developer WORLD

Developer guidelines:

Sample Extensions Tutorial

January 2012

Sony Ericsson
make.believe

Document history

Version		
January 2012	First released version	Version 1

Sony Ericsson Developer World

For the latest Sony Ericsson technical documentation and development tools, go to http://developer.sonyericsson.com.

This document is published by:

Sony Ericsson Mobile Communications AB,SE-221 88 Lund, Sweden

www.sonyericsson.com

© Sony Ericsson Mobile Communications AB, 2009-2011. All rights reserved. You are hereby granted a license to download and/or print a copy of this document.
Any rights not expressly granted herein are reserved.

First released version (January 2012)

This document is published by Sony Ericsson Mobile Communications AB, without any warranty*. Improvements and changes to this text necessitated by typographical errors, inaccuracies of current information or improvements to programs and/or equipment may be made by Sony Ericsson Mobile Communications AB at any time and without notice. Such changes will, however, be incorporated into new editions of this document.
Printed versions are to be regarded as temporary reference copies only.

*All implied warranties, including without limitation the implied warranties of merchantability or fitness for a particular purpose, are excluded. In no event shall Sony Ericsson or its licensors be liable for incidental or consequential damages of any nature, including but not limited to lost profits or commercial loss, arising out of the use of the information in this document.

Table of contents

Document history ... 2
Sony Ericsson Developer World ... 2

1. Before you start ... 4
1.1 About this document ... 4
1.2 Typography ... 4

2. Introduction ... 5

3. SmartExtensionAPI ... 6

4. SmartExtensionUtils ... 7

5. Common components ... 8
5.1 AndroidManifest.xml ... 8
5.1.1. API version ... 8
5.1.2. Permissions ... 9
5.1.3. Service ... 9
5.1.4. Broadcast receiver ... 9
5.1.5. Activity ... 10
5.2 ExtensionReceiver.java ... 10
5.3 SampleExtensionService.java ... 10
5.4 SampleRegistrationInformation.java ... 11
5.5 SamplePreferenceActivity.java ... 12

6. Sample Control Extension ... 13
6.1 SampleControlSmartWatch.java and SampleControlSmartWirelessHeadsetPro.java ... 13
6.2 SampleControlSmartWatch.java ... 14
6.3 SampleControlSmartWirelessHeadsetPro.java ... 16
6.4 SampleExtensionService.java ... 19
6.5 SampleRegistrationInformation.java ... 20
6.6 Smart Extension emulator ... 21

7. Sample Widget Extension ... 23
7.1 SampleWidget.java ... 23
7.2 SmartWatchSampleWidgetImage ... 25
7.3 smart_watch_sample_widget.xml ... 25
7.4 SampleExtensionService.java ... 26
7.5 SampleRegistrationInformation.java ... 27
7.6 Smart Extension emulator ... 27

8. Sample Notification Extension ... 29
8.1 SampleExtensionService.java ... 29
8.2 SampleRegistrationInformation.java ... 32
8.3 SamplePreferenceActivity.java ... 34
8.4 SmartWatch UI ... 35
8.5 Smart Extension emulator ... 36

9. Sample Sensor Extension ... 37
9.1 SampleSensorControl.java ... 37
9.2 SampleExtensionService.java ... 39
9.3 SampleRegistrationInformation.java ... 39
Trademarks and acknowledgements ... 41

1. Before you start

1.1 About this document

This document describes how to develop Smart Extensions for Smart Extras™ from Sony Ericsson supporting the Accessory Extension Framework.

For more information on this, please see the *Smart Extension API specification* document available at http://developer.sonyericsson.com

To benefit properly from this document you should be familiar with Java programming in the Android environment.

1.2 Typography

Code is written in Courier font:

```
<action android:name="com.sonyericsson.extras.liveware" />
```

References in text to applications and their components are written in italics:
SampleControlSmartWatch.java, *AndroidManifest.xml*

References to Intents defined in the *Smart Extension API documentation* are written in upper case:
EXTENSION_REGISTER_REQUEST_INTENT

2. Introduction

This document is a walkthrough of the sample Smart Extension Android applications for the Smart Extras™. The sample projects are built upon two Android library projects:

- *SmartExtensionAPI* defines the Smart Extension API
- *SmartExtensionUtils* provides a set of utility classes that can be useful when developing Smart Extensions.

You can import the projects into Eclipse or any other build environment. The sample projects are intended to be used as reference resources.

To try out the sample extensions without accessory hardware, you can use the *Smart Extension emulator*.

You can download these applications and its documentation from Sony Ericsson Developer World at http://developer.sonyericsson.com

LiveWare™ manager can be downloaded from Android market, https://market.android.com/details?id=com.sonyericsson.extras.liveware

The architecture used for Smart Extension is described in detail in *Smart Extension API Specification* document.

3. SmartExtensionAPI

The *SmartExtensionAPI* is an Android library project that contains the Smart Extension API. This API is the base for all Smart Extensions.

To create the library project, select **File -> New -> Project… -> Android -> Android Project -> Create** from the existing source and select the *SmartExtensionAPI* folder.

You need to add the *SmartExtensionAPI* to your project as an Android Library project, see Figure 1.

You can also copy the desired files to your project.

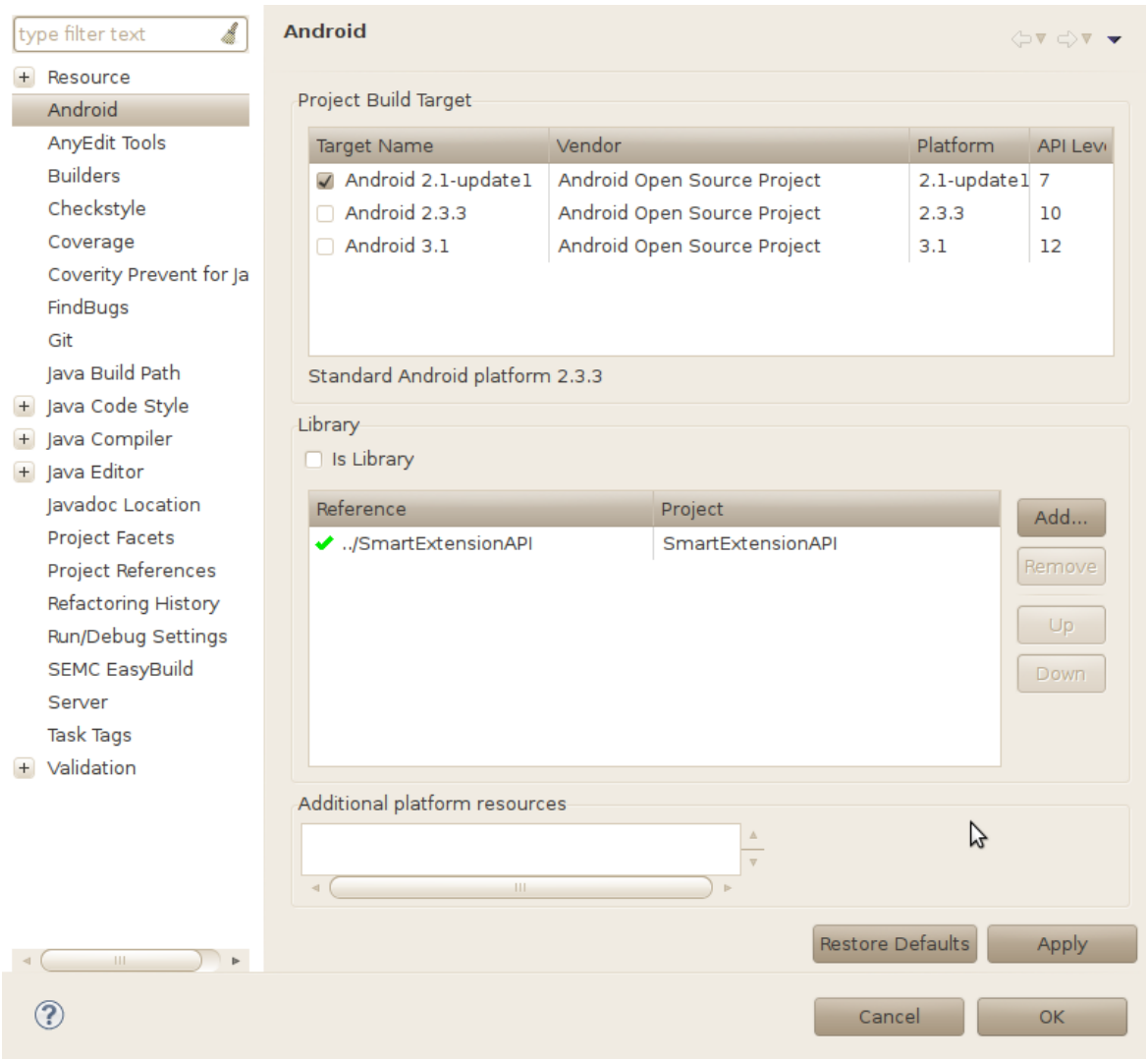

Figure 1 SmartExtensionAPI added as library project in project properties of your extension project

4. SmartExtensionUtils

The *SmartExtensionUtils* is an Android Library project that contains a set of helper classes. They can be useful when you get started with developing Smart Extensions. All sample extensions described in this document make full use of the helper classes in the *SmartExtensionUtils*.

