ROBOID LIBRARY 1.4.1

Technical Document

December 2009

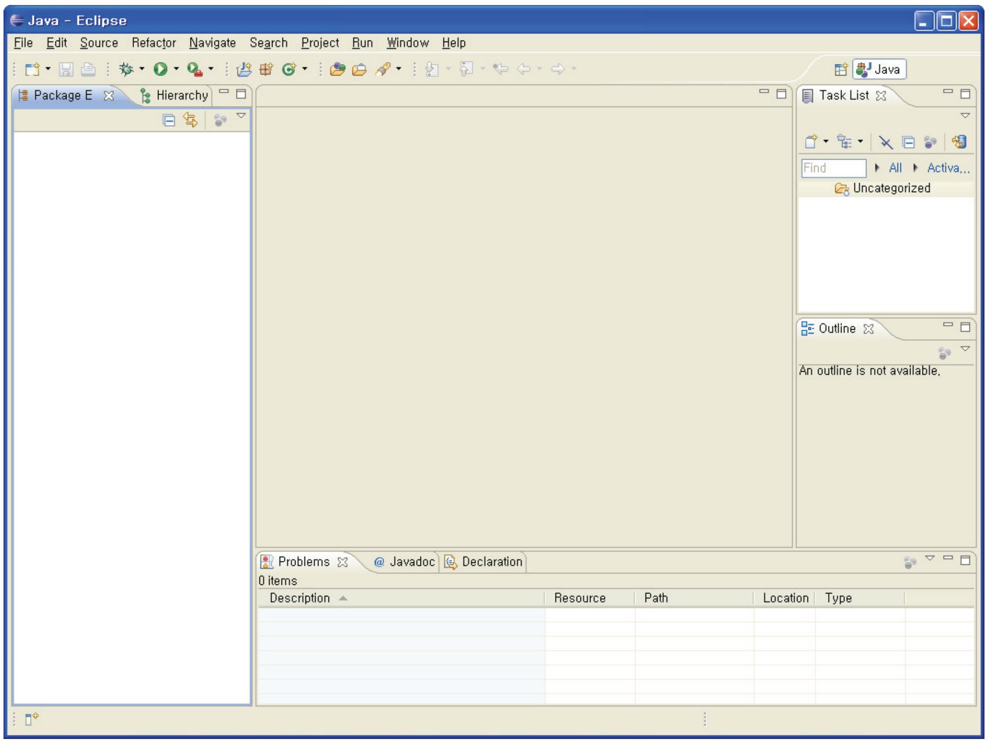
www.roboidstudio.org

1. Starting Roboid Library

Roboid Library is for teaching Java language more interestingly through moving hardware devices such as pelican and donkey and processing data from sensors with Java programming. By operating robots with sensors and actuators additionally to usual Java programming, it can provide interesting and motivating education different from conventional boring programming education.

Because Roboid Library provides the part for exchanging data with the sensors and actuators of hardware devices, it has a limitation in producing rich robot contents like Roboid Studio does. Accordingly, it is not good for creating robot contents with Roboid components and graphic tools including voice recognition, image recognition, and Web services, and Roboid Studio is recommended for such works.

Roboid Library interoperates with Java 1.5 and 1.6, and can be used in Eclipse 3.3 (Europa), 3.4 (Ganymede), and 3.5 (Galileo). In order to install Roboid Library, start Eclipse first as in Figure 1.1. Here, we explain based on Eclipse 3.5-Galileo Packages-Eclipse IDE for Java Developers.



An outline is not available.

0 items

Figure 1.1 Eclipse IDE

Select **Import…** from the **File** menu, and when dialog box ‘**Import**’ appears, select **General** > **Existing Projects into Workspace** and press the **Next** button.

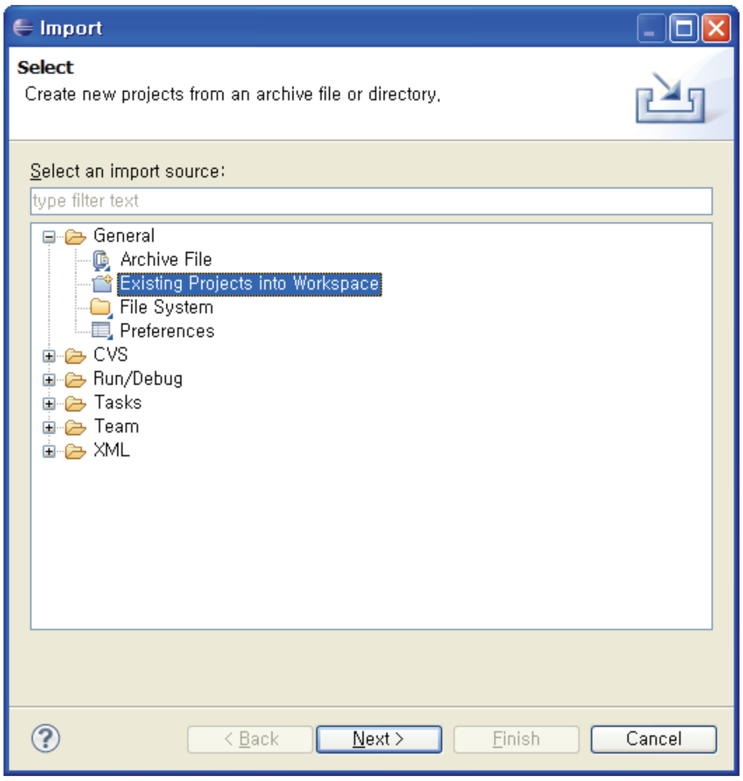
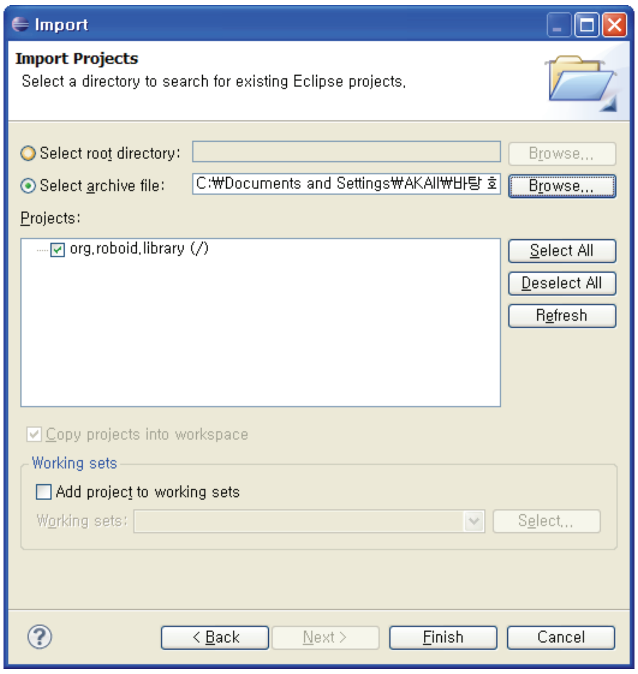


Figure 1.2 Dialog box ‘**Import**’

As in Figure 1.3, select **Select archive file**, and then press the **Browse…** button and select **org.roboid.library_1.4.1.zip** file, and press the **Finish** button. As in Figure 1.4, project **org.roboid.library** appears on the **Package Explorer** view.

The project folder consists of sub-folders **src**, **examples**, and **roboid**. The **src** folder contains Java source codes for connection to hardware devices for each hardware roboid, and the **roboid** folder contains execution files and DLL files for driving hardware devices. The **examples** folder contains example source codes that read sensor data and operate hardware devices using execution files and DLL files in the **roboid** folder.

Let’s examine example source codes one by one.


Import

Import Projects

Select a directory to search for existing Eclipse projects.

Select root directory: [ ] Browse...

Select archive file: [✓] C:₩Documents and Settings₩AKAll₩바탕 화 Browse...

Projects:

☑ org.roboid.library (/)

Select All | Deselect All | Refresh

☑ Copy projects into workspace

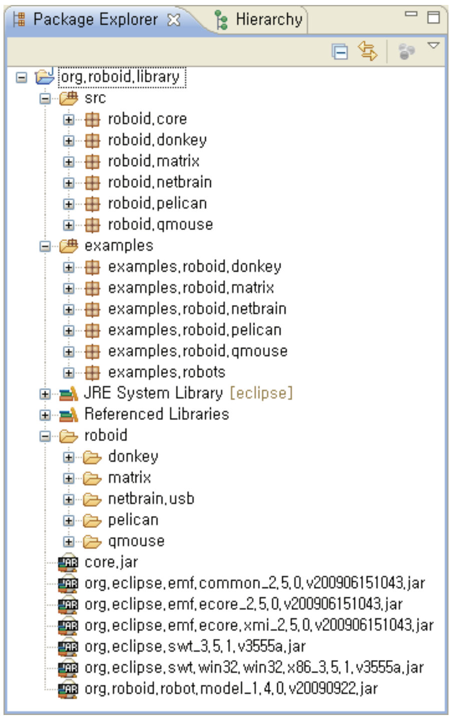
Working sets

☐ Add project to working sets

Working sets: Select...

< Back | Next > | Finish | Cancel

Figure 1.3 Zip file selection


Package Explorer | Hierarchy

```
org.roboid.library
  src
    roboid.core
    roboid.donkey
    roboid.matrix
    roboid.netbrain
    roboid.pelican
    roboid.qmouse
  examples
    examples.roboid.donkey
    examples.roboid.matrix
    examples.roboid.netbrain
    examples.roboid.pelican
    examples.roboid.qmouse
    examples.robots
  JRE System Library [eclipse]
  Referenced Libraries
  roboid
    donkey
    matrix
    netbrain.usb
    pelican
    qmouse
  core.jar
  org.eclipse.emf.common_2.5.0.v200906151043.jar
  org.eclipse.emf.ecore_2.5.0.v200906151043.jar
  org.eclipse.emf.ecore.xmi_2.5.0.v200906151043.jar
  org.eclipse.swt_3.5.1.v3555a.jar
  org.eclipse.swt.win32.win32.x86_3.5.1.v3555a.jar
  org.roboid.robot.model_1.4.0.v20090922.jar
```

Figure 1.4 Package Explorer view

2. Examining example source codes of Pelican

Here we will analyze only the example source codes of Pelican. Analyze examples for other hardware roboids by yourself. Do not forget to connect the hardware device to the PC before executing the source codes.

