S60 Module Reference

Release 2.0.0 final

09 Feb 2010

Nokia

This is Python(R) for S60 version 2.0.0 final created by Nokia Corporation.

Abstract

This document is for Python for S60 Platform (Python for S60), which simplifies application development and provides a scripting solution for the Symbian C++ APIs.

CONTENTS

CHAPTER

ONE

Getting Started

Thank you for installing Python for S60 package.
Python for S60 is a powerful scripting language with an extensive standard library and easy-to-use APIs for S60 phone features, based on Python 2.5.4.

1.1 Installing Python runtime and its dependencies

Python runtime and other libraries are available in the folder **PyS60Dependencies** under the installation folder.

Python_2.0.0.sis - Python runtime

Runtime dependent package:
- **pips.sis** - OpenC PIPS library

Optional packages/libraries:
- **stdioserver.sis** - stdioserver to run python scripts packaged with console profile.
- **ssl.sis** - SSL library if you need SSL support in socket module.

Different variants of PythonScriptShell packages:
- **PythonScriptShell_2.0.0_high_capas.sis** - ScriptShell with high capabilities (Self-signed + Location + SwEvent + WriteDeviceData + ReadDeviceData)
- **PythonScriptShell_2.0.0_3_2.sis** - ScriptShell with Self-signed + Location capability
- **PythonScriptShell_2.0.0_3_0.sis** - ScriptShell with Self-signed capability set
- **PythonScriptShell_2.0.0_unsigned_devcert.sis** - ScriptShell with Developer certificate capabilities

- **Python_2.0.0_unsigned.sis** – Unsigned Python runtime component

1.2 Packaging a Sample Application

This section describes how to create a sis file from a Python script.

1.2.1 On a Windows host machine

The following steps provide the procedure for writing and packaging a helloworld script.

- Create a ”helloworld” script with the filename ”helloworld.py” containing `print "Hello World!"` code snippet.
- Click **Start - Programs - PythonForS60 2.0.0 ¿ PyS60 Application Packager**, the PyS60 application packager dialog box opens.
- Select the **Scriptfile** radio button and then, click the **Browse** button to select the helloworld.py script from the file Open dialog.
- Click the **Create** button to create the sis file with the current settings.
- Install the sis file **helloworld_v1_0_0.sis** created in the source directory of ”helloworld.py” file.

1.2.2 On a Linux or Mac host machine

The following steps provide the procedure for writing and packaging a helloworld script.

- Create a ”helloworld” script with the filename ”helloworld.py” containing `print "Hello World!"` code snippet.
- Using a command prompt, enter the directory which contains the ensymble.py and copy the ”helloworld.py” file here.
- Execute the command "`python ensymble.py py2sis helloworld.py`".
- The sis file would be generated in the current directory. Install this on an S60 device.
- Execute "`python ensymble.py py2sis --help`", for more info on py2sis options.

CHAPTER

TWO

Operating System Services and Information

2.1 `e32` — A Symbian OS related services package

The `e32` module offers Symbian OS related utilities that are not related to the UI and are not provided by the standard Python library modules.

2.1.1 Module Level Functions

The following free functions - functions that do not belong to any class - are defined in the `e32` module:

`ao_yield``()`
Yields to the active scheduler to have ready active objects with priority above normal scheduled for running. This has the effect of flushing the eventual pending UI events. Note that the UI callback code may be run in the context of the thread that performs an `ao_yield`. For information on active scheduler, see S60 SDK documentation [4].

`ao_sleep``(`*interval* [*, callback*]`)`
Sleeps for the given *interval* without blocking the active scheduler. When the optional *callback* is given, the call to `ao_sleep` returns immediately and the *callback* gets called after *interval*. See also Section 2.1.3, Ao_timer Type.

`ao_callgate``(`*wrapped_callable*`)`
Wraps *wrapped_callable* into returned callable object *callgate* that can be called in any thread. As a result of a call to *callgate*, *wrapped_callable* gets called in the context of the thread that originally created the callgate. Arguments can be given to the call. This is actually a simple wrapping of the Symbian active object facility.

`drive_list``()`
Returns a list of currently visible drives as a list of Unicode strings `'<driveletter>:'`

`file_copy``(`*target_name, source_name*`)`
Copies the file *source_name* to *target_name*. The names must be complete paths.

`in_emulator``()`
Returns `True` if running in an emulator, or `False` if running on a device.

`set_home_time``(`*time*`)`
Set the device's time to `time` (see Section **??**).

`pys60_version`
A string containing the version number of the Python for S60 and some additional information.

Example:

```
>>> import e32
>>> e32.pys60_version
'1.9.3 svn2793'
>>>
```

pys60_version_info
A tuple containing the five components of the Python for S60 version number: major, minor, micro, release tag, and serial. All values except release level are integers; the release tag is a string. A value other than `'final'` for the release tag signifies a development release. The `pys60_version_info` value corresponding to the Python for S60 version 1.9.3 is `(1, 9, 3, 'svn2793', 0)`.

Example:

```
>>> import e32
>>> e32.pys60_version_info
(1, 9, 3, 'svn2793', 0)
>>>
```

s60_version_info
Returns the S60 platform version of the device.

Example:

```
>>> import e32
>>> e32.s60_version_info
(3, 0)
>>>
```

is_ui_thread()
Returns `True` if the code that calls this function runs in the context of the UI thread; otherwise returns `False`.

start_exe(*filename, command* [,*wait*])
Launches the native Symbian OS executable *filename* (Unicode) and passes it the *command* string. When *wait* is set, the function synchronously waits for the exit of the executable and returns a value that describes the exit type. Possible values are `0` for normal exit and `2` for abnormal exit.

start_server(*filename*)
Starts the Python script in file *filename* (Unicode) as a server in its own process. Note that `appuifw` module is not available to a server script.

reset_inactivity()
Resets the timers since the user was last active. As a consequence, the device backlight is normally turned on when this function is invoked.

inactivity()
Returns the time in seconds since the user of the device was last active.

get_capabilities()
Returns tuple of capabilities granted to the application. Example:

```
>>> import e32
>>> e32.get_capabilities()
('ReadUserData','WriteUserData')
>>>
```

has_capabilities(*capability_list*)
Check if the application has all the capabilities that are passed in a list, 'capability_list'. Returns True if all the capabilities specified in the list are present, False otherwise.

Examples:

```
>>> import e32
>>> e32.has_capabilities(['Location','ReadUserData'])
False
>>> e32.has_capabilities(['Location','SwEvent'])
False
>>> e32.has_capabilities(['WriteUserData','ReadUserData'])
True
>>>
```

2.1.2 Ao_lock Type

class Ao_lock()
Creates an Ao_lock instance. A Symbian active object based synchronization service. This can be used in the main thread without blocking the handling of UI events. The application should not exit while a thread is waiting in Ao_lock. If Ao_lock.wait is called while another wait call is already in progress, an AssertionError is raised.

Instances of Ao_lock type have the following methods:

wait()
If the lock has already been signaled, returns immediately. Otherwise blocks in wait for the lock to be signaled. Only one waiter is allowed, so you should avoid recursive calls to this service. wait can only be called in the thread that created the lock object. During the wait, other Symbian-active objects are being served, so the UI will not freeze. This may result in the UI callback code being run in the context of the thread that is waiting in Ao_lock. This must be considered when designing the application logic.

signal()
Signals the lock. The waiter is released.

2.1.3 Ao_timer Type

The rationale for the Ao_timer type is that you cannot cancel a pending e32.ao_sleep. This is problematic if e.g. the user exits an application which is sleeping. In this case a panic would occur since the sleep is not cancelled - this is the reason you should avoid using e32.ao_sleep and instead use the Ao_timer with appropriate cancel calls if there is for example a possibility for the user to exit the application during a sleep.

class Ao_timer()
Creates an Ao_timer instance. A Symbian active object based sleeping service. This can be used in the main thread without blocking the handling of UI events. The application should not exit while a thread has a pending after call in Ao_timer. Only one after invocation can be pending at time for each instance of this type.

Instances of Ao_timer type have the following methods:

after(*interval* [,*callback*])
Sleeps for the given interval without blocking the active scheduler. When the optional callback is given, the call to after returns immediately and the callback gets called after interval.

cancel()
Cancels a pending after call.

2.2 sysinfo — Access to system information

The sysinfo module offers an API for checking the system information of a S60 mobile device.

The sysinfo module has the following functions:

active_profile()
Returns the current active profile as a string, which can be one of the following: 'general',

'silent','meeting', 'outdoor','pager', 'offline',, 'drive', or 'user <profile value>'.

battery()
Returns the current battery level. The value ranges from 0 (empty) to 100 (full) on a S60 mobile device. On the emulator the value is always 0.

Note: The returned value may be incorrect while the device is being charged.

display_twips()
Returns the width and height of the display in twips. For a definition of a twip, see Chapter 9, Terms and Abbreviations.

display_pixels()
Returns the width and height of the display in pixels.

free_drivespace()
Returns the amount of free space left on the drives in bytes, for example `{u'C:' 100}`. The keys in the dictionary are the drive letters followed by a colon (:).

imei()
Returns the IMEI code of the device as a Unicode string or, if running on the emulator, the hardcoded string `u'000000000000000'`.

max_ramdrive_size()
Returns the maximum size of the RAM drive on the device.

total_ram()
Returns the amount of RAM memory on the device.

free_ram()
Returns the amount of free RAM memory available on the device.

total_rom()
Returns the amount of read-only ROM memory on the device.

ring_type()
Returns the current ringing type as a string, which can be one of the following: `'normal'`, `'ascending'`,`'ring_once'`, `'beep'`, or `'silent'`.

os_version()
Returns the operating system version number of the device as a three element tuple (major version, minor version, build number). The elements are as follows[1]:

- The major version number, ranging from 0 to 127 inclusive
- The minor version number, ranging from 0 to 99 inclusive
- The build number, ranging from 0 to 32767 inclusive.

signal_bars()
Returns the current network signal strength ranging from 0 to 7, with 0 meaning no signal and 7 meaning a strong signal. If using an emulator, value 0 is always returned.

signal_dbm()
Returns the current network signal strength in dBm. This is available SDK 2.8 onwards. If using an emulator value 0 is always returned.

sw_version()
Returns the software version as a Unicode string. On the emulator, returns the hardcoded string u'emulator'. For example, a software version can be returned as `u'V 4.09.1 26-02-04 NHL-10 (c) NMP'`.

[1] Descriptions for these values are based on information found in S60 SDK documentation [4].

CHAPTER

THREE

User Interface and Graphics

3.1 `appuifw` — Interface to the S60 GUI framework

The `appuifw` module offers an interface to the S60 UI application framework. Figure 3.1 provides an overview of the Python for S60 environment for UI application programming.

Note: The services of this interface may only be used in the context of the main thread, that is, the initial thread of a UI application script.

3.1.1 Basics of appuifw Module

Figure 3.2 shows the layout of a S60 application UI in the normal screen mode and a summary of how it relates to the services available at the `appuifw` API. For alternative layouts, see Figure 3.3.

The main application window may be set up to be occupied by a UI control.

A multi-view application can show the different views as tabs in the navigation pane and react as the users navigate between tabs.

Dialogs always take precedence over the usual UI controls and appear on top of them.

UI controls are implemented as Python types. These types are available:

- `Text`
- `Listbox`
- `Canvas`

UI controls appear on the screen as soon as an instance of the corresponding Python type is set to the body field (*app.body*) of the current application UI.

`Form` is a versatile dialog implemented as a type.

The `Content_handler` type facilitates interfacing to other UI applications and common high-level UI components. It is based on the notion that designated handlers can reduce UI application interaction to operations on MIME-type content.

The following dialogs are implemented as functions:

- `note`
- `query`
- `multi_query`
- `selection_list`
- `multi_selection_list`

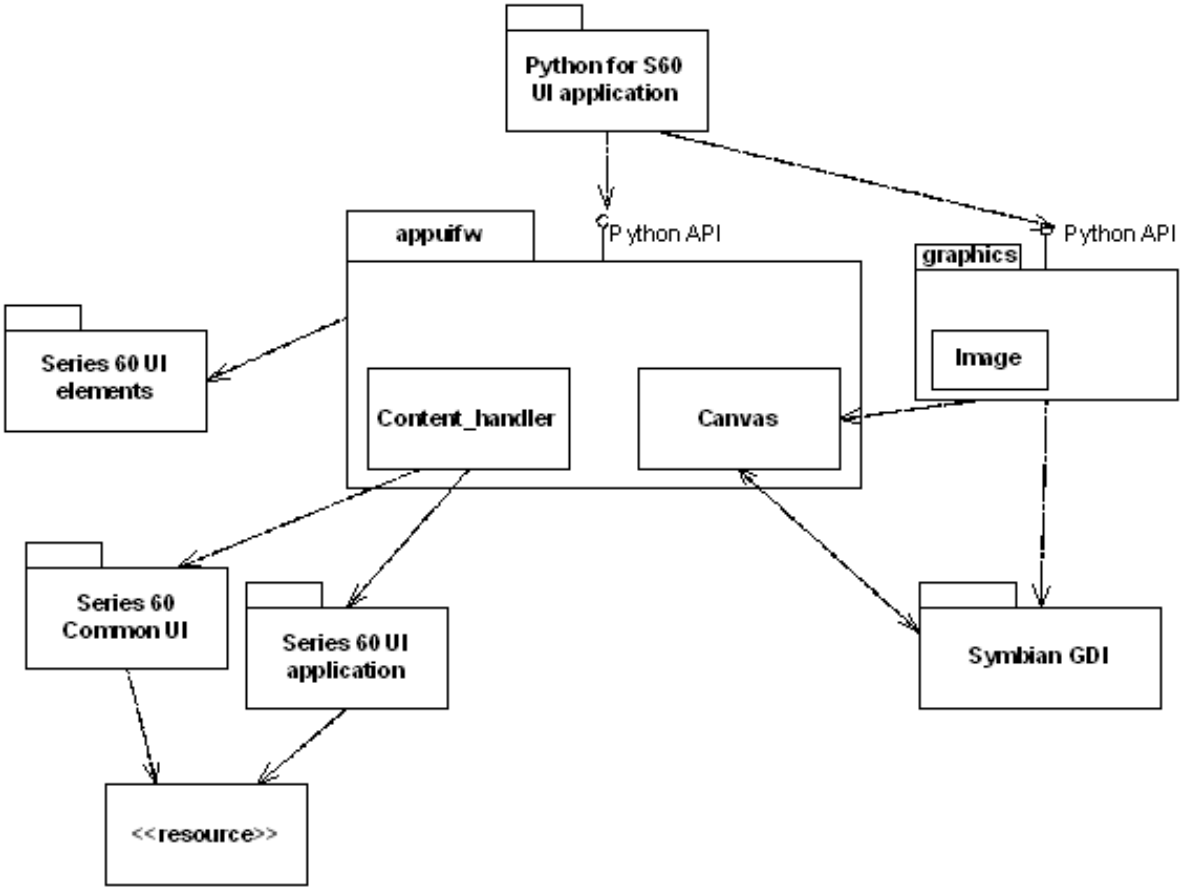

Figure 3.1: Python for S60 UI environment overview

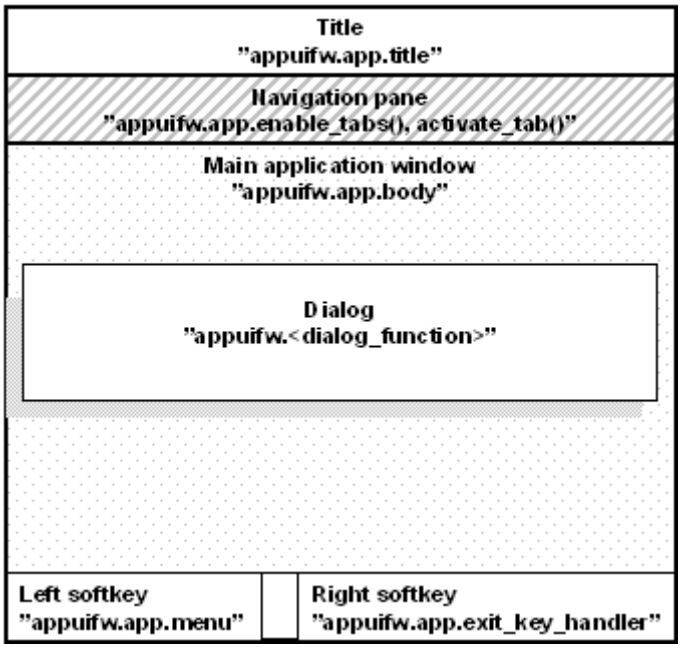

Figure 3.2: The different parts of the screen when using the 'normal' layout

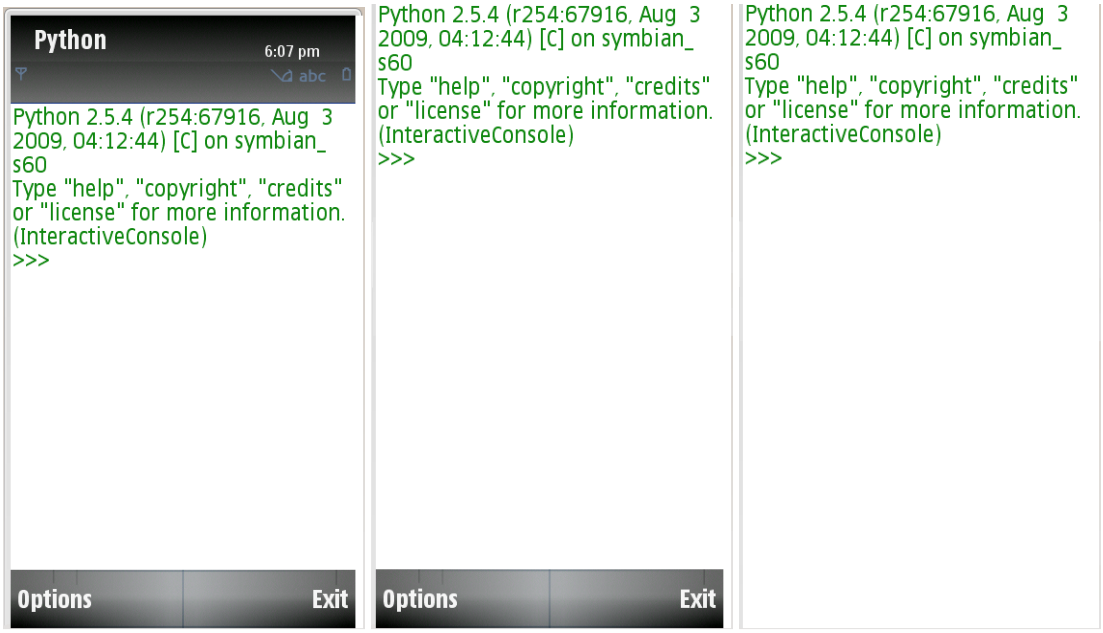

Figure 3.3: UI layouts. left: 'normal', middle: 'large', right: 'full'

- `popup_menu`

A dialog becomes visible as soon as the corresponding Python function has been called. The function returns with the eventual user input or information on the cancellation of the dialog. `Form` is an exception; it is shown when its `execute` method is called.

3.1.2 Softkeys

The softkeys are managed by the underlying S60 Platform. When no dialog is visible, the right softkey is bound to application exit and the left one represents an Options menu. Python for S60 offers an interface for manipulating the menu and for binding the Exit key to a Python-callable object (see Section 3.1.4).

The native code that implements a dialog also manages the softkeys of the dialog, typically OK and Cancel. When the user input needs to be validated before accepting it and dismissing the dialog, it is best to use `Form`.

3.1.3 Module Level Functions

The following free functions - functions that do not belong to any class - are defined in the `appuifw` module:

`available_fonts``()`
Returns a list (Unicode) of all fonts available in the device.

`touch_enabled``()`
Returns 'True' if the device supports touch input, 'False' otherwise.

`query``(`*label, type*[, *initial_value*]`)`
Performs a query with a single-field dialog. The prompt is set to *label*, and the type of the dialog is defined by *type*. The value of *type* can be any of the following strings:

- `'text'`
- `'code'`
- `'number'`
- `'date'`
- `'time'`

- `'query'`
- `'float'`

The type of the optional *initial_value* parameter and the returned input depend on the value of *type*:

- For text fields, (`'text'`, `'code'`) it is Unicode
- For number fields, it is numeric
- For date fields, it is seconds since epoch rounded down to the nearest local midnight

A simple confirmation query and time query take no initial value and return `True/None` and seconds since local midnight, correspondingly. All queries return `None` if the users cancel the dialog.

For `'float'` query the *initial_value* setting has no effect.

multi_query (*label_1, label_2*)
A two-field text (Unicode) input dialog. Returns the input values as a 2-tuple. Returns `None` if the users cancel the dialog.

note (*text*[, *type*[, *global*]])
Displays a note dialog of the chosen type with *text* (Unicode). The default value for *type* is `'info'`, which is automatically used if *type* is not set. *type* can be one of the following strings: `'error'`, `'info'` or `'conf'`.

If *global* (integer) is any other value than zero a global note is displayed. A global note is displayed even if the Python application calling this function is in background. The same set of *type*s is supported as in standard note.

popup_menu (*list*[, *label*])
A pop-up menu style dialog. *list* representing the menu contents can be a list of Unicode strings or a list of Unicode string pairs (tuples). The resulting dialog list is then a single-style or a double-style list. A single-style list is shown in full; whereas a double-style list shows the items one at a time. Returns `None` if the user cancels the operation.

selection_list (*choices*[, *search_field=0*])
Executes a dialog that allows the users to select a list item and returns the *index* of the chosen item, or `None` if the selection is cancelled by the users. *choices* is a list of Unicode strings. *search_field* is `0` (disabled) by default and is optional. Setting it to `1` enables a search field (find pane) that facilitates searching for items in long lists. If enabled, the search field appears after you press a letter key.

multi_selection_list (*choices*[, *style='checkbox', search_field=0*])
Executes a dialog that allows the users to select multiple list items. Returns a tuple of indexes (a pair of Unicode strings) of the chosen items, or empty tuple if the no selection is made by the users. *choices* is a list of Unicode strings. *style* is an optional string; the default value being `'checkbox'`. If `'checkbox'` is given, the list will be a checkbox list, where empty checkboxes indicate what items can be marked. The other possible value that can be set for *style* is `'checkmark'`. If `'checkmark'` is given, the list will be a markable list, which lists items but does not indicate specifically that items can be selected. To select items on a markable list, use the *'Options'* that has Mark/Unmark or the Edit key to select an item and the Navigation key to browse the list. For example views on checkbox and markable lists, see Figure 3.4. *search_field* is `0` (disabled) by default and is optional. Setting it to `1` enables a search field (find pane) that facilitates searching for items in long lists. If enabled, the search field is always visible with checkbox lists; with markable lists it appears by pressing a letter key.

Example:

```
tuple = appuifw.multi_selection_list([u'Harry', u'Ron', u'Hermione', u'Voldemort'], styl
```

3.1.4 Application Type

A single implicit instance of this type always exists when `appuifw` module is present and can be referred to with the name `app`. New instances cannot be created by a Python program.

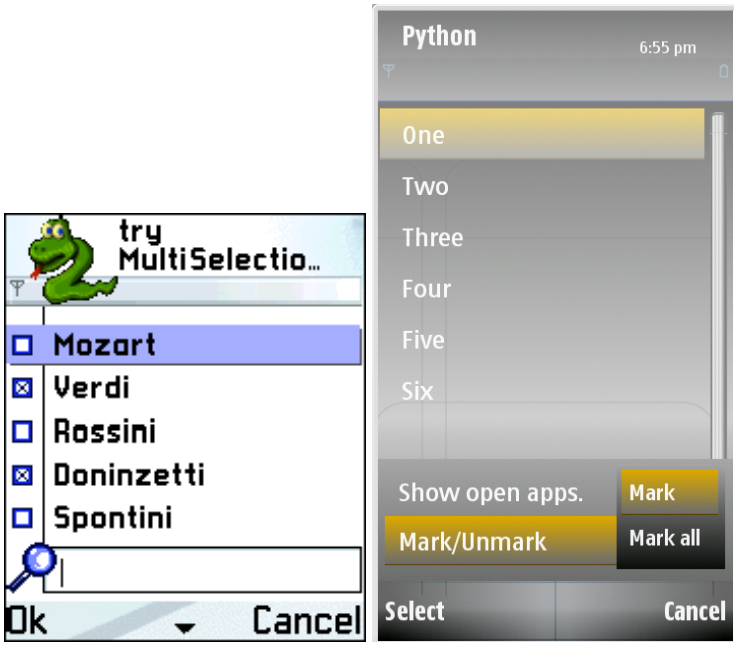

Figure 3.4: Examples of a checkbox list (left) and a markable list (right)

class Application

Instances of `Application` type have the following attributes:

body

The UI control that is visible in the application's main window. Currently either `Text`, a `Listbox` object, `Canvas`, or `None`.

directional_pad

A boolean flag which controls the appearance of a virtual 4-way directional pad that is displayed either at the bottom of the screen or on the right hand corner depending on the orientation when using a Canvas. This is enabled by default on devices that do not have a physical left and right soft key. This value is ignored on other devices, hence setting it to either True or False will have no effect.

Set it to `True` to enable 4-way directional pad and `False` to disable it.

exit_key_handler

A callable object that is called when the user presses the Exit softkey. Setting `exit_key_handler` to `None` sets it back to the default value.

focus

A callable object that is called with integer as parameter (0 = focus lost, 1 = focus regained) when the application receives focus or it is switched to background. Focus is received e.g. when the application is switched from background to foreground or when the focus is regained from screensaver. Similarly when the screensaver is displayed, focus is lost.

Examples:

```
>>> import appuifw
>>> def cb(fg):
...   if(fg):
...     print "foreground"
...   else:
...     print "background"
...
>>> appuifw.app.focus=cb
>>> # switch to background, following text is printed from callback:
>>> background
>>> # switch to foreground, following text is printed from callback:
>>> foreground
```

Note: An improper callback can cause adverse effects. If you, for example, define a callback which takes no parameters you will receive never-ending `TypeError` exceptions on the Nokia 6600.

`menu`
This is a list of the following kinds of items:

- `(title, callback)` which creates a regular menu item
- `(title, ((title, callback)[...]))` which creates a submenu

title (Unicode) is the name of the item and *callback* the associated callable object. The maximum allowed number of items in a menu, or items in a submenu, or submenus in a menu is 30.
Example:

```
appuifw.app.menu = [(u"Item 1", item1),
                    (u"Submenu 1",
                        ((u"Subitem 1", subitem1),
                         (u"Subitem 2", subitem2)))]
```

`orientation`
The orientation of the application. The orientation of the application can be one of the following values: `'automatic'` (this is the default value), `'portrait'` or `'landscape'`.

`screen`
The screen area used by an application. See Figure 3.3 for example screens. The appearance of the application on the screen can be affected by setting one of the following values: `'normal'`, `'large'` and `'full'`.
Examples:

```
appuifw.app.screen='normal'     # normal screen with title pane and softkey labels
appuifw.app.screen='large'      # only softkey labels visible
appuifw.app.screen='full'       # full screen mode on all devices
```

`title`
The title of the application that is visible in the application's title pane. Must be Unicode.

