



nutiteq

Nutiteq Maps SDK developer Guide

Version 1.1.1 (24.08.2011)

© 2008-2011 Nutiteq LLC

Nutiteq LLC

www.nutiteq.com

Skype: nutiteq

1 Contents

Nutiteq Maps SDK developer Guide.....	1
2 Introduction.....	4
2.1 Document history.....	4
3 SDK contents.....	5
4 Installation.....	6
5 Sample applications.....	6
5.1 Build and Installation.....	7
5.2 Android sample	7
5.2.1 Android Mapper	7
5.3 BlackBerry sample	7
5.3.1 BlackBerry Mapper - UIApplication demo.....	7
5.4 Java ME samples.....	7
5.4.1 Nutiteq Mapper	7
5.4.2 Nutiteq Forms mapping.....	8
5.4.3 Nutiteq KML Service	8
5.4.4 Nutiteq Directions	8
5.4.5 Nutiteq Full Screen	8
5.4.6 Nutiteq Custom Elements	8
5.4.7 Nutiteq Jar maps demo	8
5.4.8 GPS Location demo.....	9
5.4.9 Hybrid maps demo	9
5.4.10 Stored maps demo	9
6 BasicMapComponent API.....	9
7 SDK API methods	13
7.1 Methods of BasicMapComponent.....	13
7.2 Methods for Geocoding and Directions searches	13
7.3 Methods for mobile positioning.....	14
7.3.1 Internal GPS API (Location API/SR-179)	14
7.3.2 Bluetooth GPS positioning.....	14

7.3.3	Cell-ID Positioning.....	14
7.4	Common public classes	15
7.5	Callback events to application.....	16
7.6	Using public interfaces for customization.....	17
7.7	Selecting mapping service	18
7.8	Defining own on-line map service.....	18
7.8.1	Projections.....	19
7.8.2	Zoom level system.....	19
7.8.3	Pixel coordinate space.....	20
7.8.4	Building URL for tiled map service.....	20
7.8.5	Building URL for non-tiled map services.....	21
7.8.6	Streamed maps.....	21
7.9	Other features	22
7.9.1	Pointer controls and cursors	22
7.9.2	Zoom indicator	22
7.9.3	Download counter.....	22
8	Platform-specific development	23
8.1	Overview.....	23
8.2	Java ME (J2ME).....	25
8.2.1	Using Forms and high-level Custom Map Item API.....	25
8.2.2	Minimizing package.....	27
8.3	BlackBerry API	27
8.3.1	BlackBerry Midlet	27
8.3.2	BlackBerry connection parameters	27
8.3.3	BlackBerry UIApplication API.....	28
8.3.4	BlackBerry UIApplication high level API with FieldManager	31
8.4	Google Android platform.....	32
8.4.1	Getting started sample.....	32
8.5	Using J2ME Polish Pro for application development	34
9	Caching	35

9.1	Cache levels	35
9.2	RMS cache	36
9.3	File system cache.....	36

2 Introduction

This document describes MGMaps mobile mapping library (Nutiteq mapping SDK) capabilities and key services. The document is for developers who start using MGMaps Lib to build their own mobile mapping application, or to enhance existing mobile applications with mapping capabilities.

It is assumed that the developer is familiar with web services, XML technologies and has at least moderate experience in Java and Mobile Java development on their target platform. There are many on-line tutorials and books available covering these topics.

Library includes the following key features:

- a) Slippy map
 - a. Freedom of map data source
 - b. Predefined support of number of major map tile vendor APIs
 - c. WMS support with tiled maps
 - d. Support for different tile sizes: from 64 to 256 pixels
 - e. Caching in memory, RMS, file system
 - f. Streaming of maps for improved speed
 - g. Zoom in/out animations
 - h. Using offline maps (from flash card, phone memory or installation package)
- b) DirectionsService – calculate the quickest way between 2 points
 - a. Support for several vendors
- c) GeocodingService – find placenames and Points of Interest (POI)
- d) Location API – find user’s location using different positioning methods
 - a. Internal and external Bluetooth GPS
 - b. Cell-ID etc

2.1 Document history

Who	When	Rev	What
JaakL	03.07.2008	0.1	First version
JaakL	12.08.2008	0.2	Updated for lib version 0.2.2. Major additions: line drawing, KML services, Cloud Made maps

JaakL	22.08.2008	0.3	Added new methods of 0.3 release. Added chapter "9.5 Public interfaces"
JaakL	11.09.2008	0.4	Updated for 0.4.0 release. Added chapter "9.3 Methods for Geocoding and Directions searches"
JaakL	22.09.2008	0.5	Doc updated for 0.5.0 release. Added chapter "Using library in BlackBerry API"
JaakL	20.10.2008	0.6	Updated for 0.6.0.
JaakL	20.11.2008	0.7	0.7.0 updates. Added yournavigation.org to directions example, Cell-ID positioning feature.
JaakL	26.01.2009	0.8	0.8.0 updates. Bluetooth GPS support, Cell-ID positioning extended for OpenCellId.org, Routing with CloudMade API, added data counter, improved logging and caching
JaakL	08.07.2009	1.0.0	1.0.0 lib updates. Added more detailed BlackBerry and Android chapters
JaakL	12.11.2009	1.0.2	1.0.2 lib updates.
JaakL	24.08.2011	1.1.1	1.1.1 lib updates

3 SDK contents

MGMaps Lib SDK includes:

- MGMaps Lib (binary library)
 - Library builds for Java ME, BlackBerry (UIApplication and Midlet) and Android
 - Library source package is available on-line separately in www.nutiteq.com developer section (requires registration)
- Sample applications with sources
- Documentation
 - Javadoc for library API (available also on-line in www.nutiteq.com Developer section)
 - This developer guide
 - "Hello Map" Tutorial

4 Installation

Make sure that following software is installed before installation:

- Eclipse IDE (<http://www.eclipse.org/downloads/>). Classic or Java developer edition is suggested. Also other Java IDE can be used (Netbeans etc), but samples here are based on Eclipse
- Java 1.6 (<http://www.java.com/en/download/index.jsp>)
- For Java ME:
 - Sun Wireless Toolkit 2.5.2 (<http://java.sun.com/products/sjwtoolkit/download.html>)
 - Suggested for Java ME: *Mobile Tools for Java* plugin. In your work environment other building tools which support J2ME building (Ant, J2ME Polish, WTK built-in tools, other IDE-s etc) can be used.
 - Other helpful tools to build your J2ME application, depending on your preferences: e.g. J2ME Polish, ProGuard etc
- For BlackBerry:
 - Corresponding JDE packages or BlackBerry Eclipse SDK. Eclipse SDK is suggested and sample is done using it.
- For Android
 - Android SDK package from Google, Eclipse plug-in suggested

General steps for installation:

1. Unpack library package
2. Create a project for your application into the workspace (if not done already)
3. Copy Library JAR (maps_lib-XXX.jar) to you the project's lib directory
4. Add library JAR to the class path. Make sure that the library is also included to final build.
5. Optional: define javadoc locations to the lib documentation.
6. Develop your application. You can use sample applications as a guideline or template. These are located in *samples/mapper/src* directory of library package
7. Compile and run the client.

5 Sample applications

Lib package includes some sample mapping applications with sources. These should give reference and hints about how to use the library; these were also used as basic test suite for on-device testing.

5.1 Build and Installation

All the samples are Eclipse projects. Extract them and import to your workspace. The Eclipse workspace should have appropriate plug-ins installed:

- Android SDK from Google for Android
- BlackBerry Eclipse SDK with needed BB Java SDK-s from RIM for Blackberry
- J2ME SDK from Sun and Mobile Tools for Java plugin for J2ME

Sample apps may need minimal Build Path reconfiguration – make sure it includes suitable maps_lib jar file version and for BB and J2ME make sure that the maps library JAR is also exported (Android does it automatically).

5.2 Android sample

5.2.1 Android Mapper

- Android sample project is kept up-to-date in <https://bitbucket.org/nutiteq/android-map-samples>
- Create MapView
- Change map sources: different online and offline maps, including file system and MBTiles based sources and bundled maps.
- Add overlays: raster tile overlays, vectors (Points/lines/polygons, KML files, SpatialLite tables)
- Actions: start searches, add visual elements, show GPS location on map
- Use Android native ZoomControls for the MapView element

5.3 BlackBerry sample

5.3.1 BlackBerry Mapper - UIApplication demo

- Basic mapping application is using BlackBerry UIApplication API instead of standard J2ME Midlet's LCDUI API
- Shows how to use Maps SDK in a native BlackBerry application.
- Includes basic functions: Map browsing, define controls, draw marker object on map
- Set map types, including offline maps

5.4 Java ME samples

5.4.1 Nutiteq Mapper

Mapper application shows the following features:

- Show splash screen during start-up
- Draw map on Canvas
- Map zooming and panning
- Search place-names, POIs and get Directions
- Draw Places on map
- Remove selected Places from map (menu > remove places)
- Show log messages from library
- Go to another geographical location (menu > Go to...)