You need to add the *SmartExtensionUtils* to your project as an Android Library project, in the same way as *SmartExtensionAPI*.

The *SmartExtensionUtils* include the Smart Extension API from *SmartExtensionAPI* so you only need to add one of them to your project.

5. Common components

All sample extensions share some common components needed for developing a Smart Extension.
A typical Eclipse project setup can look like this:

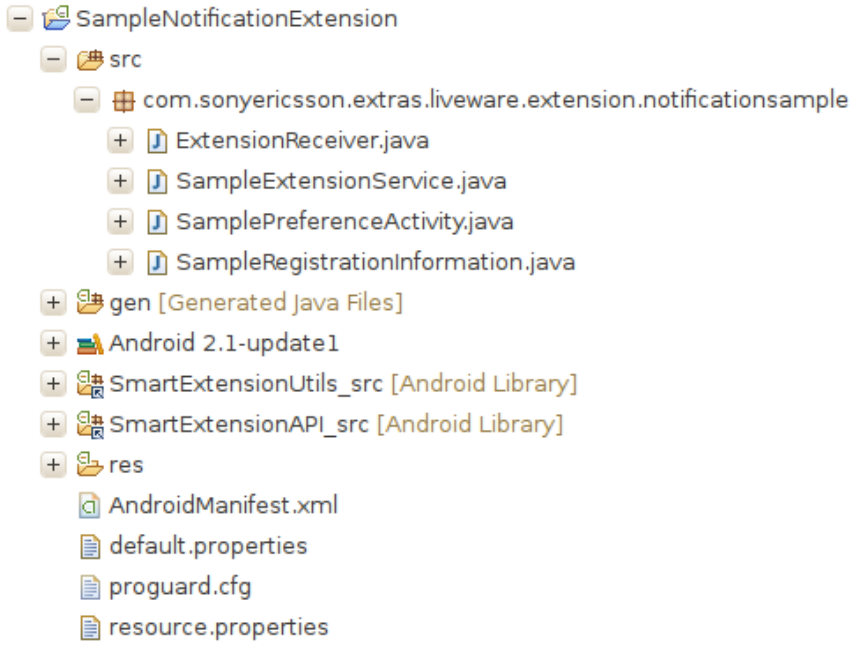

Figure 2 File structure of SampleControlExtension

These java classes are needed to develop a Smart Extension:

- The *ExtensionReceiver.java* broadcast receiver starts the *SampleExtensionsService* when broadcast intents from the *LiveWare™ manager* and the Accessory Host Applications are received.
- *SampleExtensionService.java* handles registration and performs extension specific tasks.
- The *SampleRegistrationInformation.java* contains information about the extension that is needed when registering with *LiveWare™ manager.*

5.1 AndroidManifest.xml

5.1.1. API version

The Accessory Extension Framework is dependent on Android API level 7 (corresponding to Android version 2.1).

```
<uses-sdk android:minSdkVersion="7" />
```

5.1.2. Permissions

All Smart Extensions must declare that it uses the permission specified in the manifest for *LiveWare™ manager* in order to get access to the registration and notification content providers.

```
<uses-permission
    android:name="com.sonyericsson.extras.liveware.aef.EXTENSION_PERMISSION"/>
```

5.1.3. Service

A Smart Extension needs to have at least one service declared in the manifest:

```
<service android:name=". SampleExtensionsService" />
```

5.1.4. Broadcast receiver

You need to register a receiver and define an intent filter for the components that will receive the Intents sent by the accessory host application and the *LiveWare™ manager*. All extensions must have the intent filter *EXTENSION_REGISTER_REQUEST* in their manifest. Depending on what kind of extension you want to implement you need to add more intents to the intent filter.

```
<receiver
      android:name=".ExtensionReceiver">
    <intent-filter>
      <!-- Generic extension intents. -->
      <action
      android:name="com.sonyericsson.extras.liveware.aef.registration.EXTENSION_REGISTER_REQUEST"/>
      <action
      android:name="com.sonyericsson.extras.liveware.aef.registration.ACCESSORY_CONNECTION"/>
      <action android:name="android.intent.action.LOCALE_CHANGED"/>
      <!-- Notification intents -->
      <action android:name="com.sonyericsson.extras.liveware.aef.notification.VIEW_EVENT_DETAIL"/>
      <action android:name="com.sonyericsson.extras.liveware.aef.notification.REFRESH_REQUEST"/>
      <!-- Widget intents -->
      <action android:name="com.sonyericsson.extras.aef.widget.START_REFRESH_IMAGE_REQUEST"/>
      <action android:name="com.sonyericsson.extras.aef.widget.STOP_REFRESH_IMAGE_REQUEST"/>
      <action android:name="com.sonyericsson.extras.aef.widget.ONTOUCH"/>
      <action
      android:name="com.sonyericsson.extras.liveware.extension.util.widget.scheduled.refresh"/>
      <!-- Control intents -->
      <action android:name="com.sonyericsson.extras.aef.control.START"/>
      <action android:name="com.sonyericsson.extras.aef.control.STOP"/>
      <action android:name="com.sonyericsson.extras.aef.control.PAUSE"/>
      <action android:name="com.sonyericsson.extras.aef.control.RESUME"/>
      <action android:name="com.sonyericsson.extras.aef.control.ERROR"/>
      <action android:name="com.sonyericsson.extras.aef.control.KEY_EVENT"/>
      <action android:name="com.sonyericsson.extras.aef.control.TOUCH_EVENT"/>
      <action android:name="com.sonyericsson.extras.aef.control.SWIPE_EVENT"/>
    </intent-filter>
</receiver>
```

5.1.5. Activity

If the extension has any setting, a preference activity is specified in the manifest. The preference activity is registered in the *LiveWare™ manager* registration content provider and is accessible to the end user through the host application.

```
<activity android:name=".SamplePreferenceActivity"
                  android:label="@string/preference_activity_title">
     <intent-filter>
           <action android:name="android.intent.action.MAIN" />
     </intent-filter>
</activity>
```

5.2 ExtensionReceiver.java

The *ExtensionReceiver.java* file extends the *android.content.BroadcastReceiver* service.

The extensions receiver starts the sample extension service when any of the intents listed in the manifest intent filter is received. The extension receiver is implemented as a static broadcast receiver to allow the service to be shut down when it has nothing do. A dynamic broadcast receiver registered using `Context.registerReceiver` requires a service that is always running.

```
@Override
public void onReceive(final Context context, final Intent intent) {
    Log.d(SampleExtensionService.TAG, "onReceive: " + intent.getAction());
    intent.setClass(context, SampleExtensionsService.class);
    context.startService(intent);
}
```

5.3 SampleExtensionService.java

The *SampleExtensionService.java* file extends *com.sonyericsson.extras.liveware.extension.util.ExtensionService.*

ExtensionService is an abstract class that is included in *SmartExtensionUtils*. It handles the extension registration and the API registration for the extensions that extends the class. It also provides a number of methods that can be overridden in order to handle different events related to the extension.

In your service, which extends *ExtensionService*, you need to implement `getRegistrationInformation()` that returns a *RegistrationInformation* object that provides the information needed to register the extension properly.

```
@Override
protected RegistrationInformation getRegistrationInformation() {
    return new SampleRegistrationInformation(this);
}
```

The registration is a time consuming operation and is handled in an *AsyncTask*. When registration is done the `onRegisterResult(boolean success)` is called, allowing the extension to override it and take action.

To preserve the phone battery and to reduce the process load on the system it, is important to stop the extension service when it is not needed. The *ExtensionService* helps the extensions achieve this. Normally an extension service will only be running when a widget or control is visible on a Smart Accessory.

Some extensions need to be running even though they are not visible on the screen. A common use case is that a notification extension observes a content provider and replicates events to the notification content provider. Observing a content provider requires a running service.

To support this case, you can override the *keepRunningWhenConnected* method and have the extension service running as long as there is at least one Smart Accessory connected to the phone. If no Smart Accessories are connected to the phone, the extensions service will be stopped regardless of the value returned from *keepRunningWhenConnected*. The service will be started again when a Smart Accessory is re-connected. Since the service may have been shut down for a while, it is important to check its sources to see if it has missed any events that should be replicated to the notification content provider.

```
@Override
protected boolean keepRunningWhenConnected() {
    return false;
}
```

To ensure proper handling of the extension service lifecycle, it is important to invoke the base class method in the implemented method when overriding methods in *ExtensionService.java*.

In the example below from Sample Notification Extension `onStartCommand()` in *SampleExtensionService* invokes *onStartCommand* in *ExtensionService* before it does its own intent handling.