`track_allocations`
Set this to true if the interpreter should track all memory allocations and then free the memory which was not explicitly released before application exit. The default value of this attribute is true. As a consequence if there are any memory leaks in the 3rd party extension modules they will be released at the end. To check if there are memory leaks(for debugging purposes) the following approach can be used :

```
appuifw.app.track_allocations = false

import my_extension
my_extension.do_something()

appuifw.app.track_allocations = true
```

If the extension leaks memory then it will be reported at application exit.

Instances of `Application` type have the following methods:

`activate_tab`(*index*)
Activates the tab *index* counting from zero.

`full_name`()
Returns the full name, in Unicode, of the native application in whose context the current Python interpreter session runs.

`layout`(*layout_id*)
Returns as a tuple the size and the position of the requested `layout_id`. The logical layouts are outlined partly in Figure 3.2. The position is given from the top left corner. The `layout_id` can be one of the constants defined in module `appuifw`[1]:

[1] Descriptions of the values are from the S60 SDK documentation [4].

EScreen
Screen.

EApplicationWindow
Window that fills the entire screen.

EStatusPane
Indicates common components for most of the applications.

EMainPane
The application main pane is used in all the applications.

EControlPane
Control pane.

ESignalPane
The signal pane is used to indicate signal strength.

EContextPane
The context pane is used to indicate an active application.

ETitlePane
Used to indicate the subject or the name of the main pane content.

EBatteryPane
The battery pane is used to indicate battery strength.

EUniversalIndicatorPane
The universal indicator pane is used to indicate items that require the user's attention while browsing applications.

ENaviPane
The navi pane is used to indicate navigation within an application, to provide context sensitive information to the user while entering or editing data, or to show additional information.

EFindPane
A fixed find pane is used with lists instead of the find pop-up window.

EWallpaperPane
Wallpaper pane.

EIndicatorPane
The universal indicator pane is used to indicate items that require the user's attention while browsing applications.

EAColumn
Used generally to display small sized graphics or heading texts.

EBColumn
Used generally to display large sized icons or heading texts.

ECColumn
Used generally to display data entered by the user. Overlaps with the D column.

EDColumn
Used generally to display additional icons. Overlaps with the C column.

EStaconTop
Top part of status and control panes in landscape layout.

EStaconBottom
Bottom part of status and control panes in landscape layout.

EStatusPaneBottom
Bottom part of status pane in landscape layout.

EControlPaneBottom
Bottom part of control pane in landscape layout.

EControlPaneTop
Top part of control pane in landscape layout.

EStatusPaneTop
Top part of status pane in landscape layout.

Example:

```
>>> import appuifw
>>> appuifw.app.layout(appuifw.EMainPane)
((176, 144), (0, 44))
>>> # size and position (x, y) of the main pane in Nokia N70
```

set_exit()
Requests a graceful exit from the application as soon as the current script execution returns.

set_tabs(*tab_texts*[,*callback=None*])
Sets tabs with given names on them in the navigation bar; *tab_texts* is a list of Unicode strings. When the users navigate between tabs, *callback* gets called with the index of the active tab as an argument. Tabs can be disabled by giving an empty or one-item *tab_texts* list.

uid()
Returns the UID, in Unicode, of the native application in whose context the current Python interpreter session runs.

3.1.5 Form Type

`Form` implements a dynamically configurable, editable multi-field dialog. `Form` caters for advanced dialog use cases with requirements such as free selectability of the combination of fields, possibility of validating the user input, and automatically producing the contents of some dialog fields before allowing the closing of the dialog.

class Form(*fields*[, *flags=0*])
Creates a `Form` instance. *fields* is a list of *field descriptors*: `(label, type`[`, value`]`)` where

label is a Unicode string

type is one of the following strings: `'text'`, `'number'`, `'date'`, `'time'`, `'combo'` or `'float'`

value, depending on *type*: Unicode string, numeric, float (seconds since Unix epoch rounded down to the nearest local midnight), float (seconds since local midnight), `([choice_label ...], index)` of float. For `'float'` *type* the initial value setting might not be shown in the UI.

`Form` can also be configured and populated after construction. The configuration flags are visible as an attribute. `Form` implements the list protocol that can be used for setting the form fields, as well as obtaining their values after the dialog has been executed.

Instances of `Form` type have the following attributes:

flags
This attribute holds the values of the various configuration flags. Currently supported flags are:

FFormEditModeOnly
When this flag is set, the form remains in edit mode while `execute` runs.

FFormViewModeOnly
When this flag is set, the form cannot be edited at all.

FFormAutoLabelEdit
This flag enables support for allowing the end-users to edit the labels of the form fields.

FFormAutoFormEdit
This flag enables automatic support for allowing the end-users to add and delete the form fields. Note that this is an experimental feature and is not guaranteed to work with all SDK versions.

FFormDoubleSpaced
When this flag is set, double-spaced layout is applied when the form is executed: one field takes two lines, as the label and the value field are on different lines.

menu
A list of `(title, callback)` pairs, where each pair describes an item in the form's menu bar that is active while the dialog is being executed. *title* (Unicode) is the name of the item and *callback* the associated callable object.

save_hook
This attribute can be set to a callable object that receives one argument and returns a Boolean value. It gets

called every time the users want to save the contents of an executing `Form` dialog. A candidate list for new form content - a list representing the currently visible state of the UI - is given as an argument. The list can be modified by `save_hook`. If `save_hook` returns `True`, the candidate list is set as the new contents of the form. Otherwise, the form UI is reset to reflect the field list contained in `Form` object.

Instances of `Form` type have the following methods:

`execute``()`
Executes the dialog by making it visible on the UI.

`insert``(`*index, field_descriptor*`)`
Inserts the field descriptor into the `Form` before the given *index*.

`pop``()`
Removes the last field descriptor from the `Form` and returns it.

`length``()`
the number of field descriptors in the form.

The subscript notation `f[i]` can be used to access or modify the i-th element of the form `f`. Same limitations as discussed above in the context of the flag `FFormAutoFormEdit` apply to modifying a form while it is executing. The ability to change the schema of a form while it is executing is an experimental feature.

3.1.6 Text Type

`Text` is a text editor UI control. For examples on the options available with `Text`, see Figure 3.5.

Figure 3.5: Examples of the options available for Text type

Instances of `Text` type have the following attributes:

`color`
The color of the text. `color` supports the same color representation models as the `graphics` module. For the supported color representation models, see Section 3.3.

`focus`
A Boolean attribute that indicates the focus state of the control. Editor control also takes the ownership of the navigation bar, and this feature is needed to enable the usage of this control in applications that use the navigation bar - for example, navigation tabs.

`font`
The font of the text. There are two possible ways to set this attribute:

- Using a supported Unicode font, for example `u"Latin12"`. Trying to set a font which is not supported by the device has no effect. A list of supported fonts can be retrieved by using `appuifw.available_fonts`.

  Example, setting font:

```
t = appuifw.Text()
t.font = u"albi17b" # sets font to Albi 17 bold
t.font = u"LatinPlain12" # sets font to Latin Plain 12
```

- Using one of the default device fonts that are associated with the following labels (plain strings): `'annotation'`, `'title'`, `'legend'`, `'symbol'`, `'dense'`, `'normal'`.

  Example, setting font:

```
t.font = "title" # sets font to the one used in titles
```

  Example, checking the currently set font:

```
unicodeFont = t.font
```

The attribute value retrieved is always a Unicode string. If the font has been set with a label, for example, `'title'`, the attribute will retrieve the font associated with that label.

highlight_color

The highlight color of the text. `highlight_color` supports the same color representation models as the `graphics` module. For the supported color representation models, see Section 3.3.

style

The style of the text. The flags for this attribute are defined in the `appuifw` module. These flags can be combined by using the binary operator `|`. The flags can be divided into two types: text style and text highlight. Text style flags can be freely combined with each other. However, one or more text style flags can be combined with only one text highlight flag. The flags are:

Text style:

STYLE_BOLD
Enables bold text.

STYLE_UNDERLINE
Enables underlined text.

STYLE_ITALIC
Enables italic text.

STYLE_STRIKETHROUGH
Enables strikethrough.

Text highlight:

HIGHLIGHT_STANDARD
Enables standard highlight.

HIGHLIGHT_ROUNDED
Enables rounded highlight.

HIGHLIGHT_SHADOW
Enables shadow highlight.

Only one highlight is allowed to be used at once. Therefore, it is possible to combine only one highlight with one or more text styles.

Examples:

```
t = appuifw.Text()

# These and other similar values and combinations are valid:
t.style = appuifw.STYLE_BOLD
t.style = appuifw.STYLE_UNDERLINE
t.style = appuifw.STYLE_ITALIC
t.style = appuifw.STYLE_STRIKETHROUGH
t.style = (appuifw.STYLE_BOLD|
   appuifw.STYLE_ITALIC|
   appuifw.STYLE_UNDERLINE)

# These values are valid:
t.style = appuifw.HIGHLIGHT_STANDARD
t.style = appuifw.HIGHLIGHT_ROUNDED
t.style = appuifw.HIGHLIGHT_SHADOW

# This combination is NOT valid:
# Invalid code, do not try!
t.style = (appuifw.HIGHLIGHT_SHADOW|appuifw.HIGHLIGHT_ROUNDED)
```

Instances of `Text` type have the following methods:

add(*text*)
Inserts the Unicode string *text* to the current cursor position.

bind(*event_code, callback*)
Binds the callable Python object *callback* to event *event_code*. The key codes are defined in the `key_codes` library module. The call `bind(event_code, None)` clears an existing binding. In the current implementation the event is always passed also to the underlying native UI control.

clear()
Clears the editor.

delete([*pos=0, length=len()*])
Deletes *length* characters of the text held by the editor control, starting from the position *pos*.

get_pos()
Returns the current cursor position.

len()
Returns the length of the text string held by the editor control.

get([*pos=0, length=len()*])
Retrieves `length` characters of the text held by the editor control, starting from the position *pos*.

set(*text*)
Sets the text content of the editor control to Unicode string *text*.

set_pos(*cursor_pos*)
Sets the cursor to *cursor_pos*.

3.1.7 Listbox Type

An instance of this UI control type is visible as a listbox, also known as a list in Symbian, that can be configured to be a single-line item or a double-item listbox. Figure 3.6 shows a single-line item Listbox with icons. For more information on the MBM and MIF formats, see Section 3.1.8.

class Listbox(*list, callback*)
Creates a `Listbox` instance. A callable object *callback* gets called when a listbox selection has been made. `list` defines the content of the listbox and can be one of the following:

- A normal (single-line item) listbox: a list of Unicode strings, for example `[unicode_string item1, unicode_string item2]`
- A double-item listbox: a two-element tuple of Unicode strings , for example `[(unicode_string`

Figure 3.6: Listbox with icons

```
item1, unicode_string item1description), (unicode_string item2,
unicode_string item2description)]
```

- A normal (single-line item) listbox with graphics: a two-element tuple consisting of a Unicode string and an `Icon` object, for example `[(unicode_string item1, icon1), (unicode_string item2, icon2)]`.
- A double-item listbox with graphics: a three-element tuple consisting of two Unicode strings and one `Icon` object, for example `[(unicode_string item1, unicode_string item1description, icon1), (unicode_string item2, unicode_string item2description, icon2)]`

Example: To produce a normal (single-line item) listbox with graphics:

```
icon1 = appuifw.Icon(u"z:\\resource\\apps\\avkon2.mbm", 28, 29)
icon2 = appuifw.Icon(u"z:\\resource\\apps\\avkon2.mbm", 40, 41)
entries = [(u"Signal", icon1),
           (u"Battery", icon2)]
lb = appuifw.Listbox(entries, lbox_observe)
```

Note: Known issue: Using this widget in large/full screen mode results in an unrefreshed area at the bottom of the screen.

Instances of `Listbox` type have the following methods and properties:

`bind`(*event_code, callback*)
Binds the callable Python object *callback* to event *event_code*. The key codes are defined in the `key_codes` library module. The call `bind(event_code, None)` clears an existing binding. In the current implementation the event is always passed also to the underlying native UI control.

`current`()
Returns the currently selected item's index in the `Listbox`.

`set_list`(*list*[, *current*])
Sets the `Listbox` content to a list of Unicode strings or a list of tuples of Unicode strings. The accepted structures of *list* are the same as in the `Listbox` constructor. The optional argument *current* is the index of the focused list item.

`size`
The size of the `Listbox` as a tuple (width, height) - Read only.

`position`
The coordinates (as a tuple) of the top left corner of the `Listbox` - Read only.

3.1.8 Icon Type

An instance of `Icon` type encapsulates an icon to be used together with a `Listbox` instance. Note that currently `Icon` can only be used with `Listbox` (see Section 3.1.7).

MBM is the native Symbian OS format used for pictures. It is a compressed file format where the files can contain several bitmaps and can be referred to by a number. An `.mbg` file is the header file usually associated with an `.mbm` file, which includes symbolic definitions for each bitmap in the file. For example, an 'avkon.mbm' file has an associated index file called 'avkon.mbg', which is included in S60 SDKs. For more information on the MBM format and the bitmap converter tool, see [4] and search the topics with the key term "How to provide Icons"; this topic also points you to the Bitmap Converter tool that can be used for converting bitmaps into the MBM format.

class `Icon` (*filename, bitmap, bitmapMask*)
Creates an icon. *filename* is a Unicode file name and must include the whole path. Note that MBM is the only file formats supported. *bitmap* and *bitmapMask* are integers that represent the index of the icon and icon mask inside that file respectively.

Example: The following builds an icon with the standard signal symbol:

```
icon = appuifw.Icon(u"z:\\resource\\apps\\avkon2.mbm", 28, 29)
```

3.1.9 Content_handler Type

An instance of `Content_handler` handles data content by its MIME type.

class `Content_handler` ([*callback*])
Creates a `Content_handler` instance. A Content_handler handles data content by its MIME type. The optional *callback* is called when the embedded handler application started with the `open` method finishes.

Instances of `Content_handler` type have the following methods:

`open` (*filename*)
Opens the file *filename* (Unicode) in its handler application if one has been registered for the particular MIME type. The handler application is embedded in the caller's thread. The call to this function returns immediately. When the handler application finishes, the *callback* that was given to the `Content_handler` constructor is called.

`open_standalone` (*filename*)
Opens the file *filename* (Unicode) in its handler application if one has been registered for the particular MIME type. The handler application is started in its own process. The call to this function returns immediately. Note that *callback* is not called for applications started with this method.

3.1.10 Canvas Type

`Canvas` is a UI control that provides a drawable area on the screen and support for handling raw key events. `Canvas` supports the standard drawing methods that are documented in Section 3.3.

class `Canvas` ([*redraw_callback=None, event_callback=None, resize_callback=None*])
Constructs a `Canvas`. The optional parameters are callbacks that are called when specific events occur.

Note: Watch out for cyclic references here. For example, if the callbacks are methods of an object that holds a reference to the `Canvas`, a reference cycle is formed that must be broken at cleanup time or the `Canvas` will not be freed.

redraw_callback is called whenever a part of the `Canvas` has been obscured by something, is then revealed, and needs to be redrawn. This can typically happen, for example, when the user switches away from the Python application and back again, or after displaying a pop-up menu. The callback takes as its argument a four-element tuple that contains the top-left and the bottom-right corner of the area that needs to be redrawn. In many cases redrawing the whole `Canvas` is a reasonable option.

event_callback is called whenever a raw key event is received or when a pointer event occurs(only on touch input supported devices). There are three kinds of key events: `EEventKeyDown`, `EEventKey`, and `EEventKeyUp`. When a user presses a key down, events `EEventKeyDown` and `EEventKey` are generated. When the key is released, an `EEventKeyUp` event is generated.

Pointer events are generated by touch input supported devices. When the screen is touched the `EButton1Down` event is generated, `EDrag` while the finger/stylus is dragged across the screen and then `EButton1Up` when the finger/stylus is lifted.

The argument to the *event_callback* is a dictionary that contains the following data:

For key events:

- `'type'`: one of `EEventKeyDown`, `EEventKey`, or `EEventKeyUp`
- `'keycode'`: the keycode of the key
- `'scancode'`: the scancode of the key
- `'modifiers'`: the modifiers that apply to this key event

For pointer events:

- `'type'`: one of the several pointer events - `EButton1Up`, `EButton1Down`, `EDrag` etc..
- `'modifiers'`: the modifiers that apply to this pointer event
- `'pos'`: A tuple containing the x-y pointer co-ordinates

Each key on the keyboard has one or more scancodes and zero or more keycodes associated with it. A scancode represents the physical key itself and a keycode is the result of state-related operating system defined processing done on the key. For keys that correspond to a symbol in the current character set of the phone, the keycode is equal to the code of the corresponding symbol in that character set. For example, if you are using the Nokia Wireless Keyboard (SU-8W), pressing the key A will always produce the scancode 65 (ASCII code for an upper case A), but the keycode could be either 65 or 91 (ASCII code for a lower case A) depending on whether or not the Shift key is pressed or Caps Lock is active.

The `key_codes` module contains definitions for the keycodes and scancodes. See Figure 3.7 for the codes of the most common keys on the phone keypad.

Some keys are handled in a special way:

- A short press of the Edit key causes it to stay down, meaning that no `EEventKeyUp` event is sent. The event is only sent after a long press.
- Detecting presses of the Voice tags key or the Power key is not supported.
- If the right softkey is pressed, the `appuifw.app.exit_key_handler` callback is always executed.

There is no way to prevent the standard action of the Hang-up key, the Menu key, the Power key or the Voice tags key from taking place.

resize_callback is called when screen size is changed when the `Canvas` rect size has been changed. The callback takes as its argument a two-element tuple that contains the new clientRect width and height.

Instances of `Canvas` type have the following methods:

`bind`(*pointer_event, callable*[, *((x1, y1), (x2, y2))*])

This method can be used to listen to specific pointer events. The *pointer_event* argument can be any one of the pointer events listed in the `key_codes` module.

The most common pointer events are:

- `EButton1Down` - Pen down event
- `EButton1Up` - Pen up event
- `EDrag` - Drag event (This event is only received when button1 is down)
- `ESwitchOn` - Switch on event caused by a screen tap.

Key	Keycode	Scancode
1.	EKeyLeftSoftkey	EScancodeLeftSoftkey
2.	EKeyYes	EScancodeYes
3.	EKeyMenu	EScancodeMenu
4.	EKey0...9	EScancode0...9
5.	EKeyStar	EScancodeStar
6.	EKeyLeftArrow	EScancodeLeftArrow
7.	EKeyUpArrow	EScancodeUpArrow
8.	EKeySelect	EScancodeSelect
9.	EKeyRightArrow	EScancodeRightArrow
10.	EKeyDownArrow	EScancodeDownArrow
11.	EKeyRightSoftkey	EScancodeRightSoftkey
12.	EKeyNo	EScancodeNo
13.	EKeyBackspace	EScancodeBackspace
14.	EKeyEdit	EScancodeEdit
15.	EKeyHash	EScancodeHash

Figure 3.7: Keycodes and scancodes for phone keys usable from Python applications

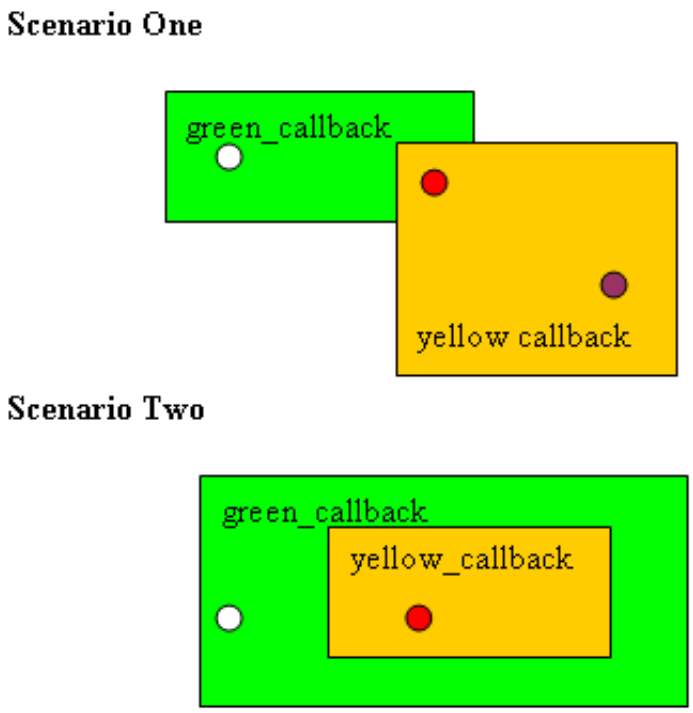

Figure 3.8: Canvas bind scenarios

callable is called when the pointer_event and the co-ordinate (if specified) criterion matches.

((x1, y1), (x2, y2)) is an optional argument that can be passed to specify the screen area to monitor for any specific pointer event. The two co-ordinate tuple corresponds to the top-left and bottom-right points. This argument will be ignored if the event is not a pointer event.

There are several ways in which bind can be used:

- `my_canv.bind(EButton1Up, callback)` - The callback is called when EButton1Up event is generated anywhere in the canvas.
- `my_canv.bind(EButton1Up, green_callback, ((x1, y1), (x2, y2)))` - The callback is called when the EButton1Up pointer event occurs inside the screen area specified.
- `my_canv.bind(EButton1Up, yellow_callback, ((x3, y3), (x4, y4)))` - Registers another callback for a different region but for the same pointer event. When two screen areas overlap, the callback registered last will be called when pointer events occur in the intersected region.
- `my_canv.bind(EButton1Up, callback3, ((x1, y1), (x2, y2)))` - If the pointer event and the screen area to be monitored are the same, the callback passed will replace the old callback already registered.
- `my_canv.bind(EButton1Up, None, ((x1, y1), (x2, y2)))` - If the pointer event and the screen area to be monitored are the same, and the callback passed is None, the callback registered previously is cleared.
- `my_canv.bind(EButton1Up, None)` - All callbacks previously registered for this pointer event are cleared, regardless of whether it was for a specific screen area or for the entire canvas.

Clicking on the white spot should result in `green_callback` to be called. Clicking on the red spot should result in `yellow_callback` to be called in both the scenarios shown above provided the

yellow_callback was registered last. Clicking on the purple spot should result in yellow_callback to be called.

begin_redraw([*((x1, y1), (x2, y2))*])
This is an explicit function that can be used to signal the window server that "I'm about to redraw this area". This method tells the window server that the window is about to respond to the last redraw event by redrawing the specified rectangle. This causes the window server to clear the rectangle, and remove it from the invalid region. The optional co-ordinates x1, y1, x2, y2 should be the rectangle that has to be marked for redrawing.

After the redraw is complete the application should call end_redraw().

Note: The begin_redraw and end_redraw methods should not be called inside the redraw callback function.

Couple of FAQs on redraw/non-redraw drawing:

Question: What is non-redraw drawing?

- "Non-redraw drawing" is any canvas/graphics drawing operation performed outside of begin_redraw()/end_redraw().

Question: What should applications do instead of non-redraw drawing?

- "Redraw drawing" is any drawing delimited by begin_redraw()/end_redraw().

Question: Why is non-redraw drawing bad for performance?

- The window server caches drawing operations in the redraw store. Delimiting drawing with begin_redraw()/end_redraw() allows window server to efficiently manage drawing operations.

  If applications perform drawing operations outside begin_redraw/end_redraw, window server cannot cull drawing operations from its cache of drawing operations, because it cannot know whether a set of drawing operations has been superceded by a new set. In this scenario every frame of drawing that is done on a non-redraw drawing window will become slower and slower as it draws all the drawing operations for the entire history of the window (well actually up until the last begin_redraw/end_redraw for the whole window).

  If an application performs begin_redraw/end_redraw, it tells the window server that it can throw away any old drawing operations it had for the area of the window specified in the redraw, thus allowing for more optimal management of drawing operations.

Question: What are the changes required for redraw drawing?

- Applications should delimit their drawing with begin_redraw()/end_redraw() - i.e. they should replace non-redraw drawing with redraw drawing. Sometimes, this is as straight forward as adding these calls to existing rendering code. In other cases (where the application has been drawing using "incremental updates" to the window, the application drawing code would need to be reworked to perform a full refresh of the area redrawn for the rect provided in begin_redraw(rect).

end_redraw()
Ends the current redraw. This function should be called when redrawing is complete.

Instances of Canvas type have the following attribute:

size
A two-element tuple that contains the current width and height of the Canvas as integers.

Instances of Canvas type have the same standard drawing methods that are documented in Section 3.3.

3.1.11 InfoPopup Type

An instance of InfoPopup type encapsulates an UI tip widget. This widget can be placed on top of other widgets to provide e.g. usage information to the user. The widget disappears as soon as the device's user presses any key or when the timer behind the InfoPopup is triggered.

class InfoPopup()
Creates an InfoPopup.

show(*text*, [*(x_coord, y_coord), time_shown, time_before, alignment*])
Show *text* (Unicode) in the `InfoPopup`. The optional parameters are the location (a tuple from the upper left corner), the time the popup is visible, *time_shown* (in milliseconds), the time before the popup, *time_before* (in milliseconds) and the alignment of the popup.

The default values are: the coordinates `(0, 0)`, *time_shown* 5 seconds, *time_before* 0 seconds and for the alignment `appuifw.EHLeftVTop`.

The *alignment* can be one of the constants defined in module `appuifw`[2]:

EHLeftVTop
Object is left and top aligned.

EHLeftVCenter
Object is left aligned and centred vertically.

EHLeftVBottom
Object is left aligned and at the bottom.

EHCenterVTop
Object is centre aligned horizontally and at the top.

EHCenterVCenter
Object is centred horizontally and vertically.

EHCenterVBottom
Object is centred horizontally and at the bottom.

EHRightVTop
Object is right and top aligned.

EHRightVCenter
Object is right aligned and centred vertically.

EHRightVBottom
Object is right aligned and at the bottom.

hide()
Hides the popup immediately.

Example:

```
>>> import appuifw
>>> i=appuifw.InfoPopup()
>>> i.show(u"Here is the tip.", (0, 0), 5000, 0, appuifw.EHRightVCenter)
>>>
```

3.2 `globalui` — Interface to the S60 global UI notifiers

The `globalui` module offers an interface to the S60 global UI notifiers. This allows a global note and query to be launched from an application which does not have a UI environment. The `globalui` module have functions:

global_note(*note_text*[, *type*])
Displays a note of the chosen type with *note_text* (Unicode). The default value for *type* is `'info'`. *type* can be one of the following strings: `'error'`, `'text'`, `'warn'`, `'charging'`, `'wait'`, `'perm'`, `'not_charging'`, `'battery_full'`, `'battery_low'`, `'recharge_battery'`, or `'confirm'`.

global_query(*query_text*[, *timeout*])
Displays a global confirmation query with *query_text* (Unicode). Returns `1` when the user presses 'Yes' and `0` otherwise. If the user does not respond to the query within *timeout* seconds, returns `None`. If the *timeout* value is 0, then the query waits indefinitely for user input. The default value for *timeout* is 0. The *timeout* value should be an integer.

