Oracle® Java ME Embedded

Device Access API Guide

Release 3.4

E35134-03

September 2013

This document is a resource for software developers who want to program embedded peripherals using the Device Access APIs of the Oracle Java ME Embedded software.

Oracle Java ME Embedded Device Access API Guide, Release 3.4

E35134-03

Contents

Preface ix

Audience ix
Documentation Accessibility ix
Related Documentation ix
Conventions ix

1 Introduction

Device Access API Permissions 1-1
The Peripheral Interface 1-2
The PeripheralConfig Interface 1-3
The PeripheralEventListener Interface 1-4
The Transactional Interface 1-4
The PeripheralEvent Class 1-5
The PeripheralManager Class 1-6
Exceptions 1-7

2 Analog-to-Digital Converter

The AcquisitionListener Interface 2-3
The ADCChannel Interface 2-3
The MonitoringListener Interface 2-6
The AcquisitionEvent Class 2-6
The ADCChannelConfig Class 2-6
The MonitoringEvent Class 2-7
Exceptions 2-8

3 AT Commands

The ATDevice Interface 3-3
The ATModem Interface 3-6
The CommandResponseHandler Interface 3-6
The DataConnection Interface 3-6
The DataConnectionHandler Interface 3-7
The UnsolicitedResponseHandler Class 3-7

4 Pulse Counters

The PulseCounter Interface 4-2
The CountingEvent Class 4-4
The PulseCounterConfig Class 4-4

5 Digital-to-Analog Converter

The DACChannel Interface 5-2
The GenerationListener Interface 5-4
The DACChannelConfig Class 5-5
The GenerationEvent Class 5-5
Exceptions 5-6

6 Generic Input/Output Classes

The GenericBufferIODevice Interface 6-4
The GenericDevice Interface 6-4
The GenericEventListener Class 6-5
The GenericStreamIODevice Class 6-5
The GenericEvent Class 6-6

7 General Purpose Input/Output (GPIO)

The GPIOPin Interface 7-3
The GPIOPort Interface 7-4
The PinListener Interface 7-6
The PortListener Interface 7-6
The GPIOPinConfig Class 7-6
The GPIOPortConfig Class 7-8
The PinEvent Class 7-9
The PortEvent Class 7-9

8 Inter-Integrated Circuit Bus

The I2CDevice Interface 8-3
The I2CDeviceConfig Class 8-5

9 Memory-Mapped Input/Output

The MMIODevice Interface 9-3
The MMIOEventListener Interface 9-5
The RawMemory Interface 9-5
The RawBlock Interface 9-5
The RawByte Interface 9-6
The RawInt Interface 9-7
The RawShort Interface 9-7
The MMIOEvent Class 9-7
The MMIODeviceConfig Class 9-8
The MMIODeviceConfig.RawMemoryConfig Class 9-9
The MMIODeviceConfig.RawBlockConfig Class 9-9

The MMIODeviceConfig.RawRegisterConfig Class 9-9
The MMIOEvent Class 9-10
AccessOutOfBoundsException 9-10

10 Modem Control Signals

The ModemSignalListener Interface 10-1
The ModemSignalsControl Class 10-1
The ModemSignalEvent Class 10-2

11 Power Management

The PowerManaged Interface 11-2
The PowerSavingsHandler Class 11-4

12 Serial Peripheral Interface Bus

The SPIDevice Interface 12-2
The SPIDeviceConfig Class 12-4
InvalidWordLengthException 12-5

13 UART

The ModemUART Interface 13-3
The UART Interface 13-3
The UARTEventListener Interface 13-5
The UARTConfig Class 13-5
The UARTEvent Class 13-7

14 Watchdog Timers

The WatchdogTimer Interface 14-2
The WindowedWatchdogTimer Interface 14-3

A Migrating from Device Access Version 3.2

The PeripheralManager Class A-1

Glossary ....

Index

List of Examples

2–1 Using the ADC APIs 2-1
3–1 Using the AT Commands API 3-1
4–1 Using the Pulse Counter API 4-1
5–1 Using the DAC API 5-1
6–1 Creating an Alarm using the Generic APIs 6-2
6–2 An Audio Capture Using the Generic APIs 6-3
7–1 Using the GPIO APIs 7-1
8–1 Using the I2C APIs to Interact with LEDs 8-1
8–2 Writing and Reading Data Using the I2C APIs 8-2
11–1 Using the Power Management APIs 11-1
12–1 Using the SPI APIs to Communicate with SPI Slaves 12-1
13–1 Communicating using the UART API 13-1
13–2 Using a ModemUART to Communicate 13-2
14–1 Using the Watchdog API 14-1

List of Tables

1–1 Device Access API Permissions 1-2
1–2 Exceptions of the com.oracle.deviceaccess Package 1-7
2–1 Exceptions of the com.oracle.deviceaccess.adc Package 2-8
3–1 Permissions for Using the AT Command APIs 3-3
3–2 Properties of the ATDevice Interface 3-3
5–1 Exceptions of the com.oracle.deviceaccess.dac Package 5-6
7–1 GPIO API Permissions 7-3
7–2 GPIOPin Direction Constants 7-3
7–3 GPIOPort Direction Constants 7-5
7–4 Direction Constants in the GPIOPinConfig Class 7-7
7–5 Mode Constants in the GPIOPinConfig Class 7-7
7–6 Trigger Constants in the GPIOPinConfig Class 7-7
7–7 Direction Constants in the GPIOPortConfig Class 7-8
12–1 Clock Modes in the SPIDeviceConfig Class 12-4
13–1 UARTConfig Constants 13-5

Preface

This document describes the Device Access APIs of the Oracle Java ME Embedded product. The Device Access APIs contain classes and interfaces for communicating with peripherals that are connected to an embedded development board using one of several communication buses.

Audience

This document focuses on providing information and guidelines for ISV engineers who want to communicate with peripheral devices. Together with this document, ISV Java ME engineers should have access to the Oracle Java ME Embedded SDK, a compatible IDE, and an embedded development board or appropriate emulator.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documentation

For a complete list of documents included with the Oracle Java ME Embedded software, see the Release Notes.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.

Convention	Meaning
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introduction

The Device Access APIs provides interfaces and classes for communicating with and controlling peripheral devices attached to the embedded board. This chapter gives an introduction to the common interfaces and classes that are used throughout the Device Access API packages.

WARNING: Any IMlet that accesses the Device Access APIs *must* be digitally signed using a trusted certificate authority, or the appropriate permissions of the Device Access APIs must be added to the policy file for your platform. An IMlet that is not signed or is not granted the appropriate permissions will encounter an authentication error when attempting to access the Device Access APIs. See the Getting Started Guide for your target platform for more information..

Device Access API Permissions

In order to utilize the Device Access APIs, applications must present the proper permissions. These permission must be requested in the application's JAD file under either the `MIDlet-Permissions` or `MIDlet-Permissions-Opt` entry and the application must be digitally signed by a trusted authority, or the permission may be granted to all applications by adding it to the `untrusted` domain of the Java security policy file (`policy.txt`).

Peripheral devices are registered, un-registered, configured and listed by invoking one of the `open()`, `register()`, `unregister()` and `list()` methods of `PeripheralManager`. The four `PeripheralManager` permissions allow access to be granted for registration, configuration, un-registration and listing of peripheral devices. Note that these permissions must be granted in addition to the permissions granting access to a specific peripheral type. For example, opening a GPIO pin with an ad-hoc configuration requires both the `com.oracle.deviceaccess.gpio` and `com.oracle.deviceaccess.PeripheralManager.configure` to be granted. Similarly, registering a GPIO pin with an ad-hoc configuration requires both the `com.oracle.deviceaccess.gpio` and `com.oracle.deviceaccess.PeripheralManager.register` to be granted.

Table 1–1 shows the API Permissions that are required in an application's JAD file or security policy file in order to access the various Device Access APIs.

Table 1–1 Device Access API Permissions

Device Access API	Description
`com.oracle.deviceaccess.PeripheralManager.configure`	Ad-hoc configuration of peripheral devices
`com.oracle.deviceaccess.PeripheralManager.register`	Registration of peripheral devices
`com.oracle.deviceaccess.PeripheralManager.unregister`	Unregistration of peripheral devices
`com.oracle.deviceaccess.PeripheralManager.list`	Listing of registered peripheral devices
`com.oracle.deviceaccess.adc`	Analog-to-Digital Converter (ADC) APIs
`com.oracle.deviceaccess.atcmd`	Access to AT devices and modems as a whole
`com.oracle.deviceaccess.atcmd.ATDevice.openDataConnection`	Opening data connections with AT devices
`com.oracle.deviceaccess.counter`	Pulse counter APIs
`com.oracle.deviceaccess.dac`	Digital-to-Analog Converter (DAC) APIs
`com.oracle.deviceaccess.generic`	Generic APIs
`com.oracle.deviceaccess.gpio`	Access to GPIO pins and ports as a whole
`com.oracle.deviceaccess.gpio.GPIOPin.setDirection`	Changing the direction of a GPIO pin
`com.oracle.deviceaccess.gpio.GPIOPort.setDirection`	Changing the direction of a GPIO port
`com.oracle.deviceaccess.i2c`	I^2C slave devices as a whole
`com.oracle.deviceaccess.mmio`	Memory-mapped IO devices
`com.oracle.deviceaccess.power`	Power management functionality
`com.oracle.deviceaccess.spi`	Serial Peripheral Interface bus devices
`com.oracle.deviceaccess.uart`	UART devices
`com.oracle.deviceaccess.watchdog`	Watchdog timers

For individual peripheral mappings, see the appropriate appendix in the Getting Started Guide for your hardware development platform.

The Peripheral Interface

The `Peripheral` interface provides generic methods for handling any peripheral. All peripherals *must* implement this interface. There are four constants in the interface.

- `static final int LITTLE_ENDIAN`

  This constant represents little-endian byte or bit ordering.

- `static final int BIG_ENDIAN`

  This constant represents big-endian byte or bit ordering.

- `static final int MIXED_ENDIAN`

  This constant represents mixed-endian (non-standard) byte ordering.
- `static final int UNIDENTIFIED_ID`

  This constant is used to signify an unidentified peripheral numerical ID.

The interface also has five methods.

- `void close() throws java.io.IOException`

  This method releases the underlying peripheral device, making it available to other applications. Once released, subsequent operations on the same `Peripheral` instance will throw a `PeripheralNotAvailableException`. This method has no effects if the peripheral device has been released.
- `int getID()`

  This method returns the numerical ID of the underlying peripheral device.
- `java.lang.String getName()`

  This method returns the given name of the underlying peripheral device. A name is a descriptive version of the peripheral ID, such as "`LED1`" or "`BUTTON2`".
- `java.lang.String[] getProperties()`

  This method returns the properties of the peripheral device.
- `boolean isOpen()`

  This method indicates whether this peripheral is open or available to the calling application.

The PeripheralConfig Interface

The `PeripheralConfig` interface is a tagging interface for all peripheral configuration classes. It contains the following elements.

- *Hardware Addressing Information*

  Information required to address the peripheral device. Examples are an I2C bus number and slave device address, or a GPIO controller number and pin index.
- *Static Configuration Parameters*

  Static configuration parameters are configuration parameters that must be set before the peripheral device is opened and may not be changed afterwards. Examples of static configuration parameters are an SPI slave device clock mode or word length.
- *Dynamic Configuration Parameters*

  Dynamic configuration parameters are configuration parameters for which a default value may be set before the peripheral device is opened, and which may still be changed while the peripheral is open. Dynamic configuration parameters can be changed after the peripheral device is open through methods of the `Peripheral` sub-interfaces. Examples of dynamic configuration parameters are a UART baud rate or the current direction of a bidirectional GPIO pin.

`PeripheralConfig` instances should be immutable. A compliant implementation of this specification *must* ensure that information encapsulated in a `PeripheralConfig` instance cannot be altered while it is being accessed. It should either create its own private copy of the instance, or a copy of the information it contains.

Some hardware addressing parameters, as well as static and dynamic configuration parameters, may be set to `PeripheralConfig.DEFAULT`. Whether default settings are supported are both platform-dependent as well as peripheral driver-dependent.

An instance of `PeripheralConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` or `PeripheralManager.open(Class, PeripheralConfig)` method to open the designated peripheral device with the specified configuration. A `PeripheralConfigInvalidException` is thrown if the user attempts to open a peripheral device with an invalid or unsupported configuration.

The `PeripheralConfig` interface contains one constant.

- `static final int DEFAULT`

  This constant indicates that the default value of a configuration parameter should be used.

The PeripheralEventListener Interface

The `PeripheralEventListener` interface is a tagging interface that all event listeners must implement. It contains no constants or methods. Event listeners provide methods for notifying applications of the occurrence of events, such as hardware interrupts or software signals, on peripheral devices.

The Transactional Interface

The `Transactional` interface contains methods for providing a communication transaction. If a `Peripheral` instance implements this interface, then a transaction will be demarcated by a call to `begin()` and a call to `end()`. The read and write operations between these two calls will be part of a single transaction. A peripheral device driver may then use this transaction demarcation to qualify the sequence of read and write operations accordingly. For example, an I2C driver will treat the sequence of read and write operations to the same I2C slave device as a combined message. An SPI driver will treat the sequence of read and write operations to the same SPI slave device as a single transaction and will assert the Slave Select line during the entire transaction.

Note that in order to ensure that the `end()` method is always invoked, these methods should be used within a `try` ... `finally` block:

```
try {
     peripheral.begin();
     // read and write operations
} finally {
     peripheral.end();
}
```

There are two methods in the `Transactional` interface:

- `void begin() throws java.io.IOException, PeripheralNotAvailableException`

  Demarcates the beginning of a transaction. This method may throw an `InvalidStateException` if a transaction is already in progress.

- `void end() throws java.io.IOException, PeripheralNotAvailableException`

  Demarcates the end of a transaction. This method may throw an `InvalidStateException` if a transaction is not already in progress.

The PeripheralEvent Class

The `PeripheralEvent` class encapsulates events fired by or on behalf of a peripheral device. Such an event may correspond to a hardware interrupt or a software signal.

An *event burst* occurs when more events are fired than can be handled. When this happens, events may be coalesced. Coalescing of events is platform and peripheral-dependent.

The `PeripheralEvent` class consists of the following fields:

- `protected long timeStamp`

  The time (in milliseconds) when this event occurred. If events were coalesced then the time is that of the first event.

- `protected long lastTimeStamp`

  The time (in milliseconds) when the last coalesced event occurred. If events were not coalesced then the time is the same as that of the first event.

- `protected int timeStampMicros`

  The additional microseconds to the timestamp for when this event first occurred. If events were coalesced then this is that of the first event. The actual timestamp in microseconds is equal to: `(timeStamp * 1000) + timeStampMicros`.

- `protected int lastTimeStampMicros`

  The additional microseconds to the timestamp for when the last coalesced event occurred. If events were not coalesced then this is the same as that of the first event. The actual last timestamp in microseconds is equal to: `(lastTimeStamp * 1000) + lastTimeStampMicros`

- `protected int count`

  The number of underlying coalesced hardware interrupts or software signals this event may represent.

- `protected Peripheral peripheral`

  The `Peripheral` instance that fired this event or for which this event was fired.

There are several methods in this class, which acts as accessors for the protected fields.

- `public final long getTimeStamp()`

  Returns the time (in milliseconds) when this event first occurred. If events were coalesced then the time is that of the first event.

- `public final int getTimeStampMicros()`

  Returns the additional microseconds to the timestamp for when this event first occurred. If events were coalesced then the time is that of the first event.

- `public final long getLastTimeStamp()`

  Returns the time (in milliseconds) when the last coalesced event occurred. If events were not coalesced then the time is the same as that of the first event.

- `public final int getLastTimeStampMicros()`

  Returns the additional microseconds to the timestamp for when the last coalesced event occurred. If events were not coalesced then this is the same as that of the first event.

- `public final int getCount()`

Returns the number of underlying coalesced hardware interrupts or software signals this event may represent.

- `public final Peripheral getPeripheral()`

  Returns the `Peripheral` instance that fired this event or for which this event was fired.

The PeripheralManager Class

The `PeripheralManager` class provides static methods for opening and registering peripheral devices. These devices can then be handled as `Peripheral` instances. A `Peripheral` instance of a particular type can be opened using its platform-specific numerical ID or name as well as its properties.

`Peripheral` instances are uniquely identified by a numerical ID. This ID is unrelated to the hardware number (hardware addressing information) that may be used to identify a device such as a GPIO pin number or an I2C slave device address. The numerical ID of a peripheral device must be an integer greater than or equal to 0 and must be unique. Yet the same peripheral device may be directly and indirectly mapped through several IDs; each ID may correspond to a different configuration, representation or abstraction for the same underlying peripheral device hardware resource.

The `PeripheralManager` class consists of the following methods:

- `public static int register(int id, java.lang.Class intf, PeripheralConfig config, java.lang.String name, java.lang.String[] properties) throws java.io.IOException, PeripheralConfigInvalidException, PeripheralTypeNotSupportedException, PeripheralNotFoundException, PeripheralNotAvailableException, PeripheralExistsException`

  This method registers a new peripheral device under the specified ID (and optionally name and properties) supporting the provided configuration. The designated peripheral may be probed to check if the provided configuration is valid.

- `public static void unregister(int id)`

  This method unregisters the peripheral device associated with the specified ID. Some peripheral devices are registered by the underlying platform and cannot be unregistered.

- `public static Peripheral open(int id) throws java.io.IOException, PeripheralNotFoundException, PeripheralNotAvailableException`

  This method looks up and opens a `Peripheral` instance for the provided numerical ID. A new `Peripheral` instance is returned upon each call.

- `public static Peripheral open(int id, java.lang.Class intf) throws java.io.IOException, PeripheralTypeNotSupportedException, PeripheralNotFoundException, PeripheralNotAvailableException`

  This method looks up and opens a `Peripheral` instance for the provided numerical ID and type. A new `Peripheral` instance is returned upon each call.

- `public static Peripheral open(java.lang.String name, java.lang.Class intf, java.lang.String[] properties) throws java.io.IOException, PeripheralTypeNotSupportedException, PeripheralNotFoundException, PeripheralNotAvailableException`

This method looks up and opens a `Peripheral` instance for the specified name, type and/or properties. A new `Peripheral` instance is returned upon each call. Property-based lookup only uses exact (case-insensitive) matching and does not perform any semantic interpretation.

- `public static Peripheral open(PeripheralConfig config) throws java.io.IOException, PeripheralConfigInvalidException, PeripheralTypeNotSupportedException, PeripheralNotFoundException, PeripheralNotAvailableException, PeripheralExistsException`

  This method opens a `Peripheral` instance with the specified hardware addressing information and configuration. Note that the returned `Peripheral` instance's ID and name are undefined. A new instance is returned upon each call.

- `public static Peripheral open(java.lang.Class intf, PeripheralConfig config) throws java.io.IOException, PeripheralConfigInvalidException, PeripheralTypeNotSupportedException, PeripheralNotFoundException, PeripheralNotAvailableException, PeripheralExistsException`

  This method opens a `Peripheral` instance of the specified type with the specified hardware addressing information and configuration. Note that the returned `Peripheral` instance's ID and name are undefined. A new instance is returned upon each call.

- `public static Peripheral[] list()`

  This method lists all platform- and user-registered peripheral devices. The `Peripheral` instances returned are in a closed state and a call to `Peripheral.isOpen()` will return `false`.

- `public static Peripheral[] list(java.lang.Class inf)` throws PeripheralTypeNotSupportedException

  This method lists all platform- and user-registered peripheral devices of the designated type. The `Peripheral` instances returned are in a closed state; a call to `Peripheral.isOpen()` will return `false`.

Exceptions

The `com.oracle.deviceaccess` package consists of eight exceptions, which are shown in Table 1–2:

Table 1–2 Exceptions of the com.oracle.deviceaccess Package

Suite Type	Description
`InvalidOperationException`	Thrown by an instance of `Peripheral` to indicate that an attempted operation is not allowed for the peripheral.
`InvalidStateException`	Thrown by an instance of `Peripheral` to indicate that an operation as been attempted at an inappropriate time. In other words, the peripheral device is not in an appropriate state for the requested operation.
`PeripheralConfigInvalidException`	Thrown to indicate that the provided peripheral configuration is invalid/is not supported.
`PeripheralException`	Thrown to indicate that a general exception occurred on a peripheral operation.

Table 1–2 (Cont.) Exceptions of the com.oracle.deviceaccess Package

Suite Type	Description
`PeripheralExistsException`	Thrown by the `PeripheralManager` to indicate that a peripheral device is already registered for the specified ID.
`PeripheralNotAvailableException`	Thrown by an instance of `Peripheral` to indicate that an operation is attempted on a peripheral which is not yet available.
`PeripheralNotFoundException`	Thrown to indicate that there is no peripheral matching the provided peripheral numerical ID or name.
`PeripheralTypeNotSupportedException`	Thrown to indicate permanent unavailability of the looked up peripheral.

2 Analog-to-Digital Converter

The `com.oracle.deviceaccess.adc` package contains interfaces and classes for reading analog inputs using an Analog-to-Digital Converter (ADC). An ADC converts a continuous physical input quantity, such as a voltage, to a digital number. The conversion involves quantization of the input, so it typically introduces a small amount of error.

One ADC converter can have several channels. Each channel can sample a continuous input voltage and convert it to a numerical value. In order to access and control a specific ADC channel, an application should first open and obtain an `ADCChannel` instance for the ADC channel using its numerical ID, name, type (interface) and/or properties.

ADC channels are opened by invoking one of the `PeripheralManager.open()` methods. This is an example of using its ID.

```
ADCChannel channel = (ADCChannel) PeripheralManager.open(8);
```

This is an example of using its name and interface.

```
ADCChannel channel = (ADCChannel) PeripheralManager.open("TEMPERATURE",
    ADCChannel.class, null);
```

Once the peripheral is opened, the application can read or monitor sampled input values using methods of the `ADCChannel` interface, such as the `getValue()` method:

```
int temp = channel.getValue();
```

When done, the application should call the `ADCChannel.close()` method to release ADC channel.

```
channel.close();
```

Example 2–1 demonstrates two ways of using the ADC APIs.

Example 2–1 Using the ADC APIs

```
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.adc.ADCChannel;
import com.oracle.deviceaccess.adc.AcquisitionEvent;
import com.oracle.deviceaccess.adc.AcquisitionListener;
import java.io.IOException;

class ADCAcquisition implements AcquisitionListener {
     private ADCChannel channel = null;

     public void start(int channelID) throws IOException,
         PeripheralNotAvailableException, PeripheralNotFoundException
     {
         channel = (ADCChannel) PeripheralManager.open(channelID);
         channel.setSamplingInterval(100); // every 100 milliseconds
         int[] values = new int[10];
         channel.startAcquisition(values, 0, values.length, false, this);
     }

     public void inputAcquired(AcquisitionEvent event) {
         for (int i = 0; i < event.getCount(); i++) {
             int value = event.getValues()[event.getOffset() + i];
             // Handle value...
         }
     }

     public void stop() throws IOException, PeripheralNotAvailableException {
         if (channel != null) {
             channel.stopAcquisition();
             channel.close();
         }
     }
 }


class ADCThreshold implements MonitoringListener {

     private ADCChannel channel = null;

     public void start(int channelID, int low, int high)
         throws IOException, PeripheralNotAvailableException,
             PeripheralNotFoundException {
         channel = (ADCChannel) PeripheralManager.open(channelID);
         channel.setSamplingInterval(100); // every 100 milliseconds
         channel.startMonitoring(low, high, this);
     }

     public void thresholdReached(MonitoringEvent event) {
         if (event.getType() == MonitoringEvent.OUT_OF_RANGE) {
             int value = event.getValue();
             // Handle condition...
         }
     }

     public void stop() throws IOException, PeripheralNotAvailableException {
         if (channel != null) {
             channel.stopMonitoring();
             channel.close();
         }
     }
 }
```

Note that the `com.oracle.deviceaccess.adc` permission allows access to be granted to ADC channels as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the ADC APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

Because of performance issue, any procedures that handle analog inputs, and especially event listeners, should be optimized to be as fast as possible.

The AcquisitionListener Interface

The `AcquisitionListener` interface contains methods for being notified of the availability of sampled values. An `AcquisitionListener` can be registered using the `ADCChannel.startAcquisition(int[], int, int, boolean, com.oracle.deviceaccess.adc.AcquisitionListener)` method.

