
        } while ( Task.taskList != task ) ;
    }

    println () ;
}

```

The show method provides a detailed account of all tasks. Individual task status can be obtained using printTaskStatus. This support is included in this class instead of the Task class so there is no added overhead using Tasks if the status methods are not used.

The SemaphoreStatus class file is very similar to the TaskStatus class.

```

package stamp.util.os;

import java.io.* ;
import stamp.core.* ;

/**
 * Display multitasking status
 * <p>
 * Shows active tasks and free memory.
 * This is a class object only.
 *
 * @version 1.0 8/23/02
 * @author Parallax Inc.
 */

public class SemaphoreStatus extends ShowStatus {
    protected SemaphoreStatus () {
        // Prevents creation of object
    }

    /**
     * Print task status including semaphore information
     */
    public static void printTaskStatus ( Task task ) {
        // Print general status
        TaskStatus.printTaskStatus ( task ) ;

        // Print acquired semaphores
        for ( Semaphore semaphore = Semaphore.semaphoreList
            ; semaphore != null
            ; semaphore = semaphore.nextSemaphore
            ) {
            if ( semaphore.acquiredTask == task ) {
                print ( " Acquired " ) ;
            }
        }
    }
}

```

```

        print ( semaphore ) ;
        println () ;
    }
}

// Print semaphore waiting status
for ( Semaphore semaphore = Semaphore.semaphoreList
    ; semaphore != null
    ; semaphore = semaphore.nextSemaphore
    ) {
    for ( Task waitingTask = semaphore.waitingTaskList
        ; waitingTask != null
        ; waitingTask = waitingTask.nextTask
        ) {
        if ( waitingTask == task ) {
            print ( " Waiting for " ) ;
            print ( semaphore ) ;
            println () ;
        }
    }
}

/**
 * Generate a random number between 0 and limit
 *
 * @param stream output stream
 */
public static void show () {
    println ( "== Semaphore Status ==" ) ;

    for ( Semaphore semaphore = Semaphore.semaphoreList
        ; semaphore != null
        ; semaphore = semaphore.nextSemaphore
        ) {
        print ( semaphore ) ;
        if ( semaphore.ready () ) {
            println ( "ready" ) ;
        } else {
            print ( "acquired by " ) ;
            print ( semaphore.acquiredTask ) ;
            println () ;

            if ( semaphore.acquiredTask.state == Task.stopped ) {
                println ( " ** Error: Task stopped **" ) ;
            }

            println ( " Waiting tasks" ) ;

            for ( Task task = semaphore.waitingTaskList
                ; task != null
                ; task = task.nextTask
                ) {
                print ( " " ) ;
                print ( task ) ;
                println () ;
            }
        }
    }

    println () ;
}
}

```

The show method operates in a fashion similar to the show method in the TaskStatus class. The method walks the semaphoreList and the waitingTaskList of each semaphore providing details about the tasks involved. The printTaskStatus method provides a more detailed report compared to the TaskStatus method of the same name. While the show methods of both are normally called together the SemaphoreStatus.printTaskStatus is normally used alone since it is a superset of the information provided by the TaskStatus class. In general, the SemaphoreStatus class is not required if semaphores are not used by an application.

Finally we have the WatchHeapTask. This task is designed to track memory usage and report any changes (which will always be an increase). The class definition is as follows.

```
package stamp.util.os;

import stamp.core.*;
import stamp.util.os.*;

/**
 * Watch heap space task.
 * <p>
 * Start this task after all memory has been allocated.
 * It will track free memory and report any changes.
 *
 * @version 1.0 8/15/02
 *
 * @author Parallax Inc.
 */

public class WatchHeapTask extends Task {
    protected int    freeMemory = 0 ;
    protected boolean enable = true ;
    public    Event    event ;

    /**
     * Create WatchHeapTask.
     * Errors reported on System.out
     */
    public WatchHeapTask () {
    }

    /**
     * Control error checking
     */
    public void enable ( boolean enableNow ) {
        enable = enableNow ;

        if ( enableNow ) {
            freeMemory = 0 ;
        }
    }

    /**
     * Return task name

```

```

    */
    public String name () {
        return "WatchHeap" ;
    }

    /**
     * Create WatchHeapTask.
     * Errors reported by notifying the event.
     *
     * @param event Event to notify when heap grows
     */
    public WatchHeapTask ( Event event ) {
        this.event = event ;
    }

    protected void execute () {
        /* Memory.freeMemory is stack point dependent
         * so it must be called at the same point each
         * time to provide consistent information
         */
        int latestFreeMemory = Memory.freeMemory () ;