The same pattern should be followed if any of these are overridden: `onCreate()`, `onDestroy()` or `onRegisterRequest()`.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    int retVal = super.onStartCommand(intent, flags, startId);
    if (intent != null) {
        // Handling of extension specific intents goes here
    }
    return retVal;
}
```

5.4 SampleRegistrationInformation.java

The *SampleRegistrationInformation.java* file extends *com.sonyericsson.extras.liveware.aef.util.registration.RegistrationInformation service.*

RegistrationInformation is an abstract class that is included in *SmartExtensionUtils*. It should be extended and has a number of abstract methods that should to be overridden in all types of extensions:

```
public ContentValues getExtensionRegistrationConfiguration()
```

Here the extension returns a *ContentValues* object with the information needed to register in the extension table in the *LiveWare™ manager* registration content provider. Depending on the extension type, it needs to register in other tables as well.

```
public int getRequiredControlApiVersion()
```

Returns the Control API version that the extension use (the current version is 1). If the extension is not using the Control API, 0 shall be returned.

```
public int getRequiredNotificationApiVersion()
```

Returns the Notification API version that the extension use (the current version is 1). If the extension is not using the Notification API, 0 shall be returned.

```
public int getRequiredSensorApiVersion()
```

Returns the Sensor API version that the extension use (the current version is 1). If the extension is not using the Sensor API, 0 shall be returned.

```
public int getRequiredWidgetApiVersion()
```

Returns the Widget API version that the extension use (the current version is 1). If the extension is not using the Widget API, 0 shall be returned.

5.5 SamplePreferenceActivity.java

The *SamplePreferenceActivity.java* file extends *android.preference.PreferenceActivity.*

This class is where your extension preferences go. The extensions preferences can be reached from *LiveWare™ manager* or the Smart Extension emulator. This class is responsible for creating dialogs, listening for clicks and sends intents to the service which will handle the clicks.

The class name should be registered in the *LiveWare™ manager*. This is done using the method *getExtensionRegistrationConfiguration* in the class *SampleRegistrationInformation*:

```
values.put(Registration.ExtensionColumns.CONFIGURATION_ACTIVITY,
                SamplePreferenceActivity.class.getName());
```

If the extension does not have any preferences, *SamplePreferenceActivity.java* should not be implemented and the class name should not be registered in the *LiveWare™ manager*.

6. Sample Control Extension

The *SampleControlExtension* is a sample project showing an example of how to use the *SmartExtensionUtils* to develop a control extension. A control extension can take full control over the Smart Accessory display.

The sample control extension starts an animation on the accessory. This extension supports both SmartWatch and Smart Wireless Headset pro.

When the control is started on SmartWatch, it shows an animation and a text. Tapping the screen on SmartWatch stops the animation and exits the control.

When the control is started on Smart Wireless Headset pro, it shows a scrolling text that can be paused and resumed with play/pause key. The text can also be moved step by step with the previous and next keys. You can change the animation speed with the volume keys on the accessory.

6.1 SampleControlSmartWatch.java and SampleControlSmartWirelessHeadsetPro.java

The sample control extension has two classes which extends *ControlExtension*:

- *SampleControlSmartWirelessHeadsetPro* handles the control of the Smart Wireless Headset pro accessory.
- *SampleControlSmartWatch* handles the control of the SmartWatch accessory.

These classes are very similar. In this section we describe the common functionality, but there are minor differences. Functionality that is accessory specific is handled in separate chapters. For simplicity, *SampleControlSmartWirelessHeadsetPro* and *SampleControlSmartWatch* are referred to as “Sample Control” in this chapter.

The *ControlExtension* classes are responsible for rendering images and handle user interaction on the accessories. The *SmartExtensionUtils* creates a *ControlExtension* object when an extension is given control over the display. The `onStart()` method is called when the control is started. The Sample Control is not visible to the user until `onResume()` has been called. You can use `onStart()` to set up things that should be running even though the control is not visible at the moment. The Sample Control does not have any such requirements and nothing is done in `onStart()`.

```
@Override
public void onStart() {
    // Nothing to do. Animation is handled in onResume.
}
```

The method `onResume()` is called when the control is visible on the display. The Sample Control displays an animation as soon as the control is visible on the accessory screen. The animation updating and scheduling takes place in the inner class *Animation* which we will not describe in detail here. The accessory

does not keep any images when paused, so the extension is expected to send a new full screen image when resumed.

```
@Override
public void onResume() {
    mIsVisible = true;
    Log.d(SampleControlService.LOG_TAG, "Starting animation");

    // Animation not showing. Show animation.
    mIsShowingAnimation = true;
    mAnimation = new Animation();
    mAnimation.run();
}
```

The method `onPause()` is called when the control is no longer visible on the display. This happens if the screen timeout elapses, if the user turns off the display or if a higher prioritised control takes over the screen (for example the call extension if a call is incoming). If the user leaves the control, `onPause()` is called followed by `onStop()` and finally `onDestroy()`

```
@Override
public void onPause() {
    Log.d(SampleControlService.LOG_TAG, "Stopping animation");
    mIsVisible = false;

    if (mIsShowingAnimation) {
         stopAnimation();
    }
}
```

The Sample Control doesn't do anything in `onStop()`. For controls that set up things in `onStart()`, this is the place to tear them down.

```
@Override
public void onStop() {
    // Nothing to do. Animation is handled in onPause.
}
```

6.2 SampleControlSmartWatch.java

The *SampleControlSmartWatch.java* file extends *com.sonyericsson.extras.liveware.extension.util.control.ControlExtension.*

Touch events are received in the method `onTouch()`. The SmartWatch supports touch events and *SampleControlSmartWatch* asks to be shut down by calling `stopRequest()` when a touch event is received. The `stopRequest()` will result in a `onStop()` call to the sample control.

```
@Override
public void onTouch(final ControlTouchEvent event) {
    Log.d(SampleControlService.TAG, "onTouch() " + event.getAction());
    if (event.getAction() == Control.Intents.TOUCH_ACTION_RELEASE) {
        if (mIsShowingAnimation) {
            Log.d(SampleControlService.TAG, "Stopping animation");
```

```
            // Stop the animation
            stopAnimation();
        }
    }
}

public void stopAnimation() {
    // Stop animation on accessory
    if (mAnimation != null) {
        mHandler.removeCallbacks(mAnimation);
        mAnimation = null;
    }
    mIsShowingAnimation = false;

    // If the control is visible then stop it
    if (mIsVisible) {
        stopRequest();
    }
}
```

Information to the Smart Accessory display can be sent in different ways. One possibility is to prepare a bitmap and use `showBitmap()` to send the bitmap to the accessory. The example below shows how to create a canvas based on the bitmap and how to draw a standard Android XML layout on the canvas.

```
// Create background bitmap for animation.
mBackground = Bitmap.createBitmap(width, height, BITMAP_CONFIG);
// Set default density to avoid scaling.
background.setDensity(DisplayMetrics.DENSITY_DEFAULT);

LinearLayout root = new LinearLayout(mContext);
root.setLayoutParams(new LayoutParams(width, height));

LinearLayout sampleLayout = (LinearLayout)LinearLayout.inflate(mContext,
        R.layout.sample_control, root);
((TextView)sampleLayout.findViewById(R.id.sample_control_text))
        .setText(R.string.sample_control_title);

sampleLayout.measure(width, height);
sampleLayout.layout(0, 0, sampleLayout.getMeasuredWidth(),
        sampleLayout.getMeasuredHeight());

// Draw on canvas
Canvas canvas = new Canvas(mBackground);
sampleLayout.draw(canvas);

// Send bitmap to accessory
showBitmap(mBackground);
```

The animation in *SampleControlSmartWatch* is sent to the Smart Accessory as partial screen updates to reduce the amount of data that has to be sent over the Bluetooth link.

First the *SampleControlSmartWatch* creates a bitmap with the size of the screen area to be updated.

The second step is to draw background on the bitmap and finally the current animation state is drawn. The partial display update is then sent to the accessory using `showBitmap()`. The ANIMATION_X_POS and

ANIMTATION_Y_POS parameters in combination with the bitmap width and height determines what area of the display to update

```
// Create a bitmap for the part of the screen that needs updating.
Bitmap bitmap = Bitmap.createBitmap(animation.getWidth(),
animation.getHeight(),
                    BITMAP_CONFIG);
bitmap.setDensity(DisplayMetrics.DENSITY_DEFAULT);
Canvas canvas = new Canvas(bitmap);
Paint paint = new Paint();
Rect src = new Rect(ANIMATION_X_POS, ANIMATION_Y_POS, ANIMATION_X_POS
            + animation.getWidth(), ANIMATION_Y_POS + animation.getHeight());
Rect dst = new Rect(0, 0, animation.getWidth(), animation.getHeight());

// Add first the background and then the animation.
canvas.drawBitmap(mBackground, src, dst, paint);
canvas.drawBitmap(animation, 0, 0, paint);

showBitmap(bitmap, ANIMATION_X_POS, ANIMATION_Y_POS);
```

6.3 SampleControlSmartWirelessHeadsetPro.java

The *SampleControlSmartWirelessHeadsetPro.java* file extends *com.sonyericsson.extras.liveware.extension.util.control.ControlExtension*

Key events are received in the method `onKey()`. Smart Wireless Headset pro has a number of keys that the user can interact with. *SampleControlSmartWirelessHeadsetPro* reacts on key release events. The scrolling text animation can be paused and resumed with the play/pause key.

The text can also be moved step by step over the display with the previous and next keys. The size of each step can be changed using the volume keys on the accessory. When the back key is pressed, the *SampleControlSmartWirelessHeadsetPro* asks to be shut down by calling `stopRequest()`. The `stopRequest()` will result in a `onStop()` call to the sample control.