Example_01.java

On the bottom of the source code is method `wait` as in Figure 2.1. Method `wait` is used to wait for a given length of `time` (`time` × 1msec).

```
private static void wait(int time)
{
        try
        {
                Thread.sleep(time);
        } catch (InterruptedException e)
        {}
}
```

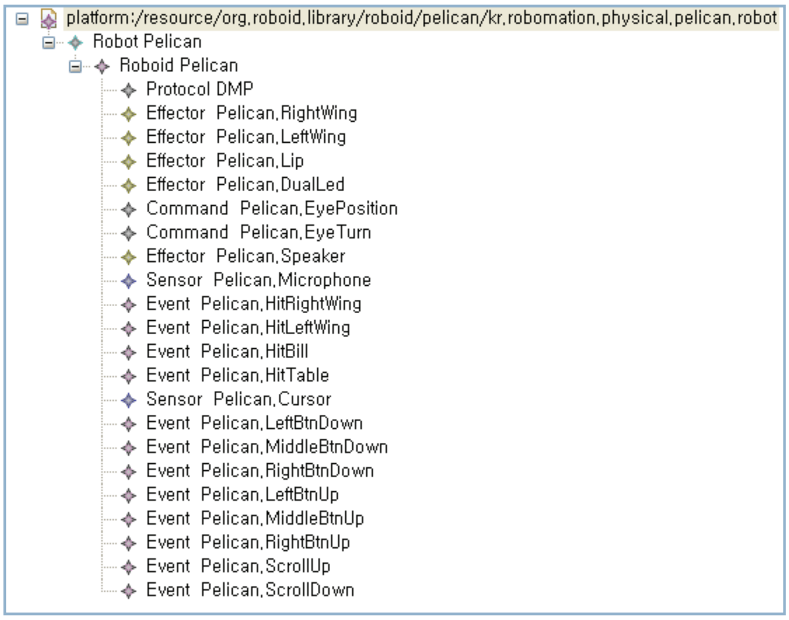
Figure 2.1 method `wait`

As in Figure 2.2, a `Pelican` instance is created at the first part of `main`, and the reference to the instance is stored in variable `pelican`. At that time, it takes a while for the PC to recognize the Pelican hardware connected by USB, so a code was added that waits for 3 seconds. If Pelican is connected to the PC, the LED blinks, the eyes turn, and '`Pelican activated...`' is displayed on the console view of Eclipse.

```
Pelican pelican = new Pelican();
wait(3000);
```

Figure 2.2 Pelican instance creation

Devices such as sensors and actuators in Pelican can be accessed through the reference obtained earlier. The reference of a device instance can be obtained by '.device_name.' The devices of Pelican are described in the model, and are as in Figure 2.3 and Table 2.1. Device data are read by method `read`, and written by method `write`. Because the code in Figure 2.4 is for moving the bill of Pelican, it sets the degree of bill opening by method `write`. The bill is opened fully at 63, and is closed completely at 0. Because it takes time for a hardware device to make an actual movement, the process of the example source code waits for a second after commanding full opening, and for a second again after commanding complete closing.



```
platform:/resource/org.roboid.library/roboid/pelican/kr.robomation.physical.pelican.robot
  Robot Pelican
    Roboid Pelican
      Protocol DMP
      Effector Pelican.RightWing
      Effector Pelican.LeftWing
      Effector Pelican.Lip
      Effector Pelican.DualLed
      Command Pelican.EyePosition
      Command Pelican.EyeTurn
      Effector Pelican.Speaker
      Sensor Pelican.Microphone
      Event Pelican.HitRightWing
      Event Pelican.HitLeftWing
      Event Pelican.HitBill
      Event Pelican.HitTable
      Sensor Pelican.Cursor
      Event Pelican.LeftBtnDown
      Event Pelican.MiddleBtnDown
      Event Pelican.RightBtnDown
      Event Pelican.LeftBtnUp
      Event Pelican.MiddleBtnUp
      Event Pelican.RightBtnUp
      Event Pelican.ScrollUp
      Event Pelican.ScrollDown
```

Figure 2.3 Pelican model (**kr.robomation.physical.pelican.robot** file in folder **roboid/pelican**)

Table 2.1 Devices in the Pelican model and their fields in class `Pelican`

Device in model	Field in class	Range	Description
RightWing	`rightWing`	0 ~ 63	The degree of lifting the right wing
LeftWing	`leftWing`	0 ~ 63	The degree of lifting the left wing
Lip	`bill`	0 ~ 63	The degree of opening the bill
DualLed	`led`	0 ~ 63	An array of size 2. The first value of the array is the degree of red, and the second one is the degree of green.
EyePosition	`eye`	0 ~ 6	The ID of eye shape. There are six eye shapes numbered between 1-6. 0 maintains the current eye shape.
EyeTurn	`eyeTurn`	-127 ~ 127	The number of segments that the eyes move, indicating how long the eyes turn from their current position.
Speaker	`speaker`	-32768 ~ 32767	Speaker output sound data, which is an array of size 960

Microphone	`microphone`	-32768 ~ 32767	Microphone input sound data, which is an array of size 160
HitRightWing	`hitRightWing`	No data	An event is triggered by hitting the right wing.
HitLeftWing	`hitLeftWing`	No data	An event is triggered by hitting the left wing.
HitBill	`hitBill`	No data	An event is triggered by hitting the bill.
HitTable	`hitTable`	No data	An event is triggered by hitting the table.
Cursor	`cursor`	-32768 ~ 32767	The coordinates of the mouse cursor, an array of size 2. The first value of the array is x coordinate, and the second y coordinate.
LeftBtnDown	`leftButtonDown`	No data	An event is triggered by pressing down the left button of the mouse.
MiddleBtnDown	`middleButtonDown`	No data	An event is triggered by pressing down the middle button of the mouse.
RightBtnDown	`rightButtonDown`	No data	An event is triggered by pressing down the right button of the mouse.
LeftBtnUp	`leftButtonUp`	No data	An event is triggered by releasing the left button of the mouse.
MiddleBtnUp	`middleButtonUp`	No data	An event is triggered by releasing the middle button of the mouse.
RightBtnUp	`rightButtonUp`	No data	An event is triggered by releasing the right button of the mouse.
ScrollUp	`scrollUp`	No data	An event is triggered by scrolling up the mouse wheel.
ScrollDown	`scrollDown`	No data	An event is triggered by scrolling down the mouse wheel.

```
pelican.bill.write(63);
wait(1000);
pelican.bill.write(0);
wait(1000);
```

Figure 2.4 Operating the bill

Likewise, the code in Figure 2.5 commands the left wing and the right wing to lift to the full, to wait for a second, and then to lower down completely, and wait for a second.

```
pelican.leftWing.write(63);
pelican.rightWing.write(63);
wait(1000);
pelican.rightWing.write(0);
pelican.leftWing.write(0);
wait(1000);
```

Figure 2.5 Operating the wings

The code in Figure 2.6 commands the eye shape to be No. 5, to wait for a second, and then to be No. 4 and No. 5, and to wait for a second.

```
pelican.eye.write(5);
wait(1000);
pelican.eye.write(4);
wait(1000);
pelican.eye.write(5);
wait(1000);
```

Figure 2.6 Changing the eye shape

Lastly, when ending the instance after all operations, call method `dispose` as in Figure 2.7. Method `dispose` ends the thread created internally, terminates network communication, and closes execution files for driving hardware devices.

```
pelican.dispose();
```

Figure 2.7 Ending an internal instance

Example_02.java

Example_02 is basically the same as Example_01 except that it makes the motions gentler by changing input data for each device more gradually.

When writing data in the left wing, the right wing, and the bill, Example_01 changed the value from 63 to 0 directly, but as in Figure 2.8 Example_02 changes the value gradually by increasing it one by one. Thus, the hardware device's motion becomes gentle. Roboid Studio has a function that interpolates motions seamlessly, but when a hardware device is controlled directly using Roboid Library, values between the two extremes should be calculated and used in order to make the motion smooth. Between each pair of commands is 20msec's wait, and the moving speed of the hardware device can be adjusted by changing the waiting time. In addition, two seconds' pause was given after a motion has been finished.

```
for(int i = 0; i < 64; i++)
{
        pelican.leftWing.write(i);
        pelican.rightWing.write(i);
        pelican.bill.write(i);
        wait(20);
}
wait(2000);
```

Figure 2.8 Lifting the wings and opening the bill gently

The code in Figure 2.9 lowers the wings and closes the bill. The operation is the same as Figure 2.8 except that the direction is opposite. As waiting time is 50msec in Figure 2.9, the moving speed of the hardware device is slower.

```
for(int i = 0; i < 64; i++)
{
        pelican.leftWing.write(63 - i);
        pelican.rightWing.write(63 - i);
        pelican.bill.write(63 - i);
        wait(50);
}
wait(2000);
```

Figure 2.9 Lowering the wings and closing the bill gently

Example_03.java

Example_02 uses method `wait` in order to set the waiting time after each round of operations. Roboid Library provides interface `Controller` as in Figure 2.10 in order to execute a command at fixed time intervals. Method `execute` in interface `Controller` is called each time when a thread, which has been created in a `Pelican` instance and is communicating with a hardware device, receives data from the hardware device. Because the hardware device sends data at 20msec's intervals, method `execute` is called every

20msec. Accordingly, if method `execute` of interface `Controller` is implemented, an action can be performed every 20msec.

```
public interface Controller
{
        void execute(Roboid roboid);
}
```

Figure 2.10 Interface `Controller`

Now, let's return to the source code of Example_03. Different from the examples above, in Figure 2.11, class `Example_03` implemented interface `Controller`. Interface `Controller` can be implemented in a separately defined class, but in this example interface `Controller` was implemented in class `Example_03` for convenience. As in Figure 2.12, accordingly, method `execute` was implemented in class `Example_03`.

```
public class Example_03 implements Controller
```