        // Exit if this is the only task running

        if ( nextTask == this ) {
            stop () ;
            return ;
        }

        if ( enable ) {
            if ( freeMemory == 0 ) {
                freeMemory = latestFreeMemory ;
            } else if ( freeMemory != latestFreeMemory ) {
                // Memory usage has increased (cannot decrease)

                if ( event == null ) {
                    System.out.println ( "Error: Memory leak" ) ;
                    System.out.print  ( " Last: " ) ;
                    System.out.println ( freeMemory ) ;
                    System.out.print  ( " Now:  " ) ;
                    System.out.println ( latestFreeMemory ) ;
                } else {
                    event.notify () ;
                }

                // Reset low water mark
                freeMemory = latestFreeMemory ;
            }
        }
    }
}

```

The execute method is a bit unique in that it does not use the state variable because there is no need. The task can be used with an Event or System.out which is used if no event is provided. The Memory class is used to obtain the amount of free memory available. This value must be taken at the same point in the program each time because it is based on the Java stack pointer and heap limit. As it turns out, the stack pointer will

always be in the same spot when the task's execute method is called. This is why the free memory value is saved in latestFreeMemory.

The task simply stores off the current free memory value the first time through. This should allow other applications to perform their initial memory allocation. After that point memory allocation normally ceases. If not, the task can be started later or initial error messages can be ignored. The task resets its freeMemory variable each time a memory increase is detected.

The memory checking can be enabled and disabled using the enable method. Enabling checking will reset the freeMemory variable so it will be set the next time the task executes. Note, this is different than suspending the task. In the latter case, the freeMemory variable will not be reset but the task will not be able to detect any changes until the task is resumed.

So what can we do with all this new found information. Looking back to the original multitasking demo program we come up with the following.

```
import stamp.core.*;
import stamp.util.os.*;

/**
 * Multitasking Test 1
 *
 * @version 1.0 8/15/02
 *
 * @author Parallax Inc.
 */

public class MutlitaskingTest6 extends Task {
    // execute() states
    final static int state1 = 1 ;
    final static int state2 = 2 ;
    final static int state3 = 3 ;

    static Semaphore semaphore = new Semaphore ( ) ;

    String name ;

    MutlitaskingTest6 ( String name ) {
        this.name = name ;
    }

    /**
     * Get task name.
     *
     * @return name of task.
     */
    public String name () {
        return name ;
    }

    public void show ( String text ) {
        System.out.print ( name ) ;
    }
}
```



```

        System.out.print ( ":" );
        System.out.println ( text );
    }

    protected void execute () {
        switch ( state ) {
            case initialState:
                show ( "Initial state" );
                nextState ( state1 );
                break ;

            case state1:
                show ( "State 1" );
                new Integer ( 0 ); // memory leak !!!
                nextState ( state2 );
                break ;

            case state2:
                show ( "State 2" );
                if ( ! semaphore.acquire ( state3 ) ) {
                    stop () ;
                }
                break ;

            case state3:
                show ( "State 3" );
            default: // terminate should be the default to catch bad states
                stop () ;
                break;
        }
    }

    public static void main() {
        Task task1 = new MutlitasKingTest6 ( "Task 1" );
        Task task2 = new MutlitasKingTest6 ( "Task 2" );
        Task task3 = new MutlitasKingTest6 ( "Task 3" );
        Task task4 = new WatchHeapTask () ;

        // Show general status
        TaskStatus.show () ;
        SemaphoreStatus.show () ;

        // Run tasks
        Task.TaskManager () ;

        // Show general status
        TaskStatus.show () ;
        SemaphoreStatus.show () ;

        // Show task status
        SemaphoreStatus.printTaskStatus ( task1 );
        SemaphoreStatus.printTaskStatus ( task2 );
        SemaphoreStatus.printTaskStatus ( task3 );
        SemaphoreStatus.printTaskStatus ( task4 );