[2] Descriptions of the values are from the S60 SDK documentation [4].

global_msg_query (*query_text, header_text*[, *timeout*])
Displays a global message query with *query_text*(Unicode). *header_text* is used to set the heading string of the query. Returns `1` when the user presses 'OK' and `0` otherwise. If the user does not respond to the query within *timeout* seconds, returns `None`. If the *timeout* value is `0`, then the query waits indefinitely for user input. The default value for *timeout* is `0`. The *timeout* value should be an integer.

global_popup_menu (*option_items*[, *header_text, timeout*])
Displays a global menu with *option_items*(Unicode). *header_text* is used to set the heading string of the menu. If no value is passed for *header_text*, then the header will not be displayed. Returns the index value of the selected item from the list. If the user does not respond to the menu within *timeout* seconds, returns `None`. If the *timeout* value is `0`, then the menu waits indefinitely for the input. The default value for *timeout* is `0`. The *timeout* value should be an integer.

Example:

```
>>> import globalui, time
...
>>> text_to_show = u"text for showing note"
>>> globalui.global_note(text_to_show,'error')
>>> time.sleep(6)
>>> globalui.global_note(text_to_show)
>>> time.sleep(6)
>>> result = globalui.global_query(u"do you want to continue ?")
>>> time.sleep(6)
>>> listresult = globalui.global_popup_menu([u"MenuItem1", u"MenuItem2"],u"Select item",5)
...
```

3.3 `graphics` — A graphics related services package

The `graphics` module provides access to the graphics primitives and image loading, saving, resizing, and transformation capabilities provided by the Symbian OS.

The module is usable from both graphical Python applications and background Python processes. However, background processes have some restrictions, namely that plain string symbolic font names are not supported in background processes since background processes have no access to the UI framework (see also Section 3.3.4).

For an example on using this module, see [**?**].

3.3.1 Module Level Functions

The following free functions - functions that do not belong to any class - are defined in the `graphics` module:

screenshot ()
Takes a screen shot and returns the image in `Image` format.

3.3.2 Image Class Static Methods

The following `Image` class static methods are defined in the `graphics` module:

Image.new (*size*[, *mode='RGB16'*])
Creates and returns a new `Image` object with the given size and mode. *size* is a two-element tuple. *mode* specifies the color mode of the `Image` to be created. It can be one of the following:

- `'1'`: Black and white (1 bit per pixel)
- `'L'`: 256 gray shades (8 bits per pixel)
- `'RGB12'`: 4096 colors (12 bits per pixel)

- `'RGB16'`: 65536 colors (16 bits per pixel)
- `'RGB'`: 16.7 million colors (24 bits per pixel)

It will also set the image size in twips according to the density of the device's primary screen.

`Image.open` (*filename*)
Returns a new `Image` object (mode `RGB16`) that contains the contents of the named file. The supported file formats are JPEG and PNG. The file format is automatically detected based on file contents. *filename* should be a full path name.

`Image.inspect` (*filename*)
Examines the given file and returns a dictionary of the attributes of the file. At present the dictionary contains only the image size in pixels as a two-element tuple, indexed by key `'size'`. *filename* should be a full path name.

3.3.3 Image Objects

An `Image` object encapsulates an in-memory bitmap.

Note on asynchronous methods: Methods `resize`, `transpose`, `save`, and `load` have an optional callback argument. If the callback is not given, the method call is synchronous; when the method returns, the operation is complete or an exception has been raised. If the callback is given, the method calls are asynchronous. If all parameters are valid and the operation can start, the method call will return immediately. The actual computation then proceeds in the background. When it is finished, the callback is called with an error code as the argument. If the given code is `0`, the operation completed without errors, otherwise an error occurred.

It is legal to use an unfinished image as a source in a blit operation; this will use the image data as it is at the moment the blit is made and may thus show an incomplete result.

`Image` objects have the following methods:

`resize` (*newsize*[, *callback=None, keepaspect=0*])
Returns a new image that contains a resized copy of this image. If *keepaspect* is set to `1`, the resize will maintain the aspect ratio of the image, otherwise the new image will be exactly the given size.

If *callback* is given, the operation is asynchronous, and the returned image will be only partially complete until *callback* is called.

`transpose` (*direction*[, *callback=None*])
Creates a new image that contains a transformed copy of this image. The *direction* parameter can be one of the following:

- `FLIP_LEFT_RIGHT`: Flips the image horizontally, exchanging left and right edges.
- `FLIP_TOP_BOTTOM`: Flips the image vertically, exchanging top and bottom edges.
- `ROTATE_90`: Rotates the image 90 degrees counterclockwise.
- `ROTATE_180`: Rotates the image 180 degrees.
- `ROTATE_270`: Rotates the image 270 degrees counterclockwise.

If *callback* is given, the operation is asynchronous and the returned image will be only partially complete until *callback* is called.

`load` (*filename*[, *callback=None*])
Replaces the contents of this `Image` with the contents of the named file, while keeping the current image mode. This `Image` object must be of the same size as the file to be loaded.

If *callback* is given, the operation is asynchronous and the loaded image will be only partially complete until *callback* is called. *filename* should be a full path name.

`save` (*filename*[,*callback=None, format=None, quality=75, bpp=24, compression='default'*])
Saves the image into the given file. The supported formats are JPEG and PNG. If *format* is not given or is set to `None`, the format is determined based on the file name extension: `'.jpg'` or `'.jpeg'` are interpreted to be in JPEG format and `'.png'` to be in PNG format. *filename* should be a full path name.

When saving in JPEG format, the *quality* argument specifies the quality to be used and can range from 1 to 100.

When saving in PNG format, the *bpp* argument specifies how many bits per pixel the resulting file should have, and *compression* specifies the compression level to be used.

Valid values for *bpp* are:

- `1`: Black and white, 1 bit per pixel
- `8`: 256 gray shades, 8 bits per pixel
- `24`: 16.7 million colors, 24 bits per pixel

Valid values for *compression* are:

- `'best'`: The highest possible compression ratio, the slowest speed
- `'fast'`: The fastest possible saving, moderate compression
- `'no'`: No compression, very large file size
- `'default'`: Default compression, a compromise between file size and speed

If *callback* is given, the operation is asynchronous. When the saving is complete, the *callback* is called with the result code.

`stop``()`
Stops the current asynchronous operation, if any. If an asynchronous call is not in progress, this method has no effect.

`Image` objects have the following attributes:

`size`
A two-element tuple that contains the size of the `Image`. Read-only.

`twipsize`
A two-element tuple that contains the size of the `Image` in twips. Read/Write.

3.3.4 Common Features of Drawable Objects

Objects that represent a surface that can be drawn on support a set of common drawing methods, described in this section. At present there are two such objects: `Canvas` from the `appuifw` module and `Image` from the `graphics` module.

Options

Many of these methods support a set of standard options. This set of options is as follows:

- *outline*: The color to be used for drawing outlines of primitives and text. If `None`, the outlines of primitives are not drawn.
- *fill*: The color to be used for filling the insides of primitives. If `None`, the insides of primitives are not drawn. If *pattern* is also specified, *fill* specifies the color to be used for areas where the pattern is white.
- *width*: The line width to be used for drawing the outlines of primitives.
- *pattern*: Specifies the pattern to be used for filling the insides of primitives. If given, this must be either `None` or a 1-bit (black and white) `Image`.

Coordinate representation

The methods accept an ordered set of coordinates in the form of a coordinate sequence. Coordinates can be of type `int`, `long`, or `float`. A valid coordinate sequence is a non-empty sequence of either

- Alternating x and y coordinates. In this case the sequence length must be even, or
- Sequences of two elements, that specify x and y coordinates.

Examples of valid coordinate sequences:

- `(1, 221L, 3, 4, 5.85, -3)`: A sequence of three coordinates
- `[(1,221L),(3,4),[5.12,6])`: A sequence of three coordinates
- `(1,5)`: A sequence of one coordinate
- `[(1,5)]`: A sequence of one coordinate
- `[[1,5]]`: A sequence of one coordinate

Examples of invalid coordinate sequences:

Invalid code, do not use!

- `[]`: An empty sequence
- `(1,2,3)`: Odd number of elements in a flat sequence
- `[(1,2),(3,4),None]`: Contains an invalid element
- `([1,2],3,4)`: Mixing the flat and nested form is not allowed

Color representation

All methods that take color arguments accept the following two color representations:

- A three-element tuple of integers in the range from 0 to 255 inclusive, representing the red, green, and blue components of the color.
- An integer of the form `0xrrggbb`, where `rr` is the red, `gg` the green, and `bb` the blue component of the color.

For 12 and 16 bit color modes the color component values are simply truncated to the lower bit depth. For the 8-bit grayscale mode images the color is converted into grayscale using the formula `(2*r+5*g+b)/8`, rounded down to the nearest integer. For 1-bit black and white mode images the color is converted into black (0) or white (1) using the formula `(2*r+5*g+b)/1024`.

Examples of valid colors:

- `0xffff00`: Bright yellow
- `0x004000`: Dark green
- `(255,0,0)`: Bright red
- `0`: Black
- `255`: Bright blue
- `(128,128,128)`: Medium gray

Examples of invalid colors:

Invalid code, do not use!

- `(0,0.5,0.9)`: Floats are not supported
- `'#ff80c0'`: The HTML color format is not supported
- `(-1,0,1000)`: Out-of-range values
- `(1,2)`: The sequence is too short
- `[128,128,192]`: This is not a tuple

Font specifications

A font can be specified in three ways:

- None, meaning the default font
- a Unicode string that represents a full font name, such as `u'LatinBold19'`
- a plain string symbolic name that refers to a font setting currently specified by the UI framework
- as a two or three element tuple, where
  - the first element is the font name (unicode or string) or None for default font
  - the second element is the font height in pixels or None for default size
  - the third (optional) element is the flags applied to the font or None for default options.

The flags are the following:

- `FONT_BOLD` bold
- `FONT_ITALIC` italic
- `FONT_SUBSCRIPT` subscript
- `FONT_SUPERSCRIPT` superscript
- `FONT_ANTIALIAS` forces the font to be antialiased
- `FONT_NO_ANTIALIAS` forces the font to not be antialiased

You can combine the flags with the binary or operator "—". For example, the flags setting `FONT_BOLD|FONT_ITALIC` will produce text that is both bold and italic.

Note: Antialiasing support is only available for scalable fonts.

You can obtain a list of all available fonts with the `appuifw` module function `available_fonts`.

The symbolic names for UI fonts are:

- `'normal'`
- `'dense'`
- `'title'`
- `'symbol'`
- `'legend'`
- `'annotation'`

Since background processes have no access to the UI framework, these symbolic names are not supported in them. You need to specify the full font name.

line (*coordseq*[, *<options>*])
Draws a line connecting the points in the given coordinate sequence. For more information about the choices available for *options*, see Section 3.3.4.

polygon (*coordseq*[, *<options>*])
Draws a line connecting the points in the given coordinate sequence, and additionally draws an extra line connecting the first and the last point in the sequence. If a fill color or pattern is specified, the polygon is filled with that color or pattern. For more information about the choices available for *options*, see Section 3.3.4.

rectangle (*coordseq*[, *<options>*])
Draws rectangles between pairs of coordinates in the given sequence. The coordinates specify the top-left and the bottom- right corners of the rectangle. The sequence must have an even number of coordinates. For more information about the choices available for *options*, see Section 3.3.4.

ellipse (*coordseq*[, *<options>*])
Draws ellipses between pairs of coordinates in the given sequence. The coordinates specify the top-left and bottom-right corners of the rectangle inside which the ellipse is contained. The sequence must have an even number of coordinates. For more information about the choices available for *options*, see Section 3.3.4.

pieslice (*coordseq, start, end*[, *<options>*])
Draws pie slices contained in ellipses between pairs of coordinates in the given sequence. The start and end parameters are floats that specify the start and end points of pie slice as the starting and ending angle in radians. The angle `0` is to the right, the angle `pi/2` is straight up, `pi` is to the left and`-pi/2` is straight down. *coordseq* is interpreted the same way as for the `ellipse` method. For more information about the choices available for *options*, see Section 3.3.4.

arc (*coordseq, start, end*[, *<options>*])
Draws arcs contained in ellipses between pairs of coordinates in the given sequence. The start and end parameters are floats that specify the start and end points of pie slice as the starting and ending angle in radians. The angle `0` is to the right, the angle `pi/2` is straight up, `pi` is to the left and`-pi/2` is straight down. *coordseq* is interpreted the same way as for the `ellipse` method. For more information about the choices available for *options*, see Section 3.3.4.

point (*coordseq*[, *<options>*])
Draws points in each coordinate in the given coordinate sequence. If the *width* option is set to greater than 1, draws a crude approximation of a circle filled with the outline color in the locations. Note that the approximation is not very accurate for large widths; use the `ellipse` method if you need a precisely formed circle. For more information about the choices available for *options*, see Section 3.3.4.

clear ([*color=0xffffff*])
Sets the entire surface of the drawable to the given color, white by default.

text (*coordseq, text*[*fill=0, font=None*])
Draws the given text in the points in the given coordinate sequence with the given color (default value is black) and the given font. The font specification format is described above.

measure_text (*text*[*font=None, maxwidth=-1, maxadvance=-1*])
Measures the size of the given text when drawn using the given font. Optionally you can specify the maximum width of the text or the maximum amount the graphics cursor is allowed to move (both in pixels).

Returns a tuple of three values:

- the bounding box for the text as a 4-tuple: (topleft-x, topleft-y, bottomright-x, bottomright-y)
- the number of pixels the graphics cursor would move to the right
- the number of characters of the text that fits into the given maximum width and advance

blit (*image*[,*target=(0,0), source=((0,0),image.size), mask=None, scale=0*])
Copies the source area from the given *image* to the target area in this drawable. The source area is copied in its entirety if *mask* is not given or is set to `None`. If the mask is given, the source area is copied where the mask is white. *mask* can be either `None`, a 1-bit (black and white) `Image` or a grayscale `Image`, and

must be of the same size as the source image. A grayscale mask acts as an alpha channel, i.e. partial transparency.

target and *source* specify the target area in this image and the source area in the given source. They are coordinate sequences of one or two coordinates. If they specify one coordinate, it is interpreted as the upper-left corner for the area; if they specify two coordinates, they are interpreted as the top-left and bottom-right corners of the area.

If *scale* is other than zero, scaling is performed on the fly while copying the source area to the target area. If *scale* is zero, no scaling is performed, and the size of the copied area is clipped to the smaller of source and target areas.

Note that a `blit` operation with scaling is slower than one without scaling. If you need to blit the same `Image` many times in a scaled form, consider making a temporary `Image` of the scaling result and blitting it without scaling. Note also that the scaling performed by the `blit` operation is much faster but of worse quality than the one done by the `resize` method, since the `blit` method does not perform any antialiasing.

3.4 `camera` — Interface for taking photographs and video recording

The `camera` module enables taking photographs and video recording.

The following data items for state information are available in `camera`:

`EOpenComplete`
The opening of the video clip has succeeded.

`ERecordComplete`
The video recording has completed (not called on explicit `stop_recording` call).

`EPrepareComplete`
The device is ready to begin video recording.

The `camera` module has the following functions[3]:

`cameras_available``()`
Returns the number of cameras available in the device.

`image_modes``()`
Returns the image modes supported in the device as a list of strings, for example: `['RGB12', 'RGB', 'JPEG_Exif', 'RGB16']`.

`image_sizes``()`
Returns the image sizes (resolution) supported in the device as a list of `(x, y)` tuples, for example: `[(640, 480), (160, 120)]`.

`flash_modes``()`
Returns the flash modes available in the device as a list of strings.

`max_zoom``()`
Returns the maximum digital zoom value supported in the device as an integer.

`exposure_modes``()`
Returns the exposure settings supported in the device as a list of strings.

`white_balance_modes``()`
Returns the white balance modes available in the device as a list of strings.

`take_photo``(`[*mode, size, zoom, flash, exposure, white_balance, position*]`)`
Takes a photograph and returns the image in:

1. `Image` format (for more information on `Image` format, see Chapter 3.3 `graphics` Module) or
2. Raw JPEG data[4].

[3] Descriptions for some of the values are based on information found in S60 SDK documentation [4]
[4] For more information, see e.g. http://en.wikipedia.org/wiki/JPEG.

The settings listed below describe all settings that are supported by the `camera` module. You can retrieve the mode settings available for your device by using the appropriate functions listed at the beginning of this chapter.

- *mode* is the display mode of the image. The default value is `'RGB16'`. The following display modes are supported for the `Image` format pictures taken:
  - `'RGB12'`: 4096 colors (12 bits per pixel)
  - `'RGB16'`: 65536 colors (16 bits per pixel). Default value, always supported
  - `'RGB'`: 16.7 million colors (24 bits per pixel)

  For the JPEG data format images the following modes are supported:
  - `'JPEG_Exif'`: JPEG Exchangeable image file format
  - `'JPEG_JFIF'`: JPEG File Interchange Format

  Note that there is variety between the devices and the supported formats.
- *size* is the resolution of the image. The default value is `(640, 480)`. The following sizes are supported, for example, in Nokia 6630: `(1280, 960)`, `(640, 480)` and `(160, 120)`.
- *flash* is the flash mode setting. The default value is `'none'`. The following flash mode settings are supported:
  - `'none'`
    No flash. Default value, always supported
  - `'auto'`
    Flash will automatically fire when required
  - `'forced'`
    Flash will always fire
  - `'fill_in'`
    Reduced flash for general lighting
  - `'red_eye_reduce'`
    Red-eye reduction mode
- *zoom* is the digital zoom factor. It is assumed to be on a linear scale from 0 to the maximum zoom value allowed in the device. The default value is `0`, meaning that zoom is not used.
- *exposure* is the exposure adjustment of the device. Exposure is a combination of lens aperture and shutter speed used in taking a photograph. The default value is `'auto'`. The following exposure modes are supported:
  - `'auto'`
    Sets exposure automatically. Default value, always supported
  - `'night'`
    Night-time setting for long exposures
  - `'backlight'`
    Backlight setting for bright backgrounds
  - `'center'`
    Centered mode for ignoring surroundings
- *white_balance* can be used to adjust white balance to match the main source of light. The term white balance refers to the color temperature of the current light. A digital camera requires a reference point to represent white. It will then calculate all the other colors based on this white point. The default value for *white_balance* is `'auto'` and the following white balance modes are supported:
  - `'auto'`
    Sets white balance automatically. Default value, always supported
  - `'daylight'`
    Sets white balance to normal daylight
  - `'cloudy'`
    Sets white balance to overcast daylight
  - `'tungsten'`
    Sets white balance to tungsten filament lighting

– 'fluorescent'
Sets white balance to fluorescent tube lighting
– 'flash'
Sets white balance to flash lighting

• *position* is the camera used if the device, such as Nokia N95, has several cameras. In Nokia N95, the camera pointing to the user of the device is located in position 1, whereas the one pointing away from the user is located in position 0. The default *position* is 0.

If some other application is using the camera, this operation fails, with error SymbianError: KErrInUse. Invoking this function right after the device boot, might result in SymbianError: KErrNotReady error.

In some Nokia devices (e.g. in N95), to be able to get the highest possible size for the captured image, you need to:

1. switch to the landscape mode (see appuifw.app.orientation)
2. import the camera module
3. take the picture in the 'JPEG_Exif' format.

start_finder (*callable*[, *backlight_on=1, size=main_pane_size*])
Starts the camera viewfinder and binds a callback to receive Image format feed. When a new viewfinder frame is ready the callback is invoked with the Image as parameter.

The optional parameter backlight_on determines whether the device backlight is kept on when the camera view finder is in operation. By default, the backlight is on (1 = on, 0 = off).

The optional parameter size (of type tuple, e.g. (176, 144)) can be used to change the size of the Image received in the callback. The default size is the same as the application's main pane size.

Example view finder code:

```
>>> import appuifw
>>> import camera
>>> def cb(im):
...   appuifw.app.body.blit(im)
...
>>> import graphics
>>> appuifw.app.body=appuifw.Canvas()
>>> camera.start_finder(cb)
>>>
```

stop_finder()
Stops the viewfinder.

release()
Releases the camera – After invocation other applications can access the camera hardware.

start_record (*filename, callable*)
Starts video recording. *filename* is the file where the video clip is saved and *callable* will be called with possible error code (int) and status information (see data in module camera) as parameter.

Prior calling this function, the view finder needs to be started.

stop_record()
Stops the video recording.

3.5 keycapture — Interface for global capturing of key events.

The keycapture module offers an API for global capturing of key events. The keycapture module provides the KeyCapturer object as a tool for listening to events.

The KeyCapturer object uses a callback method to report the key events. The callback method is called each time any of the specified keys is pressed.

Currently the keycapture module does not support capturing separate key-up or key-down events.

Note: Keycapture module requires SwEvent capability.

3.5.1 Module Level Constants

The following constants are defined in the keycapture module:

all_keys
A list of all key codes defined in the key_codes module.

3.5.2 KeyCapturer objects

KeyCapturer object takes a callback method as a mandatory parameter to its constructor. The callback method must have one single parameter for forwarding the key code of the captured key.

There can be several KeyCapturer objects existing at the same time.

KeyCapturer object has following methods and properties:

keys
List of keys to be captured. Can be read and written.
Example:

```
keys = (key_codes.EkeyUpArrow,)
keys = keycapture.all_keys
```

forwarding
Specifies whether captured key events are forwarded to other applications or not. Either has value 1 or 0. Can be read and written.

start()
Starts the actual capturing of key events.

stop()
Stops the actual capturing of key events.

last_key()
Returns last key code that is captured.

3.6 topwindow — Interface for creating windows that are shown on top of other applications.

The topwindow module offers an API for creating windows that are shown on top of other applications and managing the content of these windows. Images can be inserted into the windows and the background color, visibility, corner type and shadow of the window can be manipulated.

topwindow extension does not provide sophisticated drawing capabilities by any means but rather relies on services provided by the graphics extension: topwindow allows graphics Image objects to be put into the windows that are represented by TopWindow objects.

TopWindow object provides mainly only two services: TopWindow objects can be shown or hidden and Images can be put into the windows. However, several images can be added into one TopWindow object and several TopWindow objects can be created and shown. Since the images can be manipulated using the graphics extension this makes it possible to create many kind of content to the TopWindow objects.

3.6.1 TopWindow objects

class TopWindow()
Create a `TopWindow` object.

`TopWindow` objects have the following methods and properties:

show()
Shows the window. The window is not shown until show() is called.

hide()
Hides the window.

add_image(*image, position*)
Inserts an image object `graphics.Image` into the window. The position of the image is specified by the *(*position) parameter. If only the coordinates of the top left corner are specified, like (x1, y1) the image is not resized. If four coordinates are given, like(x1, y1, x2, y2), the image is resized to fit to the specified area.
Example:

```
add_image(image, (10,20))
add_image(image, (10,20,20,30))
```

remove_image(*image*[,*position*])
Removes the image from the window. Mandatory parameter *image* must be a `graphics.Image` object. Parameter *position* may specify the top-left corner coordinates of the image or the rectangular area of the image. If only *image* parameter is given, all the pictures representing this image object are removed from the window. If both parameters are given, only the picture that matches both parameters is removed.
Example:

```
remove_image(image)
remove_image(image, (10,10))
remove_image(image, (10,10,20,20))
```

position
Specifies the coordinates of the top left corner of the window. Can be read and written.
Example:

```
position = (10, 20)
```

size
Specifies the size of the window. Can be read and written.
Example:

```
size = (100, 200)
```

images
The images inserted into the window. Defined as a list of tuple objects. Each tuple contains a `graphics.Image` object and the *position* of the image. The *position* may specify the top-left coordinate of the image and optionally also the bottom-right coordinate of the image. Parameter (x,y) specifies the top-left coordinate, but does not resize the image while parameter like (x1,y1,x2,y2) specifies both the top-left and bottom-right coordinates and possibly also resizes the image. Can be read and written. Also see the `add_image()` and `remove_image()` methods.
Example:

```
images = [(image1,(x1,y1)), (image2,(x1,y1,x2,y2)), (image3,(50,50,100,100))]
```

sets the window content to be 3 images. `image2` and `image3` are possibly resized while the `image1` is not)

shadow
Specifies if the shadow of the window is shown and the length of the shadow. Can be read and written. Setting `shadow = 0` makes the shadow invisible.
Example: `shadow = 5`

corner_type
Specifies the corner type of the window. Can be read and written. Corner type can be one of the following values:

- `square`
- `corner1`
- `corner2`
- `corner3`
- `corner5`

Example: `corner_type = "square"`

maximum_size
Returns the maximum size of the window as a tuple (width, height). Read only property.

background_color
The background color of the window as an integer (e.g. `0xaabbcc`). The two greatest hexadecimal digits specify the red, the next two specify the blue and the last ones specify the green color. Can be read and written.
Example: `background_color = 0xffffff` (sets the white color)

visible
Can be set to 0 or 1. 1 means that window is visible, 0 means that it is not. Can be read and written. Also see the `show` and `hide` methods.

3.7 `gles` — Bindings to OpenGL ES

The `gles` module provides Python bindings to OpenGL ES 2D/3D graphics C API. OpenGL ES is a standard defined by Khronos Group (www.khronos.org). Currently S60 Python supports OpenGL ES version 1.0 from Series 60 version 2.6 onwards. Support for OpenGL ES version 1.1 should also become available in the near future, and both versions are documented here. OpenGL ES 1.1 will require Series 60 version 3.0 or newer.

For detailed description of the OpenGL ES API see the official specifications at http://www.khronos.org/opengles. This documentation contains only information that is specific to the S60 Python bindings to OpenGL ES. Where possible, the conventions of the PyOpenGL desktop OpenGL bindings (http://pyopengl.sourceforge.net) have been followed.

The display of OpenGL ES graphics is handled by separate module, `glcanvas`. See `glcanvas` module documentation for more information.

3.7.1 array type

`gles` module defines `array` type for representing numerical data of specific GL type. `array` objects are convenient when numerical data for OpenGL ES calls is specified in Python code. Class `array` also defines the standard Python sequence methods so its instances can be iterated and individual items in arrays can be manipulated easily.

class `array` (*type, dimension, sequence*)
Constructs a new `array` object that contains the given type of data that is taken from *sequence*. Parameter *dimension* specifies how many items there are in each array element. The dimension information is stored with the array and is used by those functions that need to know the element size of the input data, for example, if colors are specified with three or four components. The dimension does not affect the length of an array or its indexing: both are based on individual items.

Value of *type* must be one of the following: `GL_FLOAT`, `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, or `GL_FIXED`.

The data in *sequence* is flattened before it is used to fill the array. When *type* is `GL_FLOAT`, the sequence can contains floats or integers. With all other types, *sequence* must only contain integers. Values in *sequence* are casted in C to the requested type, so if the requested type cannot properly represent all the values the results can be unexpected.

__len__()
Returns the number of items in the array. Note that array dimension does not affect the calculation of the length.

__getitem__(*index*)
Returns the item in array with *index*. Note that array dimension does not affect indexing.

__setitem__(*index, value*)
Sets the value of the item in position *index* to *value*. Note that array dimension does not affect indexing.

3.7.2 Error handling

Errors generated by the API calls are handled similarly as in PyOpenGL: all GL errors are reported as Python exceptions of type `gles.GLerror`. The wrapper code checks GL error status after each call automatically. There is no Python binding for `glGetError` call.

3.7.3 Differences to OpenGL ES C API

Certain OpenGL ES functions require special handling in Python, mainly because of the pointer parameters in the C API. Additionally, special Python versions for some OpenGL ES functions have been added. Both of sets of functions are documented below. If a function is not listed here its Python version should exactly match the C version defined in the official OpenGL ES 1.0 and 1.1 specifications.