```
@Override
public void onKey(int action, int keyCode, long timeStamp) {
    if (action != Control.Intents.KEY_ACTION_RELEASE) {
        return;
    }

    switch (keyCode) {
        case Control.KeyCodes.KEYCODE_PREVIOUS:
            // Move animation text left
            pauseAnimation();
            mAnimatedTextXPos -= mAnimatedTextXDelta;
            if (mAnimatedTextXPos < -mAnimatedTextBounds.width()) {
                mAnimatedTextXPos = -mAnimatedTextBounds.width();
            }
            break;
        case Control.KeyCodes.KEYCODE_NEXT:
            // Move animation text right
            pauseAnimation();
            mAnimatedTextXPos += mAnimatedTextXDelta;
            if (mAnimatedTextXPos > mWidth) {
                mAnimatedTextXPos = mWidth;
```

```
            }
            break;
        case Control.KeyCodes.KEYCODE_VOLUME_UP:
            // Increase animation speed
            mAnimatedTextXDelta += ANIMATION_DELTA_X_START_VALUE;
            if (mAnimatedTextXDelta > ANIMATION_DELTA_X_MAX_VALUE) {
                mAnimatedTextXDelta = ANIMATION_DELTA_X_MAX_VALUE;
            }
            break;
        case Control.KeyCodes.KEYCODE_VOLUME_DOWN:
            // Decrease animation speed
            mAnimatedTextXDelta -= ANIMATION_DELTA_X_START_VALUE;
            if (mAnimatedTextXDelta < ANIMATION_DELTA_X_START_VALUE) {
                mAnimatedTextXDelta = ANIMATION_DELTA_X_START_VALUE;
            }
            break;
        case Control.KeyCodes.KEYCODE_PLAY:
            // Play/pause animation
            startOrPauseAnimation();
            break;
        case Control.KeyCodes.KEYCODE_BACK:
            // Stop animation and quit sample control
            stopAnimation();
            break;
        default:
            // no action
            break;
    }
    updateText();
}
/**
 * Start a paused animation or pause an ongoing animation.
 */
private void startOrPauseAnimation() {
    // Animation not showing. Show animation.
    if (mIsShowingAnimation) {
        pauseAnimation();
    } else {
        startAnimation();
    }
}

/**
 * Start animation
 */
private void startAnimation() {
    if (mAnimation != null) {
        mHandler.removeCallbacks(mAnimation);
        mAnimation = null;
    }
    mIsShowingAnimation = true;
    mAnimation = new Animation();
    mAnimation.run();
}

/**
 * Pause animation on control.
 */
public void pauseAnimation() {
```

```
    // Stop animation on accessory
    if (mAnimation != null) {
        mHandler.removeCallbacks(mAnimation);
        mAnimation = null;
    }
    mIsShowingAnimation = false;
}

/**
 * Stop showing animation and stop control.
 */
public void stopAnimation() {
    pauseAnimation();
    // If the control is visible then stop it
    if (mIsVisible) {
        stopRequest();
    }
}
```

Information to the Smart Accessory display can be sent in different ways. One possibility is to prepare a bitmap and use `showBitmap()` to send the bitmap to the accessory. The example below shows how to create a canvas based on the bitmap and how to draw a standard Android XML layout on the canvas. The position of the text is controlled by the *mAnimatedTextXPos* parameter.

In this case, an image covering the whole display is sent to the accessory in each update. If possible, a partial update should be used instead to reduce the amount of data that is sent over the Bluetooth link.

```
/**
 * Update the text on the accessory.
 */
private void updateText() {

    Bitmap bitmap = Bitmap.createBitmap(mWidth, mHeight, BITMAP_CONFIG);
    // Set the density to default to avoid scaling.
    bitmap.setDensity(DisplayMetrics.DENSITY_DEFAULT);

    Canvas canvas = new Canvas(bitmap);
    // Black background
    canvas.drawColor(Color.BLACK);

    // Draw text
    canvas.drawText(mAnimatedText, 0, mAnimatedText.length(),
mAnimatedTextXPos,
            SmartWirelessHeadsetProUtil.CONFIRM_TEXT_Y, mTextPaint);

    showBitmap(bitmap, 0, 0);
}
```

6.4 SampleExtensionService.java

The *SampleExtensionService.java* file extends *com.sonyericsson.extras.liveware.extension.util.ExtensionService.*

This section describes the part of *SampleExtensionService* that is specific for a control extension.

The method *createControlExtension* is called when a control should be visible on an accessory display. The *SampleExtensionService* returns a *SampleControlSmartWatch* or a *SampleControlSmartWirelessHeadsetPro* control object depending on the display resolution of the accessory. The *HostApplicationInfo* class is used to find the display resolution of the accessory. The *SmartExtensionUtils* then calls the methods `onStart()` and `onResume()` on the returned control object.

```
@Override
public ControlExtension createControlExtension(String hostAppPackageName) {
    final int controlSWWidth =
SampleControlSmartWatch.getSupportedControlWidth(this);
    final int controlSWHeight =
SampleControlSmartWatch.getSupportedControlHeight(this);
    final int controlSWHPWidth =
SampleControlSmartWirelessHeadsetPro.getSupportedControlWidth(this);
    final int controlSWHPHeight =
SampleControlSmartWirelessHeadsetPro.getSupportedControlHeight(this);

    for (DeviceInfo device : RegistrationAdapter.getHostApplication(this,
         hostAppPackageName).getDevices()) {
        for (DisplayInfo display : device.getDisplays()) {
            if (display.sizeEquals(controlSWWidth, controlSWHeight)) {
                return new SampleControlSmartWatch(hostAppPackageName, this,
                                                        new Handler());
            } else if (display.sizeEquals(controlSWHPWidth, controlSWHPHeight))
{
                return new SampleControlSmartWirelessHeadsetPro(
                       hostAppPackageName, this, new Handler());
            }
        }
    }
};
```

When the control is visible, the *SampleExtensionService* needs to be running in order to maintain the state of the animation. The *ExtensionService* will not stop as long as there are visible controls.

When the control is no longer visible, the *SampleControl* does not need to run and thus it can return false in `keepRunningWhenConnected()`.

```
@Override
protected boolean keepRunningWhenConnected() {
    return false;
}
```

6.5 SampleRegistrationInformation.java

The *SampleRegistrationInformation.java* file extends *com.sonyericsson.extras.liveware.extension.util.registration.RegistrationInformation.*

This section describes that part of *SampleRegistrationInformation* that is specific for a control extension.

To be displayed as a control on a Smart Accessory, the Sample Control must register with the accessory host application in the registration content provider. The *AsyncTask* in *ExtensionService* use the *RegistrationInformation* to register with all host applications that has a display supported by the extension. The `getRequiredControlApiVersion()` is used to check what version of the Control API that the extension supports (if any) and `isDisplaySizeSupported()` is used to ask the extension whether the display is supported. The sample control extension supports display sizes of both SmartWatch and Smart Wireless Headset pro.