        System.out.println ( "All done" );
    }
}

```

The MultitaskingTest6 objects all have a unique name assigned to them. This is often done in the class definition if only one instance of the class will be created. In this case there are three so differentiating the tasks is useful in debugging. There is only one Semaphore so its name is left as the default.

The TaskStatus and SemaphoreStatus show methods are used to provide a general overview before and after the task manager starts running the programs. These methods can be called while the system is running as well. Individual task status information is presented at the end using the printTaskStatus method.

A WatchHeapTask object is also created. We include a memory leak in state1 of the MultitaskingTest6 by allocating an Integer object. This could be any Java object since they all reduce the amount of free space.

Running the main method generates quite a bit of text in the message window. The initial status information shows the four tasks that are running. The WatchHeapTask will also report a memory leak although it will do so only once since all three tasks will allocate memory at almost the same time. The WatchHeapTask task will only notice one increase but it does notify you of the problem.

The information at the end shows off two errors that were programmed in the execute method. The first is that Task 1 is stopped while it has acquired the lone semaphore. The second problem is that the other two tasks are waiting on this semaphore which will never be released. This means the other two tasks would never continue.

The MultitaskingTest6 class shows off two common programming errors and how they can be detected. The status classes were also presented. These can provide insight into the operation of a multitasking application.



Summary and Applications

Congratulations, understanding multitasking is not an easy task! Through following the procedures in this chapter, you may have had your first taste of testing and troubleshooting a multitasking system. Most of the classes will be used in the following chapters although the test programs will not.

Activity #3 presented a basic obstacle avoidance system that was coupled with a psuedo sensor system that used random numbers to simulate obstacle detection. Subsequent chapters will use real sensors.

Activity #4 introduce the garbage collection problem. While the solution is good programming techniques, a sample task that checks for memory leaks was presented. This task can be used as a debugging tool by incorporating it into other applications where memory leakage have become a problem.

Real World Example

Multitasking is used in many embedded systems and most robots. It allows a processor to control many devices at the same time. For example, a system that controls the heating plant in an office may have a number of temperature sensors that need to be monitored along with temperature settings. There would also be fans to control air movement and heating and cooling units to control as well.

J-Bot Application

The Projects section challenges you to make the J-Bot do more than was presented in this chapter. Adding more tasks to the system reduces the amount of time each has to execute but it allows more functions to be performed. In this case the J-Bot gets to walk (roll actually) and chew gum (play a tune) at the same time.



Questions and Projects

Questions

1. Explain the difference between preemptive and non-preemptive multitasking systems.
2. Explain the difference between background virtual peripheral multitasking and the non-preemptive, state machine multitasking classes.
3. What happens to the other tasks if a task calls `CPU.delay(30000)`?
4. Why would you use a semaphore object?

Exercises

1. Setup two tasks to control the J-Bot. Use a semaphore object to limit access to one task at a time.
2. Use a `CallableTask` to control the J-Bot so a called state machine subroutine handles part of the arc movement within a figure 8 path. Use this so the J-Bot will circumscribe a figure 8. Try other complex figures using the same technique.
3. Create a `DistributeEvent` class that allows zero or more events to be added (and deleted) to a `DistributeEvent` object. Each of these events should be notified when the `notify` method of the `DistributeEvent` object is called.
4. Add a timer-based task to the `AvoidObstacleTaskTest1` program. The timer task should sleep for a fixed amount of time such as 15 seconds and then stop the other tasks controlling the J-Bot.
5. The `WatchHeapTask` currently checks for memory leaks. It could also check to see if the amount of free memory is getting very low. Add this type of check to the `execute` method so it generates an error if the amount of free memory goes below 1000 bytes. Create enough objects so this error occurs to verify the test. Then modify the class so the limit value is maintained in a variable.

Projects

1. A multitasking J-Bot application uses only a couple tasks. Add another task that uses the TaskToneGenerator to play a tune while the J-Bot roams around the floor.

Does the tone generation affect the movement of the J-Bot?

2. Unfortunately, the J-Bot will normally be disconnected from the PC when the J-Bot is moving. It was difficult to keep the J-Bot connected to the PC when the J-Bot was programmed for limited movement. Running programs like those in the prior activity make it impractical to do so. The J-Bot may simply stop if an error occurs. Use the WatchHeapSpace task so it will sound an audible tone when an leakage error occurs. Hint: Use FREQOUT instead of TaskToneGenerator.
3. It is often useful to provide status information on demand. This could be provided via a task that waits for PC keyboard input to the message window. Add a task that will poll for input and deliver the status information when the spacebar is pressed.