The `AcquisitionListener` interface contains only one method.

- `void inputAcquired(AcquisitionEvent event)`

  This method is invoked when a buffer of ADC sampled values has been filled and is available for processing.

The ADCChannel Interface

The `ADCChannel` interface provides methods for controlling an ADC (Analog to Digital Converter) channel.

One ADC device can have several channels. Analog input are sampled and converted according to the ADC device resolution to raw digital values between `getMinValue()` and `getMaxValue()`. Actual input voltage values can be calculated from raw digital values and the *reference voltage* value as returned by `getVRefValue()`. Each ADC channel is identified by a numerical ID and by a name.

An `ADCChannel` instance can be opened by a call to one of the `PeripheralManager.open()` methods. Once opened, an application can read the current sampled input value of an ADC channel by calling the `getValue()` method or can acquire the input values sampled over a period of time by calling the `getValues(int[], int, int)` method.

An application can also asynchronously acquire the input values sampled over a period of time by calling the `startAcquisition()` methods with an `AcquisitionListener` instance which will get cyclicly and asynchronously notified when the desired number of samples have been acquired. The input acquisition can be stopped by calling the `stopAcquisition()` method.

An application can monitor the input value by calling the `startMonitoring()` method with a low and a high threshold value and `MonitoringListener` instance which will get asynchronously notified when the input value gets out of or back in the defined range. The monitoring can be stopped by calling the `stopMonitoring()` method.

Only one acquisition (synchronous or asynchronous) and one monitoring can occur at any time. One acquisition and one monitoring can be performed concurrently at the same sampling rate (see `getSamplingInterval()`). They therefore respectively acquire and monitor the same sampled input values.

When an application is no longer using an ADC channel, it should call the `ADCChannel.close()` method to release the ADC channel. Any further attempt to set or get the value of a ADC channel which has been closed will throw a `PeripheralNotAvailableException`.

Note that asynchronous notification of range conditions or input acquisition is only loosely tied to hardware-level interrupt requests. The platform does not guarantee notification in a deterministic or timely manner.

The `ADCChannel` interface consists of several methods:

- `int getMinValue()` throws java.io.IOException, PeripheralNotAvailableException

  This method returns the minimum raw value this channel can sample. If the ADC device resolution is `n` then the minimum value returned by `getMinValue()` and the maximum value returned by `getMaxValue()` are such that `(max - min) == (2^n - 1)`.

- `int getMaxValue() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the maximum raw value this channel can sample. If the ADC device resolution is `n` then the minimum value returned by `getMinValue()` and the maximum value returned by `getMaxValue()` are such that `(max - min) == (2^n - 1)`.

- `double getVRefValue()` throws java.io.IOException, PeripheralNotAvailableException

  This method returns the reference voltage value of this ADC channel. If the reference voltage is `vRef` and the ADC device resolution is `n` then the actual input voltage value corresponding to a raw sampled value value read on this channel can be calculated as follows: `vInput = (value * vRef) / (2^n)`

- `int getValue() throws java.io.IOException, PeripheralNotAvailableException`

  This method reads the current raw sampled input value of this channel. This method may be invoked at any time. If another thread has already initiated an I/O operation upon this channel, however, then an invocation of this method will block until the first operation is complete. Only one acquisition (synchronous or asynchronous) can be going on at any time.

- `void getValues(int[] buffer, int offset, int count) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads `count` raw sampled input values from this channel and copies them into the designated array. The input will be sampled according to the current sampling interval as returned by `getSamplingInterval()`. This method may be invoked at any time. If another thread has already initiated an I/O operation upon this channel, however, then an invocation of this method will block until the first operation is complete. Only one acquisition (synchronous or asynchronous) can be going on at any time.

- `void startAcquisition(int[] buffer, int offset, int count, boolean doubleBuffering, AcquisitionListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method starts sampling this channel input and asynchronously notifies the provided `AcquisitionListener` instance when count raw sampled input values have been read from this channel. The read values are copied into the designated section of the provided buffer. Once count raw sampled input values have been read, reading will be suspended and in the event of continuous sampling, subsequent sampled input values may be lost. Reading into the buffer and notification will only resume once the event has been handled. Reading and notification will immediately start and will repeat until it is stopped by a call to `stopAcquisition()`.

If double buffering is enabled, notification will happen when `(count / 2)` raw sampled input values have been read and reading will proceed with the other half of the designated section of the provided buffer. Reading will only be suspended if the previous event has not yet been handled (this may result in the case of continuous sampling in subsequent sampled input values to be lost). If `count` is not even then one part of the designated buffer section may be longer (by 1) than the other one.

The input will be sampled according to the current sampling interval as returned by `getSamplingInterval()`. Note that only one acquisition (synchronous or asynchronous) can be going on at any time.

- `void stopAcquisition() throws java.io.IOException, PeripheralNotAvailableException`

  This method stops the asynchronous sampling of this channel input as started by a call to one of the `startAcquisition()` methods.

- `void startMonitoring(int low, int high, MonitoringListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method starts monitoring this channel input and asynchronously notifies the provided `MonitoringListener` instance when this channel's raw sampled input value gets out of or back in the specified range (as defined by a low and a high threshold value). Monitoring and notification will immediately start and will repeat until it is stopped by a call to `stopMonitoring()`. Range notification operates in toggle mode: once notified of an out-of-range condition the application will next only get notified of a back-in-range condition and so on.

  The sampled input value will be monitored according to the current sampling interval as returned by `getSamplingInterval()`. To only be notified when the input value gets over some threshold, call this method with the low parameter set to the value of `getMinValue()`. Conversely, to only be notified when the input value gets under some threshold, call this method with the high parameter set to the value of `getMaxValue()`. If `low` is lower than the minimum value returned by `getMinValue()`, then the minimum value is assumed. If `high` is higher the maximum value returned by `getMaxValue()`, then the maximum value is assumed. Only one monitoring can occur at any time.

- `void stopMonitoring()`

  This method stops the range monitoring of this channel input as started by a call to the `startMonitoring()` method.

- `void setSamplingInterval(int interval) throws java.io.IOException, PeripheralNotAvailableException`

  This method sets the sampling interval, in microseconds.

- `int getSamplingInterval() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the sampling interval, in microseconds. If the sampling interval was not set previously using `setSamplingInterval(int)`, the peripheral configuration-specific default value is returned.

- `int getMinSamplingInterval()` throws java.io.IOException, PeripheralNotAvailableException

  This method returns the minimum sampling interval, in microseconds, that can be set using `setSamplingInterval(int)`.

The MonitoringListener Interface

The `MonitoringListener` interface defines a method for being notified of ADC channel under- and over-threshold input value conditions. A `MonitoringListener` can be registered using the `ADCChannel.startMonitoring(int, int, MonitoringListener)` method.

The `MonitoringListener` interface consists of only one method:

- `void thresholdReached(MonitoringEvent event)`

  This method is invoked when the input value has reached the low or high threshold.

The AcquisitionEvent Class

The `AcquisitionEvent` class encapsulates ADC channel input acquisition conditions. Note that this kind of event is never coalesced.

There are two constructors in the `AcquisitionEvent` class:

- `public AcquisitionEvent(ADCChannel channel, int[] values, int offset, int number)`

  This method creates a new `AcquisitionEvent` with the specified raw sampled values and time-stamped with the current time.

- `public AcquisitionEvent(ADCChannel channel, int[] values, int offset, int number, long timeStamp, int timeStampMicros)`

  This method creates a new `AcquisitionEvent` with the specified raw sampled values and timestamp.

There are three methods in this class as well.

- `public int[] getValues()`

  This method returns the buffer containing the sampled values.

- `public int getOffset()`

  This method returns the offset in the values buffer where the sampled values start.

- `public int getNumber()`

  This method returns the number of sampled values.

The ADCChannelConfig Class

The `ADCChannelConfig` class encapsulates the hardware addressing information, and static and dynamic configuration parameters, of an ADC channel. Some hardware addressing parameter, and static and dynamic configuration parameters, may be set to `PeripheralConfig.DEFAULT`. Whether such default settings are supported is both platform-dependent and peripheral driver-dependent.

An instance of `ADCChannelConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` or `PeripheralManager.open(Class, PeripheralConfig)` method to open the designated ADC channel with the specified configuration. A `PeripheralConfigInvalidException` is thrown when attempting to open a peripheral device with an invalid or unsupported configuration.

The `ADCChannelConfig` class consists of one constructor and several methods.

- `public ADCChannelConfig(int converterNumber, int channelNumber, int resolution, int samplingInterval, int samplingTime)`

  This constructor creates a new `ADCChannelConfig` with the specified hardware addressing information and configuration parameters.

- `public int getChannelNumber()`

  This method returns the configured channel number.

- `public int getResolution()`

  This method returns the configured resolution.

- `public int getConverterNumber()`

  This method returns the configured converter number.

- `public int getSamplingInterval()`

  This method returns the configured default/initial sampling interval (the amount of time between two samples) in microseconds.

- `public int getSamplingTime()`

  This method returns the configured sampling time (the amount of time to take a sample) in microseconds.

The MonitoringEvent Class

The `MonitoringEvent` class encapsulates ADC channel under- and over-threshold value conditions. If range events for the same ADC channel are coalesced, the value and the type (either in or out of range) are that of the last occurrence.

The `PeripheralManager` class contains two constants:

- `public static final int OUT_OF_RANGE`

  Indicates that the ADC channel value exceeded the defined range.

- `public static final int BACK_TO_RANGE`

  Indicates that the ADC channel value returned to the defined range.

The `PeripheralManager` class consists of the following constructors and methods:

- `public MonitoringEvent(ADCChannel channel, int type, int value)`

  This constructor creates a new `MonitoringEvent` with the specified raw sampled value. It will automatically be time-stamped with the current time.

- `public MonitoringEvent(ADCChannel channel, int type, int value, long timeStamp, int timeStampMicros)`

  This constructor creates a new `MonitoringEvent` with the specified raw sampled value and the specified timestamp.

- `public int getType()`

  This method returns the type of range condition being notified.

- `public int getValue()`

  This method returns the new ADC channel's value.

Exceptions

The `com.oracle.deviceaccess.adc` package consists of one exception, which is shown in Table 2–1:

Table 2–1 Exceptions of the com.oracle.deviceaccess.adc Package

Suite Type	Description
`InvalidSamplingRateException`	Thrown by an instance of `ADCChannel` when the requested sampling rate is higher than the maximum sampling rate the ADC device can support.

3 AT Commands

The `com.oracle.deviceaccess.atcmd` package contains interfaces and classes for controlling data communication equipment such as a modem or a cellular module using *AT commands*. AT commands for GSM phone or modem are standardized through ETSI GSM 07.07 and ETSI GSM 07.05 specifications. A typical modem or an cellular module supports most of its features through AT commands and many manufactures provide additional features by adding proprietary extensions to the AT commands set.

In this specification, a device that can be controlled using AT commands is generically referred to as an *AT device*. To control a specific AT device, an application should first open and obtain an `ATDevice` or `ATModem` instance for the device using its numerical ID, name, type (interface) and properties:

This example obtains an `ATDevice` using its ID:

```
ATDevice device = (ATDevice) PeripheralManager.open(15);
```

This is an example of using the name and interface, returns as either an `ATDevice` or an `ATModem` object.

```
ATDevice device = (ATDevice) PeripheralManager.open("MODEM", ATDevice.class,
    new String[] { "javax.deviceaccess.atcmd.psd=true",
   "javax.deviceaccess.atcmd.sms=true" });

ATModem device = (ATModem) PeripheralManager.open("MODEM", ATModem.class,
    new String[] { "javax.deviceaccess.atcmd.psd=true",
   "javax.deviceaccess.atcmd.sms=true" });
```

Once the peripheral opened, the application can issue AT commands to the peripheral using methods of the `ATDevice` interface such as the `sendCommand()` methods.

```
device.sendCommand("AT\n");
```

When done, the application should call the `Peripheral.close()` method to release AT device.

```
device.close();
```

Example 3–1 shows how to use the AT Commands API to send an SMS message:

Example 3–1 Using the AT Commands API

```
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.PeripheralTypeNotSupportedException;
import com.oracle.deviceaccess.atcmd.ATDevice;
import com.oracle.deviceaccess.atcmd.CommandResponseHandler;
import java.io.IOException;

public class SMSExample {

 public static final int SUBMITTED = 1;
 public static final int SENT = 2;
 public static final int ERROR = 3;
 private ATDevice modem = null;
 private int status = 0;

 private class SMSHandler implements CommandResponseHandler {

     String text;

     public SMSHandler(String text) {
         this.text = text;
     }

     public String processResponse(ATDevice modem, String response) {
         // Assume that command echo has been previously disabled
         // (such as with an ATE0 command).

         if (response.equals("> \n")) { // Prompt for text
             return text;
         } else if (response.equals("OK\n")) {
             status = SENT;  // Sent succesfully
         } else if (response.indexOf("ERROR") >= 0) {
             status = ERROR; // Failed to send
         }
         return null;
     }
 }

 public boolean sendSMS(final String number, final String text) {
     // Acquire a modem with "sms" properties
     try {
         if (modem == null) {
             modem = (ATDevice) PeripheralManager.open(null, ATDevice.class,
                     new String[] { "javax.deviceaccess.atcmd.sms=true" });
         }
         // Send SMS command
         SMSHandler sh = new SMSHandler(text);
         modem.sendCommand("AT+CMGS=\"" + number + "\"\n", sh);
         status = SUBMITTED;
         return true; // Submitted succesfully
     } catch (IOException ex) {
     } catch (PeripheralNotFoundException ex) {
     } catch (PeripheralTypeNotSupportedException ex) {
     } catch (PeripheralNotAvailableException ex) {
     }
     return false;
 }

 public int getStatus() {
     return status;
 }

 public void close() {
     if (modem != null) {
         try {
             modem.close();
         } catch (IOException ex) {
             // Ignored
         }
     }
 }
}
```

AT devices are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The permissions shown in Table 3–1 allow access to be granted to AT devices as a whole or to only some of their protected functions. These permissions must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permissions may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

Table 3–1 Permissions for Using the AT Command APIs

Permissions	Description
`com.oracle.deviceaccess.atcmd`	Access to AT devices and modems (as a whole)
`com.oracle.deviceaccess.atcmd.ATDevice.openDataConnection`	Opening data connections

The ATDevice Interface

The `ATDevice` interface provides methods for controlling data communication equipment. Each AT device is identified by a numerical ID and optionally by a name and by a set of capabilities (properties), as shown in Table 3–2.

Table 3–2 Properties of the ATDevice Interface

Permissions	Description
`javax.deviceaccess.atcmd.config`	Supports access to configuration, control, and identification commands.
`javax.deviceaccess.atcmd.csd`	Supports access to circuit switched data (CSD) related AT commands.
`javax.deviceaccess.atcmd.psd`	Supports access to packet switched data, such as GPRS or EDGE, related AT commands.
`javax.deviceaccess.atcmd.voice`	Supports access to voice call related AT commands.
`javax.deviceaccess.atcmd.sms`	Supports access to SMS related AT commands.
`javax.deviceaccess.atcmd.sim`	Supports access to SIM related AT commands.
`javax.deviceaccess.atcmd.phonebook`	Supports access to phonebook related AT commands.

This list may be extended to designate other, possibly proprietary, capabilities (properties). As per convention, when one of this capabilities is supported by an AT device it must be assigned as a positively-asserted boolean capability: `<keyword>=true`. For example:

```
javax.deviceaccess.atcmd.phonebook=true
```

When a capability is not supported by an AT device, negatively asserting the boolean capability is optional.

An `ATDevice` instance can be opened by a call to one of the `PeripheralManager.open()` methods. Commands can be issued to an ATdevice either synchronously or asynchronously. When submitted synchronously using the `sendCommand(String)`, the response string will be available as the return value to the call. When submitted asynchronously using the `sendCommand(String, CommandResponseHandler)` a `CommandResponseHandler` instance must be provided to handle the response when it becomes available.

Note that the command strings passed as parameter to the `sendCommand()` methods are the complete AT command lines, including the AT prefix and a termination character.

An `ATDevice` can only handle one command at a time. Commands cannot be sent or queued while a command is already being handled. Nevertheless, if supported by the underlying AT device, several commands may be concatenated in a single command line.

An `ATDevice` may report responses that are either information text or result codes. A result code can be one of three types: *final*, *intermediate*, and *unsolicited*. A final result code, such as `OK` or `ERROR`, indicates the completion of command and the readiness for accepting new commands. An intermediate result code (such as `CONNECT`) is a report of the progress of a command. An unsolicited result code (such as `RING`) indicates the occurrence of an event not directly associated with the issuance of a command.

Information text, final result code and intermediate result code responses are reported as return values of calls to the `sendCommand(String)` method or as the parameter to the `processResponse()` method of a `CommandResponseHandler` instance provided as parameter to a call to `sendCommand(String, CommandResponseHandler)`.

Note that such response strings may include command echoes, unless command echo has been disabled, such as with an `ATE0` command.

Unsolicited result code responses are reported and passed as parameter to the `processResponse()` method of a `UnsolicitedResponseHandler` instance.

A data connection can be established by calling the `openDataConnection(java.lang.String, com.oracle.deviceaccess.atcmd.CommandResponseHandler, com.oracle.deviceaccess.atcmd.DataConnectionHandler)` with a dedicated AT command such as ATD. The state of the connection can be monitored by additionally providing an `DataConnectionHandler` instance..

When done, an application should call the `ATDevice.close()` method to release the AT device. Any further attempt to use an `ATDevice` instance which has been closed will result in a `PeripheralNotAvailableException` been thrown.

Note that the `sendCommand()` methods of `ATDevice` do not parse the provided AT commands. The AT command line should include the AT prefix and the proper termination character when it is needed.

The `ATDevice` interface contains several command-related methods.

- `void sendCommand(java.lang.String cmd, CommandResponseHandler handler) throws java.io.IOException, PeripheralNotAvailableException`

  This method sends an AT command and handle the response asynchronously. The call will return immediately and the provided `CommandResponseHandler` instance will be invoked to handle the response when available. The command line may or may not include payload text (such as SMS body text), in which case the the provided `CommandResponseHandler` instance will be invoked to provide the additional input text (text prompt mode). If the command line includes payload text, it must be properly terminated.

- `java.lang.String sendCommand(java.lang.String cmd) throws java.io.IOException,PeripheralNotAvailableException`

  This method sends an AT command and waits for the response. If the command line includes payload text, it must be properly terminated, otherwise the operation may block. A blocked call may be canceled by a call to `abortCommand(java.lang.String)`. Note that the return response string may include the command echo unless command echo has been disabled, such as with an `ATE0` command.

- `void abortCommand(java.lang.String abortSequence) throws java.io.IOException,PeripheralNotAvailableException`

  This method aborts the currently executing command by sending the provided `abortSequence` string. Abortion depends on the command's definition, or more accurately, if it supports cancellation. Note that calling this method does *not* guarantee abortion of the currently executing command. It only aborts if the command supports cancellation and it is currently in a proper state for it.

- `void escapeToCommandMode() throws java.io.IOException, PeripheralNotAvailableException`

  When in data mode, calling this method will try to switch to command mode such as sending "+++" escape sequence.

- `boolean isInCommandMode() throws java.io.IOException, PeripheralNotAvailableException`

  This method queries if this AT device is in command mode. When the device is in command mode, a new command can be sent, provided no command is currently being processed.

- `boolean isConnected() throws java.io.IOException, PeripheralNotAvailableException`

  This method queries if this AT device has an opened data connection.

- `void setUnsolicitedResponseHandler(UnsolicitedResponseHandler handler) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a handler for handling unsolicited result code responses. If handler is `null`, then the previously registered handler will be removed. Only one handler can be registered at a particular time.

- `DataConnection openDataConnection(java.lang.String cmd, CommandResponseHandler crHandler, DataConnectionHandler dcHandler) throws java.io.IOException, PeripheralNotAvailableException`

  This method opens a data connection by issuing the specified AT command, and optionally handles the response and the opened connection asynchronously. The call will return immediately and the provided `CommandResponseHandler` and `DataConnectionHandler` instances will be invoked to handle the error or intermediate and final result response, respectively, when available. Finally, the connection will be subsequently closed.

- `int getMaxCommandLength() throws java.io.IOException, PeripheralNotAvailableException`

This method returns the maximum length of the command string that can be processed by the underlying AT parser. Command string exceeding this value may be cut off without warning as this is a default behavior of modems.

- `void close()`

  This method closes and releases the underlying peripheral device, making it available to other applications. Once released, subsequent operations on the same `Peripheral` instance will throw a `PeripheralNotAvailableException`.This method has no effects if the peripheral device has already been closed. Note that closing an `ATDevice` will also close the device's `DataConnection` as well as its `InputStream` and `OutputStream`.

The ATModem Interface

The `ATModem` provides methods for controlling data communication equipment such as a modem or a cellular module using AT commands and modem control signals. It extends the `ATDevice` and `ModemSignalsControl` interfaces, but otherwise does not define any methods of its own.

The CommandResponseHandler Interface

The `CommandResponseHandler` interface defines methods for handling responses to AT commands. When commands are submitted asynchronously using the `sendCommand(String, CommandResponseHandler)` method, a `CommandResponseHandler` instance must be provided to handle the response when it becomes available.

Only information text, final result code, and intermediate result code responses can be handled by a `CommandResponseHandler` instance. Unsolicited result code responses can be handled by a `UnsolicitedResponseHandler` instance.

The `CommandResponseHandler` interface consists of only one method:

- `java.lang.String processResponse(ATDevice atDevice, java.lang.String response)`

  This method is invoked to process an information text, final result code or intermediate result code response.

The DataConnection Interface

The `DataConnection` interface provides methods for retrieving the underlying input and output streams of a data connection opened by issuing an AT command (such as ATD).

There are three methods in this interface.

- `java.io.InputStream getInputStream() throws java.io.IOException`

  This method returns this data connection's input stream. The same `InputStream` instance is returned upon subsequent calls. Note that if this data connection's input stream has been previously closed, the method returns the same closed input stream without attempting to re-open it.

- `java.io.OutputStream getOutputStream() throws java.io.IOException`

  This method returns this data connection's output stream. The same `OutputStream` instance is returned upon subsequent calls. Note that if this data connection's output stream has been previously closed, the method returns the same closed output stream without attempting to re-open it.

- `void close() throws java.io.IOException`

  This method closes this data connection. When a connection has been closed, accessing any of its methods that involve an I/O operation will throw an `IOException`. Closing an already closed connection has no effect. Closing a connection will also close the connection's `InputStream` and `OutputStream`.

The DataConnectionHandler Interface

The `DataConnectionHandler` interface defines methods for handling connection state changes.

There are two methods in this interface.

- `void handleOpenedDataConnection(ATDevice atDevice, DataConnection connection)`

  This method is invoked to handle a data connection when first opened.

- `void handleClosedDataConnection(ATDevice atDevice, DataConnection connection)`

  This method is invoked to handle a data connection when it has been closed.

The UnsolicitedResponseHandler Class

The `UnsolicitedResponseHandler` interface defines methods for handling unsolicited result code responses from an AT device. Unsolicited result codes (such as `RING`) indicate the occurrence of an event not directly associated with the issuance of an AT command. To receive unsolicited result codes an `UnsolicitedResponseHandler` instance must be registered with the AT device using the `ATDevice.setUnsolicitedResponseHandler(UnsolicitedResponseHandler)` method.

The `UnsolicitedResponseHandler` class consists of one method.

- `void processResponse(ATDevice atDevice,java.lang.String code)`

  This method is invoked to process an unsolicited result code response.

4 Pulse Counters

The `com.oracle.deviceaccess.counter` package contains interfaces and classes for counting pulses on a digital input line. In order to access and control a specific pulse counter, an application should first obtain an `PulseCounter` instance for the pulse counter the application wants to access and control, using its numerical ID, name, type (interface) and properties.