- Map Component resizing and repositioning on screen
- Define custom cursor for map centre mark-up
- Show download counter as main screen overlay (since 0.8.0) and download traffic information details, from menu

5.4.2 Nutiteq Forms mapping

- Create screen Form with Map Item
- Add a Place to the Form map
- Show callback events from the Form (MapListener)
- Define controls: Zoom in and Zoom out (#, * and softkey selections), and for map panning (numeric keys) for the Map Item

5.4.3 Nutiteq KML Service

- Show splash screen during startup
- Draw map on Canvas
- Map zooming and panning
- Opens KML with Panoramio Popular places KML and shows on map, plus some other test KML files
- Shows rendering of point objects, with custom marker images
- Note that reading KML is not optimized here, every map moving and panning requests new set of Panoramio images, which is traffic-hungry and can be expensive.

5.4.4 Nutiteq Directions

- Selecting points from map – **click key “5”** to select start, and then end point for Directions
- Show line on map
- Show places (start and stop point) on map
- Select routing service: Cloud Made GPX API, yournavigation.org (global) or OpenLS based (Estonia only).
- For Cloud Made and Yournavigation API some additional parameters can be set: route shortest or quickest, pedestrian or car navigation etc
- CloudMade and OpenLS services enable to show also turn points on map.

5.4.5 Nutiteq Full Screen

- Switch map screen to Full Canvas mode, and back

5.4.6 Nutiteq Custom Elements

- Re-define textual label shown for Places when cursor is over them
- Re-define icons (markers) for Places
- Define custom graphics for Copyright overlay

5.4.7 Nutiteq Jar maps demo

- Use maps offline, from application package (JAR file)
- OpenStreetMap global map is included, with one zoom level (but it could have several levels, just depends how large JAR file is acceptable)

- Note that this way the application does not require any on-line connection

5.4.8 GPS Location demo

- Get user location via supported library methods:
 - Internal GPS via JSR-179 API
 - Bluetooth GPS. It has a feature to list Bluetooth devices in range, so user has to make the selection.
 - Cell-ID location, specific service for SonyEricsson JP 7.3 and newer phones, BlackBerry and some Motorola's phones
- Show GPS location on map using dynamic Place for that, instead of the usual immutable Place.
- Note that there is special dynamic marker object (GPSTMarker) for GPS locations

5.4.9 Hybrid maps demo

- Use offline "jared" maps (read from JAR package) for one zoom level
- If user zooms out from the level, then on-line maps are used

5.4.10 Stored maps demo

- User can browse file directory from where to read the files
- Most J2ME devices will ask user permission for every file system action (every listing of directory, every reading of file); also WTK emulator will do this. Maps are stored in many files, so you can expect far too many questions coming. There are two ways how to solve this:
 - Configure J2ME permissions for the application, so the asking is done only once (but this is impossible in many devices)
 - Sign the J2ME application, as this enables one-time asking configuration for almost any devices. You must have your own code certificate for this from Verisign or Thawte, or you need to proceed Java Certified process. Some operators may provide or even require own certification for your J2ME application. Ready-built applications by Nutiteq (in *samples/mapper/dist* directory) are signed by Nutiteq; if you build it yourself, then you have to use J2ME signing signature. We suggest Verisign or Java Verified signing.
- Demo uses JSR-75 file system API to access files. Some devices (Motorola, Siemens, IDEN) have their own API-s, and library supports also these. Application needs small modification to use these API-s, see Javadocs for *MotoFileSystem*, *SiemensFileSystem* and *IDENFileSystem* classes in the *com.nutiteq.utils.fs* package.
- Format for stored maps is described in a separate document (available from <http://www.nutiteq.com/> Developer section)
- A tool and some hints for stored maps creation is available at www.mgmaps.com/create

6 BasicMapComponent API

BasicMapComponent provides the main API for map manipulation and viewing.

Let's assume that the basic mobile mapping application has following functions:

- Show an interactive slippy map, which user can pan and zoom

- Application shows some own specific objects (like points of interest) as markers on map, the data is loaded on-line from application-specific server
- User can select a marker and view its details, e.g. photo or video about it

From mapping library point of view, the following general steps are needed:

- Create map object, link it to graphics canvas
- Handle map panning/moving (define and forward key codes)
- Add markers to map
- Handle selection of a marker
- Finish using map component (destroy the object)

Following call flow shows steps what application needs to perform, using Java ME as sample (other platforms are mostly the same). The flow is simplified; full application needs typically to be more complicated (handle own logics, GUI, data persistence etc).

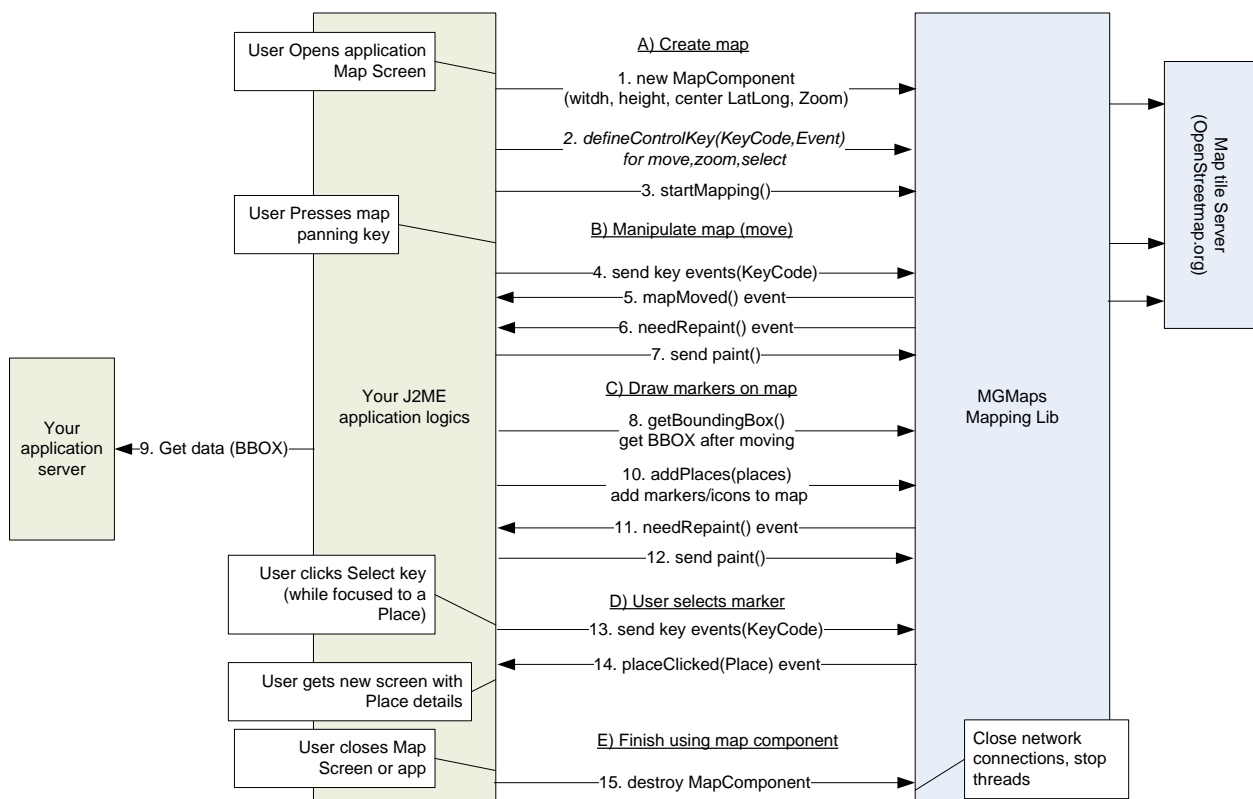


Figure 1 Interaction flow for basic map app

Some steps, like initiating listeners to listen events are not given here.