```
@Override
public int getRequiredControlApiVersion() {
    return 1;
}

@Override
public boolean isDisplaySizeSupported(int width, int height) {
    return (
        (width == SampleControlSmartWatch.getSupportedControlWidth(mContext)
         && height == SampleControlSmartWatch
         .getSupportedControlHeight(mContext))
         || (width == SampleControlSmartWirelessHeadsetPro
         .getSupportedControlWidth(mContext)
         && height == SampleControlSmartWirelessHeadsetPro
         .getSupportedControlHeight(mContext)));
}
```

6.6 Smart Extension emulator

The Smart Extension emulator emulates the SmartWatch when *SampleControlExtension* runs on the phone.

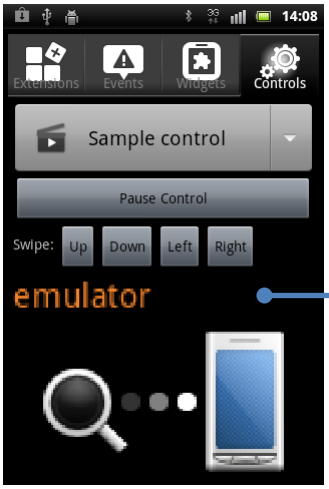

Figure 3 Controls: Running in Smart Extension emulator

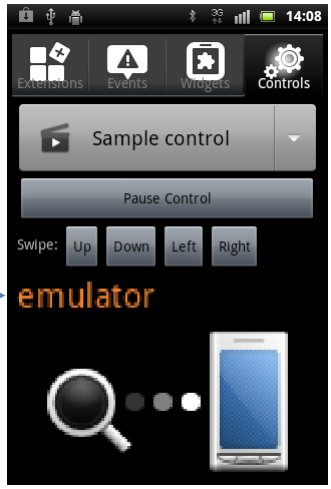

Figure 4 Controls: Paused

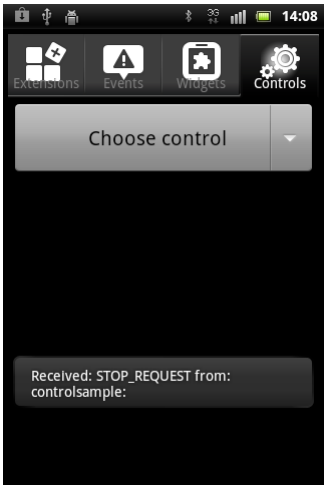

Figure 5 Controls: Stopped

Figure 6 Select Extension preferences

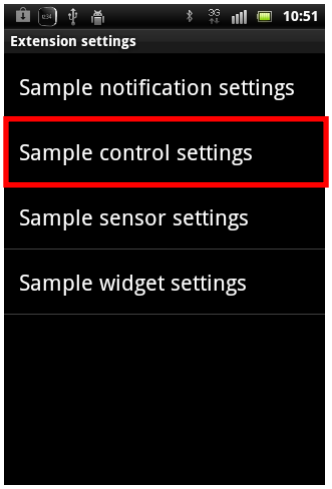

Figure 7 Select extension

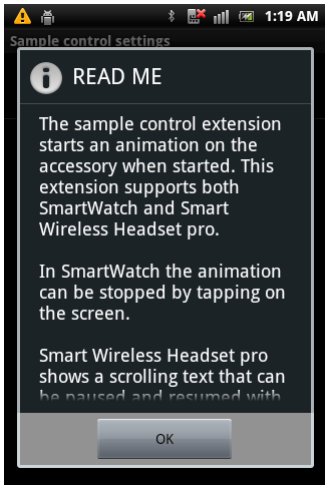

Figure 8 SamplePreferenceActivity

Figure 9 Select Emulator preferences

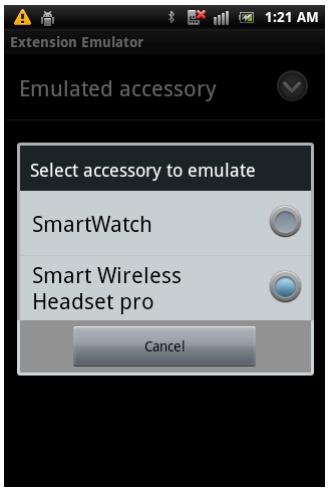

Figure 10 Switch to Smart Wireless Headset pro emulator

Figure 11 Controls: Running in Smart Wireless Headset pro emulator

7. Sample Widget Extension

The *SampleWidgetExtension* is a sample project that shows an example of how to use the *SmartExtensionUtils* to develop a Widget Extension. A Widget Extension is an Android application which communicates with *LiveWare™ manager* and displays images that give the user a quick preview of the current state on compatible accessories.

This sample extension shows a digital clock which can be toggled to show either 24H or 12H when clicked.

7.1 SampleWidget.java

The *SampleWidget.java file* extends *com.sonyericsson.extras.liveware.extension.util.widget.WidgetExtension.*

This class is responsible for rendering and updating the widget and then sending it to the Smart Accessory. The *SmartExtensionUtils* creates the *WidgetExtension* object when the accessory wants the extension to start updating the widget. The method `onStartRefresh()` is called after the object has been created to request that the widget start refreshing its images.

Note: The widget does not know whether it is visible or not. The method `onStartRefresh()` only indicates that the widget soon might become visible and that the accessory needs an updated image to be able to display the information to the user immediately.

The *SampleWidget* uses the method `scheduleRepeatingRefresh()` to update the time every 10 seconds on the widget image.

The first parameter determines when the first update is done. Setting that time to the current time will immediately cause a refresh of the display. The scheduled refresh relies on the *AlarmManager* and will wake up the phone if it's in sleep mode. The pending intent from the *AlarmManager* will result in an *onScheduledRefresh* call in which the *SampleWidget* generates an updated widget image and sends it to the accessory.

```
@Override
public void onStartRefresh() {
    Log.d(SampleWidgetService.LOG_TAG, "startRefresh");
    // Update now and every 10th second
    cancelScheduledRefresh(SampleWidgetService.EXTENSION_KEY);
    scheduleRepeatingRefresh(System.currentTimeMillis(), UPDATE_INTERVAL,
            SampleWidgetService.EXTENSION_KEY);
}

@Override
public void onScheduledRefresh() {
    Log.d(SampleWidgetService.LOG_TAG, "scheduledRefresh()");
    updateWidget();
}
```

When the accessory no longer needs updated widget images, the method `onStopRefresh()` is called and the widget object will be recycled by the garbage collector. When stopped, the *SampleWidget* no longer needs to update the time and cancels the scheduled refresh.

```
@Override
public void onStopRefresh() {
    Log.d(SampleWidgetService.LOG_TAG, "stopRefesh");

    // Cancel pending clock updates
    cancelScheduledRefresh(SampleWidgetService.EXTENSION_KEY);
}
```

Touch events are received in the method *onTouch*. First the sample widget checks that the touch event is in the screen centre by checking that the x and y coordinates are inside the rectangle defined in *SmartWatchConst.ACTIVE_WIDGET_TOUCH_AREA*. This is to minimize the risk that failed swipe events are interpreted as touch events. If the touch event is in the screen centre and is a short tap, the sample widget changes the time format and updates the widget.

```
@Override
public void onTouch(final int type, final int x, final int y) {
    Log.d(SampleWidgetService.LOG_TAG, "onTouch() " + type);
    if (!SmartWatchConst.ACTIVE_WIDGET_TOUCH_AREA.contains(x, y)) {
        Log.d(SampleExtensionService.LOG_TAG, "Ignoring touch outside active
area x: " + x
              + " y: " + y);
        return;
    }
    if (type == Widget.Intents.EVENT_TYPE_SHORT_TAP) {
        // Change clock mode on short tap.
        setClockMode24h(!isClockMode24h());
        // Update clock widget now
        updateWidget();
    }
}
```

The widget is updated by sending a complete new bitmap to the Smart Accessory. The `updateWidget()` uses *SmartWatchSampleWidgetImage* to build the image and `showBitmap()` to send the bitmap to the accessory.

```
private void updateWidget() {
    Log.d(SampleExtensionService.LOG_TAG, "updateWidget");
    // Get time
    String time = null;
    if (isClockMode24h()) {
        time = TIME_FORMAT_24_H.format(new Date());
    } else {
        time = TIME_FORMAT_AM_PM.format(new Date());
    }

    showBitmap(new SmartWatchSampleWidgetImage(mContext, time).getBitmap());
}
```

7.2 SmartWatchSampleWidgetImage

SmartWatchSampleWidgetImages extends the *SmartWatchWidgetImage* class which includes functionality to give all widgets on SmartWatch the same look and feel by drawing the frame around the widget, an optional widget icon and an optional badge. The *SmartWatchSampleWidgetImage* is only responsible for drawing the content inside the frame.

When created, *SmartWatchSampleWidgetImage* sets the layout resource id for the content that should be shown inside the widget frame and save the time string to be shown in the widget.

```
public SmartWatchSampleWidgetImage(final Context context, final String time) {
    super(context);
    setInnerLayoutResourceId(R.layout.smart_watch_sample_widget);
    mTime = time;
}
```

When `getBitmap()` is called on the super class, it calls `applyInnerLayout()` to allow *SmartWatchSampleWidgetImage* to update the contents of the layout. Here the sample widget finds the text view and sets the text to the current time string.

```
@Override
protected void applyInnerLayout(LinearLayout innerLayout) {
    // Set time
    ((TextView)innerLayout.findViewById(
           R.id.smart_watch_sample_widget_time)).setText(mTime);
}
```

7.3 smart_watch_sample_widget.xml

The *smart_watch_sample_widget.xml* is an Android layout describing the content of the frame in the sample widget. The width of the layout is set to *smart_watch_widget_width_inner* and the height is set to *smart_watch_widget_height_inner,* these are constants defined in the *SmartExtensionUtils*.

The text is rendered using *AefTextView* which is a custom text view defined in the *SmartExtensionUtils*. The only difference compared to normal text view is that a text that does not fit the view is faded.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="@dimen/smart_watch_widget_width_inner"
    android:layout_height="@dimen/smart_watch_widget_height_inner">

    <com.sonyericsson.extras.liveware.extension.util.AefTextView
        android:id="@+id/smart_watch_sample_widget_time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:textStyle="bold"
        android:textSize="20px"
        android:textColor="@color/smart_watch_text_color_orange" />

    <com.sonyericsson.extras.liveware.extension.util.AefTextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="3px"
        android:text="@string/extension_info"
        android:textSize="12px"
        android:textColor="@color/smart_watch_text_color_grey" />

</RelativeLayout>
```

When you use an .xml file to specify the layout of an accessory, consider this:

- All sizes should be expressed in px. This to avoid scaling based on the phone density.
- If an *ImageView* is used to show an image, we recommend that you specify *layout_width* and *layout_height* in px to avoid scaling based on the phone density.

