developer WORLD

Developer guidelines:

Smart extension API specification

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

First released version (January 2012)

Table of contents

1. Before you start

1.1 About this document

This document is intended for developers who want to develop apps that communicate with Sony Ericsson Smart Extras™. You will find a detailed architectural view of the Smart Extension API and also an overview of the different APIs that reside under the Smart Extension API umbrella.

Detailed API documentation can be found in the javadoc folder and is called Smart Extension API documentation.

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

1.3 Terminology

AEA	Accessory Extension Application (same as Extension)
AHA	Accessory Host Application / Host Application
APK	Android Package
Extension	Android Application that uses one or more of the APIs defined in this document
FPS	Frames Per Second
LWM	LiveWare™ manager
SEA	Smart Extension API
SDK	Software Development Kit

2. Introduction

The Smart Extension API is the successor of the plug-in API introduced in the LiveView™ SDK. It supports all Sony Ericsson advanced accessories.

The Smart Extension API consists of the following Android apps developed by Sony Ericsson:

- LiveWare™ Manager
- One host application per accessory, e.g. SmartWatch
- Extension SDK and its APIs

The Smart Extension API is used by extensions to communicate with Sony Ericsson Smart Extras™. Extensions are developed by third-party developers.

3. Design

3.1 Design overview

Since the processing capacity of mobile phones is getting quite substantial we would like to utilise this for making more advanced accessories.

The idea is to make the accessory hardware as lightweight as possible and put a big part of the logic on the phone side. Most of the functionality resides in an application on the phone. We call this application Accessory Host Application (AHA), see Figure 1 below.

3.2 Architecture

3.1.1. 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.

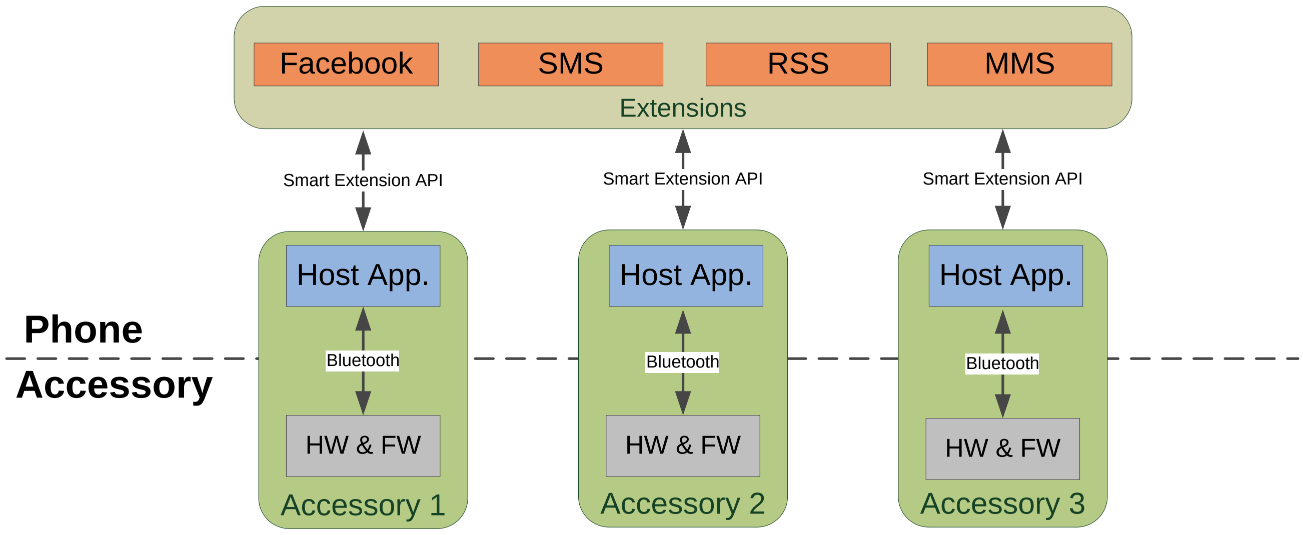

Figure 1 System overview

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 and provides services to:

- Communicate with the hardware unit
- Provide building blocks for the UI on the accessory
- Handle related Smart Extension signals/APIs
- Display a UI on the phone enabling users to customise certain behaviours (enable/disable extensions, enable/disable vibrator etc.).

This architecture is used for all Smart Extras™ from Sony Ericsson.

3.1.2. Component architecture

You can write extensions based on five different APIs. See Figure 2 below.

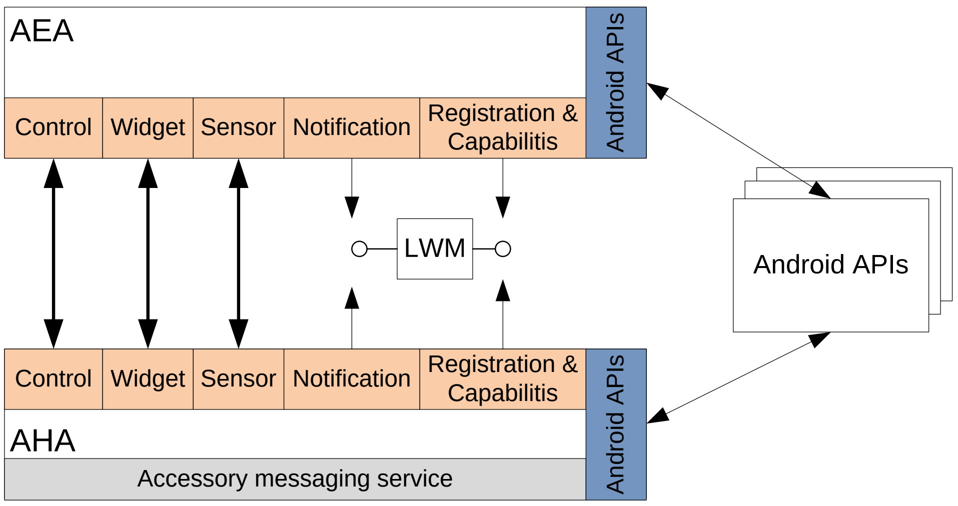

Figure 2 Detailed system overview

- 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.
- Widget API Lets you display a widget on the accessory. It is a small subset of the Control API.
- Sensor API Some of our advanced accessories will have sensors. This API makes the sensor data available to the extensions.

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.

All APIs are described in detail in the *Smart Extension API documentation*.

3.1.3. Security

An application that wants to use the APIs must in its *AndroidManifest.xml* declare that it uses the permission specified in the LiveWare™ manager APK.

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

If an extension fails to declare the use of this permission in its manifest, the implementation in the LiveWare™ manager does not allow the application to register as an extension and it will not be able to communicate with a host application.

Basically, all interactions via the LiveWare™ manager content provider interface will fail and a Security Exception is thrown. This forces extension developers to declare the above *<uses-permission>* tag.

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:*

```
Intent intent = new Intent(Control.Intents.CONTROL_CLEAR_DISPLAY_INTENT);
intent.putExtra(Control.Intents.EXTRA_AEA_PACKAGE_NAME,
        mContext.getPackageName());
intent.setPackage(mHostAppPackageName);
mContext.getApplicationContext().sendBroadcast(intent,
        Registration.HOSTAPP_PERMISSION);
```

4. The Registration and capabilities API

Before an extension can start to communicate with the host application and send data, it needs to register itself in the LiveWare™ manager. The registration is needed because:

- Every host application needs to display a list of available extensions in its UI so that the user can modify settings per extension.

- LiveWare™ manager has to be able to display a list of available extensions for a certain host application.

For the Notification API, it is enough that the extension registers itself only once. Other APIs are host application dependent, not every host application might support them. Also, every extension might not be designed to work with every host application. For this reason, extensions (other than those implementing the Notification API) need to also register themselves once per host application. This way the LWM has the complete picture and every host application is able to ask the LWM about its registered extensions.

Before an extension can decide if it wants to register to a certain host application, it has to have some information about it. Basically it needs to know the capabilities of the host application; does it have a display, how big is the display, etc.