Figure 2.11 Implementing interface `Controller`

Method `execute` in Figure 2.12 is called every 20msec, and at each call, the value used by the devices increases by 1, and after the value reaches 63 it becomes 0. Thus, the wings are lifted and the bill is opened gently, but after fully lifted and opened, they are lowered and closed immediately.

```
public void execute(Roboid roboid)
{
        int j = ++i % 64;
        pelican.leftWing.write(j);
        pelican.rightWing.write(j);
        pelican.bill.write(j);
}
```

Figure 2.12 Method `execute`

In order to call method `execute` cyclically, the implementation of interface `Controller` should be registered. That is, as in Figure 2.13, the instance that implemented interface `Controller` is registered through method `setController` in class Pelican, method execute in the registered instance is called whenever data are received from the hardware device (every 20msec). Here, 'this' is the reference to the instance of class Example_03 that implemented interface Controller. The code in Figure 2.13 performs the operation described in method execute for 500 × 20msec, namely, a second and then ends.

```
pelican.setController(this);

while(i < 500){}

pelican.dispose();
```

Figure 2.13 Setting interface Controller

Example_04.java

Example_04 is an example for the output of sound through the speaker of Pelican. Roboid Studio has a code that reads a music file and outputs the sound through the speaker of Pelican, so the user does not need to take care of it. In Roboid Library, however, data to be output through the speaker should be assigned, and with regard to a music file, the parts of reading the file, making data, and outputting the data at specified time should be all implemented. This example implements a code that outputs sound while changing the tone. For this, method execute was called every 20msec using interface Controller, and as in Figure 2.14, data are written in the speaker device within method execute.

```
public void execute(Roboid roboid)
{
        pelican.speaker.write(wave);
}
```

Figure 2.14 Outputting sound data through the speaker

Because the speaker device of Pelican receives data as an array of size 960 (see Table 2.1), a wave is created as an array of size 960 as in Figure 2.15. That is, because 960 units of data are output every 20msec, it is equivalent to sampling at a rate of 960 × 50 = 48khz.

```
int[] wave = new int[960];
```

Figure 2.15 Speaker sound data array

Figure 2.16 shows a code that creates sound data to be output. Because the same length of data is repeated at intervals of the given `duration`, if the `duration` is long, sound frequency becomes low.

```
private void generate(int duration)
{
        for(int j = 0; j < 960; j++)
        {
                wave[j] = (j % duration) * 5000;
        }
}
```

Figure 2.16 Creating sound data

Now let’s examine the part of setting sound data by calling method `generate`. In the first part of Figure 2.17, after an instance of class `Pelican` is created, it waits for three seconds until the PC recognizes the Pelican hardware. After registering the instance that implements interface `Controller` through method `setController` in order to call method `execute` every 20msec, the code changes sound data by method `generate` at 100msec’s intervals. At that time, because `i` increases by 1, the frequency of the sound data decreases gradually.

```
wait(3000);
pelican.setController(this);
while(i < 20)
{
        i++;
        generate(i + 30);
        wait(100);
}
```

Figure 2.17 Setting sound data

Example_05.java

Example_05 is an example that processes sensor data coming from a sensor of Pelican. If the user hits Pelican's bill, Pelican responds to it by lifting and lowering the wings.

As in other examples above, method `execute` in Figure 2.18 is called every 20msec. All devices have a method called `e()`. If data are written in a device anywhere, an event is triggered and the trigger can be confirmed by `e()`. That is, if an event is triggered, method `e()` returns `true` or it returns `false`.

If the user hits the bill of Pelican, the hardware device informs through communication that a sensor has been detected. On receiving this message, the thread in the `Pelican` instance begins to write data in the device and therefore method `e()` returns `true`. As the thread in the `Pelican` instance calls method `execute` immediately after receiving data, it is possible to confirm whether an event has been triggered in method `execute`.

```java
public void execute(Roboid roboid)
{
        if(pelican.hitBill.e()) HIT = true;
}
```

Figure 2.18 Detecting a sensor event

Figure 2.19 describes Pelican's operation when an event has been triggered. First, it turns LED red and waits until field `HIT` becomes `true`. In Figure 2.18, field `HIT` becomes `true` if an event has been triggered, so if the user hits the bill of Pelican, the code exits the `while` statement.

The LED device has an array of size 2 for red and green, respectively, first value indicating the degree of red and the second one the degree of green. If an integer value is written by method `write`, only the first value of the array is set and, as a result, the value for red becomes 63 and that for green 0 and consequently LED turns red. In order to set both colors, creates an array of size 2, set the first value (array index 0) red and the second value (array index 1) green, and then transfer the array to the argument of method `write`.

```
pelican.led.write(63);
HIT = false;

while(!HIT){}

pelican.leftWing.write(63);
pelican.rightWing.write(63);
wait(2000);
```
Figure 2.19 Setting Pelican operation

Example_06.java

Example_06 operates in the same way as Example_05, but is implemented without using interface `Controller`. In the `while` statement of Figure 2.20, it is checked every 10msec whether an event has been triggered, and if an event has been triggered the code exits the `while` statement.

The event is maintained for 20msec, but because this example does not use method `execute`, the process of receiving data and triggering an event is performed separately from the process of checking by method `e()` whether an event has been triggered. Accordingly, even if an event has been triggered, it can be missed. Thus, it is checked every 10msec whether an event has been triggered.

```
pelican.led.write(63);

while(!pelican.hitBill.e())
{
        wait(10);
}

pelican.leftWing.write(63);
pelican.rightWing.write(63);
wait(2000);
```
Figure 2.20 Setting Pelican operation after a sensor event has been triggered

Example_07.java

Example_07 is an example that processes sound data coming from the microphone of Pelican. If the user inputs sound into the microphone, in response to that, Pelican lifts and lowers the wings.

In Figure 2.21, method `execute` reads data of the microphone device, and if the data value is larger than 1000, that is, if the volume of sound is larger than 1000, the method assumes that the user has input sound into the microphone. The sound data of the microphone device are sent every 20msec continuously, it is not necessary to check by method `e()` if sound data are written in the device.

```
public void execute(Roboid roboid)
{
        pelican.microphone.read(voice);

        for(int i = 0; i < 160; i++)
        {
                if(voice[i] > 1000) HIT = true;
        }
}
```

Figure 2.21 Processing sound data of the microphone

Because the microphone device of Pelican receives data as an array of size 160 (see Table 2.1), it creates voice as an array of size 160 as in Figure 2.22. That is, because 160 units of data are input every 20msec, it is equivalent to sampling at a rate of 160 × 50 = 8khz.

```
int[] voice = new int[160];
```

Figure 2.22 Microphone sound data array

Figure 2.23 describes Pelican’s operation when the user inputs sound into the microphone. First, it turns LED red and waits until field `HIT` becomes `true`. In Figure 2.21, field `HIT` becomes `true` if the value of sound data is larger than 1000, so if the user inputs sound into the microphone, the code exits the `while` statement.

```
pelican.led.write(63);

HIT = false;
while(!HIT){}

pelican.leftWing.write(63);
pelican.rightWing.write(63);
wait(2000);
```

Figure 2.23 Setting Pelican operation

3. Class `AbstractRoboid`

All Roboid classes including class `Pelican` are implemented by inheriting class `AbstractRoboid`. Thus, here we examine class `AbstractRoboid` first.

Figure 3.1 shows the constructor of class AbstractRoboid. modelFileName is the name of the Roboid model file, and its value is the name of the model file in the roboid folder and its relative path from the project folder. For the Pelican roboid, it is "roboid/pelican/kr.robomation.physical.pelican.robot."

```
public AbstractRoboid(String modelFileName)
```

Figure 3.1 The constructor of class `AbstractRoboid`

As in Figure 3.2, the first part of the constructor of class `AbstractRoboid` creates a Robot instance based on the input model file, and gets a roboid instance. The roboid instance has the instances of all the devices specified in the model file, and an array is created by method `createDeviceMemory` method for storing the data of each device. Method `getProtocol` returns the reference to the Protocol instance contained in the roboid instance and the reference is used in communication with the hardware device.

```
roboid =
createRobot(modelFileName).getRoboids().get(0);
roboid.createDeviceMemory();
protocol = roboid.getProtocol();
```

Figure 3.2 Getting a roboid instance

Next, a device adaptor is registered as in Figure 3.3. The method of the registered adapter is called when data have been received from the hardware device, but because the method is basically empty, it does not carry out any specific task. In order to set a specific task to be carried out when the device has received data, just inherit and implement the method of class `DeviceAdapter`. For devices related to sensors, however, the device adaptor must be registered because such devices do not work without the registration of the device adaptor.

```
for(Device device : roboid.getDevices())
       device.addDeviceListener(new DeviceAdapter());
```

Figure 3.3 Registering the device adaptor

Method `getDevices` in Figure 3.4 gets the reference to the instance of each device from the roboid instance. Because each roboid has different devices, method `getDevices` is declared as an abstract method as in Figure 3.5 and should be implemented a class that inherits class `AbstractRoboid` as in class `Pelican`.

```
getDevices(roboid);
```

Figure 3.4 Getting references to device instances

```
protected abstract void getDevices(Roboid roboid);
```

Figure 3.5 Defining method `getDevices`

Figure 3.6 is the part of creating a socket for communication. If a socket has not been created or it is closed, a new datagram socket is created. In addition, the IP address of the local host is obtained in order to create and send a datagram packet.

```
try
{
       if(recvSocket == null || recvSocket.isClosed())
              recvSocket = new DatagramSocket();
       ia = InetAddress.getByName(LOCAL_ADDRESS);
} catch (Exception e)
{
       return;
}
```

Figure 3.6 Setting a communication socket

The process in Figure 3.7 executes an execution file for driving a hardware device, and waits for 500msec during the execution. Because each roboid has different execution files for driving hardware devices, method `executeDriver` is declared as an abstract method as in Figure 3.8, and should be implemented in classes that inherit class `AbstractRoboid`. Execution files for driving hardware devices and related DLL files are grouped by roboid in the roboid folder. For example, those for the Pelican roboid are contained in the **roboid/pelican** folder.

```
process = executeDriver();

try
{
        Thread.sleep(500);
} catch (InterruptedException e)
{
        e.printStackTrace();
}
```

