A1. Info From the PDF Smartwatch_WhitePaper_1
==========

The SmartWatch display is 128 pixels wide, 128 pixels high and has 65536 colours. The touch screen is limited to 9 distinct touch areas in a 3 x 3 matrix.

A control extension is in control of the entire display. The images are sent as RGB565 (no alpha) to the accessory, so it is recommended to use `Bitmap.Config.RGB_565` for the bitmaps drawn by the extension. Images from other sources should not have a transparent background.

This extension sends its images to the host application which converts the images to a format supported by the accessory. Images sent to the host application should be in .png format.

For a control extension, the SmartWatch supports both tap touch (through CONTROL_TOUCH_EVENT_INTENT or `ControlExtension.onTouch()`) and swipe motions (through CONTROL_SWIPE_EVENT_INTENT or `ControlExtension.onSwipe()`).

Users can search for compatible extensions via the host application. Your market description should end with “\n\nLiveWareTM extension for SmartWatch“ to ensure that your extensions are found.

A2. From the PDF SmartExtension SampleExtension Tutorial
==========

1) You need to add the ***SmartExtensionUtils*** to your project as an Android Library project. The SmartExtensionUtils include the Smart Extension API from SmartExtensionAPI so you only need to add one of them to your project.

2) ***Android Manifest.xml***

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

<!-- Extension service  -->
<service android:name=". SampleExtensionsService" />

<!-- Extension receiver -->
<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>
```

3.1) ***SampleRegistrationInformation.java***

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

3.2) ***SampleControlSmartWatch.java***

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.

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.

3.3) ***SampleExtensionService.java***

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.

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().

A3. From the PDF SmartExtension API Specification
==========

1) Global architecture

There is one host application per accessory model. The hardware unit communicates via Bluetooth with a host application that runs on the phone. The host application controls what is shown on the hardware. The hardware just shows images on its display, prepared by the host application.

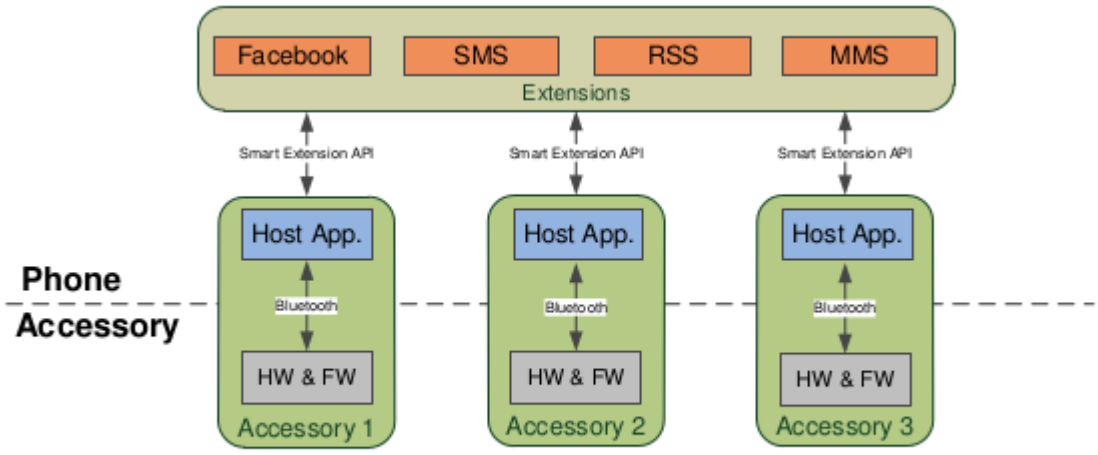

Figure 1 System overview

This architecture is used for all Smart ExtrasTM from Sony Ericsson.

2) Component architecture

A host application lets you customise what to transfer/view to the hardware. The content shown on the display (Facebook, Twitter, missed calls etc.) is typically not part of a host application. Those are separate small applications that we call Accessory Extension Application (AEA).

