



MOTOROLA

Global Telecom Solution Sector



i88s

**Multi-Communication Device
J2ME Developers' Guide**

TABLE OF CONTENTS

DOCUMENT OVERVIEW	3
DISCLAIMER	3
CONTACT INFORMATION	4
ACRONYMS AND DEFINITIONS	4
1 AGPS ON THE I88S PHONE	5
1.1 Overview	5
1.2 The Position API	6
1.3 PositionConnection Class	7
1.4 Recommendation.....	13
2 J2ME NETWORKING	14
2.1 Overview	14
2.2 Class Descriptions	15
2.3 Implementation Notes.....	20
2.4 Tips.....	21
3 FILE I/O	22
3.1 Overview	22
3.2 Class Description.....	22
3.3 Method Descriptions	22
3.4 Code Examples	23
3.5 Tips.....	27
3.6 Caveats.....	28

Document Overview

This guide describes the procedures used to develop a J2ME compliant application for the i88s multi-communication device.

Detailed information on the Java 2 Micro Edition environment is not provided.

Disclaimer

Motorola reserves the right to make changes without notice to any products or services described herein. "Typical" parameters, which may be provided in Motorola Data sheets and/or specifications can and do vary in different applications and actual performance may vary. Customer's technical experts must validate all "Typicals" for each customer application.

MOTOROLA MAKES NO WARRANTY WITH REGARD TO THE PRODUCTS OR SERVICES CONTAINED HEREIN. IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE GIVEN ONLY IF SPECIFICALLY REQUIRED BY APPLICABLE LAW. OTHERWISE, THEY ARE SPECIFICALLY EXCLUDED.

No warranty is made as to coverage, availability, or grade of service provided by the products or services, whether through a service provider or otherwise.

No warranty is made that the software will meet your requirements or will work in combination with any hardware or applications software products provided by third parties, that the operation of the software products will be uninterrupted or error free, or that all defects in the software products will be corrected.

IN NO EVENT SHALL MOTOROLA BE LIABLE, WHETHER IN CONTRACT OR TORT (INCLUDING NEGLIGENCE) FOR ANY DAMAGES RESULTING FROM USE OF A PRODUCT OR SERVICE DESCRIBED HEREIN, OR FOR ANY INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR LOSS OF REVENUE OR PROFITS, LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, OR OTHER FINANCIAL LOSS ARISING OUT OF OR IN CONNECTION WITH THE ABILITY OR INABILITY TO USE THE PRODUCTS, TO THE FULL EXTENT THESE DAMAGES MAY BE DISCLAIMED BY LAW.

Some states and other jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, or limitation on the length of an implied warranty, so the above limitations or exclusions may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights, which vary from jurisdiction to jurisdiction.

Motorola products or services are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product or service could create a situation where personal injury or death may occur.

Should the buyer purchase or use Motorola products or services for any such unintended or unauthorized application, buyer shall release, indemnify and hold Motorola and its officers,

employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the designing or manufacture of the product or service.

Contact Information

Motorola, Inc.

IDEN Subscriber Group

Phone: 1-800-453-0920

URL: <https://commerce.motorola.com/idenonline/ideveloper/index.cfm>

Acronyms and Definitions

Table 0-1: Acronyms Used In This Guide

Acronym	Terminology	Definition
AGPS	Assisted Global Positioning System	Assisted Global Positioning System
API	Application Programming Interface	A set of classes, interfaces, and methods that can be used for creating an application.
CLDC	Connected Limited Device Configuration	Sun's J2ME configuration aimed at small, wirelessly connected, memory and user interface limited devices.
GPS	Global Positioning System	A navigational system using satellite signals to fix the location of a receiver on or above the earth's surface
JAD	Java Application Descriptor	Describes a J2ME application. Typically provides application size, distributor, version, etc.
JAL	Java Application Loader	Application used to load J2ME applications to the i88s multi-communication device.
JAR	Java Archive	Archive format used by J2ME applications for compression and packaging (similar to zip files).
J2ME	Java 2 Micro Edition	The Java platform on the phone
MIDP	Mobile Information Device Profile	Sun's J2ME profile for connected, mobile devices like the i88s phone.
OEM	Original Equipment Manufacturer	Refers to the company or entity that manufactures the hardware described.
RMS	Record Management System	One of the mechanisms for persistent storage of data on the i88s phone.

1 AGPS On The i88s Phone

1.1 Overview

The i88s phone provides users and developers access to GPS position information such as latitude, longitude, altitude, speed, and etc. This feature is provided as a built-in application in the phone's standard ergonomics and as a J2ME API developers can use to create custom AGPS based applications.

This section describes some of the phone's features that affect AGPS accuracy and availability from a J2ME MIDlet.

Location Privacy

To protect users' position information privacy, the i88s phone allows users to choose their privacy setting before or while a GPS application is running on the device.

Privacy Settings

There are three privacy settings in the i88s phone:

- By Permission – Applications must request permission from the user before accessing GPS services
- Restricted – Does not allow any applications access to GPS services
- Unrestricted – Allow applications to access GPS services

By Permission

By default, the Privacy setting under the GPS Main Menu item is set to "By Permission". With this privacy setting, when the user launches any GPS MIDlet for the first time, the phone will ask the user for permission to access GPS position information. Once "By Permission" has been selected, the phone creates the Privacy Settings menu item under the Java Privacy Settings screen for this MIDlet. The MIDlet Privacy Settings has three different Location Read settings: Always, Ask, and Never.

When "Always" has been selected for Location Read, this MIDlet is allowed to access GPS data at any time even if "By Permission" is selected at the GPS Main Menu. If "Ask" has been selected, the phone will request user for permission to access GPS data; however, if "Never" is selected, then the MIDlet cannot access GPS data when it is launched.