OpenGL ES 1.0

glColorPointer(*size, type, stride, sequence*)
Parameter *sequence* must be either a `gles.array` object or some other Python sequence object. `gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object is used, the type and dimension of its data are ignored and *size* and *type* are used instead.

glColorPointerub(*sequence*)
Special Python version of `glColorPointer` that accepts either a `gles.array` object or some other Python sequence object. Other parameters of `glColorPointer` will be determined as follows:

- *size* If *sequence* is an instance of `gles.array`, its dimension is used; otherwise the length of *sequence* is used.
- *type* `GL_UNSIGNED_BYTE`
- *stride* 0

glColorPointerf(*sequence*)
Special Python version of `glColorPointer` that behaves exactly as `glColorPointerub` except `GL_FLOAT` is used as *type*.

glColorPointerx(*sequence*)
Special Python version of `glColorPointer` that behaves exactly as `glColorPointerub` except `GL_FIXED` is used as *type*.

glCompressedTexImage2D(*target, level, internalformat, width, height, border, imageSize, data*)
Parameter *data* must be either a `gles.array` or a Python string.

glCompressedTexSubImage2D(*target, level, xoffset, yoffset, width, height, format, imageSize, data*)
Parameter *data* must be either a `gles.array` or a Python string.

glDeleteTextures (*sequence*)
Parameter *sequence* must be a Python sequence containing integers.

glDrawElements (*mode, count, type, indices*)
Parameter *indices* must be either a `gles.array` or some other Python sequence object. `gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object is used, the type of its data is ignored and *type* is used instead.

glDrawElementsub (*mode, indices*)
Special Python version of `glDrawElements` that uses length of the sequence *indices* as *count* and `GL_UNSIGNED_BYTE` as *type*.

glDrawElementsus (*mode, indices*)
Special Python version of `glDrawElements` that uses length of the sequence *indices* as *count* and `GL_UNSIGNED_SHORT` as *type*.

glFogv (*pname, params*)
Parameter *params* must be a Python sequence containing float values.

glFogxv (*pname, params*)
Parameter *params* must be a Python sequence containing integer values.

glGenTextures (*n*)
The generated texture names are returned in a Python tuple.

glGetIntegerv (*pname*)
The values are returned in a Python tuple.

glGetString (*name*)
The value is return as a Python string.

glLightModelfv (*pname, params*)
Parameter *params* must be a Python sequence containing float values.

glLightModelxv (*pname, params*)
Parameter *params* must be a Python sequence containing integer values.

glLightfv (*light, pname, params*)
Parameter *params* must be a Python sequence containing float values.

glLightxv (*light, pname, params*)
Parameter *params* must be a Python sequence containing integer values.

glLoadMatrixf (*m*)
Parameter *m* must be a Python sequence containing float values. The sequence is flattened before its items are read.

glLoadMatrixx (*m*)
Parameter *m* must be a Python sequence containing integer values. The sequence is flattened before its items are read.

glMaterialfv (*face, pname, params*)
Parameter *params* must be a Python sequence containing float values.

glMaterialxv (*face, pname, params*)
Parameter *params* must be a Python sequence containing integer values.

glMultMatrixf (*m*)
Parameter *m* must be a Python sequence containing float values. The sequence is flattened before its items are read.

glMultMatrixx (*m*)
Parameter *m* must be a Python sequence containing integer values. The sequence is flattened before its items are read.

glNormalPointer (*type, stride, sequence*)
Parameter *sequence* must be either a `gles.array` object or some other Python sequence object. `gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object

is used, the type of its data is ignored and *type* is used instead.

glNormalPointerb(*sequence*)
Special Python version of `glNormalPointer` that uses *type* `GL_BYTE` and *stride* 0.

glNormalPointers(*sequence*)
Special Python version of `glNormalPointer` that uses *type* `GL_SHORT` and *stride* 0.

glNormalPointerf(*sequence*)
Special Python version of `glNormalPointer` that uses *type* `GL_FLOAT` and *stride* 0.

glNormalPointerx(*sequence*)
Special Python version of `glNormalPointer` that uses *type* `GL_FIXED` and *stride* 0.

glReadPixels(*x, y, width, height, format, type*)
The pixel data read is returned in a Python string.

glTexCoordPointer(*size, type, stride, sequence*)
Parameter *sequence* must be either a `gles.array` object or some other Python sequence object. `gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object is used, the dimension and type of its data are ignored and *size* and *type* are used instead.

glTexCoordPointerb(*sequence*)
Special Python version of `glTexCoordPointer` that accepts either a `gles.array` object or some other Python sequence object. Other parameters of `glTexCoordPointer` will be determined as follows:

- *size* If *sequence* is an instance of `gles.array`, its dimension is used; otherwise the length of *sequence* is used.
- *type* `GL_BYTE`
- *stride* 0

glTexCoordPointers(*sequence*)
Special Python version of `glTexCoordPointer` that behaves exactly as `glTexCoordPointerb` except `GL_SHORT` is used as *type*.

glTexCoordPointerf(*sequence*)
Special Python version of `glTexCoordPointer` that behaves exactly as `glTexCoordPointerb` except `GL_FLOAT` is used as *type*.

glTexCoordPointerx(*sequence*)
Special Python version of `glTexCoordPointer` that behaves exactly as `glTexCoordPointerb` except `GL_FIXED` is used as *type*.

glTexEnvfv(*face, pname, params*)
Parameter *params* must be a Python sequence containing float values.

glTexEnvxv(*face, pname, params*)
Parameter *params* must be a Python sequence containing integer values.

glTexImage2D(*target, level, internalformat, width, height, border, format, type, pixels*)
Parameter *pixels* must be either a Python string, a `gles.array` object, or `graphics.Image` object. Python strings are taken as literal data with no conversion. The dimension and type of data in `gles.array` objects are ignored: the raw data in the array is used.

Use of `graphics.Image` objects is limited to only some combinations of *format* and *type*. Table A.2 below shows the accepted combinations. To get the best results and performance, the `CFbsBitmap` object in the `graphics.Image` object should be in the equivalent display mode, also shown in the table below. Otherwise, the `CFbsBitmap` object will be first converted to the equivalent display mode before reading its pixel data, which can degrade the visual quality in some cases.

glTexSubImage2D(*target, level, xoffset, yoffset, width, height, format, type, pixels*)
The handling of *pixels* is the same as with `glTexImage2D`.

glVertexPointer(*size, type, stride, sequence*)
Parameter *sequence* must be either a `gles.array` object or some other Python sequence object.

format	*type*	The equivalent display mode
GL_LUMINANCE, GL_ALPHA	GL_UNSIGNED_BYTE	EGray256
GL_RGB	GL_UNSIGNED_BYTE	EColor16M
GL_RGB	GL_UNSIGNED_SHORT_5_6_5	EColor64K

Table 3.1: Legal combinations of format and type with the equivalent Symbian display mode

`gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object is used, the dimension and type of its data are ignored and *size* and *type* are used instead.

`glVertexPointerb`(*sequence*)
Special Python version of `glVertexPointer` that accepts either a `gles.array` object or some other Python sequence object. Other parameters of `glVertexPointer` will be determined as follows:

- *size* If *sequence* is an instance of `gles.array`, its dimension is used; otherwise the length of *sequence* is used.
- *type* `GL_BYTE`
- *stride* 0

`glVertexPointers`(*sequence*)
Special Python version of `glVertexPointer` that behaves exactly as `glVertexPointerb` except `GL_SHORT` is used as *type*.

`glVertexPointerf`(*sequence*)
Special Python version of `glVertexPointer` that behaves exactly as `glVertexPointerb` except `GL_FLOAT` is used as *type*.

`glVertexPointerx`(*sequence*)
Special Python version of `glVertexPointer` that behaves exactly as `glVertexPointerb` except `GL_FIXED` is used as *type*.

OpenGL ES 1.1

`glBufferData`(*target, size, data, usage*)
Parameter *data* must be a `gles.array` object. If *size* is -1, the in-memory size of *data* is used in its place.

`glBufferDatab`(*target, data, usage*)
Special Python version of `glBufferData` that accepts either a `gles.array` object or some other Python sequence object for *data*. If `gles.array` object is used, its in-memory size in bytes is used as *size*. Other sequences are first converted to flat lists of `GL_BYTE` data by casting. The length of the resulting sequence in bytes is used as *size*.

`glBufferDataub`(*target, data, usage*)
Special Python version of `glBufferData` that works exactly like `glBufferDatab` except `GL_UNSIGNED_BYTE` is used instead of `GL_BYTE`.

`glBufferDatas`(*target, data, usage*)
Special Python version of `glBufferData` that works exactly like `glBufferDatab` except `GL_SHORT` is used instead of `GL_BYTE`.

`glBufferDataus`(*target, data, usage*)
Special Python version of `glBufferData` that works exactly like `glBufferDatab` except `GL_UNSIGNED_SHORT` is used instead of `GL_BYTE`.

`glBufferDataf`(*target, data, usage*)
Special Python version of `glBufferData` that works exactly like `glBufferDatab` except `GL_FLOAT` is used instead of `GL_BYTE`.

`glBufferDatax`(*target, data, usage*)
Special Python version of `glBufferData` that works exactly like `glBufferDatab` except `GL_FIXED` is used instead of `GL_BYTE`.

glBufferSubData(*target, size, data, usage*)
Parameter *data* must be a `gles.array` object. If *size* is -1, the in-memory size of *data* is used in its place.

glBufferSubDatab(*target, data, usage*)
Special Python version of `glBufferSubData` that accepts either a `gles.array` object or some other Python sequence object for *data*. If `gles.array` object is used, its in-memory size (in bytes) is used as *size*. Other sequences are first converted to flat lists of `GL_BYTE` data by casting. The length of the resulting sequence is used as *size*.

glBufferSubDataub(*target, data, usage*)
Special Python version of `glBufferSubData` that works exactly like `glBufferSubDatab` except `GL_UNSIGNED_BYTE` is used instead of `GL_BYTE`.

glBufferSubDatas(*target, data, usage*)
Special Python version of `glBufferSubData` that works exactly like `glBufferSubDatab` except `GL_SHORT` is used instead of `GL_BYTE`.

glBufferSubDataus(*target, data, usage*)
Special Python version of `glBufferSubData` that works exactly like `glBufferSubDatab` except `GL_UNSIGNED_SHORT` is used instead of `GL_BYTE`.

glBufferSubDataf(*target, data, usage*)
Special Python version of `glBufferSubData` that works exactly like `glBufferSubDatab` except `GL_FLOAT` is used instead of `GL_BYTE`.

glBufferSubDatax(*target, data, usage*)
Special Python version of `glBufferSubData` that works exactly like `glBufferSubDatab` except `GL_FIXED` is used instead of `GL_BYTE`.

glClipPlanef(*plane, equation*)
Parameter *equation* must be a Python sequence that contains four float values.

glClipPlanex(*plane, equation*)
Parameter *equation* must be a Python sequence that contains four integer values.

glDeleteBuffers(*buffers*)
Parameter *buffers* must be a Python sequence that contains integer values.

glDrawTexsvOES(*coords*)
Parameter *coords* must be a Python sequence that contains integer values.

glDrawTexivOES(*coords*)
Parameter *coords* must be a Python sequence that contains integer values.

glDrawTexfvOES(*coords*)
Parameter *coords* must be a Python sequence that contains float values.

glDrawTexfvOES(*coords*)
Parameter *coords* must be a Python sequence that contains integer values.

glGenBuffers(*n*)
The generated buffer names are returned in a Python tuple.

glGetBooleanv(*pname*)
The values are returned in a Python tuple.

glGetBufferParameteriv(*target, pname*)
The value is returned as an integer.

glGetClipPlanef(*plane*)
The values are returned in a Python tuple.

glGetClipPlanef(*plane*)
The values are returned in a Python tuple.

glGetFixedv(*pname*)
The values are returned in a Python tuple.

glGetFloatv(*pname*)
The values are returned in a Python tuple.

glGetLightfv(*light, pname*)
The values are returned in a Python tuple.

glGetLightxv(*light, pname*)
The values are returned in a Python tuple.

glGetMaterialfv(*face, pname*)
The values are returned in a Python tuple.

glGetMaterialxv(*face, pname*)
The values are returned in a Python tuple.

glGetTexEnvf(*face, pname*)
The values are returned in a Python tuple.

glGetTexEnvx(*face, pname*)
The values are returned in a Python tuple.

glGetTexParameterf(*target, pname*)
The value is returned as a float.

glGetTexParameterx(*target, pname*)
The value is returned as an integer.

glMatrixIndexPointerOES(*size, type, stride, sequence*)
Parameter *sequence* must be either a `gles.array` object or some other Python sequence object. `gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object is used, the dimension and type of its data are ignored and *size* and *type* are used instead.

glMatrixIndexPointerOESub(*sequence*)
Special Python version of `glMatrixIndexPointerOES` that accepts either a `gles.array` object or some other Python sequence object. Other parameters of `glMatrixIndexPointerOES` will be determined as follows:

- *size* If *sequence* is an instance of `gles.array`, its dimension is used; otherwise the length of *sequence* is used.
- *type* `GL_UNSIGNED_BYTE`
- *stride* 0

glPointParameterfv(*pname, params*)
Parameter *params* must be a Python sequence containing float values.

glPointParameterxv(*pname, params*)
Parameter *params* must be a Python sequence containing integer values.

glPointSizePointerOES(*type, stride, sequence*)
Parameter *sequence* must be either a `gles.array` object or some other Python sequence object. `gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object is used, the type of its data is ignored and *type* is used instead.

glPointSizePointerOESf(*sequence*)
Special Python version of `glPointSizePointerOES` uses `GL_FLOAT` as *type* and 0 as *stride*.

glPointSizePointerOESx(*target, data, usage*)
Special Python version of `glPointSizePointerOES` uses `GL_FIXED` as *type* and 0 as *stride*.

glWeightPointerOES(*size, type, stride, sequence*)
Parameter *sequence* must be either a `gles.array` object or some other Python sequence object. `gles.array` objects require less processing and can be therefore slightly faster. If `gles.array` object is used, the dimension and type of its data are ignored and *size* and *type* are used instead.

glWeightPointerOESf(*sequence*)
Special Python version of `glWeightPointerOES` that accepts either a `gles.array` object or some other Python sequence object. Other parameters of `glWeightPointerOES` will be determined as

follows:

- *size* If *sequence* is an instance of `gles.array`, its dimension is used; otherwise the length of *sequence* is used.
- *type* `GL_FLOAT`
- *stride* 0

`glWeightPointerOESx`(*sequence*)
Special Python version of `glWeightPointerOES` that behaves exactly as `glWeightPointerOESf` except `GL_FIXED` is used as *type*.

3.8 `glcanvas` — UI Control for Displaying OpenGL ES Graphics

The `glcanvas` module provides a UI control, `GLCanvas`, for displaying OpenGL ES graphics. `GLCanvas` component is similar to the `appuifw` `Canvas` component that supports Symbian OS -level drawing.

Internally `GLCanvas` uses EGL for displaying the OpenGL ES graphics. EGL, as OpenGL ES, is a standard API defined by the Khronos Group (www.khronos.org). Specifically, `GLCanvas` uses an EGL window surface, which supports double-buffered rendering. It is possible to affect selection of the EGL config that is used to create the window surface; for details, see the documentation of the `GLCanvas` constructor.

`GLCanvas` instances also hold the OpenGL ES context object, which together with the surface, are needed for rendering. When one wants to render with a specific OpenGL ES context to a specific surface, they need to be *made current*. This also applies to `GLCanvas`, which has the `makeCurrent` method for this purpose. Generally, calling `makeCurrent` has to be done only if multiple `GLCanvas` objects are used in the same program, as each `GLCanvas` object is automatically made current when it is created and it remains current until it is destroyed or `makeCurrent` of some other `GLCanvas` object is called.

class `GLCanvas`(*redraw_callback,* [*event_callback=None, resize_callback=None, attributes=None*])
Constructs a new `GLCanvas` object that can be used as a UI control for displaying OpenGL ES graphics. Parameters *redraw_callback*, *event_callback*, and *resize_callback* have the same meaning as with `appuifw` module `Canvas`. Using *redraw_callback* to specify the OpenGL ES drawing is preferred as it will be automatically called by `drawNow` method.

Parameter *attributes* can be used to specify attributes used in EGL config selection. It must be a Python dictionary where keys are EGL attribute names (which are defined in the `glcanvas` module) and values are integers defining the desired attribute values. Unless specified in *attributes*, `EGL_BUFFER_SIZE` is set to value based on the display mode of the window owned by the underlying `CCoeControl` object and `EGL_DEPTH_SIZE` is set to 16. Attributes specified in *attributes* are given to `eglChooseConfig`. Refer to the EGL specification for a detailed list of config attributes and explanation of how the selection of EGL configs works.

The new `GLCanvas` object will be made current when the constructor returns so `makeCurrent` does not have to be called before starting to use OpenGL ES calls.

`bind`(*key_code, c* [*, ((x1, y1), (x2, y2))*])
Sets a callback to be called when a specific key is pressed or pointer event occurs. Parameter *key_code* should be one of the standard Symbian key codes defined in `key_codes`. Parameter *c* must be a callable object. The optional two co-ordinate tuple corresponds to the top left and bottom right points of the rectangle to be monitored for pointer events. This argument is ignored for Key events.
For different bind scenarios refer `Canvas`'s bind method.

`drawNow`()
Calls the redraw callback (if set) and then calls `eglSwapBuffers` to render and display the OpenGL ES graphics.

`makeCurrent`()
Makes this `GLCanvas` object current, meaning that it will be used to display the results of the subsequent OpenGL ES calls. In EGL terms this means that the EGL context and surface held by this object will be passed to `eglMakeCurrent`. Using `makeCurrent` makes it possible to use several `GLCanvas` objects in a single application: the receiver of the OpenGL ES calls can be switched with `makeCurrent` easily.

3.9 `sensor` — Module to access the device sensors.

3.9.1 Module for devices that support S60 Sensor API

The `Sensor` module offers direct access to a device's physical sensors. It has been tested with the following sensors:

- acceleration sensor: raises events about the 3-axes acceleration of the device
- tapping sensor: raises an event when the device was tapped twice on the front side
- rotation sensor: raises an event based on the orientation of the device.

Instead of passing on raised events, event filtering is also supported. Two examples of using an event filter are also provided by the `Sensor` module, namely the class `OrientationEventFilter` and `RotEventFilter`. Both filters can be used to raise events when the orientation changes in the device (For example, when it is turned to the right). The support is device dependent. For example, Nokia 5500 supports `OrientationEventFilter` and Nokia N95 supports `RotEventFilter`.

Note: The module `Sensor` is available from S60 3rd Edition onwards. (inclusive).

Module Level Functions

On the module level, `Sensor` provides the following functions:

`sensors``()`
Returns a dictionary containing all available sensors. The format of the dictionary is as follows:

```
{
  { 'sensor name 1': { 'id': sensor_id_1, 'category': category_id_1 } },
  { 'sensor name 2': { 'id': sensor_id_2, 'category': category_id_2 } },
  ...
}
```

with `sensor_id_X` and `category_id_X` being integer values.

Constants

The following `orientation` constants are used by the `OrientationEventFilter` class. Callbacks which have been connected to a `Sensor` object that utilises the `OrientationEventFilter` event filter will receive one of these constants as an argument upon a detected orientation change. The names of the constant are the side of the device that is currently turned upwards from your point of view. (For example `FRONT` means that the device is lying on its back - its front side is turned upwards.)

`orientation.TOP`
Represents the orientation where the device is held upwards.

`orientation.BOTTOM`
Represents the orientation where the device is held upside down.

`orientation.LEFT`
Represents the orientation where the side of the device that is left of the display is turned downwards.

`orientation.RIGHT`
Represents the orientation where the side of the device that is right of the display is turned downwards.

`orientation.FRONT`
Represents the orientation where the device is lying on its back that is, the front side points upwards.

`orientation.BACK`
Represents the orientation where the device is lying on its front that is, the back side points upwards.

Classes

The following classes are provided by the `Sensor` module:

class `Sensor`

The `Sensor` class represents a physical sensor which delivers (possibly filtered) events. By default, events are not filtered. A filter can be applied by using the `set_event_filter` method. An example for an event filter is given by `OrientationEventFilter`, which can be applied to an acceleration sensor of the device.

In case different filters must be used for the same physical sensor, different `Sensor` objects have to be created for the same physical sensor.

`__init__` (*sensor_id, category_id*)

Initialises the `Sensor` object. `sensor_id` and `category_id` must represent a valid sensor id and category id, respectively. This means that the ids passed on to `__init__` must also appear in the dictionary returned by the `sensors` function. In case `sensor_id` and `category_id` do not represent a valid sensor, the `connect` method will raise an exception.

`connect` (*callback*)

This method connects the sensor to the given `callback`. A sensor can only be connected to one callback, so this will destroy any pre-existing connection to another callback. If an event filter has been set, the events passed on to `callback` will first pass this event filter of the `Sensor` object. If the connection is properly established, this method returns 1, otherwise 0. **Note:** The connection can be established also if the callback does not exist or cannot be called for any other reason.

`disconnect`()

Disconnects this callback connection of the `Sensor` object. After a successful call to this method, a callback that has been previously connected through `connect` will not receive any more events. If a connection existed and is successfully removed, this method returns 1, otherwise 0.

`connected`()

Retrieves this `Sensor` object's connection status. Returns `True` if the sensor is connected, `False` otherwise.

`set_event_filter` (*event_filter*)

Sets an event filter for this `Sensor` object. After the event filter has been successfully installed, the connected callback of the `Sensor` object will receive only events that have passed the filter. `event_filter` must be derived from `EventFilter` in order to function properly. If a callback connection has already been established before calling this method, the connection will be re-established after the event filter has been installed.

class `EventFilter`

The `EventFilter` class provides a generic interface for event filters. The default implementation only passes events on, that is, events are not filtered. Classes deriving from `EventFilter` can decide if an event should be delivered at all as well as they can alter the data that is passed on to the callback.

`callback`

This is where the callback of the event filter is stored. In case, the `EventFilter` object is used together with a `Sensor` object, the `Sensor` object will handle correct setting of this variable.

`__init__`()

Initialises the event filter object. The `callback` member is initialised to `None`.

`__del__`()

Destructs the event filter object. This method calls `cleanup`, which can be overridden by deriving classes to clean up resources.

`event` (*data*)

This method is the place where event filtering takes place, and hence this method should be overridden by deriving classes. Overridden `event` methods can deliver their own data to the callback, the data delivered can be `data` or any other set of data. In case the event is decided to be delivered, overriding instances must call `self.callback`, which by default takes one argument.

`cleanup`()

Cleans up any resources needed by the event filter. The default implementation does not need this feature. This method is called by the destructor `__del__`.

class `OrientationEventFilter`

Derived from `EventFilter`. This event filter is meant to be used together with the acceleration sensors of the device. Note that it is not required to use it with any other sensor type. It generates events when the devices orientation changes. For example, if it is turned from the upright position to lying on the back side. If an `OrientationEventFiler` is used with a `Sensor` object, the callback of the `Sensor` object will not receive the raw acceleration data as an argument, but only one of the `orientation` constants, representing the new orientation of the device. In case the algorithm needs calibration on the device to be used, you must check the `OrientationCalibration` variables in the file `sensor.py`.

`__init__``()`

Initialises the `OrientationEventFilter` object.

`event`(*sensor_val*)

Overridden method. Filters 3-axis acceleration events such that it detects orientation changes. Only upon detection of such an orientation change, the callback is invoked. The argument passed to the callback is a value from the `orientation` constants of this module.

`cleanup``()`

Cleans up the timer resource of this filter. This will be called by destructor of the `EventFilter` class.

class `RotEventFilter`

Derived from `EventFilter`.

This event filter generates events when the devices orientation changes. For example, if it is turned from the left side up position to right side up position. This sensor is resident. For example, in Nokia N95.

`event`(*sensor_val*)

Overridden method. Upon detection of an orientation change, the callback is invoked. The argument passed to the callback is a value from this module's `orientation` constants.

3.9.2 Module for devices that support S60 Sensor FrameWork

The Python S60 sensor module supports access of sensors on the devices that have S60 Sensor Framework libraries. The S60 Sensor Framework is introduced in S60 Fifth Edition. It is also backported to S60 Third Edition, Feature Pack 2 for some mobile devices and to the Nokia E66 device, which is an S60 3rd Edition, Feature Pack 1 device with sensor APIs based on the S60 Sensor Framework.

The sensor module offers direct access to physical sensors of a device. The following sensor channels are supported by the sensor module, provided the device supports them:

- Accelerometer XYZ sensor channel
- Rotation sensor channel
- Orientation sensor channel
- Accelerometer double-tap sensor channel
- Proximity monitor sensor channel
- Ambient light sensor channel
- Magnetic North sensor channel
- Magnetometer XYZ sensor channel.

The following table lists the sensors available on different S60 devices:

These sensors are mapped to a class using which the sensor channel data can be accessed. To access a particular sensor data, an object of the respective class is created. Then the data callback function can be set using the `set_callback()` method. To start and stop receiving updates the `start_listening()` and `stop_listening()` methods can be used.

	Accelerometer double tap	Accelerometer XYZ	Orientation	Rotation	Ambient light	Magnetic north	Proximity monitor	Magnetometer XYZ	S60 platform
N85	x	x	x	x	NA	NA	NA	NA	3rdFP2
E66	x	x	x	x	NA	NA	NA	NA	3rdFP1
N96	x	x	x	x	NA	NA	NA	NA	3rdFP2
E75	x	x	x	x	NA	NA	NA	NA	3rdFP2
6720	NA	x	x	NA	x	x	NA	NA	3rdFP2
5800	x	x	x	x	x	NA	x	NA	5thEd
6210	x	x	x	x	NA	x	NA	x	3rdFP2
6710	x	x	x	x	NA	x	NA	x	3rdFP2
E55	x	x	NA	x	x	x	NA	x	3rdFP2

Module Level Functions

List Channels
Function signature: `list_channels`

This returns a list of dictionaries containing all the available sensors on the device. The returned dictionary has the following format:

```
[
 {'id': channel id, 'type': channel type, 'name': channel name}

 {'id': channel id, 'type': channel type, 'name': channel name}

....
]
```

where, `channel_id`, `channel_type`, and `channel_name` have strings as values of the respective channels.

Query Logical Name
Function signature : `get_logicalname(<DataLookupClass>, value)`

This function can be used for querying the logical name based on value. The file **sensor_defs.py** has the mapping of different sensor properties to their respective hex/decimal values. The following table contains the sensor classes, supported by `get_logicalname()` and the respective data lookup classes.

Sensor Class	DataLookupClass
`ProximityMonitor`	`ProximityState`
`OrientationData`	`DeviceOrientation`
`AmbientLightData`	`AmbientLightData`
`AccelerometerDoubleTappingData`	`AccelerometerDirection`

Base Class

The base class to all types of sensor class is `_Sensor`. This class provides the methods: `set_callback`, `start_listening`, and `stop_listening` that are common to all the sensor class objects. The individual sensor class objects must be used for a specific sensor.

Object Creation
Function signature: `__init__([data_filter=None])`

The `data_filter` argument is only applicable for `*XYZAxisData` and `RotationData` sensor classes.