AEA communicates with an AHA via the Smart Extension API which is described in this document. An extension is not tied to a certain accessory. Specifically, one extension that enables some data can provide this data to several different accessories. It is a host application's task to render the data so that it matches the accessory's capabilities.

A host application acts as a backend for the Smart Extension API.

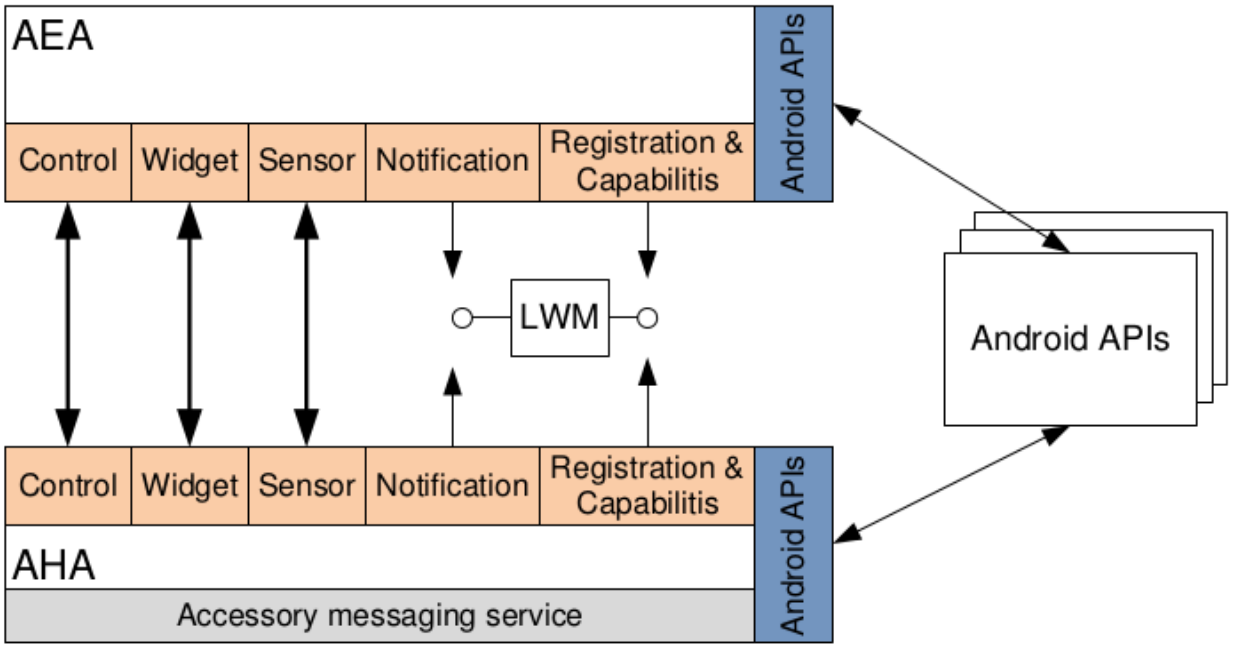

Figure 2 Detailed system overview

- Registration & Capabilities API handles the AEA and AHA registrations and also provides the capabilities of the AHA.
- Notification API can typically be used by simple event driven data providers such as SMS, MMS, Missed Calls, Facebook, Twitter etc.
- Control API is the most advanced, it lets you take full control of the accessory screen.

Some of the APIs are Intent based (Control, Widget, Sensor) and some are based on ContentProviders (Notification and Registration & Capabilities).

An AEA will have to register itself before it can start communicating with the AHA. This means that every AEA will have to implement at least the Registration & Capabilities API plus one or more of the other APIs.

Security

To make sure that intents sent from an extension are delivered only to host applications, the permission HOSTAPP_PERMISSION must be inserted in the call to sendBroadcast:

getApplicationContext().sendBroadcast(intent,Registration.HOSTAPP_PERMISSION);

Registration

Before an extension can start to communicate with the host application and send data, it needs to register tself in the LiveWareTM manager.
Every host application also registers to the LiveWareTM manager by sending its capabilities.

Before an application can register itself as an extension, there must be at least one host application installed on the phone. This is to prevent that extensions start writing data into the databases when there are no host applications (user has no accessories).

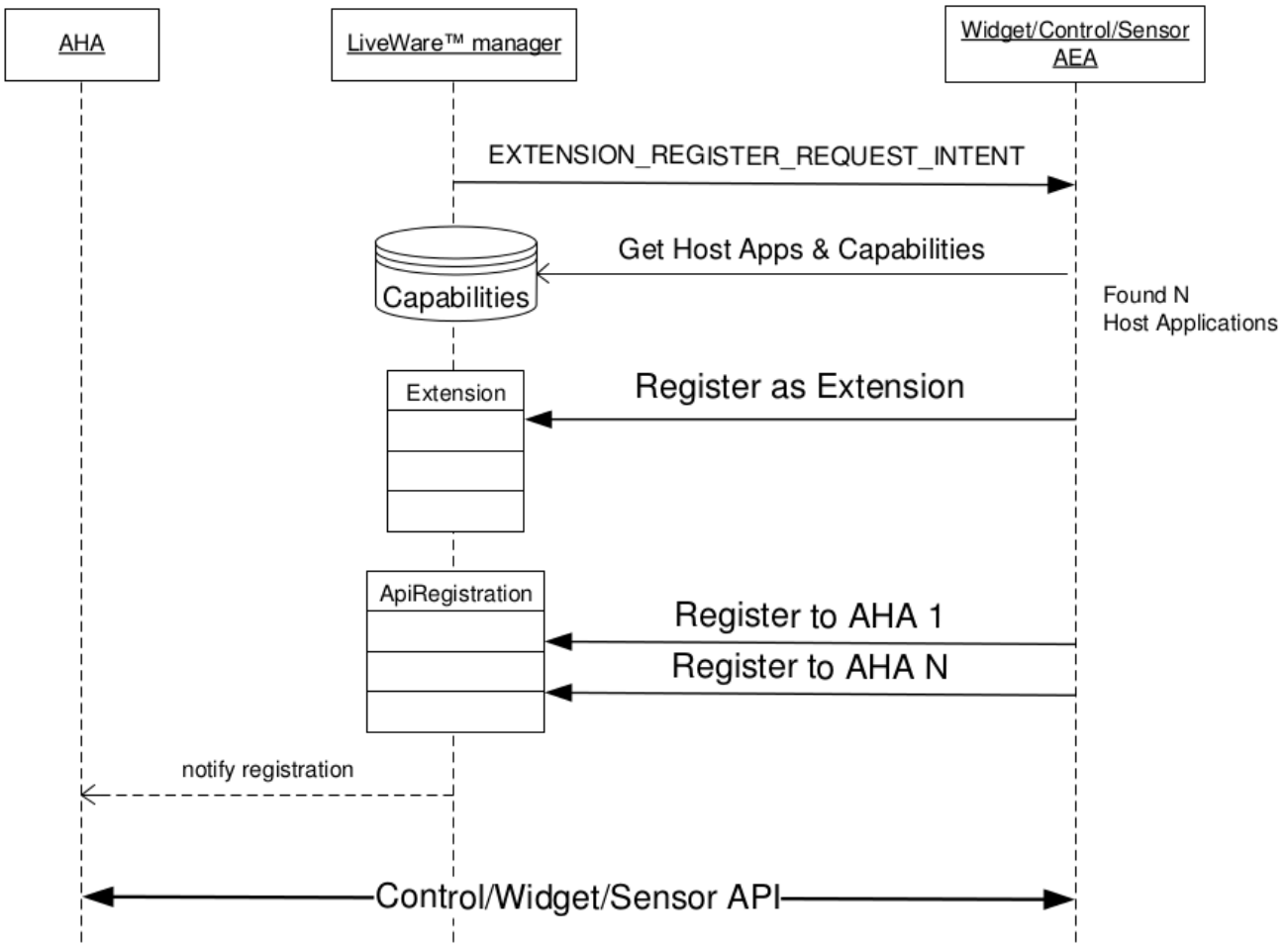

Figure 7 LiveWare™ manager extension registration

The control extension lifecycle

Since only one extension can run at a time, the lifecycle needs to be controlled by the host application. An extension cannot just start executing at a random time. It needs to make sure that no other extension is running, therefore the extension can only request to be started using CONTROL_START_REQUEST_INTENT.

When the host application is ready to start the extension, it will send a CONTROL_START_INTENT, see Figure 11.
When the extension requests to take control of the accessory, host applications can choose either to
accept the request giving control to the extension or, if something is not right, send a CONTROL_ERROR_INTENT.