Restricted

If the GPS Privacy setting is set to "Restricted", all GPS MIDlets will not be able to retrieve GPS data. In addition, when this setting is selected, the phone does not create the Privacy Settings menu item under the Java MIDlet System Screen.

Unrestricted

If the GPS Privacy setting is set to "Unrestricted", then all MIDlets can access GPS data. When this setting is selected, the phone does not create the Privacy Settings menu item.

Accuracy

The i88s phone is designed to receive location fixes within a preset level of geographic accuracy as determined by the network provider. Using the Position API, J2ME developers can retrieve a fix; however, the location value is not guaranteed to be within this level of accuracy. The API provides methods to determine whether a given fix is accurate or not.

Assist Data

AGPS uses cellular assisted data to retrieve a location fix. The Position API provides J2ME developers with a method to determine whether cellular assisted data is used for a given fix.

1.2 The Position API

Position API is located in the `com.motorola.iden.position` package. The API provides the location functionalities required for Java applications to access GPS position information such as the following:

- Latitude
- Longitude
- Altitude
- Time Stamp
- Travel Direction
- Speed
- Altitude Uncertainty
- Speed Uncertainty

The Position API uses the GPS Privacy setting in the Main Menu of the i88s phone when a MIDlet invokes the API. Based on the GPS Privacy setting value, the MIDlet does or does not have the access to the position information. The API will use the users' Privacy setting accordingly before providing position information. Some examples include:

- If the user's GPS Privacy setting is set to "Restricted", the Java API will return the position with all the attributes set to `UNAVAILABLE` and with the `PositionConnection`'s status code set to `POSITION_RESPONSE_RESTRICTED`.
- If the user's GPS Privacy setting is set to "Unrestricted", the Java API will be able to access GPS data and will return the position.
- If the user's GPS Privacy setting is set to "By Permission", the application will be suspended as the Java API will bring up a system screen to prompt the user for permission to grant position access for this application. If the user does not grant permission, the Java API will return the position with all the attributes set to `UNAVAILABLE` and with the `PositionConnection`'s status code set to `POSITION_RESPONSE_RESTRICTED`. After selecting one of the permission options, the user needs to resume the application.

ALMANAC Out Of Date

After permission is granted, the Java API will bring up a system screen to prompt the user if the Almanac data in the phone is out of date or invalid, and the phone is not provisioned for packet data service. This is done only once after the phone powers up. If the user gives permission to override Almanac data, the Java API will try to retrieve position data. If user does not grant the Almanac override, the Java API will return the position with its attributes set to `UNAVAILABLE` and the status of `PositionConnection` set to `POSITION_RESPONSE_NO_ALMANAC_OVERRIDE`.

1.3 PositionConnection Class

This interface supports the creation of a connection to the GPS receiver (driver). GPS position can be retrieved and status can be obtained after creating a connection. Only one connection is allowed at a time. This API must be called from a separate thread from the main application thread.

To get a `PositionConnection`, the MIDlet must use the generic `Connector` class.

Example

```
com.motorola.iden.PositionConnection sc =  
(com.motorola.iden.PositionConnection)Connector.open(String name);
```

String name should be one of the following:

- name = "mposition:delay=no"
- name = "mposition:delay=low"
- name = "mposition:delay=high"

delay=no

This option has been designed to provide the cell latitude and longitude to an application immediately after it requests them. Because all other attributes in the `AggregatePosition` class may be set to `UNAVAILABLE`, an application should use this connection only to access the cell latitude and longitude. This request does not make use of the GPS chipset. If the handset is outside of the network coverage area, the cell latitude and longitude will be set to 0.

delay=low

This option provides a response to the application in a few seconds. New assist data is retrieved only if no assist data exists or if the assist data is older than the Maximum Assist Data Age (MADA). This operation is transparent to the application. This option is designed to provide all the position attributes with assistance from the Location Enhanced Service (LES) Server. To exercise this option, the device needs to have packet data service. Currently the maximum response time for this type of request is 32 seconds. If the API times out, the position will be returned with appropriate status and error code. If a low-delay request is made outside of the network coverage area, then the API will not get the assist data from the LES. The fix will proceed without assist data, and the timeout will remain at the low-delay value of 32 seconds.

delay=high

This option provides a response to the application where delay is longer than a `delay=low` setting. It provides for an assisted or autonomous fix for the application. The phone uses existing assist data only if it is available and valid; otherwise, the location fix shall proceed autonomously. Currently, maximum response time for this type of request is 128 seconds. If the response times out, position will be returned with the appropriate status and error code.

Retrieving Position in Java

Only one request of `getPosition()` can be made or be pending at any time. If the application makes multiple requests without getting a response to the previous request, a null position value will be returned or an Exception will be thrown. The next section provides more detail on this method.

1.3.1 Method Descriptions**getPosition()**

This method allows an application to obtain position by using the same delay setting used for `Connector.open()`.

getPosition(String name)

This method allows an application to obtain a new position with different delay parameters. This method also allows an application to obtain a fix with an accurate velocity and heading direction. Note that obtaining an accurate velocity and heading direction may cause a significant delay with weak GPS signal strength. In strong GPS signal coverage this operation may take no longer than a standard fix.

The argument required for accurate velocity and heading direction is as follows:

```
String name = "delay=low;fix=extended";    // or  
  
String name = "delay=high;fix=extended";
```

`getPosition()` and `getPosition(String name)` are synchronous, blocking methods which means these methods block until a response, error, or timeout occurs. Closing `PositionConnection` from a separate thread can unblock these calls. Once the connection is closed, it needs to be opened again using `Connector.open()`.