Figure 3.7 Executing an execution file for driving a hardware device

```
protected abstract Process executeDriver();
```

Figure 3.8 Defining method `executeDriver`

In the last part of the constructor of class `AbstractRoboid`, a thread is created and executed for communication as in Figure 3.9. Here, field `ACTIVE` is used to close the created thread. That is, the `while` statement within the thread is repeated while field `ACTIVE` is `true`, and if field `ACTIVE` becomes `false`, the process exits the `while` statement and the thread ends.

```
ACTIVE = true;
new UpLinkThread().start();
```

Figure 3.9 Executing a thread for communication

Method `dispose` in Figure 3.10 is called in order to end several instances created internally when the roboid is closed. That is, it ends the thread created for communication, closes network communication, and terminates execution files for driving hardware devices.

```
public void dispose()
{
        ACTIVE = false;
        recvSocket.close();
        process.destroy();
        while(!recvSocket.isClosed())
        {}
        System.out.println("deactivated !!!");
}
```

Figure 3.10 Ending internal instances

Method `createRobot` in Figure 3.11, which is called in the constructor of class `AbstractRoboid`, reads the model file in `modelFileName`, creates a Robot instance, and return it. The model file is stored in the form of XMI, and a Robot instance is created by EMF of Eclipse.

```
private Robot createRobot(String modelFileName)
{
	// Create a resource set to hold the resources.
	ResourceSet resourceSet = new ResourceSetImpl();

	// Register the appropriate resource factory to handle 
all file extentions.
	resourceSet.getResourceFactoryRegistry().getExtensionTo
FactoryMap().put
	(Resource.Factory.Registry.DEFAULT_EXTENSION,
	 new XMIResourceFactoryImpl());

	// Register the package to ensure it is available 
during loading.
	resourceSet.getPackageRegistry().put(RobotPackage.eNS_U
RI, RobotPackage.eINSTANCE);

	URI uri = URI.createFileURI(getBasePath() + 
modelFileName);

	try
	{
		Resource resource = resourceSet.getResource(uri, 
true);
		return (Robot)(resource.getContents().get(0));
	}
	catch (RuntimeException exception)
	{
		System.out.println("Problem loading " + uri);
		exception.printStackTrace();
	}
	return null;
}
```

Figure 3.11 Creating a Robot instance

The method `getOsName` in Figure 3.12 is for identifying the OS type of the computer in which Roboid Library is running. Currently it distinguishes only Windows XP and Vista.

```
public String getOsName()
{
	String osPropertyName =
				System.getProperty("os.name");
	String osName = "xp"; // default : xp
	if(osPropertyName.toLowerCase().indexOf("vista") !=
-1)
		osName = "vista";

	return osName;
}
```

Figure 3.12 Identifying the computer OS type

The method `getBasePath` in Figure 3.13 returns the absolute path of project **org.roboid.library**.

```
public String getBasePath()
{
	String path = new File("").getAbsolutePath();
	return path + "/../" + Activator.PLUGIN_ID + "/";
}
```

Figure 3.13 Finding the absolute path of the project

The method `getUpLinkPort` and method `getDownLinkPort` in Figure 3.14 return, respectively, the port number for receiving data from hardware devices and the port number for sending data to hardware devices. These methods find internally empty ports and assign their numbers automatically.

```
public String getUpLinkPort()
{
	return "" + recvSocket.getLocalPort();
}

public int getDownLinkPort()
{
	return portDown;
}
```

Figure 3.14 Getting port numbers

The method `setController` in Figure 3.15 registers an instance that implemented interface `Controller` in order to call method `execute` every 20msec. Method execute in the registered instance is called whenever the thread created in class AbstractRoboid receives data through communication..

```
public void setController(Controller controller)
{
	this.controller = controller;
}
```

Figure 3.15 Registering an instance that implemented interface Controller

Now, let's examine the inside of the thread processing communication. As in Figure 3.16, the thread creates a datagram packet by allocating an array of 4048 bytes.

```
private byte[] buffer = new byte[4048];
DatagramPacket dp = new DatagramPacket(buffer, 4048);
```

Figure 3.16 Creating a datagram packet

In method run executed by a thread, the while statement is performed while field ACTIVE is true as in Figure 3.17. As in Figure 3.9, field ACTIVE is set true before the thread is created and is set false when the thread is closed as in Figure 3.10.

```
public void run()
{
	while(ACTIVE)
	{
		...
	}
}
```

Figure 3.17 Executing a thread

Figure 3.18 shows the part of receiving data from a hardware device through a socket. Data received using the datagram packet created earlier is written in the data array of each device by method setSimulacrum of interface Protocol. Let's get the port number of the datagram packet and use it in sending data to a hardware device.

```
recvSocket.receive(dp);
protocol.setSimulacrum(dp.getData(), true);
portDown = dp.getPort();
```

Figure 3.18 Receiving data from a hardware device

In Figure 3.19, when an instance implementing interface `Controller` has been registered, method `execute` in the registered instance is called. In addition, if data on a device are included in data received by method `updateDeviceState` in `Roboid` interface, an event is triggered for the corresponding device. The triggered event can be confirmed by method `e()` of the device.

```
if(controller != null) controller.execute(roboid);

roboid.updateDeviceState();
```

Figure 3.19 Executing method `execute` and triggering an event

Figure 3.20 is the part of sending data to a hardware device. If `portDown` is not -1, that is, data have been received from a hardware, data to be sent are obtained by method `getSimulacrum` in interface `Protocol`. In method `getSimulacrum`, a byte array (simulacrum) is created according to DMP (Device Map Protocol) by collecting values in the data array of each device. Lastly, a datagram packet to be sent is created using the IP address of the local host and the port number obtained earlier, and if the socket is opened, the datagram packet is sent.

```
if(portDown != -1)
{
       byte[] out = protocol.getSimulacrum();
       DatagramPacket packet = new DatagramPacket(out,
                              out.length, ia, portDown);

       if(!recvSocket.isClosed())
       {
              try
              {
                     recvSocket.send(packet);
              } catch (IOException e)
              {
                     e.printStackTrace();
              }
       }
}
```

Figure 3.20 Sending data to a hardware device

4. Class Pelican

Now, let's examine class Pelican implemented by inheriting class AbstractRoboid. As in Figure 4.1, various Device-type fields are defined in Figure 4.1. These fields store references to device instances, and are used to read or write data from/to devices by method read and method write in interface Device. For the meaning of each field, see Table 2.1.

```
public Device leftWing, rightWing;
public Device bill, led, eye, eyeTurn;
public Device speaker, microphone;
public Device hitLeftWing, hitRightWing, hitBill,
hitTable;
public Device cursor, leftButtonDown, middleButtonDown,
              rightButtonDown;
public Device leftButtonUp, middleButtonUp,
rightButtonUp,
              scrollUp, scrollDown;
```

Figure 4.1Fields for accessing devices

Figure 4.2 shows the constructor of class Pelican. The constructor calls the constructor of parent class AbstractRoboid, which gives the model file name of Pelican as a relative path from the project folder. Then, 'Pelican activated…' is displayed on the console view of Eclipse.

```
public Pelican()
{
        super("roboid/pelican/kr.robomation.physical.pelican.ro
bot");
        System.out.println("Pelican activated...");
}
```

Figure 4.2 The constructor of class Pelican

Figure 4.3 implements method executeDriver declared as an abstract method in class AbstractRoboid. This method carries out the task of executing execution files for driving hardware devices. The way of implementation may be different among hardware devices but the form is almost identical. Because the execution file of the Pelican roboid is different according to OS, the absolute path of the execution file is obtained by getting the absolute path of the project folder and OS type. Moreover, because a port number for hardware devices to send data needs to be given, the port number of the created socket is obtained by method `getUpLinkPort`. The part of actually executing the execution file uses method `exec` of Eclipse Runtime class. When the execution file is executed, the port number and the volume size are transferred as arguments. Here, the volume size was set to 128. The reference to Process returned when the execution file was executed is used to terminate the execution file in method `dispose` in Figure 3.10.

```
protected Process executeDriver()
{
        String virtual = getBasePath() +
"roboid/pelican/" +
  getOsName() + "/roboid.pelican.usb.exe";
        String portUp = getUpLinkPort();

        Runtime r = Runtime.getRuntime();
        Process process = null;
        try
        {
                process = r.exec(new String[]{virtual,
portUp, "128"});
        }catch(Exception e)
        {
                System.out.println("Error executing
virtual driver");
        }
        return process;
}
```

Figure 4.3 Implementing method `executeDriver`

Figure 4.4 implements method `getDevices` declared as an abstract method in class `AbstractRoboid`. The reference to a device instance created based on the contents of the model file is obtained by method `findDevice`. In method `findDevice`, the name of a device specified in the model file is provided, and if there is an instance of the device with the name exists, the method returns the reference to the instance or returns `null`. Device instances obtained by method `findDevice` in this way are used to read or write data from/to each device as in the exercises presented above.

```
protected void getDevices(Roboid roboid)
{
	rightWing = roboid.findDevice("RightWing");
	leftWing = roboid.findDevice("LeftWing");
	bill = roboid.findDevice("Lip");
	led = roboid.findDevice("DualLed");
	eye = roboid.findDevice("EyePosition");
	eyeTurn = roboid.findDevice("EyeTurn");
	speaker = roboid.findDevice("Speaker");

	microphone = roboid.findDevice("Microphone");
	hitRightWing = roboid.findDevice("HitRightWing");
	hitLeftWing = roboid.findDevice("HitLeftWing");
	hitBill = roboid.findDevice("HitBill");
	hitTable = roboid.findDevice("HitTable");
	cursor = roboid.findDevice("Cursor");
	leftButtonDown =
roboid.findDevice("LeftBtnDown");
	middleButtonDown =
roboid.findDevice("MiddleBtnDown");
	rightButtonDown =
roboid.findDevice("RightBtnDown");
	leftButtonUp = roboid.findDevice("LeftBtnUp");
	middleButtonUp =
roboid.findDevice("MiddleBtnUp");
	rightButtonUp = roboid.findDevice("RightBtnUp");
	scrollUp = roboid.findDevice("ScrollUp");
	scrollDown = roboid.findDevice("ScrollDown");
}
```

Figure 4.4 Implementing method `getDevices`

5. Interfaces and methods of Roboid Library

Now let's examine various interfaces and methods available in Roboid Library. Most of methods provided in Roboid Library are implemented to use Roboid Library internally or for Roboid Studio, so we need only a number of methods as presented in the exercises above in order to drive hardware devices using Roboid Library. That is, hardware devices can be driven simply using a few methods including the creation of an instance of class `Roboid`, reading and writing from/to hardware devices by method `read` and method `write`, and terminating an instance by method `dispose`. In some cases, an instance implements interface `Controller` and is registered by method `setController`, and then method `execute` is implemented and called every 20msec.