For this reason we also need to store information about every host application in the LiveWare™ manager so that we can provide this info to the extensions when they ask for it. Therefore every host application also registers to the LiveWare™ manager by sending its capabilities.

4.1 Host application capabilities registration

The first thing a host application does when started is to write its capabilities into the capabilities database that resides in the LiveWare™ manager. This way, applications trying to register as extensions can read the information and decide whether to register or not, see Figure 3 below.

Capability data can be modified by the same host application that inserted it, as well as by the LiveWare™ manager (in case the host application is uninstalled).

Every application that uses the extension permission (see the Security section) can read this information. It is available to everyone because applications need it in order to decide if they want to register as extensions for a certain host application.

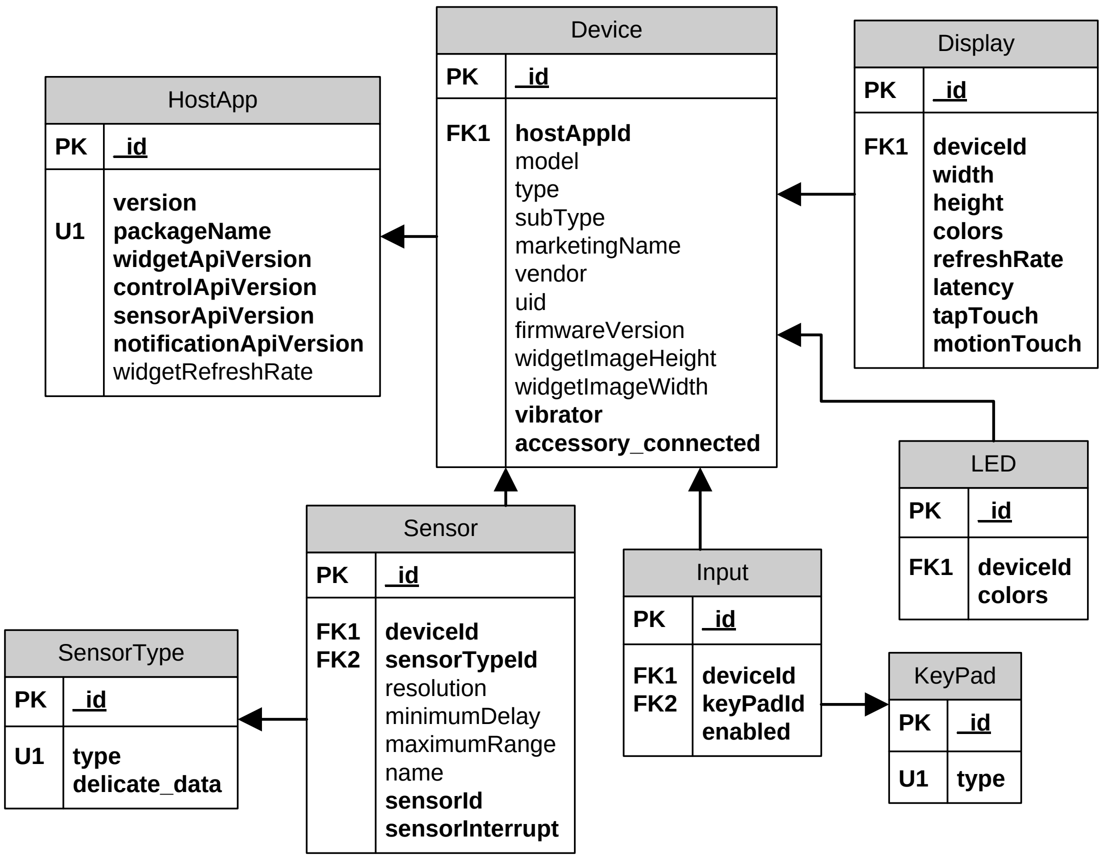

Figure 3 The capabilities database in LiveWare™ manager

4.2 Extension registration

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).

The registration is actually divided into two parts:

General extension registration just to become an extension, this also enables access to the Notification API.

Host application specific registration for the rest of the APIs, since these APIs are dependent of host application capabilities.

All applications using the extension permission (see section Security) can read the capabilities database in the LiveWare™ manager without being registered as extensions. This so that applications can decide whether to register or not.

The notification API does not depend on a host application and therefore it is not required to register to a host application. It is enough to register just as an extension to be able to use the Notification API.

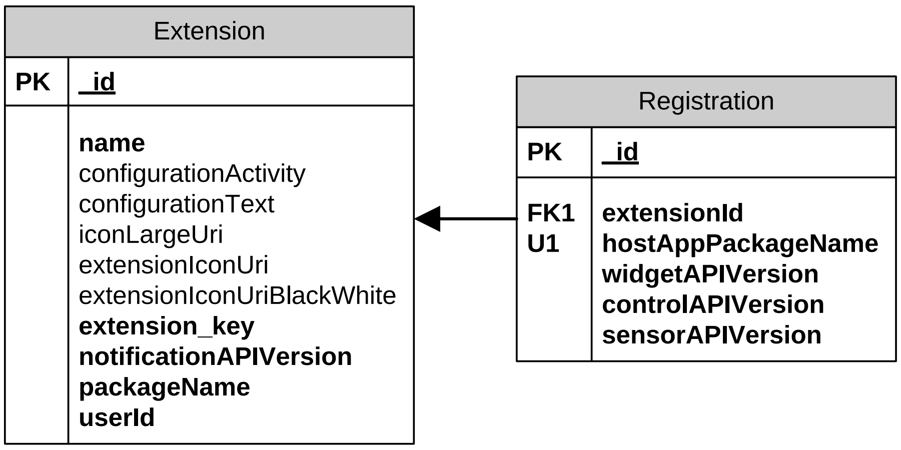

Figure 4 The extension registration database in LiveWare™ manager

4.2.1. General extension registration

Registration to be an extension means that there is information about the application in the extension table in the LiveWare™ manager. The information is described in Table 1.

When installed, the application may try to call the registration interface in LiveWare™ manager to add its information. It might also listen for a registration broadcast that is sent out by the LiveWare™ manager, EXTENSION_REGISTER_REQUEST_INTENT.

This broadcast is broadcasted at certain occasions, such as:

- When the LiveWare™ manager starts
- When the LiveWare™ manager detects that a new application has been installed
- When the LiveWare™ manager notices that a new host application has added its capabilities to the tables.

Note: EXTENSION_REGISTER_REQUEST_INTENT is only broadcasted if there is at least one registered host application.

If the Application tries to register itself and there are no host applications installed on the phone, the method `ContentProvider.insert()` throws an exception.

Column	**Info**	**Presence**	**Type**	**Data inserted by**	**Data readable by**
name	Displayable name of the extension that may be presented, e.g. in settings.	Required	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
configurationActivity	Package name and Class name of the Activity that contains the settings for the extension	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
configurationText	Short text to describe the current configuration state of the extension.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
iconLargeUri	URI of the Android launcher icon representing the extension. This icon is used by the host application when listing extensions.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
extensionIconUri	URI of the icon representing the extension. This icon is used on the accessory UI. The size is 34x34 pixels.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
extensionIconUriBlackWhite	URI of the monochrome icon representing the extension. This icon is used on the accessory UI. The size is 18x18 pixels.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.

extension_key	Used for security reasons for the extension's benefit. This key is sent as an extra-data in Intents sent to the extension from a host application. This enables the extension to verify that the sender has valid access to the registration content provider.	Required	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
notificationApiVersion	API version. If the extension uses the notification API, this field should tell what version of the notification API it uses. Otherwise it should be set to 0.	Required	Intege r 0..n	Extension	LiveWare™ manager, host applications and the extension that added it.
packageName	The package name of an extension. If an extension supports shared user id, the package name must be specified.	Optional (required if shared user id is used by extension)	Text	Extension	LiveWare™ manager, host applications and the extension that added it.

Table 1 Extension table columns, data insert responsibility and data readability

After the application is listed in the extension table, it is free to use the Notification API (if it supports that API) or to register itself for some of the other APIs, see section Host application specific registration.