- a. **Create map:** As first step, new map object must be created and set up
 1. New object is created from *mapComponent* class
 2. The component needs also minimum configuration based on device and application. Application must configure map control keys (there are following controls available: UP, DOWN, RIGHT, LEFT, ZOOM_IN, ZOOM_OUT, SELECT). Specific devices have different key codes, e.g. these may have no arrow keys, and it is assumed that application will handle this diversity
 3. You must also initialize needed resources for mapping and start internal threads by calling *startMapping()* method
- b. **Map manipulation** is generally handled by the library map component, but the component will not do everything by itself.
 4. Application must send key pressing, releasing, repeating and pointer events to the map component, also paint events when necessary
 5. Map component will move or zoom map, according to the key events (and based on control key configuration), and it will download new maps if necessary
 6. After changing the map, the component will send *needRepaint()* event back to the application.
 7. As the following step, the application should react to it, typically needs to send *paint()* event to the library and platform to have final redraw. It depends on the platform, e.g. BlackBerry API requires *invalidate()* here
- c. **Draw markers on map** could have different strategies. If there are less than about 20 markers in total, then application can just add all of them to the map (also if they are actually outside of the current map view), and map component will show them. If there are significantly more of them, then phone memory (available heap size) could become a limitation, therefore it is not suggested to show more markers than 20 at a time.

If application requires more objects to be shown as markers, then there should be some kind of filtering, so only relevant ones will be added to the map. Let's assume that objects are kept in the server database, so they are requested over the air. Then also the download data amount should be reduced. Size per one server request should not be over 10-20 KB, to have a smooth user experience. One possible strategy to solve both problems (max amount of markers on map and download size) would be to use bounding box (BBOX) based filtering, combined with object count limitation in the server requests. Application will request objects from a server, giving BBOX and maximum number of objects (e.g. 20); the server will respond maximum of 20 objects which are within the given BBOX. Application should take area of current map image as BBOX value,.

8. Application sends *getBoundingBox()* method to map component, and gets bounding box (which is defined by two *WgsPoint* objects: min and max, where min is SW, max is NE).
9. Application makes request to back-end server. For example, if KML is used, then KML standard URLs allow also using BBOX value (in string format), so it is just a matter of server-side implementation for the filter. Also limit to number of return objects limit should be used, as with a wide range zoom too many (or all) objects could be inside defined BBOX, possibly more relevant objects should be given. Also server-side clustering strategy could be used

10. When the application has read requested objects, these should be added to the map, using `addPlaces()` method. The method accepts objects (places) one-by-one (`addPlace` method), or as an array (`addPlaces` method). If objects have extra data, then this should be handled by application, so user may get later details. Place for map component must include name of place and marker graphics for the icon. Optimal size of icon graphics depends on phone resolution, for typical phones 14x14 pixel images are ok, but for e.g. some high (VGA) resolution Nokia phones bigger icons should be used. Most phones accept alpha-channel graphics for transparency and it is better to use this
 11. Map component sends `needRepaint()` event to app, because new stuff was added to the map
 12. Application will send `paint()` to `mapComponent`, via application's Canvas
- d. **User selects a marker (Place)** by clicking `SELECT` button, while the map cursor is over some object; or by clicking directly on the object with touch-screen phones. Map Component will show tooltip style label with name of Place on map, then user knows that place can be selected
13. To be able to get event of place selection, application needs to send key pressing event with code of `SELECT`, and the key code of `SELECT` has to be defined also, this is typically done right after map component creation
 14. Map object clicking events are passed when **`OnMapElementListener`** is defined and implemented. Required methods are following:
 - `elementClicked(element)`** – when user clicks the object. With touchscreen, the user needs two clicks – first for “`elementEntered`” to view label, and another to actually select the object.
 - `elementEntered(element)`** - when cursor is on top of the element (hover). Label is displayed now automatically.
 - `elementLeft(element)`** – when cursor is not on the element anymore.
 15. Application sends **`elementClicked(OnMapElement)`** event, giving the selected map element as parameter. Now the application knows that a map object was clicked, and also its ID. Application can now show place details (description, address, phone number, actions, photos, videos, etc) as a new application-specific screen according to data known by application.
For KML places you can use `getAdditionalInfo(place)` method of `MapComponent` to retrieve extra fields of the placemark (name, description, snippet, address)
- e. Finish working with map
16. To free up connections and threads which are used by map component, it is suggested to call **`stopMapping()`** method when the specific map component is not needed any more

Note that similar data on-line loading, parsing and display could be also be done by library, using **`addKmlService()`** method.

7 SDK API methods

7.1 Methods of BasicMapComponent

BasicMapComponent deals with slippy map. It has many methods, these are described in the javadoc, which is available from library package and online from <http://www.nutiteq.com/javadoc/>. The main interaction classes are in package **com.nutiteq.BasicMapComponent**.

7.2 Methods for Geocoding and Directions searches

Maps Lib has methods for Placename and Directions searches, these are in package **com.nutiteq.services**. Current implementations allow to search placenames from Cloud Made Geocoding service (www.cloudmade.com), these are currently accessed from mobile using KML protocol via Nutiteq custom proxy. The proxy converts from Cloud Made JSON API to KML in lbs.nutiteq.com.



Figure 2 Proxy is used for Cloud Made Directions

Directions (routing) service supports Cloud Made GPX, Yournavigation and OpenLS APIs and these are accessed directly.

Note that OpenLS API is slightly modified: mainly to use HTTP GET request (which is more mobile friendly) instead of standard POST request.

Both services return data via callback (“waiter”) methods on the application side, specific code examples is included in the Nutiteq Mapper and Directions sample applications.

Geocoding methods are in package `com.nutiteq.services`:

- `GeoCodingService()` –search parameters are passed to method constructor, URL of service and also reference to the callback method (which needs to implement `GeocodingResultWaiter`) to retrieve answers. Additional parameters
 - Search type: nearest POI or address search
 - Search near point – `WgsPoint` to specify region where address or POI is searched
 - Categories – list of category id-s for more filtered POI search
- `execute()` – geocoding is executed by calling this method

Application will get geocoding answer as array of `KmlPlace` objects.

Directions methods are also in package `com.nutiteq.services`:

- `OpenLSDirections()` – constructor sets search parameters (start and end as `WgsPoint`), callback method to get answers and URL for the actual service
- `execute()` – runs routing

Result of Directions is a `Route` object, which is has line for full route and an array with route instructions. See directions example code for how to visualize the data on a map.

There is no internal implementation of Reverse Geocoding, but it is quite easy to implement the reverse geocoder using e.g. CloudMade REST-based API.

Note that Google Terms of Service does not allow for using their geocoder API with 3rd party maps, therefore using it with Nutiteq maps library is probably considered illegal by Google.

7.3 Methods for mobile positioning

Library has generic API to get device coordinates. It has several implementations, and application is expected to select one of these. See Location example to see how it should be done. Applicability depends not only on device platform, but also on specific device models; and in some cases even on specific mobile operators. Some operators (notably in the US) have blocked using internal GPS API via 3rd party unauthorized Java application.

7.3.1 Internal GPS API (Location API/SR-179)

Internal GPS is generally configured and determined by phone (Java platform implementation), from application point of view there is only question whether you'll get the location or not.

- This is available for Java ME and BlackBerry devices
- Windows Mobile JVM-s generally do not support JSR-179 API, even if the phone has internal GPS. Also serial and Bluetooth connections are not available for most JVM implementations, at least for the ones which are bundled with phones (Esmertec, JBed, Sun). Therefore currently the library has no method to get internal GPS data on these devices.
- Some phones (like SonyEricsson phones) do not allow to install and/or to run applications which try to use JSR-179 API methods. Therefore your application needs to have specific check to exclude method calls on these devices. Use following check to see if device has Location API support:

```
if (System.getProperty("microedition.location.version") != null) {...
```

7.3.2 Bluetooth GPS positioning

Nutiteq SDK supports external Bluetooth devices for J2ME. Bluetooth positioning API is similar to Internal GPS API, main difference is that Bluetooth positioning requires one additional set-up step: selection of particular GPS device.

There is no way to decide whether a Bluetooth device is GPS, so user has to make the configuration by selecting device by its name. In the first time phone platform will probably also ask for Bluetooth pairing code.

See Location sample application how to create Bluetooth device browser, Maps Lib includes also some helper methods to get list of Bluetooth devices in range.