7.4 SampleExtensionService.java

The *SampleExtensionService.java* file extends *com.sonyericsson.extras.liveware.extension.util.ExtensionService.*

This section describes the part of *SampleExtensionService* that is specific for a Widget Extension.

The method *createWidgetExtension* is called when an accessory wants an extension to refresh the widget image. The *SampleExtensionService* returns a *SampleWidget* and the *SmartExtensionUtils* then calls the method `onStartRefresh()` on the returned widget object.

```
@Override
public WidgetExtension createWidgetExtension(String hostAppPackageName) {
    return new SampleWidget(hostAppPackageName, this);
}
```

When the widget is refreshing its images, the *SampleWidget SampleExtensionService* needs to be running in order to maintain the state of the widget. When the widget is not refreshing, the *SampleWidget* does not need to run and can it return false in `keepRunningWhenConnected()`.

```
@Override
protected boolean keepRunningWhenConnected() {
    return false;
}
```

7.5 SampleRegistrationInformation.java

The *SampleRegistrationInformation.java* file extends *com.sonyericsson.extras.liveware.extension.util.registration.RegistrationInformation.*

This section describes that part of *SampleRegistrationInformation* that is specific for a Widget Extension.

In order to be displayed as a widget on a Smart Accessory, the sample widget must register with the accessory host application in the registration content provider.

The *AsyncTask* in *ExtensionService* uses the *RegistrationInformation* to register with all host applications that has a widget size supported by the extension. The `getRequriedWidgetApiVersion()` is called to check what version of the Widget API the extension supports and `isWidgetSizeSupported()` is used to ask the extension whether the widget size is supported.

```
@Override
public int getRequiredWidgetApiVersion() {
    return 1;
}

@Override
public boolean isWidgetSizeSupported(final int width, final int height) {
    return (width == SampleWidget.WIDTH && height == SampleWidget.HEIGHT);
}
```

7.6 Smart Extension emulator

The Smart Extension emulator emulates the SmartWatch when *SampleControlExtension* runs on the phone.

As you can see in the Widgets tab, the only thing *Sample Widget Extension* does is to display a clock which toggles between 24H and 12H when clicked.

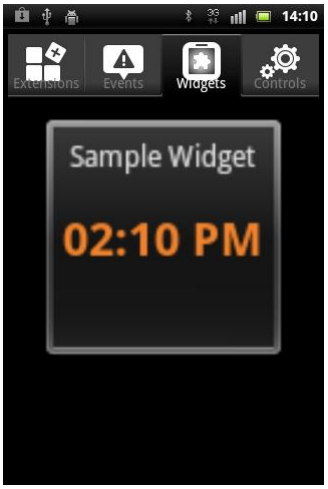

Figure 12 Widgets 12H

Figure 13 Widgets: 24H

Figure 14 Select Extension preferences

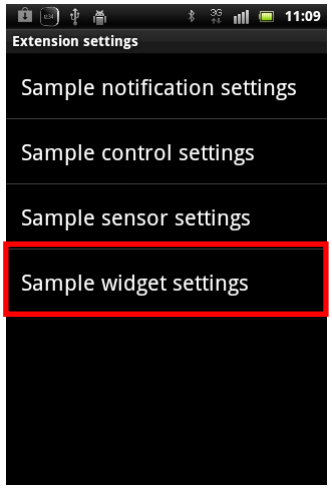

Figure 15 Select extension

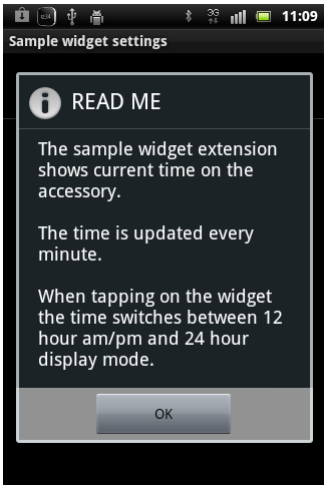

Figure 16 SamplePreferenceActivity

8. Sample Notification Extension

The *Sample Notification Extension* is a sample project that shows an example of how to use the *SmartExtensionUtils* to develop a Notification Extension. A Notification Extension is an Android application which communicates with *LiveWare™ manager* and inserts events in its notification content provider to be displayed on compatible accessories.

This sample extension inserts events in the *LiveWare™ manager* content provider every 10 seconds.

8.1 SampleExtensionService.java

The *SampleExtensionService.java* file extends *com.sonyericsson.extras.liveware.extension.util.ExtensionService.*

When the extension is successfully registered, it checks that it is active.

```
@Override
public void onRegisterResult(boolean result) {
    super.onRegisterResult(result);
    Log.d(LOG_TAG, "onRegisterResult");

    // Start adding data if extension is active in preferences
    if (result) {
        SharedPreferences prefs = PreferenceManager
               .getDefaultSharedPreferences(this);
        boolean isActive = prefs.getBoolean(
                  getString(R.string.preference_key_is_active), false);
        if (isActive) {
             startAddData();
        }
    }
}
```

If the active preference is set, the extensions calls `startAddData()` to schedule a pending INTENT_ACTION_ADD intent from the Alarm Manager every 5 seconds.

```
private void startAddData() {
    AlarmManager am = (AlarmManager)getSystemService(ALARM_SERVICE);
    Intent i = new Intent(this, SampleExtensionService.class);
    i.setAction(INTENT_ACTION_ADD);
    PendingIntent pi = PendingIntent.getService(this, 0, i, 0);
    am.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
SystemClock.elapsedRealtime(),
           INTERVAL, pi);
}
```

The intent INTENT_ACTION_ADD is delivered to the service through the `onStartCommand()`.
The `addData()` is called to insert a new event in the notification content provider.

The `stopSelfCheck()` is called to check if the service should be stopped. It checks whether a control or a widget is visible and if there are any other operations that require the extension service to be running. If not, it will call `stopSelf()` to stop the service. The `stopSelfCheck()` also checks `keepRunningWhenConnected()` and if it returns true the extension service will not be stopped as long as there is at least one accessory connected.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    int retVal = super.onStartCommand(intent, flags, startId);
    if (intent != null) {
        if (INTENT_ACTION_START.equals(intent.getAction())) {
            Log.d(LOG_TAG, "onStart action: INTENT_ACTION_START");
            startAddData();
            stopSelfCheck();
        } else if (INTENT_ACTION_STOP.equals(intent.getAction())) {
            Log.d(LOG_TAG, "onStart action: INTENT_ACTION_STOP");
            stopAddData();
            stopSelfCheck();
        } else if (INTENT_ACTION_ADD.equals(intent.getAction())) {
            Log.d(LOG_TAG, "onStart action: INTENT_ACTION_ADD");
            addData();
            stopSelfCheck();
        }
    }

    return retVal;
}
```

The `addData()` generates random messages and inserts them into the notification content provider.

A Notification Extension may have several sources to group the events. For example, an email extension can have one source per mail account, allowing easy management of all events that belong to a certain mail account. The sample Notification Extension uses a single source for all events and `NotificationUtils.getSourceId()` is used to retrieve the source id for that source.

As background image the sample Notification Extension uses a picture stored in the extension resources. The `ExtensionUtils.getUriString()` is used to build a URI string to that resource so that the host application can retrieve this image.

Note: If new resources are added in a new version of the extension, the URI may change. It might then be necessary to update old events when a new version of the extension is installed.

```
private void addData() {
    Random rand = new Random();
    int index = rand.nextInt(5);
    String name = NAMES[index];
    String message = MESSAGE[index];
    long time = System.currentTimeMillis();
    long sourceId = NotificationUtil
           .getSourceId(this, EXTENSION_SPECIFIC_ID);
    if (sourceId == NotificationUtil.INVALID_ID) {
        Log.e(LOG_TAG, "Failed to insert data");
        return;
    }
    String profileImage = ExtensionUtils.getUriString(this,
            R.drawable.widget_default_userpic_bg);

    ContentValues eventValues = new ContentValues();
    eventValues.put(Notification.EventColumns.EVENT_READ_STATUS, false);
    eventValues.put(Notification.EventColumns.DISPLAY_NAME, name);
    eventValues.put(Notification.EventColumns.MESSAGE, message);
    eventValues.put(Notification.EventColumns.PERSONAL, 1);
    eventValues.put(Notification.EventColumns.PROFILE_IMAGE_URI, profileImage);
    eventValues.put(Notification.EventColumns.PUBLISHED_TIME, time);
    eventValues.put(Notification.EventColumns.SOURCE_ID, sourceId);

    try {
        getContentResolver().insert(Notification.Event.URI, eventValues);
    } catch (IllegalArgumentException e) {
        Log.e(LOG_TAG, "Failed to insert event", e);
    } catch (SecurityException e) {
        Log.e(LOG_TAG, "Failed to insert event, is Live Ware Manager
installed?", e);
    }
}
```

When you click the action button in an event, the `onViewEvent()` is called. Data from this event is read from the *LiveWare™ manager* content provider and a toast is displayed.