4.2.2. Host application specific registration

Once the application has performed a successful extension registration, it is free to use only the Notification API (if the host application supports it).

If it wants to use some of the other APIs, e.g. the Control API, it needs to check the capabilities of the available host applications to verify whether they support this API (how big display they have etc).

If the Extension finds a matching host application, it will have to register itself for that specific host application in order to be able to use the API.

This is done by writing some data into the *API registration* table using the *ContentProvider* interface, see Table 2.

Column	Info	Presence	Type	Data inserted by	Data readable by
extensionId	Foreign key into the Extension table	Required	Integer	Extension	LiveWare™ manager, host applications and the Extension that added it.
hostAppPackageName	Package name of the host application you are registering yourself to.	Required	Text	Extension	LiveWare™ manager, host applications and the Extension that added it.
widgetAPIVersion	API version. If the widget API is used, this field should tell what version of the widget API that is used. Otherwise it should be set to 0.	Required	Integer 0..n	Extension	LiveWare™ manager, host applications and the Extension that added it.
controlAPIVersion	API version. If the control API is used, this field should tell what version of the control API that is used. Otherwise it should be set to 0.	Required	Integer 0..n	Extension	LiveWare™ manager, host applications and the extension that added it.

sensorAPIVersion	API version. If the sensor API is used, this field should tell what version of the sensor API that is used. Otherwise it should be set to 0.	Required	Integer 0..n	Extension	LiveWare™ manager, host applications and the extension that added it.

Table 2 API registration table columns, data insert responsibility and data readability

Registration on API level is per host application. If an extension wants to function with several host applications, it needs to register for each one of them.

4.2.3. Non-registered applications are prevented from writing data

A calling application that is not registered as an Extension (there is no information about it in the Extension table) will not be able to manipulate the data.

If the application tries to access a *ContentProvider* based API calling the `insert()`, `delete()` and `update()` methods, *null* is returned unless the application UID is listed in the Extension table.

If the application tries to access an Intent based API, the calls are ignored if the application package is not listed in the Extension table. If it tries to use some of the APIs that requires registration per host application and it hasn't registered for that host application, the calls are also ignored.

4.2.4. When the calling application is a registered extension

If an application is registered as an extension, it can successfully communicate with the Notification API. If other APIs are needed, a registration per host application is required (see section Host application specific registration).

If the extension tries to access a *ContentProvider* based API, can access and manipulate data previously inserted by the extension itself. However, it will not be able to read or update data that was not inserted by it.

If the extension tries to access an Intent based API, the calls are processed and appropriate feedback is given.

4.2.5. When no host applications are installed on the phone

You might end up in a situation where you have LiveWare™ manager installed and you have downloaded one or more extensions.

LiveWare™ manager will in that case not allow registrations. There is no point in registering if there are no host applications. This is to prevent the databases from growing and taking up space on the phone.

As soon as a host application is installed and it registers its capabilities into the LiveWare™ manager database, LiveWare™ manager will send a broadcast to all extensions to indicate that it is ok to register.