7.3.3 Cell-ID Positioning

Cell-ID positioning service gets radio network data from the phone. This data (typically CID, LAC, Current MCC and Current MNC values, depending on device) identifies uniquely cell what phone handles as the current cell. This data is usually referred shortly as Cell-ID. The Cell-ID can be converted to geographical location, using database of Cell locations. There are some public databases to collect the Cell location

databases, one of these could be used. Opencellid.org API is pre-implemented in the library, and your own service should be set up for that.

- Library default implementation uses Opencellid.org Cell-ID geolocating service, which works globally, but has quite limited actual coverage
- Different phones have different methods to get Cell-ID data, there is no standard. Still many phones do not have any method to get the data, at least with Java application.
 - SonyEricsson JP 7.3 and newer software – Library has SonyEricssonCellIdLocationProvider method for these. It works with unsigned applications, without security checks
 - BlackBerry phones – can get Cell-ID for most modern OS/model versions, requires signed application
 - Motorola phones (not all of these) – Library has MotorolaCellIdLocationProvider. It requires that midlet is signed. Signing is possible via Javacertified or Motorola, usual 3rd party midlet signatures (Verisign, Thawte) do not work with Motorola.
 - SonyEricsson older phones - there is method to get Cell-ID
 - SonyEricsson Symbian phones – no Java method. Possible with native API
 - Nokia Series 40 phones – there is no method to get Cell-ID
 - Nokia Series 60 Symbian phones - no Java method. Possible with native API
 - Windows Mobile phones - no Java method. Possible with native API
 - Other Java phones (LG, Samsung etc)– do not have method to get Cell-ID via Java

7.4 Common public classes

Following universal classes are used for both Map Item and Map Component:

- Place – for places displayed on map
 - a. Id Integer
 - b. Name String
 - c. WgsPoint (long, lat)
 - d. Icon Image, image or Placelcon can be used
 - e. PlaceLabel - how place name is shown on map, as text label
- WgsPoint – main structure for geographical locations. It is expected in WGS84 coordinate system
 - a. Longitude in decimal degrees, float, i.e. 27.12345 , -12.32423
 - b. Latitude in decimal degrees, float
- WgsBoundingBox
 - a. WgsPoint min
 - b. WgsPoint max
- PlaceInfo – using to get details for the KML places which are read and parsed by the mapComponent. Values are read from the corresponding KML tags.
 - a. Address
 - b. WgsPoint Coordinates
 - c. Description
 - d. Name
 - e. Snippet
- Placelcon – used for places to display more parameters for the icon

- a. Image
 - b. Anchor point x and y (if not given, default anchor is used which is in the centre)
- PlaceLabel – used for customizing text label of place
 - a. displayStyle – define label placement (default = DISPLAY_TOP)
- ZoomRange – defines map max and min zoom
 - a. minZoom
 - b. maxZoom
- Line
 - a. Array of WgsPoint
 - b. LineStyle (optional)
- LineStyle
 - a. Color – int of RGB in the same form like J2ME uses (e.g BLUE = 0x0000FF)
 - b. Width in pixels
- Polygon and PolyStyle – enables filled and hashed polygons

7.5 Callback events to application

Some callback events are available from the map component; these are delivered to different *listeners*. The callback events are same for both level APIs: Map Item and Map Component, these are in package **com.nutiteq.listeners**

a) MapListener

- mapClicked() – when SELECT button was keyPressed. It requires that SELECT key was defined beforehand. Note that this event is not called if a Place is in the center of map; then only placeClicked(Place) is called
- mapMoved() – after map was moved using pointer or defined arrow keys, and key or pointer is released
- needRepaint() – after any change of map. For Map Component application is suggested to repaint the Canvas (map): call repaint() method of canvas, which will call paint of Midlet, which calls paintAt() of map component. Application may also choose not to call repaint. For Map Item the repaint is done automatically, so there is no need to make handler for this if high level API is used. The method has Boolean parameter (mapsComplete) which indicates whether map is complete, or still some parts of map are to be downloaded. Application can in practice ignore the parameter and call repaint of the Canvas with every needRepaint event; this could be useful if application will generate image (e.g. for MMS) from the map or makes some other extra processing of the map. needRepaint will happen if:
 - Map is moved (panned)
 - Map is zoomed
 - New part of map was downloaded to screen buffer and is ready to be displayed. The map is downloaded in tiles, so and if tiles are smaller (e.g. 64x64 pixels, depending on which map service is used) then there could be over ten tiles for full map image for full new map, and after downloading of each tile needRepaint(false) is called. After last tile needRepaint(true) is called.
 - New marker is added
 - New array of markers is added

b) OnMapElementListener

- elementEntered (Place) – map was moved (or place was added), so place is in focus (center of map). Map Component shows also tooltip with place's name. It assumes that markers are added to the map.
- elementLeft(Place) – place is out of focus now
- elementClicked (element) – map was, a place is in focus and user has pressed SELECT. It assumes that SELECT key code is defined in map component

c) ErrorListener

- networkError(String message) – notify map/content download errors in mapComponent
- licenseError(String message) – notify license check errors. Format “-<code>:<text>”

7.6 Using public interfaces for customization

Mapping library uses a kind of *strategy pattern* to extend functionality, and there are many extension points (Interfaces) open for application developers. So application can use one of the built-in strategies or implement own method for the same task. This strategy gives flexibility and is also obfuscator-friendly, so unused classes and methods are automatically removed by Java (J2ME) obfuscator, reducing size of application package (JAR file).

Open interfaces are following:

1) Mapping interfaces

- **GeoMap, UnstreamedMap**– enables to have custom on-line map sources. Some of the built-in implementations are:
 - a) OpenStreetMap(baseUrl = OSM_MAPNIK_URL, tileSize = 256, minZoom = 0, maxZoom=17, Projection = GoogleProjection) – Default tile server, this is OpenStreetMap.org Mapnik server
 - b) CloudMade(licenseKey, tileSize, mapLayout) – Cloud Made (www.cloudmade.com) map tile server
 - c) SimpleWmsMap() – enables to open OGC WMS services, version 1.1.1. It makes direct GetMap requests, and supports only EPSG:4326. Note that WMS requests are done for 256x256 pixel tiles (several requests per map screen), not for a screen, also *GetCapabilities* is not requested.
 - d) MicrosoftMap.LIVE_MAP – Bling map tiles
 - e) QKMap – Quad-key map service, flexible service for different services which use Quad-key addressing of map tiles, e.g. MapTP also Microsoft.
 - f) TMSMap – Tiled Map Service API service. Flexible service for different services which use similar API like OpenStreetMap by default uses (e.g. CloudMade)
- **GeoMap, UnstreamedMap**– use off-line map sources, read data from JAR file. Built-in implementations:
 - a) JaredOpenStreetMap(tileSize, minZoom, maxZoom) – load map tiles from the JAR resource, so application can work totally off-line if necessary. The map tiles must be in OpenStreetMap common tile system (in terms of x, y and zoom values and Google coordinate system), tileSize can be either 256 or 64, and minZoom and maxZoom limits available zoom range.

- Map tile files must be in PNG format and in the root directory of the JAR file (common j2me package builders will put resources there anyway), with file name format: <zoom>_<tile-x>_<tile-y>.png (e.g. 3_4_2.png would be tile with x=4 and y=2, zoom=3, which will cover north-east-central Europe, same image from OSM Mapnik server it is <http://tile.openstreetmap.org/3/4/2.png>)
- b) **StoredMap**(name, path, filesystem, readCacheConf) – reads stored maps in MGMaps format, from flash drive. Description of the file format is available from <http://www.nutiteq.com/libsdk.html>, document “MGMaps stored maps spec”
- **GeoMap, StreamedMap** – enables to have on-line maps which are streamed in terms of HTTP connections. Key feature of this API is that there is single http connection to request several 64x64 pixel map tiles, this makes downloading and display of maps significantly faster. See separate *MGMaps maps API* document for built-in streamed map API description.
- 2) **KML Service** is to define custom loader for external place data which is published in KML format. It has one built-in method for basic implementation:
- **KMLURLReader**(url, needsUpdate)
- 3) **Label** enables to define custom labels
- **PlaceLabel**(label text, style) – basic text label, style is one of DISPLAY_NONE, DISPLAY_CENTER, DISPLAY_TOP, DISPLAY_BOTTOM, and marks where label is written
 - **BallonLabel**(str name, str extrainfo) – nicer label next to the Place.

For method details and methods see the [Javadoc](#) and example applications.

7.7 Selecting mapping service

Nutiteq SDK supports many different map sources. By default we use on-line OpenStreetMap service. List of different map services is online in: <http://www.nutiteq.com/maps-supported-nutiteq-mapping-apps>, there is also code snippet examples.