```
@Override
protected void onViewEvent(Intent intent) {
    String action = intent.getStringExtra(Notification.Intents.EXTRA_ACTION);
    int eventId = intent.getIntExtra(Notification.Intents.EXTRA_EVENT_ID, -1);
    if (Notification.SourceColumns.ACTION_1.equals(action)) {
         doAction1(eventId);
    }
}
```

```
/**
 * Show toast with event information
 *
 * @param eventId The event id
 */
public void doAction1(int eventId) {
    Log.d(LOG_TAG, "doAction1 event id: " + eventId);
    Cursor cursor = null;
    try {
        String name = "";
        String message = "";
        cursor = getContentResolver().query(Notification.Event.URI, null,
                Notification.EventColumns._ID + " = " + eventId, null, null);
        if (cursor != null && cursor.moveToFirst()) {
            int nameIndex =
                cursor.getColumnIndex(Notification.EventColumns.DISPLAY_NAME);
            int messageIndex =
                cursor.getColumnIndex(Notification.EventColumns.MESSAGE);
            name = cursor.getString(nameIndex);
            message = cursor.getString(messageIndex);
        }

        String toastMessage = getText(R.string.action_event_1) + ", Event: "
            + eventId
                + ", Name: " + name + ", Message: " + message;
        Toast.makeText(this, toastMessage, Toast.LENGTH_LONG).show();
    } finally {
        if (cursor != null) {
            cursor.close();
        }
    }
}
```

8.2 SampleRegistrationInformation.java

The *SampleRegistrationInformation.java* file extends *com.sonyericsson.extras.liveware.extension.util.registration.RegistrationInformation.*

This section describes the part of *SampleRegistrationInformation* that is specific for a Notification Extension.

First of all, *getRequiredNotificationApiVersion* is called to check what version of the Notification API the extension supports (if any).

```
@Override
public int getRequiredNotificationApiVersion() {
    return 1;
}
```

In addition to the normal extension registration, a Notification Extension also needs to register its event sources (at least one source is needed). The *getSourceRegistrationConfiguration* is used by the *ExtensionService* to retrieve the sources from *RegistrationInformation*.

```
@Override
public ContentValues[] getSourceRegistrationConfigurations() {
    List<ContentValues> bulkValues = new ArrayList<ContentValues>();
    bulkValues

.add(getSourceRegistrationConfiguration(SampleNotificationService.EXTENSION_SPE
CIFIC_ID));
    return bulkValues.toArray(new ContentValues[bulkValues.size()]);
}

public ContentValues getSourceRegistrationConfiguration(String
extensionSpecificId) {
     ContentValues sourceValues = null;

     String iconSource1 = ExtensionUtils.getUriString(mContext,
             R.drawable.icn_30x30_message_notification);
     String iconSource2 = ExtensionUtils.getUriString(mContext,
             R.drawable.icn_18x18_message_notification);
     String actionEvent1 =
         mContext.getString(R.string.action_event_1);

     sourceValues = new ContentValues();
     sourceValues.put(Notification.SourceColumns.ACTION_1, actionEvent1);
     sourceValues.put(Notification.SourceColumns.ENABLED, true);
     sourceValues.put(Notification.SourceColumns.ICON_URI_1, iconSource1);
     sourceValues.put(Notification.SourceColumns.ICON_URI_2, iconSource2);
     sourceValues.put(Notification.SourceColumns.UPDATE_TIME,
         System.currentTimeMillis());

     // Source specific values
     sourceValues.put(Notification.SourceColumns.NAME,
             mContext.getString(R.string.source_name));
     sourceValues.put(Notification.SourceColumns.EXTENSION_SPECIFIC_ID,
             extensionSpecificId);

     return sourceValues;
}
```

8.3 SamplePreferenceActivity.java

The preference activity of the sample Notification Extension allows the user to activate and deactivate the periodical insertion of new events. The *OnPreferenceChangeListener* is set up in the *onCreate* method of the preference activity.

```
// Handle active
preference = (CheckBoxPreference)
findPreference(getText(R.string.preference_key_is_active));
    preference.setOnPreferenceChangeListener(new OnPreferenceChangeListener() {

    public boolean onPreferenceChange(Preference preference, Object newValue) {
        if ((Boolean)newValue) {
            startSampleExtensionService();
        } else {
            stopSampleExtensionService();
        }
        return true;
    }
});


/**
 * Activate event generation
 */
private void startSampleExtensionService() {
    Intent serviceIntent = new Intent(this, SampleExtensionService.class);
    serviceIntent.setAction(SampleExtensionService.INTENT_ACTION_START);
    startService(serviceIntent);
}

/**
 * Cancel event generation
 */
private void stopSampleExtensionService() {
    Intent serviceIntent = new Intent(this, SampleExtensionService.class);
    serviceIntent.setAction(SampleExtensionService.INTENT_ACTION_STOP);
    startService(serviceIntent);
}
```

You can delete all events from the notification content provider. You can only do this when the preference activity is opened from a host application where the accessory supports browsing old events.

In SmartWatch, you can see old events but in Smart Wireless Headset pro you can only see the latest one. In the headset there is no need to show the Clear All Events option. This option is hidden if the *EXTRA_ACCESSORY_SUPPORTS_HISTORY* is set to false in the intent that started the preference activity. The *ExtensionUtils.supportsHistory* method is used to find this parameter in the preference activity's *onCreate* method.

```
// Remove preferences that are not supported by the accessory
if (!ExtensionUtils.supportsHistory(getIntent())) {
    preference =
        findPreference(getString(R.string.preference_key_clear));
    getPreferenceScreen().removePreference(preference);
}
```

8.4 SmartWatch UI

The images show where the different content provider fields are displayed in the *SmartWatch*.

Figure 17 Widget level overview

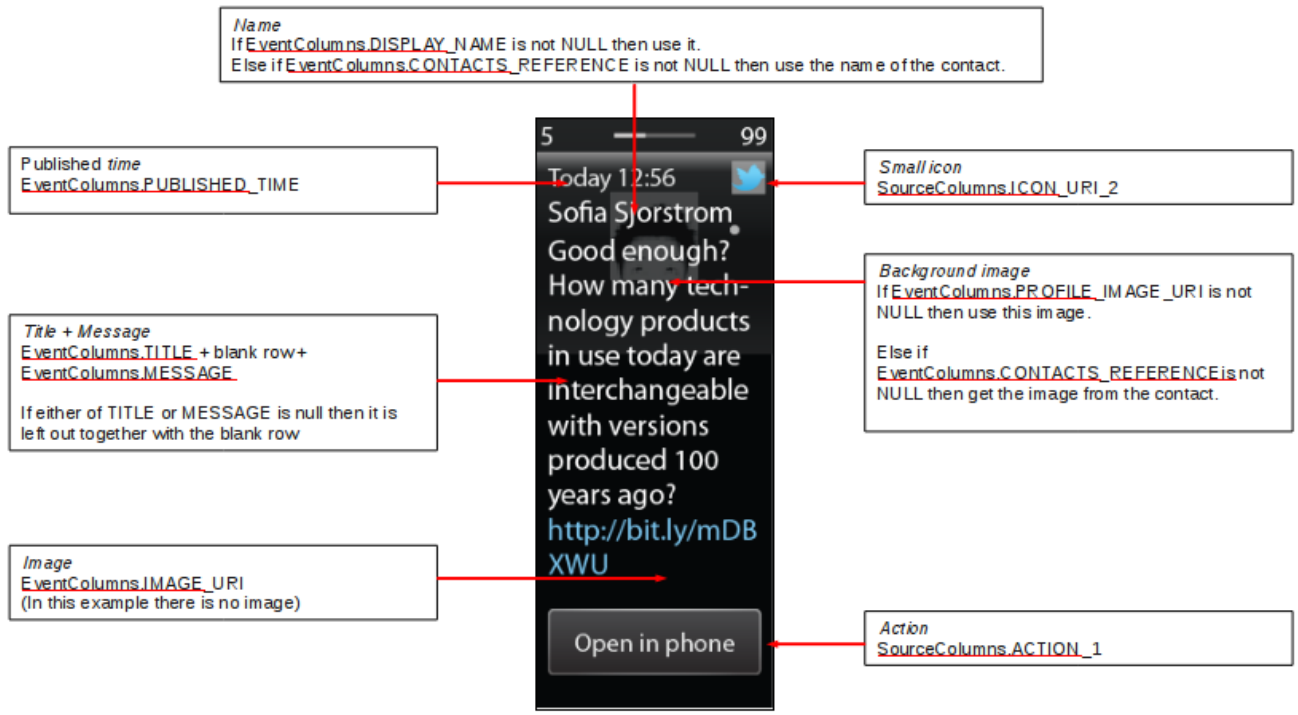

Figure 18 Event level overview

8.5 Smart Extension emulator

The Smart Extension emulator emulates the SmartWatch when a *Sample Notification Extension* runs on the phone.

As you can see in the Events tab, the only thing *Sample Notification Extension* does when activated, is to insert events in the *LiveWare™ manager* content provider every 5 seconds.

Figure 19 Events

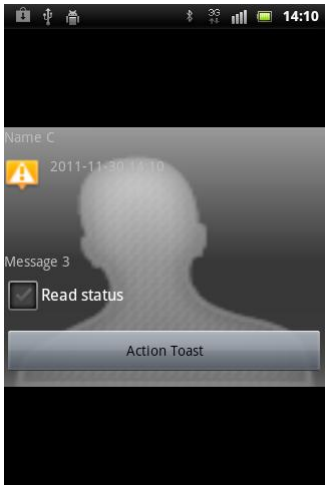

Figure 20 Event details

Figure 21 Action

Figure 22 Select Preferences

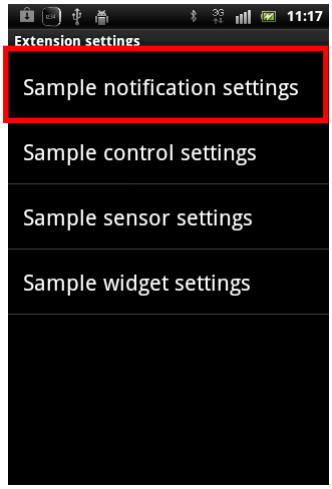

Figure 23 Select Extension

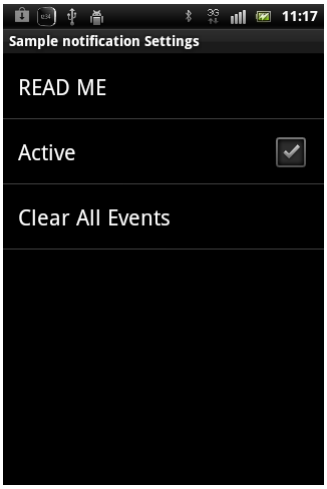

Figure 24 SamplePreferenceActivity. Active has been checked

9. Sample Sensor Extension

The *Sample Sensor Extension* is a sample project showing an example of how to use the *SmartExtensionUtils* to develop an application that uses the sensors in a Smart Accessory.

The Sensor Extension displays the latest values from the accelerometer sensor on the accessory display. To update the display, the sample Sensor Extension also uses the Control API. This control is also needed since the *SmartWatch* accessory only provides sensor data when a control is active.

9.1 SampleSensorControl.java

The *SampleSensorControl.java* file extends *com.sonyericsson.extras.liveware.extension.util.control.ControlExtension.*

This class is responsible for retrieving sensor information, rendering image and handling user interaction on the Smart Accessory. This section describes how sensor information is accessed.

The *SmartExtensionUtils* creates the *ControlExtension* object when the extension is given control over the display. The Sample Sensor Extension creates an *AccessorySensorManager* to access the sensors on an accessory. The *AccessorySensorManager* is then used to retrieve the accelerometer.