The CONTROL_RESUME_INTENT is sent when the extension is visible on the accessory. From this point on, the extension controls everything and the host application just forwards the information between the accessory and the extension.

An extension can be paused, either if a high priority extension needs to run for a while, or if the host application is in charge of the display state and the display is turned off. In this case, the host application sends out a CONTROL_PAUSE_INTENT to the extension. This means that there is no point for the extension to update the display since it is either turned off or someone else is running for a while.

When the extension is in a paused state, it no longer has control over the display/LEDs/vibrator/key events. For example, a phone extension like an incoming call notification has higher priority than a random extension. In this case, we want to pause the running extension and let the phone extension take control. When the call is finished, the other extension can resume after receiving a CONTROL_RESUME_INTENT from a Host application.

When the user exits the extension, the host application sends a CONTROL_PAUSE_INTENT followed by a CONTROL_STOP_INTENT. From this point on the host application regains control.

If the extension wants to stop, it can send a CONTROL_STOP_REQUEST_INTENT to the host application. The host application will then make sure to stop it and send a CONTROL_STOP_INTENT. If the extension was not already paused, it will be paused before it is stopped and a CONTROL_PAUSE_INTENT is sent before the CONTROL_STOP_INTENT.

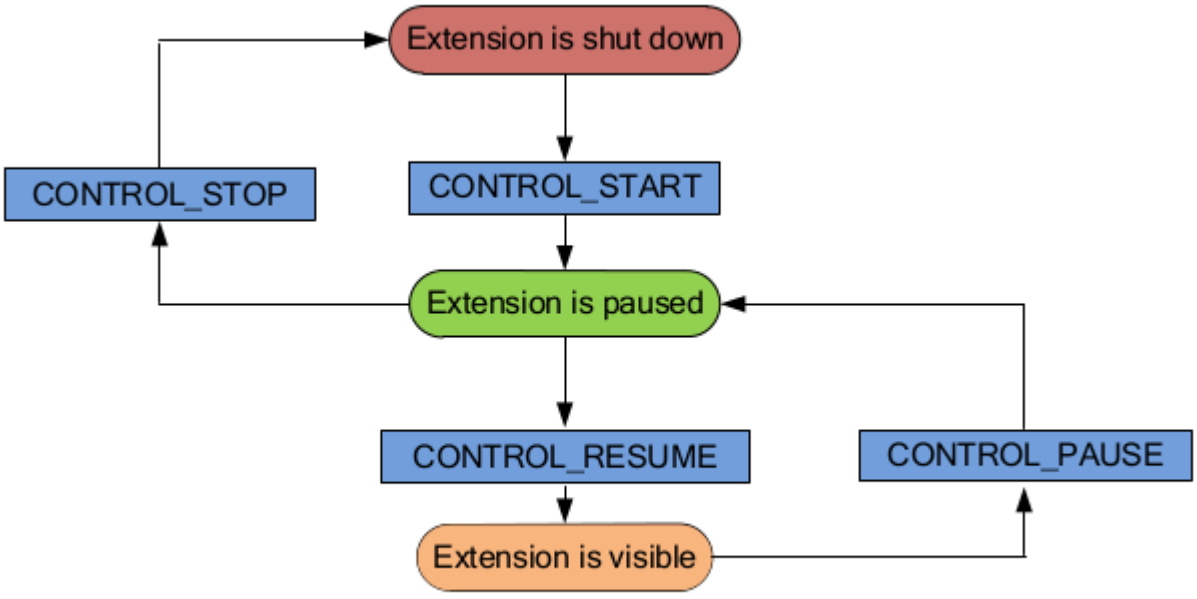

Figure 11 Control extension lifecycle

With a control extension, one can ***control***

- the ***display***, via the CONTROL_SET_SCREEN_STATE_INTENT
- the ***LEDs***, via the CONTROL_LED_INTENT
- the ***Vibrator***, via the CONTROL_VIBRATE_INTENT

Since the extension is controlling the accessory, it also controls ***what is visible on the display***. The content that the user sees comes from the extension. Basically, the extension sends images to be displayed on the accessory display. To find out the dimensions of the display and the colour depth it supports, the extension can use the *Registration and capabilities API*.

CONTROL_DISPLAY_DATA_INTENT is sent from the extension when it wants to update the accessory display.

Extensions can also clear the accessory display at any point by sending CONTROL_CLEAR_DISPLAY_INTENT. (`ControlExtension.clearDisplay()`)

Note that we are using Bluetooth for connectivity, which means that we can't send that many frames per second (FPS). Refresh rate of the display can be found in *Registration and capabilities API*.

Control extension can send images as a resource (R.drawable.<resource>) :