See also Android sample application for usage of many map services. Also there is code to define raster tile overlay to create “hybrid” map.

7.8 Defining own on-line map service

Own on-line map data can be added by implementing interfaces Geomap and UnstreamedMap. The Library includes samples source where you can see how it was implemented with Yahoo maps, you can use this as a template. Also Library internal map service implementations will give useful hints.

The map requests (at least in the current version) are based on map tiles, just like in typical interactive web mapping services. This means that map is not fetched exactly for the current screen window, but images with particular fixed size (typically 256x256 pixels) are read (also map portions outside of map window) and the library puts together single slippy map from that. This has following advantages:

- Map moving in small portion does not necessarily require downloading new maps, so it is much faster and user gets immediately new portion of the map. This is why it is called slippy maps.
- Map portions will be updated after each downloaded tile, so user does not need to wait until whole map is downloaded
- Tiled maps can be cached easily, in both server and client side, or in intermediate smart proxies (e.g. geowebcache). Note that unlike web browsers and Javascript applications J2ME HTTP connection does not cache maps in phone/implementation level, but Maps Lib does have own internal in-memory caching.

Tile approach has also some drawbacks to be taken care of:

- For the single map view a more data is read. If phone has 256x256 pixel screen, then typically it requires 4 tiles to cover map, so for 256-pixel tiles 4 times more data must be read, so in the worst case four times more expensive wireless bandwidth is used. For this there is also solution: using smaller tiles, e.g. 64x64 or 128x128 pixels. MGMaps Lib supports this also, and some map tile providers support it also (e.g. Cloud Made provides 64x64 tiles as mobile tiles).
- Map edge problem: tile edges cannot be optimized for particular view window. This may mean that copyright in the tile edge will be repeated on map, or place-names of general area may be repeated in each tile. This problem can be solved by tuning map service specifically for tile generation; but sometimes, especially with 3rd party map services, this may be problematic.

7.8.1 Projections

You should know and use proper projection, in terms of Java your map service must extend it. Currently library includes implementations for EPSG:4326 (Longitude-Latitude, also known as Plate Carree or Geographic Projection, very usual for OGC WMS mapping services) and EPSG:3785 (also known as Google Spherical Mercator, or EPSG: 900913, it is most common in web mapping services). So you can extend either classes EPSG4326 or EPSG3785 from the package *com.nutiteq.maps.projections* to use these projections for your own map service.

Library enables also defining own additional projections, for this public interface *Projection* (from package *com.nutiteq.maps.projections*) must be implemented in the application. There are two methods to be implemented: conversion from your new projection coordinates to the global pixel coordinates and vice versa (*mapPosToWgs* and *wgsToMapPos*). Calculations must take also zoom into account, as pixel coordinate space is different in each zoom, see next chapters.

7.8.2 Zoom level system

MGMaps Lib uses by default internally integer numbers for zoom levels, where zero (0) is whole world, and each next zoom level is previous one divided by two. This is same system like e.g. OpenStreetMap uses. Some map services use rotated system, e.g. for Yahoo you have 0 as maximum zoom and world is 18. Some maps, e.g. ka-Map (<http://ka-map.maptools.org/>) style server uses zoom system with approximate scale numbers, this is implemented in KaMap mapping service.

If your map service uses different zoom system, then you should do appropriate conversion in the map service implementation.

7.8.3 Pixel coordinate space

Basic calculations in map service are made in global pixel coordinates, and map portion is calculated out from that. Pixel coordinates start from left-top, with x increasing to right, y increasing to the bottom. Each zoom level has own coordinate plane, for example with 256x256 pixel tiles you have following pixel coordinate planes:

- Zoom 0 – x 0...256, y 0...256
- Zoom 1 – x 0...512, y 0...512
- Zoom 2 – x 0...1024, y 0...1024
- ...

So size of coordinate plane in pixels can be calculated as: $tile\ size * (2^{zoom})$.

Now the next thing to know is how tiles are put to the plane of particular zoom. Basically they are aligned by the left-top corner, see following illustration for zoom=1:

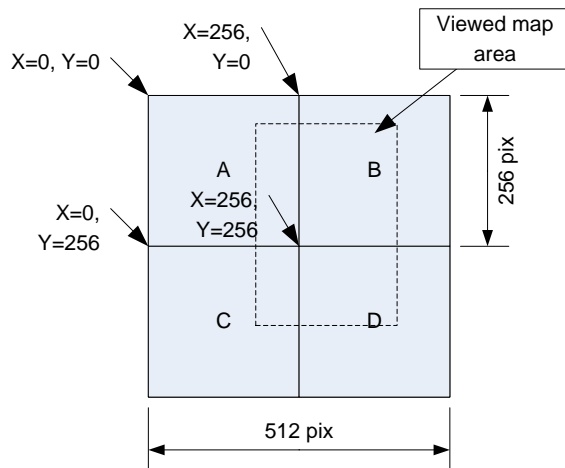


Figure 3 Coordinate plane for tiles, for zoom=1

So for tile A the pixel coordinates are (0,0), for tile B (256,0), for tile C (0,256) and for tile D (256,256). Dashed area marks current visible map area on phone's screen (Canvas).

7.8.4 Building URL for tiled map service

According to user actions and events sent to Map Component, the Library will give to your custom map service implementation the top-left pixel coordinate in this zoom's global coordinate plane, and zoom number. The key task of your map service implementation is to calculate URL for this map tile based on these parameters. For this map service must implement method BuildURL.

The simplest case is with OpenStreetMap 256-pixel tile numbering system: tile URL X and Y can be calculated simply by integer dividing of pixel x and y with tile size, and zoom level is same as library internal zoom. Therefore URL builder code is very straightforward:

```
public String buildPath (int mapX, int mapY, int zoom) {
```

```
final StringBuffer result = new StringBuffer();

result.append("http://tile.openstreetmap.org/");
result.append(zoom);
result.append('/');
result.append((mapX / tileSize);
result.append('/');
result.append(mapY / tileSize);
result.append(".png");
return result.toString();
}
```

7.8.5 Building URL for non-tiled map services

If your map service API is not based to tile coordinates, but is based to geographical or projected coordinates, then URL building is a bit more complicated. But this is also possible, if you use converter classes from the projection implementation. Following buildURL implementation builds OGC WMS 1.1.1 GetMap request (baseurl and tileSize value have been already set in the Map Service's constructor):

```
public String buildURL(final int mapX, final int mapY, final int zoom) {
    final StringBuffer result = new StringBuffer(baseurl);

    final MapPos minPos = new MapPos(mapX, mapY + tileSize, zoom);
    final MapPos maxPos = new MapPos(mapX + tileSize, mapY, zoom);
    final WgsPoint minWgs = mapPosToWgs(minPos).toWgsPoint();
    final WgsPoint maxWgs = mapPosToWgs(maxPos).toWgsPoint();

    result.append(minWgs.getLon()).append(",").append(minWgs.getLat()).append(",");
    result.append(maxWgs.getLon()).append(",").append(maxWgs.getLat());

    result.append("&WIDTH=").append(tileSize).append("&HEIGHT=").append(tileSize);
    return result.toString();
}
```

Note that also in this case map library will load maps just like these were tiles, so for each map screen there would be typically 4 map image requests. This enables slippy maps and caching of map images in both server and client level.

7.8.6 Streamed maps

MGMaps supports map tile "streaming" to get faster downloads. Streaming means that there is single HTTP connection used to load several smaller map tiles, this may reduce significantly download time due to network latency for new connections; and also user can see portions of map before all map is downloaded. The streamed map API is described in the MGMaps server API specification (in

www.nutiteq.com Developers section). The Mapper sample application uses some sample streamed map sources.

7.9 Other features

7.9.1 Pointer controls and cursors

Map component supports pointer events: dragging will move the map, and user can click on objects. This requires that the device has a touch-screen or pointing stick or similar pointer.

Application can also add graphical zoom controls overlay, similar ones like Google Map has in the web (zoom in/out, panning icons), so the icons are clickable with the pointer.

In other devices you may want to display a cursor in the middle of the screen, so user sees where selected object is. Default cursor is a small cross, but check from Custom Elements sample application how to define any other graphics there.