Possible Values: `MedianFilter()`, `LowPassFilter()`

- If nothing is passed then the data remains in its present condition without any filtering.
- `MedianFilter` and `LowPassFilter` are standard noise filtering algorithms that provide a smoother form of a signal removing the short-term oscillations, leaving only the long-term trend.

Set Data and Error Callback
Function signature: `set_callback(data_callback, [error_callback=None])`

Sets the data and error callback function. The error callback function will get an argument that contains a map with Channel ID and error string. The data callback function is not passed with any arguments.

Open and Listen
Function signature: `start_listening()`

Opens the sensor channel and start listening. Returns `True` on success and `False` on failure.

Stop and Close
Function signature: `stop_listening()`

Stop listening to the open channel and close the channel. To start receiving updates again the `start_listening` method can be called on the same sensor object.

Set/Get Sensor Channel Property

Each sensor class has methods which can be used to set or get the sensor channel properties like data rate, measure range, axis active etc...

Class Attributes

The sensor classes have one or more attributes which contains the data returned by the respective sensor. These attributes will be set before the registered data callback function is called and can be accessed using the respective sensor class object.

class AccelerometerXYZAxisData

- Detects movement gestures, such as moving the device up or down.
- Inherits from the `_Sensor` base class.

Class Attributes

- **x**: X-axis value
- **y**: Y-axis value
- **z**: Z-axis value

Set/Get Property

This sensor class provides additional functions that can be used to set or get some of the properties specific to this sensor.

The following table lists the set/get properties of the sensor class:

Example

Set/Get properties	Description
`get_available_data_rates()`	Returns the data rates that can be used for this channel.
`set_data_rate(data_rate)`	Sets the data rate to be used for this channel.
`get_data_rate()`	Returns the current data rate for this channel.
`set_measure_range(measurerange)`	Sets the measure range. Pass 0 to set +-2g, 1 for +-8g
`get_measure_range()`	Returns the current measure range. Returns 0 for +-2g, 1 for +-8g

```
from sensor import *
import e32
import time


class DemoApp():

    def __init__(self):
        self.accelerometer = \
            AccelerometerXYZAxisData(data_filter=LowPassFilter())
        self.accelerometer.set_callback(data_callback=self.my_callback)
        self.counter = 0

    def my_callback(self):
        # For stream sensor data the callback is hit 35 times per sec(On 5800).
        # The device cannot handle resource hungry operations like print in the
        # callback function for such high frequencies. A workaround is to
        # sample the data as demonstrated below.
        if self.counter % 5 == 0:
            print  "X:%s, Y:%s, Z:%s" % (self.accelerometer.x,
                                         self.accelerometer.y, self.accelerometer.z)
            print "Timestamp:", self.accelerometer.timestamp
        self.counter = self.counter + 1

    def run(self):
        self.accelerometer.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(5)
    d.accelerometer.stop_listening()
    print "Exiting Accelorometer"
```

class AccelerometerDoubleTappingData

- Detects a double-tap on the device where the taps occur in quick succession, in the same direction.
- Inherits from the `_Sensor` base class.

Class Attribute

Direction: Hex value indicating the tap direction. The direction can be determined in human readable form using `get_logicalname` API and class name as `AccelerometerDirection`.

Set/Get Property

This sensor class provides additional functions that can be used to set or get some of the properties specific to this sensor.

The following table lists the set/get properties of the sensor class:

Set/Get properties	Description
`get_axis_active()`	Returns **x**, **y**, **z** values: 1 if axis is active else 0.
`set_axis_active([x=None, y=None, z=None])`	Sets one or more axis as active. Pass 1 to set the axis and 0 to disable it.
`get_properties()`	Returns a dictionary with `"DoubleTapThreshold"`, `"DoubleTapDuration"`, `"DoubleTapLatency"`, `"DoubleTapInterval"` as the keys and their respective values.
`set_properties([DoubleTapThreshold = None, DoubleTapDuration = None, DoubleTapLatency = None, DoubleTapInterval = None])`	Sets the tap related properties.

Example

```
from sensor import *
import e32


class DemoApp():

    def __init__(self):
        self.doubletap = AccelerometerDoubleTappingData()
        self.doubletap.set_axis_active(x=0, y=1, z=1)
        print "Active Axis are: ", self.doubletap.get_axis_active()
        self.doubletap.set_callback(data_callback=self.my_callback)

    def my_callback(self):
        print "Raw Direction value:", self.doubletap.direction
        print "Direction:", get_logicalname(AccelerometerDirection,
                                            self.doubletap.direction)
        print "Timestamp:", self.doubletap.timestamp

    def run(self):
        self.doubletap.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(15)
    d.doubletap.stop_listening()
    print "Exiting Double Tap"
```

class MagnetometerXYZAxisData

- Indicates the strength of the geomagnetic flux density in the X, Y and Z axes.
- Only calibrated axis data is exposed right now and not raw data.
- Inherits from the `_Sensor` base class.

Class Attributes

- **x** : X-axis value

- **y** : Y-axis value
- **z** : Z-axis value
- **calib_level**: Indicates the calibration level.
  - **Possible values:**
    - `0` - Not calibrated
    - `1` - Low calibration
    - `2` - Medium calibration
    - `3` - High accuracy

Example

```
from sensor import *
import e32


class DemoApp():

    def __init__(self):
        self.magnetometer = \
            MagnetometerXYZAxisData(data_filter=LowPassFilter())
        self.magnetometer.set_callback(data_callback=self.my_callback)
        self.counter = 0

    def my_callback(self):
        # For stream sensor data the callback is hit 35 times per sec(On 5800).
        # The device cannot handle resource hungry operations like print in the
        # callback function for such high frequencies. A workaround is to
        # sample the data as demonstrated below.
        if self.counter % 5 == 0:
            print  "Calib:", self.magnetometer.calib_level
            print "X:%s, Y:%s, Z:%s" % (self.magnetometer.x,
                                        self.magnetometer.y, self.magnetometer.z)
            print "Timestamp:", self.magnetometer.timestamp
        self.counter = self.counter + 1

    def run(self):
        self.magnetometer.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(5)
    d.magnetometer.stop_listening()
    print "Exiting MagnetometerAxis"
```

class MagneticNorthData

- Indicates the number of degrees between the device and magnetic north.
- Inherits from the `_Sensor` base class.

Class Attribute

Azimuth: 0 to 359 clockwise degrees from magnetic north.

Example

```
from sensor import *
import e32


class DemoApp():

    def __init__(self):
        self.magnetic_north = MagneticNorthData()
        self.magnetic_north.set_callback(data_callback=self.my_callback)

    def my_callback(self):
        azimuth = str(self.magnetic_north.azimuth)
        print "Azimuth:", azimuth
        print "Timestamp:", timestamp

    def run(self):
        self.magnetic_north.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(1)
    d.magnetic_north.stop_listening()
    print "Exiting MagneticNorth"
```

class AmbientLightData

- Indicates the current light level.
- Inherits from the `_Sensor` base class.

Class Attribute

Ambient_light: 0 to 100 percent light. To get the logical names use `get_logicalname` API with class name as `AmbientLightData`.

Example

```
from sensor import *
import e32


class DemoApp():

    def __init__(self):
        self.ALS = AmbientLightData()
        self.ALS.set_callback(data_callback=self.my_callback)

    def my_callback(self):
        print 'ALS:', get_logicalname(AmbientLightData,
                                      self.ALS.ambient_light)
        print 'Timestamp:', self.ALS.timestamp

    def run(self):
        self.ALS.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(30)
    d.ALS.stop_listening()
    print "Exiting Ambient Light"
```

class ProximityMonitor

- Indicates how close the device is to your hand or ear.
- Inherits from the `_Sensor` base class.

Class Attribute

Proximity_state: The possible values are 0, 1 and 2. To get the logical names of these values use `get_logicalname` API with `ProximityState` as the class name.

Example

```
from sensor import *
import e32


class DemoApp():

    def __init__(self):
        self.proxi = ProximityMonitor()
        self.proxi.set_callback(data_callback=self.my_callback)

    def my_callback(self):
        print 'Proxi:', get_logicalname(ProximityState,
                                        self.proxi.proximity_state)
        print 'Timestamp:', self.proxi.timestamp

    def run(self):
        self.proxi.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(10)
    d.proxi.stop_listening()
    print "After Stop Listening"
    e32.ao_sleep(5)
    print "Exiting Proximity"
```

class OrientationData

- Indicates the orientation of the device, for example: display up or down.
- Inherits from the `_Sensor` base class.

Class Attribute

device_orientation: Values range from -1 to 6. To determine the logical names of these values `get_logicalname` API can be used with class name as `DeviceOrientation`.

Example

```
from sensor import *
import e32


class DemoApp():

    def __init__(self):
        self.orientation = OrientationData()
        self.orientation.set_callback(data_callback=self.my_callback)

    def my_callback(self):
        print 'Orientation:', get_logicalname(DeviceOrientation,
                                  self.orientation.device_orientation)
        print 'Timestamp:', self.orientation.timestamp

    def run(self):
        self.orientation.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(10)
    d.orientation.stop_listening()
    print "Exiting Orientation"
```

class RotationData

- Detects the rotation of the device about each axis.
- Inherits from the `_Sensor` base class.

Class Attribute

- **x**: X-axis value
- **y**: Y-axis value
- **z**: Z-axis value

Example

```
from sensor import *
import e32


class DemoApp():

    def __init__(self):
        self.rotation = RotationData()
        self.rotation.set_callback(data_callback=self.my_callback)
        self.counter = 0

    def my_callback(self):
        # For stream sensor data the callback is hit 35 times per sec(On 5800).
        # The device cannot handle resource hungry operations like print in the
        # callback function for such high frequencies. A workaround is to
        # sample the data as demonstrated below.
        if self.counter % 5 == 0:
            print "X:%s, Y:%s, Z:%s" % (self.rotation.x,
                                        self.rotation.y, self.rotation.z)
            print "Timestamp:", self.rotation.timestamp
        self.counter = self.counter + 1

    def run(self):
        self.rotation.start_listening()

if __name__ == '__main__':
    d = DemoApp()
    d.run()
    e32.ao_sleep(5)
    d.rotation.stop_listening()
    print "Exiting Rotation"
```

CHAPTER

FOUR

Audio and Communication Services

4.1 audio — An audio related services package

The audio module enables recording and playing audio files and access to device text-to-speech engine. The audio module supports all the formats supported by the device, typically: WAV, AMR, MIDI, MP3, AAC, and Real Audio[1]. For more information on the audio types supported by different devices, see the *Forum Nokia* Web site [5] and *S60 Platform* Web site [6].

The following Sound class static methods are defined in the audio module:

Sound.open (*filename*)
Returns a new initialized Sound object with the named file opened. Note that *filename* should be a full Unicode path name and must also include the file extension, for example u'c:\\foo.wav'.

The following data items for state information are available in audio:

ENotReady
The Sound object has been constructed but no audio file is open.

EOpen
An audio file is open but no playing or recording operation is in progress.

EPlaying
An audio file is playing.

ERecording
An audio file is being recorded.

The following data item is provided for continuous playback of an audio file:

KMdaRepeatForever
Possible value for *times* parameter in open.

The following method is available in the audio module:

say (*text, prefix=audio.TTS_PREFIX*)
Passes the text to the device text-to-speech engine. The default prefix is the text-to-speech prefix "(tts)".

text should be either Unicode or a UTF-8 encoded plain string. The speech synthesizer pronounces the text according to the current device language.

4.1.1 Sound Objects

class Sound
Sound objects have the following functions:

play ([*times=1, interval=0, callback=None*])
Starts playback of an audio file from the beginning. Without the parameters *times* and *interval* it

[1] The dynamically loaded audio codec for the sound file is based on the MIME-type information inside the audio file and file extension.

plays the audio file one time. *times* defines the number of times the audio file is played, the default being *1*. If the audio file is played several times, *interval* gives the time interval between the subsequent plays in microseconds.

The optional callback is called when the playing starts and when the end of the sound file is reached. The callback should take three parameters: the previous state, the current state and the possible error code. The possible states given as parameters to the callback are data items in the module `audio`.

Other issues:

- Calling `play(audio.KMdaRepeatForever)` will repeat the file forever.
- If an audio file is played but not stopped before exiting, the Python script will leave audio playing on; therefore `stop` needs to be called explicitly prior to exit.
- Currently the module does not support playing simultaneous audio files, calling `play` to a second `Sound` instance while another audio file is playing, stops the earlier audio file and starts to play the second `Sound` instance.
- Calling `play` while a telephone call is ongoing plays the sound file to uplink. In some devices the sound file is also played to the device speaker.
- Calling `play` when already playing or recording results in `RuntimeError`. Calling `stop` prior to `play` will prevent this from happening.

`stop``()`
Stops playback or recording of an audio file.

`record``()`
Starts recording audio data to a file. If the file already exists, the operation appends to the file. For Nokia devices, WAV is typically supported for recording. For more information on the audio types supported by different devices, see the *Forum Nokia* Web site [5] and *S60 Platform* Web site [6].

Other issues:

- Calling `record` while a telephone call is ongoing starts the recording of the telephone call.
- Calling `record` when already playing or recording results in `RuntimeError`. Calling `stop` prior to `record` will prevent this from happening.

`close``()`
Closes an opened audio file.

`state``()`
Returns the current state of the `Sound` type instance. The different states (constants) are defined in the `audio` module. The possible states[2] are:

- `ENotReady`
  The Sound object has been constructed but no audio file is open.
- `EOpen`
  An audio file is open but no playing or recording operation is in progress.
- `EPlaying`
  An audio file is playing.
- `ERecording`
  An audio file is being recorded.

`max_volume``()`
Returns the maximum volume of the device.

`set_volume``(`*volume*`)`
Sets the volume. If the given volume is negative, then the volume is set to zero which mutes the device. If the volume is greater than `max_volume`, then `max_volume` is used.

`current_volume``()`
Returns the current volume set.

`duration``()`
Returns the duration of the file in microseconds.

`set_position``(`*microseconds*`)`
Set the position for the playhead.

`current_position``()`
Returns the current playhead position in microseconds.

[2] Descriptions for these options are based on information found in S60 SDK documentation [4].

4.2 telephone — Telephone services

This module provides an API to a telephone.

Since the users of the device can also hang-up the phone explicitly, they might affect the current status of the call. In addition, using this extension in an emulator has no effect since no calls can be connected.

The telephone module has the following functions:

dial(*number*)
Dials the number set in *number*. *number* is a string, for example u'+358501234567' where '+' is the international prefix, '358' is the country code, '50' is the mobile network code (or the area code), and '1234567' is the subscriber number. If there is an ongoing phone call prior to calling dial from Python, then the earlier call is put on hold and a new call is established. Calling dial multiple times when, for example, the first call has been answered and a line has been established results in subsequent calls not being connected.

hang_up()
Hangs up if a call initiated by dial is in process. If this call has already been finished, SymbianError: KErrNotReady is raised.

incoming_call()
Wait for incoming call, returns immediately. If a call arrives, answer can be called to answer the call. Without the invocation of function incoming_call, the function answer has no effect.

answer()
Answers an incoming call - see also incoming_call.

call_state(*callable*)
The callable will be called when there are changes in the telephone line (lines) in the device. The argument for the call is a tuple with first item the possible new state and the second item, the possible incoming call number as a Unicode string.

The possible states in the tuple are defined as telephone module constants.

The following data items for state information are available in telephone[3]:

EStatusUnknown
Indicates that the status is unknown.

EStatusIdle
Idle line status (no active calls).

EStatusDialling
Call dialling status.

EStatusRinging
Call ringing status.

EStatusAnswering
Call answering status.

EStatusConnecting
Call connecting status.

EStatusConnected
Call connected status.

EStatusReconnectPending
Call is undergoing temporary channel loss and it may or may not be reconnected.

EStatusDisconnecting
Call disconnecting status.

EStatusHold
Call on hold.

[3] The descriptions are taken from the S60 SDK documentation [4]

EStatusTransferring
Call is transferring.

EStatusTransferAlerting
Call in transfer is alerting the remote party.

4.3 `messaging` — A messaging services package

The `messaging` module offers APIs to messaging services. Currently, the `messaging` module has functions:

sms_send(*number, msg,* [*encoding='7bit', callback=None, name=""*])
Sends an SMS message with body text *msg*[4] (Unicode)to telephone number *number* (string).

The optional parameter *encoding* is used to define encoding in the message. The parameter values can be `'7bit'`,`'8bit'` or `'UCS2'`.

The optional parameter *callback* is invoked with the current status of the send operation as parameter. The possible states are data items in the module `messaging`. Invoking another send while a previous send request is ongoing will result in `RuntimeError` being raised.

If the callback is not given, the `sms_send` function will block until the message in the queue is either deleted or the sending has failed[5].

The optional parameter *name* will be shown in the sent item message entry as recipient's name after successfully sending message to *number*. If this parameter is not specified, then the recipient's phone number will be shown in the sent item message entry[6].

mms_send(*number, msg,* [*attachment=None*])
Sends an MMS message with body text *msg* (Unicode) to telephone number *number* (string). The optional parameter *attachment* is full path to e.g. image file attached to the message.

The following data items for SMS sending state information are available in the module `messaging`:

ECreated

EMovedToOutBox

EScheduledForSend

ESent
The SMS message has been sent.

EDeleted
The SMS message has been deleted from device's outbox queue. The `sms_send` operation has finalized and subsequent SMS sending is possible.

EScheduleFailed

ESendFailed
This state information is returned when the SMS subsystem has tried to send the message several times in vain. The `sms_send` operation has finalized and subsequent SMS sending is possible.

ENoServiceCentre
This state information is returned by the SMS subsystem in S60 3.x emulator. In emulator this indicates that the `sms_send` operation has finalized and subsequent SMS sending is possible.

EFatalServerError

The underlying messaging subsystem in S60 devices might give error messages to the user if the device is not connected to a network while trying to send a message – An "SMS send failed!" note is a common error message.

[4]The maximum length of a message that can be sent using sms_send function is either 39015 characters or Max network capacity whichever is lower.

[5]Please note that this blocking might last for several minutes and hence supplying the callback might be more suitable in many cases.

[6]The name can be of maximum 60 characters and will be shown in the sent item message entry as specified by sender without making any check in the contact database.

When sending messages in offline-mode or with no network connection these messages are actually added to an outgoing message queue and they might be sent if the device is later on connected to a suitable network[7]. This occurs despite the possibly misleading error messages. The current network conditions can be checked e.g. with `sysinfo.active_profile()` and `sysinfo.signal_bars()` invocations.

The following is example code for state information processing with `sms_send` operation:

```
>>> import messaging
>>>
>>> def cb(state):
...   if state==messaging.ESent:
...     print "**Message was sent**"
...   if state==messaging.ESendFailed:
...     print "**Something went wrong - Truly sorry for this**"
...
>>> messaging.sms_send("1234567", "Hello from PyS60!", '7bit', cb, "Mary")
>>> **Message was sent** # This is printed from the callback
```

4.4 `inbox` — Interface to device inbox

The `inbox` module offers APIs to device inbox, outbox, sent and drafts folders. Currently, the `inbox` module supports only SMS handling and notifications of incoming messages to the device inbox.

class Inbox([*folder_type*])
Create an `Inbox` object.

The optional parameter `folder_type` defines the type of the folder to which the created `Inbox` object has access to. The default is the device's inbox folder, `inbox.EInbox`.

The following data items are available in the `inbox` module to define the type of the folder for `Inbox` objects:

EInbox
The device's inbox folder.

EOutbox
The device's outbox folder.

ESent
The sent messages folder.

EDraft
The draft messages folder.

4.4.1 Inbox Objects

`Inbox` objects have the following functions:

sms_messages()
Returns a list of SMS message IDs in device inbox.

content(*sms_id*)
Retrieve the SMS message content in Unicode.

time(*sms_id*)
Retrieve the SMS message time of arrival in seconds since epoch.

address(*sms_id*)
Retrieve the SMS message sender address in Unicode.

[7]Note also that prior this the user of the device can explicitly delete the messages from the native messaging application. The amount of resending is approx. 4 times – After this the sending operation is cancelled and the user of the device will see a visual cue of the failure in the status pane.

delete(*sms_id*)
Delete the SMS message from inbox.

unread(*sms_id*)
Returns the status (1=unread, 0=read) of the SMS with id.

set_unread(*sms_id, status*)
Set the status (1=unread, 0=read) of the SMS with id.

bind(*callable*)
Bind a callback to receive new message events in device inbox. When a new message arrives to the device inbox the `callback` gets called with the received message ID. The received message can be other than an SMS message.

If the message received is deleted immediately after e.g. checking the message content, the ”new message” sound and dialog are not activated. This functionality might be useful in notification type of applications.

Examples:

```
>>> import inbox
>>> i=inbox.Inbox() # Give inbox.ESent as parameter for sent SMSes
>>> m=i.sms_messages()
>>> i.content(m[0])
u'foobar'
>>> i.time(m[0])
1130267365.03125
>>> i.address(m[0])
u'John Doe'
>>> i.delete(m[0])
>>>
```

```
>>> import inbox
>>> id=0
>>> def cb(id_cb):
...   global id
...   id=id_cb
...
>>> i=inbox.Inbox()
>>> i.bind(cb)
>>> # Send an SMS to your inbox here. The "id" gets updated
>>> i.address(id)
u'John Doe'
>>> i.content(id)
u'print 1'
>>>
```

4.5 `location` — GSM location information

The `location` module offers APIs to location information related services. Currently, the `location` has one function:

Note: Location module requires capabilities ReadDeviceData, ReadUserData and Location.

gsm_location()
Retrieves GSM location information: Mobile Country Code, Mobile Network Code, Location Area Code, and Cell ID. A location area normally consists of several base stations. It is the area where the terminal can move without notifying the network about its exact position. mcc and mnc together form a unique identification number of the network into which the phone is logged.

4.5.1 Examples

Here is an example of how to use the `location` package to fetch the location information:

```
>>> import location
>>> print location.gsm_location()
```

4.6 `positioning` — Simplified interface to the position information

The `positioning` module provides basic access to the S60 position information [8]. The module can be e.g. used to access position information provided by external Bluetooth GPS-devices and by built-in GPS-receivers[9] from S60 devices.

The module offers a large amount of information (cost of service, device power consumption etc.) about accessible positioning devices (like GPS-modules), position, course, accuracy and satellite information (depending on the position device used) and much more. This module can also be used to obtain device/vendor specific extended information.

Note: The module `position` requires Location capability.

The following data items are available in `positioning`:

`POSITION_INTERVAL`
The time interval (in microseconds) between the `position` function callback invocation. The default value set is 1000000 microseconds (= 1 second)

The `positioning` module has the following functions (for examples of the values returned, see Section 4.6.1):

`modules``()`
Get information about available positioning modules.

`default_module``()`
Get default module id.

`module_info``(`*module_id*`)`
Get detailed information about the specified module.

`select_module``(`*module_id*`)`
Select a module.

`set_requestors``(`*requestors*`)`
Set the *requestors* of the service (at least one must be set).

`position``(`*course=0,satellites=0,callback=None, interval=positioning.POSITION_INTERVAL, partial=0*`)`
By default, returns the position information in a dictionary. With *course* and/or *satellites* set to 1, information about course and satellites is also returned (if available).

With no *callback* provided, this call blocks until the position information is available.

The call returns immediately if a valid *callback* function is given. This *callback* function is then invoked with the specified time *interval* (in microseconds) in between the invocations. The *callback* function is called with the the current position information as parameter.

If *partial* update is set to 1, the function might return e.g. information about satellites before the final location fix has been calculated.

For an example of the dictionary returned and the detailed keys, see Section 4.6.1.

`stop_position``()`
Stops an ongoing `position` request.

[8] For details, please see the Location Acquisition API in the S60 API documentation. The Location Acquisition API gathers different positioning technologies together to be used through a consistent interface.

[9] For more information on GPS, please see http://en.wikipedia.org/wiki/Global_Positioning_System.

last_position()
Get last position information. This method returns the cached position information if it is available.

4.6.1 Example

The following example (invoked in a Nokia N95 device) demonstrates how to use the Python `positioning` module to obtain information about the positioning technologies in the device:

```
>>> import positioning
>>> positioning.modules()
[{'available': 0, 'id': 270526873, 'name': u'Bluetooth GPS'}, {'available': 1, '
id': 270526858, 'name': u'Integrated GPS'}, {'available': 1, 'id': 270559509, 'n
ame': u'Network based'}]
>>> positioning.default_module()
270526858
>>> positioning.module_info(270526858)
{'available': 1, 'status': {'data_quality': 3, 'device_status': 7}, 'version': u
'1.00(0)', 'name': u'Integrated GPS', 'position_quality': {'vertical_accuracy':
10.0, 'time_to_first_fix': 1000000L, 'cost': 1, 'time_to_next_fix': 1000000L, 'h
orizontal_accuracy': 10.0, 'power_consumption': 3}, 'technology': 1, 'id': 27052
6858, 'capabilities': 127, 'location': 1}
>>>
```

The following example demonstrates how to use the Python `positioning` module.

```
# information about available positioning modules
print "***available modules***"
print positioning.modules()
print ""

# id of the default positioning module
print "***default module***"
print positioning.default_module()
print ""

# detailed information about the default positioning module
print "***detailed module info***"
print positioning.module_info(positioning.default_module())
print ""

# select a module (in practise, selecting default module has no
# relevance.).
positioning.select_module(positioning.default_module())

# set requestors.
# at least one requestor must be set before requesting the
# current position or last position.
# the last requestor must always be service requestor
# (whether or not there are other requestors).
positioning.set_requestors([{"type":"service",
                             "format":"application",
                             "data":"test_app"}])

# get the last position.
print positioning.last_position()
```

An example dictionary returned/printed from the above call to last_position function could be as follows

```
{'vertical_accuracy':59.0,'time':1206530248.329,'latitude':12.956741,'altitude':
811.0,'horizontal_accuracy':41.77254404,'longitude':77.715568724}

# get the last position if the device's position has not previously been
# discovered.
print positioning.last_position()
```

An example dictionary returned/printed from the above call to last_position function could be as follows

```
{'vertical_accuracy':NaN,'time':1206530248.329,'latitude':NaN,'altitude':NaN,
'horizontal_accuracy':NaN,'longitude':NaN}

# Example 1. Blocking call

# get the position.
# note that the first position()-call may take a long time
# (because of gps technology).
print "***position info***"
print positioning.position()
print ""

# re-get the position.
# this call should be much quicker.
# ask also course and satellite information.
print "***course and satellites***"
print positioning.position(course=1,satellites=1)
print ""

# Example 2. Non-blocking call

def cb(event):
    print "---"
    print event
    print "---"

print "***starts the position feed***"
print positioning.position(course=1,satellites=1,
                           callback=cb, interval=500000,
                           partial=0)
```

An example dictionary returned/printed from the above example script could be as follows:

```
{'satellites': {'horizontal_dop': 2.34999990463257, 'used_satellites': 5, 'verti
cal_dop': 2.29999995231628, 'time': 1187167353.0, 'satellites': 11, 'time_dop':
1.26999998092651}, 'position': {'latitude': 60.217033666473, 'altitude': 42.0, '
vertical_accuracy': 58.0, 'longitude': 24.878942093007, 'horizontal_accuracy': 4
7.531005859375}, 'course': {'speed': 0.0500000007450581, 'heading': 68.959999084
4727, 'heading_accuracy': 359.989990234375, 'speed_accuracy': NaN}}
```

To run the script in the emulator you must configure PSY emulation from your emulator (SimPSYConfigurator → Select Config File → <some config files>or Tools → Position).