If the `PositionConnection` is closed while a `getPosition()` call is pending or a second call has been made to `getPosition()`, then `getPosition()` and `getPosition(String name)` return a null position. Unknown errors may occur during a location fix, which may also cause null position value to be returned.

requestPending()

This method provides an application with an ability to check for pending position requests on a connection before making a new request from another thread.

getStatus()

This method returns the connection status response of the last location fix operation. This method should be called only after calling `getPosition()` or `getPosition(String name)`. The obtained position information should be used only when `getStatus()` returns `POSITION_RESPONSE_OK` status code. The following is a list of returned responses for this method:

- `POSITION_NO_RESPONSE` – This constant indicates that the device is not responding. No position information will be available, and all the attributes of the position will be set to `UNAVAILABLE`.
- `POSITION_RESPONSE_ERROR` – This constant indicates that an error occurred while retrieving the position. If possible, the cell latitude and longitude will be available, but all position's attributes will be set to `UNAVAILABLE`.
- `POSITION_RESPONSE_OK` – This constant indicates that the obtained position is a valid position. All position's attributes will be available.
- `POSITION_RESPONSE_RESTRICTED` – This constant indicates that the user has set the device to not provide the position information. No position information will be available, and the position's attributes will be set to `UNAVAILABLE`.
- `POSITION_WAITING_RESPONSE` – This constant indicates that the API is waiting for a response from the position device. `POSITION_WAITING_RESPONSE` will be returned if `getStatus()` method is called before `getPosition()` method.
- `POSITION_RESPONSE_NO_ALMANAC_OVERRIDE` - This constant indicates that the Almanac is outdated, and the user is restricted to override. No position information will be available, and all the attributes of the position will be set to `UNAVAILABLE`.

1.3.2 Using Position in Java

1.3.2.1 `getResponseCode()`

Each position response has an associated response code. The `getResponseCode()` method in the `AggregatePosition` Interface allows an application to get the response code. The Application can use this response code to validate the position. The following is a list of returned response codes:

- `POSITION_OK` – This constant indicates that the obtained position is valid and accurate.
- `ACC_NOT_ATTAIN_ASSIST_DATA_UNAV` – This constant indicates that the location fix has timed out. The fix could not be accurately attained and assist data is not unavailable.
- `ALMANAC_OUT_OF_DATE` – This constant indicates that the Almanac is out of date.
- `ACCURACY_NOT_ATTAINABLE` – This constant indicates that the location fix has timed out, and the required accuracy is not attainable.

- `BATTERY_TOO_LOW` – A constant to indicate that the battery is too weak to retrieve a fix.
- `FIX_NOT_ATTAIN_ASSIST_DATA_UNAV` – This constant indicates that the location fix has timed out because a fix is not attainable, and assist data is unavailable.
- `FIX_NOT_ATTAINABLE` – This constant indicates that the location fix has timed out because a fix is not attainable.
- `GPS_CHIPSET_MALFUNCTION` – This constant indicates that the GPS chipset is malfunctioning.
- `UNAVAILABLE` – This constant indicates that an unknown error has occurred. This is the default response code.

These response codes are used in conjunction with `PositionConnection.getStatus()` to determine the quality of the retrieved position. These values are only valid when either `POSITION_RESPONSE_ERROR` or `POSITION_RESPONSE_OK` have been returned.

The following table shows possible response codes for these two values:

PositionConnection Status Values	Response Codes
<code>POSITION_RESPONSE_OK</code>	<code>POSITION_OK</code> <code>ACCURACY_NOT_ATTAINABLE</code> <code>ACC_NOT_ATTAIN_ASSIST_DATA_UNAV</code>
<code>POSITION_RESPONSE_ERROR</code>	<code>FIX_NOT_ATTAINABLE</code> <code>FIX_NOT_ATTAIN_ASSIST_DATA_UNAV</code> <code>BATTERY_TOO_LOW</code> <code>GPS_CHIPSET_MALFUNCTION</code> <code>ALMANAC_OUT_OF_DATE, UNAVAILABLE</code>

1.3.2.2 getAssistanceUsed()

Applications can use the `getAssistanceUsed()` method to check if a fix has been retrieved using assistance.

1.3.3 Example – Using PositionConnection

```
void getViaPositionConnection() throws IOException {
    com.motorola.iden.PositionConnection c = null;
    String name = "mposition:delay=low";
    try{
        c = (PositionConnection)Connector.open(name);
        AggregatePosition oap = c.getPosition();
        // Returns the AggregatePosition which contains the position using the
        // parameter passed when connection was opened.
        // Application should only check status by calling getStatus() after
        // getPosition() or getPosition(String name) returns,
        // otherwise it returns the same status and is
        // considered an invalid call of getStatus().
        // check the status code for permission and almanac over ride
        if(c.getStatus() ==
```

```

com.motorola.iden.position.PositionConnection.POSITION_RESPONSE_RESTRICTED)
{
    // means user has restricted permission to get position
}
else if(c.getStatus() ==
com.motorola.iden.position.PositionConnection.
POSITION_RESPONSE_NO_ALMANAC_OVERRIDE)
{
    // means device has Almanac out of date and User has not granted to
override
}
else if(c.getStatus() ==
com.motorola.iden.position.PositionConnection. POSITION_NO_RESPONSE)
{
    // means no response from device
}
if (oap != null ) {
    if(c.getStatus() ==
com.motorola.iden.position.PositionConnection.POSITION_RESPONSE_OK)
    {
        // Good position
        // Check for any error from device on position
        // Application needs to check for null position
        if(oap.getResponseCode() == PositionDevice.POSITION_OK) {
            // no error in the position
            if(oap.hasLatLon()) {
                // int value of Latitude and Longitude of the position in arc
                // minutes multiplied by 100,000 to maintain accuracy or
                // UNAVAILABLE if not available
                int lat = oap.getLatitude();
                int lon = oap.getLongitude();
                // String representation of the Latitude and Longitude.
                String LATDEGREES = oap.getLatitude(Position2D.DEGREES);
                String LONGDEGREES = oap.getLongitude(Position2D.DEGREES);
            }
            if(oap.hasSpeedUncertainty()) {
                // speed and heading value are valid
                int speed = oap.getSpeed();
                if (hasTravelDirection()) {
                    // heading is available
                    int travelDirection = oap.getTravelDirection();
                }
            }
            if(oap.hasAltitudeUncertainty()) {
                int alt = oap.getAltitude(); //altitude of position in meters.
            }
        }
        // handle the errors...or request again for good position
        // or display message to the user.
        else if(oap.getResponseCode() ==
PositionDevice.ACCURACY_NOT_ATTAINABLE) {
            // the position information was provided but enough accuracy
            // may not be attainable
        }
        else if(oap.getResponseCode() ==
PositionDevice.ACC_NOT_ATTAIN_ASSIST_DATA_UNAV) {
            // the position information was provided but enough accuracy