The following code is an example of using its ID.

```
PulseCounter counter = (PulseCounter) PeripheralManager.open(8);
```

This is an example of using its name and interface, returned as a `PulseCounter` object:

```
PulseCounter counter = (PulseCounter) PeripheralManager.open("ENCODER",
    PulseCounter.class, null);
```

Once opened, an application can start a pulse counting session by one of two ways. First, an application can use the `startCounting()` method and retrieve the current pulse count on-the-fly by calling the `PulseCounter.getCount()` method. Alternatively, the application can start a pulse counting session with a terminal count value and a counting time interval using the `PulseCounter.startCounting(int, long, com.oracle.deviceaccess.counter.CountingListener)`. The application can then be asynchronously notified once the terminal count value has been reached or the counting time interval has expired. In both cases, the application can retrieve the current pulse count value at any time by calling the `PulseCounter.getCount()`, as shown in the following example:

```
counter.startCounting(); // Start counting pulses
  // Perform some task...
int count = counter.getCount();
  // Retrieve the number of pulses that occurred while performing the task
counter.stopCounting(); // Stop counting pulses
```

When done, the application should call the `PulseCounter.close()` method to release `PulseCounter`.

```
counter.close();
```

Example 4–1 shows how to use the pulse counter API.

Example 4–1 Using the Pulse Counter API

```
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.counter.CountingEvent;
import com.oracle.deviceaccess.counter.CountingListener;
import com.oracle.deviceaccess.counter.PulseCounter;
import java.io.IOException;

 class PulseCounting implements CountingListener {

     private PulseCounter counter = null;

     public void start(int counterID) throws IOException,
         PeripheralNotAvailableException, PeripheralNotFoundException
 {
         counter = (PulseCounter) PeripheralManager.open(counterID);
         counter.startCounting(-1, 1000, this);
           // Count events occuring during 1 second (without terminal count value)
     }

     public void countValueAvailable(CountingEvent event) {
         int count = event.getValue();
         // Handle pulse count...
     }

     public void stop() throws IOException, PeripheralNotAvailableException {
         if (counter != null) {
             counter.stopCounting();
             counter.close();
         }
     }
 }
```

Because of performance issue, procedures handling pulse counting events should be optimized to be as fast as possible.

Pulse counters are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.counte`r permission allows access to be granted to pulse counter devices as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The PulseCounter Interface

The `PulseCounter` interface provides methods for controlling a pulse counter. A pulse counter can count pulses on a digital input line, possibly a GPIO pin.

A `PulseCounter` instance can be opened by a call to one of the `PeripheralManager.open()` methods. Once opened, an application can either start a pulse counting session using the `startCounting()` method and retrieve the current pulse count on-the-fly by calling the `getCount()` method; alternatively, it can start a pulse counting session with a terminal count value and a counting time interval using the `startCounting(int, long, com.oracle.deviceaccess.counter.CountingListener)` and get asynchronously notified once the terminal count value has been reached or the counting time interval has expired. In both cases, the application can retrieve the current pulse count at any time (on-the-fly) by calling the `getCount()` method.

The pulse counting session can be suspended by calling `suspendCounting()` and later on resumed from its previous count value by calling `resumeCounting()`. Suspending the pulse counting also suspends the session counting time interval timer if active. The pulse count value can be reset at anytime during counting by calling `resetCounting()`. This also resets the session counting time interval timer if active. Finally, the pulse counting can be stopped by calling `stopCounting()`.

When an application is no longer using a pulse counter it should call the `PulseCounter.close()` method to release the pulse counter. Any further attempt to use a pulse counter which has been closed will result in a `PeripheralNotAvailableException` been thrown. Note that asynchronous notification of pulse counting conditions is only loosely tied to hardware-level interrupt requests. The platform does not guarantee notification in a deterministic or timely manner.

The `PulseCounter` interface contains seven methods.

- `int getCount() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the pulse count measured so far during the current or previous counting session.

- `void startCounting() throws java.io.IOException, PeripheralNotAvailableException`

  This method starts a pulse counting session. The pulse count value is reset to zero (0).

- `void startCounting(int limit, long interval, CountingListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method starts an asynchronous pulse counting session. The provided `CountingListener` instance will be asynchronously invoked when the pulse count reaches the provided terminal count value or the specified counting time interval expires, whichever happens first. The pulse count value is first reset to zero (0), and will be reset every time the terminal count value is reached or the counting time interval expires.

  If `limit` is equal or less than 0 then the counting time interval will end only after the time specified by interval has passed. If `interval` is equal to or less than 0, then the counting time interval will end only after the pulse count has reached the terminal count value specified by `limit`.

  Pulse counting and notification will immediately start and will repeat until it is stopped by a call to `stopCounting()`. Only one pulse counting session can be going on at any time.

- `void resetCounting() throws java.io.IOException, PeripheralNotAvailableException`

  This method resets the current count value.

- `void stopCounting() throws java.io.IOException, PeripheralNotAvailableException`

  This method stops the pulse counting and freezes the current count value. The count value will be reset upon the next start.

- `void suspendCounting() throws java.io.IOException, PeripheralNotAvailableException`

  This method suspends the pulse counting and freezes the current count value.

- `void resumeCounting() throws java.io.IOException, PeripheralNotAvailableException`

  This method resumes the counting starting from the frozen count value.

The CountingEvent Class

The `CountingEvent` class encapsulates pulse counting conditions such as counter terminal value reached or counting session time interval expired. If counting events for the same pulse counter are coalesced the count value and the type (either the terminal value is reached or the time interval has expired) retained are that of the last occurrence.

The `CountingEvent` class consists of two constants, which describes the type:

- `public static final int TERMINAL_VALUE_REACHED`

  This constant indicates that the pulse count value has reached the defined terminal value.

- `public static final int INTERVAL_EXPIRED`

  This constant indicates that the pulse counting time interval has expired.

The `CountingEvent` class also consists of two constructors and three methods:

- `public CountingEvent(PulseCounter counter, int type, int value, long interval)`

  This constructor creates a new `CountingEvent` with the specified type, pulse count value and actual counting time interval. The event is then time-stamped with the current time.

- `public CountingEvent(PulseCounter counter, int type, int value, long interval, long timeStamp, int timeStampMicros)`

  This constructor creates a new `CountingEvent` with the specified type, pulse count value, actual counting time interval and timestamp.

- `public int getType()`

  This method returns the type of counting condition being notified.

- `public int getValue()`

  This method returns the pulse count value.

- `public long getInterval()`

  This method returns the actual counting time interval, in milliseconds. The actual counting time interval may be smaller than the defined counting time interval if the count terminal value has been reached.

The PulseCounterConfig Class

The `PulseCounterConfig` class encapsulates the hardware addressing information, as well as the static and dynamic configuration parameters of a pulse counter.

Some hardware addressing parameter, and static and dynamic configuration parameters may be set to `PeripheralConfig.DEFAULT`. Whether such default settings are supported is both platform-dependent and peripheral driver-dependent.

An instance of `PulseCounterConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` or `PeripheralManager.open(Class, PeripheralConfig)` method to open the designated counter with the specified configuration. A `PeripheralConfigInvalidException` is thrown when attempting to open a peripheral device with an invalid or unsupported configuration

There are four constants in this class.

- `public static final int TYPE_FALLING_EDGE_ONLY`

  This constants represents a falling pulse edge (counting only falling pulse edges).

- `public static final int TYPE_RISING_EDGE_ONLY`

  This constants represents a rising pulse edge (counting only rising pulse edges).

- `public static final int TYPE_POSITIVE_PULSE`

  This constants represents a positive edge pulse: measured from rising edge to falling edge (counting well-formed positive edge pulses).

- `public static final int TYPE_NEGATIVE_PULSE`

  This constants represents a negative edge pulse: measured from falling edge to rising edge (counting well-formed negative edge pulses).

There are two constructors and three methods in this class.

- `public PulseCounterConfig(int counterNumber, int type)`

  This constructor creates a new `PulseCounterConfig` with the specified hardware addressing information and type. The source of the pulse counter is implicit, such as a dedicated input pin.

- `public PulseCounterConfig(int counterNumber, int type, GPIOPinConfig source)`

  This constructor creates a new `PulseCounterConfig` with the specified hardware addressing information, type and GPIO pin source.

- `public int getCounterNumber()`

  This method returns the configured counter number.

- `public GPIOPinConfig getSource()`

  This method returns the configured input source on which the pulses are to be counted or measured.

- `public int getType()`

  This method returns the configured pulse or pulse edge type.

5 Digital-to-Analog Converter

The `com.oracle.deviceaccess.dac` package contains interfaces and classes for writing analog outputs using a Digital to Analog Converter (DAC). A DAC is a device that converts a digital, typically binary, code to an analog signal, such as a current, voltage, or electric charge.

One DAC converter can have several channels. Each channel can sample an analog output from numerical values that are converted to output voltages. In order to access and control a specific DAC channel, an application should first open and obtain an `DACChannel` instance for the DAC channel the application wants to access and control, using its numerical ID, name, type (interface) and properties.

This is an example of using its ID.

```
 DACChannel channel = (DACChannel) PeripheralManager.open(5);
```

This is an example of using its name and interface.

```
 DACChannel channel = (DACChannel) PeripheralManager.open("LED", DACChannel.class,
null);
```

Once the peripheral is opened, an application can write output values to a DAC channel using methods of the `DACChannel` interface, such as the `setValue()` method.

```
 channel.setValue(brightness);
```

When completed, the application should call the `DACChannel.close()` method to release the DAC channel.

```
channel.close();
```

Example 5–1 shows how to use the DAC API.

Example 5–1 Using the DAC API

```
import com.oracle.deviceaccess.InvalidStateException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.dac.DACChannel;
import com.oracle.deviceaccess.dac.GenerationEvent;
import com.oracle.deviceaccess.dac.GenerationListener;
import java.io.IOException;

class VaryingDimmer implements GenerationListener {

     private DACChannel channel = null;
```

```
    public void start(int channelID) throws IOException,
      PeripheralNotAvailableException, PeripheralNotFoundException
    {
        if (channel != null) {
            throw new InvalidStateException();
        }
        channel = (DACChannel) PeripheralManager.open(channelID);
        channel.setSamplingInterval(1000); // every 1000 milliseconds
        // Creates a series of samples varying from min value to max value
        int[] values = new int[10];
        int min = channel.getMinValue();
        int max = channel.getMaxValue();
        for (int i = 0; i < values.length; i++) {
            values[i] = min + (((max - min) / (values.length - 1)) * i);
        }
        channel.startGeneration(values, 0, values.length, false, this);
    }

    public void outputGenerated(GenerationEvent event) {
        event.setActualNumber(event.getNumber());
             // Replay the same sample series
    }

    public void stop() throws IOException, PeripheralNotAvailableException {
        if (channel != null) {
            channel.stopGeneration();
            channel.close();
        }
    }
}
```

Because of performance issue, procedures handling analog outputs should be optimized to be as fast as possible.

DAC channels are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. Note that the `com.oracle.deviceaccess.dac` permission allows access to be granted to DAC channels as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The DACChannel Interface

The `DACChannel` interface provides methods for controlling a DAC (Digital to Analog Converter) channel.

One DAC device can have several channels. Raw digital output values are converted to analog output values according to the DAC channel resolution. According to the DAC channel resolution, the raw digital output values may range from `getMinValue()` to `getMaxValue()`. Actual output voltage values can be calculated from raw digital values and the Reference Voltage value as returned by `getVRefValue()`.

Each DAC channel is identified by a numerical ID and by a name. A `DACChannel` instance can be opened by a call to one of the `PeripheralManager.open()` methods. Once opened, an application can write an output value to a DAC channel by calling the `setValue(int)` method or can write a series of output values to be sampled over a period of time by calling the `setValues(int[], int, int)` method. An application can also asynchronously write a series of output values to be sampled over a period of time by calling by calling the `startSampling()` methods with a `SamplingListener` instance which will get cyclicly and asynchronously notified when the requested number of samples have been written.

The output sampling can be stopped by calling the `stopSampling()` method. Only one output operation (synchronous or asynchronous) can occur at any time. When an application is no longer using an DAC channel, it should call the `DACChannel.close()` method to release the DAC channel. Any further attempt to set or get the value of a DAC channel which has been closed will result in a `PeripheralNotAvailableException` been thrown.

Note that asynchronous notification of output sampling completion is only loosely tied to hardware-level interrupt requests. The platform does not guarantee notification in a deterministic or timely manner.

The `DACChannel` interface contains ten methods.

- `int getMinValue() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the minimum raw value this channel can sample. If the DAC device resolution is `n` then the minimum value returned by `getMinValue()` and the maximum value returned by `getMaxValue()` are such that: `(max - min) == (2^n - 1)`.

- `int getMaxValue() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the maximum raw value this channel can sample. If the DAC device resolution is `n` then the minimum value returned by `getMinValue()` and the maximum value returned by `getMaxValue()` are such that: `(max - min) == (2^n - 1)`.

- `double getVRefValue() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the reference voltage value of this DAC channel. If the reference voltage is `vRef` and the DAC device resolution is `n`, then the actual output voltage value corresponding to a raw value value written to this channel can be calculated as follows: `vOutput = (value * vRef) / (2^n)`

- `void setValue(int value) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes the provided raw output value to this channel. The corresponding converted analog output value will be held until it is overwritten by another output operation. This method may be invoked at any time. If another thread has already initiated a synchronous output operation upon this channel then an invocation of this method will block until the first operation is complete. Only one conversion,. synchronous or asynchronous, can occur at any time.

- `void setValues(int[] buffer, int offset, int count) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes `count` raw output values from the designated array to this channel for sampling. The analog output will be sampled according to the current sampling interval as returned by `getSamplingInterval()`. This method may be invoked at any time. If another thread has already initiated an I/O operation upon this channel, however, then an invocation of this method will block until the first operation is complete. Only one conversion, synchronous or asynchronous, can occur at any time.

- `void startGeneration(int[] buffer, int offset, int count, boolean doubleBuffering, GenerationListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method starts asynchronous analog output generation on this channel from a series of raw output values (samples). More values to be converted are asynchronously fetched by notifying the provided `GenerationListener` instance once the initial count raw output values have been converted. The initial raw output values to be converted are read from the designated section of the provided buffer. Values subsequently fetched using the provided `GenerationListener` instance are read from the same buffer section. Analog output generation can be stopped by a call to `stopGeneration()`.

  If double buffering is enabled, notification will happen when `(count / 2)` raw output values have been written and writing will proceed with the other half of the designated section of the provided buffer. Writing will only be suspended if the previous event has not yet been handled. if count is not even then one part of the designated buffer section may be longer (by 1) than the other one. The analog output will be sampled according to the current sampling interval as returned by `getSamplingInterval()`. Only one conversion, synchronous or asynchronous, can occur at any time.

- `void stopGeneration() throws java.io.IOException, PeripheralNotAvailableException`

  This method stops the asynchronous sampling of this channel output as started by a call to the `startGeneration()` method.

- `void setSamplingInterval(int interval) throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the sampling interval, in microseconds. Whether changing the sampling interval has an immediate effect on an active (synchronous or asynchronous) generation is peripheral device-dependent as well as platform-dependent.

- `int getSamplingInterval() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the sampling interval, in microseconds. If the sampling interval was not set previously using `setSamplingInterval(int)`, the peripheral configuration-specific default value is returned.

- `int getMinSamplingInterval() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the minimum sampling interval, in microseconds, that can be set using `setSamplingInterval(int)`.

The GenerationListener Interface

The `GenerationListener` interface defines methods for being notified of the completion of the conversion of a set of raw output values and that more output values to be converted may be provided. A `GenerationListener` can be registered using the `DACChannel.startGeneration(int[], int, int, boolean, com.oracle.deviceaccess.dac.GenerationListener)` method.

The `GenerationListener` interface consists of only one method:

- `void outputGenerated(GenerationEvent event)`

This method is invoked when a buffer of DAC output values has been converted, and the buffer is available for copying more output values for convertion.

The DACChannelConfig Class

The `DACChannelConfig` class encapsulates the hardware addressing information, and static and dynamic configuration parameters of an DAC channel. Some hardware addressing parameters, as well as static and dynamic configuration parameters, may be set to `PeripheralConfig.DEFAULT`. Whether such default settings are supported is both platform-dependent and peripheral driver-dependent.

An instance of `DACChannelConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` or `PeripheralManager.open(Class, PeripheralConfig)` method to open the designated DAC channel with the specified configuration. A `PeripheralConfigInvalidException` is thrown when attempting to open a peripheral device with an invalid or unsupported configuration

The `DACChannelConfig` interface consists of one constructor and four methods:

- `public DACChannelConfig(int converterNumber, int channelNumber, int resolution, int samplingInterval)`

  This constructor creates a new `DACChannelConfig` with the specified hardware addressing information and configuration parameters.

- `public int getChannelNumber()`

  This method returns the configured channel number.

- `public int getResolution()`

  This method returns the configured resolution.

- `public int getConverterNumber()`

  This method returns the configured converter number.

- `public int getSamplingInterval()`

  This method returns the default/initial configured sampling interval, in microseconds.

The GenerationEvent Class

The `GenerationEvent` class encapsulates DAC channel output sampling completion conditions. A `GenerationEvent` may indicate that: either all the values to output have been written and the designated buffer section is available for more values to output, or, in case of double buffering, half of the values to output have been written and the designated buffer section is available for more values to output.

When handling a `GenerationEvent`, the application may copy more output values to be converted in the buffer section designated by the `getValues(),getOffset()` and `getNumber()` methods. The application must set the actual number of output values copied by calling `setActualNumber(int)`. If the actual number is set to a value smaller than the length of the designated buffer section, as given by the `getNumber()` method, the current asynchronous analog output generation will stop after the last provided output values have been converted, as if from a call to `DACChannel.stopGeneration()`.

Note that this kind of event is never coalesced.

The `GenerationEvent` interface consists of two constructors and five methods:

- `public GenerationEvent(DACChannel channel, int[] values, int offset, int number)`

  This constructor creates a new `GenerationEvent` with the specified raw output value buffer and time-stamped with the current time..

- `public GenerationEvent(DACChannel channel, int[] values, int offset, int number, long timeStamp, int timeStampMicros)`

  This constructor creates a new `GenerationEvent` with the specified raw output value buffer and timestamp.

- `public int[] getValues()`

  This method returns the buffer where the values to output must be copied. This buffer is the same buffer that was passed as parameter to `DACChannel.startGeneration(int[], int, int, boolean, com.oracle.deviceaccess.dac.GenerationListener)`.

- `public int getOffset()`

  This method returns the offset in the values buffer where to start copying the values to output. This offset is within the range defined by the parameters passed to `DACChannel.startGeneration(int[], int, int, boolean, com.oracle.deviceaccess.dac.GenerationListener)`.

- `public int getNumber()`

  This method returns the maximum number of values to output that can be copied to the `values` buffer.

- `public void setActualNumber(int actualNumber)`

  This method sets the actual number of values to output that were copied to the `values` buffer. If the provided value is smaller than the length of the designated buffer section, as given by the `getNumber()` method, then the current asynchronous analog output generation will be stopped, as if from a call to `DACChannel.stopGeneration()`.

- `public int getActualNumber()`

  This method returns the actual number of values to output that were copied to the `values` buffer.

Exceptions

The `com.oracle.deviceaccess.dac` package consists of one exception, which is shown in Table 5–1:

Table 5–1 Exceptions of the com.oracle.deviceaccess.dac Package

Suite Type	Description
`InvalidSamplingRateException`	Thrown by an instance of `DACChannel` in case the requested sampling rate is higher than the maximum sampling rate the DAC device can support.

6 Generic Input/Output Classes

The `com.oracle.deviceaccess.generic` package contains interfaces and classes for controlling devices using generic I/O operations.

The generic device API allows for accessing peripheral devices when there are no more specific standard Java APIs, such as `I2CDevice`, `SPIDevice`, `GPIOPin` or `GPIOPort`. This API offers three primary interfaces to encapsulate these devices:

- `GenericDevice`

  This interface encapsulates device control operations and event listener registration. A device may implement this sole interface if it does not support any read and write operations.

- `GenericBufferIODevice`

  This interface encapsulates device control operations and event listener registration as inherited from `GenericBufferIODevice` as well as byte buffer read and write operations.

- `GenericStreamIODevice`

  This interface encapsulates device control operations and event listener registration as inherited from `GenericBufferIODevice` as well as stream-based read and write operations.

In order to access a device using its generic interface, an application should first open and obtain a `GenericDevice` instance for the device using its numerical ID, name, type (interface) and properties.

This is an example of using its ID.

```
GenericDevice device = (GenericDevice) PeripheralManager.open(17);
```

This is an example of using its name and interface.

```
GenericStreamIODevice device = (GenericDevice) PeripheralManager.open("STORAGE",
GenericStreamIODevice.class, null);
```

Once the peripheral is opened, the application can set and get its controls, as well as read and write data using methods of the `GenericDevice`, `GenericBufferIODevice` or `GenericStreamIODevice` interfaces.

```
 device.read(buffer, 0, buffer.length);
```

When completed, the application should call the `GenericDevice.close()` method to release the device.

```
device.close();
```

Example 6–1 and Example 6–2 show how to use the generic API to communicate with Real Time Clock device and an audio capture microphone, which may be accessible over USB.

Example 6–1 Creating an Alarm using the Generic APIs

```
import com.oracle.deviceaccess.PeripheralException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.generic.GenericDevice;
import com.oracle.deviceaccess.generic.GenericEvent;
import com.oracle.deviceaccess.generic.GenericEventListener;
import java.io.IOException;

public class GenericAlarm {

    public static final int EVT_ALARM = 0;
    public static final int SECONDS = 0;
    public static final int SEC_ALARM = 1;
    public static final int MINUTES = 2;
    public static final int MIN_ALARM = 3;
    public static final int HR_ALARM = 4;
    public static final int HOURS = 5;
    public static final int ALARM_ENABLED = 6;
    private GenericDevice rtc = null;

    // Sets the daily alarm for after some delay
    public void setAlarm(byte delaySeconds, byte delayMinutes, byte delayHours)
        throws IOException, PeripheralException
   {
        rtc = (GenericDevice) PeripheralManager.open("RTC",
              GenericDevice.class, (String[]) null);
        byte currentSeconds = ((Byte) rtc.getControl(SECONDS)).byteValue();
        byte currentMinutes = ((Byte) rtc.getControl(MINUTES)).byteValue();
        byte currentHours = ((Byte) rtc.getControl(HOURS)).byteValue();
        byte i = (byte) ((currentSeconds + delaySeconds) % 60);
        byte j = (byte) ((currentSeconds + delaySeconds) / 60);
        rtc.setControl(SEC_ALARM, new Byte(i));
        i = (byte) ((currentMinutes + delayMinutes + j) % 60);
        j = (byte) ((currentMinutes + delayMinutes + j) / 60);
        rtc.setControl(MIN_ALARM, new Byte(i));
        i = (byte) ((currentHours + delayHours + j) % 24);
        rtc.setControl(HR_ALARM, new Byte(i));

        rtc.setEventListener(EVT_ALARM, new GenericEventListener() {
            public void eventDispatched(GenericEvent event) {
                GenericDevice rtc = (GenericDevice) event.getPeripheral();
                // Notify application of alarm
            }
        });
        // Enable alarm.
        rtc.setControl(ALARM_ENABLED, Boolean.TRUE);
    }

    public void close() {
        try {
            rtc.close();
        } catch (IOException ex) {
        }
    }
}
```

Example 6–2 An Audio Capture Using the Generic APIs

```
import com.oracle.deviceaccess.PeripheralException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.generic.GenericBufferIODevice;
import com.oracle.deviceaccess.generic.GenericDevice;
import com.oracle.deviceaccess.generic.GenericEvent;
import com.oracle.deviceaccess.generic.GenericEventListener;
import java.io.IOException;

public class GenericAudioCapture {

    public static final int EVT_VOLUME_CHANGED = 0;
    public static final int MIC_VOLUME = 0;
    public static final int MIC_SAMPLE_RATE = 1;
    public static final int MIC_AUTOMATIC_GAIN = 2;
    public static final int MIC_MUTE = 3;

    public void audioCapture(byte[] buffer, float sampleRate, boolean agc) throws
IOException, PeripheralException {
        GenericBufferIODevice mic = null;
        try {
            mic = (GenericBufferIODevice) PeripheralManager
                    .open("MICROPHONE", GenericBufferIODevice.class,
                        (String[]) null);
            mic.setControl(MIC_SAMPLE_RATE, new Float(sampleRate));
            mic.setControl(MIC_AUTOMATIC_GAIN, agc ?
                    Boolean.TRUE : Boolean.FALSE);
            mic.setControl(MIC_MUTE, Boolean.FALSE);

            mic.setEventListener(EVT_VOLUME_CHANGED, new GenericEventListener() {
                public void eventDispatched(GenericEvent event) {
                    GenericDevice mic = (GenericDevice) event.getPeripheral();
                    try {
                        float currentVolume = ((Float)
                             mic.getControl(MIC_VOLUME)).floatValue();
                        // ...
                    } catch (IOException ex) {
                        ex.printStackTrace();
                    } catch (PeripheralNotAvailableException ex) {
                        ex.printStackTrace();
                    }
                }
            });
            mic.read(buffer, 0, buffer.length);
        } finally {
            if (mic != null) {
                mic.close();
            }
        }
    }
}
```

Generic devices are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.generic` permission allows access to be granted to generic devices as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The GenericBufferIODevice Interface

The `GenericBufferIODevice` interface defines generic methods for accessing and controlling peripheral devices using read and write operations.