```
SampleSensorControl(final String hostAppPackageName, final Context context) {
    super(context, hostAppPackageName);

    AccessorySensorManager manager = new AccessorySensorManager(context,
hostAppPackageName);
    mSensor = manager.getSensor(Sensor.SENSOR_TYPE_ACCELEROMETER);
}
```

The method `onResume()` is called when the control is visible on the display. The Sensor Extension registers a *SensorEventListener* in interrupt mode that notifies when new sensor data is available. Interrupt mode means that events will only be sent when the sensor values changes. Finally the display is updated.

```
@Override
public void onResume() {
    Log.d(SampleExtensionService.LOG_TAG, "Starting control");
    // Note: Setting the screen to be always on will drain the accessory
    // battery. It is done here solely for demonstration purposes
    setScreenState(Control.Intents.SCREEN_STATE_ON);

    // Start listening for sensor updates.
    if (mSensor != null) {
        try {
            mSensor.registerInterruptListener(mListener);
        } catch (AccessorySensorException e) {
            Log.d(SampleExtensionService.LOG_TAG, "Failed to register
listener");
        }
    }

    updateDisplay(null);
}
```

New sensor data is delivered through the method `onSensorEvent()` in the *AccessorSensorEventListener* interface. The received sensor event is used to update the display.

```
private final AccessorySensorEventListener mListener = new
AccessorySensorEventListener() {

    public void onSensorEvent(AccessorySensorEvent sensorEvent) {
         updateDisplay(sensorEvent);
    }
};
```

The actual sensor values are read from the sensor event and the text views on the display are updated.

```
// Update the values.
if (sensorEvent != null) {
    float[] values = sensorEvent.getSensorValues();

    if (values != null && values.length == 3) {
        // Show values with one decimal.
        xView.setText(String.format("%.1f", values[0]));
        yView.setText(String.format("%.1f", values[1]));
        zView.setText(String.format("%.1f", values[2]));
    }

    // Show time stamp in milliseconds. (Reading is in nanoseconds.)
    timeStampView.setText(String.format("%d", (long)(sensorEvent.getTimestamp()
/ 1e6)));

    // Show sensor accuracy.
    accuracyView.setText(getAccuracyText(sensorEvent.getAccuracy()));
}
```

The `onPause()` is called when the extension is no longer visible on the display. The event listener is unregistered to make sure no more sensor updates are received.

```
@Override
public void onPause() {
    // Stop sensor
    if (mSensor != null) {
        mSensor.unregisterListener();
    }
}
```

9.2 SampleExtensionService.java

The *SampleExtensionService.java* file extends *com.sonyericsson.extras.liveware.extension.util.ExtensionService.*

This section describes the part of *SampleExtensionService* that is specific for the sample Sensor Extension.

The method *createControlExtension* is called when a control should be visible on the accessory display. The *SampleExtensionService* returns a *SampleSensorControl* and the *SmartExtensionUtils* then calls the method `onStart()` on the returned control object.

```
@Override
public ControlExtension createControlExtension(String hostAppPackageName) {
    return new SampleSensorControl(hostAppPackageName, this);
}
```

When the control is visible, the *SampleExtensionService* needs to be running in order to maintain the state of the animation. When the control is no longer visible, the sample Sensor Extension does not need to run and can it return false in `keepRunningWhenConnected()`.

```
@Override
protected boolean keepRunningWhenConnected() {
    return false;
}
```

9.3 SampleRegistrationInformation.java

The *SampleRegistrationInformation.java* file extends *com.sonyericsson.extras.liveware.extension.util.registration.RegistrationInformation.*

This section describes that part of *SampleRegistrationInformation* that is specific for the sample Sensor Extension.

The `getRequiredSensorApiVersion()` and `getRequiredControlApiVersion()` is implemented to set that the Sample Sensor Extension implements version 1 of both the Control and the Sensor API.

```
@Override
public int getRequiredControlApiVersion() {
    return 1;
}

@Override
public int getRequiredSensorApiVersion() {
    return 1;
}
```

In order to be displayed as a control on a Smart Accessory, the sample Sensor Extension must register with the accessory host application in the registration content provider.

The *AsyncTask* in *ExtensionService* uses the *RegistrationInformation* to register with all host applications that has a display supported by the extension. The `isDisplaySizeSupported()` asks the extension whether the display is supported, and `isSensorSupported()` asks the extension whether the sensor is supported.

```
@Override
public boolean isDisplaySizeSupported(int width, int height) {
    return (width == SampleSensorControl.WIDTH && height ==
SampleSensorControl.HEIGHT);
}

@Override
public boolean isSensorSupported(AccessorySensor sensor) {
    return Sensor.SENSOR_TYPE_ACCELEROMETER.equals(sensor.getType().getName());
}
```

Normally, the extension will be registered as a control as soon as a supported display is found and registered as a sensor as soon as a supported sensor is found. The sample Sensor Extension requires both a supported control and a supported sensor.

This is achieved by overriding `isSupportedSensorAvailable()` and `isSupportedControlAvailable()` and only returning true when the base class implementation for both these methods return true.

```
@Override
public boolean isSupportedSensorAvailable(Context context, HostApplicationInfo
hostApplication) {
    // Both control and sensor needs to be supported to register as sensor
    return super.isSupportedSensorAvailable(context, hostApplication)
            && super.isSupportedControlAvailable(context, hostApplication);
}

@Override
public boolean isSupportedControlAvailable(Context context, HostApplicationInfo
hostApplication) {
    // Both control and sensor needs to be supported to register as control.
    return super.isSupportedSensorAvailable(context, hostApplication)
            && super.isSupportedControlAvailable(context, hostApplication);
}
```

Trademarks and acknowledgements

Sony, "make.believe" is a trademark or registered trademark of Sony Corporation.

Ericsson is a trademark or registered trademark of Telefonaktiebolaget LM Ericsson.

Android and Android Market, are trademarks or registered trademarks of Google Inc.

All other trademarks and copyrights are the property of their respective owners.