```

```

        // assistant data unavailable
    }
} // end of position response ok
else if(c.getStatus() ==
com.motorola.iden.position.PositionConnection.POSITION_RESPONSE_ERROR)
{
    // indicate an error occurred while getting the position
    if(oap.getResponseCode() == PositionDevice.FIX_NOT_ATTAINABLE) {
        // means position information not provided (timeout)
    }
    else if(oap.getResponseCode() ==
        PositionDevice.FIX_NOT_ATTAIN_ASSIST_DATA_UNAV) {
        // means position information not provided (timeout) and
        // assistant data unavailable
    }
    else if(oap.getResponseCode() == PositionDevice.BATTERY_TOO_LOW) {
        // means battery is too low to provide fix
    }
    else if(oap.getResponseCode() ==
        PositionDevice.GPS_CHIPSET_MALFUNCTION) {
        // means GPS chipset malfunction
    }
    else if(oap.getResponseCode() == PositionDevice.ALMANAC_OUT_OF_DATE) {
        // means almanac out of date to get fix
        // This scenario occurs when user overrides almanac but device is
        // not packet data provisioned
    }
    else{
        // Unknown error occurs
    }
} // end of position response error
// position is null
} finally {
if ( c != null)
c.close();
}

```

New positions can be obtained using the following method on the same `PositionConnection` object until the close method is called.

```

AggregatePosition cell = c.getPosition("delay=no");
or
AggregatePosition oap = c.getPosition("delay=low");
or
AggregatePosition oap = c.getPosition("delay=high");

```

In addition, to obtain better accurate speed and direction

```

AggregatePosition oap = c.getPosition("delay=low;fix=extended");
or
AggregatePosition oap = c.getPosition("delay=high;fix=extended");

```

1.4 Recommendation

- GPS Driver requires about one second to calculate a new fix, so any request for a new fix during this one-second period may result in the exact same position information including the time stamp. Therefore it is recommended that an application request a new position no more than once per second.
- If an application needs continuous position it is recommended to use “delay=low” once and “delay=high” there after even if the first fix does not succeed. The reason for this is because of network failures. When there is a network failure, there is a 12 to 24 second communication timeout from the LES.
- Use “delay=no” if the application only needs the cell latitude and longitude.
- Applications must handle all response codes returned by the `AggregatePosition.getResponseCode()` method and the `PositionConnection.getStatus()` method. `getStatus()` provides the connection’s status after the fix and user interaction status with regards to permission. `getResponseCode()` provides information about the position itself.
- Applications must always check the speed uncertainty value before using speed and heading. Although it is counter-intuitive, the presence of speed uncertainty denotes that the speed and heading value are accurate. Therefore, if a call to `hasSpeedUncertainty()` returns true, the speed and heading returned by the API are valid.
- If an application calls `getPosition(String name)` method with the “fix=extended” tag, this method will return accurate velocity and heading direction; however, there is a time penalty since it takes longer to calculate the accurate velocity and heading direction when the method is called.
- The method, `PositionConnetion.getStatus()` provides the status of the connection when the method `PositionConnection.getPosition()` was called. Whereas, `AggregatePosition.getResponseCode()` returns the detailed response code.
- Getting a position for the first time after the phone powers on is referred as a “cold start”. A position retrieved within ten seconds of the previous fix is referred to as a “hot start”. A position retrieved after ten seconds of the previous fix is a “warm start”. After 1 hour since the last fix will set the device back to “cold start”. Therefore, “hot start” is the quickest way of retrieving a fix.

2 J2ME Networking

2.1 Overview

The J2ME platform on the i88s phone provides a variety of networking functionalities beyond those specified in MIDP. The additional networking protocols are added through the Generic Connection Interface in order to simplify the interface to the application as well as to reduce the need for additional classes. Most of the additional network connections are invoked using a runtime parameter similar to HTTP, reducing the learning curve for developers as well as the reducing potential application porting efforts. The following is a list of networking features for the i88s phone:

- HTTP
- HTTPS
- TCP Sockets
- SSL Secure Sockets
- Server Sockets
- UDP Sockets
- Serial Port Access

The standard networking protocol specified in MIDP 1.0 is HTTP. Although HTTP is useful and flexible for most data exchanges, many of the applications fall outside the standard request/response models of most browsers. Applications such as games and stock tickers require networking protocols with different characteristics. In order to accommodate these types of applications with reasonable efficiency, additional protocol stacks including UDP, TCP Sockets, SSL, and HTTPS have been added. These added networking functionalities not only provide the application developer with more communication options, it alleviates the need to perform inefficient workarounds for a strict HTTP environment. Other applications may also choose to take advantage of the bottom connector on the devices. The bottom connector is a serial port enabling communication with a variety of other devices. The i88s phone also provides serial port access through the Generic Connection Framework in order to provide applications a means to communicate to external devices such as GPS, OBD, PCs, etc.