A platform implementer may allow access and control of peripheral devices for which there exist no other more specific APIs through this interface.

The `GenericBufferIODevice` interface contains three methods.

- `int read(byte[] rxBuf, int rxOff, int rxLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads up to `rxLen` bytes of data from this device into an array of bytes. Note that the availability of new input data may be notified through an `GenericEvent` with ID `GenericEvent.INPUT_DATA_AVAILABLE`.

- `int read(int skip, byte[] rxBuf, int rxOff, int rxLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads up to `rxLen` bytes of data from this device into an array of bytes skipping the first skip bytes read. Note that the availability of new input data may be notified through an `GenericEvent` with ID `GenericEvent.INPUT_DATA_AVAILABLE`.

- `void write(byte[] txBuf, int txOff, int txLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes to this device `txLen` bytes from buffer `txBuf`. Note that an empty output buffer condition may be notified through an `GenericEvent` with ID `GenericEvent.OUTPUT_BUFFER_EMPTY`.

The GenericDevice Interface

The `GenericDevice` interface defines methods for setting and getting peripheral device-specific configuration and access (I/O) controls as well as registering event listeners.

An application can use this interface to set and get configuration and access (I/O) controls. A control is identified by a numerical ID and can be set or gotten using the `setControl(int, java.lang.Object)` and `getControl(int)` methods. Controls can be used to configured a peripheral device a well as performing basic input/output operations. The list of controls supported by a peripheral device is peripheral-device-specific.

An application can also register an `GenericEventListener` instance to monitor native events of the designated type fired by the peripheral device. To register a `GenericEventListener` instance, the application must call the `setEventListener(int, GenericEventListener)` method. The registered listener can later on be removed by calling the same method with a null listener parameter. Asynchronous notification may not be supported by all devices. An attempt to set a listener on a device which does not supports it will result in an `InvalidOperationException` being thrown.

A platform implementer may allow through this interface access and control of peripheral devices which do not require byte stream or buffer I/O (read, write) and for which there exist no other more specific API.

The `GenericDevice` interface consists of three methods:

- `void setControl(int id, java.lang.Object value) throws java.io.IOException, PeripheralNotAvailableException`

  This method sets the value of the specified control.

- `java.lang.Object getControl(int id) throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the value of the specified control.

- `void setEventListener(int eventId, GenericEventListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `GenericEventListener` instance to monitor native events of the designated type fired by the peripheral device associated to this `GenericDevice` object. While the listener can be triggered by hardware interrupts, there are no real-time guarantees of when the listener will be called. A list of event type IDs is defined in `GenericEvent`. This list can be extended with peripheral-specific IDs. If listener is null then listener previously registered for the specified event type will be removed. Only one listener can be registered at a particular time for a particular event type.

The GenericEventListener Class

The `GenericEventListener` interface defines methods for being notified of events fired by peripheral devices that implement the `GenericDevice` interface. A `GenericEventListener` can be registered using the `GenericDevice.setEventListener(int, GenericEventListener)` method.

The `GenericEventListener` interface consists of one method:

- `void eventDispatched(GenericEvent event)`

  This method is invoked when an event is fired by peripheral device.

The GenericStreamIODevice Class

The `GenericStreamIODevice` interface defines generic methods for accessing and controlling peripheral devices capable of working with input and output streams. A platform implementer may allow access and control of peripheral devices for which there exist no other more specific API through this interface.

The `GenericStreamIODevice` interface consists of two methods:

- `java.io.InputStream getInputStream() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns an input stream to this device. The same `InputStream` instance is returned upon subsequent calls. Note that if this device's input stream has been previously closed, this method returns that same closed input stream without attempting to re-open it. The availability of new input data may be notified through an `GenericEvent` using the ID `GenericEvent.INPUT_DATA_AVAILABLE`.

- `java.io.OutputStream getOutputStream() throws java.io.IOException, PeripheralNotAvailableException`

This method returns an output stream to this device. The same `OutputStream` instance is returned upon subsequent calls. Note that if this device's output stream has been previously closed, this method returns that same closed output stream without attempting to re-open it. An empty output buffer condition may be notified through an `GenericEvent` with the ID `GenericEvent.OUTPUT_BUFFER_EMPTY`.

- `void close() throws java.io.IOException`

  This method closes and releases the underlying peripheral device, making it available to other applications. Once released, subsequent operations on that very same `Peripheral` instance will throw a `PeripheralNotAvailableException`. This method has no effects if the peripheral device has already been closed. Note that closing a `GenericStreamIODevice` will also close the device's `InputStream` and `OutputStream`.

The GenericEvent Class

The `GenericEvent` class encapsulates events fired by peripherals that implement the `GenericDevice` interface.

The `GenericEvent` interface consists of three constants:

- `public static final int INPUT_DATA_AVAILABLE`

  This constant is an event ID indicating that input data is available for reading.

- `public static final int INPUT_BUFFER_OVERRUN`

  This constant is an event ID indicating an input buffer overrun.

- `public static final int OUTPUT_BUFFER_EMPTY`

  This constant is an event ID indicating that the output buffer is empty and that additional data may be written.

The `GenericEvent` interface consists of two constructors and one method:

- `public GenericEvent(GenericDevice device, int id)`

  This constructor creates a new `GenericEvent` with the specified value. The event is then time-stamped with the current time.

- `public GenericEvent(GenericDevice device, int id, long timeStamp, int timeStampMicros)`

  This constructor creates a new `GenericEvent` with the specified value and timestamp.

- `public int getID()`

  This method returns this event ID.

7

General Purpose Input/Output (GPIO)

This chapter discusses the interfaces and classes for reading from and writing to the General Purpose Input/Output (GPIO) pins and ports of the embedded device board.

A GPIO pin is a generic pin whose value consists of one of two voltage settings (*high* or *low*) and whose behavior can be programmed through software. A GPIO port is a platform-defined grouping of GPIO pins (often 4 or more pins). However, GPIO pins that are part of a GPIO port cannot be retrieved or controlled individually as GPIO pins.

In order to use a specific pin or port, an application should first open and obtain a `GPIOPin` or `GPIOPort` instance for the pin or port it wants to use, using its numerical ID, name, type (interface), or properties.

Here is an example of obtaining a `GPIOPin` and a `GPIOPort` using its ID:

```
GPIOPin pin = (GPIOPin) PeripheralManager.open(1);
GPIOPort port = (GPIOPort) PeripheralManager.open(0);
```

Here is an example of using its name and interface:

```
GPIOPin pin = (GPIOPin) PeripheralManager.open("LED_PIN", GPIOPin.class, null);
GPIOPort port = (GPIOPort) PeripheralManager.open("LCD_DATA_PORT",
    GPIOPort.class, null);
```

Once a pin is opened, an application can obtain the current value of a GPIO pin by calling the `GPIOPin.getValue()` method and set its value by calling the `GPIOPin.setValue(boolean)` method. Likewise, once a port opened, an application can obtain the current value of a GPIO port by calling the `GPIOPort.getValue()` method and set its value by calling the `GPIOPort.setValue(int)` method.

```
pin.setValue(true);
port.setValue(0xFF);
```

When done, the application should call the `GPIOPin.close()` or `GPIOPort.close()` method to release the pin or port, respectively.

```
pin.close();
port.close();
```

Example 7–1 gives an demonstration of using the GPIO API. First, it registers a pin listener for the GPIO input pin that a switch button is attached to. When the button is pressed, the listener is notified. The listener then turns the LED on or off by setting the GPIO output pin that the LED is attached to accordingly.

Example 7–1 Using the GPIO APIs

```
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.gpio.GPIOPin;
import com.oracle.deviceaccess.gpio.PinEvent;
import com.oracle.deviceaccess.gpio.PinListener;
import java.io.IOException;

public class GPIODemo {

    GPIOPin switchPin = null;
    GPIOPin ledPin = null;

    public GPIODemo() {
        try {
            switchPin = (GPIOPin) PeripheralManager.open(1);
            ledPin = (GPIOPin) PeripheralManager.open(3);
            switchPin.setInputListener(new PinListener() {
                public void valueChanged(PinEvent event) {
                    try {
                        ((GPIOPin) event.getPeripheral()).
                            setValue(event.getValue()); // turn LED on or off
                    } catch (IOException ex) {
                        // Ignored
                    } catch (PeripheralNotAvailableException ex) {
                        // Ignored
                    }
                }
            });
        } catch (IOException ex) {
            // Handle exception
        } catch (PeripheralNotFoundException ex) {
            // Handle exception
        } catch (PeripheralNotAvailableException ex) {
            // Handle exception
        } finally {
            if (switchPin != null) {
                try {
                    switchPin.close();
                } catch (IOException ex) {
                }
            }
            if (ledPin != null) {
                try {
                    ledPin.close();
                } catch (IOException ex) {
                }
            }
        }
    }
}
```

Note that the underlying platform configuration may allow for some GPIO pins or ports to be set by an application for either output or input, while others may be used for input only or output only and their direction cannot be changed by an application. Note also that asynchronous notification of pin or port value changes is only loosely tied to hardware-level interrupt requests. The platform does not guarantee notification in a deterministic or timely manner.

Because of performance issue, procedures handling GPIO pins, and especially event listeners, should be implemented to be as fast as possible.

GPIO pins and ports are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The permissions in Table 7–1 allow access to be granted to GPIO pins and ports. as a whole as well as to some of their protected functions. These permissions must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permissions may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

Table 7–1 GPIO API Permissions

Permission	Description
`com.oracle.deviceaccess.gpio`	Access to GPIO pins and ports (as a whole)
`com.oracle.deviceaccess.gpio.GPIOPin.setDirection`	Changing the direction of a GPIO pin
`com.oracle.deviceaccess.gpio.GPIOPort.setDirection`	Changing the direction of a GPIO port

The GPIOPin Interface

The `GPIOPin` interface provides methods for controlling a GPIO pin. A GPIO pin can be configured for output or input. Output pins are both writable and readable while input pins are only readable. The interface contains two constants, as shown in Table 7–2:

Table 7–2 GPIOPin Direction Constants

Constant	Description
`GPIOPin.INPUT`	The GPIO pin is configured for input and is only readable.
`GPIOPin.OUTPUT`	The GPIO pin is configured for output and is both readable and writable.

Each GPIO pin is identified by a numerical ID and by a name. A `GPIOPin` instance can be opened by a call to one of the `PeripheralManager.open()` methods. Once opened, an application can obtain the current value of a GPIO pin by calling the `getValue()` method and set its value by calling the `setValue(boolean)` method.

An application can either monitor a GPIO pin value changes using polling or can register a `PinListener` instance, which will be asynchronously notified of any pin value changes. To register a `PinListener` instance, the application must call the `setInputListener(PinListener)` method. The registered listener can later on be removed by calling the same method with a null listener parameter. Asynchronous notification is only supported for GPIO pins configured for input. An attempt to set a listener on a GPIO pin configured for output will throw an `InvalidOperationException`.

When an application is no longer using a GPIO pin, it should call the `GPIOPin.close()` method to release the GPIO pin. Any further attempt to set or get the value of a GPIO pin which has been closed will throw a `PeripheralNotAvailableException`.

The initial direction of a GPIO pin which may be used for output or input. The initial value of a GPIO pin set for output is configuration-specific. An application should always initially set the GPIO pin's direction; or first query the GPIO pin's direction then set it if necessary.

Note that the configuration may allow for some GPIO pins to be set by the application for either output or input, while others may be used for input only or output only and their direction cannot be changed by the application. Note also that asynchronous notification of pin value changes is only loosely tied to hardware-level interrupt requests. The platform does not guarantee notification in a deterministic or timely manner.

The `GPIOPin` interface consists of the following methods:

- `int getDirection() throws IOException, PeripheralNotAvailableException`

  This method returns the current pin direction: `GPIOPin.OUTPUT` if this GPIO pin is currently set as output, or `GPIOPin.INPUT` if it is set as input.

- `boolean getValue() throws SecurityException, IOException, PeripheralNotAvailableException`

  This method returns the current value of the GPIO pin. This method can be called on both output and input pins. This method returns `true` if this pin is currently high, or `false` if it is low.

- `void setDirection(int direction) throws IOException, PeripheralNotAvailableException`

  This methods sets the GPIO pin direction, either for output or input. Any attempt to set a GPIO pin to a direction not supported by the platform configuration throws an `InvalidOperationException`.

- `void setValue(boolean value) throws IOException, PeripheralNotAvailableException`

  This method sets the value of this GPIO pin. The boolean parameter represents the new pin value: `true` for high, `false` for low. Any attempt to set the value on a GPIO pin currently not configured for output throws an `InvalidOperationException`.

- `void setInputListener(PinListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `PinListener` instance which will get asynchronously notified when this GPIO pin's value changes and according to the current trigger mode (see `GPIOPinConfig.getTrigger()`). Notification will automatically begin after registration completes. A listener can only be registered for a GPIO pin currently configured for input, and only one listener can be registered at a time. If the parameter passed in is `null`, the current listener is removed.

The GPIOPort Interface

The `GPIOPort` interface provides methods for controlling a GPIO port. A GPIO port is a platform-defined grouping of GPIO pins that can be configured for output or input. Like GPIO pins, each GPIO port is identified by a numerical ID and by a name. Output ports are both writable and readable while input ports are only readable. GPIO pins that are part of a GPIO port cannot be retrieved or controlled as individual `GPIOPin` instances.

The `GPIOPort` interface contains two constants, as shown in Table 7–3:

Table 7–3 GPIOPort Direction Constants

Constant	Description
`GPIOPort.INPUT`	The GPIO port is configured for input, and is only readable.
`GPIOPort.OUTPUT`	The GPIO port is configured for output, and is both readable and writable.

A `GPIOPort` instance can be opened by a call to one of the `PeripheralManager.open()` methods. Once opened, an application can obtain the current value of a GPIO port by calling the `getValue()` method and set its value by calling the `setValue(int)` method. A GPIO port has a minimum and maximum value range. The minimum value is zero. An application can check the maximum value by calling `getMaxValue()` method. An attempt to set a GPIO port with a value that exceeds its maximum range value will throw an `IllegalArgumentException`.

An application can either monitor a GPIO port value changes using polling or can register a `PortListener` instance, which will be asynchronously notified of any value changes. To register a `PortListener` instance, the application must call the `setInputListener(PortListener)` method. The registered listener can later on be removed by calling the same method with a null listener parameter. Asynchronous notification is only supported for GPIO port configured for input. An attempt to set a listener on a GPIO port configured for output will throw an `InvalidOperationException`.

When an application is no longer using a GPIO port, it should call the `GPIOPort.close()` method to release the GPIO port. Any further attempt to set or get the value of a GPIO port which has been closed will throw a `PeripheralNotAvailableException`.

The initial direction of a GPIO port which may be used for output or input. The initial value of a GPIO port set for output is configuration-specific. An application should always initially set the GPIO port's direction; or first query the GPIO port's direction then set it if necessary.

Note that the configuration may allow for some GPIO ports to be set by an application for either output or input, while others may be used for input only or output only and their direction cannot be changed by an application. Note also that asynchronous notification of port value changes is only loosely tied to hardware-level interrupt requests. The platform does not guarantee notification in a deterministic or timely manner.

The `GPIOPort` interface consists of the following methods:

- `int getDirection() throws IOException, PeripheralNotAvailableException`

  This method returns the current port direction: `GPIOPort.OUTPUT` if this GPIO port is currently set as output, or `GPIOPort.INPUT` if the port is set as input.

- `int getMaxValue() throws IOException, PeripheralNotAvailableException`

  This method returns the maximum value of this GPIO port. The value returned should be interpreted as an unsigned 32-bit integer.

- `int getValue() throws SecurityException, IOException, PeripheralNotAvailableException`

  This method returns the current value of this GPIO port. The value returned is interpreted as an unsigned 32-bit integer, and depending on the platform configuration, the value of each pin can be tested against a bit in the resulting value. This method can be called on both output and input pins.

- `void setDirection(int direction) throws IOException, PeripheralNotAvailableException`

  This method sets the GPIO port for output or input. Any attempt to set the direction of a GPIO port to a value that is not supported by the platform configuration throws an `InvalidOperationException`.

- `void setValue(int value) throws IOException, PeripheralNotAvailableException`

  This method sets the value of this GPIO port. Any attempt to set the value on a GPIO port currently not configured for output throws an `InvalidOperationException`. The value passed is interpreted as an unsigned 32-bit integer.

- `void setInputListener(PortListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `PortListener` instance that is asynchronously notified when this GPIO port's value changes. Notification automatically begins after registration completes. A listener can only be registered for a GPIO port currently configured for input, and only one listener can be registered at a time. If the parameter is `null`, the current listener is removed.

The PinListener Interface

The `PinListener` interface provides a means of notification if a GPIO pin value changes. A `PinListener` can be registered using the `GPIOPin.setInputListener(com.oracle.deviceaccess.gpio.PinListener)` method.

The interface consists of only one method, `void valueChanged(PinEvent event)`, which is invoked when a GPIO pin's value has changed.

The PortListener Interface

The `PortListener` interface provides a means of notification if a GPIO port value changes. A `PortListener` can be registered using the `GPIOPort.setInputListener(com.oracle.deviceaccess.gpio.PortListener)` method.

The interface consists of only one method, `void valueChanged(PortEvent event)`, which is invoked when a GPIO port's value has changed.

The GPIOPinConfig Class

The `GPIOPinConfig` class encapsulates the configuration parameters of a GPIO pin. An instance of `GPIOPinConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` method to open the designated GPIO pin with the specified configuration.

The `GPIOPinConfig` class consists of several constants. The first four represent possible directions for the GPIO pin, and are shown inTable 7–4.

Table 7–4 Direction Constants in the GPIOPinConfig Class

Constant	Description
`DIR_INPUT_ONLY`	Input direction
`DIR_OUTPUT_ONLY`	Output direction
`DIR_BOTH_INIT_INPUT`	Bidirectional with initial input direction.
`DIR_BOTH_INIT_OUTPUT`	Bidirectional with initial output direction.

The next four are possible values for the mode, and are shown in Table 7–5. Note that the mode can also be `PeripheralConfig.DEFAULT`.

Table 7–5 Mode Constants in the GPIOPinConfig Class

Constant	Description
`MODE_INPUT_PULL_UP`	Input pull-up drive mode.
`MODE_INPUT_PULL_DOWN`	Input pull-down drive mode.
`MODE_OUTPUT_PUSH_PULL`	Output push-pull drive mode.
`MODE_OUTPUT_OPEN_DRAIN`	Output open-drain drive mode.

Finally, the last seven are possible values for the trigger, and are shown in Table 7–6.

Table 7–6 Trigger Constants in the GPIOPinConfig Class

Constant	Description
`TRIGGER_NONE`	No interrupt trigger.
`TRIGGER_FALLING_EDGE`	Falling edge trigger.
`TRIGGER_RISING_EDGE`	Rising edge trigger.
`TRIGGER_BOTH_EDGES`	Both edges trigger.
`TRIGGER_HIGH_LEVEL`	High level trigger.
`TRIGGER_LOW_LEVEL`	Low level trigger.
`TRIGGER_BOTH_LEVELS`	Both levels trigger.

The `GPIOPinConfig` class consists of one constructor and six methods:

- `public GPIOPinConfig(int portNumber, int pinNumber, int direction, int mode, int trigger, boolean initValue)`

  This constructor creates a new `GPIOPinConfig` with the provided parameters. See the earlier discussion for possible constant values for `direction`, `mode`, and `trigger`.

- `public int getDirection()`

  This method returns the configured pin direction. The pin direction can be one of: `DIR_INPUT_ONLY`, `DIR_OUTPUT_ONLY`, `DIR_BOTH_INIT_INPUT`, or `DIR_BOTH_INIT_OUTPUT`.

- `public boolean getInitValue()`

  This method returns the configured initial boolean value of the pin, if configured for output.

- `public int getPortNumber()`

This method returns the configured port number for the pin.

- `public int getPinNumber()`

  This method returns the configured pin number.

- `public int getDriveMode()`

  This method returns the configured pin drive mode. The possible values can be: either `PeripheralConfig.DEFAULT` or a bitwise OR of at least one of : `MODE_INPUT_PULL_UP`, `MODE_INPUT_PULL_DOWN`, `MODE_OUTPUT_PUSH_PULL`, and `MODE_OUTPUT_OPEN_DRAIN`.

- `public int getTrigger()`

  This method returns the configured pin interrupt trigger. The pin interrupt trigger can be one of: `TRIGGER_NONE`, `TRIGGER_FALLING_EDGE`, `TRIGGER_RISING_EDGE`, `TRIGGER_BOTH_EDGES`, `TRIGGER_HIGH_LEVEL`, `TRIGGER_LOW_LEVEL`, `TRIGGER_BOTH_LEVELS`.

The GPIOPortConfig Class

The `GPIOPortConfig` class encapsulates the configuration parameters of a GPIO port. An instance of `GPIOPortConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` method to open the designated GPIO port with the specified configuration. Note that the interrupt trigger of a GPIO port is defined by the interrupt triggers configured for its pins. For more information, see `GPIOPinConfig.getTrigger()`.

The `GPIOPortConfig` class contains four constants, representative of the direction of the port. The constants are shown in Table 7–7.

Table 7–7 Direction Constants in the GPIOPortConfig Class

Constant	Description
`DIR_INPUT_ONLY`	Input port direction.
`DIR_OUTPUT_ONLY`	Output port direction.
`DIR_BOTH_INIT_INPUT`	Bidirectional port direction with initial input direction.
`DIR_BOTH_INIT_OUTPUT`	Bidirectional port direction with initial output direction.

The `GPIOPortConfig` class consists of one constructor and three methods:

- `GPIOPortConfig(int direction, int initValue, GPIOPinConfig[] pins)`

  This constructor creates a new `GPIOPortConfig` with the provided parameters. See the earlier discussion for possible constant values for `direction`.

- `public int getDirection()`

  This method returns the configured pin direction.

- `public boolean getInitValue()`

  This method returns the configured initial boolean value of the pin, if configured for output.

- `public GPIOPinConfig[] getPins()`

  This method returns the configured pins composing the port, in the exact same order that they compose the port.

The PinEvent Class

The `PinEvent` class encapsulates GPIO pin value changes. If value change events for the same GPIO pin are coalesced, the value returned is that of the last occurrence. The class consists of two constructors and one accessor.

- `PinEvent(GPIOPin pin, boolean value)`

  This constructor creates a new `PinEvent` with the specified value. The event is then time-stamped with the current time.

- `PinEvent(GPIOPin pin, boolean value, long timeStamp, int timeStampMicros)`

  This constructor creates a new `PinEvent` with the specified value and timestamp. Additional microseconds can also be added using the fourth parameter, if necessary.

- `boolean getValue()`

  This method returns a boolean indicating the GPIO pin's new value, `true` for high or `false` for low.

The PortEvent Class

The `PortEvent` class encapsulates GPIO port value changes. If value change events for the same GPIO port are coalesced, the value returned is that of the last occurrence.

- `PortEvent(GPIOPort port, boolean value)`

  This constructor creates a new `PortEvent` with the specified value. The event is then time-stamped with the current time.

- `PortEvent(GPIOPort port, boolean value, long timeStamp, int timeStampMicros)`

  This constructor creates a new `PortEvent` with the specified value and timestamp. Additional microseconds can also be added using the fourth parameter, if necessary.

- `int getValue()`

  This method returned is interpreted as an unsigned 32-bit integer representing the GPIO port's new value.

8 Inter-Integrated Circuit Bus

This chapter describes the interfaces and classes for Inter-Integrated Circuit Bus control. I²C (often pronounced "i-squared C") is a multi-master serial single-ended computer bus that is used to attach low-speed peripherals to an embedded system or other electronic device.