4.3 Use cases

4.3.1. LiveWare™ manager initiates Notification Extension registration

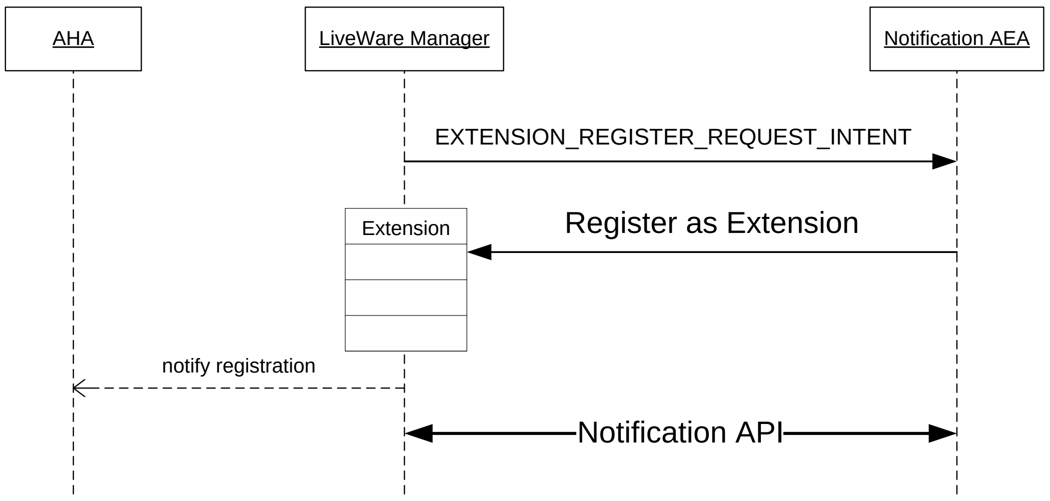

Figure 5 LiveWare™ manager notification

4.3.2. Extension initiates Notification Extension registration

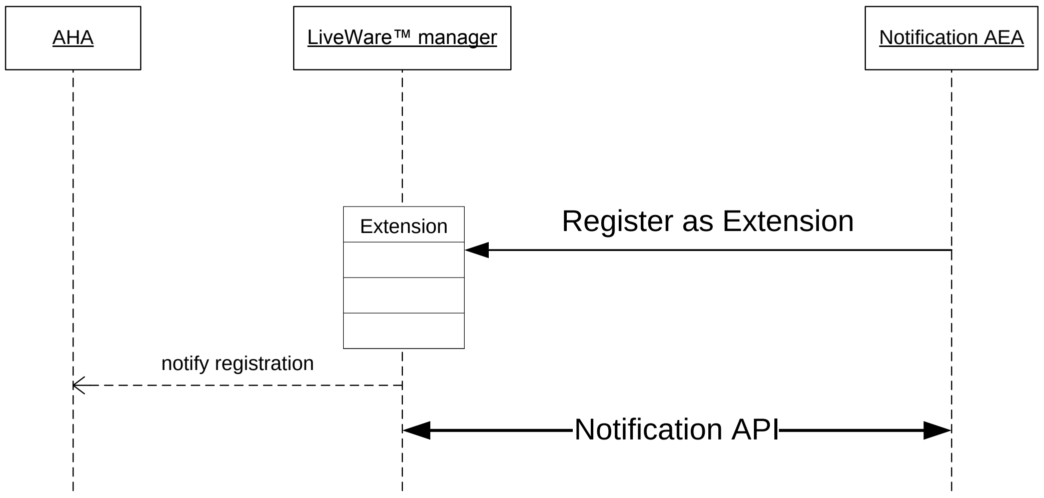

Figure 6 Extension notification

4.3.3. LiveWare™ manager initiates widget/control/sensor Extension registration

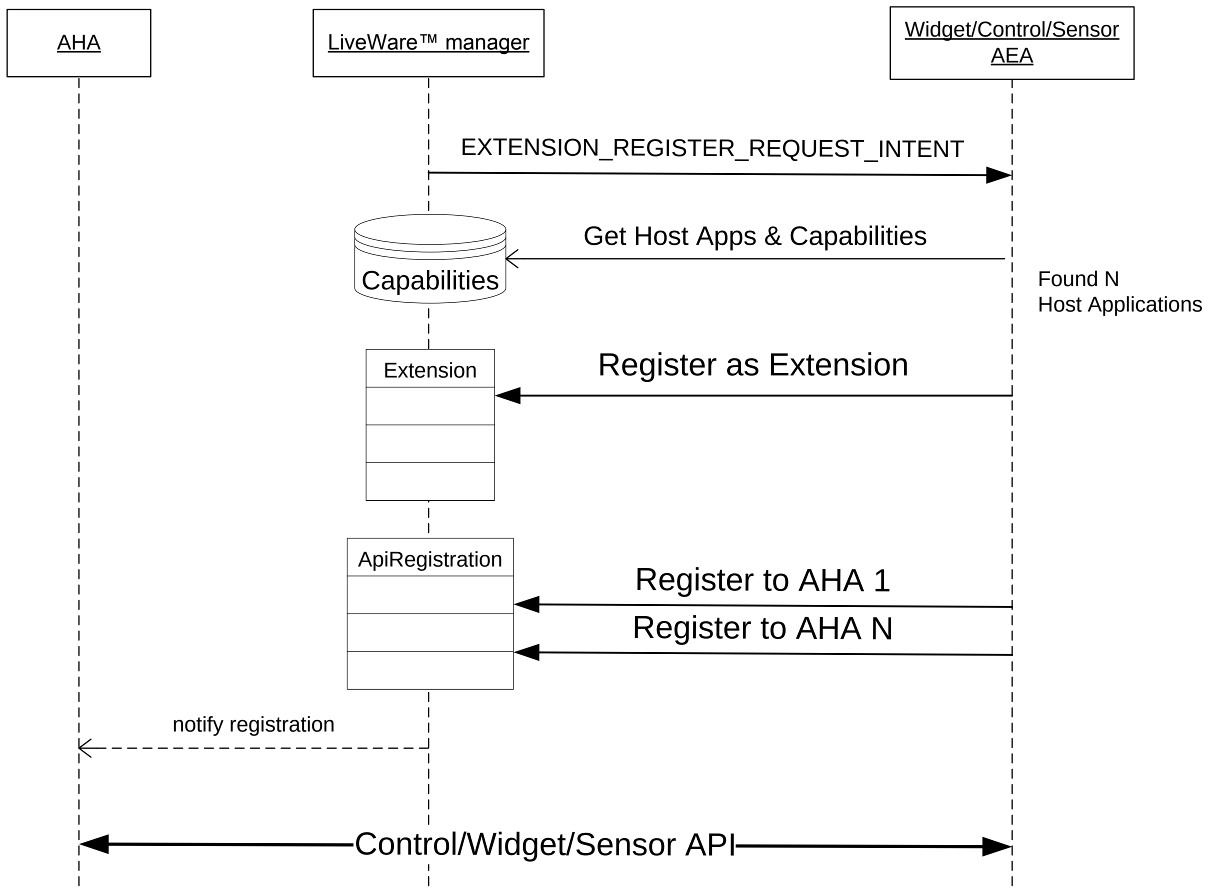
AHA
LiveWare™ manager
Widget/Control/Sensor AEA
EXTENSION_REGISTER_REQUEST_INTENT
Get Host Apps & Capabilities
Capabilities
Found N Host Applications
Extension
Register as Extension
ApiRegistration
Register to AHA 1
Register to AHA N
notify registration
Control/Widget/Sensor API

Figure 7 LiveWare™ manager extension registration

5. The Notification API

In order to use this API your application must have registered itself properly to the LiveWare™ manager.

Applications implementing this API are simple event driven data providers, such as SMS, MMS, Facebook and Twitter.

One of the major drivers behind this API is the fact that the extensions should be totally reusable and easy for developers to write.

Detailed API definition can be found in the *Smart Extension API documentation*.

5.1 Architecture

The data is stored into the LiveWare™ manager, see Figure 8 below.

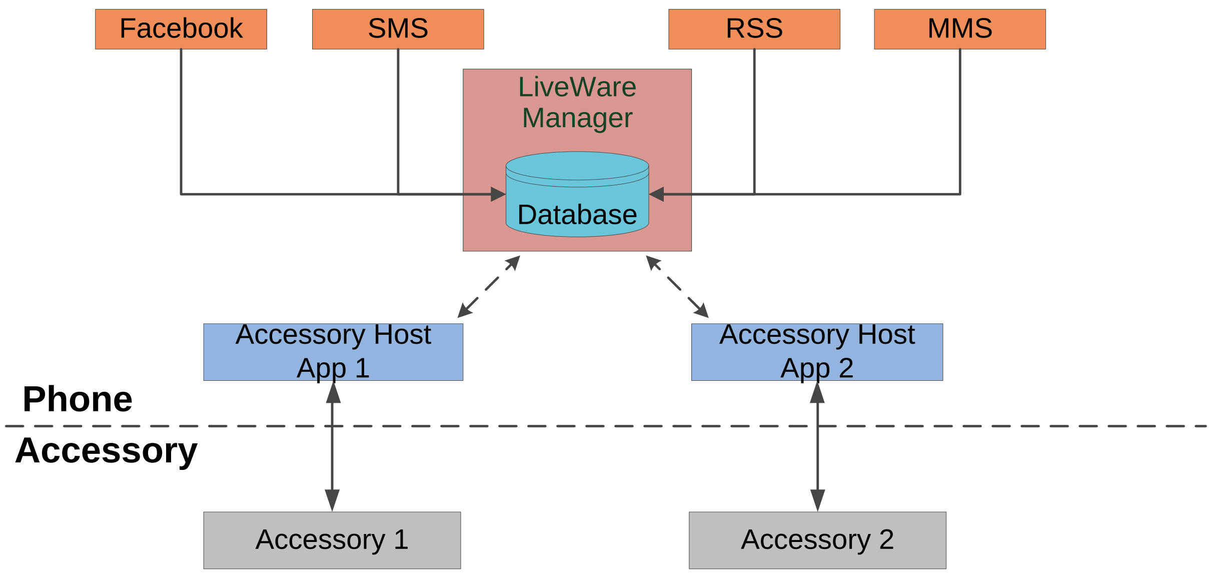

Figure 8 Notification API design overview

There are two sorts of data consumers stored in the database; host applications and extensions that have registered with the LiveWare™ Manager to provide their data to be rendered.

These communicate with the database via a content provider interface. Host applications can access all information in the database, while extensions can access their own data (read/write).

There are a few data access rules:

- An extension that wants to store data in LiveWare™ manager's content provider must register itself as an extension to the LiveWare™ manager.
- An extension is always able to access its own data stored in the database.
- When the extension is uninstalled, all its data is removed by the LiveWare™ manager.
- Host applications are allowed to read and modify extension-inserted data in the LiveWare™ manager.

The API consists of a content provider interface. The backend database consists of two tables, *Source* and *Event*.

These tables are populated with event data from the extensions. The data is read by host applications and displayed on the accessory screen. This way, extensions don't communicate directly with host applications.

The sole content provider is not enough because the event data stored in the database might just be a snapshot of the information and the user might wish to view the details.

For this reason we also need a mechanism to notify the extension that the user would like to view details. See section Show the Detail View of an event.

5.1.1. Source

Source is a logical abstraction introduced to enable extension developers to distinguish the presentation of data connected to different backends and at the same time be able to package these as a single APK.

For example, an email aggregator extension that allows the user to connect to several email accounts through the installation of only one APK. The developer of this extension can choose whether each email account should have its own Source or define only one Source. In the latter scenario, emails from a specific email account could not easily be filtered, manipulated and displayed.

Host applications and extensions may use this to filter event data by Source or to provide configuration options in the user interface to filter event data by Source.

Extension developers who want host applications to display events from different Sources should register source information. There is a limit of eight Sources per extension. If the limit is reached, an exception is thrown.

Column	Info	Presence	Type	Inserted by	Readable by
name	Displayable name of the source	Required	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
iconUri1	Each Source can use up to 3 icons with different sizes and one monochrome. URI of the largest icon (30x30 pixels).	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
iconUri2	Each Source can use up to 3 icons	Optional	Text	Extension	LiveWare™ manager, host

	with different sizes and one monochrome. URI of the second largest icon (18x18 pixels)				applications and the extension that added it.
iconUriBlackWhite	Each Source can use up to 3 icons with different sizes and one monochrome. URI of the monochrome icon	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
enabled	Indicates if the source is enabled.	Required	Boolean	Extension	LiveWare™ manager, host applications and the extension that added it.
action_1	Action supported by the extension (Reply, Call, etc.). The action is defined by the extension and supported for this source.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
action_2	Action supported by the extension (Reply, Call, etc.). The action is defined by the extension and supported for this source.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
action_3	Action supported by the extension (Reply, Call, etc.). The action is defined by the extension and supported for this source.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
updateTime	The time when an event linked to this source was created/updated.	Optional	Integer	Extension	LiveWare™ manager, host applications and the extension that added it.
textToSpeech	Text to speech specific text. The text in this column is used in combination with the events of the	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.

	source to create speech events. The text in this column is read out before the events.				
extension_specific_id	Extension specific identifier of the source It is up to the extension to define this identifier.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
packageName	The package name of a plug-in. If an extension supports shared user id, the package name must be specified.	Optional (required if shared user id is used by extension)	Text	Extension	LiveWare™ manager, host applications and the Extension that added it.