4.7 `btsocket` — Provides Bluetooth (BT) support

The `socket` module of the previous PyS60 releases has been renamed as `btsocket`. For information on the usage of this module refer to Ensymble README. The following related constants and functions are defined:

Note: In release 1.0 the functions `bt_advertise_service`, `bt_obex_receive`, and `bt_rfcomm_get_available_server_channel` incorrectly expected to be given the internal `e32socket.socket` object as the socket parameter instead of the proper `socket` object. Now the functions work correctly. The old calling convention is still supported but it is deprecated and may be removed in a future release.

AF_BT
Represents the Bluetooth address family.

BTPROTO_RFCOMM
This constant represents the Bluetooth protocol RFCOMM.

RFCOMM

OBEX
Bluetooth service classes supported by `bt_advertise_service`.

AUTH

ENCRYPT

AUTHOR
Bluetooth security mode flags.

bt_advertise_service(*name, socket, flag, class*)
Sets a service advertising the service *name* (Unicode) on local channel that is bound to *socket*. If *flag* is `True`, the advertising is turned on, otherwise it is turned off. The service class to be advertised is either `RFCOMM` or `OBEX`.

bt_discover([*address*])
Performs the Bluetooth device discovery (if the optional BT device address is not given) and the discovery of RFCOMM class services on the chosen device. Returns a pair: BT device address, dictionary of services, where Unicode service name is the key and the corresponding port is the value.

bt_obex_discover([*address*])
Same as `discover`, but for discovery of OBEX class services on the chosen device.

bt_obex_send_file(*address, channel, filename*)
Sends file *filename* (Unicode) wrapped into an OBEX object to remote *address*, *channel*.

bt_obex_receive(*socket, filename*)
Receives a file as an OBEX object, unwraps and stores it into *filename* (Unicode). *socket* is a bound `OBEX` socket.

bt_rfcomm_get_available_server_channel(*socket*)
Returns an available RFCOMM server channel for *socket*.

set_security(*socket, mode*)
Sets the security level of the given bound *socket*. The *mode* is an integer flag that is formed using a binary `or` operation of one or more of: `AUTH` (authentication), `ENCRYPT`, `AUTHOR` (authorization). Example: `set_security(s, AUTH | AUTHOR)`.

Note: When listening to a Bluetooth socket on the phone, it is necessary to set the security level.

For examples on the usage of these functions, see Programming with Python for S60 Platform [**?**].

Setting default Access Point (AP) has been added to the standard `socket` module. The following related constants and functions are defined:

select_access_point()
This opens popup selection where access points are listed and can be selected. Returns selected access point id.

access_point(*apid*)
This creates access point object by given apid. Returns access point object.

set_default_access_point(*apo*)
This sets the default access point that is used when socket is opened. Setting *apo* to "None" will clear default access point.

access_points()
This lists access points id's and names that are available.

Example 1:

```
import btsocket
#access point is selected from the list
apid = btsocket.select_access_point()
apo = btsocket.access_point(apid)
btsocket.set_default_access_point(apo)

s = btsocket.socket(btsocket.AF_INET, btsocket.SOCK_STREAM)
print apo.ip()
s.connect(('www.sourceforge.net',80))
s.send('GET /\r\n\r\n')
s.recv(100)
s.close()
apo.stop()
```

Example 2:

```
import btsocket
#Access point id is already known
apo = btsocket.access_point(1)
btsocket.set_default_access_point(apo)

s = btsocket.socket(btsocket.AF_INET, btsocket.SOCK_STREAM)
s.connect(('www.sourceforge.net',80))
s.send('GET /\r\n\r\n')
s.recv(100)
s.close()
apo.stop()
```

Example 3:

```
import btsocket
#display interface ip.
#access point is selected from the list
apid = btsocket.select_access_point()
apo = btsocket.access_point(apid)
apo.start()
#Note that ip-address is given by operator, if static ip-address is not defined,
#when connection is started
print apo.ip()
#When connection is closed dynamic ip-address is released
apo.stop()
```

CHAPTER

FIVE

Data Management

5.1 `contacts` — A contacts related services package

The `contacts` module offers an API to address book services allowing the creation of contact information databases. The `contacts` module represents a Symbian contact database as a dictionary-like `ContactDb` object, which contains `Contact` objects and which is indexed using the unique IDs of those objects. A `Contact` object is itself a list-like object, which contains `ContactField` objects and which is indexed using the field indices. Unique IDs and field indices are integers. The `ContactDb` object supports a limited subset of dictionary functionality. Therefore, only `__iter__`, `__getitem__`, `__delitem__`,`__len__`, `keys`, `values`, and `items` are included.

`ContactDb` objects represent a live view into the database. If a contact is changed outside your Python application, the changes are visible immediately, and conversely any changes you commit into the database are visible immediately to other applications. It is possible to lock a contact for editing, which will prevent other applications from modifying the contact for as long as the lock is held. This can be done in, for example, a contacts editor application when a contact is opened for editing, very much like with the Contacts application in your Nokia device. If you try to modify a contact without locking it for editing, the contact is automatically locked before the modification and released immediately afterwards.

5.1.1 Module Level Functions

The following free functions - functions that do not belong to any class - are defined in the `Contact` module:

open(*[filename[, mode]]*)

Opens a contacts database and returns a `ContactDb` object. *filename* should be a full Unicode path name. If *filename* is not given, opens the default contacts database. If *mode* is not given, the database must exist. If *mode* is '`c`', the database is created if it does not already exist. If *mode* is '`n`', a new, empty database is created, overwriting the possible previous database.

> **Warning:** Using `open` together with the additional parameters *filename* or *mode* is intended for testing purposes only. Due to S60 SDK functionality, the `open` method can sometimes be unreliable with these parameters.

5.1.2 ContactDb Object

There is one default contact database, but it is possible to create several databases with the `open` function.

class ContactDb

`ContactDb` objects have the following methods:

add_contact()

Adds a new contact into the database. Returns a `Contact` object that represents the new contact. The returned object is already locked for modification. Note that a newly created contact will contain

some empty default fields. If you do not want to use the default fields for anything, you can ignore them.

find(*searchterm*)
Finds the contacts that contain the given Unicode string as a substring and returns them as a list.

import_vcards(*vcards*)
Imports the vCard(s) in the given string into the database.

export_vcards(*ids*)
Converts the contacts corresponding to the ID's in the given tuple *ids* to vCards and returns them as a string.

keys()
Returns a list of unique IDs of all `Contact` objects in the database.

compact_required()
Verifies whether compacting is recommended. Returns an integer value indicating either a true or false state. Returns `True` if more than 32K of space is unused and if this comprises more than 50 percent of the database file, or if more than 256K is wasted in the database file.

compact()
Compacts the database to its minimum size.

__delitem__(*id*)
Deletes the given contact from the database.

field_types()
Returns a list of dictionary objects that contains information on all supported field types. The list contains dictionary objects, which each describe one field type. The most important keys in the dictionary are `'type'` and `'location'` which together indentify the field type. `'type'` can have string values such as `'email_address'`. `'location'` can have the string values `'none'`, `'home'`, or `'work'`. Another important key is `'storagetype'`, which defines the storage type of the field. `'storagetype'` can have the string values `'text'`, `'datetime'`, `'item_id'`, or `'binary'`. Note that the `Contacts` extension does not support adding, reading, or modifying fields of any other type than 'text' or `'datetime'`. The other content returned by `field_types` is considered to be advanced knowledge and is not documented here.

groups
Returns contact groups of the database. Read-only.

5.1.3 Contact Object

A `Contact` object represents a live view into the state of a single contact in the database. You can access the fields either with a contact's numeric field ID as `contact[fieldid]`, or using the `find` method. Attempting to modify a contact while it has been locked for editing in another application will raise the exception `ContactBusy`.

class Contact
`Contact` objects have the following attributes:

id
The unique ID of this `Contact`. Read-only.

title
The title of this `Contact`. Read-only.

last_modified
The date/time when this `Contact` object was last modified. Read-only.

is_group
Returns 1 if this contact is a contact group. Returns 0 if normal contact entry. Read-only.

`Contact` objects have the following methods:

begin()
Locks the contact for editing. This prevents other applications from modifying the contact for as long as the lock is held. This method will raise the exception `ContactBusy` if the contact has already been locked.

commit()
Releases the lock and commits the changes made into the database.

rollback()
Releases the lock and discards all changes that were made. The contact remains in the state it was before `begin`.

as_vcard()
Returns the contact as a string in vCard format.

add_field(*type* [, *value* [, *label=field_label*][, *location=location_spec*]])
Adds a new field into this `Contact`. This method raises `ContactBusy` if the contact has been locked by some other application. *type* can be one of the supported field types as a string.

The following field types can be added:

- `city`
- `company_name`
- `country`
- `date`
- `dtmf_string`
- `email_address`
- `extended_address`
- `fax_number`
- `first_name`
- `job_title`
- `last_name`
- `mobile_number`
- `note`
- `pager_number`
- `phone_number`
- `po_box`
- `postal_address`
- `postal_code`
- `state`
- `street_address`
- `url`
- `video_number`
- `picture`
- `second_name`
- `voip`
- `sip_id`
- `personal_ringtone`
- `share_view`
- `prefix`
- `suffix`
- `push_to_talk`
- `locationid_indication`

The following field types are recognized but cannot be created at present:

- `first_name_reading`
- `last_name_reading`
- `speed_dial`
- `thumbnail_image`
- `voice_tag`
- `wvid`

All supported field types are passed as strings or Unicode strings, except for `'date'` which is a float that represents Unix time. For more information on Unix time, see Section **??**, Date and Time.

field_label is the name of the field shown to the user. If you do not pass a label, the default label for the field type is used.

location_spec, if given, must be `'home'` or `'work'`. Note that not all combinations of type and location are valid. The settings of the current contacts database in use determine which ones are valid.

find([*type=field_type*][, *location=field_location*])
Finds the fields of this contact that match the given search specifications. If no parameters are given, all fields are returned.

__delitem__(*fieldindex*)
Deletes the given field from this contact. Note that since this will change the indices of all fields that appear after this field in the contact, and since the `ContactField` objects refer to the fields by index, old `ContactField` objects that refer to fields after the deleted field will refer to different fields after this operation.

5.1.4 ContactField Object

A `ContactField` represents a field of a `Contact` at a certain index. A `ContactField` has attributes, some of which can be modified. If the parent `Contact` has not been locked for editing, modifications are committed immediately to the database. If the parent `Contact` has been locked, the changes are committed only when `commit` is called on the `Contact`.

class ContactField
`ContactField` objects have the following attributes:

label
The user-visible label of this field. Read-write.

value
The value of this field. Read-write.

type
The type of this field. Read-only.

location
The location of this field. This can be `'none'`, `'work'`, or `'home'`.

schema
A dictionary that contains some properties of this field. The contents of this dictionary correspond to those returned by the `ContactDb` method `field_types`.

5.1.5 Groups Object

A `Groups` object represents Symbian contact groups as a dictionary like object with limited subset of dictionary functionality. Each group can be accessed using the group's unique id as a key. The Groups object returns a list like `Group` object as the value matching the given key.

The following common methods are supported: `__iter__`, `__getitem__`, `__delitem__` and `__len__`.

class Groups
`Groups` objects have the following attributes:

add_group([*name*])
Creates new contact group and returns corresponding `Group` object. Group name can be given as an optional parameter.

5.1.6 Group Object

A `Group` object represents single Symbian contact group as a list object with limited subset of list functionality. The `Group` object lists Contact entry ids that belong to the group.

The native Symbian group objects are represented as Symbian contact entries in the database. Therefore they can also be accessed as Python `Contact` objects, but this way their group handling properties cannot be used from Python. Use `Groups` and `Group` objects to access group functionalities.

The following common methods are supported: `__iter__`, `__getitem__`, `__delitem__` and `__len__`.

class `Group`

`Group` objects have the following attributes:

`id`

The unique id of the `Group` object. Read-only.

`name`

The name of the `Group` object. Read-write.

5.2 e32calendar — Access to calendar related services

The calendar module of the previous PyS60 releases has been renamed as e32calendar. For information on the usage of this module refer to Ensymble README.

The e32calendar module offers an API to calendar services. The e32calendar module represents a Symbian agenda database as a dictionary-like CalendarDb object, which contains Entry objects and which is indexed using the unique IDs of those objects. There are five types of entry objects: AppointmentEntry, EventEntry, AnniversaryEntry, ReminderEntry, and TodoEntry.

CalendarDb objects represent a live view into the database. If an entry is changed outside your Python application, the changes are visible immediately, and conversely any changes you commit into the database are visible immediately to other applications.

All time parameters use Unix time unless stated otherwise. For more information on Unix time, see Section **??**, Date and Time.

5.2.1 Module Level Functions

The following free functions - functions that do not belong to any class - are defined in the calendar module:

open([*filename=None, mode=None*])
Opens a calendar database and returns a new CalendarDb object.

If filename is None, the default database is opened.

If *filename* is given, it should contain drive letter, colon and file's name, but no absolute path.

mode can be:

- None: Opens an existing calendar database.
- 'c': Opens an existing calendar database, or creates it if it doesn't exist.
- 'n': Creates a new, empty calendar database. If *filename* exists, the previous contents are erased.

5.2.2 CalendarDb Objects

Calendar entries are stored in a calendar database. There is one default calendar database but more calendar databases can be created by invoking open with parameters 'n' or 'c'.

class CalendarDb
CalendarDb objects have the following methods:

add_appointment()
Creates and returns a new appointment entry AppointmentEntry. The entry is not added and saved into the database until Entry.commit is called.

add_event()
Creates and returns a new event entry EventEntry. The entry is not added and saved into the database until Entry.commit is called.

add_anniversary()
Creates and returns a new anniversary entry AnniversaryEntry. The entry is not added and saved into the database until Entry.commit is called.

add_todo()
Creates and returns new todo entry TodoEntry. The entry is not added and saved into the database until Entry.commit is called.

add_reminder()
Creates and returns new reminder entry ReminderEntry. The entry is not added and saved into the database until Entry.commit is called.

find_instances (*start_date, end_date, search_str=u"*[*,appointments=0,events=0,anniversaries=0,todos=0,reminders=0*]*)*

The parameters for this function include the start date, end date, search string, and optional parameters. The optional parameters define the entry types to be included into the search. By default all entry types are included. Returns a list that contains `Entry` instances found in the search. An instance is a dictionary that contains the entry ID and the datetime value. An entry may have several instances if it is repeated, for example once every week, etc.

In some Nokia models the `search_str` needs to be less or equal to 32 characters, otherwise an error `KErrArgument` or a premature application exit (i.e. a panic) might occur.

monthly_instances (*month, appointments=0, events=0, anniversaries=0, todos=0, reminders=0*)

The parameters for this function include *month* (float) and optional parameters. The optional parameters define the entry types to be returned. Returns a list that contains entry instances occurring during the specified calendar month.

daily_instances (*day, appointments=0, events=0, anniversaries=0, todos=0*)

The parameters for this function include *day* (float) and optional parameters. The optional parameters define the entry types to be returned. Returns a list that contains entry instances occurring on the specified day.

export_vcalendars (*(int,...)*)

Returns a `vcalendar` string that contains the specified entries in vCalendar format. The parameter for this function is a tuple that contains the entry IDs of the exported entries.

import_vcalendars (*string*)

Imports `vcalendar` entries, given in the string parameter, to the database. Returns a list that contains the unique IDs of the imported entries.

__delitem__ (*id*)

Deletes the given calendar `Entry` from the database. `id` is the unique ID of the calendar `Entry`.

__getitem__ (*id*)

Returns a calendar `Entry` object indicated by the unique ID. The returned object can be one of the following: `AppointmentEntry`, `EventEntry`, `AnniversaryEntry`, `ReminderEntry`, or `TodoEntry`. `id` is the unique ID of the calendar `Entry`.

5.2.3 Entry Objects

An `Entry` object represents a live view into the state of a single entry in the database. You can access the entries with an entry's unique ID. If you create a new entry using `db.add_appointment` etc., it is saved into the database only if you call the entry's `commit` method. In case an entry is already saved into the database, the autocommit mode is on by default and all the changes are automatically saved into the database, unless you call the entry's `begin` method. If you call the entry's `begin` method, the changes are not saved into the database until you call the entry's `commit` method.

Database entries cannot be locked. In other words, other applications are able to make changes to the database entries you are using (not directly to the `EntryObjects` you are using, but to their representation in the database) at the same time you are modifying them, even if you use `begin` and `commit` methods.

class Entry

`Entry` objects have the following methods and properties:

content

Sets or returns the entry's content text (Unicode).

commit ()

Saves the entry or in case of a new entry adds the entry into the database. Note that this can be called only in case of a new entry, created with `db.add_appointment` etc., or after `begin` is called.

rollback ()

Undoes the changes made after last `commit`.

set_repeat (*dictionary*)

Sets the repeat data of the entry. *dictionary* is a repeat data dictionary that contains all the repeat rules. For more information on repeat rules, see Section 5.2.4, Repeat Rules.

get_repeat()
Returns the repeat data dictionary of the entry.

location
Sets or returns the entry's location data (Unicode), for example meeting room information.

set_time(*start*[, *end*])
Sets the start and end datetime values of the entry (floats). If only one parameter is given, the other will have the same value.

In case of events, anniversaries, and todo entries the datetime values are truncated to corresponding date values.

`TodoEntries` can be made undated with `TodoEntry.set_time(None)`. Making the todo entry undated means removing the start and end date and all the repeat rules.

start_time
The start datetime value (float) of the entry or `None` if the start datetime of the entry is not set.

end_time
The end datetime value (float) of the entry or `None` if the end datetime of the entry is not set.

id
The unique ID of the entry.

last_modified
The datetime value (float) of the entry's last modification in universal time.

originating
An integer value indicating if the entry is an originating entry or a modifying entry.

alarm
The alarm datetime value (float) for the entry. `None` if `alarm` is not set. Alternatively removes the alarm if the value is set to `None`.

Alarms can be set to all `Entry` types. However, only alarms set to Appointments and Anniversaries will actually cause an alarm; this is similar to the Calendar application in your Nokia device, which allows you to set an alarm only for Meetings and Anniversaries. In addition, alarms set to any entries residing in a database other than the default database do not cause actual alarms either.

priority
The priority of the entry, which can be an integer ranging from 0 to 255. Native Phonebook and Calendar applications in Nokia devices use value 1 for high priority, 2 for normal priority, and 3 for low priority.

crossed_out
The crossed out value of an entry. Only valid for todo entries. A value that is interpreted as false means that the entry is not crossed out, whereas a value that is interpreted as true means that the entry is crossed out. Note that `TodoEntries` must also have a cross-out time. If `TodoEntry` is crossed out using this method, the moment of crossing out is set to the cross-out time of the `TodoEntry`. See also Section 5.2.3, TodoEntry, `cross_out_time`.

replication
Sets or returns the entry's replication status, which can be one of the following: `'open'`, `'private'`, or `'restricted'`.

as_vcalendar()
Returns this entry as a vCalendar string.

AppointmentEntry Objects

class AppointmentEntry

`AppointmentEntry` class contains no additional methods compared to the `Entry` class from which it is derived.

EventEntry

class EventEntry

EventEntry class contains no additional methods compared to the Entry class from which it is derived.

AnniversaryEntry

class AnniversaryEntry

AnniversaryEntry class contains no additional methods compared to the Entry class from which it is derived.

ReminderEntry

class ReminderEntry

ReminderEntry class contains no additional methods compared to the Entry class from which it is derived.

TodoEntry

TodoEntryobjects represent todo entry types. They have additional properties compared to the Entry class from which they are derived.

class TodoEntry

TodoEntryobjects have the following additional properties:

cross_out_time

The cross-out date value of the entry. The value can be None meaning that the entry is not crossed out, or the cross-out date (float). The set value must be date (float). Setting a cross-out time also crosses out the entry. See also Section 5.2.3, Entry Object, crossed_out.

5.2.4 Repeat Rules

Repeat rules specify an entry's repeat status, that is, the recurrence of the entry. There are six repeat types:

- daily: repeated daily
- weekly: repeat on the specified days of the week, such as Monday and Wednesday, etc.
- monthly_by_dates: repeat monthly on the specified dates, such as the 15th and 17th day of the month
- monthly_by_days: repeat monthly on the specified days, such as the fourth Wednesday of the month, or the last Monday of the month
- yearly_by_date: repeat yearly on the specified date, such as December 24
- yearly_by_day: repeat yearly on the specified day, such as every third Tuesday of May

There are exceptions to repeat rules. For example, you can specify the datetime value (float) in such a way that the entry is not repeated on a specific day even if the repeat rule would specify otherwise.

You must set the start and end dates (floats) of the repeat. The end date can also be set to None to indicate that the repeating continues forever. You can set interval defining how often the repeat occurs, for example in a daily repeat: 1 means every day, 2 means every second day, etc. You can also set the days specifier which lets you explicitly specify the repeat days; for example in a weekly repeat you can set "days":[0,2] which sets the repeat to occur on Mondays and Wednesdays. If you do not set the days specifier, the repeat days are calculated automatically based on the start date.

You can modify repeat data by calling rep_data = entry.get_repeat(), then making changes to rep_data dictionary, and then calling entry.set_repeat(rep_data).

Repeating can be cancelled by calling entry.set_repeat with a parameter that is interpreted to be false, such as entry.set_repeat(None).

Repeat definition examples:

```
repeat = {"type":"daily", #repeat type
          "exceptions":[exception_day, exception_day+2*24*60*60],
          #no appointment on those days
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1} #interval (1=every day, 2=every second day etc.)

repeat = {"type":"weekly", #repeat type
          "days":[0,1], #which days in a week (Monday, Tuesday)
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every week, 2=every second week etc.)

repeat = {"type":"monthly_by_days", #repeat type
          # appointments on second Tuesday and last Monday of the month
          "days":[{"week":1, "day":1},{"week":4, "day":0}],
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every month, 2=every second month etc.)

repeat = {"type":"monthly_by_dates", #repeat type
          "days":[0,15],
          # appointments on the 1st and 16th day of the month.
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+30*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every month, 2=every second month etc.)

repeat = {"type":"yearly_by_date", #repeat type
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+3*365*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every year, 2=every second year etc.)

repeat = {"type":"yearly_by_day", #repeat type
          # appointments on the second Tuesday of February
          "days":{"day":1, "week":1, "month":1},
          "exceptions":[exception_day], #no appointment on that day
          "start":appt_start_date, #start of the repeat
          "end":appt_start_date+3*365*24*60*60, #end of the repeat
          "interval":1}
          #interval (1=every year, 2=every second year etc.)
```

5.3 `e32db` — Interface to the Symbian native DB

The `e32db` module provides an API for relational database manipulation with a restricted SQL syntax. For details of DBMS support, see the S60 SDK documentation. For examples on using this module, see [**?**].

The `e32db` module defines the following functions:

format_rawtime(*timevalue*)
Formats *timevalue* (Symbian time) according to the current system's date/time formatting rules and returns it as a Unicode string.

format_time(*timevalue*)
Returns *timevalue* as a Unicode string formatted so that it is acceptable as a SQL time. To make a time literal, surround the return value with hash (#) characters.

5.3.1 Dbms Objects

class Dbms()
Creates a Dbms object. Dbms objects support basic operations on a database.

Dbms objects have the following methods:

begin()
Begins a transaction on the database.

close()
Closes the database object. It is safe to try to close a database object even if it is not open.

commit()
Commits the current transaction.

compact()
Compacts the database, reclaiming unused space in the database file.

create(*dbname*)
Creates a database with path *dbname*.

execute(*query*)
Executes an SQL *query*. On success, returns `0` if a DDL (SQL schema update) statement was executed. Returns the number of rows inserted, updated, or deleted, if a DML (SQL data update) statement was executed.

open(*dbname*)
Opens the database in file *dbname*. This should be a full Unicode path name, for example, `u'c:\\foo.db'`.

rollback()
Rolls back the current transaction.

5.3.2 DB_view Objects

class Db_view()
Creates a `Db_view` object. `DB_view` objects generate rowsets from a SQL query. They provide functions to parse and evaluate the rowsets.

Db_view objects have the following methods:

col(*column*)
Returns the value in *column*. The first column of the rowset has the index `1`. If the type of the column is not supported, a `TypeError` is raised. See Table 5.1 for a list of supported data types.

col_count()
Returns the number of columns defined in the rowset.

col_length(*column*)
Gets the length of the value in *column*. Empty columns have a length of zero; non-empty numerical and date/time columns have a length of 1. For text columns, the length is the character count, and for binary columns, the length is the byte count.

col_raw(*column*)
Extracts the value of *column* as raw binary data, and returns it as a Python string. The first column of the

rowset has the index 1. See Table 5.1 for a list of supported data types.

col_rawtime(*column*)
Extracts the value of a date/time column at index *column* as a long integer, which represents the raw Symbian time value. The first column of the rowset has the index 1. See Table 5.1 for a list of the supported data types.

col_type(*column*)
Returns the numeric type of the given column as an integer from a Symbian-specific list of types. This function is used in the implementation of method `col`.

count_line()
Returns the number of rows available in the rowset.

first_line()
Positions the cursor on the first row in the rowset.

get_line()
Gets the current row data for access.

is_col_null(*column*)
Tests whether *column* is empty. Empty columns can be accessed like normal columns. Empty numerical columns return a `0` or an equivalent value, and text and binary columns have a zero length.

next_line()
Moves the cursor to the next row in the rowset.

prepare(*db, query*)
Prepares the view object for evaluating an SQL select statement. *db* is a `Dbms` object and *query* the SQL query to be executed.

5.3.3 Mapping Between SQL and Python Data Types

See Table 5.1 for a summary of mapping between SQL and Python data types. The `col` function can extract any value except `LONG VARBINARY` and return it as the proper Python value. In addition, the `col_raw` function can extract any column type except `LONG VARCHAR` and `LONG VARBINARY` as raw binary data and return it as a Python string.

Inserting, updating, or searching for `BINARY`, `VARBINARY`, or `LONG VARBINARY` values is not supported. `BINARY` and `VARBINARY` values can be read with `col` or `col_raw`.

5.3.4 Date and Time Handling

The functions `col` and format_time use Unix time, seconds since January 1, 1970, 00:00:00 UTC, as the time format. Internally the database uses the native Symbian time representation that provides greater precision and range than the Unix time. The native Symbian time format is a 64-bit value that represents microseconds since January 1st 0 AD 00:00:00 local time, nominal Gregorian. BC dates are represented by negative values. Since converting this format to Unix time and back may cause slight round-off errors, you have to use the functions `col_rawtime` and `format_rawtime` if you need to be able to handle these values with full precision.

The representation of date and time literals in SQL statements depends on the current system date and time format. The only accepted ordering of day, month, and year is the one that the system is currently configured to use. The recommended way to form date/time literals for SQL statements is to use the functions `format_time` or `format_rawtime` that format the given date/time values properly according to the current system's date/time format settings.