2.2 Class Descriptions

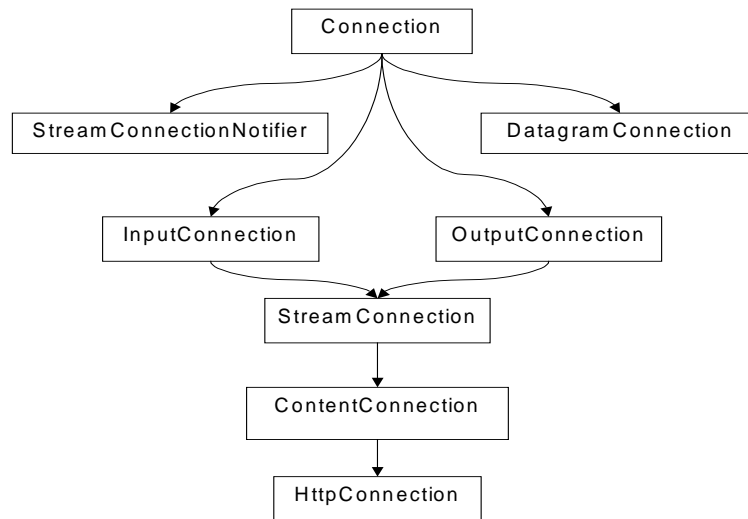


Figure 2-1: The Connection Framework

Since all the additional communication protocols have been added to the Generic Connection Framework, the access methods and parameters are very similar. The main calls are to the Connector class, which provides three static methods that accept different compile time parameters. The commonality between the three static methods is the first parameter in their signatures. This particular runtime parameter accepts Strings formatted in the standard Uniform Resource Locator format. The following is the list of method signatures:

```

Connector.open(String URL) – default READ_WRITE, no timeout
Connector.open(String URL, int mode) - defaults to no timeout
Connector.open(String URL, int mode, Boolean timeout)
  
```

`String URL` – parameter string describing the target conforms to the URL format as described in RFC 2396 for all networking protocols except for Serial Port.

`int mode` – READ/WRITE/READ_WRITE

`boolean timeout` - An optional third parameter, protocol may throw an IOException when it detects a timeout condition.

The timeout period for the TCP implementation on the i88s phone is 40 seconds on an open and about 120 seconds on read/write operations if the timeout flag is set to true. If the timeout flag is set to false, the timeout occurs in 180 seconds. The lingering time for closing sockets is 10 seconds, so if a new socket is requested within this time frame and the maximum number of sockets opened has been reached, then an IOException is thrown.

Applications requesting a network resource for any protocol must use one of the three methods above. The URL is the distinguishing argument that determines the difference between HTTP, UDP, Serial, etc. The following chart details the prefixes that should be used for the supported protocols.

Table 2-1: Supported Protocols on the i88s Phone

Protocol	URL Format
HTTP	http://
HTTPS	https://
TCP Sockets	socket://
SSL Secure Sockets	ssocket://
Server Sockets	serversocket://
UDP Sockets	datagram://
Serial Port	comm:0;

2.2.1 HTTP

The HTTP implementation follows the MIDP 1.0 standard. The `Connector.open()` methods return a `HttpConnection` object that is then used to open streams for reading and writing. The following is a code example:

```
HttpConnection hc =
(HttpConnection)Connector.open("http://www.motorola.com/");
```

In this particular example, the standard port 80 is used, but this parameter can be specified as shown in the following example:

```
HttpConnection hc =
(HttpConnection)Connector.open("http://www.motorola.com:8080");
```

The other static `Connector` methods work in the same manner, but they provide the application additional control in dealing with the properties of the connection. By default, HTTP 1.1 persistency is used to increase efficiency while requesting multiple pieces of data from the same server. In order to disable persistency, set the "Connection" property of the HTTP header to "close".

2.2.2 HTTPS

The HTTPS implementation follows the MIDP 1.0 standard, save for the security aspects. The `Connector.open()` methods return a `HttpConnection` object that is then used to open streams for reading and writing. The following is a code example:

```
HttpConnection hc =
(HttpConnection)Connector.open("https://www.motorola.com/");
```

In this particular example, the standard port 443 is used, but this parameter can be specified as shown in the following example:

```
HttpConnection hc =  
(HttpConnection)Connector.open("http://www.motorola.com:8888");
```

The other static Connector methods work in the same manner, but they provide the application additional control in dealing with the properties of the connection.

Note – Only Verisign Certificates are supported in the i88s phone. The following is a list of supported features:

- SSL 3.0
- TLS 1.0
- Server Authentication

2.2.3 TCP Sockets

The low-level socket used to implement the higher-level HTTP protocol is exposed to applications via the Generic Connection Framework. The usage is similar to the examples above, however, a StreamConnection is returned by the Connection.open() method, as shown in the following example:

```
StreamConnection sc =  
(StreamConnection)Connector.open("socket://www.motorola.com:8000");
```

Although similar to HTTP, notice the required port number at the end of the remote address. In the previous protocols, those ports are well known and registered so they are not required, but in the case of low level sockets, this value is not defined. The port number is a required parameter for this protocol stack.

2.2.4 SSL Secure Sockets

The low-level socket used to implement the higher-level HTTPS protocol is also exposed to applications via the Generic Connection Framework. The usage is similar to the examples above.

```
StreamConnection sc =  
(StreamConnection)Connector.open("ssocket://www.motorola.com:8000");
```

As with non-secure sockets, the port number is a required parameter for this protocol stack.