Table 3 Source table columns, data type, data insert responsibility and data readability

5.1.2. Event

An Event is a representation of a notification that may be noteworthy to present to the end user. Examples of events are:

- Incoming SMS message notifications.
- A missed call notification.
- Events from friends on a social network.

The Event table is used to store events provided by the extensions. Host applications typically use the information in this table to present the data on the accessory display.

An Event is always connected to a Source but a Source may not always have to have an Event.

The number of events that can be stored in the Event table is limited to 100 per source. When the limit is reached, the oldest events are automatically removed.

Column	**Info**	**Presence**	**Type**	**Inserted by**	**Readable by**
sourceId	The ID of the host source.	Required	Integer	Extension	LiveWare™ manager, host applications and the extension that added it.
message	Message associated with this event.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
imageUri	Content URI to an image linked to this event.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.

publishedTime	The time (in milliseconds since January 1, 1970 00:00:00 UTC UNIX EPOCH) when the content linked with this data row was published on the source. Should be stored as GMT+0.	Required	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
title	Short text describing the title for event linked with this data row. This can be the phone number, username, email address etc.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
personal	Whether the event linked with this data row is specifically directed to the user ("me") or concerns the user ("me"), e.g. received SMS, Facebook private message to the logged-in user etc.	Required		Extension	LiveWare™ manager, host applications and the extension that added it.
geoData	Geo data associated with this event	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
readStatus	Indicates if the event has been read.	Optional	Boolean	Extension or host application	LiveWare™ manager, host applications and the extension that added it.
timeStamp	The time (in milliseconds since January 1, 1970 00:00:00 UTC UNIX EPOCH) when this row was created. The time stamp is set automatically.	Required		LiveWare™ manager	LiveWare™ manager, host applications and the extension that created the row.
display_name	Displayable name of the sender, linked with this data row, e.g. full name.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
profile_image_uri	URI to the profile image of the sender, linked with this data row.	Optional	Text	Extension	LiveWare™ manager, host applications and the extension that added it.
contacts_reference	A reference to the	Optional	Text	Extension	LiveWare™ manager,

	contacts content provider. Name and profile image from this contact reference are used if not explicitly set in *display_name* and *profile_image_uri*				host applications and the extension that added it.
friend_key	Generic data column for use by the extension to store information that is used by the extension only. This can for instance be URL to a web page that shall be opened when VIEW_EVENT_INTENT is received.	Optional	Text	Extension	LiveWare™ manager, host applications and the Extension that added it.

Table 4 Event table columns, data type, data insert responsibility and data readability

5.2 Add a source

The extension developer decides how to segment its event data, but all event data must be connected to a Source, otherwise the insert operation for event data will fail.

Setting the Source information in the *ContentProvider* should take place after the extension is successfully registered and before inserting event-data.

```
ContentValues values = new ContentValues();
Builder iconUriBuilder = new Uri.Builder()
    .scheme(ContentResolver.SCHEME_ANDROID_RESOURCE).authority(getPackageName()
    .appendPath(Integer.toString(R.drawable.icon));
values.put("name", "RSS News Feed");
values.putNull(“timeStamp”);
values.put(“enabled”, 1);
values.put("icon_uri1", iconUriBuilder.toString());
uri = cr.insert(Uri.parse(
    "content://com.sonyericsson.extras.liveware.aef.notification/source"),
    values);
```

5.3 End user extension configuration

Extensions may require configuration by the end user before they can fully function. For example, the end user might need to login to a service on the Internet before events can be retrieved. The extension is responsible for providing this configuration UI to be displayed.

Host applications have its own configuration UIs from within the configuration UIs of registered extensions can be reach. When registering to the LiveWare™ manager, there is a column in the extension table called *configurationActivity*. If your extension needs to be configured after it is registered, insert your Android *Activity* class name in this column.

```
...
String configName = new ComponentName(getPackageName(),
    RssPluginConfig.class.getName()).flattenToShortString();
values.put("<com.your.package.name.configurationActivity>", configName);
...
cr.insert(Uri.parse(
    "content://com.sonyericsson.extras.liveware.aef.registration/extensions"),
values);
```

When the user wishes to launch your configuration UI, the host application will launch the registered Activity.

Even though an extension may require end user configuration, it may still provide event data. However, this would be data that is not related to the user. For example, the extension may show the latest event data posted on the service or event data posted by users at the current location. This behaviour is encouraged as this would promote your extension and enable the user to "discover" more about the services provided by your extension.

5.4 Show the Detail View of an event

The event data supplied by an extension in the Event table may be a snapshot of the information and the user has limited possibilities to interact with the presented information on the accessory display.

The user may wish to see all details related to that event and react to it, for example mark it as a favourite, or reply. If your extension provides any interaction with the event in your application or on a website, listen for the VIEW_EVENT_INTENT Intent. This Intent is sent by the host applications when the user performs an action signalling the intention to view the event details.

A registered Source can define up to three actions. When viewing an Event and the user indicates they want to execute an action on it, a list of actions is shown. The list of actions is constructed from the actions registered in the specific Source. The action is just a String and it is up to the extension to handle the translation of that string accordingly.

The Intent (VIEW_EVENT_INTENT) contains intent data about the specific event data and the chosen action that will enable an extension to launch the detail view of that event.

5.5 Handle images

The location of images is represented as a string. Images may be stored locally on the device or on the SD card. The following URI schemes are supported:

android.resource://

content://

5.6 Limitations

With multiple applications feeding data to a single content provider in an open and free environment, there is a possibility that rogue extensions may be created and the end user unwittingly installs these.

A rogue extension may attempt to fill up the content provider with thrash event data or bogus data. From development perspective, there is no governance over the quality of extensions. Guarding against these situations completely in the implementation is difficult and ultimately end user intervention is required. However, it is possible for the implementation to set some limitations.

The Smart Extension API has various limits on the number of entries that may be added to the different tables in the content provider. The purpose is to prevent a rogue extension from filling up the content provider completely until the memory on the device runs out.

The limits have been chosen based on what appears to be reasonable use cases. The limitations and their reasons can be seen in Table 5 below:

Table name	Limits	Reason for design
Source	8	For a single extension APK to support more than 8 Sources, the configuration page will be quite complicated for the normal user to use.
Event	100 per source	When the number of Events for an extension reaches the limit the implementation will trigger a cleanup job that will remove the oldest Events.
Event.message	None/?	The host application may choose to put a limit on the content of an event.