Code snippet to define zoom controls or cursor on screen, depending whether device supports pointer events:

```
if (hasPointerEvents()) {
    mapComponent.setOnScreenZoomControls(new OnScreenZoomControls(Utils
        .createImage(OnScreenZoomControls.DEFAULT_ZOOM_IMAGE)));
} else {
    mapComponent.setCursor(new DefaultCursor(0xFFFF0000));
}
```

7.9.2 Zoom indicator

Zoom indicator is temporarily displayed and animated overlay to show what zoom level is right now. Default indicator can be activated with following code:

```
map.setZoomLevelIndicator(new DefaultZoomIndicator(0, 1));
```

Check Mapper sample application to see how to define own customized zoom indicator.

7.9.3 Download counter

Maps library will do approximate counting of loaded bytes, and estimates data download amount based on this. Downloaded data number can be shown as a default or custom overlay indicator.

Relevant BasicMapComponent methods for default implementation are:

- 1) setDownloadCounter
- 2) setDownloadDisplay



8 Platform-specific development

8.1 Overview

Generally the mapping library is designed to be as device-independent as possible. Most of API methods are the same, independent on the platform. However, different Java platforms and producers have more or less different APIs, therefore there are some important differences how exactly mapping library is tied to the specific application platform.

There are three major flavors of mobile Java: Java ME (J2ME), RIM BlackBerry API and Google Android API. Java ME and BlackBerry share many API packages (JSR-s), therefore there are many shared packages. Android has more or less completely different phone APIs, therefore internal differences are bigger. However, for MGMaps API differences are not very significant for developers.

	Java ME	RIM BlackBerry	Google Android
Common mobile mapping API	Common API: <ul style="list-style-type: none"> • Slippy map • Map tile streaming • Map tile caching • Various map providers • Custom map providers • KML overlays • Places on map • Geocoding • Routing 		
High-level APIs	Custom Form Item: <i>MapItem</i> Key handler: <i>MGMapsKeysHandler</i>	FieldManager item: <i>MapFieldTouch</i> (in Blackberry Sample app) Key handler: <i>MGMapsKeysHandler</i>	<i>MapView</i> (special version by Nutiteq) Key handler: <i>AndroidKeysHandler</i>
Graphics back-end	LCDUI	LCDUI or RIM UIApplication	android.graphics
Cache options	Memory Cache RMS Stored maps	Memory Cache RMS Stored maps	Memory Cache, AndroidFileSystemCache, AndroidMbTileCache (SQLite - based)
File System	JSR75FileSystem MotoFileSystem IDENFileSystem	JSR75FileSystem	AndroidFileSystem – standard Java IO API file system
Location methods	<i>LocationAPIProvider</i> (JSR-179) NokiaCellIdDataReader SonyEricssonCellIdDataReader MotorolaCellIdDataReader BluetoothProvider	<i>LocationAPIProvider</i> (JSR-179)	Android Location API AndroidGPSProvider
Other	Nutiteq logger	Nutiteq logger	AndroidLogger AndroidHttpConnection

Table 1 Similarities and differences of the Library on base platforms

8.2 Java ME (J2ME)

Java ME has been our first target platform for the mapping library, therefore most sample applications are focused on this platform.

Basic minimum requirements from Java ME devices are MIDP 2.0 and CLDC 1.1.

Java ME has to handle much larger device and manufacturer diversity than other mobile Java platforms; therefore following aspects must be kept in mind while developing on that:

1. Selecting optimal memory cache parameters, to avoid Out of Memory exceptions.
2. Setting best size for map, according to available screen/canvas size
3. Selecting and defining specific key codes for map actions (zooming, panning)
4. Deciding whether to show on-screen controls (functional only with stylus or touchscreen)
5. Selecting best size for map markers. E.g. high resolution screen phones may be better with larger images for place Markers.

Sample applications have tested in following device platforms:

- SonyEricsson JP-7, JP-8, SJP-3
- Nokia S40 v2, v3 and v5
- Nokia S60 v3
- Motorola (Razr Vxx)
- Samsung
- Windows Mobiles with Esmertec JBed, TAO Intent, Sun and IBM J9 JVM-s
- Mpowerplayer, Microemulator, Sun WTK 2.5.2 MIDP 2.0/2.1-based emulators, some Nokia emulators

The sample applications may have functional limitations on certain devices, because of the different keycodes (of e.g. Motorolas), specific memory limitations or Midlet package system.

8.2.1 Using Forms and high-level Custom Map Item API

MapItem is J2ME custom UI element (CustomItem); it enables simpler usage of map inside application. It has slightly limited functionality, compared to MapComponent (which draws map on Canvas), but it is much easier to be used in screens using J2ME Forms.

To get started see separate document **Hello Map Tutorial** (see <http://www.nutiteq.com/> Developer section) and javadocs for full reference of MapItem methods.

Following are the key limitations of MapItem:

- CustomItem cannot receive arrow key events (which are usually used for map panning), because the arrow keys are reserved by device platform for switching between UI elements. Therefore map can be panned using other keys, e.g. numeric keys (8 – North, 2 – South etc). In some phones (e.g. Nokia S60 v3) also Select key is used in form to open menu, so it cannot be used for object selection.
- Screen graphics (layout) is controlled by phone implementation, and not by the application. MapItem (as a typical Custom Item) will only paint inside own box automatically, and screen with a Form is designed fully by phone implementation. This means that screen graphics control methods are not available in MapItem.

Simple, map as custom item can be created with just a couple of lines of Java. For example, we need map to show geographical location of some object (place). For this into Form object a mapItem should be added. Element's title will be "Map", license key is "abcdtrial", application MIDlet class is *Mapper*, suggested map item size is 320x200 pixels and center location is 24.764580, 59.437420 (Tallinn city), zoom level is 12. Code will be following:

```
mapItem = new MapItem("Map", "abcdtrial", Mapper.instance, 320, 200, new WgsPoint(24.764580,
59.437420), 12);
form = new Form("Here is Tallinn");
form.append(mapItem);
```

To show an icon marker (Place) on the center point, we could add following line (icon is Image object and needs to be created before):

```
mapItem.addPlace(new Place("Tallinn", icon, 24.764580, 59.437420));
```

Note that arrow keys of phone keyboard cannot be received to the Form Item, as these are used to move between the Items. This is general limitation of J2ME Forms. So to enable map panning application should define other keys, for instance following defines moving to north using numeric key [2]:

```
mapItem.defineControlKey(ControlKeys.MOVE_UP_KEY, Canvas.KEY_NUM2);
mapItem.defineControlKey(ControlKeys.MOVE_LEFT_KEY, Canvas.KEY_NUM4);
mapItem.defineControlKey(ControlKeys.MOVE_RIGHT_KEY, Canvas.KEY_NUM6);
mapItem.defineControlKey(ControlKeys.MOVE_DOWN_KEY, Canvas.KEY_NUM8);
```

By default [*] will work as zoom out, and [#] will work as zoom in key, there are no defaults for panning.

Finally, also mapping must be started. This line should be in startApp() method:

```
mapItem.startMapping();
```

Note that you should call stopMapping() method during closing of application. Otherwise in some phones some threads stay running and errors come when application is started again.

8.2.2 Minimizing package

It is strongly suggested to use obfuscator to pack your application before releasing your application. Maps Library includes many classes which are probably not used by the application and Java obfuscators will remove them from the final package, so your distribution JAR significantly smaller and faster. The free ProGuard is usually just fine, also other obfuscators can be used.

If your application does not use on-screen controls then it does need to have image for this and you can remove image *m-l-controls.png* from the library jar file. Similar thing for KML places: the library JAR has *def_kml.png* image as default icon; if default icons are not needed (your KML will have style definitions or KMLService is not used), then this image can be removed to reduce package size a little bit more. Obfuscator will not remove these images.

The library includes some 3rd party packages which may be useful also for your application; or maybe are already used: like XML parser, GZIP uncompressing etc. If possible we suggest using same packages from maps library, and then there will be no duplication of same or very similar class files in the JAR file. These packages are un-obfuscated to enable this; however, Maps Lib javadoc does not include their documentation. Get this from the original vendor's sources.

8.3 BlackBerry API

There are two methods how to create BlackBerry applications: MIDlets and so-called native BlackBerry applications (CLDC applications). The latter does not use MIDP, and provides BlackBerry own APIs instead.

Depending on application type you need to use also the mapping library differently:

8.3.1 BlackBerry Midlet

You can create a standard MIDP 2.0 J2ME midlet with your IDE (e.g. Eclipse) and LCDUI and convert final build package (JAD/JAR) it to the RIM native application package using *rapc* tool (included with BlackBerry JDE), or can use JDE to create Midlet type of application.