5.4 `e32dbm` — DBM implemented using the Symbian native DBMS

The `e32dbm` module provides a DBM API that uses the native Symbian RDBMS as its storage back-end. The module API resembles that of the `gdbm` module. The main differences are:

SQL type	Symbian column type (in the DBMS C++ API)	Python type	Supported
BIT	EDbColBit	int	yes
TINYINT	EDbColInt8	int	yes
UNSIGNED TINYINT	EDbColUint8	int	yes
SMALLINT	EDbColInt16	int	yes
UNSIGNED SMALLINT	EDbColUint16	int	yes
INTEGER	EDbColInt32	int	yes
UNSIGNED INTEGER	EDbColUint32	int	yes
COUNTER	EDbColUint32 (with the TDbCol::EAutoIncrement attribute)	int	yes
BIGINT	EDbColInt64	long	yes
REAL	EDbColReal32	float	yes
FLOAT	EDbColReal64	float	yes
DOUBLE	EDbColReal64	float	yes
DOUBLE PRECISION	EDbColReal64	float	yes
DATE	EDbColDateTime	float (or long, with col_raw-time()	yes
TIME	EDbColDateTime	float (or long, with col_raw-time()	yes
TIMESTAMP	EDbColDateTime	float (or long, with col_raw-time()	yes
CHAR(n)	EDbColText	Unicode	yes
VARCHAR(n)	EDbColText	Unicode	yes
LONG VARCHAR	EDbColLongText	Unicode	yes
BINARY(n)	EDbColBinary	str	read only
VARBINARY(n)	EDbColBinary	str	read only
LONG VARBINARY	EDbColLongBinary	n/a	no

Table 5.1: Mapping between SQL and Python types

- The `firstkey()` - `nextkey()` interface for iterating through keys is not supported. Use the "`for key in db`" idiom or the `keys` or `keysiter` methods instead.
- This module supports a more complete set of dictionary features than `gdbm`
- The values are always stored as Unicode, and thus the values returned are Unicode strings even if they were given to the DBM as normal strings.

5.4.1 Module Level Functions

The `e32dbm` defines the following functions:

open (*dbname*[,*flags, mode*])
Opens or creates the given database file and returns an `e32dbm` object. Note that *dbname* should be a full path name, for example, u'c:\\foo.db'. Flags can be:

- `'r'`: opens an existing database in read-only mode. This is the default value.
- `'w'`: opens an existing database in read-write mode.
- `'c'`: opens a database in read-write mode. Creates a new database if the database does not exist.
- `'n'`: creates a new empty database and opens it in read-write mode.

If the character `'f'` is appended to flags, the database is opened in *fast mode*. In fast mode, updates are written to the database only when one of these methods is called: `sync`, `close`, `reorganize`, or `clear`.

Since the connection object destructor calls `close`, it is not strictly necessary to close the database before exiting to ensure that data is saved, but it is still good practice to call the `close` method when you are done with using the database. Closing the database releases the lock on the file and allows the file to be reopened or deleted without exiting the interpreter.

If you plan to do several updates, it is highly recommended that you open the database in fast mode, since inserts and updates are more efficient when they are bundled together in a larger transaction. This is especially important when you plan to insert large amounts of data, since inserting records to `e32db` is very slow if done one record at a time.

5.4.2 e32dbm Objects

The `e32dbm` objects returned by the `open` function support most of the standard dictionary methods. The supported dictionary methods are:

- `__getitem__`
- `__setitem__`
- `__delitem__`
- `has_key`
- `update`
- `__len__`
- `__iter__`
- `iterkeys`
- `iteritems`
- `itervalues`
- `get`

- setdefault
- pop
- popitem
- clear

These work the same way as the corresponding methods in a normal dictionary.

In addition, e32dbm objects have the following methods:

close()
Closes the database. In fast mode, commits all pending updates to disk. close raises an exception if called on a database that is not open.

reorganize()
Reorganizes the database. Reorganization calls compact on the underlying e32db database file, which reclaims unused space in the file. Reorganizing the database is recommended after several updates.

sync()
In fast mode, commits all pending updates to disk.

5.5 logs — Module to access the phone logs.

The logs offers generic access to the phone's log. Via logs's API it is possible to access, for example, the list of received calls or the list of sms received. At this stage, it is only possible to read logs.

All of the accessor functions return a list of dictionaries containing the log events. The first item on the list is the latest event.

Each dictionary has the following entries:

- number: The (phone) number associated with the log event
- name
- description: A description of the event
- direction: The direction associated with the event (i.e. whether incoming or outgoing)
- status: Event status
- subject
- id: The event's id
- contact
- duration
- duration type
- flags
- link
- time: The time associated with the event as a unix timestamp.
- data

The current log types are currently supported:

- 'call'

- `'sms'`
- `'data'`
- `'fax'`
- `'email'`
- `'scheduler'`

For those functions providing an optional *mode* parameter, the default mode is currently set to `'in'`. *mode* can take one of the following values:

- `'in'`
- `'out'`
- `'fetched'`
- `'missed'`
- `'in_alt'`
- `'out_alt'`

5.5.1 Module Level Functions

The following functions are provided:

raw_log_data()
Returns the phone's log events of all supported types. For the list of supported types, see 5.5.

log_data(*type,* [*start_log=0, num_of_logs=_all_logs, mode=_default_mode*])
Returns a list of *num_of_logs* events of a certain *type*, the latest one being at position *start_log* in the event logs. Only logs with the specified *mode* are taken into account.

log_data_by_time(*type, start_time, end_time,* [*mode=_default_mode*])
Returns the list of log events of type *type* that have occurred in the time interval between *start_time* and *end_time*. Only logs with the specified *mode* are taken into account.

The variables *start_time*and *end_time* are passed as a unix timestamp.

calls([*start_log=0, num_of_logs=_all_logs, mode=_default_mode*])
Returns a list of *num_of_logs* events of type `'call'`, the latest one being at position *start_log* in the event logs. Only logs with the specified *mode* are taken into account.

faxes([*start_log=0, num_of_logs=_all_logs, mode=_default_mode*])
Returns a list of *num_of_logs* events of type `'fax'`, the latest one being at position *start_log* in the event logs. Only logs with the specified *mode* are taken into account.

emails([*start_log=0, num_of_logs=_all_logs, mode=_default_mode*])
Returns a list of *num_of_logs* events of type `'email'`, the latest one being at position *start_log* in the event logs. Only logs with the specified *mode* are taken into account.

sms([*start_log=0, num_of_logs=_all_logs, mode=_default_mode*])
Returns a list of *num_of_logs* events of type `'sms'`, the latest one being at position *start_log* in the event logs. Only logs with the specified *mode* are taken into account.

scheduler_logs([*start_log=0, num_of_logs=_all_logs, mode=_default_mode*])
Returns a list of *num_of_logs* events of type `'scheduler'`, the latest one being at position *start_log* in the event logs. Only logs with the specified *mode* are taken into account.

data_logs([*start_log=0, num_of_logs=_all_logs, mode=_default_mode*])
Returns a list of *num_of_logs* events of type `'data'`, the latest one being at position *start_log* in the event logs. Only logs with the specified *mode* are taken into account.

5.6 Acronyms and Abbreviations

Acronym and meaning	Description
MMS- Multimedia Messaging Service	Multimedia Messaging Service (MMS) is a new standard in mobile messaging. The diff
Runtimes	Execution environments for applications.
S60	A software platform for mobile phones using Symbian Operating System.
SA- System Attribute	System Attribute.
SAPI- Service API	A set of language-independent APIs integrated into S60 runtimes (WRT and Flash). The
SDK- Software Development Kit	It is a set of programs used by a computer programmer to write application programs.
SMS- Short Messaging Service	SMS is a service for sending short messages of up to 160 characters (224 characters if us
URI- Uniform Resource Indicator	Uniform Resource Indicator is a short string of characters that represent the address or lo
URL- Universal Resource Locator	A specially formatted sequence of characters representing a location on the internet.
WGS-84- World Geodetic System	The World Geodetic System defines a fixed global reference frame for the Earth, for use

Table 5.2: Acronyms and Abbreviations

CHAPTER

SIX

Scriptext - Platform Service API Usage from Python runtime

`scriptext --- Platform Service API usage from Python runtime`

Platform Service API Overview

This section describes the basic principles for using Platform Service APIs from Python. The S60 Service APIs were introduced in the S60 5th Edition and back ported to S60 3rd edition FP2 platform. Platform Service APIs are a set of language-independent APIs integrated into S60 runtimes including Python, Flash and Web Runtime. Future S60 versions may support additional runtimes. S60 added Python binding for integrating Platform Service APIs into Python Runtime.

Note: The service is available from 'S60 third edition FP2 and later'.

The following steps describe how to use Service APIs from Python Runtime:

- Create a service handle for a particular service.
- Define the input parameters.
- If an asynchronous service needs to be requested, define a callback function to process the results.
- Make a request for the required operation.
- Process the output parameters.

Service Objects and Interfaces
For using Platform Service API, a service handle needs to be created. Each Service API has a service provider name and it can support one or more interfaces. A service handle can be instantiated by specify the service provider and interface name. Each service provider supports one or more interfaces.

Note: Interfaces define a set of common methods for service objects.

For example, the location Service API supports ILocation interface. The ILocation interface defines a `GetLocation` method to retrieve the current location of the device.

6.1 Overview of scriptext usage

The following section describes the usage of scriptext module in detail.

6.1.1 Module level functions and Data Types

Load

This is the only module level function. It can be used to load a particular service provider. It returns a handler object that can be used to invoke further services supported by the provider.

scriptextHandle

This is the type of the object which is returned by `Load()` API. Services can be invoked using the `Call()` API of this object.

The provider returns data wrapped in the objects as mentioned in the following:

Note: Examples of using these objects are listed in the following sections.

scriptextmap

Python wrapper over Platform Service API map object. The following operations are supported over this object:

- Finding length of the map
- Getting value with a key
- Finding if a key exists in the map
- Iterating over it

scriptextlist

Python wrapper over Platform Service API List object. The following operations are supported over this object:

- Finding its length
- Getting value at a given index
- Iterating over it

scriptextiterable

Python wrapper over Platform Service API Iterable object. The following operation is supported over this object:

- Iterating over it

6.1.2 Instantiating a Service Object

A Service object can be instantiated using the `load` API, with the required service provider details. Import scriptext module before using the method.

Syntax

```
import scriptext
scriptext_handle = scriptext.load(<provider>, <interface>)
```

Arguments

The first argument is a string that specifies the name of the service provider.

The second argument is a string, that specifies one of the supported interfaces of the ¡provider¿.

Return Value

The load() method returns a service object for a successful call. This method raises `ScriptextError` when the service provider name or the interface name is null.

Example

The following sample code illustrates how to instantiate messaging service object:

```
import scriptext

try:
    messaging_handle = scriptext.load("Service.Messaging", "IMessaging")

except err:
    # Handle error while instantiating the service.
```

6.1.3 Making Synchronous Request

This method is used to request a specific synchronous service or operation from a service provider, using `call()` API.

Syntax

```
result = service_instance_object.call(operation, parameters)
```

Arguments

The operation argument describes the service requested from the service provider.

The parameters argument is a dictionary, which specifies input parameters to the specified request.

Return Value

The Return value contains the service output, and its data type depends on the service requested.

Example

The following sample code illustrates how to retrieve all the Sender `IDs` from Inbox using `GetList`:

```
import scriptext

messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')

# This 'GetList' request returns all the SMS in the inbox as an iterable map

sms_iter = messaging_handle.call('GetList', {'Type': u'Inbox'})

sender_list = []

for sms_dict in sms_iter:
 if sms_dict['MessageType'] == 'SMS':
  sender_list.append(sms_dict['Sender'])

print "ID list :", sender_list
```

6.1.4 Making Asynchronous Request

For making an asynchronous request, a call back function needs to be defined and passed as an additional parameter to the `call()` API.

Syntax

```
result = service_instance_object.call(operation, parameters, callback=callback_function)
```

Arguments

The operation argument describes the service requested from the service provider. The parameters argument is a dictionary, which specifies input parameters to the specified request.

`callback_function` is an user defined callback function.

The following sample code illustrates how to define a callback handler function to handle the response from an asynchronous request:

```
def callback_function(transactionID, eventID, outParam)
```

The following table describes the arguments of the call back function:

Argument	Description	Value
transactionID	This is the unique transaction ID associated with the particular asynchronous request.	It is returned as part of the result of the initial asynchronous call.
eventID	Specifies the asynchronous operation status.	For a complete list of EventID, see EventID 6.12.2 section in the **Appendix**.
outParam	This argument is a dictionary that holds the output of an asynchronous call.	Refer to the following table for the dictionary items in outParam.

The outParam argument of callback method is a map containing the return value, an error code, and an error message.

Properties	Description	Values
ReturnValue	This key contains the information requested by the asynchronous call that initiated the callback. This key is present only if the requested service has a value to return. In this case, outParam contains only ErrorCode and ErrorMessage.	Depends on the Platform Service API and the asynchronous method that was called. Not all calls return this property.
ErrorCode	Specifies a pre-defined error code	For detail information about Platform Service API error codes and their descriptions, see Service API Error Codes and Description 6.12.1 section in the **Appendix**.
ErrorMessage	Describes the error	Depends on the Platform Service API and the asynchronous method that is called.

Example

The following sample code illustrates how to retrieve media files from a database, using the operation:

```
import scriptext
import e32

def media_callback(trans_id, event_id, output_params):
    # Check if we are interested in this transaction
    if trans_id == media_trans_id:
        print "Not the transaction in which we are interested!"
        return

    # Check if the transaction is complete
    if event_id != scriptext.EventCompleted:
        print "Transaction not complete!"
        return

    # Check if the transaction has resulted in any error
    if output_params['ReturnCode'] != 0:
        print output_params['ReturnMessage']
    else:
        song_list = []
        for item in output_params['ReturnValue']:
            song_list.append(item['FileName'])
        print "List of files retrieved:", song_list

    lock.signal()


lock = e32.Ao_lock()
media_handle = scriptext.load('Service.MediaManagement', 'IDataSource')

# Request for the list of mp3s in ascending order

media_trans_id = media_handle.call('GetList',
                                   {'Type': u'FileInfo': u'FileExtension',
                                              'StartRange': u'.mp3'},
                                    'Sort': {'Key': u'FileName', 'Order': u'Ascending'}},
                                   callback=media_callback)

lock.wait()
```

6.1.5 Cancelling of Asynchronous Service Request

To cancel an asynchronous request, `Cancel` is passed as the operation argument in the `call()` API. The `transactionID` associated with the asynchronous operation also needs to be passed. After completing the cancel operation, the callback function is called with `Event_id` as `scriptext.EvenCanceled`.

Syntax

```
serviceInstance.call('Cancel', {'TransactionID': serviceTransactionID})
```

where, `transactionID` is associated with the asynchronous operation, which needs to be cancelled.

Example

The following sample code illustrates how to send and cancel an SMS in asynchronous mode:

```
import scriptext
messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')

def sms_send_callback(trans_id, event_id, output_params):
    if sms_trans_id == trans_id:
        if event_id == scriptext.EventCanceled:
            print "SMS Send Canceled"
        else:
            print "Event_id was not scriptext.EventCanceled"
    else:
        print "Invalid transaction ID received"

sms_trans_id = messaging_handle.call('Send', {'MessageType': u'SMS',
                                              'To': u'12345678', 'BodyText': u'Hi'},
                                     callback=sms_send_callback)
try:
    messaging_handle.call('Cancel', {'TransactionID': sms_trans_id})
except scriptext.ScriptextError, err:
    print "Error cancelling request ", err
```

6.2 Application Manager

The Application Manager service enables Python applications to perform the following tasks:

- Retrieve information about the applications and user installed packages from the phone.
- Request for a particular operation by passing the input parameters. If you make an asynchronous service, define a callback function to process the results.

The following sample code is used to load the provider:

```
import scriptext
appmanager_handle = scriptext.load('Service.AppManager', 'IAppManager')
```

The following table summarizes the Application Manager Interface:

Service provider	`Service.AppManager`
Supported interfaces	`IAppManager`

The following table lists the services available in Application Manager:

Services	**Description**
`GetList` 6.2.1	Retrieves the required information of user installed packages, all applications, or handler application.
`LaunchApp` 6.2.2	Launches application based on the specified application UID.
`LaunchDoc` 6.2.3	Launches application based on the Document.

6.2.1 GetList

`GetList` is used to retrieve information about user installed packages, all applications, and handler applications. It takes a set of input parameters that define Type and Filter to retrieve the required information. It is available only in synchronous mode.

The following is an example for using `GetList`:

```
appmanager_info = appmanager_handle.call('GetList', {'Type': u'Application'})
```

The following table summarizes the specification of `GetList`:

Interface	`IAppManager`
Description	Retrieves information about user installed packages or handler applications based on document path or MIME type.
Response Model	Synchronous
Pre-condition	Valid instance of `IAppManager` interface is instantiated.
Post-condition	Nil

Input Parameters

Input parameters specify the Type of package or application to retrieve, and the Filter for the retrieved information. Input parameter has properties called Type and Filter.

Name	Type	Range	Description
Type	unicode string	`UserInstalledPackage`, `Application`	Performs service based on the following content types. This field is mandatory: For `Application` content type, this API returns all the application present in the device, whether it is user installed or pre-installed. For `UserInstalledPackage` content type, this API returns all user installed packages. This package contains either the application and the supporting DLL, or only the DLLs.
[Filter]	map	**Key**: `DocumentPath` or `MimeType` **Value**: `unicode string`	This Filter Criteria is applicable when the Type is `Application`. It specifies the Document path or MIME type of the application. For example, document path: **C:¿\\data\\abcd.txt** and MIME type: **image/jpeg**. You can use the filter criteria to find out the Handler application. If both `DocumentPath` and `MimeType` are present in the Filter map then, `DocumentPath` gets preference over `MimeType`.

Table 6.1: Input parameters for Getlist

Output Parameters

Output parameters contain the requested information. They also contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

`ReturnValue`	Iterable items	**InstalledPackage**:
		`PackageName`
		`Version`
		`UID`
		`Vendor`
		`Drive`
		Application:
		`UID`
		`Path`
		`Caption`
		`ShortCaption`
			An Installed package contains the metadata field and value (for example: name, version, and UID) of the package in the form of a map. Installed package map contains the UID of package (In S60, it is UID of the **.sisx** file).
			
			In case of `Application`, the API returns the UID, path, and caption of the application. The API returns the appropriate error code if the application does not match the given criteria. For example, if the Mime type given for one SDK or Device is not valid for another, it returns an error code.

Table 6.2: Output parameters for GetList

Errors

The following table lists the error codes and their values:

Error code value	Description
`0`	Success
`1002`	Bad argument type
`1004`	Service not supported
`1012`	Item not found

Table 6.3: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`AppManager:GetList:Type Missing`	Indicates missing of a content type or a mismatch in the datatype of the given content type.
`AppManager:GetList:Filter type mismatch`	Indicates a mismatch in the datatype of the given filter.
`AppManger:GetList:Asynchronous version of API is not supported`	Indicates that the asynchronous version of unsupported `GetList` is called.

Table 6.4: Error messages

Example

The following sample code illustrates how to get the list of applications on S60 device:

```
import scriptext

# Load the desired SAPI
appmanager_handle = scriptext.load('Service.AppManager', 'IAppManager')
try:
   f = open('c:\\data.txt', 'a+')
   app_info = []
   appmanager_info = appmanager_handle.call('GetList', {'Type': u'Application'})
   for item in appmanager_info:
       app_info.append(item['UID'])
       app_info.append(item['Caption'])
       print item['UID']
       print item['Path']
       print item['Caption']
       print item['ShortCaption']
   f.write(str(app_info))
except scriptext.ScriptextError, err:
   print "Error getting the list of Installed Application: ", err
```

6.2.2 LaunchApp

`LaunchApp` is used to launch an application. It takes a set of input parameters that define application ID and the options for launching the application.

The following are the examples for using `LaunchApp`:

Synchronous

```
appmanager_id = appmanager_handle.call('LaunchApp', {'ApplicationID': u's60uid://0x10005a22'
```

Asynchronous

```
appmanager_id = appmanager_handle.call('LaunchApp',
                                       {'ApplicationID': u's60uid://0x10005a22'},
                                       callback=launch_app_callback)
```

where, `launch_app_callback` is a user defined callback function.

The following table summarizes the specification of `LaunchApp`:

Interface	`IAppManager`
Description	Launches the application based on UID.
Response Model	Synchronous and asynchronous
Pre-condition	Valid instance of `IAppManager` interface is instantiated.
Post-condition	Nil

Input Parameters

Input parameter specifies the `ApplicationID` and the mode for launching the application. Input parameter has three properties: application ID, command line argument, and options. Options contain mode, position, and document path.

Name	**Type**	**Range**
`ApplicationID`	string	`s60uid://<UID>`
[CmdLine]	unicode string	Command line argument
[Options]	map	For detail information on Options, refer to the following table 6.6

Table 6.5: Input parameters for LaunchApp

Key	**Value**
[Mode]	`Chained` or *Standalone*
[Position]	`Background` or *foreground*
[DocumentPath]	unicode string

Table 6.6: Options that can be used with LaunchApp, default values are emphasized

In Asynchronous mode the launching application receives the notification when the launched application dies. The notification is not received if this request is cancelled. Cancelling the request does not close the launched application.

Chained mode is applicable for UI based applications only. You will not be able to launch the application in background position in chained mode.

Output Parameters

In asynchronous mode, the `input_params` that is passed to the callback function contains `ErrorCode`, and an `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Name	Type	Range	Description
ErrorCode	int	NA	Contains the SAPI specific error code when the operation fails.
ErrorMessage	string	NA	Error Description in Engineering English.

Table 6.7: Output parameters for LaunchApp

Error code value	Description
0	Success
1002	Bad argument type
1004	Service not supported
1012	Item not found

Table 6.8: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
AppManager:LaunchApp:Application ID Missing	Indicates missing of Application ID or a mismatch in the datatype of the given Application ID.
AppManager:LaunchApp:Command Line type mismatch	Indicates a mismatch in the datatype of Command Line.
AppManger:LaunchApp:OptionMap type mismatch	Indicates a mismatch in the datatype of Options.

Table 6.9: Error messages

Example

The following sample code illustrates how to launch the **Help.exe**, in asynchronous mode:

```
import scriptext
import e32
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete.
def launch_app_callback(trans_id, event_id, input_params):
    if trans_id != appmanager_id and event_id != scriptext.EventCompleted:
        print "Error in servicing the request"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message is: " + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "Application Launched Successfully: "

    lock.signal()

# Load appmanage service
appmanager_handle = scriptext.load('Service.AppManager', 'IAppManager')

# Make a request to query the required information in asynchronous mode
appmanager_id = appmanager_handle.call('LaunchApp', {'ApplicationID': u's60uid://0x10005a22'

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.2.3 LaunchDoc

`LaunchDoc` is used to launch a document in standalone mode or embedded mode. It takes a set of input parameters that specifies the `DocumentPath`, `MimeType`, and options.

The following are the examples for using `LaunchDoc`:

Synchronous

```
appmanager_id = appmanager_handle.call('LaunchDoc',
                                       {'Document': {'DocumentPath': u'c:\\data\\beak.jpg'}}
```

Asynchronous

```
appmanager_id = appmanager_handle.call('LaunchDoc',
                                       {'Document': {'DocumentPath': u'c:\\data\\beak.jpg'}}
                                       callback=launch_doc_callback)
```

where, `launch_doc_callback` is a user defined callback function.

The following table summarizes the specification of `LaunchDoc`:

Interface	`IAppManager`
Description	Launches the application based on a given document.
Response Model	Synchronous and asynchronous
Pre-condition	Valid instance of `IAppManager` interface is instantiated.
Post-condition	Nil

Input Parameters

Input parameter specifies the `DocumentPath`, `MimeType`, and mode options.

Name	Type	Range	Description
Document	map	**Key**: `DocumentPath` or `Handle` **Value**: `string`	Specifies path of the document to launch. If `MimeType` is not given in input then, it is mandatory to give document as input parameter. If Handle and DocumentPath both are present in map then Handle will get preference.
`MimeType`	unicode string	NA	`MimeType` of the application to be Launch. If document is not given in input then it is mandatory to give `MimeType` as input parameter.
[Options]	map	**Key**: `Mode` **Value**: Chained or Standalone	By default the mode is Standalone.

Table 6.10: Input parameters for Launchdoc

`Launchdoc` finds the Handler application internally, in the absence of `MimeType`. It launches the application based on the MIME type and returns the path of the new document, if the `Document` is absent from the input.

In Asynchronous mode the launching application receives a notification when the launched application dies. The notification is not received if this request is cancelled. Cancelling the request does not close the launched application.

Chained mode is applicable for UI based applications only.

Output Parameters

Output parameters contain `ReturnValue`. They also contain `ErrorCode`, and an `ErrorMessage`, if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.
[ReturnValue]	string	`LaunchDoc` returns the document name if it creates a new one. (that is, Return value is optional as only some application creates default document.) If `Document` is not mentioned and only the `MimeType` is mentioned, then application is launched based on the `MimeType` and returns the default document of the application. Creation of the default document depends upon the launched application.	

Table 6.11: Output parameters for LaunchDoc

Errors

The following table lists the error codes and their values:

Error code value	Description
`0`	Success
`1004`	Service not supported
`1012`	Item not found

Table 6.12: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`AppManager:LaunchDoc: Document/MimeType Missing or datatype mismatch`	Indicates missing of `Document` or `MimeType` or a mismatch in the datatype of the given `Document` or `MimeType`.
`AppManager:LaunchDoc: OptionMap type mismatch`	Indicates a mismatch in the datatype of Command Line.
`AppManager:LaunchDoc: OptionMap type mismatch`	Indicates a mismatch in the datatype of Options.

Table 6.13: Error messages

Example

The following sample code illustrates how to launch an application on S60 device, in asynchronous mode:

```
import scriptext
import e32
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def launch_doc_callback(trans_id, event_id, input_params):
    if trans_id != appmanager_id and event_id != scriptext.EventCompleted:
        print "Error in servicing the request"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message is: " + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "Application Launched Successfully: "

    lock.signal()

# Load appmanage service
appmanager_handle = scriptext.load('Service.AppManager', 'IAppManager')

# Make a request to query the required information in asynchronous mode
# Path dependent on the environment on which the application is run
appmanager_id = appmanager_handle.call('LaunchDoc', {'Document': {'DocumentPath': u'c:\\data
print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.3 Calendar

The Calendar service enables Python applications to access, create, and manage calendars and their entries stored on a device.

The following sample code is used to load the provider:

```
import scriptext
calendar_handle = scriptext.load('Service.Calendar', 'IDataSource')
```

The following table summarizes the Calendar Interface:

Service provider	`Service.Calendar`
Supported interfaces	`IDataSource`

The following table lists the services available in Calendar:

Services	**Description**
`GetList` 6.3.1	Retrieves a list of available calendars or a list of calendar entries.
`Add` 6.3.2	Adds a new calendar in the device or a new entry in the specified calendar file.
`Delete` 6.3.3	Deletes a specific calendar from the device or, one or more entries / instances from a specific calendar file.
`Import` 6.3.4	Imports calendar entries from an input file.
`Export` 6.3.5	Exports calendar entries to an output file.
`RequestNotification` 6.3.6	Notifies when add, delete, or modify operation is performed on the entries in the calendar store.

6.3.1 GetList

`GetList` is used to retrieve the information about available calendar databases or calendar entries. It takes a set of input parameters that define the type of information to return, and how to filter the returned list. It is available only in synchronous.