Table 5 Limitations

5.7 Use cases

This is a normal Event flow with one host application:

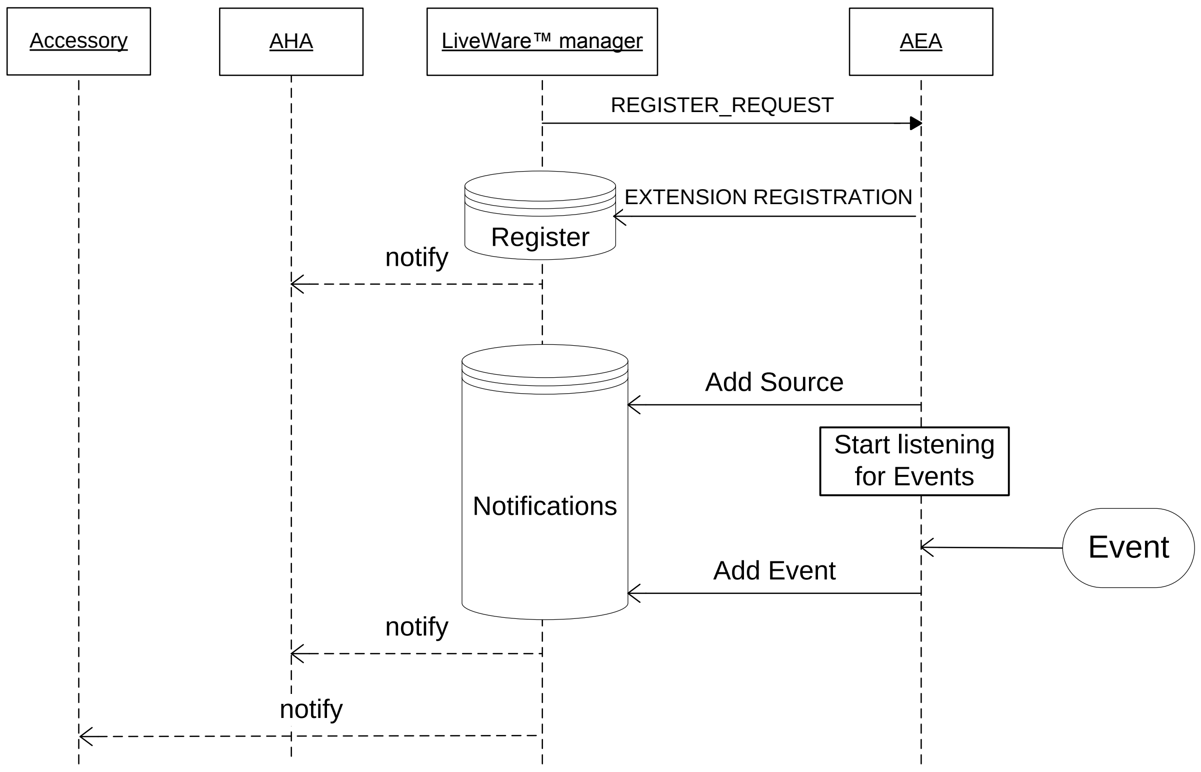

Figure 9 Normal event flow, one application

This is a normal Event flow with several host applications:

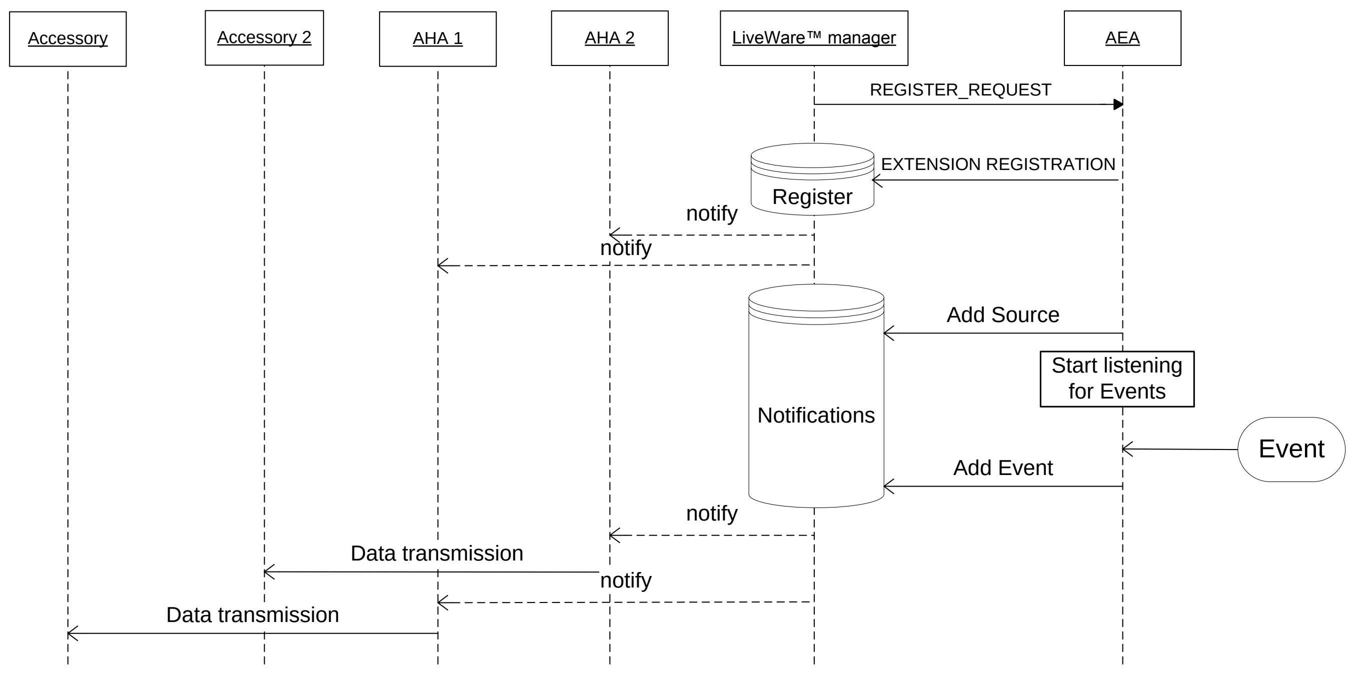

Figure 10 Normal event flow, several applications

6. The Control API

When writing extensions, the Control API is the most powerful component. In practice it means that the extension completely takes control over the accessory display.

Since the control extension takes control of the display, LEDs, vibrator, key and touch input, only one control extension at a time can run.

In order to use this API, an application must have registered itself properly to the LiveWare™ manager.

Detailed API definition can be found in the *Smart Extension API documentation*.

6.1 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.

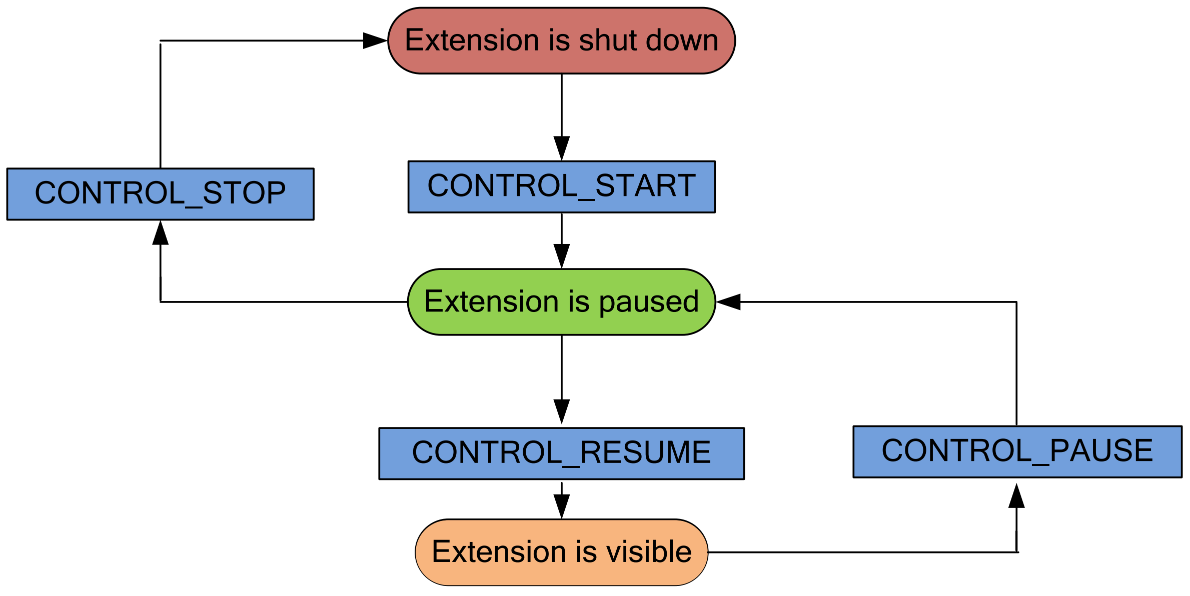

Figure 11 Control extension lifecycle

6.2 Control the display

Extensions implementing the Control API also have the possibility to control the state of the accessory display. The display can be controlled via the CONTROL_SET_SCREEN_STATE_INTENT.