The functionalities supported by this API are those of an I²C master. In order to communicate with a specific slave device, an application should first open and obtain an `I2CDevice` instance for the I²C slave device the application wants to exchange data with, using its numerical ID, name, type (interface) or properties. This is an example of using its ID:

```
I2CDevice slave = (I2CDevice) PeripheralManager.open(3);
```

This is an example of using its name and interface:

```
I2CDevice slave = (I2CDevice) PeripheralManager.open("ADC1",
    I2CDevice.class, null);
```

Once the peripheral opened, the application can exchange data with the I²C slave device using methods of the `I2CDevice` interface such as the `write()` method.

```
slave.write(sndBuf, 0, 1);
```

When the data exchange is over, the application should call the `Peripheral.close()` method to release I²C slave device.

```
slave.close();
```

Example 8–1 and Example 8–2 demonstrate two ways of using the I²C API to communicate with an I²C slave device.

Example 8–1 Using the I2C APIs to Interact with LEDs

```
import com.oracle.deviceaccess.PeripheralException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.i2cbus.I2CDevice;
import java.io.IOException;

public class I2CExample1 {

    public static final String LED_SLAVE_NAME = "LED_CONTROLLER";
    public static final byte[] LED_STOP_COMMAND = null;
    public static final byte[] LED_OFF_COMMAND = null;
    public static final byte[] LED_ON_COMMAND = null;
    public static int LED_LOOP_COUNT = 10;
    public static long LED_BLINK_TIME = 1500;
    I2CDevice slave = null;

    public I2CExample1() {
        try {
            slave = (I2CDevice) PeripheralManager.open(
                LED_SLAVE_NAME, I2CDevice.class, (String[]) null);
            // Clear all status of the 'LED' slave device
            slave.write(LED_STOP_COMMAND, 0, LED_STOP_COMMAND.length);
            slave.write(LED_OFF_COMMAND, 0, LED_OFF_COMMAND.length);

            for (int i = 0; i < LED_LOOP_COUNT; i++) {
                // turning 'LED' on and keeping it on for 1500ms
                slave.write(LED_ON_COMMAND, 0, LED_ON_COMMAND.length);
                try {
                    Thread.sleep(LED_BLINK_TIME);
                } catch (InterruptedException ex) {
                }

                // turning 'LED' off keeping it off for 1500ms
                slave.write(LED_OFF_COMMAND, 0, LED_OFF_COMMAND.length);
                try {
                    Thread.sleep(LED_BLINK_TIME);
                } catch (InterruptedException ex) {
                }
            }
        } catch (IOException ex) {
            // Handle exception
        } catch (PeripheralException ex) {
            // Handle exception
        } finally {
            if (slave != null) {
                try {
                    slave.close();
                } catch (IOException ex) {
                }
            }
        }
    }
}
```

Example 8–2 Writing and Reading Data Using the I2C APIs

```
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.PeripheralTypeNotSupportedException;
import com.oracle.deviceaccess.i2cbus.I2CDevice;
import java.io.IOException;

public class I2CExample2 {

    I2CDevice slave = null;

    public I2CExample2() {
        try {
            slave = (I2CDevice) PeripheralManager.open("EEPROM",
                     I2CDevice.class, (String[]) null);

            byte[] addr = new byte[4];
```

```
                byte[] data = new byte[4];

                try {
                    slave.begin();
                    slave.write(addr, 0, 2); // Writes the address
                    int count = slave.read(data, 0, 1);
                    // Read the data at that EEPROM address
                } finally {
                    slave.end();
                }

            } catch (PeripheralNotAvailableException ex) {
                ex.printStackTrace();
            } catch (PeripheralNotFoundException ex) {
                ex.printStackTrace();
            } catch (PeripheralTypeNotSupportedException ex) {
                ex.printStackTrace();
            } catch (IOException ex) {
                ex.printStackTrace();
            } finally {
                if (slave != null) {
                    try {
                        slave.close();
                    } catch (IOException ex) {
                        ex.printStackTrace();
                    }
                }
            }
        }

}
```

More information about the I²C-bus specification can be found at
`http://www.nxp.com/documents/user_manual/UM10204.pdf`

I²C slave devices are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.i2c` permission allows access to be granted to I²C slave devices as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The I2CDevice Interface

The `I2CDevice` interface provides methods for sending and receiving data to and from an I²C slave device. Each I²C slave device is identified by both a numerical ID and a name. An `I2CDevice` instance can be acquired by a call to one of the `PeripheralManager.open()` methods.

On an I²C bus, data is transferred between the I²C master device and an I²C slave device through *single* or *combined* messages.

With single messages, the approach is simple: the I²C master can read data from an I²C slave using one of the `read()` methods and can write data to an I²C slave using one of the `write()` methods.

With combined messages, the I²C master issues at least two reads or writes to one or more slaves. Issuing multiple reads and writes to several slaves is not supported.

However, if the master is communicating with a single slave, it can explicitly start a combined message by calling the `begin()` method, issuing several read or write operations using the `read()` and `write()` methods, then end the combined message by calling the `end()` method. An application can also use the convenience methods `read(subaddress, subaddressSize,...)` and `write(subaddress, subaddressSize,...)`, which read and write from slave device subaddresses or register addresses.

The following example illustrates the use of `begin()` and `end()` to implement the `read(subaddress, subaddressSize,...)` method:

```
public int read(int subaddress, int subaddressSize, byte[] dstBuf, int dstOff,
     int dstLen) throws IOException, PeripheralNotAvailableException
 {
     if (subaddress < 0 || subaddressSize < 1 || subaddressSize > 4)
         throw IllegalArgumentException();

     byte[] subaddr = new byte[] { (byte) ((subaddress >> 24) & 0xFF),
                                   (byte) ((subaddress >> 16) & 0xFF),
                                   (byte) ((subaddress >> 8) & 0xFF),
                                   (byte) ((subaddress >> 0) & 0xFF)
     };

     try {
         begin();
         write(subaddr, subaddr.length - subaddressSize,
               subaddressSize); // Writes the subaddress
         return read(dstBuf, dstOff, dstLen); // Read the data at that subaddress
     } finally {
         end();
     }
}
```

When exchanging data, the most significant bytes of data are stored at the lower index (first) in the sending and receiving byte buffers.

When the data exchange is over, an application should call the `I2CDevice.close()` method to release the I^{2}C slave device. Any further attempt to write to or read from an I^{2}C slave device which has been closed will throw a `PeripheralNotAvailableException`. Note that the current API does not allow for reading and writing subsequently to and from different buffers without sending a repeated start between subsequent reads or writes.

The `I2CDevice` interface consists of the following methods:

- `void begin() throws java.io.IOException, PeripheralNotAvailableException`

  This method demarcates the beginning of an I^{2}C transaction. Subsequent read and write operations will be part of the same I^{2}C combined message.

- `void end() throws java.io.IOException, PeripheralNotAvailableException`

  This method demarcates the end of a transaction, hence ending the I^{2}C combined message.

- `void int read() throws java.io.IOException, PeripheralNotAvailableException`

  This method reads one byte of data from this slave device. The byte is returned as an int in the range 0 to 255.

- `int read(byte[] dstBuf, int dstOff, int dstLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads up to `dstLen` bytes of data from this slave device into an array of bytes represented by `dstBuf`, starting at the offset `dstOff` in the array.

- `int read(int skip, byte[] dstBuf, int dstOff, int dstLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads up to `dstLen` bytes of data, skipping the first `skip` bytes, from this slave device into an array of bytes represented by `dstBuf`, starting at the offset `dstOff` in the array.

- `int read(int subaddress, int subaddressSize, byte[] dstBuf, int dstOff, int dstLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads from a subaddress or register address of this slave device, writing `dstLen` bytes into the buffer `dstBuf`, starting at the offset `dstOff`. The most significant bytes (MSB) of the subaddress or register address are transferred first. The `subaddressSize` field represents the size of the subaddress or register address, from 1 to 4 bytes.

- `int read(int subaddress, int subaddressSize, int skip, byte[] dstBuf, int dstOff, int dstLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads from a subaddress or register address of this slave device, after skipping `skip` bytes, and writing `dstLen` bytes into the buffer `dstBuf`, starting at the offset `dstOff`. The most significant bytes (MSB) of the subaddress or register address are transferred first. The `subaddressSize` field represents the size of the subaddress or register address, from 1 to 4 bytes.

- `void write(int srcData) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes one byte to this slave device. The eight low-order bits of the argument data are written; the 24 high-order bits of `srcData` are ignored.

- `void write(byte[] srcBuf, int srcOff, int srcLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes to this slave device `srcLen` bytes from buffer `srcBuf`, starting at the offset `srcOff`.

- `void write(int subaddress, int subaddressSize, byte[] srcBuf, int srcOff, int srcLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes to a subaddress or register address of this slave device `srcLen` bytes from buffer `srcBuf`, starting at the offset `srcOff`. The most significant bytes (MSB) of the subaddress or register address are transferred first. The `subaddressSize` field represents the size of the subaddress or register address, from 1 to 4 bytes.

The I2CDeviceConfig Class

The `I2CDeviceConfig` class encapsulates the configuration parameters of an I^{2}C slave device. An instance of `I2CDeviceConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` method to open the designated I^{2}C slave device with the specified configuration.

The `I2CDeviceConfig` class consists of one constructor and five methods.

- `public I2CDeviceConfig(int busNumber, int address, int addressSize, int clockFrequency)`

  This constructor creates a new `I2CDeviceConfig` with the provided parameters. Unused or not applicable numerical parameters should be set to `PeripheralConfig.DEFAULT`.

- `public int getBusNumber()`

  This method retrieves the configured I^2C bus number the I^2C slave device is connected to.

- `public int getAddress()`

  This method retrieves the configured address of the I^2C slave device.

- `public int getAddressSize()`

  This method retrieves the configured address size of the I^2C slave device: 7 bits or 10 bits or `PeripheralConfig.DEFAULT`.

- `public int getClockFrequency()`

  This method retrieves the configured clock frequency (in Hertz) supported by the I^2C slave device.

9 Memory-Mapped Input/Output

This chapter describes the interfaces and classes for embedded memory-mapped input and output (MMIO).

Memory mapped I/O is typically used for controlling hardware peripherals by reading from and writing to registers or memory blocks mapped to the hardware's system memory. The MMIO API allows for low-level control over the peripheral.

In order to access a specific memory block that a device has been mapped to, an application should first open and obtain an `MMIODevice` instance for the memory-mapped I/O device, using its numerical ID, name, type (interface) or properties. This is an example of using the ID.

```
 MMIODevice device = (MMIODevice) PeripheralManager.open(7);
```

This is an example of using its name and interface.

```
 MMIODevice device = (MMIODevice) PeripheralManager.open("RTC", MMIODevice.class,
null);
```

Once the peripheral is opened, the application can retrieve registers using methods of the `MMIODevice` interface such as the `MMIODevice.getByteRegister(String)` method.

```
 RawByte seconds = (RawByte) device.getByteRegister("Seconds");
```

When done, the application should call the `Peripheral.close()` method to release MMIO device.

```
 device.close();
```

The following code give examples of using the MMIO API to communicate Real Time Clock device.

```
import com.oracle.deviceaccess.PeripheralException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.mmio.MMIODevice;
import com.oracle.deviceaccess.mmio.MMIOEvent;
import com.oracle.deviceaccess.mmio.MMIOEventListener;
import com.oracle.deviceaccess.mmio.RawBlock;
import com.oracle.deviceaccess.mmio.RawByte;
import java.io.IOException;

public class MMIOExample {

    static final int INTERRUPT = 0;
    MMIODevice rtc = null;
    public MMIOExample() {

        try {
            rtc = (MMIODevice) PeripheralManager.open("RTC",
                   MMIODevice.class, (String[]) null);
            //The RTC device has 14 bytes of clock/control registers and 50 bytes
            // of general purpose RAM (see data sheet of the HITACHI HD146818 RTC)
            RawByte seconds = rtc.getByteRegister("Seconds");
            RawByte secAlarm = rtc.getByteRegister("SecAlarm");
            RawByte minutes = rtc.getByteRegister("Minutes");
            RawByte minAlarm = rtc.getByteRegister("MinAlarm");
            RawByte hours = rtc.getByteRegister("Hours");
            RawByte hrAlarm = rtc.getByteRegister("HrAlarm");

            RawByte registerA = rtc.getByteRegister("RegisterA");
            RawByte registerB = rtc.getByteRegister("RegisterB");
            RawByte registerC = rtc.getByteRegister("RegisterC");
            RawByte registerD = rtc.getByteRegister("RegisterD");
            RawBlock userRAM = rtc.getBlock("UserRam");

        } catch (PeripheralException pe) {
        } catch (IOException ioe) {
        } finally {
            if (rtc != null) {
                try {
                    rtc.close();
                } catch (IOException ex) {
                }
            }
        }
    }


    // Sets the daily alarm for after some delay

    public void setAlarm(byte delaySeconds, byte delayMinutes, byte delayHours)
        throws IOException, PeripheralException
   {
        MMIODevice rtc = (MMIODevice) PeripheralManager.open("RTC",
            MMIODevice.class, (String[]) null);
        RawByte seconds = rtc.getByteRegister("Seconds");
        RawByte secAlarm = rtc.getByteRegister("SecAlarm");
        RawByte minutes = rtc.getByteRegister("Minutes");
        RawByte minAlarm = rtc.getByteRegister("MinAlarm");
        RawByte hours = rtc.getByteRegister("Hours");
        RawByte hrAlarm = rtc.getByteRegister("HrAlarm");
        RawByte registerB = rtc.getByteRegister("RegisterB");

        // Directly read from/write to the registers using RawByte instances.
        byte currentSeconds = seconds.get();
        byte currentMinutes = minutes.get();
        byte currentHours = hours.get();
        int i = (currentSeconds + delaySeconds) % 60;
        int j = (currentSeconds + delaySeconds) / 60;
        secAlarm.set((byte) i);
        i = (currentMinutes + delayMinutes + j) % 60;
        j = (currentMinutes + delayMinutes + j) / 60;
        minAlarm.set((byte) i);
        i = (currentHours + delayHours + j) % 24;
        hrAlarm.set((byte) i);
        rtc.setMMIOEventListener(INTERRUPT, new MMIOEventListener() {
            public void eventDispatched(MMIOEvent event) {
                try {
                    MMIODevice rtc = (MMIODevice) event.getPeripheral();
                    RawByte registerC = rtc.getByteRegister("RegisterC");
                    // Check the Alarm Interrupt Flag (AF)
                    if ((registerC.get() & 0X20) != 0) {
                        // Notify application of alarm
                    }
                } catch (IOException ex) {
                } catch (PeripheralNotAvailableException ex) {
                }
            }
        });
        // Set the Alarm Interrupt Enabled (AIE) flag
        registerB.set((byte) (registerB.get() | 0X20));
    }
}
```

Alternatively, in this example, the value of `RegisterC` could be automatically captured upon occurrence of an interrupt request from the Real Time Clock device as follows:

```
rtc.setMMIOEventListener(INTERRUPT, "RegisterC", new MMIOEventListener() {

     public void eventDispatched(MMIOEvent event) {
          byte v = (byte) event.getCapturedRegisterValue();
          // Check the Alarm Interrupt Flag (AF)
          if ((v & 0X20) != 0) {
               // Notify application of alarm
          }
     }
 });
```

MMIO devices are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.mmio` permission allows access to be granted to MMIO devices as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

Note that version 3.3 of the Oracle Java ME Embedded platform has discarded all functions from version 3.2 that employed the `long` datatype.

The MMIODevice Interface

The `MMIODevice` class provides methods to retrieve memory-mapped registers and memory blocks of a peripheral device. Each memory-mapped I/O device is identified by a numerical ID and by a name. An `MMIODevice` instance can be acquired by a call to `MMIOManager.getDevice(int)` or `MMIOManager.getDevice(java.lang.String)`.

With memory-mapped I/O, peripheral devices can be controlled by directly reading or writing to memory areas representing the registers or memory blocks of the peripheral device. Each register or memory block is represented by a `RawMemory` instance. All the mapped registers, including memory blocks, of an MMIO device can be retrieved by a call to the appropriate `get...Registers()` method. The `RawMemory` instance associated to a register has a fixed, determined index in the array returned by those methods.

Each register or memory block is also usually assigned a name that can be used for name-based lookup.

An application can register an `MMIOEventListener` instance to monitor native events of the designated type fired by the peripheral device. To register a `MMIOEventListener` instance, the application must call the `setMMIOEventListener(int, MMIOEventListener)` method. The registered listener can later on be removed by calling the same method with a `null` parameter. Asynchronous notification might not be supported by all memory-mapped devices. An attempt to set a listener on a memory-mapped device that does not supports it throws an `InvalidOperationException`.

The `MMIODevice` interface consists of the following methods:

- `RawBlock getAsRawBlock() throws java.io.IOException, PeripheralNotAvailableException`

  This method retrieves the complete memory area this device is mapped to as a `RawBlock` instance.

- `RawBlock getBlock(java.lang.String name) throws java.io.IOException, PeripheralNotAvailableException`

  This method retrieves the designated memory block.

- `int getByteOrdering() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the byte ordering of this memory-mapped peripheral device. The three possible values are `MMIODevice.BIG_ENDIAN` if big-endian, `MMIODevice.LITTLE_ENDIAN` if little-endian, and `MMIODevice.MIXED_ENDIAN` otherwise.

- `RawByte getByteRegister(java.lang.String name) throws java.io.IOException, PeripheralNotAvailableException`

  This method retrieves the designated register holding a byte value.

- `RawShort getShortRegister(java.lang.String name) throws java.io.IOException, PeripheralNotAvailableException`

  This method retrieves the designated register holding a short value.

- `RawInt getIntRegister(java.lang.String name) throws java.io.IOException, PeripheralNotAvailableException`

  This method retrieves the designated register holding an int value.

- `void setMMIOEventListener(int eventId, MMIOEventListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `MMIOEventListener` instance to monitor native events of the designated type fired by the peripheral device mapped to this `MMIODevice` object. While the listener can be triggered by hardware interrupts, there are no real-time guarantees of when the listener is called. If the listener parameter is `null`, the listener previously registered for the specified event type is removed. Only one listener can be registered at a particular time for a particular event type.

- `void setMMIOEventListener(int eventId, String capturedName, MMIOEventListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `MMIOEventListener` instance to monitor native events of the designated type fired by the peripheral device mapped to this `MMIODevice` object. The `captureName` parameter indicates the name of the register or memory block whose content is to be captured at the time of the underlying event occurs. While the listener can be triggered by hardware interrupts, there are no real-time guarantees of when the listener is called. If the listener parameter is `null,` the listener previously registered for the specified event type is removed. Only one listener can be registered at a particular time for a particular event type.

- `void setMMIOEventListener(int eventId, byte[] captureBuffer, int capturedIndex, int capturedLength, MMIOEventListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `MMIOEventListener` instance to monitor native events of the designated type fired by the peripheral device mapped to this `MMIODevice` object. When the event occurs, the memory is captured in the specified byte buffer, starting at the designated index and length. While the listener can be triggered by hardware interrupts, there are no real-time guarantees of when the listener is called. If the listener parameter is `null,` the listener previously registered for the specified event type is removed. Only one listener can be registered at a particular time for a particular event type.

The MMIOEventListener Interface

The `MMIOEventListener` interface defines methods for getting notified of events fired by peripherals mapped to memory. A `MMIOEventListener` can be registered using the `MMIODevice.setMMIOEventListener(int, MMIOEventListener)` method.

The interface consists of only one method, `void eventDispatched(MMIOEvent event).` This method is invoked when an event is fired by a memory-mapped peripheral.

The RawMemory Interface

The `RawMemory` interface provides generic methods for the different types of raw memory area to which a peripheral device's registers may be mapped.

The interface consists of only one method, `java.lang.String getName()`. This method returns the name assigned to this `RawMemory` instance.

The RawBlock Interface

The `RawBlock` interface provides methods to access a continuous range of physical memory (raw memory). A `RawBlock` instance can be obtained from a `MMIODevice` instance. The index values map to physical memory addresses and are measured in bytes. The index values are relative to the base address of the raw memory area. The index value 0 corresponds to the base address of raw memory area. The byte ordering of the underlying raw memory area can be retrieved using the `MMIODevice.getByteOrdering()` method.

The `RawBlock` interface consists of the following methods:

- `int getSize()`

  This method returns the size in bytes of the raw memory area associated with this object.

- `byte getByte(int index)`

  This method reads the `byte` at the given index in the raw memory area associated with this object.

- `void getBytes(int index, byte[] dst, int offset, int length)`

  This method reads bytes starting at the given index in the raw memory area associated with this object.
- `int getInt(int index)`

  This method reads the `int` at the given index in the raw memory area associated with this object.
- `void getInts(int index, int[] dst, int offset, int length)`

  This method reads integers starting at the given index in the raw memory area associated with this object.
- `short getShort(int index)`

  This method reads the `short` at the given index in the raw memory area associated with this object.
- `void getShorts(int index, short[] dst, int offset, int length)`

  This method reads short integers starting at the given index in the raw memory area associated with this object.
- `void setByte(int index, byte value)`

  This method writes the given `byte` at the given index in the raw memory area associated with this object.
- `void setBytes(int index, byte[] src, int offset, int length)`

  This method writes bytes starting at the given index in the raw memory area associated with this object.
- `void setInt(int index, int value)`

  This method writes the given `int` at the given index in the raw memory area associated with this object.
- `void setInts(int index, int[] src, int offset, int length)`

  This method writes integers starting at the given index in the raw memory area associated with this object.
- `void setShort(int index, short value)`

  This method writes the given `short` at the given index in the raw memory area associated with this object.
- `void setShorts(int index, short[] src, int offset, int length)`

  This method writes short integers starting at the given index in the raw memory area associated with this object.

The RawByte Interface

The `RawByte` interface provides methods for setting and getting the value of a register or memory area holding a byte value. A `RawByte` instance can be obtained from a `MMIODevice` instance.

- `void set(byte value) throws PeripheralNotAvailableException`

  This method sets the byte value at the memory area associated with this object.
- `byte get() throws PeripheralNotAvailableException`

This method retrieves the byte value at the memory area associated with this object.

The RawInt Interface

The `RawInt` interface provides methods for setting and getting the value of a register or memory area holding an int value. A `RawInt` instance can be obtained from a `MMIODevice` instance.

- `void set(int value) throws PeripheralNotAvailableException`

  This method sets the int value at the memory area associated with this object.

- `int get() throws PeripheralNotAvailableException`

  This method retrieves the int value at the memory area associated with this object.

The RawShort Interface

The `RawShort` interface provides methods for setting and getting the value of a register or memory area holding a short value. A `RawShort` instance can be obtained from a `MMIODevice` instance.

- `void set(short value) throws PeripheralNotAvailableException`

  This method sets the short value at the memory area associated with this object.

- `short get() throws PeripheralNotAvailableException`

  This method retrieves the short value at the memory area associated with this object.

The MMIOEvent Class

The `MMIOEvent` class encapsulates events fired by peripherals mapped to memory. The `MMIOEvent` class consists of the following constructors and methods.

- `public MMIOEvent(MMIODevice device, int id)`

  This constructor creates a new `MMIOEvent` with the specified device and ID. It is then time-stamped with the current time.

- `public MMIOEvent(MMIODevice device, int id, long timeStamp, int timeStampMicros)`

  This constructor creates a new `MMIOEvent` with the specified device, ID and timestamp.

- `public MMIOEvent(MMIODevice device, int id, int capturedRegisterValue, long timeStamp, int timeStampMicros)`

  This constructor creates a new `MMIOEvent` with the specified value and timestamp. The `capturedRegisterValue` parameter is the captured value of the register designated upon registration, specified as a 32-bit integer.

- `public MMIOEvent(MMIODevice device, int id, byte[] capturedMemoryContent, long timeStamp, int timeStampMicros)`

  This constructor creates a new `MMIOEvent` with the specified value and timestamp. The `capturedRegisterContent` parameter is the captured content of the memory area or memory block designated upon registration.

- `public int getID()`

This method returns the event ID.

- `public byte[] getCapturedMemoryContent()`

  This method returns the captured content of the memory area or block; or `null` if no memory area or block content was captured.

- `public int getCapturedRegisterValue()`

  This method returns the captured value of the register designated upon registration as a 32-bit integer.

The MMIODeviceConfig Class

The `MMIODeviceConfig` class encapsulates the hardware addressing information, and static and dynamic configuration parameters of an MMIO device.

