

Porting BREW Games To Java™MIDP

Version 1.0; June 28, 2004

Mobile Games

NOKIA

Contents

1	Introduction	4
2	Designing BREW Games for Ease in Porting	4
2.1	Architecture and Heap	4
2.2	Native Calls	5
2.3	State Machine	5
3	Performing the Port	6
3.1	Turning #define into Integer Constants.....	6
3.2	Converting Game Data	6
3.3	Timers	7
3.4	Graphics.....	8
3.5	Full-Screen Canvas.....	8
3.6	Key Presses	8
3.7	Device Testing	9
3.8	Text Strings.....	10
4	Supporting Multiple Devices	10
5	MIDP 2.0 and BREW 2.0	11
6	Conclusion	11
7	Terms and Abbreviations.....	12
8	References.....	12

Change History

June 28, 2004	V1.0	Initial document release

Summus, Inc. and Thumbworks, Inc. are contributors to this article.

Copyright © 2004 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

1 Introduction

The Java™ 2 Platform, Micro Edition (J2ME™) is supported by mobile devices worldwide, by virtually all mobile device manufacturers, and across all air networking standards. However, a variety of other development environments are also used to develop mobile games. One such environment is Binary Runtime Environment for Wireless (BREW), a standard used widely in North America and parts of Asia.

The purpose of this document is to assist developers who want to port BREW games to the J2ME platform, and who want to plan the development of BREW titles to make it easier to port them.

The document assumes that readers are quite familiar with both C/C++ development and Java application development, and have some familiarity with both BREW and J2ME technology.

In many cases, without too much effort, it is possible to port existing BREW games to the J2ME platform in order to increase the number of devices and regions where the game is available.

2 Designing BREW Games for Ease in Porting

2.1 Architecture and Heap

The single most important architectural consideration for creating easy-to-port BREW games is to separate the game logic from the user interface code. Ideally the game logic should be its own class, and take the key presses or networking events and change the state of the game, nothing else. Trying to draw parts of the screen during the game logic means that the port will probably have to move all the game logic into the `paint()` function or force the use of an extra screen-sized buffer.

A better method is to design the game so that it only keeps track of where things should be painted, and then pass this object to a function that knows how to paint the game screen. This gives the porter more freedom in implementing the painting process in a way that is suitable for the Java platform. The second big advantage to this method is that because the Java and C languages share the same basic syntax, the conversion of the game logic can be exact, often requiring no more than simple search-and-replace operations to remove the “`pme->`” prefixes. If done correctly, no other changes have to be made to the game logic’s source code. This also eases the synchronization of bug fixes or upgrades to the BREW source, because the code can be directly copied over.

To ensure a simple architecture, thereby making it easier to port from BREW to the J2ME platform, a BREW game should bring as much of the working set for the game into heap memory — data, graphics, sounds, and other assets — rather than swapping assets in and out as needed. This produces straightforward code that is relatively easy to port.

This is not feasible for every game, particularly if heap limitations are tight. When porting a BREW game developed in this manner to a Mobile Information Device Profile (MIDP) device with tightly constrained heap memory, such as a Series 30 device, or to the Series 40 Developer Platform, you may find it necessary to swap assets in and out of heap to keep memory use within those limitations. (Series 40 devices typically have about 200 kB of heap memory.) However, it is generally easier to begin with a simple architecture and modify it as necessary to suit a device’s limitations, than to begin with a complex one.

In general, through careful pruning of image and sound data, it is usually possible to have the game run smoothly on almost any device.

2.2 Native Calls

Examples of the different computer code styles appear in the following sections.

BREW makes extensive use of callbacks for handling events, while by contrast J2ME provides most of its event handling through overloading methods in the `Canvas` class. If callbacks and native calls are used too extensively, modifying the code for the rather different J2ME platform environment will be difficult.

As an example, it is wise to use only one native timer in your BREW application, with a subset of timers managed on top of the main timer. This way, there is only one native entry point for timers into the application, which makes it easier to migrate to the J2ME platform.