This method enables to have common similar application for RIM and standard J2ME devices, but the limitation is that you cannot use BlackBerry native UI libraries (UIApplication APIs). Also you have to take care some general BlackBerry development issues regarding Mapping Library:

- Use special MMaps Library JAR file: **bbmidlet_maps_lib-x.x.x.jar**
- Use component same way like in Java ME applications
- Do not obfuscate final application or parts of it. At least some obfuscators like proguard add headers to the class files which BlackBerry platform does not accept.
- BlackBerry has strict class verifier: make sure that your application does not include APIs which blackberry does not support.

8.3.2 BlackBerry connection parameters

BlackBerry TCP connectivity requires that you set correct "deviceside=xx" parameter value for downloader. The library has special method to set it:

```
final DefaultDownloadStreamOpener opener = new DefaultDownloadStreamOpener(";deviceside=true");
```

```
mapComponent.setDownloadStreamOpener(opener);
```

This has to be done before startMapping() method is called.

The DefaultDownloadStreamOpener has second optional parameter as connection timeout. By default BlackBerry has 120 seconds timeout for connection failures; and sometimes the connection failures may come in unexpected situations (e.g. from other network activities of your application). To avoid occasional long, 2-minute map loading “hangings” the timeout value should be set to some smaller value, but still big enough so normal map tile download is done during the time. It could be 10-20 seconds.

8.3.3 BlackBerry UIApplication API

You can use BlackBerry JDE or BlackBerry Eclipse plug-in to build RIM specific so-called CLDC applications, which use RIM specific APIs, including UIApplication API.

- Use special library version: **rimui_maps_lib-<version>.jar** , and different *UIApplicationDemo* sample.
- For BlackBerry API 6.0 and newer use **rimui6_maps_lib-<version>.jar** to avoid certain duplicate packages (json parser in particular).
- You can use lower level API BasicMapComponent in BlackBerry.
- Also you can use MapFieldTouch.java classes from Nutiteq Blackberry sample application. These provide convenient high-level UI element for mapping, if you use Field Manager UI control in your BlackBerry application.
- For BlackBerry API older than 4.7 make sure that you remove references to TouchEvent, otherwise the code will not run on device, with the nice Verification Error during app startup.
- If you link library as separate library project in JDE, then it is also handled and installed as separate module “maplib” for the BlackBerry Device; and it remains to separate COD file. This is also visible to the application installer. It generally works, but we suggest to embed library inside the same application project and compile to the same COD file, then it is hidden inside the actual application.

Following is a basic UIApplication mapping application sample code, with basic parts like imports skipped.

1. Create application main class, and open map screen during startup:

```
public class WrappedMapDemo extends UiApplication {
    public WrappedMapDemo() {
        pushScreen(new MapDemoScreen());
    }

    public static void main(final String[] args) {
        final WrappedMapDemo demo = new WrappedMapDemo();
        demo.enableKeyUpEvents(true);
        demo.enterEventDispatcher();
    }
}
```

Here is important to enable KeyUpEvents, as proper map panning (panning stop to be exact) depends on these.

2. Initiate a screen for the map

```
class MapDemoScreen extends MainScreen implements MapListener {
    private static final int SCREEN_WIDTH = Display.getWidth();
    private static final int SCREEN_HEIGHT = Display.getHeight();
    private final BasicMapComponent mapComponent;

    private Graphics wrapped;
    private com.nutiteq.wrappers.rimui.Graphics graphicsWrapper;
```

Here are two important fields defined:

- *wrapped*, which is UIApplication graphics object. This is where the map will be placed into.
- *graphicsWrapper* is reference to UIApplication graphics wrapper. You can think of it as a converter for MapComponent from Java ME to UIApplication graphics. You can see that the MapComponent will not draw graphics natively to UIApplication, but does it through Java ME API. This is invisible to you and does not create any overhead.

3. Create MapComponent object in constructor of the screen:

```
public MapDemoScreen() {
    final DefaultDownloadStreamOpener opener = new
    DefaultDownloadStreamOpener(";deviceside=true");
    mapComponent = new BasicMapComponent("licensekey", new AppContext(this), SCREEN_WIDTH,
    SCREEN_HEIGHT, new WgsPoint(-74.0, 40.717), 5);
    mapComponent.setMap(OpenStreetMap.MAPNIK);
    mapComponent.setDownloadStreamOpener(opener);
```

This is important to note that download stream handler is defined explicitly. Most real RIM deployment environments require that the "deviceside" parameter is added to the requests. There can be other possible values than "=true", but experienced BlackBerry developers know what to define there on which conditions. If no, you can get started from BlackBerry developer documentation and forums.

See also that you cannot reference Midlet in MapComponent constructor (as you do not have one): you must use explicit *appname* and *vendor* parameter values instead. Make sure that you used the same values in your license code request form.

4. Define following paint method using, graphics wrapper:

```
public void paint(final Graphics g) {
    if (wrapped != g) {
        wrapped = g;
        graphicsWrapper = new com.nutiteq.wrappers.rimui.Graphics(g);
```

```
}  
  
//paint on wrapper (in effect painting on native graphics)  
mapComponent.paint(graphicsWrapper);  
graphicsWrapper.popAll();  
}  
  
public void needRepaint(final boolean screenComplete) {  
    invalidate();  
}
```

5. Implement listener for navigationMovement to enable map move using the scrollbar:

```
protected boolean navigationMovement(final int dx, final int dy, final int status, final int time) {  
    mapComponent.panMap(dx * 10, dy * 10);  
    invalidate();  
    return true;  
}
```

Note following here:

- map panning is done by 10 pixels for each movement. Otherwise scrolling is too slow. You can have more complex acceleration logic here.
- invalidate() call has same effect like repaint() in Java ME applications

6. Implement KeyDown and KeyUp methods:

```
protected boolean keyDown(final int keyCode, final int time) {  
    mapComponent.keyPressed(Keypad.key(keyCode));  
    return false;  
}  
  
protected boolean keyUp(final int keyCode, final int time) {  
    mapComponent.keyReleased(Keypad.key(keyCode));  
    return false;  
}  
  
protected boolean keyRepeat(final int keyCode, final int time) {  
    mapComponent.keyRepeated(Keypad.key(keyCode));  
    return false;  
}
```

Note here that generally the methods should return **false**. Otherwise device assumes, that your application has done all necessary handling for the keypress, and all the keypresses get trapped to your application. This includes important keys what you probably want system to handle: the green and red button, menu button, back button. So if you return here true, then it may be hard to find even method to exit your application, as it looks like none of the device keys work.

7. Make sure that application has clean exit:

```
public boolean onClose() {
    mapComponent.stopMapping();
    System.exit(0);
    return true;
}
```

8. Implement your specific map manipulation and business logics

After *mapComponent* instance is created you can use all the methods of it, exactly same way like in other Java variations: set center, zoom, add KML layers, add markers etc.

Do not forget to call **startMapping()** to actually start map download and painting threads. Otherwise the app does not show anything for the map.

See full source code in BlackBerry package "UIApplication sample" application.

8.3.4 BlackBerry UIApplication high level API with FieldManager

Library sample application package includes now MapField.java, which is high-level map manipulation element. You can use it like following:

```
// create a title field for your screen
    LabelField title = new LabelField("Map screen", LabelField.ELLIPSIS |
LabelField.USE_ALL_WIDTH);
    setTitle(title);

// create FieldManager
    fieldManagerMiddle = new VerticalFieldManager();

// create Map and add it to screen (FieldManager)
    map = new MapFieldTouch("Map", "NUTITEQ-LICENSE-KEY", "MyCompany", "MyApp", 320,
450, new WgsPoint(-71.023865,42.416867),12);
    fieldManagerMiddle.add(map);

// do not forget set your correct parameter for BlackBerry BIS/BES/APN-based connections
    map.setDownloadStreamOpener(new DefaultDownloadStreamOpener(";deviceside=true"));

// start mapping, this is also mandatory
    map.startMapping();
```

For Touch-enabled devices (Storm, Storm 2) there is different class MapFieldTouch.java, with only difference that it handles also touch events. Due to BlackBerry API specifics you must have anyway different application if you use API elements (like touch events) which are available for specific API versions only.

8.4 Google Android platform

You must use special Android maps library package, and **android_maps_lib-<version>.jar** library file from it, as Android graphics implementation is specific.

To make open sourced mapping easier and consistent with Android SDK, the Library includes similar high-level MapView object like bundled Google Maps API, and this view can be used in your application similar way, as part of Layouts. However, specific Mapping Lib methods like adding overlays and markers, setting map types etc are not similar to the Google Maps.