Some hardware addressing parameter, and static and dynamic configuration parameters may be set to `PeripheralConfig.DEFAULT`. Whether such default settings are supported is platform- as well as peripheral driver-dependent.

An instance of `MMIODeviceConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` or `PeripheralManager.open(Class, PeripheralConfig)` method to open the designated MMIO device with the specified configuration. A `PeripheralConfigInvalidException` is thrown when attempting to open a peripheral device with an invalid or unsupported configuration.

The `MMIODeviceConfig` class itself contains three nested classes.

- `static class MMIODeviceConfig.RawBlockConfig`

  The `RawBlockConfig` class encapsulates the configuration parameters of a memory block.

- `static class MMIODeviceConfig.RawMemoryConfig`

  The `RawMemoryConfig` class encapsulates the configuration parameters of a generic raw memory area.

- `static class MMIODeviceConfig.RawRegisterConfig`

  The `RawRegisterConfig` class encapsulates the configuration parameters of a register.

The `MMIODeviceConfig` class also contains three constants.

- `public static final int REGISTER_TYPE_BYTE`

  This is the type for a register holding a byte value.

- `public static final int REGISTER_TYPE_INT`

  This is the type for a register holding an integer value.

- `public static final int REGISTER_TYPE_SHORT`

  This is the type for a register holding a short integer value.

Finally, the `MMIODeviceConfig` class consists of one constructor and four accessors.

- `public MMIODeviceConfig(long address, int size, int byteOrdering, MMIODeviceConfig.RawMemoryConfig[] memConfigs)`

  This constructor creates a new `MMIODeviceConfig` with the specified hardware addressing information and configuration parameters. Note that if no raw block and raw register configuration is provided, the specified memory area will be mapped to the `RawBlock` instance returned by a call to `MMIODevice.getAsRawBlock()`.

- `public long getAddress()`

  This method returns the configured memory address of the MMIO device.

- `public int getByteOrdering()`

  This method returns the configured byte ordering of the MMIO device.

- `public MMIODeviceConfig.RawMemoryConfig[] getRawMemoryConfigs()`

  This method returns the set of configured registers and memory blocks.

- `public int getSize()`

  This method returns the configured size of the memory-mapped area of the MMIO device.

The MMIODeviceConfig.RawMemoryConfig Class

The abstract `MMIODeviceConfig.RawMemoryConfig` class encapsulates the configuration parameters of a generic raw memory area. The abstract class consists of two methods.

- `public String getName()`

  This method returns the configured name for the raw memory area.

- `public int getOffset()`

  This method returns the configured offset of the raw memory area from the base address.

The MMIODeviceConfig.RawBlockConfig Class

The `MMIODeviceConfig.RawBlockConfig` class extends the abstract `MMIODeviceConfig.RawMemoryConfig` class and encapsulates the configuration parameters of a memory block. The class consists of one constructor and one accessor.

- `public MMIODeviceConfig.RawBlockConfig(int offset, java.lang.String name, int size)`

  This constructor creates a new `RawBlockConfig` with the provided parameters.

- `public int getSize()`

  This method returns the configured size in bytes of the memory block.

The MMIODeviceConfig.RawRegisterConfig Class

The `MMIODeviceConfig.RawRegisterConfig` class extends the abstract `MMIODeviceConfig.RawMemoryConfig` class and encapsulates the configuration parameters of a register. The class consists of one constructor and one accessor.

- `public MMIODeviceConfig.RawRegisterConfig(int offset, java.lang.String name, int type)`

  This constructor creates a new `RawRegisterConfig` with the provided parameters.

- `public int getType()`

  This method returns the configured type of the value held by the register. See the constants in the `MMIODeviceConfig` for possible values.

The MMIOEvent Class

The `MMIOEvent` class encapsulates events fired by peripherals mapped to memory..The class consists of four constructors and three methods.

- `public MMIOEvent(MMIODevice device, int id)`

  This constructor creates a new `MMIOEvent` with the specified value and time-stamped with the current time.

- `public MMIOEvent(MMIODevice device, int id, byte[] capturedMemoryContent, long timeStamp, int timeStampMicros)`

  This constructor creates a new `MMIOEvent` with the specified values and timestamp.

- `public MMIOEvent(MMIODevice device, int id, int capturedRegisterValue, long timeStamp, int timeStampMicros)`

  This constructor creates a new `MMIOEvent` with the specified values and timestamp.

- `public MMIOEvent(MMIODevice device, int id, long timeStamp, int timeStampMicros)`

  This constructor creates a new `MMIOEvent` with the specified value and timestamp.

- `public byte[] getCapturedMemoryContent()`

  This method returns the captured content of the memory area or memory block designated upon registration.

- `public int getCapturedRegisterValue()`

  This method returns the captured value of the register designated upon registration as a 32-bit integer.

- `public int getID()`

  This method returns the event ID.

AccessOutOfBoundsException

`AccessOutOfBoundsException` is an exception that is thrown by an instance of `RawBlock` if the offset used is out of valid boundary of the specified memory block.

10 Modem Control Signals

The `com.oracle.deviceaccess.modem` package contains interfaces and classes for controlling modem signals.

The ModemSignalListener Interface

The `ModemSignalListener` interface defines methods for being notified of modem signal changes.

The `ModemSignalListener` interface contains one method.

- `void signalStateChanged(ModemSignalEvent event)`

  This method is invoked when the state of a modem signal has changed.

The ModemSignalsControl Class

The `ModemSignalsControl` interface provides methods for controlling and monitoring modem signals.

The `ModemSignalsControl` class consists of six constants:

- `static final int DTR_SIGNAL`

  This constant represents the Data Terminal Ready (DTR) signal. This bit flag can be bitwise-combined (OR) with other signal bit flags.

- `static final int DCD_SIGNAL`

  This constant represents the Data Carrier Detect (DCD) signal. This bit flag can be bitwise-combined (OR) with other signal bit flags.

- `static final int DSR_SIGNAL`

  This constant represents the Data Set Ready (DSR) signal. This bit flag can be bitwise-combined (OR) with other signal bit flags.

- `static final int RI_SIGNAL`

  This constant represents the Ring Indicator (RI) signal. This bit flag can be bitwise-combined (OR) with other signal bit flags.

- `static final int RTS_SIGNAL`

  This constant represents the Ready To Send (RTS) signal. This bit flag can be bitwise-combined (OR) with other signal bit flags.

- `static final int CTS_SIGNAL`

This constant represents the Clear To Send (CTS) signal. This bit flag can be bitwise-combined (OR) with other signal bit flags.

The `ModemSignalsControl` class also consists of three methods:

- `void setSignalState(int signalID, boolean state) throws java.io.IOException,PeripheralNotAvailableException`

  This method sets or clears the designated signal.

- `boolean getSignalState(int signalID) throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the state of the designated signal.

- `void setSignalChangeListener(ModemSignalListener listener, int signals) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `ModemSignalListener` instance which will get asynchronously notified when one of the designated signals changes. Notification will automatically begin after registration completes. If listener is `null` then the previously registered listener will be removed. Only one listener can be registered at a particular time.

The ModemSignalEvent Class

The `ModemSignalEvent` class encapsulates modem signal state changes. If signal state change events for the same peripheral are coalesced the value retained is that of the last occurrence.

The `ModemSignalEvent` class consists of two constants:

- `protected int signalID`

  This constant represents the signal ID.

- `protected boolean signalState`

  This constant represents the signal state.

The `ModemSignalEvent` class also consists of two constructors and several methods:

- `public ModemSignalEvent(Peripheral peripheral, int signalID, boolean signalState)`

  This constructor creates a new `ModemSignalEvent` with the specified value. It is then time-stamped with the current time.

- `public ModemSignalEvent(Peripheral peripheral, int signalID, boolean signalState, long timeStamp, int timeStampMicros)`

  This constructor creates a new `ModemSignalEvent` with the specified value and timestamp.

- `public int getSignalID()`

  This method returns the signal ID.

- `public boolean getSignalState()`

  This method returns the new signal state.

11 Power Management

The com.oracle.deviceaccess.power package contains interfaces and classes for power management of peripheral devices. A Peripheral implementing class may implement the PowerManaged interface if the underlying peripheral device supports some form of power management and saving states that can be mapped to the states defined by this API.

Example 11–1 demonstrates how to use the power management API.

Example 11–1 Using the Power Management APIs

```
import com.oracle.deviceaccess.Peripheral;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.adc.ADCChannel;
import com.oracle.deviceaccess.adc.MonitoringEvent;
import com.oracle.deviceaccess.adc.MonitoringListener;
import com.oracle.deviceaccess.power.PowerManaged;
import com.oracle.deviceaccess.power.PowerSavingHandler;
import java.io.IOException;

class SignalLevelMonitor implements MonitoringListener, PowerSavingHandler {

    private ADCChannel channel = null;
    private boolean inRange = false;

    public void start(int channelID, int low, int high) throws
            IOException, PeripheralNotAvailableException,
            PeripheralNotFoundException
    {
        channel = (ADCChannel) PeripheralManager.open(channelID);
        channel.setSamplingInterval(1000); // every 1 seconds
        channel.startMonitoring(low, high, this);
        if (channel instanceof PowerManaged) {
            ((PowerManaged) channel).enablePowerSaving(
                  PowerManaged.LOW_POWER, this);
            // Only enable LOW_POWER saving mode (POWER_ON is implicit)
        }
    }

    public void thresholdReached(MonitoringEvent event) {
        inRange = (event.getType() == MonitoringEvent.BACK_TO_RANGE);
    }

    public long handlePowerStateChangeRequest(Peripheral peripheral,
            int currentState, int requestedState, long duration)
    {
        if (requestedState == PowerManaged.LOW_POWER) {
            return inRange ? duration : 0;
            // Only accept to change to LOW_POWER if signal is back in range
        }
        return duration; // Accept returning to POWER_ON
    }

    public void stop() throws IOException, PeripheralNotAvailableException {
        if (channel != null) {
            channel.stopMonitoring();
            if (channel instanceof PowerManaged) {
                ((PowerManaged) channel).disablePowerSaving();
            }
            channel.close();
        }
    }
}
```

As any other peripheral devices, peripheral devices that can be power-managed are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.power` permission allows access to be granted to peripheral power management. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The PowerManaged Interface

The `PowerManaged` interface provides methods that a `Peripheral` class may implement to control how the underlying peripheral hardware resource is managed by the power management facility of the device.

The power management states defined are peripheral device as well as host device-dependent. For peripherals on a microcontroller unit, there may be no distinction between `POWER_OFF`, `LOW_POWER` and `LOWEST_POWER` and they may all be supported by clock-gating the unused peripherals. Conversely, a peripheral device external to the host device could support the four power management modes and could be powered off.

A power state change may be ordered by the power management facility of the device, or it may be requested by the power management facility on behalf of the application itself or of another application using the method `requestPowerStateChange(int, int)`. A power state change for a specific peripheral device may be requested by another application if the peripheral device or some of the underlying peripheral device hardware resources are shared. This is the case on a `GPIOPin` instance, for example: another application may have opened a different GPIO pin controlled by the same GPIO controller; the application will get notified of any power state changes requested by the other application.

An application may register to be notified of power state changes. When notified, the application may take the following actions:

1. The application may save or restore the state or configuration of the peripheral if needed. Saving the peripheral's state or configuration may be needed when the application is being notified of a power state change requested by another

application on a peripheral device with hardware resources shared with the current application. The saving and restoration of the peripheral's state or configuration may be needed when changing to or from `POWER_OFF` or `LOWEST_POWER` to `POWER_ON`, as the peripheral context may not be preserved.

2. The application may veto a power state change. For example, the application may veto a power state change from `POWER_ON` to `LOWEST_POWER` if the application is currently using or is about to use the designated peripheral.
3. The application may grant a shorter power state change duration. For example, the application may grant a duration of a power state change from `POWER_ON` to `LOWEST_POWER` shorter than the specified duration if the application anticipates it will use the designated peripheral earlier than the specified duration.

If application-dictated power saving for a peripheral device is not explicitly enabled by a call to one of the `enablePowerSaving()` method, the default power saving strategy of the platform applies. This strategy is both platform-dependent and implementation-dependent. It may define power saving rules, such as changing the power state of a peripheral device when certain conditions are met, that may or may not differ from peripheral device to peripheral device. It may, for example, forcefully change all peripherals' power state to `LOWEST_POWER` upon some condition; in such a situation, attempting to access the peripheral without restoring its state or configuration may result in unexpected behavior. Therefore an application should always either:

1. Register for power state changes on the peripherals it uses.
2. Register for system-wide power state changes (if supported by the platform) and close the peripherals when going to power saving modes that may not preserve the peripheral context and then open again the peripherals when returning from such power saving modes.

The `PowerManaged` interface contains five constants.

- `static final int UNLIMITED_DURATION`

  This constant represents unlimited or unknown power state change requested duration.

- `static final int POWER_ON`

  This constant represents the device as fully powered on. This bit flag can be bitwise-combined (OR) with other power state bit flags.

- `static final int LOW_POWER`

  This constant represents the device in a low power mode. It may save less power while preserving more peripheral device context than `LOWEST_POWER`, hence allowing for a faster return to full performance. When transitioning from this state to `POWER_ON` no state or configuration restoration of the peripheral device must be needed. This bit flag can be bitwise-combined (OR) with other power state bit flags.

- `static final int LOWEST_POWER`

  This constant represents the lowest power mode. In this mode, the device may save more power while preserving less peripheral device context/state than `LOW_POWER`, hence only allowing for a slower return to full performance. When transitioning from this state to `POWER_ON` some state or configuration restoration of the peripheral device may be needed. This state or configuration restoration of the peripheral device may be handled by a `PowerSavingHandler`. This bit flag can be bitwise-combined (OR) with other power state bit flags.

- `static final int POWER_OFF`

  This constant represents that the power has been fully removed from the device. When transitioning from this state to `POWER_ON` a complete state or configuration restoration of the peripheral device may be needed. This state or configuration restoration of the peripheral device may be handled by a `PowerSavingHandler`. This bit flag can be bitwise-combined (OR) with other power state bit flags.

The `PowerManaged` interface also contains five methods.

- `void enablePowerSaving(int powerStates) throws java.io.IOException, PeripheralNotAvailableException`

  This method enables application-dictated power saving for the `Peripheral` instance. Note that the `POWER_ON` state is always implicitly enabled.

- `void enablePowerSaving(int powerStates, PowerSavingHandler handler) throws java.io.IOException, PeripheralNotAvailableException`

  This method enables application-dictated power saving for the `Peripheral` instance and registers a `PowerSavingHandler` instance to get asynchronously notified when the power management facility is about to change the power state of the `Peripheral` instance. This in turn allows the application to veto the power state change on the peripheral. Note that the `POWER_ON` state is always implicitly enabled.

- `void disablePowerSaving() throws java.io.IOException, PeripheralNotAvailableException`

  This method disables application-dictated power saving for the `Peripheral` instance. The power saving strategy of the platform applies. If a `PowerSavingHandler` instance was registered using `enablePowerSaving(int, com.oracle.deviceaccess.power.PowerSavingHandler)`, it will be unregistered.

- `int getPowerState() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the current power state of the `Peripheral` instance. If application-dictated power saving is disabled using the `disablePowerSaving()` method, the power state depends on the power saving strategy of the platform.

- `int requestPowerStateChange(int powerState, int duration) throws java.io.IOException, PeripheralNotAvailableException`

  This method requests the change of the peripheral's current power state to the specified power state. If a `PowerSavingHandler` instance is registered, it will be notified.

The PowerSavingsHandler Class

The `PowerSavingHandler` interface defines methods for getting notified of power state change requests on a specific `Peripheral` instance. A `PowerSavingHandler` can be registered using the `PowerManaged.enablePowerSaving(int, com.oracle.deviceaccess.power.PowerSavingHandler)` method.

The `PowerSavingHandler` class consists of one method:

- `long handlePowerStateChangeRequest(Peripheral peripheral, int currentState, int requestedState, long duration)`

  This method is invoked to allow the application to handle a power state change request on the designated `Peripheral` instance. The application may veto the power state change by returning the number zero (0). Otherwise, it should return a duration lesser or equals to the proposed state change duration. An application may veto altogether a power state change from `PowerManaged.POWER_ON` to `PowerManaged.LOWEST_POWER` if, for example, the application is currently using or is about to use the designated peripheral. An application may grant a power state change duration lesser than the specified duration if for example the application anticipates it will use the designated peripheral earlier than the specified duration.

12 Serial Peripheral Interface Bus

This package provides interfaces and classes for SPI (Serial Peripheral Interface Bus) device access.

The Serial Peripheral Interface (SPI) bus is a synchronous serial data link standard that operates in full duplex mode. Devices communicate in master or slave mode, where the master device initiates the communication data frame. Multiple slave devices are allowed with individual slave select lines.

The functionalities supported by this API are those of an SPI master. In order to communicate with a specific slave, an application should first open and obtain an `SPIDevice` instance for the SPI slave device the application wants to exchange data with, using its numerical ID, name, type (interface), or properties. The following example demonstrates how to obtain an SPI device using its ID.

```
SPIDevice slave = (SPIDevice) PeripheralManager.open(3);
```

This example demonstrates how to access a device using its name and interface.

```
 SPIDevice slave = (SPIDevice) PeripheralManager.open("RTC1",
     SPIDevice.class, null);
```

Once the peripheral opened, the application can exchange data with the SPI slave device using methods of the `SPIDevice` interface such as the `writeAndRead()` method.

```
slave.writeAndRead(sndBuf, 0, 1, rcvBuf, 0, 1);
```

When the data exchange is over, the application should call the `Peripheral.close()` method to release the SPI slave device.

```
slave.close();
```

Example 12–1 shows how to use the SPI API to communicate with SPI slaves.

Example 12–1 Using the SPI APIs to Communicate with SPI Slaves

```
import com.oracle.deviceaccess.PeripheralException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.spibus.SPIDevice;
import java.io.IOException;

public class SPIExample {

    SPIDevice slave = null;

    public SPIExample() {
        try {
            slave = (SPIDevice) PeripheralManager.open("SPI1",
                SPIDevice.class, (String[]) null);
            byte[] sndBuf1 = {0x01};
            byte[] sndBuf2 = {0x02};
            byte[] rcvBuf = new byte[3];
            slave.writeAndRead(sndBuf1, 0, sndBuf1.length, rcvBuf, 0, 1);
                // received data will be stored in rcvBuf[0]
            slave.writeAndRead(sndBuf2, 0, sndBuf2.length, rcvBuf, 1, 2);
                // received data will be stored in rcvBuf[1] and rcvBuf[2]

        } catch (PeripheralException pe) {
            // Handle exception
        } catch (IOException ioe) {
            // Handle exception
        } finally {
            if (slave != null) {
                try {
                    slave.close();
                } catch (IOException ex) {
                }
            }
        }
    }
}
```

Information about the SPI-bus specification can be found at `http://www.freescale.com/files/microcontrollers/doc/ref_manual/M68HC11RM.pdf`.

SPI slave devices are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.spi` permission allows access to be granted to SPI slave devices as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The SPIDevice Interface

The `SPIDevice` interface provides methods for transmitting and receiving data to and from an SPI slave device. Each SPI slave device is identified by a numerical ID and by a name.

An `SPIDevice` instance can be opened by a call to one of the `PeripheralManager.open()` methods. On an SPI bus, data is transferred between the SPI master device and an SPI slave device in full duplex. That is, data is transmitted by the SPI master to the SPI slave device at the same time data is received from the SPI slave device by the SPI master.

To perform such a bidirectional exchange of data with an SPI slave device, an application may use one of the `writeAndRead()` methods. When an application only wants to send data to or receive data from an SPI slave device, it may use a `write()` or `read()` method, respectively. When writing only, the data received from the SPI slave device will be ignored and discarded. When reading only, dummy data will be sent to the slave.

A data exchange consists of words of a certain length which may vary from one SPI slave device to another. Words in the sending and receiving byte buffers are not packed bit-wise and must be byte-aligned. The most significant bits of a word are stored at the lower index (first). If a word's bit length is not a multiple of eight, then the most significant bits will be undefined when receiving or unused when sending. If the designated portion of a sending or receiving byte buffer cannot contain a positive integral number of words then an `InvalidWordLengthException` is thrown. For example, if the word length is 16 bits and the designated portion of buffer is only 1-byte long or 3-bytes long, an `InvalidWordLengthException` is thrown.

Assuming a word length `w`, the length `l` of the designated portion of the sending or receiving byte buffer must be such that:

```
((l % (((w - 1) / 8) + 1)) == 0)
```

When the data exchange is over, an application should call the `SPIDevice.close()` method to release the SPI slave device. Any attempt to read or write to an SPI slave device which has been closed will thow a `PeripheralNotAvailableException`.

The following methods are contained in the `SPIDevice` interface.

- `void begin()`

  This method demarcates the beginning of an SPI transaction so that this slave's Select line (SS) will be remain asserted during the subsequent read and write operations and until the transaction ends.

- `void end()`

  This method demarcates the end of a transaction, hence ending the assertion of this slave's Select line (SS).

- `int getWordLength() throws java.io.IOException, PeripheralNotAvailableException`

  This method retrieves the transfer word length in bits supported by this slave device. If the length of data to be exchanged is not a multiple of this word length, an `InvalidWordLengthException` is thrown.

- `int read() throws java.io.IOException, PeripheralNotAvailableException`

  This method reads one data word of up to 32 bits from this slave device.

- `int read(byte[] rxBuf, int rxOff, int rxLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads data from this slave device. During the duplex, dummy data is sent to this slave device by the platform. The length of the designated portion of the sending byte buffers must correspond to a (positive) integral number of words. This slave's Select line (SS) is asserted for the duration of the reception.

- `int read(int rxSkip, byte[] rxBuf, int rxOff, int rxLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method reads up to `rxLen` bytes of data from this slave device into an array of bytes skipping the first `rxSkip` bytes read.

- `void write(int txData) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes one data word of up to 32 bits to this slave device.

- `void write(byte[] txBuf, int txOff, int txLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method writes to the slave device `txLen` bytes from buffer `txBuf`.

- `int writeAndRead(int txData) throws java.io.IOException, PeripheralNotAvailableException`

This method exchanges (send and receives) one data word of up to 32 bits with this slave device. This slave's Select line (SS) is asserted for the duration of the exchange.

- `int writeAndRead(byte[] txBuf, int txOff, int txLen, byte[] rxBuf, int rxOff, int rxLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method exchanges (send and receives) data with this slave device. The designated portions of the sending and receiving byte buffers might not have the same length. When sending more than is being received, the extra received bytes are ignored and discarded. Conversely, when sending less than is being received, extra dummy data are sent. This slave's Select line (SS) is asserted for the duration of the exchange

- `int writeAndRead(byte[] txBuf, int txOff, int txLen, int rxSkip, byte[] rxBuf, int rxOff, int rxLen) throws java.io.IOException, PeripheralNotAvailableException`

  This method exchanges (send and receives) data with this slave device skipping the specified number of bytes received. The designated portions of the sending and receiving byte buffers might not have the same length. When sending more than is being received, the extra received bytes are ignored and discarded. Conversely, when sending less than is being received, extra dummy data are sent. This slave's Select line (SS) is asserted for the duration of the exchange.

The SPIDeviceConfig Class

The `SPIDeviceConfig` class encapsulates the configuration parameters of an SPI slave device.

Each SPI slave device has a clock mode. The clock mode is a number from 0 to 3 which represents the combination of the CPOL (SPI Clock Polarity Bit) and CPHA (SPI Clock Phase Bit) signals, where CPOL is the high order bit and CPHA is the low order bit, as shown in Table 12–1.

Table 12–1 Clock Modes in the SPIDeviceConfig Class

Mode	CPOL	CPHA
0	0 = Active-high clocks selected.	0 = Sampling of data occurs at odd edges of the SCK clock
1	0 = Active-high clocks selected.	1 = Sampling of data occurs at even edges of the SCK clock
2	1 = Active-low clocks selected.	0 = Sampling of data occurs at odd edges of the SCK clock
3	1 = Active-low clocks selected.	1 = Sampling of data occurs at even edges of the SCK clock

An instance of `SPIDeviceConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` method to open the designated SPI slave device with the specified configuration.

The `SPIDeviceConfig` class consists of one constructor and the following methods

- `SPIDeviceConfig(int busNumber, int address, int clockFrequency, int clockMode, int wordLength, int bitOrdering)`

This constructor creates a new `SPIDeviceConfig` with the provided parameters. Note that unused or not applicable numerical parameters should be set to `PeripheralConfig.DEFAULT`.