As part of the Intent, you also need to specify what state you want to set. The following options are available:

- SCREEN_STATE_OFF
- SCREEN_STATE_DIM
- SCREEN_STATE_ON
- SCREEN_STATE_AUTO

When designing your extension it is important to make it consume a minimum of power, both on the phone and on the accessory.

The accessory has a much smaller battery then the phone, so use the mode SCREEN_STATE_ON with caution. When possible, let the host application take control of the display state. That way you don't have to bother about the power consumption on the accessory.

This can be done by setting the display state to SCREEN_STATE_AUTO.

When an extension starts, the display state is set to SCREEN_STATE_AUTO by default, which means that the host application controls the on/off/dim behavior. If the extension wants to control the display state, it must explicitly change the state.

If the extension controls the display state and it get a CONTROL_STOP_INTENT, meaning the extension is no longer running, the host application will automatically take over the display control.

Note that when in mode SCREEN_STATE_AUTO, the extension will receive a CONTROL_PAUSE_INTENT when display goes off and a CONTROL_RESUME_INTENT when the display goes on again.

6.3 Key events

An accessory might have several hardware keys. An extension will receive the key events when one of the keys is pressed.

The CONTROL_KEY_EVENT_INTENT is sent to the extension when a user presses a key on the accessory.

The Intent carries a few parameters, such as the timestamp of the event, the type of event (press, release and repeat) and also the key code. The accessory might have one or more keypads defined.

Extensions can look this up in the registration and capabilities API. Each key will have a unique key code for identification. Key codes can be found in a separate table and also in the product SDK.

6.4 Touch events

Certain accessories have a touch display. Extensions can find this out using the Registration and capabilities API. CONTROL_TOUCH_EVENT_INTENT is sent to the active the extension when a user taps the accessory display. The intent carries extra parameters with information of the package name of the host application, touch action (press, release or long press), time stamp, X and Y coordinates and the extension key.

6.5 Swipe events

Certain accessories have a touch display and support swipe events. Extensions can find this out using the Registration and capabilities API. CONTROL_SWIPE_INTENT is sent to the active extension when the user swipes over the display. The intent carries extra parameters with information of the package name of the host application, direction of the swipe (up, down, left or right), and the extension key.

6.6 Show content on the display

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.

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>), bitmap or as the URI of the image to be displayed.

6.6.1. Display an image from raw data

See `ControlExtension.showImage(final int resourceId)` and `ControlExtension.showBitmap(final Bitmap)`.

6.6.2. Display an image from URI

The following is used to display the URI of the image:

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

6.6.3. Clear the display

The Extension can at any time while controlling the accessory choose to clear the accessory display. This can be done via the CONTROL_CLEAR_DISPLAY_INTENT.

See `ControlExtension.clearDisplay()`

6.7 Control the LEDs

The accessory might have one or more LEDs that notify the user about events. The extension can read this via the Registration and capabilities API.

If the accessory has LEDs, the extension can control them via the Control API. The LEDs can be controlled via the CONTROL_LED_INTENT.

Note: The host application might overtake the control of the LED at any time if it wants to show some important notification to the user, for example when the accessory battery level is low. The extension is not affected by this. It will continue controlling all the other things.

The following parameters are available for tweaking when controlling the LEDs:

- EXTRA_LED_ID
- EXTRA_LED_COLOR
- EXTRA_ON_DURATION
- EXTRA_OFF_DURATION
- EXTRA_REPEATS

6.8 Control the vibrator

Some accessories have vibrators. The extension can find this out by checking the capabilities of a host application. If the accessory has a vibrator, it is controllable via the Control API, CONTROL_VIBRATE_INTENT.

The following parameters are available for tweaking when controlling the vibrator:

- EXTRA_ON_DURATION
- EXTRA_OFF_DURATION
- EXTRA_REPEATS

7. The Widget API

The main menu on some accessories supporting this API is built up with different widgets. Basically a widget is a live image that reflects the latest information from an extension. An extension implementing the Widget API will in some cases also implement one of the other APIs.

In order to use this API, an application must have registered itself properly to the LiveWare™ manager, see Registration and capabilities API.

Detailed API definition can be found in the *Smart Extension API documentation*.

7.1 Start and stop

When the accessory is turned on and connected to the phone, its main menu is built up with widget images. Before the accessory can build up the menu, it needs to have some initial images from the extensions. Therefore a host application sends a WIDGET_START_REFRESH_IMAGE_INTENT as soon as the accessory gets connected. It will also send this intent when it notices that a new widget extension has been registered. As a response to this, the extension must send an initial image, WIDGET_IMAGE_UPDATE_INTENT, so that the main menu can be built up.

The Intents in Android cannot be traced, so the extension does not know where the Intent comes from or which host application sent it. Therefore a host application, as part of the WIDGET_START_REFRESH_IMAGE_INTENT, sends its package name. This way, the extension will know who requested the widget image and also figure out the required image size.

7.2 Find out the widget image size

Before an extension sends the widget image to a host application, it has to figure out what size the image should be. This might vary between accessories, as some have a larger display. This information can be found using the Registration and capabilities API. Every host application will register its parameters into the Capabilities database. Note that this value is the maximum size in pixels that the widget can use. You can also use a smaller image. In that case the host application makes sure it is centred on the accessory display.

7.3 Refresh the widget image

The widget can at any time update its image, WIDGET_IMAGE_UPDATE_INTENT.

Note that the idea behind the widgets is that they should behave like lightweight control extensions. This means that the widget image should not be updated too often.

Every host application will have a recommended widget image refresh rate declared in the capabilities as different accessories might have different hardware capabilities. If the extension ignores these limitations, the host application will simply ignore the image update Intents.

7.4 User interaction

In order to allow the user to interact with the widget, the extension will get touch events that occur when its widget is in focus on the accessory screen, WIDGET_ONTOUCH_INTENT. This way, the extension receives user feedback and can refresh the widget image.

For example, in a media player control the initial image just shows a couple of buttons, play/pause, next, previous etc. When the user presses somewhere on the display, Intent is sent to the extension together with X and Y coordinates. Since the extension provided the initial image, it knows the exact layout of the buttons and can determine what button was pressed and take appropriate action (start the media player). It can also choose to update the widget image so that it reflects the latest state (instead of play button, it might show the pause button and the title of the song being played).

7.5 Use cases

The initial flow can be seen in Figure 12 and the result of user interaction in Figure 13 below.

The precondition is that the widget is in focus on the accessory.