8.4.1 Getting started sample

Android mapping development basic steps are following:

1. Define Activity for your application:

```
import com.nutiteq.android.MapView;

public class AndroidMapper extends Activity {
    private MapView mapView;
    private BasicMapComponent mapComponent;
    private boolean onRetainCalled;
```

Here it is very important to import **com.nutiteq.android.MapView** from the mapping library, and not *com.google.android.maps.MapView*, what e.g. Eclipse can also offer to you. Basically, this import does the main switch from Google Maps to Nutiteq maps library.

2. Create MapComponent object, here it is done right in startup of your Activity:

```
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    onRetainCalled = false;

    requestWindowFeature(Window.FEATURE_NO_TITLE);

    final Object savedMapComponent = getLastNonConfigurationInstance();
    if (savedMapComponent == null) {
        mapComponent = new BasicMapComponent("abcdtrial", "Nutiteq", "Android Mapper", 1, 1,
            new WgsPoint(24.764580, 59.437420), 10);

        mapComponent.setMap(new CloudMade("cloudmade key", 256, 1));

        final MemoryCache memoryCache = new MemoryCache(1024 * 1024);
        mapComponent.setNetworkCache(memoryCache);

        mapComponent.setPanningStrategy(new ThreadDrivenPanning());
```

```
mapComponent.setControlKeysHandler(new AndroidKeysHandler());

mapComponent.startMapping();

} else {
    mapComponent = (BasicMapComponent) savedMapComponent;
}
```

The code needs to include control whether mapComponent is already started. This is needed, as in Android it is always possible that Intent is re-activated from the background.

Note there here lowest level of the Map library object is used here, which is BasicMapComponent. This does not have default parameters like MapComponent has (map type, caching, panning strategy), so you need always remember to define these for it. Otherwise its methods are the same as in MapComponent. Cache size in memory is defined to 1 megabyte here.

See also that you cannot reference Midlet in MapComponent constructor (as you do not have one): you must use explicit *appName* and *vendor* parameter values instead. Make sure that you used the same values in your license code request form.

Mapping is started now.

3. Create MapView object and Layout, and put MapView to the layout:

```
mapView = new MapView(this, mapComponent);

final RelativeLayout relativeLayout = new RelativeLayout(this);
setContentView(relativeLayout);

final RelativeLayout.LayoutParams mapViewLayoutParams = new RelativeLayout.LayoutParams(
    RelativeLayout.LayoutParams.FILL_PARENT, RelativeLayout.LayoutParams.FILL_PARENT);

relativeLayout.addView(mapView, mapViewLayoutParams);

mapView.setClickable(true);
mapView.setEnabled(true);
```

You noticed that in this sample the layout is created programmatically; but you can get similar result by using XML definition for the layout.

4. Do clean-up of mapComponent when Activity is finished, remember state during configuration change:

```
public Object onRetainNonConfigurationInstance() {
    onRetainCalled = true;
    return mapComponent;
}
```

```
protected void onDestroy() {
    super.onDestroy();
    if (!onRetainCalled) {
        mapComponent.stopMapping();
    }
}
```

Check sample application from the Android library package to get full source code of a simple mapping Activity. It includes also code how to add Android native zoom controls to the map.

8.5 Using J2ME Polish Pro for application development

J2MEPolish from Enough Software (www.j2mepolish.org) is the leading open source Mobile Java toolkit, which enables cross-platform development of mobile applications. It provides tools for simplify device diversity handling on Java ME, and also converts applications built on Java ME to BlackBerry and Android platforms.

If you use J2MEPolish UI polishing features, then you need to use LCDUI Forms API and high-level Form elements. Map would be also a CustomItem of Form here. Nutiteq library already has Form element: MapItem, but J2MEPolish cannot convert it from the library package, as it wants it as part of application source code. The solution to this problem would be to create own similar MapItem to your application package. J2MEPolish mapping sample application includes MapItem.java, which is actually copy of the MapItem in the Library own package.

General procedure of adding maps to your J2MEPolish application are following:

1. Include mapping library to your J2MEPolish application project, using always Java ME version of it, regardless of your final platform. You need to modify your J2MEPolish build scripts for it, inclusion to build path in Eclipse is not enough. Add following parameter to your <build> target context:

```
<libraries>
    <library file="${maps.lib}" />
</libraries>
```

This requires also to modify build.properties file to add reference to the library JAR file, something like:

```
maps.lib= ${basedir}/lib/maps_lib-1.0.0.jar
```

2. Create MapItem class as CustomItem and wrapper to MapComponent. You can create your own, copy the one from the sample application or take from library source package (thiw has the latest one), and if needed extend it.

```
import com.nutiteq.MapComponent;
...
    public class MapItem extends CustomItem implements MapListener, OnMapElementListener {
...
    }
```

3. Create map object of MapItem, start mapping. See previous samples, map handling code is the same.

4. Screen which displays the map

```
public class MapScreen extends Form implements CommandListener, PlaceListener {
    public MapScreen(final MapItem mapDisplay) {
        //style mapScreen
        super(null);
        this.mapDisplay = mapDisplay;
        mapDisplay.setPlaceListener(this);
        mapDisplay.resize(getWidth() - 5, getHeight() - 10);
        instance = this;

        this.append(mapDisplay);

        setCommandListener(this);
    }
}
```

Note here following:

- There is “//style mapScreen”, J2MEPolish preprocessing command which enables to restyle map screen using CSS.
- There is method for resizing mapDisplay (a MapItem object) according to screen size. Original MapItem had arbitrary size, as form size was not known then
- MapItem is added to Form just like any other Form item

If you want to use library features which are made specifically for other platforms, e.g. Android FileSystemCache, then you need to include and use also Android library package and preprocessing.

J2ME Polish distribution package (since 3.1 version) includes sample mapping application with Nutiteq openstreetmap mapping library. It is suggested to use this sample, as it has latest J2MEPolish and build scripts which are compatible with it.

9 Caching

9.1 Cache levels

Nutiteq mapping library can cache map tiles, and there are several levels to do it.

- 1) Memory cache is keeps latest loaded tiles and other images (like KML placemark icons) in RAM. The size of it is smallest, and must be taken in control depending on specific device capabilities

to avoid out of memory situations. This cache is not persistent, therefore it is cleaned with application restart.

- 2) RMS cache is available in Java ME and BlackBerry, not on Android. This is persistent and will live over application restarts. However, many some devices have size limitations for it, and so sometimes maximum size of it can be no more than some hundreds of kilobytes. Some other modern devices allow it to grow to megabytes without issues.
- 3) File system cache is available only on Android. It is possible to use default cache directory (kind of temp directory), or specific directory on device or flash card to be used for tile caching. This cache is persistent in principle.
- 4) Stored maps feature can be regarded as a read-only cache. This is available for Java ME and BlackBerry platforms only. Something outside library: like application or end-user must be take care of copying or downloading tiles to the cache directory when this is used. See stored maps demo and documents for more details about it.

Maximum size of all the cache levels can be reconfigured by application. MapComponent and MapItem elements use some default values for the in-memory caching. Currently it is not possible to define cache expiration time period.

You can assume that average map tile with 256x256 pixels and 16-colour PNG format takes a bit below 10KB, and one map view is 4-6 tiles.

9.2 RMS cache

Map component uses own RMS storage to cache KML icon and map tile images. Library generates several RMS storages, with name starting with ML_NETWORK_CACHE_. As it is part of same Midlet, then application Midlet has technical access to the RMS, but structure of the RMS is not public and application direct usage of this is not supported. Therefore Midlet just has to take care that same RMS name is not used.

9.3 File system cache

Android has no RMS, and for Android library uses file system for persistent cache, and SQLite database table to store cache index. The file format and structure is there different from the stored maps (which has no index in database).

Following code shows how to turn on file system caching on Android, together with memory cache. Note that application has to specify safe directory where to keep the map images.

SQLite at least on first Android devices is too slow to store map data directly on the database.

```
final MemoryCache memoryCache = new MemoryCache(1024 * 1024);
final File cacheDir = new File("/sdcard/maps_lib_cache");
if (!cacheDir.exists()) {
    cacheDir.mkdir();
}
final AndroidFileSystemCache fileSystemCache = new AndroidFileSystemCache(
    this, "network_cache", cacheDir, 1024 * 128);
mapComponent.setNetworkCache(new CachingChain(new Cache[] {
    memoryCache, fileSystemCache }));
```



nutiteq