In order to utilize Roboid Library more sophisticatedly, we explain all interfaces and methods available. What we should know first is the composition of the model file (see Figure 2.3). The Robot of the model file is composed of Roboid and Control. Because Control is used in Roboid Studio but not in Roboid Library, the model file in Roboid Library excluded Control. A roboid has a hierarchical structure consisting of Protocol and Device, and Device is divided into Sensor, Effector, Command, and Event according to the characteristic of device. Here, Sensor and Event are devices that obtain data from hardware devices. Sensor is a device like microphone, and Event is an event like mouse click. Effector and Command are devices that write data into hardware devices. Effector is an effector like wing and speaker, and Command is a command like eye shape. Depending on the way of handling data, Device can be divided into Sensor/Effector and Event/Command. In case of Sensor/Effector, if data are written in the same device simultaneously or several times during the cycle of exchanging data, the sum of the data is written in the device. On the contrary, Event/Command erases existing data and writes new data, so the data written lastly are sent through communication.

Robot, Roboid, Protocol and Device in the model file can have properties. Robot has properties such as Comment, Name, Provider, Standard, and Version, and Roboid has properties such as Address, Comment, Id, Name, Provider, and Version. In addition, Protocol has properties Buffer Size, Comment, Name, and Version. Device has slightly different properties depending on whether it is Sensor, Effector, Event or Command, but the common properties are Comment, Data Size, Data Type, Default, Max, Min, and Name. In addition to these common properties, Sensor has property Throttle, and Effector has Sustain and Throttle. The values of these properties can be set in the model editor when the model is designed, or can be read or set by methods provided by Roboid Library. However, because execution files for driving hardware devices are created and communication is performed based on the composition of the model file and the set values of properties, the model should not be changed in Roboid Library. If the model file has been changed, data received through communication may be misinterpreted and cause an unwanted operation.

.

From now on, we will examine interfaces and methods provided in Roboid Library one by one.

6. Interface **`NamedElement`**

`NamedElement` has an interface as in Figure 6.1. The interfaces of `Robot`, `Roboid`, `Protocol`, `Device`, etc. inherit interface `NamedElement`.

```
public interface NamedElement
{
        String getName();
        void setName(String value);
        String getLiteral();
        void setLiteral(String value);
        String getComment();
        void setComment(String value);
        NamedElement getParent();
        List<NamedElement> getChildren();
        boolean equalsContent(Object obj);
}
```

Figure 6.1 Interface `NamedElement`

```
String getName();
```

Return the name designated in the robot model.

Return	name

```
void setName(String value);
```

Set the name of a robot, roboid, protocol, device, etc.

Parameters	**`value`**	name

```
String getLiteral();
```

Return the name designated in the robot model. Name can be changed by the user, and Literal is an unchangeable unique name defined by the robot developer.

Return	unique name

```
void setLiteral(String value);
```

Set the unique name of a robot, roboid, protocol, device, etc.

Parameters	**`value`**	unique name

```
String getComment();
```

Return a comment.

Return	comment

```
void setComment(String value);
```

Set a comment on robot, roboid, protocol, device, etc.

Parameters	**`value`**	comment

```
NamedElement getParent();
```

Return the parent object in the hierarchical structure of the robot model. That is, it returns Roboid for a device or a protocol, and Robot for a roboid.

Return	parent object

```
List<NamedElement> getChildren();
```

Return a list of child objects in the hierarchical structure of the robot model. That is, it returns all roboids included for a robot, and all devices included for a roboid.

Return	A list of child objects

```
boolean equalsContent(Object obj);
```

Determine whether the content of the object is the same as that of the given object (obj). That is, for a robot, it determines whether not only the name and provider of the robot but also the content of all the roboids are the same.

Parameters	**`obj`**	the object to be compared
Return	`true` if the content is the same, or `false`	

7. Interface `Storable`

`Storable` has an interface as in Figure 7.1.

```
public interface Storable
{
        void createDeviceMemory();
        void clearDeviceMemory();
}
```

Figure 7.1 Interface `Storable`

```
void createDeviceMemory();
```

Allocate an array for storing data in each device at the data size of the device specified in the robot model.

```
void clearDeviceMemory();
```

Fill the data arrays of all devices forming the robot with the default of each device.

8. Interface `Findable`

`Findable` has an interface as in Figure 8.1.

```
public interface Findable
{
        Device findDevice(String name);
        Roboid findRoboid(String name);
        Roboid findRoboidById(String id);
        Roboid findRoboidByUid(String uid);
}
```

Figure 8.1 Interface `Findable`

```
Device findDevice(String name);
```

Find the device whose name is the same as the given `name` among devices forming the robot, and return the instance. Because devices form a hierarchical structure under a roboid from the viewpoint of the robot, the name of a device uses "." as in "roboid.device" (e.g. `Pelican.RightWing`). Return `null` if there is no device whose name is the same as the given `name`.

Parameters	**`name`**	device name
Return	a device instance	

```
Roboid findRoboid(String name);
```

Find the roboid whose name is the same as the given `name` among roboids forming the robot, and return the instance. Return `null` if there is no roboid whose name is the same as the given `name`.

Parameters	**`name`**	roboid name
Return	a roboid instance	

```
Roboid findRoboidById(String id);
```

Find the roboid whose ID is the same as the given `id` among roboids forming the robot, and return the instance. Here, `id` means the ID of a roboid. Return `null` if there is no roboid whose ID is the same as the given `id`.

Parameters	**`id`**	roboid ID
Return	a roboid instance	

```
Roboid findRoboidByUid(String uid);
```

Find the roboid whose UID is the same as the given `uid` among roboids forming the robot, and return the instance. Here, `uid` means the UID of a roboid. Return `null` if there is no roboid whose UID is the same as the given `uid`.

Parameters	**`uid`**	roboid UID
Return	a roboid instance	

9. Interface `Robot`

`Robot` has an interface as in Figure 9.1 by inheriting interface `Storable` and `Findable`.

```
public interface Robot extends NamedElement, Storable,
Findable
{
	String getProvider();
	void setProvider(String value);
	String getVersion();
	void setVersion(String value);
	List<Roboid> getRoboids();
	List<Control> getControls();
	String getStandard();
	void setStandard(String value);
	Protocol getProtocol();
}
```

Figure 9.1 Interface `Robot`

```
String getProvider();
```

Return the provider name specified in the robot model.

Return	provider name

```
void setProvider(String value);
```

Set the provider name.

Parameters	**`value`**	provider name

```
String getVersion();
```

Return the software version.

Return	software version

```
void setVersion(String value);
```

Set the software version.

Parameters	**`value`**	software version

```
List<Roboid> getRoboids();
```

Return a list of Roboid instances forming the robot.

Return	a list of Roboid instance

```
List<Control> getControls();
```

Return a list of Control instances forming the robot. Because Roboid Library does not have Control, it returns an empty list.

Return	a list of Control instances

```
String getStandard();
```

Return the name of the standard that the software follows.

Return	standard name (Example: RUPI 2.0)

```
void setStandard(String value);
```

Set the name of the standard that the software follows.

Parameters	**`value`**	standard name (Example: RUPI 2.0)

```
Protocol getProtocol();
```

Return a protocol instance forming the robot.

Return	a protocol instance

10. Interface `Simulacra`

`Simulacra` is in charge of setting and processing simulacrum when communicating by the device map protocol, and has an interface as in Figure 10.1.

```
public interface Simulacra
{
	int setDeviceMap(int index, byte[] deviceMap,
boolean isMaster);
	void setPayload(ByteArrayInputStream simulacrum,
boolean isMaster);
	void getSimulacrum(ByteArrayOutputStream
deviceMap, ByteArrayOutputStream payload);
	boolean isReceived();
	boolean canSend();
	void updateDeviceState();
}
```

Figure 10.1 Interface `Simulacra`

```
int setDeviceMap(int index, byte[] deviceMap, boolean
isMaster);
```

[Uplink] Set a bit of the device map for data transmitted from a hardware device. In Roboid Library, `isMaster` is always `true`.

Parameters	**`index`**	an index of the corresponding device in the model
	`deviceMap`	device map array
	`isMaster`	whether to be the master generating data
Return	an index of the next device in the model	

```
void setPayload(ByteArrayInputStream simulacrum,
boolean isMaster);
```

[Uplink] Copy payload data transmitted from a hardware device to the data array of the corresponding device. In Roboid Library, `isMaster` is always `true`.

Parameters	**`simulacrum`**	transmitted simulacrum
	`isMaster`	whether to be the master generating data

```
void getSimulacrum(ByteArrayOutputStream deviceMap,
ByteArrayOutputStream payload);
```

[Downlink] Get a device map and payload for sending data to a hardware device.

Parameters	**`deviceMap`**	device map
	`payload`	payload

```
boolean isReceived();
```

[Uplink] Ask whether the corresponding bit of the device map is 1 in the data transmitted from a hardware device. Return `true` if data for the corresponding device have been transmitted.

Return	on/off of the device bit

```
boolean canSend();
```

[Downlink] Ask whether the corresponding bit of the device map will be 1 when data are sent to a hardware device. Return `true` if there are data to be sent to the corresponding device.