- `public int getBusNumber()`

  This method returns the configured SPI bus number the slave is connected to.

- `public int getAddress()`

  This method returns the configured address of the SPI slave device.

- `public int getClockFrequency()`

  This method returns the clock frequency (in Hertz) supported by the SPI slave device.

- `public int getClockMode()`

  This method returns the configured clock mode (combining clock polarity and phase) for communicating with the SPI slave device. See Table 12–1 for more information on possible values for the clock mode.

- `public int getWordLength()`

  This method returns the configured word length for communicating with the SPI slave device.

- `public int getBitOrdering()`

  This method returns the configured bit (shifting) ordering of the SPI slave device, one of: `Peripheral.BIG_ENDIAN`, `Peripheral.LITTLE_ENDIAN` or `PeripheralConfig.DEFAULT`.

InvalidWordLengthException

The `InvalidWordLengthException` is thrown by an instance of `SPIDevice` in case of mismatch between the length of data to be exchanged and the slave's word length as indicated by `SPIDevice.getWordLength()`.

13
UART

The `com.oracle.deviceaccess.uart` package contains interfaces and classes for controlling and reading and writing from/to Universal Asynchronous Receiver/Transmitter (UART), with optional Modem signals control. In order to access and control a specific UART device, an application should first open and obtain an UART instance for the UART device using its numerical ID, name, type (interface) and/or properties.

Example 13–1 and Example 13–2 demonstrate how to use the UART APIs to communicate with a host.

Example 13–1 Communicating using the UART API

```
import com.oracle.deviceaccess.PeripheralException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.uart.UART;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class UARTExample1 {

    UART host = null;
    InputStream is = null;
    OutputStream os = null;

    public UARTExample1() {
        try {
            host = (UART) PeripheralManager.open("HOST", UART.class,
                (String[]) null);
            is = host.getInputStream();
            os = host.getOutputStream();
            StringBuffer cmd = new StringBuffer();
            int c = 0;
            while (true) {
                os.write('$');
                os.write(' '); // echo prompt
                while (c != '\n' && c != '\003') { // echo input
                    c = is.read();
                    os.write(c);
                    cmd.append(c);
                }
                if (c == '\003') { // CTL-C
                    break;
                }
                // process(cmd);
            }
        } catch (IOException ioe) {
            // Handle exception
        } catch (PeripheralException pe) {
            // Handle exception
        } finally {
            if (is != null) {
                try {
                    is.close();
                } catch (IOException ex) {
                }
            }
            if (os != null) {
                try {
                    os.close();
                } catch (IOException ex) {
                }
            }
            if (host != null) {
                try {
                    host.close();
                } catch (IOException ex) {
                }
            }
        }
    }
}
```

Example 13–2 Using a ModemUART to Communicate

```
import com.oracle.deviceaccess.PeripheralException;
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.modem.ModemSignalEvent;
import com.oracle.deviceaccess.modem.ModemSignalListener;
import com.oracle.deviceaccess.modem.ModemSignalsControl;
import com.oracle.deviceaccess.uart.ModemUART;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class UARTExample2 {

    ModemUART modem = null;
    InputStream is = null;
    OutputStream os = null;

    public UARTExample2() {
        try {
            modem = (ModemUART) PeripheralManager.open("HOST",
                       ModemUART.class, (String[]) null);
            is = modem.getInputStream();
            os = modem.getOutputStream();
            modem.setSignalChangeListener(new ModemSignalListener() {
                public void signalStateChanged(ModemSignalEvent event) {
                    if (event.getSignalState() == false) {
                        ModemUART modem = (ModemUART) event.getPeripheral();
                        // Process MODEM hang-up...
                    }
                }
            }, ModemSignalsControl.DCD_SIGNAL);
            // Process input and output...
        } catch (IOException ioe) {
            // Handle exception
        } catch (PeripheralException pe) {
            // Handle exception
        } finally {
            // Close UART, and input and output streams
        }

    }
}
```

Note that the preceding example is using a *try-with-resources* statement and that the `UART.close()`, `InputStream.close()` and `OutputStream.close()` methods are automatically invoked by the platform at the end of the statement.

UARTs are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.uart` permission allows access to be granted to UART devices as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The ModemUART Interface

The ModemUART interface provides methods for controlling and accessing a UART (Universal Asynchronous Receiver/Transmitter) with Modem control lines.

Note that even if CTS/RTS hardware flow control is enabled using the `UARTConfig.getFlowControlMode()` method, registering for notification of CTS signal state changes (see `ModemSignalsControl.setSignalChangeListener(com.oracle.deviceaccess.modem.ModemSignalListener, int)` may not always be supported. Additionally, when supported, CTS signal state change notification may only be indicative because of latency: CTS flow control may be handled directly by the hardware or by the native driver.

The UART Interface

The UART interface provides methods for controlling and accessing a UART (Universal Asynchronous Receiver/Transmitter). Each UART device is identified by a numerical ID and by a name. A UART instance can be opened by a call to one of the PeripheralManager.open() methods.

Once opened, an application can obtain an input stream and an output stream using the `getInputStream()` and `getOutputStream()` methods and can then read the received data bytes and respectively write the data bytes to be transmitted through the UART.

An application can register a `UARTEventListener` instance which will get asynchronously notified of input data availability, input buffer overrun, or empty output buffer conditions. Note that the input and output buffers for which these events may be notified may not necessarily correspond to the transmit and receive FIFO buffers of the UART hardware, but may be buffers allocated by the underlying native driver. To register a `UARTEventListener` instance, the application must call the `setEventListener(int, UARTEventListener)` method. The registered listener can later on be removed by calling the same method with a `null` listener parameter.

When done, an application should call the `UART.close()` method to release the UART. Any further attempt to access or control a UART which has been closed will result in a `PeripheralNotAvailableException` been thrown.

The `UART` interface consists of several methods:

- `void close() throws java.io.IOException, PeripheralNotAvailableException`

  This method closes and releases the underlying peripheral device, making it available to other applications. Once released, subsequent operations on that very same `Peripheral` instance will throw a `PeripheralNotAvailableException`. This method has no effects if the peripheral device has already been closed. Note that closing a UART will also close the device's `InputStream` and `OutputStream`..

- `int getBaudRate() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the current baud rate. If the baud rate was not set previously using the `setBaudRate(int)` method, the peripheral configuration-specific default value is returned.

- `void setBaudRate(int baudRate) throws java.io.IOException, PeripheralNotAvailableException`

  This method sets the baud rate.

- `int getDataBits() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the current number of bits per character.

- `void setDataBits(int dataBits) throws java.io.IOException, PeripheralNotAvailableException`

  This method sets the number of bits per character.

- `int getParity() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the current parity.

- `void setParity(int parity) throws java.io.IOException, PeripheralNotAvailableException`

  This method sets the parity.

- `int getStopBits() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the current number of stop bits per character.

- `void setStopBits(int stopBits) throws java.io.IOException, PeripheralNotAvailableException`

  This method sets the number of stop bits per character.

- `java.io.InputStream getInputStream() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns this UART's input stream. The same `InputStream` instance is returned upon subsequent calls. Note that if this UART's input stream has been previously closed, this method returns the closed input stream without attempting to re-open it.

- `java.io.OutputStream getOutputStream() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns this UART's output stream. The same `OutputStream` instance is returned upon subsequent calls. Note that if this UART's output stream has been previously closed, this method returns the same closed output stream without attempting to re-open it.

- `void setEventListener(int eventId, UARTEventListener listener) throws java.io.IOException, PeripheralNotAvailableException`

  This method registers a `UARTEventListener` instance to monitor input data availability, input buffer overrun and/or empty output buffer conditions. While the listener can be triggered by hardware interrupts, there are no real-time guarantees of when the listener will be called. A list of event type IDs is defined in `UARTEvent`. If listener is `null` then listener previously registered for the specified event type will be removed. Only one listener can be registered at a particular time for a particular event type.

The UARTEventListener Interface

The `UARTEventListener` interface defines methods for getting notified of events fired by peripheral devices that implement the UART interface. A `UARTEventListener` can be registered using the `UART.setEventListener(int, UARTEventListener)` method.

The `UARTEventListener` interface consists of only one method:

- `void eventDispatched(UARTEvent event)`

  This method is invoked when an event is fired by peripheral device.

The UARTConfig Class

The `UARTConfig` class encapsulates the hardware addressing information, as well as the static and dynamic configuration parameters, of a UART. Some hardware addressing parameter, and static and dynamic configuration parameters, may be set to `PeripheralConfig.DEFAULT`. Whether such default settings are supported is both platform-dependent and peripheral driver-dependent. An instance of `UARTConfig` can be passed to the `PeripheralManager.open(PeripheralConfig)` or `PeripheralManager.open(Class, PeripheralConfig)` method to open the designated UART with the specified configuration. A `PeripheralConfigInvalidException` is thrown when attempting to open a peripheral device with an invalid or unsupported configuration.

The `UARTConfig` class consists of a number of static integer constants, as shown in Table 13–1.

Table 13–1 UARTConfig Constants

Constant	Description
`DATABITS_5`	5 data bit format
`DATABITS_6`	6 data bit format
`DATABITS_7`	7 data bit format
`DATABITS_8`	8 data bit format

Table 13–1 (Cont.) UARTConfig Constants

Constant	Description
`DATABITS_9`	9 data bit format
`FLOWCONTROL_NONE`	Flow control off
`FLOWCONTROL_RTSCTS_IN`	RTS/CTS (hardware) flow control on input
`FLOWCONTROL_RTSCTS_OUT`	RTS/CTS (hardware) flow control on output
`FLOWCONTROL_XONXOFF_IN`	XON/XOFF (software) flow control on input
`FLOWCONTROL_XONXOFF_OUT`	XON/XOFF (software) flow control on output
`PARITY_EVEN`	EVEN parity scheme
`PARITY_MARK`	MARK parity scheme
`PARITY_NONE`	No parity bit
`PARITY_ODD`	ODD parity scheme
`PARITY_SPACE`	SPACE parity scheme
`STOPBITS_1`	Number of STOP bits is 1
`STOPBITS_1_5`	Number of STOP bits is 1.5
`STOPBITS_2`	Number of STOP bits is 2

The `UARTConfig` class consists of two constructors and several methods:

- `public UARTConfig(int uartNumber, int baudRate, int dataBits, int parity, int stopBits, int flowcontrol)`

  This constructor creates a new `UARTConfig` with the specified hardware addressing information and configuration parameters.

- `public UARTConfig(int uartNumber, int baudRate, int dataBits, int parity, int stopBits, int flowcontrol, int inputBufferSize, int outputBufferSize)`

  This constructor creates a new `UARTConfig` with the specified hardware addressing information and configuration parameters. The platform or underlying driver may or may not allocate the requested sizes for the input and output buffers.

- `public int getUARTNumber()`

  This method returns the configured UART number.

- `public int getFlowControlMode()`

  This method returns the configured flow control mode.

- `public int getBaudRate()`

  This method returns the configured default/initial speed in Bauds.

- `public int getDataBits()`

  This method returns the configured default/initial number of bits per character.

- `public int getParity()`

  This method returns the configured default/initial parity.

- `public int getStopBits()`

This method returns the configured default/initial number of stop bits per character.

- `public int getInputBufferSize()`

  This method returns the requested input buffer size. The platform/underlying driver may or may not allocate the requested size for the input buffer.

- `public int getOutputBufferSize()`

  This method returns the requested output buffer size. The platform/underlying driver may or may not allocate the requested size for the output buffer.

The UARTEvent Class

The `UARTEvent` class encapsulates events fired by peripherals that implement the UART interface.

The `UARTEvent` class consists of three constants:

- `public static final int INPUT_DATA_AVAILABLE`

  Event ID indicating that input data is available for reading.

- `public static final int INPUT_BUFFER_OVERRUN`

  Event ID indicating that input buffer overrun.

- `public static final int OUTPUT_BUFFER_EMPTY`

  Event ID indicating that the output buffer is empty and that additional data may be written.

The `UARTEvent` class also consists of two constructors and one method:

- `public UARTEvent(UART uart, int id)`

  This constructor creates a new `UARTEvent` with the specified value. The event is then time-stamped with the current time.

- `public UARTEvent(UART uart, int id, long timeStamp, int timeStampMicros)`

  This constructor creates a new `UARTEvent` with the specified value and timestamp.

- `public int getID()`

  This method returns the event's ID.

  .

14 Watchdog Timers

The `com.oracle.deviceaccess.power` package contains interfaces and classes for using system watchdog timers (WDT).

A watchdog timer is used to reset or reboot the system in case of hang or critical failure from a unresponsive state to a normal state. A watchdog timer can be set with a time interval. Continuously refreshing the watchdog timer within the specified time interval prevents the a reset or reboot. If the watchdog timer has not been refreshed within the specified time interval, a critical failure is assumed and a system reset or reboot is carried out. Note that a *windowed watchdog timer* must be refreshed within an open time window. If the watchdog is refreshed too soon, during the closed window, or if it is refreshed too late, after the watchdog timeout has expired, the device will be rebooted.

A watchdog timer can be created using its ID.

```
WatchdogTimer wdt = (WatchdogTimer)PeripheralManager.open(8);
```

A watchdog timer can also be created using its name and interface.

```
WatchdogTimer wdt = (WatchdogTimer) PeripheralManager.open("WDT",
    WatchdogTimer.class, null);
```

Here is an example of how to create a windowed watchdog timer.

```
WindowedWatchdogTimer wdt = (WindowedWatchdogTimer)
    PeripheralManager.open("WWDT", WindowedWatchdogTimer.class, null);
```

Once the peripheral opened, the application can start using it and can especially start the timer using the `WatchdogTimer.start()` method and subsequently refresh the timer periodically using the `WatchdogTimer.refresh()` method, as shown here:

```
wdt.start(1000);
...
wdt.refresh();
```

When done, the application should call the `WatchdogTimer.close()` method to release the watchdog timer.

```
wdt.close();
```

Example 14–1 gives a demonstration of using the watchdog timer API.

Example 14–1 Using the Watchdog API

```
import com.oracle.deviceaccess.PeripheralManager;
import com.oracle.deviceaccess.PeripheralNotAvailableException;
import com.oracle.deviceaccess.PeripheralNotFoundException;
import com.oracle.deviceaccess.watchdog.WatchdogTimer;
import java.io.IOException;

public class WatchdogExample {

    public boolean checkSomeStatus() {
        // check some status....
        // if status is ok then return true to kick watch dog timer.
        return true;
    }

    public void test_loop() {
        try {
            WatchdogTimer watchdogTimer = (WatchdogTimer)
                 PeripheralManager.open(30);
            watchdogTimer.start(180000);
            // Start watch dog timer with 3 min duration.

            while (true) {
                if (checkSomeStatus() == true) {
                    // Everything goes fine, timer will be kick.
                    watchdogTimer.refresh();
                    // do something more...
                } else {
                    // Something goes wrong. Timer will not be kick.
                    // If status not recovered within 2-3 turns then
                    // system will be reboot.
                }
                try {
                    Thread.sleep(60000); // sleep for 1 min.
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
        } catch (PeripheralNotAvailableException ex) {
            ex.printStackTrace();
        } catch (PeripheralNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Watchdog timers are opened by invoking one of the `com.oracle.deviceaccess.PeripheralManager.open()` methods. The `com.oracle.deviceaccess.watchdog` permission allows access to be granted to watchdog timers devices as a whole. This permission must be requested in the JAD file under `MIDlet-Permissions` or `MIDlet-Permissions-Opt`, and the application must be digitally signed by a trusted authority to gain access to the APIs. Alternatively, the permission may be allowed for all applications in the `untrusted` domain of the security policy file (`policy.txt`).

The WatchdogTimer Interface

The `WatchdogTimer` interface provides methods for controlling a watchdog timer that can be used to force the device to reboot (or depending on the platform, the Java Virtual Machine to restart).

A `WatchdogTimer` instance may represent a virtual watchdog timer. If the device has a single physical watchdog timer, all of the virtual watchdog timers are mapped onto this one physical watchdog timer. This timer is set to expire when the virtual watchdog with the earliest timeout is scheduled to expire. The corresponding watchdog timer peripheral is therefore shared and several applications can concurrently acquire the same watchdog timer peripheral.

Each watchdog timer is identified by a numerical ID and by a name. If a watchdog timer is virtualized, a particular platform implementation may allow for several `WatchdogTimer` instances representing each a virtual instance of that same physical watchdog timer to be opened concurrently. Alternatively, it may assign each virtual watchdog timer instance a distinct peripheral ID and optionally a common name.

A `WatchdogTimer` instance can be opened by a call to one of the `PeripheralManager.open()` methods. Once the peripheral opened, the application can start using it and can especially start the timer using the `WatchdogTimer.start()` method and subsequently refresh the timer periodically using the `WatchdogTimer.refresh()` method

When done, an application should call the `WatchdogTimer.close()` method to release the watchdog timer. Any further attempt to access or control a watchdog timer which has been closed will result in a `PeripheralNotAvailableException` been thrown.

The `WatchdogTimer` interface contains six methods.

- `void start(long timeout) throws java.io.IOException, PeripheralNotAvailableException`

  This method starts the watchdog timer with the specified timeout. If the watchdog timer is not refreshed by a call to `refresh()` prior to the watchdog timing out, the device will be rebooted (or the JVM restarted).

- `void refresh() throws java.io.IOException, PeripheralNotAvailableException`

  This method refreshes the watchdog timer. This method must be called periodically to prevent the watchdog from timing out.

- `void stop() throws java.io.IOException, PeripheralNotAvailableException`

  This method stops this watchdog timer.

- `long getTimeout() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the current timeout for the watchdog timer. A value of zero (0) indicates that the watchdog timer is disabled.

- `long getMaxTimeout() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the maximum timeout that can be set for the watchdog timer.

- `boolean causedLastReboot() throws java.io.IOException, PeripheralNotAvailableException`

  This method checks if the last device reboot (or JVM restart) was caused by the watchdog timing out.

The WindowedWatchdogTimer Interface

The `WindowedWatchdogTimer` interface provides methods for controlling a watchdog timer that can be used to force the device to reboot (or depending on the platform, the

Java Virtual Machine to restart). A windowed watchdog timer must be refreshed within an open time window. If the watchdog is refreshed too soon, during the closed window, or if it is refreshed too late, after the watchdog timeout has expired, the device will be rebooted

A `WindowedWatchdogTimer` instance may represent a virtual windowed watchdog timer. If the device has a single physical windowed watchdog timer, all of the virtual watchdog timers are mapped onto this one physical watchdog timer. It gets set with a refresh window starting when the virtual windowed watchdog with the longest closed window delay is scheduled to end and ending when the virtual windowed watchdog with the earliest timeout is scheduled to expire. The corresponding watchdog timer peripheral is therefore shared and several applications can concurrently acquire the same watchdog timer peripheral.

The `WindowedWatchdogTimer` class consists of three methods:

- `void start(long timeout) throws java.io.IOException, PeripheralNotAvailableException`

  This method starts the watchdog timer with the specified timeout and with a closed window delay set to zero (0). If the watchdog timer is not refreshed by a call to `WatchdogTimer.refresh()` prior to the watchdog timing out, the device will be rebooted (or the JVM restarted).

- `void start(long closedWindowDelay, long timeout) throws java.io.IOException, PeripheralNotAvailableException`

  This method starts a windowed watchdog timer with the specified closed window time and timeout. If the `WatchdogTimer.refresh()` method is called too soon, within the closed window delay, or is called too late, not prior to the watchdog timing out, the watchdog timer will not be reset.

- `long getClosedWindowTimeout() throws java.io.IOException, PeripheralNotAvailableException`

  This method returns the current closed window delay for the watchdog timer.

A
Migrating from Device Access Version 3.2

The Device Access APIs have changed between Oracle Java ME Embedded version 3.2 and version 3.3 to add more flexibility for new devices. As such, many of the techniques used to access peripherals have changed as well. This appendix provides a brief description of the most common changes that programmers are likely to encounter: the use of the `PeripheralManager` class. For more information, please see the Device Access 3.2 and 3.3 specifications.

The PeripheralManager Class

With earlier versions of the Device Access APIs, each bus had its own singleton "manager" class that programmers would call upon to access the devices connected to the embedded board. With the 3.3 version of the Oracle Java ME Embedded platform, each of these managers has been coalesced into the `PeripheralManager` class.

Here is sample code from the 3.2 version of the Device Access API used to access the General Purpose I/O (GPIO) pins:

```
GPIOPin switchPin = null;
GPIOPin ledPin = null;

try {

    switchPin = GPIOManager.getPin("SWITCH_PIN", GPIOPin.class, null);
    ledPin = GPIOManager.getPin("LED_PIN", GPIOPin.class, null);

    if(switchPin != null && ledPin != null){
        switchPin.setInputPinListener(listener);
    }

} catch (Exception e) {
    //  Handle exceptions
}
```

Here is the equivalent version with version 3.3 of the Device Access APIs:

```
  GPIOPin switchPin = null;
  GPIOPin ledPin = null;

  try {

    switchPin=(GPIOPin)PeripheralManager.open("SWITCH_PIN", GPIOPin.class, null);
    ledPin=(GPIOPin)PeripheralManager.open("LED_PIN", GPIOPin.class, null););

    if(switchPin != null && ledPin != null){
            switchPin.setInputPinListener(listener);
        }

    } catch (Exception ex) {
            //  Handle exceptions
        }
```

Note that the newer version uses the `PeripheralManager` class to obtain access to the GPIO pins. The `PeripheralManager` class returns an object which is then cast to the appropriate type. In addition, here is a short example of how to access the MMIO bus using version 3.2 of the Device Access APIs:

```
MMIODevice rtc = null;

try {

    rtc = MMIOManager.getDevice("RTC");

    RawByte seconds = rtc.getByteRegister("Seconds");
    RawByte secAlarm = rtc.getByteRegister("SecAlarm");
    RawByte minutes = rtc.getByteRegister("Minutes");
    RawByte minAlarm = rtc.getByteRegister("MinAlarm");
    ...
} catch (Exception e) {
    //  Handle exceptions
}
```

Here is the equivalent code using version 3.3:

```
MMIODevice rtc = null;

try {

    rtc = (MMIODevice) PeripheralManager.open("RTC",
        MMIODevice.class, (String[]) null);