2.2.5 ServerSockets

In addition to acting as a data requestor, some applications may act as data providers or servers. In order to accomplish this without workarounds or polling, a server type socket is required. The i88s phone provides this functionality within the GenericConnectionFramework. Opening a ServerSocket via the Connector object

returns a `StreamConnectionNotifier`. Unlike the other networking protocols, the `StreamConnectionNotifier` does not contain any accessor methods to retrieve data, but rather only one method to accept and open a Socket Connection. This method blocks until a Socket connection is available at which time, it returns a `StreamConnection` object. The following example illustrates this:

```
StreamConnectionNotifier scn =
(StreamConnectionNotifier)Connector.open("serversocket://:8000");

StreamConnection sc = (StreamConnection)scn.acceptAndOpen();
```

The URL parameter passed in is similar to that used for TCP Sockets, with the exception of the target address. In this particular instance, the target address is left blank, assuming the `serversocket` is to be opened on the local device. The port number however, is still required. The `acceptAndOpen()` method of the `StreamConnectionNotifier` object is a blocking call, so applications that utilize the particular protocol, should take this into consideration.

Note - To close the `serversocket`, close the associated `StreamConnectionNotifier`.

2.2.6 UDP Sockets

If networking efficiency is of greater importance than reliability, datagrams (UDP) sockets are also available to the application in much the same manner as other networking protocols. The Connector object in this case returns a `DatagramConnection` object, as is shown in the following example:

```
DatagramConnection dc =
(DatagramConnection)Connector.open("datagram://170.169.168.167:8000");
```

Much like low-level sockets, accessing UDP requires both a target address and a port number. The i88s phone supports a maximum outgoing and incoming payload of 1472 bytes and 2944 bytes, respectively.

2.2.7 Serial Port Access

Applications utilizing the bottom connector (serial port) to communicate with a variety of devices are given exclusive access to the port until either the application voluntarily releases the port or is terminated. Much like any other networking connection, opening a serial port is not guaranteed and an exception can be thrown. If another application native or Java is using the port, or a cable is not attached to the device, an `IOException` may be thrown. In the normal usage scenario, the Connector object in this instance returns a `StreamConnection`, as is shown in the following example:

```
StreamConnection sc =
(StreamConnection)Connector.open("comm:0;baudrate=19200;parity=n;datab
its=8;stopbits=1;flowcontrol=n/n");
```

Although serial port access is integrated into the Generic Connection Framework, the URL parameters passed in deviates from the other networking protocols. The optional parameters, such as baud rate, parity, etc are appended to the base parameter of "comm.:0". Optional parameters are listed below along with the default values when not explicitly specified:

Table 2-2: Connection Optional Parameters

Parameter	Syntax	Options	Default
baudrate	baudrate = x	[300,1200,2400,4800,9600,19200,38400,57600,115200]	19200
databits	databits = x	[8,7]	8
stopbits	stopbits = x	1	1
parity with mapping	parity = x	[n,o,e,s,m] n=none, o = odd, e=even, s=space, m=mark	n
Flow control	flowcontrol = outflow/inflow	[n, s, h] / [n, s, h] n=none, s=software,h=hardware	N/n

Note - The following combinations of properties are not supported.

7 databits with none parity
 8 databits with mark parity
 8 databits with space parity
 8 databits with odd parity
 8 databits with even parity.

IOException will be thrown while trying to use any of the unsupported combinations in Connector.open().

All properties must be semicolon separated. If all properties are not passed, the remaining properties will be taken as default values. The order of properties in the argument does not matter.

```
name = "comm:0;baudrate=38400;"
```

Here, the flow control, parity, data bits and stop bits will use the default values.

For mode and timeout refer to the CLDC API specification for the Connector class.

Note – The timeout period for Comm Port is 10000ms ~ 10 sec.

Communicating on a Port

The open method of the Connector class returns a StreamConnection object for the serial port. StreamConnection has methods for obtaining input and output streams from a port. The base interface, Connection, has a method to close the port. (Refer to the class hierarchy from StreamConnection from J2ME CLDC API specification).

There are five basic steps to communicating with a port:

1. Open the port using the `open()` method of Connector. If the port is available, this returns a StreamConnection object for Comm port. Otherwise, an IOException is thrown.
2. Get the output stream using the `openOutputStream()` method of OutputConnection.

3. Get the input stream using the `openInputStream()` method of `InputConnection`.
4. Read and write data onto those streams.
5. Close the port using the `close()` method of both the `Connection` and open Streams.

Once the connection has been established, simply use the normal methods of any input or output stream to read and write data. The `openInputStream` and `openOutputStream` methods of `StreamConnection` are similar to the methods of the socket `StreamConnection`.

Example using `StreamConnection`

`Connector.open` is used to open the serial port and a `StreamConnection` is returned. From the `StreamConnection` the `InputStream` and `OutputStream` are opened. It is used to read and write every character until the connection is closed(-1). If an exception is thrown the connection and stream are closed.

```
StreamConnection sc = null;
InputStream is = null;
OutputStream os = null;

/*
 * Create the parameter String with options specified
 */
String parameter =
"comm:0;baudrate=19200;parity=n;databits=8;stopbits=1;flowcontrol=n/n";

try{
    sc = (StreamConnection)Connector.open(parameter,
        Connector.READ_WRITE, false);
    os = sc.openOutputStream();
    is = sc.openInputStream();
    int ch;
    while ((ch = is.read() ) != -1) {
        os.write(ch);
    }
} finally {
    if (sc != null)
        sc.close();
    if(is != null)
        is.close();
    if(os != null)
        os.close();
}
```

2.3 Implementation Notes

As stated in the previous sections, the i88s phone supports a vast array of networking options. The networking options however are limited by both memory and bandwidth, which place hard restrictions on the applications. These limitations manifest themselves mainly in

the number of simultaneous connections that can be opened. The following chart characterizes the boundary conditions for each networking stack.

Table 2-3: Concurrent Connections For The i88s Phone

Protocol	Maximum Concurrent	Notes
HTTP/ HTTPS	2	If the maximum number of HTTP Connections are concurrently opened by the application and a third HTTP Connection is requested, connections that have not been active within the past 60 seconds are reclaimed and reused if found, otherwise an exception is thrown to the calling application.
Socket/ SecureSocket	2	If the maximum number of sockets is concurrently opened by the application, and a third socket is requested, an exception is thrown to the calling application.
ServerSocket	2	Only two ServerSocket is available. Any attempts to open 3 concurrent ServerSockets results in a thrown exception.
UDP	11	If a twelfth socket is requested, an exception is thrown to the calling application.
Serial Port	1	Only one serial port is available. Any attempts to open 2 concurrent serial port connections results in a thrown exception.