Return	on/off of the device bit

```
void updateDeviceState();
```

Prepare data of each device and trigger an event. Whether an event has been triggered can be checked by method `e()`.

11. Interface `Roboid`

`Roboid` has an interface as in Figure 11.1.

```
public interface Roboid extends NamedElement, Storable,
Findable, Simulacra
{
	String getId();
	void setId(String value);
	String getUid();
	void setUid(String value);
	List<Roboid> getRoboids();
	Protocol getProtocol();
	void setProtocol(Protocol value);
	List<Device> getDevices();
	String getVersion();
	void setVersion(String value);
	String getProvider();
	void setProvider(String value);
	String getAddress();
	void setAddress(String value);
	Roboid getHostRoboid();
	List<String> getDeviceNames(List<String> names,
String prefix);
	Robot getRobot();
}
```

Figure 11.1 Interface `Roboid`

```
String getId();
```

Return the ID of the roboid designated in the robot model.

Return	roboid ID

```
void setId(String value);
```

Set the ID of a roboid

Parameters	**`value`**	roboid ID

```
String getUid();
```

Return the UID of the roboid designated in the robot model. A robot model may have multiple roboids with the same ID, and they are distinguished by their UID.

Return	roboid UID

```
void setUid(String value);
```

Set the UID of a roboid.

Parameters	**`value`**	roboid UID

```
List<Roboid> getRoboids();
```

In the hierarchical structure of a robot model, a roboid can be a child of another roboid. This method returns a list of child roboid instances.

Return	a list of roboid instances

```
Protocol getProtocol();
```

Return a Protocol instance forming the roboid.

Return	a Protocol instance

```
void setProtocol(Protocol value);
```

Set the protocol.

Parameters	**`value`**	a protocol instance

```
List<Device> getDevices();
```

Return a list of Device (Sensor, Effector, Command, Event) instances forming the roboid.

Return	a list of device instances

```
String getVersion();
```

Return the software version.

Return	software version

```
void setVersion(String value);
```

Set the software version.

Parameters	**`value`**	software version

```
String getProvider();
```

Return the name of the provider specified in the robot model.

Return	provider name

```
void setProvider(String value);
```

Set the name of the provider.

Parameters	**`value`**	provider name

```
String getAddress();
```

Return the address specified in the robot model. When the multiple instances of the same Roboid are executed at the same time, there should be a value for identifying them and address is used as the value.

Return	address

```
void setAddress(String value);
```

Set the address.

Parameters	**value**	address

```
Roboid getHostRoboid();
```

Addon has a roboid that becomes the host, and this method returns the host roboid instance. Host roboid means a roboid chosen among roboids with the model ID set in the host item in Manifest.mf when the Addon component is developed. This method returns `null` if host roboid has not been set. Because Roboid Library does not use components, it does not need method `getHostRoboid`.

Return	a host roboid instance

```
List<String> getDeviceNames(List<String> names, String
prefix);
```

Add a prefix to the name of each device forming the roboid, and add the name to the list `names`. The returned list is the same as the list `names`. If there is a child roboid in the hierarchical structure, it is distinguished by “.” as in “Roboid.childRoboid.device”.

Parameters	**names**	a list of device names to which a prefix was added
	prefix	prefix
Return	a list of device names to which a prefix was added	

Sample code

```
List<String> names = new ArrayList();
names.add("'value'");
for(Roboid roboid : robot.getRoboids())
{
	names = roboid.getDeviceNames(names,
roboid.getName() + ".");
}
```

Sample results

If the name of the roboid is Pelican, the list `names` includes `'value'`, `Pelican.RightWing`, `Pelican.LeftWing`, etc.

```
Robot getRobot();
```

Return the robot instance containing the roboid.

Return	a robot instance

12. Interface `Protocol`

`Protocol` is in charge of setting necessary properties when communication is made by the device map protocol, and has an interface as in Figure 12.1..

```
public interface Protocol extends NamedElement
{
        String getVersion();
        void setVersion(String value);
        int getBufferSize();
        void setBufferSize(int value);
        int getRemainingBuffer();
        void setRemainingBuffer(int value);
        byte[] getSimulacrum();
        void setSimulacrum(byte[] simulacrum, boolean
isMaster);
        void clearBuffer();
        void setEvents();
        int getBufferId();
        public static final int HEADER_SIZE = 20;
}
```

Figure 12.1 Interface Protocol

```
String getVersion();
```

Return the protocol version.

Return	protocol version

```
void setVersion(String value);
```

Set the protocol version.

Parameters	**`value`**	protocol version

```
int getBufferSize();
```

Return the buffer size specified in the robot model.

Return	buffer size

```
void setBufferSize(int value);
```

Set the buffer size.

Parameters	**`value`**	buffer size

```
int getRemainingBuffer();
```

Return the size of remaining buffer.

Return	the size of remaining buffer

```
void setRemainingBuffer(int value);
```

Set the size of remaining buffer for buffer control.

Parameters	**`value`**	the size of remaining buffer

```
byte[] getSimulacrum();
```

[Downlink] Return the simulacrum of a roboid containing protocol in order to send data to a hardware device.

Return	simulacrum

```
void setSimulacrum(byte[] simulacrum, boolean
isMaster);
```

[Uplink] Copy data transmitted from a hardware device to the data array of the corresponding device. In Roboid Library, `isMaster` is always `true`.

Parameters	**`simulacrum`**	simulacrum
	`isMaster`	whether to be the master generating data

```
void clearBuffer();
```

Command to empty the buffer of a hardware device for buffer control.

```
void setEvents();
```

Trigger events in all the devices.

```
int getBufferId();
```

Whenever the buffer of a hardware device is emptied the ID of the buffer is increased, and this method returns the current buffer ID.

Return	buffer ID

13. Interface `Device`

`Device`, which abstracted `Sensor`, `Effector`, `Command`, and `Event` devices, has an interface as in Figure 13.1. When data are written or read into/from a device, various forms of write or read method can be used, but the process works only for methods whose data type is compatible with the data type set in the model. That is, if the data type is `BYTE`, `SHORT`, `INTEGER` or `FLOAT`, data of integer or float type can be written or read, but data of string or image type cannot be written or read. In addition, if the data type is `STRING` only string-type data can be written or read, and if it is `IMAGE` only image-type data can be written or read.

```
public interface Device extends NamedElement, Storable,
Simulacra
{
	int getDataSize();
	void setDataSize(int value);
	DataType getDataType();
	void setDataType(DataType value);
	int getMax();
	float getMaxFloat();
	String getMaxString();
	void setMax(String value);
	int getMin();
	float getMinFloat();
	String getMinString();
	void setMin(String value);
	int getDefault();
	float getDefaultFloat();
	String getDefaultString();
	ImageData getDefaultImageData();
	AccessType getAccess();
	void setDefault(String value);
	void setAccess(AccessType newAccess);
	boolean isProxy();
	void setProxy(boolean value);
	boolean write();
	boolean write(int data);
	boolean write(int index, int data);
	boolean write(int[] data);
	boolean write(float data);
	boolean write(int index, float data);
	boolean write(float[] data);
	boolean write(String text);
	boolean write(int index, String text);
```

```
        boolean write(String[] text);
        boolean write(ImageData imageData);
        boolean write(int index, ImageData imageData);
        boolean write(ImageData[] imageData);
        boolean writeFloat(float data);
        boolean writeFloat(int index, float data);
        boolean writeFloat(float[] data);
        boolean writeString(String text);
        boolean writeString(int index, String text);
        boolean writeString(String[] text);
        boolean writeImageData(ImageData imageData);
        boolean writeImageData(int index, ImageData
imageData);
        boolean writeImageData(ImageData[] imageData);
        int read();
        int read(int index);
        int read(int[] data);
        int read(float[] data);
        int read(String[] data);
        int read(ImageData[] data);
        float readFloat();
        float readFloat(int index);
        int readFloat(float[] data);
        String readString();
        String readString(int index);
        int readString(String[] text);
        ImageData readImageData();
        ImageData readImageData(int index);
        int readImageData(ImageData[] imageData);
        boolean e();
        void setEvent();
        void setFired();
        void addDeviceListener(DeviceListener listener);
        void removeDeviceListener(DeviceListener
listener);
        List<DeviceListener> getDeviceListeners();
        Device getProxyFor();
}
```

Figure 13.1 Interface `Device`

```
int getDataSize();
```

Return the size of the data array of the device.

Return	device data array size

```
void setDataSize(int value);
```

Set the size of the data array of the device.

Parameters	**`value`**	device data array size

```
DataType getDataType();
```

Return the data type of the device. The data type can be `BYTE`, `SHORT`, `INTEGER`, `FLOAT`, `STRING`, or `IMAGE`.

Return	device data type

```
void setDataType(DataType value);
```

Set the data type of the device.

Parameters	**`value`**	device data type

```
int getMax();
```

Return the integer-type maximum value of device data. Return 0 if the data type is `STRING` or `IMAGE`.

Return	the maximum value of device data

```
float getMaxFloat();
```

Return the float-type maximum value of device data. Return 0 if the data type is `STRING` or `IMAGE`.

Return	the maximum value of device data

```
String getMaxString();
```

Return the string-type maximum value of device data. Return “0” if the data type is `STRING` or `IMAGE`.

Return	the maximum value of device data

```
void setMax(String value);
```

Set the maximum value of device data. The maximum value is not set if the data type is `STRING` or `IMAGE`.

Parameters	**`value`**	the maximum value of device data

```
int getMin();
```

Return the integer-type minimum value of device data. Return 0 if the data type is `STRING` or `IMAGE`.

Return	the minimum value of device data

```
float getMinFloat();
```

Return the float-type minimum value of device data. Return 0 if the data type is `STRING` or `IMAGE`.

Return	the minimum value of device data

```
String getMinString();
```

Return the string-type minimum value of device data. Return “0” if the data type is `STRING` or `IMAGE`.

Return	the minimum value of device data

```
void setMin(String value);
```

Set the minimum value of device data. The minimum value is not set if the data type is `STRING` or `IMAGE`.