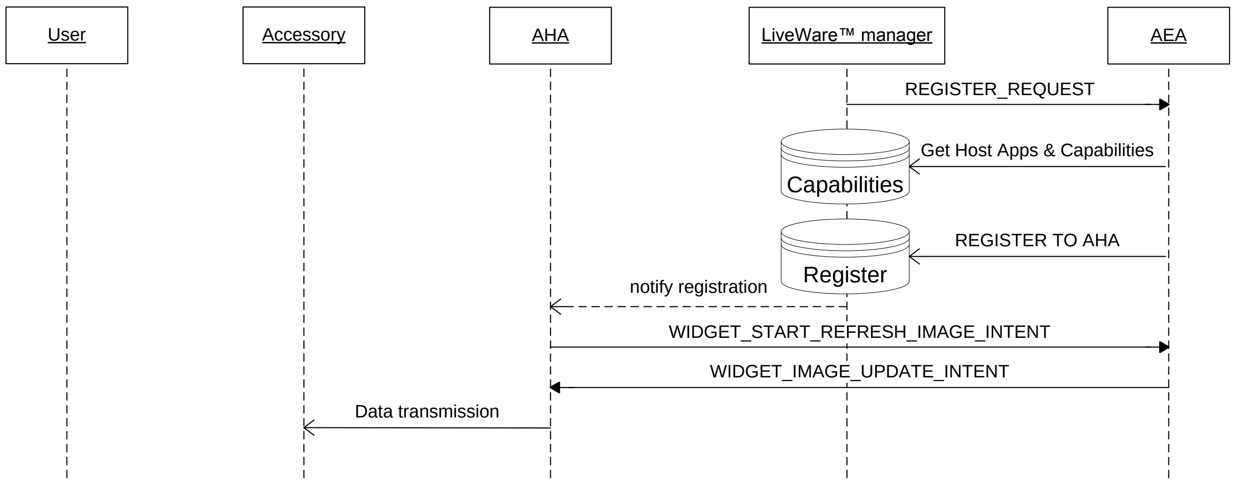

Figure 12 Initial flow

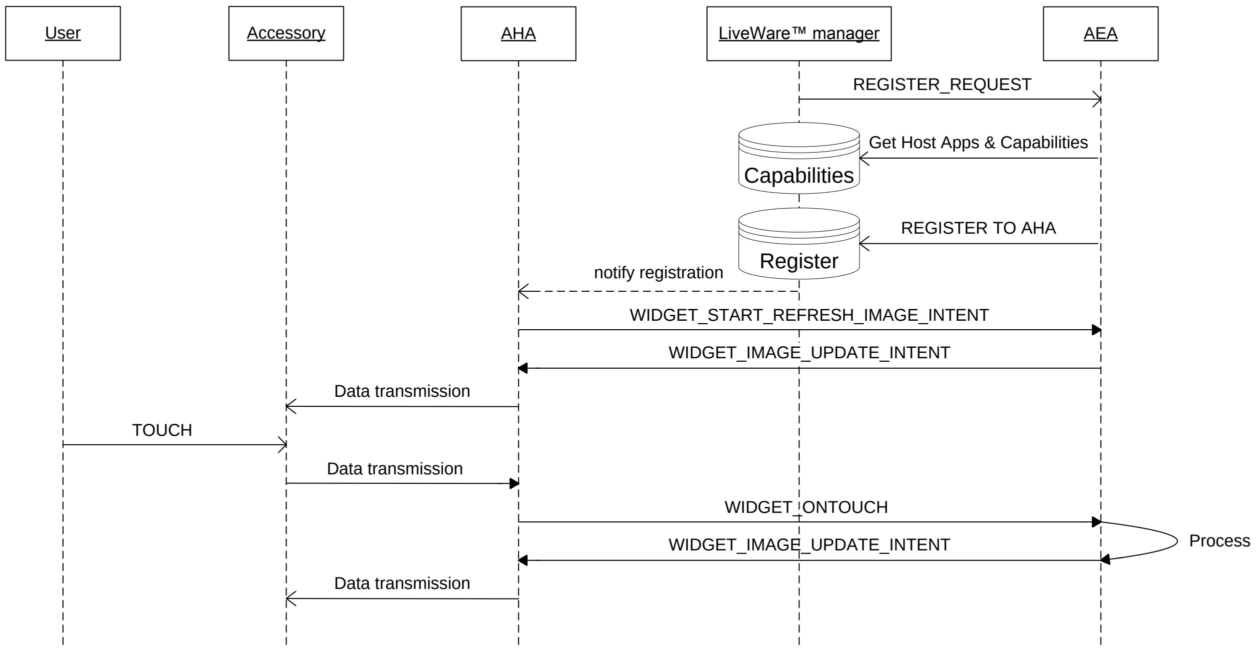

Figure 13 User interaction

8. The Sensor API

Some smart accessories have one or more sensors built in. The Sensor API is used to send accessory sensor data from a host application to an extension.

When a host application registers its capabilities, it will declare if it supports the Sensor API and what sensors it exposes. Extensions can use the capability API to query host applications for supported sensors.

Note that this API only defines the communication between host applications and the extension. The data transfer between the accessory and the host application is not part of this API.

In order to use this API, an application must have registered itself properly to the LiveWare™ manager, see Registration and capabilities API.

Detailed API definition can be found in the *Smart Extension API documentation*.

8.1 Register a listener

Sensor data is sent over a *LocalSocket*. In order to start communication the extension should setup a LocalServerSocket in listening mode and send the SENSOR_REGISTER_LISTENER_INTENT intent to a specific host application.

The extension must specify following extras in the intent:

- EXTRA_AEA_PACKAGE_NAME
- EXTRA_SENSOR_ID
- EXTRA_SENSOR_LOCAL_SERVER_SOCKET_NAME
- EXTRA_SENSOR_REQUESTED_RATE
- EXTRA_SENSOR_INTERRUPT_MODE

If multiple sensors are used, multiple *LocalServerSocket* objects must also be used since one sensor is bound to exactly one *LocalServerSocket*.

Some sensors support interrupt mode. They can be configured to only send sensor data when new values are available. The interrupt flag is part of the sensor data registration Intent.

Note that not all sensors support interrupt mode. This information will be part of the SDK for a certain accessory.

8.2 Unregister a listener

To unregister a listener the extension must send the SENSOR_UNREGISTER_LISTENER_INTENT.

The extension must specify the following extras in the intent:

- EXTRA_AEA_PACKAGE_NAME
- EXTRA_SENSOR_ID

8.3 Sensor data rates

Sensor data rates are predefined. When an extension registers a listener it also specifies what sample rate it wants to have.

The following rates are available:

- SENSOR_DELAY_NORMAL: Rate suitable for screen orientation changes.
- SENSOR_DELAY_UI: Rate suitable for user interface.
- SENSOR_DELAY_GAME: Rate suitable for games.
- SENSOR_DELAY_FASTEST: Get sensor data as fast as possible.

8.4 Sensor data format

The sensor data is sent from the host application over a *LocalSocket* and can be accessed through an *InputStream*.

Length	Accuracy	Timestamp	Length of sensor values	Value 1	Value N-1	Value N
4 Bytes	**4 Bytes**	**8 Bytes**	**4 Bytes**	**4 Bytes**	**4 Bytes**	**4 Bytes**

Figure 14 Sensor data

The data format is identical to the format of a SensorEvent in Android:

- **Byte 0 - 3**: Total length of the data package.
- **Byte 4 - 7**: Accuracy
- **Byte 8 - 15**: Timestamp. The time in nanosecond when the event fired.
- **Byte 16 - 19**: Length of sensor values in bytes.
- **Byte 20 - nn**: Sensor values. This should be interpreted as an array of float values; each float value is 4 bytes long.

The length and contents of the values array depends on which sensor type is being monitored.

8.5 Sensitive sensor data

Some accessories might have sensors that provide “sensitive” data. Host applications will declare this in its capabilities.

If an extension wishes to register a listener to a sensor that provides “sensitive” data, the host application will prompt the user about this. The user can then allow if a certain extension gets access to the data.

8.6 Sensor accuracy

Sensor accuracy is part of the sensor data received. Defined accuracy values:

SENSOR_STATUS_ACCURACY_HIGH	The sensor is reporting data with maximum accuracy.
SENSOR_STATUS_ACCURACY_MEDIUM	The sensor is reporting data with an average level of accuracy, calibrating with the environment may improve the reading.
SENSOR_STATUS_ACCURACY_LOW	The sensor is reporting with low accuracy, calibrating with the environment is needed.
SENSOR_STATUS_UNRELIABLE	The sensor data cannot be trusted, calibrating is needed or the environment doesn’t allow reading.

Table 6 Sensor accuracy

Recommendations: To achieve this on the phone you can get an idea by studying how the sample extensions use the SDK. There is also a good post here:

http://blogs.sonyericsson.com/wp/2010/08/23/android-tutorial-reducing-power-consumption-of-connected-apps/

8.7 Development

We recommend studying the SDK and its APIs and to use it extensively as it provides support for short lived services.

9. Additional information

Whitepapers for Sony Ericsson advanced accessories are available for download at the accessory homepage**.**

Extension SDK and its API's, sample extensions and LiveView Touch™ Emulator are available for download at http://developer.sonyericsson.com/.

Trademarks and acknowledgements