2.4 Tips

- A factor to take into consideration while developing networked applications is the blocking nature of many `javax.microedition.io` and `java.io` Object methods. It is advisable to spawn another thread specifically dedicated to retrieving data in order to keep the user interface interactive. If a single thread is used to retrieve data on a blocking call, the user interface becomes inactive with the end-user perceiving the application as “dead”.
- Reading from an `InputStream` using an array is faster than reading byte by byte, when the length of the data is known. For example, if the content length is provided in the header of the `HttpConnection`, then an array of the specified size can be used to read the data.
- The `InputStream` and `OutputStream` as well as the `Connection` Object need to be completely closed.
- An application in the suspended state can still continue to actively use the networking facilities of the i88s phone.
- The platform does not support simultaneous voice and data transmissions. The i88s phone will not be able to receive a phone call when exchanging packet data; however, it can receive a phone call on an idle open connection. An idle open connection means that a socket is opened but not transferring data. There is at least 10 seconds of linger time between switching from the data to voice, but switching from the voice to data is almost instantaneous. Therefore, a phone call may be received only after 10 or more seconds since packet data transmission.

3 File I/O

3.1 Overview

The objective of the File I/O API is to provide a generic platform for the Java developer to use to open, read, and/or write, append and delete a file sequentially. The goal is to provide UNIX like file access APIs, as a simple alternative to Record Management System (RMS).

Almost all MIDlets need to be able to save information to be retained between invocations; this is called "persistent storage."

Examples include:

- Saving data such as notes, phone numbers, tasks, etc...
- Keeping a history of recent URLs

3.2 Class Description

```
import javax.microedition.io.Connector
```

3.3 Method Descriptions

3.3.1 Opening a file

Opening a file gives your application exclusive access to that particular file until it is explicitly closed or the program is ended.

To open a file, a J2ME application can use all the APIs defined by the Connector class.

```
StreamConnection sc =(StreamConnection)Connector.open(String name)  
throws java.io.IOException
```

Argument `name` is the name of the file to open, and can include the mode in which to open the file

e.g. `name = "file://temp.txt"` specifies that file is to be opened in the default mode, `READ_WRITE` *or* `name = "file://temp.txt;APPEND"` which specifies that file is to be opened in an `APPEND` mode.

```
StreamConnection sc =(StreamConnection)Connector.open(String name,  
int mode);
```

`name = "file://temp.txt"` and `mode` may have three values: `Connector.READ`, `Connector.WRITE`, and `Connector.READ_WRITE`.

```
StreamConnection sc =(StreamConnection)Connector.open(String name, int mode,
boolean timeout);
```

name = "file://temp.txt" and mode may have three values: Connector.READ, Connector.WRITE, and Connector.READ_WRITE. The third parameter, timeout, has no effect on the interface.

The open method of the Connector class returns a StreamConnection object for file access, which implements InputConnection and OutputConnection. The InputConnection and OutputConnection have methods for getting input and output streams. Base interface Connection has a method to close the file. (Refer to the class hierarchy of StreamConnection from the CLDC 1.0 specification).

3.3.2 Deleting a file

An application can delete a file in the following manner:

```
StreamConnection sc=(StreamConnection)Connector.open(String name);
```

Where name = "file://temp.txt;DELETE"; is entered, "temp.txt" will be deleted. In addition, if the file cannot be deleted, an I/O Exception will be thrown.

NOTE: All the InputStreams, OutputStreams and StreamConnections associated with a file should be closed before deleting the file.

3.3.3 Reading and Writing

Since the File IO class utilizes the Generic Connection Framework, reading and writing to a file is accomplished by standard InputStream and OutputStream APIs.

3.4 Code Examples

Example # 1 (file snippet)

The following example shows how to open a file, write bytes to the file and read the same number of bytes.

```
StreamConnection sc = null;
InputStream is = null;
OutputStream os = null;
String name = "file://temp.txt";
try {
    // open a file, default mode: READ_WRITE
    sc = (StreamConnection)Connector.open(name);
    // get OutputStream
    os = sc.openOutputStream();
    // get InputStream
    is = sc.openInputStream();
    String b = "Hello World";
    // write the bytes
    os.write(b.getBytes());
    int dataAvailable = is.available();
    byte [] b1 = new byte[dataAvailable];
    // read the bytes
```

```

        is.read(b1);
    } finally {
        if (sc != null)
            sc.close();
        if (is != null)
            is.close();
        if (os != null)
            os.close();
    }
}

```

Example # 2 (Complete MIDlet code)

The following example is a simple MIDlet that will provide the overall operation of the file I/O interface and how most of the API's can be used. The MIDlet also shows a simple alternative to RMS to store data as a persistent storage.

```

import javax.microedition.io.*;
import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class Example2 extends MIDlet implements CommandListener{
/**
 * List of available tests
 */
StreamConnection sc;
String[] testList = {"file to w/r","setData","write/append/read","delete"};
TextBox tf1;
TextBox tf2;
/**
 * Reference to Display object associated with this Display
 */
Display myDisplay;
/* default file name */
String fileURL = "temp.txt";
/*default amount of data*/
int dataNum = 0;
/*default string to write in file*/
String stringNum ="Hello World";
/**
 * The output screen
 */
Form myOutput;
/**
 * The list of tests
 */
List myList;
/**
 * Ok command to indicate a test was selected
 */
Command okCommand;
/**
 * Create NetTests
 */
Example2( ) {
}
}