Similarly, your BREW application should pass all of the key handling and suspend/resume handling behaviors into helper functions to eliminate the need to redo portions of your application's lower-level code.

The BREW SDK also provides many extensions that can be used to create screens of text (`IStatic`), menus (`IMenuCtl`), and text entry (`ITextCtl`). The implementations of these extensions vary from device to device, and are difficult to port to the J2ME platform. Although it will result in more work in the beginning, creating custom versions of these extensions for your own applications is strongly recommended. The result will look more consistent across multiple devices and, since you will have more control over the colors, text, and images used, the application will also look more pleasing.

2.3 State Machine

Another useful rule when building a BREW application with an eye to J2ME platform portability is to make the game as state-driven as possible. If the game is set up in BREW as a set of state machines, the language constructs in both BREW and the J2ME platform will be very similar, reducing your porting time.

In BREW, this looks like:

```
void SGame_SetState( SGame *pMe, int NewState )
{
    pMe->m_OldState = pMe->m_State;
    pMe->m_State = NewState;

    switch( NewState ) {
        case GS_NEW_GAME:
            SGame_NewGame( pMe );
            break;
    }
}
```

and for the J2ME platform, it becomes:

```

public class CGame {
    void SetState( int NewState )
    {
        m_OldState = m_State;
        m_State = NewState;

        switch( NewState ) {
            case GS_NEW_GAME:
                NewGame();
                break;
        }
    }
};

```

3 Performing the Port

3.1 Turning #define into Integer Constants

In C, most of the switch-case statements are created using `#define` preprocessors, while the J2ME platform version will need to create a set of integer constants. This process can be automated with a simple regular expression (commonly known as a “regex”) search-and-replace operation. Most common developers tools support using regular expressions but if yours doesn’t there are many freely available editors that do. The following strings may need to be adjusted slightly to conform to the exact syntax of your editor, but once correctly formatted it will perform this tedious conversion automatically.

```

Find String: ^#define[\t]+(.*)[\t]+(.*)[\t\n\r]
Replace String: static final const int \1 = \2;

```

```
#define IDC_NUMBER_PLAYERS 10
```

becomes:

```
static final int IDC_NUMBER_PLAYERS = 10;
```

3.2 Converting Game Data

The next step is to take all of the `structs` that are defined in the BREW application, including the global application context, and convert them to Java classes. One approach is to create a J2ME global game class (`CGame`) from the BREW application context, and have the game MIDlet contain an instance of this class. Since the BREW system is heavily callback driven, each callback contains a pointer to the global class. Each of these functions can be added easily to the `CGame` class. Your favorite code editor’s search-and-replace function will take a function such as:

```

void SGame_DisplayCurrentScore( SGame *pMe, int Data )
{

```

```
...
}
```

and replace it with:

```
public class CGame {
    void DisplayCurrentScore( int Data )
    {
        ...
    }
};
```

Once this has been accomplished, the majority of the game logic is now ported over. Arrays that were formerly declared in the old structures need to be “new”ed, “#define” variables need to be replaced with integer constants (with `static final const int`), and a number of other changes need to be made.

Once those changes have been completed, it is necessary to create native drawing, sound, key-press, and timer routines. By minimizing the number of callbacks used natively in your BREW application, it is easier to write entry points to the J2ME platform version of the game.

3.3 Timers

One approach is to set up the BREW application so that a generic callback handles all timers:

```
SGame_HandleTimer( SGame *pMe )
{
    ... // Timer system handled here

    ISHELL_SetTimer (pME->applet.m_pIShell, TIME_TO_NEXT_FRAME,
        (PFNOTIFY) (SGame_HandleTimer), pMe);
}
```

For the J2ME platform, a class can handle timers in a parallel fashion:

```
handleTimer();
int currentTime = System.currentTimeMillis();
if( currentTime > m_nextFrameTime ) {
    m_nextFrameTime = currentTime + 1;
}
sleepCurrent( m_nextFrameTime - currentTime );
m_nextFrameTime += TIME_TO_NEXT_FRAME;
```

By setting `TIME_TO_NEXT_FRAME` in BREW to be based on the number of milliseconds to sleep, and defining it in the J2ME version as an integer constant (`static final const int`), it is easy to have the `SGame_HandleTimer()` event running the same on both devices.