    RawByte seconds = rtc.getByteRegister("Seconds");
    RawByte secAlarm = rtc.getByteRegister("SecAlarm");
    RawByte minutes = rtc.getByteRegister("Minutes");
    RawByte minAlarm = rtc.getByteRegister("MinAlarm");
    ...
} catch (Exception e) {
    //  Handle exceptions
}
```

The important thing to remember is to use the correctly overloaded version of the `PeripheralManager.open()` method to obtain access to the respective device on the appropriate bus. See the examples at the beginning of each chapter for information on how to format each call for the respective API, or the equivalent Javadoc for more precise usage.

Glossary

Access Point

A network-connectivity configuration that is predefined on a device. An access point can represent different network profiles for the same bearer type, or for different bearer types that may be available on a device, such as WiFi or bluetooth.

ADC

Analog-to-Digital Converter. A hardware device that converts analog signals (time and amplitude) into a stream of binary numbers that can be processed by a digital device.

AMS

Application Management System. The system functionality that completes tasks such as installing applications, updating applications, and managing applications between foreground and background.

APDU

Application Protocol Data Unit. A communication mechanism used by SIM Cards and smart cards to communicate with card reader software or a card reader device.

API

Application Programming Interface. A set of classes used by programmers to write applications that provide standard methods and interfaces and eliminate the need for programmers to reinvent commonly used code.

ARM

Advanced RISC Machine. A family of computer processors using reduced instruction set (RISC) CPU technology, developed by ARM Holdings. ARM is a licensable instruction set architecture (ISA) and is used in the majority of embedded platforms.

AT commands

A set of commands developed to facilitate modem communications, such as dialing, hanging up, and changing the parameters of a connection. Also known as the Hayes command set, AT means *attention.*

AXF

ARM Executable Format. An ARM executable image generated by ARM tools.

BIP

Bearer Independent Protocol. Allows an application on a SIM Card to establish a data channel with a terminal, and through the terminal, to a remote server on the network.

CDMA

Code Division Multiple Access. A mobile telephone network standard used primarily in the United States and Canada as an alternative to GSM.

CLDC

Connected Limited Device Configuration. A Java ME platform configuration for devices with limited memory and network connectivity. It uses a low-footprint Java virtual machine such as the CLDC HotSpot Implementation, and several minimalist Java platform APIs for application services.

Configuration

Defines the minimum Java runtime environment (for example, the combination of a Java virtual machine and a core set of Java platform APIs) for a family of Java ME platform devices.

DAC

Digital-to-Analog Converter. A hardware device that converts a stream of binary numbers into an analog signal (time and amplitude), such as audio playback.

ETSI

European Telecommunications Standards Institute. An independent, non-profit group responsible for the standardization of information and communication technologies within Europe. Although based in Europe, it carries worldwide influence in the telecommunications industry.

Foreground switching

Changing which application is in the foreground by shifting the focus from one application to another.

GCF

Generic Connection Framework. A part of CLDC, it is a Java ME API consisting of a hierarchy of interfaces and classes to create connections (such as HTTP, datagram, or streams) and perform I/O.

GPIO

General Purpose Input/Output. Unassigned pins on an embedded platform that can be assigned or configured as needed by a developer.

GPIO Port

A group of GPIO pins (typically 8 pins) arranged in a group and treated as a single port.

GSM

Global System for Mobile Communications. A 3G mobile telephone network standard used widely in Europe, Asia, and other parts of the world.

HTTP

HyperText Transfer Protocol. The most commonly used Internet protocol, based on TCP/IP that is used to fetch documents and other hypertext objects from remote hosts.

HTTPS

Secure HyperText Transfer Protocol. A protocol for transferring encrypted hypertext data using Secure Socket Layer (SSL) technology.

ICCID

Integrated Circuit Card Identification. The unique serial number assigned to an individual SIM Card.

IMP-NG

Information Module Profile Next Generation. A profile for embedded "headless" devices, the IMP-NG specification (JSR 228) is a subset of MIDP 2.0 that leverages many of the APIs of MIDP 2.0, including the latest security and networking+, but does not include graphics and user interface APIs.

IMEI

International Mobile Equipment Identifier. A number unique to every mobile phone. It is used by a GSM or UMTS network to identify valid devices and can be used to stop a stolen or blocked phone from accessing the network. It is usually printed inside the battery compartment of the phone.

IMlet

An application written for IMP-NG. An IMlet does not differ from MIDP 2.0 MIDlet, except by the fact that an IMlet can not refer to MIDP classes that are not part of IMP-NG. An IMlet can only use the APIs defined by the IMP-NG and CLDC specifications.

IMlet Suite

A way of packaging one or more IMlets for easy distribution and use. Similar to a MIDlet suite, but for smaller applications running in an embedded environment.

IMSI

International Mobile Subscriber Identity. A unique number associated with all GSM and UMTS network mobile phone users. It is stored on the SIM Card inside a phone and is used to identify itself to the network.

I2C

Inter-Integrated Circuit. A multi-master, serial computer bus used to attach low-speed peripherals to an embedded platform

ISA

Instruction Set Architecture. The part of a computer's architecture related to programming, including data type, addressing modes, interrupt and exception handling, I/O, and memory architecture, and native commands. Reduced instruction set computing (RISC) is one kind of instruction set architecture.

JAD file

Java Application Descriptor file. A file provided in a MIDlet or IMlet suite that contains attributes used by application management software (AMS) to manage the MIDlet or IMlet life cycle, and other application-specific attributes used by the MIDlet or IMlet suite itself.

JAR file

Java Archive file. A platform-independent file format that aggregates many files into one. Multiple applications written in the Java programming language and their required components (class files, images, sounds, and other resource files) can be bundled in a JAR file and provided as part of a MIDlet or IMlet suite.

JCP

Java Community Process. The global standards body guiding the development of the Java programming language.

JDTS

Java Device Test Suite. A set of Java programming language tests developed specifically for the wireless marketplace, providing targeted, standardized testing for CLDC and MIDP on small and handheld devices.

Java ME platform

Java Platform, Micro Edition. A group of specifications and technologies that pertain to running the Java platform on small devices, such as cell phones, pagers, set-top boxes, and embedded devices. More specifically, the Java ME platform consists of a configuration (such as CLDC) and a profile (such as MIDP or IMP-NG) tailored to a specific class of device.

JSR

Java Specification Request. A proposal for developing new Java platform technology, which is reviewed, developed, and finalized into a formal specification by the JCP program.

Java Virtual Machine

A software "execution engine" that safely and compatibly executes the byte codes in Java class files on a microprocessor.

KVM

A Java virtual machine designed to run in a small, limited memory device. The CLDC configuration was initially designed to run in a KVM.

LCDUI

Liquid Crystal Display User Interface. A user interface toolkit for interacting with Liquid Crystal Display (LCD) screens in small devices. More generally, a shorthand way of referring to the MIDP user interface APIs.

MIDlet

An application written for MIDP.

MIDlet suite

A way of packaging one or more MIDlets for easy distribution and use. Each MIDlet suite contains a Java application descriptor file (`.jad`), which lists the class names and files names for each MIDlet, and a Java Archive file (`.jar`), which contains the class files and resource files for each MIDlet.

MIDP

Mobile Information Device Profile. A specification for a Java ME platform profile, running on top of a CLDC configuration that provides APIs for application life cycle, user interface, networking, and persistent storage in small devices.

MSISDN

Mobile Station Integrated Services Digital Network. A number uniquely identifying a subscription in a GSM or UMTS mobile network. It is the telephone number to the SIM Card in a mobile phone and used for voice, FAX, SMS, and data services.

MVM

Multiple Virtual Machines. A software mode that can run more than one MIDlet or IMlet at a time.

Obfuscation

A technique used to complicate code by making it harder to understand when it is decompiled. Obfuscation makes it harder to reverse-engineer applications and therefore, steal them.

Optional Package

A set of Java ME platform APIs that provides additional functionality by extending the runtime capabilities of an existing configuration and profile.

Preemption

Taking a resource, such as the foreground, from another application.

Preverification

Due to limited memory and processing power on small devices, the process of verifying Java technology classes is split into two parts. The first part is preverification which is done off-device using the preverify tool. The second part, which is verification, occurs on the device at runtime.

Profile

A set of APIs added to a configuration to support specific uses of an embedded or mobile device. Along with its underlying configuration, a profile defines a complete and self-contained application environment.

Provisioning

A mechanism for providing services, data, or both to an embedded or mobile device over a network.

Pulse Counter

A hardware or software component that counts electronic pulses, or events, on a digital input line, for example, a GPIO pin.

Push Registry

The list of inbound connections, across which entities can push data. Each item in the list contains the URL (protocol, host, and port) for the connection, the entity permitted to push data through the connection, and the application that receives the connection.

RISC

Reduced Instruction Set Computing. A CPU design based on simplified instruction sets that provide higher performance and faster execution of individual instructions. The ARM architecture is based on RISC design principles.

RL-ARM

Real-Time Library. A group of tightly coupled libraries designed to solve the real-time and communication challenges of embedded systems based on ARM processor-based microcontroller devices.

RMI

Remote Method Invocation. A feature of Java SE technology that enables Java technology objects running in one virtual machine to seamlessly invoke objects running in another virtual machine.

RMS

Record Management System. A simple record-oriented database that enables an IMlet or MIDlet to persistently store information and retrieve it later. MIDlets can also use the RMS to share data.

RTOS

Real-Time Operating System. An operating system designed to serve real-time application requests. It uses multi-tasking, an advanced scheduling algorithm, and minimal latency to prioritize and process data.

RTSP

Real Time Streaming Protocol. A network control protocol designed to control streaming media servers and media sessions.

SCWS

Smart Card Web Server. A web server embedded in a smart card (such as a SIM Card) that allows HTTP transactions with the card.

SD card

Secure Digital cards. A non-volatile memory card format for use in portable devices, such as mobile phones and digital cameras, and embedded systems. SD cards come in three different sizes, with several storage capacities and speeds.

SIM

Subscriber Identity Module. An integrated circuit embedded into a removable SIM card that securely stores the International Mobile Subscriber Identity (IMSI) and the related key used to identify and authenticate subscribers on mobile and embedded devices.

Slave Mode

Describes the relationship between a master and one or more devices in a Serial Peripheral Interface (SPI) bus arrangement. Data transmission in an SPI bus is initiated by the master device and received by one or more slave devices, which cannot initiate data transmissions on their own.

Smart Card

A card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Smart cards carry both processing power and information. A SIM Card is a special kind of smart card for use in a mobile device.

SMS

Short Message Service. A protocol allowing transmission of short text-based messages over a wireless network. SMS messaging is the most widely-used data application in the world.

SMSC

Short Message Service Center. The SMSC routes messages and regulates **SMS** traffic. When an SMS message is sent, it goes to an SMS center first, then gets forwarded to the destination. If the destination is unavailable (for example, the recipient embedded board is powered down), the message is stored in the SMSC until the recipient becomes available.

SOAP

Simple Object Access Protocol. An XML-based protocol that enables objects of any type to communicate in a distributed environment. It is most commonly used to develop web services.

SPI

Serial Peripheral Interface. A synchronous bus commonly used in embedded systems that allows full-duplex communication between a master device and one or more slave devices.

SSL

Secure Sockets Layer. A protocol for transmitting data over the Internet using encryption and authentication, including the use of digital certificates and both public and private keys.

SVM

Single Virtual Machine. A software mode that can run only one MIDlet or IMlet at a time.

Task

At the platform level, each separate application that runs within a single Java virtual machine is called a task. The API used to instantiate each task is a stripped-down version of the Isolate API defined in JSR 121.

TCP/IP

Transmission Control Protocol/Internet Protocol. A fundamental Internet protocol that provides for reliable delivery of streams of data from one host to another.

Terminal Profile

Device characteristics of a terminal (mobile or embedded device) passed to the SIM Card along with the IMEI at SIM Card initialization. The terminal profile tells the SIM Card what values are supported by the device.

UART

Universal Asynchronous Receiver/Transmitter. A piece of computer hardware that translates data between serial and parallel formats. It is used to facilitate communication between different kinds of peripheral devices, input/output streams, and embedded systems, to ensure universal communication between devices.

UICC

Universal Integrated Circuit Card. The smart card used in mobile terminals in GSM and UMTS networks. The UICC ensures the integrity and security of personal data on the card.

UMTS

Universal Mobile Telecommunications System. A third-generation (3G) mobile communications technology. It utilizes the radio spectrum in a fundamentally different way than GSM.

URI

Uniform Resource Identifier. A compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both.

USAT

Universal SIM Application Toolkit. A software development kit intended for 3G networks. It enables USIM to initiate actions that can be used for various value-added services, such as those required for banking and other privacy related applications.

USB

Universal Serial Bus. An industry standard that defines the cables, connectors, and protocols used in a bus for connection, communication, and power supply between computers and electronic devices, such as embedded platforms and mobile phones.

USIM

Universal Subscriber Identity Module. An updated version of a SIM designed for use over 3G networks. USIM is able to process small applications securely using better cryptographic authentication and stronger keys. Larger memory on USIM enables the addition of thousands of contact details including subscriber information, contact details, and other custom settings.

WAE

Wireless Application Environment. An application framework for small devices, which leverages other technologies, such as Wireless Application Protocol (WAP).

WAP

Wireless Application Protocol. A protocol for transmitting data between a server and a client (such as a cell phone or embedded device) over a wireless network. WAP in the wireless world is analogous to HTTP in the World Wide Web.

Watchdog Timer

A dedicated piece of hardware or software that "watches" an embedded system for a fault condition by continually polling for a response. If the system goes offline and no response is received, the watchdog timer initiates a reboot procedure or takes other steps to return the system to a running state.

WCDMA

Wideband Code Division Multiple Access. A detailed protocol that defines how a mobile phone communicates with the tower, how its signals are modulated, how datagrams are structured, and how system interfaces are specified.

WMA

Wireless Messaging API. A set of classes for sending and receiving Short Message Service (SMS) messages.

XML Schema

A set of rules to which an XML document must conform to be considered valid.

Index

A

AccessOutOfBoundsException, 9-10
AcquisitionEvent, 2-6
  getNumber, 2-6
  getOffset, 2-6
  getValues, 2-6
AcquisitionListener, 2-3
  inputAcquired, 2-3
ADC
  examples, 2-1
ADC (Analog-to-Digital Converter), 2-1
ADCChannel, 2-3
  getMaxValue, 2-4
  getMinSamplingInterval, 2-5
  getMinValue, 2-4
  getSamplingInterval, 2-5
  getValue, 2-4
  getValues, 2-4
  getVRefValue, 2-4
  setSamplingInterval, 2-5
  startAcquisition, 2-4
  startMonitoring, 2-5
  stopAcquisition, 2-5
  stopMonitoring, 2-5
ADCChannelConfig, 2-6
  getChannelNumber, 2-7
  getConverterNumber, 2-7
  getResolution, 2-7
  getSamplingInterval, 2-7
  getSamplingTime, 2-7
API Permissions
  com.oracle.deviceaccess.adc, 2-2
  com.oracle.deviceaccess.atcmd, 3-3
  com.oracle.deviceaccess.counter, 4-2
  com.oracle.deviceaccess.dac, 5-2
  com.oracle.deviceaccess.generic, 6-3
  com.oracle.deviceaccess.gpio, 7-3
  com.oracle.deviceaccess.gpio.GPIOPin.setDirection, 7-3
  com.oracle.deviceaccess.gpio.GPIOPort.setDirection, 7-3
  com.oracle.deviceaccess.i2c, 8-3
  com.oracle.deviceaccess.mmio, 9-3
  com.oracle.deviceaccess.power, 11-2
  com.oracle.deviceaccess.spi, 12-2
  com.oracle.deviceaccess.uart, 13-3
  com.oracle.deviceaccess.watchdog, 14-2
AT Commands, 3-1
ATDevice, 3-1, 3-3
  abortCommand, 3-5
  capabilities, 3-3
  close, 3-6
  escapeToCommandMode, 3-5
  getMaxCommandLength, 3-5
  isConnected, 3-5
  isInCommandMode, 3-5
  openDataConnection, 3-5
  sendCommand, 3-4
  setUnsolicitedResponseHandler, 3-5
ATModem, 3-1, 3-6

C

CommandResponseHandler, 3-6
  processResponse, 3-6
CountingEvent, 4-4
  getInterval, 4-4
  getType, 4-4
  getValue, 4-4
  INTERVAL_EXPIRED, 4-4
  TERMINAL_VALUE_REACHED, 4-4

D

DACChannel, 5-2
  getMaxValue, 5-3
  getMinSamplingInterval, 5-4
  getMinValue, 5-3
  getSamplingInterval, 5-4
  getVRefValue, 5-3
  setSamplingInterval, 5-4
  setValue, 5-3
  setValues, 5-3
  startGeneration, 5-4
  stopGeneration, 5-4
DACChannelConfig, 5-5
  getChannelNumber, 5-5
  getConverterNumber, 5-5
  getResolution, 5-5
  getSamplingInterval, 5-5
DataConnection, 3-6

close, 3-7
getInputStream, 3-6
getOutputStream, 3-6
DataConnectionHandler, 3-7
handleClosedDataConnection, 3-7
handleOpenedDataConnection, 3-7
Device Access
Migrating from 3.2, A-1
PeripheralManager, A-1
Digital Signatures, 1-1
keytool, 1-1
Digital-to-Analog Conversion, 5-1

G

General Purpose Input/Output (GPIO), 7-1
GenerationEvent, 5-5
getActualNumber, 5-6
getNumber, 5-6
getOffset, 5-6
getValues, 5-6
setActualNumber, 5-6
GenerationListener, 5-4
outputGenerated, 5-4
GenericBufferIODevice, 6-1, 6-4
read, 6-4
write, 6-4
GenericDevice, 6-1, 6-4
getControl, 6-5
setControl, 6-5
setEventListener, 6-5
GenericEvent, 6-6
getID, 6-6
INPUT_BUFFER_OVERRUN, 6-6
INPUT_DATA_AVAILABLE, 6-6
OUTPUT_BUFFER_EMPTY, 6-6
GenericEventListener, 6-5
eventDispatched, 6-5
GenericStreamIODevice, 6-1, 6-5
close, 6-6
getInputStream, 6-5
getOutputStream, 6-5
GPIOPin, 7-3
INPUT, 7-3
OUTPUT, 7-3
GPIOPinConfig, 7-6
getDirection, 7-7
getDriveMode, 7-8
getInitValue, 7-7
getPinNumber, 7-8
getPortNumber, 7-7
getTrigger, 7-8
GPIOPin.getDirection, 7-4
GPIOPin.getValue, 7-4
GPIOPin.setDirection, 7-4
GPIOPin.setInputListener, 7-4
GPIOPin.setValue, 7-4
GPIOPort, 7-4
getDirection, 7-5
getMaxValue, 7-5
getValue, 7-5
INPUT, 7-5
OUTPUT, 7-5
setDirection, 7-6
setInputListener, 7-6
setValue, 7-6
GPIOPortConfig, 7-8
getDirection, 7-8
getInitValue, 7-8
getPins, 7-8

I

I2CDevice, 8-3
begin, 8-4
end, 8-4
read, 8-4
write, 8-5
I2CDeviceConfig, 8-5
getAddress, 8-6
getAddressSize, 8-6
getBusNumber, 8-6
getClockFrequency, 8-6
Inter-Integrated Circuit (I2C) Bus, 8-1
InvalidOperationException, 1-7
InvalidSamplingRateException, 2-8, 5-6
InvalidStateException, 1-7
InvalidWordLengthException, 12-5

K

keytool, 1-1

M

Memory-Mapped Input/Output, 9-1
MMIODevice, 9-3
getAsRawBlock, 9-4
getBlock, 9-4
getByteOrdering, 9-4
getByteRegister, 9-4
getIntRegister, 9-4
getShortRegister, 9-4
setMMIOEventListener, 9-4
MMIODeviceConfig, 9-8
getAddress, 9-9
getByteOrdering, 9-9
getRawMemoryConfigs, 9-9
getSize, 9-9
REGISTER_TYPE_BYTE, 9-8
REGISTER_TYPE_INT, 9-8
REGISTER_TYPE_SHORT, 9-8
MMIODeviceConfig.RawBlockConfig, 9-8, 9-9
getSize, 9-9
MMIODeviceConfig.RawMemoryConfig, 9-8, 9-9
getName, 9-9
getOffset, 9-9
MMIODeviceConfig.RawRegisterConfig, 9-8, 9-9
getType, 9-9
MMIOEvent, 9-7, 9-10
getCapturedMemoryContent, 9-8, 9-10

getCapturedRegisterValue, 9-8, 9-10
getID, 9-7, 9-10
MMIOEventListener, 9-5
eventDispatched, 9-5
ModemSignalEvent, 10-2
getSignalID, 10-2
getSignalState, 10-2
signalID, 10-2
signalState, 10-2
ModemSignalListener, 10-1
signalStateChanged, 10-1
ModemSignalsControl, 10-1
CTS_SIGNAL, 10-1
DCD_SIGNAL, 10-1
DSR_SIGNAL, 10-1
DTR_SIGNAL, 10-1
getSignalState, 10-2
RI_SIGNAL, 10-1
RTS_SIGNAL, 10-1
setSignalChangeListener, 10-2
setSignalState, 10-2
ModemUART, 13-3
MonitoringEvent, 2-7
BACK_TO_RANGE, 2-7
getType, 2-7
getValue, 2-7
OUT_OF_RANGE, 2-7
MonitoringListener, 2-6
thresholdReached, 2-6

P

Peripheral, 1-2
BIG_ENDIAN, 1-2
close, 1-3
getID, 1-3
getName, 1-3
getProperties, 1-3
isOpen, 1-3
LITTLE_ENDIAN, 1-2
MIXED_ENDIAN, 1-3
UNIDENTIFIED_ID, 1-3
PeripheralConfig, 1-3
DEFAULT, 1-4
Dynamic Configuration Parameters, 1-3
Hardware Addressing Information, 1-3
Static Configuration Parameters, 1-3
PeripheralConfigInvalidException, 1-7
PeripheralEvent, 1-5
count, 1-5
getCount, 1-5
getLastTimeStamp, 1-5
getLastTimeStampMicros, 1-5
getPeripheral, 1-6
getTimeStamp, 1-5
getTimeStampMicros, 1-5
lastTimeStamp, 1-5
lastTimeStampMicros, 1-5
peripheral, 1-5
timeStamp, 1-5
timeStampMicros, 1-5
PeripheralEventListener, 1-4
PeripheralException, 1-7
PeripheralExistsException, 1-8
PeripheralManager
API Permissions, 1-1
PeripheralManagerClass, 1-6
list, 1-7
open, 1-6
register, 1-6
unregister, 1-6
PeripheralNotAvailableException, 1-8
PeripheralNotFoundException, 1-8
PeripheralTypeNotSupportedException, 1-8
Permissions
com.oracle.deviceaccess.atcmd.ATDevice.openDat
aConnection, 3-3
PinEvent, 7-9
getValue, 7-9
PinListener, 7-6
PortEvent, 7-9
getValue, 7-9
PortListener, 7-6
Power Management, 11-1
PowerManaged, 11-2
disablePowerSaving, 11-4
enablePowerSaving, 11-4
getPowerState, 11-4
LOW_POWER, 11-3
LOWEST_POWER, 11-3
POWER_OFF, 11-4
POWER_ON, 11-3
requestPowerStateChange, 11-4
UNLIMITED_DURATION, 11-3
PowerSavingsHandler, 11-4
handlePowerStateChangeRequest, 11-4
Pulse Counters, 4-1
PulseCounter, 4-2
getCount, 4-3
resetCounting, 4-3
resumeCounting, 4-3
startCounting, 4-3
stopCounting, 4-3
suspendCounting, 4-3
PulseCounterConfig, 4-4
getCounterNumber, 4-5
getSource, 4-5
getType, 4-5
TYPE_FALLING_EDGE_ONLY, 4-5
TYPE_NEGATIVE_PULSE, 4-5
TYPE_POSITIVE_PULSE, 4-5
TYPE_RISING_EDGE_ONLY, 4-5

R

RawBlock, 9-5
getByte, 9-5
getBytes, 9-6
getInt, 9-6
getInts, 9-6

getShort, 9-6
getShorts, 9-6
getSize, 9-5
setByte, 9-6
setBytes, 9-6
setInt, 9-6
setInts, 9-6
setShort, 9-6
setShorts, 9-6
RawByte, 9-6
get, 9-6
set, 9-6
RawInt, 9-7
get, 9-7
set, 9-7
RawMemory, 9-5
getName, 9-5
RawShort, 9-7
get, 9-7
set, 9-7

S

Serial Peripheral Interface Bus, 12-1
SMS
Sending a message using AT Commands, 3-1
SPIDevice, 12-2
begin, 12-3
end, 12-3
getWordLength, 12-3
read, 12-3
write, 12-3
writeAndRead, 12-3
SPIDeviceConfig, 12-4
getAddress, 12-5
getBitOrdering, 12-5
getBusNumber, 12-5
getClockFrequency, 12-5
getClockMode, 12-5
getWordLength, 12-5

T

Transactional, 1-4
begin, 1-4
end, 1-4

U

UART, 13-3
close, 13-4
getBaudRate, 13-4
getDataBits, 13-4
getInputStream, 13-4
getOutputStream, 13-5
getParity, 13-4
getStopBits, 13-4
setBaudRate, 13-4
setDataBits, 13-4
setEventListener, 13-5
setParity, 13-4
setStopBits, 13-4
UARTConfig, 13-5
getBaudRate, 13-6
getDataBits, 13-6
getFlowControlMode, 13-6
getInputBufferSize, 13-7
getOutputBufferSize, 13-7
getParity, 13-6
getStopBits, 13-6
getUARTNumber, 13-6
UARTEvent, 13-7
getID, 13-7
INPUT_BUFFER_OVERRUN, 13-7
INPUT_DATA_AVAILABLE, 13-7
OUTPUT_BUFFER_EMPTY, 13-7
UARTEventListener, 13-5
eventDispatched, 13-5
Universal Asynchronous Receiver/Transmitter (UART), 13-1
UnsolicitedResponseHandler, 3-7
processResponse, 3-7

W

Watchdog Timers, 14-1
WatchdogTimer, 14-2
causedLastReboot, 14-3
getMaxTimeout, 14-3
getTimeout, 14-3
refresh, 14-3
start, 14-3
stop, 14-3
WindowedWatchdogTimer, 14-3
getClosedWindowTimeout, 14-4
start, 14-4