```

```

/**
 * Start running
 */
protected void startApp() {
    myDisplay = Display.getDisplay(this);
    myOutput = new Form("Results");
    myList = new List("Select test:", List.IMPLICIT, testList,
        null);
    okCommand = new Command("OK", Command.OK, 1);
    myOutput.addCommand(okCommand);
    myList.addCommand(okCommand);
    myOutput.setCommandListener(this);
    myList.setCommandListener(this);
    tf1 = new TextBox("file to w/r", fileURL, 28, TextField.ANY);
    tf1.addCommand(okCommand);
    tf1.setCommandListener(this);
    tf2 = new TextBox("Set Data to Send", stringNum, 28, TextField.ANY);
    tf2.addCommand(okCommand);
    tf2.setCommandListener(this);
    myDisplay.setCurrent(myList);
}

/**
 * Stop running
 */
protected void pauseApp() {
}

/**
 * Destroy App
 */
protected void destroyApp(boolean unconditional) {
}

/**
 * Handle ok command
 */
public void commandAction(Command c, Displayable s) {
    if (((s == tf1) || (s == tf2)) && (c == okCommand)) {
        if(s==tf1) fileURL = tf1.getString();
        if(s==tf2) {
            /* data in the string form */
            stringNum = tf2.getString();
            /* convert the string into the integer form */
            dataNum = stringNum.length();
        }
    }
    if (s == myList) {
        switch (((List)s).getSelectedIndex()) {
            case 0:
                myDisplay.setCurrent(myOutput);
                setFileName();
                break;
            case 1:
                myDisplay.setCurrent(myOutput);
                setData();
                break;
            case 2:
                myDisplay.setCurrent(myOutput);
        }
    }
}

```

```

        readWrite();
            break;
        case 3:
            myDisplay.setCurrent(myOutput);
deleteFile();
            break;
        }
    } else {
        myDisplay.setCurrent(myList);
    }
}
private void setFileName() {
    myDisplay.setCurrent(tf1);
}
private void setData() {
    myDisplay.setCurrent(tf2);
}
private void readWrite() {
    int length = dataNum;
    byte[] message = new byte[length];
    message = stringNum.getBytes();
    OutputStream os = null;
    InputStream is = null;
    try {
        //open a file in the mode APPEND
        sc =
(StreamConnection)Connector.open("file://"+"fileURL"+" "+"APPEND");
        //get OutputStream
        os = sc.openOutputStream();
        //get InputStream
        is = sc.openInputStream();
        //write the bytes to the file
        os.write(message);
        myOutput.append("write/append done");
        //create an array to store available data from the file
        byte [ ] b1 = new byte[is.available()];
        //read the bytes
        is.read(b1);
        String readString = new String(b1);
        //printout the data in the phone screen
        myOutput.append(readString);
        myOutput.append("read finished");
        //close all the opened streams
        if (is != null)
            is.close();
        if (os != null)
            os.close();
        if (sc != null)
            sc.close();
    } catch (Exception e) {
        System.out.println("Exception: " + e.getMessage());
        try {
            if (is != null)
                is.close();
            if (os != null)
                os.close();
            if (sc != null)

```


- If a MIDlet suite is updated to another version, then the file(s) associated with the current version of MIDlet suite, can be maintained for the new version to use. The user is prompted to keep the old data or delete it.
- If the MIDlet suite is deleted, all files associated with it are deleted.
- It is the MIDlets responsibility to coordinate the use of multiple threads to access a file since unintended consequences may result.
- Whenever you close a file, the close command will not return until all the pending writes have been completed; thus closing a file guarantees that all of the data was written. It is then safe to power off the device. One consequence is that the close command may take a while to return. Therefore, if you open and close the file every time a write is required, performance will be greatly affected.

3.6 Caveats

- This File Access System is a sequential system. This means once you write particular chunk of data to the file you can't go back and manipulate it.
- The maximum number of files that the i88s phone supports is 1024. Once the phone contains 1024 files, it will not be able to create more. MIDI ringers, voice notes, wallpapers, PNG images included with a MIDlet etc. are all files. If a MIDlet is to have many images, such as sprites used in animations, it may be advantageous to have them all in one image file and use clipping to display only what you need.
- A file can be of any size as long as file space is available. A zero byte file is also allowed.
- It is recommended that only 15 files remain open at one time. Exceeding the maximum number of opened files can result in unintended behavior.
- The `InputStream` method `markSupported()` method returns true only if the file open mode is `READ` or `APPEND`. This means that in any other file open mode, the `mark()` and `reset()` methods do not work.
- In the `InputStream` method `mark()`, the `readlimit` argument tells the input stream to allow that many bytes to be read before the mark position gets invalidated. Since this operation is on a file, "remembering" the entire contents of stream/file does not incur any type of cost, so the `readlimit` parameter is ignored preventing mark position invalidation.

MOTOROLA, the Stylized M Logo and all other trademarks indicated as such herein are trademarks of Motorola, Inc. ® Reg. U.S. Pat. & Tm. Off. © 2002 Motorola, Inc. All rights reserved.

Microsoft and, Microsoft WEB Explorer, are registered trademarks of Microsoft Corporation.

Java and all other Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All other product or service names mentioned in this manual are the property of their respective trademark owners.

Software Copyright Notice

The Motorola products described in this manual may include copyrighted Motorola and third party software stored in semiconductor memories or other media. Laws in the United States and other countries preserve for Motorola and third party software providers certain exclusive rights for copyrighted software, such as the exclusive rights to distribute or reproduce the copyrighted software. Accordingly, any copyrighted software contained in the Motorola products may not be modified, reverse-engineered, distributed, or reproduced in any manner to the extent allowed by law. Furthermore, the purchase of the Motorola products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents, or patent applications of Motorola or any third party software provider, except for the normal, non-exclusive, royalty-free license to use that arises by operation of law in the sale of a product.