3.4 Graphics

Similarly, graphic utilities need to be added. Functions that provide clipped drawing access in BREW to bitmapped images can be migrated from BREW to the J2ME platform without much trouble.

For example, the BREW C code:

```
void SGame_DrawImage( SGame *pme, IImage *Image, int X, int Y, int XSize,
int YSize, int XOffset, int YOffset );
```

will become this J2ME platform code:

```
private final void drawFrame( Image img, int X, int Y, int XSize, int
Ysize, int Xoffset, int Yoffset );
```

3.5 Full-Screen Canvas

In an ideal world, graphics are modified for devices with different screen sizes, and different versions of the game are compiled to ensure that each player gets the best gameplay experience.

If the game is designed so that the areas at the edge of the screen are less important in play — for example, in a racing game where the player-controlled car always remains at the screen center — it's possible to use the same graphics for most devices. Small screens will simply omit the less important areas around the edge of the view, and large screens may have some blank space around the play area. This is not optimal, but if the desire is to accomplish as many ports to as many devices in as short a time as possible, it is a quick-and-dirty solution.

The trick is to put an offset into the graphical drawing routines that will measure the screen resolution and the maximum art asset size, and move all the graphics accordingly.

```
xOffset = (xScreenSize - xImageSize)/2;
yOffset = (yScreenSize - yImageSize)/2;
```

In the ideal case (that is, with graphics perfectly matched to the screen size), `xOffset = 0` and `yOffset = 0`.

This allows you to modify the game quickly for many different screen sizes.

3.6 Key Presses

In MIDP 1.0, key presses are handled poorly. You need to have a `keyPressed()` method in your class. To process input, the key-press function is called from a thread that is different from your game loop thread. In addition, the key codes are not always the same from device to device.

One solution is to create a `CDeviceData` class that contains specific mapping of key codes for a particular device, as well as other device-specific functions such as sound APIs and constants that differ from device to device.

For example:

```
class CDeviceData {
    static final int SCREEN_X = 128;
    static final int SCREEN_Y = 128;
    public static void keyPressed( int keyCode ) {
        // put key handlers here
    }
}
```

and in your main Canvas key-press handler:

```
protected void keyPressed(int keyCode)
{
    deviceData.keyPressed(keyCode);
}
```

Keep the `CDeviceData.java` file in a separate directory. Then, you can simply modify the file for each new device: to accommodate its screen size (if you aren't modifying graphics by screen size); keypad layout; and any APIs or other functions such as vibration, sound, and custom graphic routines available on that device. The different `CDeviceData.java` files can be reused for future ports of other games, as long as you adhere to the same basic object and method invocations in each — and if not, you can probably modify the `CDeviceData.java` file to accommodate whatever special features the new game needs, saving development time.

3.7 Device Testing

Once the application is converted and running in the emulator, the application must be tested in the target device. A Series 40 device such as the Nokia 7210 mobile phone is a good device to test the J2ME platform version on, for two reasons. First, because the Nokia 7210 phone has an infrared port, it is easy to load the application to the phone for testing with off-the-shelf parts including a USB-to-serial converter available from mobile stores. (Most other manufacturers require applications to be downloaded over the air.) Second, Series 40 devices are quite constrained: their maximum MIDlet size is 64 kB, and they have around 200 kB of heap memory. This means that if your MIDlet runs well on a Series 40 device, it should run well on other J2ME platform devices as well.

A note on device testing: one of the best ways to shorten the amount of time needed to debug the game on the physical device is to first run it in every J2ME emulator you can get your hands on. Each of the many emulators behaves differently, and this can often bring to light bugs that are very difficult to detect on the device. An example of one type of problem this can find appears below:

```
void keyPressed(int key){
    repaint();
    image = Image.createImage("dog.png");
};
void paint(Graphics g){
    g.drawImage(image,0,0,0);
};
```

This code may execute perfectly on the emulator you tested it on because it waits until the UI thread has finished the `keyPressed` function before calling `paint`, but on the real device the `paint()` may be called immediately and the application will crash with little to no debug information. Running this MIDlet on a range of emulators can help you find this sort of problem as well as other threading related bugs such as deadlocks.