Parameters	**`value`**	the minimum value of device data

```
int getDefault();
```

Return the integer-type default of device data. Return 0 if the data type is `STRING` or `IMAGE`.

Return	the default of device data

```
float getDefaultFloat();
```

Return the float-type default of device data. Return 0 if the data type is `STRING` or `IMAGE`.

Return	the default of device data

```
String getDefaultString();
```

Return the string-type default of device data. Return `null` if the data type is `IMAGE`.

Return	the default of device data

```
ImageData getDefaultImageData();
```

Return the image-type default of device data. Return `null` always.

Return	`null`

```
void setDefault(String value);
```

Set the default of device data. The default is not set if the data type is `IMAGE`.

Parameters	**`value`**	the default of device data

```
AccessType getAccess();
```

Return the level of access to the device. `AccessType` can be `PRIVATE` or `PUBLIC`. This is used limitedly in Version 1.4.1, and will be improved in future versions.

Return	device access level

```
boolean isProxy();
```

Determine whether the device is set as a proxy.

Return	`true` if the device is set as a proxy, or `false`.

```
void setProxy(boolean value);
```

Set the device as a proxy if the `value` is `true`, or set that the device is not a proxy.

Parameters	**`value`**	whether to be set as a proxy

```
boolean write();
```

Trigger an event although data are not written in the device.

Return	true always

```
boolean write(int data);
```

Write `data` at the position of index 0 in the data array of the device, and trigger an event. `data` is not written if the data type is `STRING` or `IMAGE`.

Parameters	**`data`**	data
Return	`false` if the size of the device data array is 0, or `true`	

```
boolean write(int index, int data);
```

Write `data` at the position of the given `index` in the data array of the device, and trigger an event. `data` is not written if the data type is `STRING` or `IMAGE`.

Parameters	**`index`**	an index of device data memory
	`data`	data
Return	`false` if the `index` is equal to or larger than the size of the device data array of the device, or `true`	

```
boolean write(int[] data);
```

Write a `data` array in the device data array, and trigger an event. If the size of the device data array is smaller than the size of the `data` array, the method writes `data` as much as the size of the device data array, or writes the given `data` array and fills the remaining with 0. `data` are not written if the data type is `STRING` or `IMAGE`.

Parameters	**`data`**	data
Return	`true` if `data` can be written, or `false`	

```
boolean write(float data);
```

Write `data` at the position of index 0 in the data array of the device, and trigger an event. `data` is not written if the data type is `STRING` or `IMAGE`.

Parameters	**`data`**	data
Return	`false` if the size of the device data array is 0, or `true`	

```
boolean write(int index, float data);
```

Write `data` at the position of the given `index` in the data array of the device, and trigger an event. `data` is not written if the data type is `STRING` or `IMAGE`.

Parameters	**`index`**	an index of device data memory
	`data`	data
Return	`false` if the `index` is equal to or larger than the size of the data array of the device, or `true`	

```
boolean write(float[] data);
```

Write a `data` array in the device data array, and trigger an event. If the size of the device data array is smaller than the size of the `data` array, the method writes `data` as much as the size of the device `data` array, or writes the given `data` array and fills the remaining with 0. `data` are not written if the data type is `STRING` or `IMAGE`.

Parameters	**`data`**	data
Return	`true` if `data` can be written, or `false`	

```
boolean write(String text);
```

Write string data `text` at the position of index 0 in the data array of the device, and trigger an event. `text` is not written if data type is not `STRING`.

Parameters	**`text`**	string data
Return	`false` if the text is `null`, or `true`	

```
boolean write(int index, String text);
```

Write string data `text` at the position of the given `index` in the data array of the device, and trigger an event. `text` is not written if the data type is not `STRING`.

Parameters	**`index`**	an index of device data memory
	`text`	data
Return	`false` if the `index` is equal to or larger than the size of the data array of the device, or `true`	

```
boolean write(String[] text);
```

Write a `text` array in the device data array, and trigger an event. If the size of the device data array is smaller than the size of the `text` array, the method writes `text` as much as the size of the device data array, or writes the given `text` array and fills the remaining with “”. `text` are not written if the data type is not `STRING`.

Parameters	**`text`**	data
Return	`true` if `text` can be written, or `false`	

```
boolean write(ImageData imageData);
```

Write `imageData` at the position of index 0 in the data array of the device, and trigger an event. `imageData` is not written if the data type is not `IMAGE`.

Parameters	**`imageData`**	image data
Return	`false` if `imageData` is `null`, or `true`	

```
boolean write(int index, ImageData imageData);
```

Write `imageData` at the position of the given `index` in the data array of the device, and trigger an event. `imageData` is not written if the data type is not `IMAGE`.

Parameters	**`index`**	an index of device data memory
	`imageData`	data
Return	`false` if the `index` is equal to or larger than the size of the data array of the device, or `true`	

```
boolean write(ImageData[] imageData);
```

Write a `imageData` array in the device data array, and trigger an event. If the size of the device data array is smaller than the size of the `imageData` array, the method writes `imageData` as much as the size of the device data array, or writes the given `imageData` array and fills the remaining with `null`. `imageData` are not written if the data type is not `IMAGE`.

Parameters	**`imageData`**	data
Return	`true` if `imageData` can be written, or `false`	

```
boolean writeFloat(float data);
```

Write `data` at the position of index 0 in the data array of the device, and trigger an event. `data` is not written if the data type is `STRING` or `IMAGE`.

Parameters	**`data`**	data
Return	`false` if the size of the device data array is 0, or `true`	

```
boolean writeFloat(int index, float data);
```

Write `data` at the position of the given `index` in the data array of the device, and trigger an event. `data` is not written if the data type is `STRING` or `IMAGE`.

Parameters	**`index`**	an index of device data memory
	`data`	data
Return	`false` if the `index` is equal to or larger than the size of the data array of the device, or `true`	

```
boolean writeFloat(float[] data);
```

Write a `data` array in the device data array, and trigger an event. If the size of the device data array is smaller than the size of the `data` array, the method writes `data` as much as the size of the device data array, or writes the given `data` array and fills the remaining with 0. `data` are not written if the data type is STRING or IMAGE.

Parameters	**`data`**	data
Return	`true` if `data` can be written, or `false`	

```
boolean writeString(String text);
```

Write string data `text` at the position of index 0 in the data array of the device, and trigger an event. `text` is not written if data type is not `STRING`.

Parameters	**`text`**	string data
Return	`false` if the `text` is `null`, or `true`	

```
boolean writeString(int index, String text);
```

Write string data `text` at the position of the given `index` in the data array of the device, and trigger an event. `text` is not written if the data type is not `STRING`.

Parameters	**`index`**	an index of device data memory
	`text`	data
Return	`false` if the `index` is equal to or larger than the size of the data array of the device, or `true`	

```
boolean writeString(String[] text);
```

Write a `text` array in the device data array, and trigger an event. If the size of the device data array is smaller than the size of the `text` array, the method writes `text` as much as the size of the device data array, or writes the given `text` array and fills the remaining with “”. `text` are not written if the data type is not `STRING`.

Parameters	**`text`**	data
Return	`true` if `text` can be written, or `false`	

```
boolean writeImageData(ImageData imageData);
```

Write `imageData` at the position of index 0 in the data array of the device, and trigger an event. `imageData` is not written if the data type is not `IMAGE`.

Parameters	**`imageData`**	image data
Return	`false` if `imageData` is `null`, or `true`	

```
boolean writeImageData(int index, ImageData imageData);
```

Write `imageData` at the position of the given `index` in the data array of the device, and trigger an event. Data are not written if the data type is not `IMAGE`.

Parameters	**`index`**	an index of device data memory
	`imageData`	data
Return	`false` if the `index` is equal to or larger than the size of the data array of the device, or `true`	

```
boolean writeImageData(ImageData[] imageData);
```

Write an `imageData` array in the device data array, and trigger an event. If the size of the device data array is smaller than the size of the `imageData` array, the method writes `imageData` as much as the size of the device data array, or writes the given `imageData` array and fills the remaining with `null`. `imageData` are not written if the data type is not `IMAGE`.

Parameters	**`imageData`**	data
Return	`true` if `imageData` can be written, or `false`	

```
int read();
```

Return integer-type data at the position of index 0 in the data array of the device when the data type is neither `STRING` nor `IMAGE`.

Return	0 if the size of the device data array is 0, or the value at the position of index 0 in the device data array if the size is 1 or larger. 0 if the data type is `STRING` or `IMAGE`.

```
int read(int index);
```

Return integer-type data at the position of the given `index` in the data array of the device when the data type is neither `STRING` nor `IMAGE`.

Parameters	**`index`**	an index of the device data array
Return	0 if the `index` is equal to or larger than the size of the data array of the device, or the read data. 0 if the data type is `STRING` or `IMAGE`.	

```
int read(int[] data);
```

Copy the device data array to the given `data` array and return the size of the copied data when the data type is neither `STRING` nor `IMAGE`. If the size of the device data array is larger than the size of the given `data` array, this method copies as much as the size of the given `data` array, or copies as much as the size of the device data array and fills the remaining with 0.

Parameters	**`data`**	an array for getting device data
Return	the size of the copied data. 0 if the data type is `STRING` or `IMAGE`.	

```
int read(float[] data);
```

Copy the device data array to the given `data` array and return the size of the copied data when the data type is neither `STRING` nor `IMAGE`. If the size of the device data array is larger than the size of the given `data` array, this method copies as much as the size of the given `data` array, or copies as much as the size of the device data array and fills the remaining with 0.

Parameters	**`data`** an array for getting device data
Return	the size of the copied data. 0 if the data type is `STRING` or `IMAGE`.

```
int read(String[] data);
```

Copy the device data array to the given `data` array and return the size of the copied data when the data type is `STRING`. If the size of the device data array is larger than the size of the given `data` array, this method copies as much as the size of the given `data` array, or copies as much as the size of the device data array and fills the remaining with "".

Parameters	**`data`** an array for getting device data
Return	the size of the copied data. 0 if the data type is not `STRING`.

```
int read(ImageData[] data);
```

Copy the device data array to the given `data` array and return the size of the copied data when the data type is `IMAGE`. If the size of the device data array is larger than the size of the given `data` array, this method copies as much as the size of the given `data` array, or copies as much as the size of the device data array and fills the remaining with `null`.