```
ControlExtension.showImage(final int resourceId)
```

, as a bitmap:

```
ControlExtension.showBitmap(final Bitmap)
```

or as the URI of the image to be displayed:

```
Uri uri = ContentUris.withAppendedId(contentUri, id);
InputStream in = res.openInputStream(uri);
Bitmap bitmap = BitmapFactory.decodeStream(in, null, bitmapOptions);
showBitmap(bitmap);
```

A control application can also ***handle events***, like

- ***Key*** events,
  a CONTROL_KEY_EVENT_INTENT is sent to the extension when a user presses a

key on the accessory.

- ***Touch*** events,
  a CONTROL_TOUCH_EVENT_INTENT is sent to the active the extension when a user taps the accessory display.

- ***Swipe*** events,
  a CONTROL_SWIPE_INTENT is sent to the active extension when the user swipes over the display.

B. STEPS TO PAIR SMARTPHONE WITH SMARTWATCH

1) On the phone, download the “LiveWare Manager / SmartConnect” app
2) Open the app -> Settings -> Add Bluetooth Device
3) Boot the Smartwatch and hold the power key even after smartwatch boots
4) Check the tick on the watch
5) Check the tick on the phone

C. SENDING AN IMAGE TO SMARTWATCH FOR DISPLAY: FUNCTION CALLS

```
-> onButtonClicked();
-> intent.setAction(START_CONTROL); startService(intent);
-> SWExtensionService.onStartCommand(intent, flags, startId);
-> ExtensionService.handleIntent(intent);
-> ExtensionService.handleControlIntent();
-> SWExtensionService.createControlExtension();
-> new SWControlExtension();
```