3.8 Text Strings

In addition to migrating the code from BREW to the J2ME platform, issues with text and strings also need to be addressed. BREW provides an API that reads out strings from the resource file based on a 16-bit integer. The tools for creating these resource strings store the data in an intermediate binary format and compile it to a binary file that is loaded to the phone. Thus, the text data in the game often needs to be retyped for the J2ME platform version.

If heap memory is a serious concern, then it may be necessary to put the text into some sort of resource file or use the following trick. If the text is made a field in a class, it is loaded onto the heap when the class is loaded, taking up valuable heap space. A trick to reduce the amount of heap memory the text takes up is to spread it among a few different classes, only instantiate the class that has the necessary text for the current state, and keep all the text nonstatic. Nonstatic fields are only on the heap when the class is alive; when the class is unloaded the text is taken off the heap. Static fields are loaded permanently onto the heap by the class loader when the application starts up, and that space cannot be reclaimed.

4 Supporting Multiple Devices

Managing the port of a game for the J2ME platform across multiple devices can be difficult without something as powerful as the C preprocessor. In fact, some developers run the C preprocessor over J2ME platform code to handle device-specific issues.

Another solution is to create a device class that contains device-dependent constant data, as suggested in Section 3.6, Key Presses. In addition to handling screen offsets and key presses, this might be the most suitable class for managing sound, since sound APIs vary widely from manufacturer to manufacturer. In the case of Nokia devices, this class might also include the code that uses elements of the Nokia UI API, such as `FullCanvas`, which allows a game to access the full screen, something not included in the standard MIDP 1.0 specification.

To take full advantage of these device-specific classes, create a project structure in which the device-specific data files are copied into two directories: a directory that is separate from the common code base and the common directory from which the Java Archive (JAR) and Java Application Descriptor (JAD) files are created.

With the use of build tools like Apache Ant (see Chapter 8, References), managing this process is straightforward, and makes for significantly more readable code than some C projects that use `#define`'s throughout the source.

5 MIDP 2.0 and BREW 2.0

Both MIDP 2.0 and BREW 2.0 include some custom game-related classes and libraries that did not exist in the 1.0 versions. These new classes for sprites and media will make it easier to develop mobile games.

When moving older games to newer devices with MIDP 2.0, it is possible to reduce the amount of code in older BREW applications that, for example, perform sprite management. However, it may take longer to complete the port if developers are overzealous in their use of the new features available in MIDP 2.0. In some cases, it may be easier to implement the game requirements using the earlier J2ME platform primitives.

The introduction of `GameCanvas` will help by reducing the amount of extra calls required to handle key presses and the like, since these can be inlined in the main game loop. This scheme will mirror the BREW event handler very closely. In the case of Nokia devices, it will mean that the code will not require the use of the Nokia UI API, making the code more easily portable to other MIDP devices.

6 Conclusion

By applying a set of simple guidelines for both initial development and porting, it is possible to have a J2ME platform version of a BREW application up and running in very short order. With the development of a simple and reusable framework, it is easy to scale this process across multiple titles to more effectively leverage games into the marketplace.

7 Terms and Abbreviations

Term or Abbreviation	Description
BREW	Binary Runtime Environment for Wireless. An environment for running applications on mobile devices that is mainly restricted to phones for CDMA networks in North America and parts of Asia.
J2ME™	Java™ 2 Platform, Micro Edition. The version of Java technology that runs on mobile devices worldwide.
MIDP	Mobile Information Device Profile. A “profile” for the J2ME platform that defines Java technology functionality on mobile phones.

8 References

BREW Developer Resources Site:

www.qualcomm.com/brew/developer/resources/dev_resources.html

J2ME Developers' Site:

<http://java.sun.com/j2me/>

Apache Ant site:

<http://ant.apache.org/>