Parameters	**`data`** an array for getting device data
Return	the size of the copied data. 0 if the data type is not `IMAGE`.

```
float readFloat();
```

Return float-type data at the position of index 0 in the data array of the device when the data type is neither `STRING` nor `IMAGE`.

Return	0 if the device data array size is 0, or the value at the position of index 0 in the device data array. 0 if the data type is `STRING` or `IMAGE`.

```
float readFloat(int index);
```

Return a float-type data at the position of the given `index` in the data array of the device when the data type is neither `STRING` nor `IMAGE`.

Parameters	**`index`** an index of the device data array
Return	0 if the `index` is equal to or larger than the size of the data array of the device, or the read data. 0 if the data type is `STRING` or `IMAGE`.

```
int readFloat(float[] data);
```

Copy the device data array to the given `data` array and return the size of the copied data when the data type is neither `STRING` nor `IMAGE`. If the size of the device data array is larger than the size of the given `data` array, this method copies as much as the size of the given `data` array, or copies as much as the size of the device data array and fills the remaining with 0.

Parameters	**`data`**	an array for getting device data
Return	the size of the copied data. 0 if the data type is `STRING` or `IMAGE`.	

```
String readString();
```

Return string-type data at the position of index 0 in the data array of the device when the data type is `STRING`.

Return	"" if the device data array size is 0, or the value at the position of index 0 in the device data array. `null` if the data type is not `STRING`.

```
String readString(int index);
```

Return string-type data at the position of the given `index` in the data array of the device when the data type is `STRING`.

Parameters	**`index`**	an index of the device data array
Return	"" if the `index` is equal to or larger than the size of the data array of the device, or the read data. `null` if the data type is not `STRING`.	

```
int readString(String[] text);
```

Copy the device data array to the given `text` array and return the size of the copied data when the data type is `STRING`. If the size of the device data array is larger than the size of the given `text` array, this method copies as much as the size of the given `text` array, or copies as much as the size of the device data array and fills the remaining with "".

Parameters	**`text`**	an array for getting device data
Return	the size of the copied data. 0 if the data type is not `STRING`.	

```
ImageData readImageData();
```

Return the image at the position of index 0 in the data array of the device when the data type is `IMAGE`.

Return	null if the device data array size is 0, or the value at the position of index 0 in the device data array. `null` if the data type is not `IMAGE`.

```
ImageData readImageData(int index);
```

Return the image at the position of the given `index` in the data array of the device when the data type is `IMAGE`.

Parameters	**`index`**	an index of the device data array
Return	`null` if the `index` is equal to or larger than the size of the data array of the device, or the read data. `null` if the data type is not `IMAGE`.	

```
int readImageData(ImageData[] imageData);
```

Copy the device data array to the given `imageData` array and return the size of the copied data when the data type is `IMAGE`. If the size of the device data array is larger than the size of the given `imageData` array, this method copies as much as the size of the given `imageData` array, or copies as much as the size of the device data array and fills the remaining with `null`.

Parameters	**`imageData`**	an array for getting device data
Return	the size of the copied data. 0 if the data type is not `IMAGE`.	

```
boolean e();
```

Check whether a device event has occurred.

Return	`true` if an event has occurred, or `false`

```
void setEvent();
```

Trigger a synchronized event of the device. Even if method `setFired` has been called, method `e()` returns `false` until method `setEvent` is called.

```
void setFired();
```

Trigger an asynchronized event of the device. Method `setFired` is called internally when method `setPayload` or `write` is called. Then, method `setEvent` is called by method `updateDeviceState`.

```
void addDeviceListener(DeviceListener listener);
```

Add a device listener.

Parameters	**`listener`**	a listener to be added

```
void removeDeviceListener(DeviceListener listener);
```

Remove a device listener.

Parameters	**`listener`**	a listener to be removed

```
List<DeviceListener> getDeviceListeners();
```

Return a list of device listeners
Return a listener list

```
Device getProxyFor();
```

Return an instance of the connected device if the device is a proxy.
Return an instance of the connected device

14. Interfaces `Sensor`, `Effector`, `Command`, and `Event`

`SensoryDevice` and `MotoringDevice` are interfaces inheriting `Device`, `Sensor` and `Event` are interfaces inheriting `SensoryDevice`, and `Effector` and `Command` are interfaces inheriting `MotoringDevice`. When these interfaces are used, an instance of the corresponding device is obtained by `findDevice`, and then data are written or read using method `write` or `read`. The internal implementation of how to write or read data is different among the devices, but data writing and reading are not different among interfaces, so it is desirable to use abstracted `Device` rather than specialized `Sensor`, `Effector`, `Command` and `Event`.

```
Sample code
Device leftWing = roboid.findDevice("LeftWing");
leftwing.write(100);
```

`SensoryDevice` has an interface as in Figure 14.1.

```
public interface SensoryDevice extends Device
{
        List<? extends MotoringDevice> getReceptors();
}
```

Figure 14.1 Interface `SensoryDevice`

```
List<? extends MotoringDevice> getReceptors();
```

If a sensory device is not a proxy, it can be connected to monitoring devices. This method returns a list of `MotoringDevice` instances connected in this way.

Return	a list of connected `MotoringDevice` instances

`MotoringDevice` has an interface as in Figure 14.2.

```
public interface MotoringDevice extends Device
{
}
```

Figure 14.2 Interface `MotoringDevice`

`Sensor` has an interface as in Figure 14.3.

```java
public interface Sensor extends SensoryDevice
{
	int getThrottle();
	void setThrottle(int value);
	List<Effector> getReceptors();
	Sensor getProxyFor();
	void setProxyFor(Sensor value);
	void addReceptor(Effector receptor);
	void removeReceptor(Effector receptor);
}
```

Figure 14.3 Interface `Sensor`

`int getThrottle();`

Return the throttle value of this sensor device. Sensor data are sent continuously when data are received through communication. In order to reduce communication, it is possible to send data once at every certain number of times. At that time, the number is the throttle value.

Return	a throttle value

`void setThrottle(int value);`

Set the throttle value of this sensor device.

Parameters	**`value`**	a throttle value

`List<Effector> getReceptors();`

When this device is not a proxy, it can be connected to effector devices. This method returns a list of `Effector` instances connected in this way.

Return	a list of connected `Effector` instances

`Sensor getProxyFor();`

Return the connected `Sensor` instance if this device is a proxy.

Return	the connected `Sensor` instance

`void setProxyFor(Sensor value);`

Set the sensor device to be connected if this device is a proxy.

Parameters	**`value`**	the `Sensor` instance to be connected

`void addReceptor(Effector receptor);`

If this device is not a proxy, connect it to the given `Effector` device.

Parameters	**`receptor`**	the `Effector` instance to be connected

```
void removeReceptor(Effector receptor);
```

If this device is not a proxy, release it from connection to the given `Effector` device.

Parameters	**`receptor`**	an `Effector` instance, connection to which is to be disconnected.

`Event` has an interface as in Figure 14.4.

```
public interface Event extends SensoryDevice
{
	int getId();
	List<Command> getReceptors();
	Event getProxyFor();
	void setProxyFor(Event value);
	void addReceptor(Command receptor);
	void removeReceptor(Command receptor);
}
```

Figure 14.4 Interface `Event`

```
int getId();
```

Return an event ID

Return	an event ID

```
List<Command> getReceptors();
```

When this device is not a proxy, it can be connected to command devices. This method returns a list of `Command` instances connected in this way.

Return	a list of connected `Command` instances

```
Event getProxyFor();
```

Return the connected `Event` instance if this device is a proxy.

Return	the connected `Event` instance

```
void setProxyFor(Event value);
```

Set the event device to be connected if this device is a proxy.

Parameters	**`value`**	the `Event` instance to be connected

```
void addReceptor(Command receptor);
```

If this device is not a proxy, connect it to the given command device.

Parameters	**`receptor`**	the `Command` instance to be connected

```
void removeReceptor(Command receptor);
```

If this device is not a proxy, release it from connection to the given command device.

Parameters	**`receptor`**	a `Command` instance, connection to which is to be disconnected

`Effector` has an interface as in Figure 14.5.

```
public interface Effector extends MotoringDevice
{
        int getSustain();
        void setSustain(int value);
        int getThrottle();
        void setThrottle(int value);
        Effector getProxyFor();
        void setProxyFor(Effector value);
        boolean hasNext();
}
```

Figure 14.5 Interface `Effector`

```
int getSustain();
```

Return the sustain value of the effector device. An effector device can sustain previous data for a specific length of time even if there is no more data to be set. At that time, the length of time is the sustain value.

Return	A sustain value

```
void setSustain(int value);
```

Set the sustain value of the effector device.

Parameters	**`value`**	a sustain value

```
int getThrottle();
```

Return the throttle value of the effector device. Effector data are sent continuously when data are received through communication. In order to reduce communication, it is possible to send data once at every certain number of times. At that time, the number is the throttle value.

Return	a throttle value

```
void setThrottle(int value);
```

Set the throttle value of the effector device.

Parameters	**`value`**	a throttle value

```
Effector getProxyFor();
```

Return the connected `Effector` instance if this device is a proxy.

Return	the connected `Effector` instance

```
void setProxyFor(Effector value);
```

Set the effector device to be connected if this device is a proxy.

Parameters	**`value`**	the `Effector` instance to be connected

```
boolean hasNext();
```

Return whether there are next data in the buffer.

Return	`true` if there is the next data in the buffer, or `false`

`Command` has an interface as in Figure 14.6.

```
public interface Command extends MotoringDevice
{
	int getId();
	Command getProxyFor();
	void setProxyFor(Command value);
}
```

Figure 14.6 Interface `Command`

```
int getId();
```

Return a command ID

Return	a command ID

```
Command getProxyFor();
```

Return the connected `Command` instance if this device is a proxy.

Return	the connected `Command` instance

```
void setProxyFor(Command value);
```

Set the command device to be connected if this device is a proxy.

Parameters	**`value`**	the `Command` instance to be connected