The following is an example for using `GetList`:

```
meeting_list = calendar_handle.call('GetList', {'Type': u'CalendarEntry', 'Filter': {'Calend
```

The following table summarizes the specification of `GetList`:

Interface	`IDataSource`
Description	Returns a list of available calendars or calendar entries.
Response Model	Synchronous
Pre-condition	`IDataSource` interface is loaded.
Post-condition	Nil
Note	The calendar file must be present to get a list of entries from a specific calendar.

Input Parameters for Calendar

Input parameter specifies the Type and Filter to perform `GetList` service.

Name	Type	Range	Description
Type	unicode string	`Calendar`	Indicates that the `GetList` service is to be performed on a calendar.
[Filter]	map	`DefaultCalendar`: bool	This is an optional parameter. If `DefaultCalendar` is set to **True**, `GetList` returns the list with one element (default calendar) else, it returns a list of all calendars.

Table 6.14: Input parameters for Calendar Getlist

Output Parameters for Calendar

Output parameters contain the requested information. It also contains `ErrorCode`, and an `ErrorMessage` if the operation fails. `ReturnValue` contains an array of all Calendars.

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue`	`ScriptextIterableWrapper`	This is an Iterable list of available calendars in the format : **Drivexxx:FileNamexxx**.	

Table 6.15: Output parameters for Calendar Getlist

Input Parameters for Calendar Entry

Input parameter specifies the Type and Filter to perform `GetList` service.

Output Parameters for Calendar Entry

Output parameter contains `ReturnValue`. It also contain `ErrorCode`, and an `ErrorMessage` if the operation fails.

`ReturnValue` of Calendar Entry is an iterable list of entries, which contains all relevant fields of the calendar entry based on the Entry Type (`Meeting`, `To-Do`, `Reminder`, `DayEvent`, `Anniversary`).

Errors

The following table lists the errors and their values:

Error Message

The following table lists the error messages and their description:

Example

The following sample code illustrates how to display all calendar entries:

Name	Type	Range	Description
Type	unicode string	`CalendarEntry`	Indicates that the `GetList` service is performed on calendar entries.

[Filter]	map	**[CalendarName]**: unicode string
		[id]: unicode string
		[LocalId]: unicode string
		[StartRange]: datetime
		[EndRange]: datetime
		[SearchText]: unicode string
		[Type]: unicode string
		where, [Type] is one of the following:
		`Meeting`
		`ToDo`
		`Anniversary`
		`Reminder`
		`DayEvent`
		`IncludeAll`
			All instances are fetched if Filter is not present. `CalendarName` specifies the calendar used in the format **Drivexxx:Filenamexxx**. If this parameter is not specified then, the default calendar is used. `GetList` returns the entries matching with `id` or `LocalId`, if only `id` or `LocalId` is specified. In case of `id`, the first entry is the parent entry. This case ignores the other fields in the input map.
			
			If any of the Ids (`id` and `LocalId`) are not specified then, `GetList` interprets the input as follows:
			
			Returns the instances falling within StartRange and EndRange, if they are specified.
			Returns all instances present in Calendar, if StartRange and EndRange are not specified.
			Returns all instances present on or after the specified date if only StartRange is specified.
			Returns all instances present on or before the specified date if only EndRange is specified.
			Matches the string with the summary field of the entry if Search Text is specified. The match is not case sensitive.
			Includes only entries of the Type specified in the output if Type parameter is present else, includes all entry types.

Table 6.16: Input parameters for Calendar Entry Getlist

Name	Type	Range	Description
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.
ReturnValue	ScriptextIterableWrapper	For specific information on Types, refer to the following tables: Meeting: 6.18 To-Do: 6.19 Anniversary: 6.20 DayEvent: 6.21 Reminder: 6.22	ReturnValue of Calendar Entry is an iterable list of entries, which contains all relevant fields of the calendar entry based on the Entry Type (Meeting, To-Do, Reminder, DayEvent, Anniversary). The output is an Iterable list of instances if id and LocalId are not specified in filter. For more information on keys, refer to the section Key Values 6.3.7.

Table 6.17: Output parameters for Calendar Entry Getlist

Type	string	Meeting
id	string	NA
LocalId	string	NA
Summary	string	NA
SeqNum	32-bit int	NA
StartTime	datetime	NA
EndTime	datetime	NA
InstanceStartTime	datetime (Valid only for instance list)	NA
InstanceEndTime	datetime (Valid only for instance list)	NA
Replication	string	Open Private Restricted
Description	string	NA
Priority	32-bit int	NA
AlarmTime	datetime	NA
Location	string	NA
Status	string	Tentative Confirmed Cancelled NullStatus
RepeatDates	List of dates	NA
ExDates	List of dates	NA
Method	string	NA
PhoneOwner	string	NA
Organizer	CommonName: string Address: string	NA
Attendees	List of maps	NA
RepeatRule	map	NA

Table 6.18: Entry type: Meeting

Type	string	ToDo
id	string	NA
LocalId	string	NA
Summary	string	NA
EndTime	datetime	NA
Replication	string	`Open` `Private` `Restricted`
Description	string	NA
Priority	32-bit int	NA
AlarmTime	datetime	NA
Status	string	`TodoNeedsAction` `TodoCompleted` `TodoInProcess` `Cancelled` `NullStatus`

Table 6.19: Entry type: ToDo

Type	string	Anniversary
id	string	NA
LocalId	string	NA
Summary	string	NA
StartTime	datetime	NA
Replication	string	`Open` `Private` `Restricted`
Description	string	NA
Priority	32-bit int	NA
AlarmTime	datetime	NA

Table 6.20: Entry type: Anniversary

Type	string	DayEvent
id	string	NA
LocalId	string	NA
Summary	string	NA
StartTime	datetime	NA
EndTime	datetime	NA
Replication	string	`Open` `Private` `Restricted`
Description	string	NA
Priority	32-bit int	NA
AlarmTime	datetime	NA

Table 6.21: Entry type: DayEvent

Type	string	Reminder
id	string	NA
LocalId	string	NA
Summary	string	NA
StartTime	datetime	NA
Replication	string	`Open` `Private` `Restricted`
Description	string	NA
Priority	32-bit int	NA
AlarmTime	datetime	NA

Table 6.22: Entry type: Reminder

Error code value	Description
`1000`	Invalid service argument
`1012`	Item not found

Table 6.23: Error codes

Error messages	Description
Calendar:GetList:Type is invalid	Type is missing or invalid Type is passed
Calendar:GetList:Filter is invalid	Type of Filter parameter is invalid
Calendar:GetList:DefaultCalendar is invalid	Type passed for DefaultCalendar is invalid
Calendar:GetList:Id is invalid	Type passed for Id is invalid
Calendar:GetList:LocalId is invalid	Type passed for LocalId is invalid
Calendar:GetList:StartRange is invalid	Type passed for StartRange is invalid
Calendar:GetList:EndRange is invalid	Type passed for EndRange is invalid
Calendar:GetList:SearchText is invalid	Type passed for SearchText is invalid
Calendar:GetList:CalendarName is invalid	Type passed for CalendarName is invalid

Table 6.24: Error messages

```
import scriptext

# Load Calendar service
calendar_handle = scriptext.load('Service.Calendar', 'IDataSource')
meeting_list = calendar_handle.call('GetList', {'Type': u'CalendarEntry', 'Filter':  {'Calen
for meeting in meeting_list:
print 'Id = ' + meeting['id']
        print 'Description = ' + meeting['Description']

        value = meeting['StartTime']
        print "Meeting starting time is ", value.day, value.month, value.year, value.hour, "

        value = meeting['EndTime']
        print "Meeting End time is ", value.day, value.month, value.year, value.hour, ":", v
```

6.3.2 Add

Add is used to create a new calendar on the device, add an entry to a calendar, or modify the entry if an entry with the same LocalId already exists in the calendar. The entry is added to the specified calendar or, if no calendar is specified, to the default one. In case the default calendar does not exist, it is created. It is available only in synchronous mode.

The following is an example for using Add:

```
calendar_handle.call('Add', {'Type': u'CalendarEntry', 'Item': {'Description': u'This is the
where start_time and end_time are datetime objects.
```

The following table summarizes the specification of Add:

Interface	IDataSource
Description	Adds a new calendar in the device or a new entry in a specific calendar file.
Response Model	Synchronous
Pre-condition	IDataSource interface is loaded. For updating a specific entry, the Id must exist and can be retrieved by a call to Add or GetList.
Post-condition	Nil

Input Parameters for Calendar

Input parameter specifies the details of a new calendar. Input parameter has two properties: Type, and Item.

Name	Type	Range	Description
Type	unicode string	`Calendar`	Adds a new calendar
Item	map	`CalendarName`: unicode string	Specifies the name of the calendar to be added in the format **Drivexxx: FileNamexxx**.

Table 6.25: Input parameters for Calendar Add

Output Parameters for Calendar

Output parameter contains an error code and an optional error message if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.26: Output parameters for Calendar Add

Input Parameters for Calendar Entry

`Add` performs add or update operations depending on the input parameters of Calendar Entry. Input parameters differ based on the Entry Type (`Meeting`, `To-Do`, `Reminder`, `DayEvent`, `Anniversary`).

Name	Type	Range	Description
Type	unicode string	`CalendarEntry`	Adds a new entry or modifies an existing entry depending on the input of Calendar Entry.
Item	map	**[CalendarName]**: unicode string For specific information on Type, refer to the following tables: `Meeting`: 6.28 `To-Do`: 6.29 `Anniversary`: 6.30 `DayEvent`: 6.31 `Reminder`: 6.32	For different entry types, the corresponding input maps are given. For more information about keys, refer the section Key Values 6.3.7. The keys mentioned in the tables for each 'Type' are only applicable for that Type, rest are ignored. The attendee field has a value of type 'map'. The `phoneowner` must match the 'Address' field of one of the attendees, which means that a `phoneowner` is an attendee.

Table 6.27: Input parameters for Calendar Entry Add

Output Parameters for Calendar Entry

Output parameter contains the requested information, an `ErrorCode`, and an `ErorrMessage` if the operation fails.

Input Parameters for Update

Input parameter specifies the type on which an operation is performed and the details of the particular Type.

Output Parameters for Update

Type	unicode string	Meeting
[Summary]	unicode string	NA
[SeqNum]	32-bit int	NA
StartTime	datetime	NA
EndTime	datetime	NA
[Replication]	unicode string	NA
[Description]	unicode string	NA
[Priority]	32-bit int	NA
[AlarmTime]	datetime	NA
[Location]	unicode string	NA
[Status]	unicode string	`Tentative` `Confirmed` `Cancelled` `NullStatus`
[RepeatDates]	List of dates	NA
[ExDates]	List of dates	NA

[Method]	`None`
	`Publish`
	`Request`
	`Reply`
	`Add`
	`Cancel`
	`Refresh`
	`Counter`
	`DeclineCounter`
		NA

[PhoneOwner]	unicode string	NA
[Organizer]	[CommonName]: string Address: string	NA
[Attendees]	List of maps	NA
[RepeatRule]	map	NA

Table 6.28: Entry type: Meeting

Type	unicode string	ToDo
[Summary]	unicode string	NA
[EndTime]	datetime	NA

[Replication]	unicode string	`Open`
		`Private`
		`Restricted`

[Description]	unicode string	NA
[Priority]	32-bit int	NA
[AlarmTime]	datetime	NA

[Status]	unicode string	`TodoNeedsAction`
		`TodoCompleted`
		`TodoInProcess`
		`Cancelled`
		`NullStatus`

Table 6.29: Entry type: ToDo

Type	unicode string	Anniversary
[Summary]	unicode string	NA
StartTime	datetime	NA
[Replication]	unicode string	`Open` `Private` `Restricted`
[Description]	unicode string	NA
[Priority]	32-bit int	NA
[AlarmTime]	datetime	NA

Table 6.30: Entry type: Anniversary

Type	unicode string	DayEvent
[Summary]	unicode string	NA
StartTime	datetime	NA
[EndTime]	datetime (If not specified, the value will be same as StartTime)	NA
[Replication]	unicode string	`Open` `Private` `Restricted`
[Description]	unicode string	NA
[Priority]	32-bit int	NA
[AlarmTime]	datetime	NA

Table 6.31: Entry type: DayEvent

Type	string	Reminder
[Summary]	unicode string	NA
StartTime	datetime	NA
[Replication]	unicode string	`Open` `Private` `Restricted`
[Description]	unicode string	NA
[Priority]	32-bit int	NA
[AlarmTime]	datetime	NA

Table 6.32: Entry type: Reminder

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue`	string	NA	Returns `Id` string of the new entry added. The `LocalId` is obtained by a call to `GetList`, with `Id` as filter parameter.

Table 6.33: Output parameters for Calendar Entry Add

Name	Type	Range	Description
Type	unicode string	`CalendarEntry`	Adds a new entry or modifies an existing entry depending on the input of Calendar Entry.
Item	map	**[CalendarName]**: unicode string **[LocalId]**: unicode string **[InstanceStartTime]**: int	`CalendarName` must be specified in **Drivexxx:Filenamexxx**. If **CalendarName** is not specified then update operation is performed on default calendar. Identifies the Entry by the `LocalId`. In case of repeating entry, `InstanceStartTime` (applicable only for `Meeting` type) is used to identify the instance to be modified. If `InstanceStartTime` is not specified then it modifies the whole entry. The modifiable fields(except type) is taken from `CalendarEntry`. `RepeatRule` can be modified or added for Parent entry only. The keys mentioned in the tables for each 'Type' are only applicable for that Type, rest are ignored.

Table 6.34: Input parameters for Update

Output parameter contains the `Id` of the new entry added, `ErrorCode`, and an `ErrorMessage`, if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue`	string	NA	Returns `Id` string of the new entry added. The `LocalId` is obtained by a call to `GetList`, with `Id` as filter parameter.

Table 6.35: Output parameters for Update

Errors

The following table lists the errors and their values:

Error code value	Description
`1000`	Invalid service argument
`1002`	Bad argument type
`1004`	Service not supported
`1010`	Entry exists
`1012`	Item not found

Table 6.36: Error codes

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to add a calendar entry:

```
import scriptext
import datetime

# Load Calendar service
calendar_handle = scriptext.load('Service.Calendar', 'IDataSource')

start_time = datetime.datetime(2009,03,12,17,0,0)
end_time = datetime.datetime(2009,03,12,18,0,0)

try:
    calendar_handle.call('Add', {'Type': u'CalendarEntry', 'Item': {'Type': u'Meeting', 'Des
except scriptext.ScriptextError:
    print 'Error in servicing the request'
else:
    print "Add request successfully complete!"
```

6.3.3 Delete

`Delete` is used to remove a calendar from the device or, one or more entries from a calendar. Entries are deleted from the specified calendar or, from the default one if no calendar is specified. You can delete a calendar in synchronous mode. You can delete calendar entries both in synchronous and asynchronous mode.

Note:

Error messages	Description
Calendar:Add:Entry Type is invalid	Invalid type is passed for Type parameter
Calendar:Add:InstanceStartTime is invalid	Invalid type is passed for InstanceStartTime parameter
Calendar:Add:LocalId is invalid	Invalid type is passed for LocalId parameter
Calendar:Add:Summary is invalid	Invalid type is passed for Summary parameter
Calendar:Add:Description is invalid	Invalid type is passed for Description parameter
Calendar:Add:Location is invalid	Invalid type is passed for Location parameter
Calendar:Add:Replication is invalid	Invalid type is passed for Replication parameter
Calendar:Add:Status is invalid	Invalid type is passed for Status parameter
Calendar:Add:Method is invalid	Invalid type is passed for Method parameter
Calendar:Add:SeqNum is invalid	Invalid type is passed for SeqNum parameter
Calendar:Add:Priority is invalid	Invalid type is passed for Priority parameter
Calendar:Add:StartTime is invalid	Invalid type is passed for StartTime parameter
Calendar:Add:EndTime is invalid	Invalid type is passed for EndTime parameter
Calendar:Add:AlarmTime is invalid	Invalid type is passed for AlarmTime parameter
Calendar:Add:PhoneOwner is invalid	Invalid type is passed for PhoneOwner parameter
Calendar:Add:Organizer is invalid	Invalid type is passed for Organizer parameter
Calendar:Add:Attendees is invalid	Invalid type is passed for Attendees parameter
Calendar:Add:CommonName is invalid	Invalid type is passed for CommonName parameter
Calendar:Add:Address is invalid	Invalid type is passed for Address parameter
Calendar:Add:Role is invalid	Invalid type is passed for Role parameter
Calendar:Add:Status is invalid	Invalid type is passed for Status parameter
Calendar:Add:Rsvp is invalid	Invalid type is passed for Rsvp parameter
Calendar:Add:RepeatDates is invalid	Invalid type is passed for RepeatDates parameter
Calendar:Add:ExDates is invalid	Invalid type is passed for ExDates parameter
Calendar:Add:RepeatRule is invalid	Invalid type is passed for RepeatRule parameter
Calendar:Add:Type is invalid	Invalid type is passed for RepeatRule:Type parameter
Calendar:Add:Type is missing	RepeatRule:Type parameter is missing
Calendar:Add:DaysInWeek is invalid	Invalid type passed for RepeatRule:DaysInWeek or list contains invalid data
Calendar:Add:UntilDate is invalid	Invalid type passed for RepeatRule:UntilDate
Calendar:Add:RepeatRule:StartDate is invalid	Invalid type passed for RepeatRule:StartDate
Calendar:Add:Interval is invalid	Invalid type is passed for RepeatRule:Interval parameter
Calendar:Add:MonthDays is invalid	Invalid type passed for RepeatRule:MonthDays or list contains invalid data
Calendar:Add:DaysInWeek is invalid	Invalid type passed for RepeatRule:DaysInWeek or list contains invalid data
Calendar:Add:DaysOfMonth is invalid	Invalid type passed for RepeatRule:DaysOfMonth or list contains invalid data
Calendar:Add:RepeatRule:DaysOfMonth:Day	Invalid type passed for RepeatRule:DaysOfMonth:Day
Calendar:Add:RepeatRule:DaysOfMonth:WeekNumber	Invalid type passed for RepeatRule:DaysOfMonth:WeekNumber
Calendar:Add:Month	Invalid type passed for RepeatRule:Month
Calendar:Add:Item is invalid	Invalid type is passed for Item
Calendar:Add:CalendarName is invalid	Invalid type is passed for CalendarName
Calendar:Add: is invalid	Invalid type is passed for parameter

Table 6.37: Error messages

- You cannot delete the default calendar.
- To delete a calendar or entries from a calendar, the corresponding calendar file must exist on the device.

The following are the examples for using `Delete`:

Synchronous

```
event_id = calendar_handle.call('Delete', {'Type': u'CalendarEntry', 'id': del_id_list})
```

Asynchronous

```
event_id = calendar_handle.call('Delete', {'Type': u'CalendarEntry', 'id': del_id_list}, cal
```

where `del_callback` is a user defined callback function.

The following table summarizes the specification of `Delete`:

Interface	`IDataSource`
Description	Deletes the specified calendar from the Device or, one or more entries / instances from a specific calendar file.
Response Model	Synchronous for type `Calendar` and both synchronous and asynchronous for type `CalendarEntry`.
Pre-condition	`IDataSource` interface is loaded.
Post-condition	Nil

Input Parameters for Calendar

Input parameter specifies the type on which the operation is performed and the details of the particular type.

Name	Type	Range	Description
Type	unicode string	`Calendar`	Performs the operation on all available calendars if the type is `Calendar`.
Data	map	`CalendarName`: unicode string	Deletes the given calendar. You cannot delete the default calendar.

Table 6.38: Input parameters for Calendar Delete

Output Parameters for Calendar

Output parameter contains `ErrorCode` and an optional `ErrorMessage`, which is displayed when the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.39: Output parameters for Calendar Delete

Input Parameters for Calendar Entry

Input parameter specifies the type on which the operation is performed and the details of the particular type.

Name	Type	Range	Description
Type	unicode string	`CalendarEntry`	Performs the operation on entries of the specified calendar, if the type is `CalendarEntry`.
Data	map	**[CalendarName]**: unicode string **[IdList]** or **[LocalIdList]**: List of unicode string **[StartRange]**: datetime **[EndRange]**: datetime **[DeleteAll]**: bool	Uses the default calendar if the `CalendarName` is not specified. You can specify either `IdList` or `LocalIdList` with StartRange or EndRange or, both. Deletes the instances within the specified range if range is specified. Deletes entries that match the `IdList` or `LocalIdList` if no range is specified. Deletes all entries within the specified calendar if the `DeleteAll` field is set. One of the fields from the set `IdList` or `LocalIdList`, `StartRange`, `EndRange`, and `DeleteAll` must be passed to delete entries. If not, error is returned. Invalid `id` or `LocalIds` from list are ignored.

Table 6.40: Input parameters for Calendar Entry Delete

Output Parameters for Calendar Entry

Output parameter contains `ErrorCode`, and an `ErrorMessage`, which is displayed when the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.41: Output parameters for Calendar Entry Delete

Errors

The following table lists the errors and their values:

Error code value	Description
`1000`	Invalid service argument
`1004`	Service not supported
`1012`	Item not found

Table 6.42: Error codes

Error Messages

The following table lists the errors messages and their description:

Example

Error messages	Description
Calendar:Delete:Type is invalid	Delete called with invalid Type
Calendar:Delete:CalendarName is missing	Delete (type Calendar) called without passing CalendarName
Calendar:Delete:CalendarName is invalid	Invalid type is passed for CalendarName
Calendar:Delete:StartRange is invalid	Invalid type is passed for StartRange parameter
Calendar:Delete:EndRange is invalid	Invalid type is passed for EndRange parameter
Calendar:Delete:DeleteAll is invalid	Invalid type is passed for DeleteAll parameter
Calendar:Delete:IdList is invalid	Invalid type is passed for IdList parameter
Calendar:Delete:LocalIdList is invalid	Invalid type is passed for LocalIdList parameter
Calendar:Delete:Data is missing	Delete (type CalendarEntry) called with invalid delete Data
Calendar:Delete:Data is invalid	Invalid type is passed for Data parameter

Table 6.43: Error messages

The following sample code illustrates how to delete a specified calendar entry in asynchronous mode:

```
import scriptext
import e32

# Using e32.Ao_lock() so that the main function can wait
# till the callback is hit.
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def del_callback(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in the operation"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message is: " + input_params["ReturnValue"]["ErrorMessage"]
    elif event_id == scriptext.EventCompleted:
        print "Entry deleted successfully."

    lock.signal()

# Returns the list of calendar id's that needs to be deleted.
del_id_list = get_cal_del_id()

# Load Calendar service
calendar_handle = scriptext.load('Service.Calendar', 'IDataSource')
event_id = calendar_handle.call('Delete', {'Type': u'CalendarEntry', 'IdList': del_id_list},

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.3.4 Import

`Import` is used to import entries into a calendar. The information must be imported from an **ICal** or **VCal** file.

Note: If entries are imported to a calendar other than the default one, the corresponding calendar file must exist on the device.

The following the examples for using `Import`:

Synchronous

```
calendar_handle.call('Import', {'Type': u'CalendarEntry', 'FileName': u'C:\\data\\input.txt'
```

Asynchronous

```
calendar_handle.call('Import', {'Type': u'CalendarEntry', 'FileName': u'C:\\data\\input.txt'
```

where, `imp_callback` is an user defined callback function.

The following table summarizes the specification of `Import`:

Interface	`IDataSource`
Description	Imports the calendar entries from an input file.
Response Model	Synchronous and asynchronous.
Pre-condition	`IDataSource` interface is loaded.
Post-condition	Nil
Note	The specified calendar must exist.

Input Parameters

Input parameter specifies the Type and its details to import. Input parameter properties are Type and Data.

Name	Type	Range	Description
Type	unicode string	`CalendarEntry`	Performs the operation on calendar entries.
Data	map	**[CalendarName]**: unicode string **Buffer** or **FileName**: unicode string **Format**: unicode string	Imports entries to a specified calendar or to the default calendar if not specified. `CalendarName` must be in the format **Drivexxx:FileNamexxx**. Either Buffer or `FileName` can be given. `FileName` must contain the complete path of the file. For example, **C:¿\\data\\importfile.txt** Buffer or `Filename` holds the entries to be imported. Format specifies the data format of buffer or file. Format can have values **ICal** or **VCal**. **ICal** is supported from Fifth Edition devices onwards.

Table 6.44: Input parameters Import

Output Parameters

Output contains `ReturnValue`. It also contains `ErrorCode`, and an `ErrorMessage`, if the operation fails. **ReturnValue** contains the Ids of the entries imported.

Errors

The following table lists the errors and their values:

Name	Type	Range	Description
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.
ReturnValue	iterator	string	An Iterate list of Ids of the entries successfully imported to the specified calendar file. **Note:** The Id can repeat in case of Modifying entries.

Table 6.45: Output parameters Import

Error code value	Description
1000	Invalid service argument
1004	Service not supported

Table 6.46: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
Calendar:Import:CalendarName is invalid	Invalid type is passed for CalendarName
Calendar:Import:FileName is invalid	Invalid type is passed for FileName or, FileName exceeds 239 characters
Calendar:Import:Buffer is invalid	Invalid type is passed for Buffer
Calendar:Import:Type is invalid	Import called with invalid Type
Calendar:Import:Data is missing	Data parameter is missing
Calendar:Import:Data is invalid	Invalid type is passed for Data parameter
Calendar:Import:Format is missing	Import Format parameter not specified.
Calendar:Import:FileName is missing	FileName is not passed

Table 6.47: Error messages

Example

The following sample code illustrates how to import a calendar entry:

```
# Load Calendar service
calendar_handle = scriptext.load('Service.Calendar', 'IDataSource')

try:
    calendar_handle.call('Import', {'Type': u'CalendarEntry', 'FileName': u'C:\\Data\\import
except scriptext.ScriptextError:
    print 'Error in servicing the request'
else:
    print "Import request successfully complete!"
```

6.3.5 Export

Export is used to export the calendar entries to an output file. The information is exported to an **ICal** or **VCal** file. This method can be called both in synchronous and asynchronous mode.

The following are the examples for using Export:

Synchronous

```
calendar_handle.call('Export', {'Type': u'CalendarEntry', 'FileName': u'C:\\Data\\output.txt
```

Asynchronous

```
calendar_handle.call('Export', {'Type': u'CalendarEntry', 'FileName': u'C:\\data\\output.txt
```

where, `exp_callback` is an user defined callback function.

The following table summarizes the specification of `Export`:

Interface	`IDataSource`
Description	Exports the calendar entries to an output file.
Response Model	Synchronous and asynchronous
Pre-condition	`IDataSource` interface is loaded.
Post-condition	Nil
Note	The specified calendar must exist.

Input Parameters

Input parameter specifies the Type and its details to export. Input parameter properties are Type and Data.

Name	Type	Range	Description
Type	unicode string	`CalendarEntry`	Performs the operation on calendar entries.
Data	map	**[CalendarName]**: unicode string **Idlist** or **LocalIdList**: List of unicode strings **FileName**: unicode string **Format**: unicode string	Exports entries to the default calendar if not specified. `CalendarName` must be in the format **Drivexxx:FileNamexxx**. Exports entries in the given format. Format can have values **ICal** or **VCal**. **ICal** is supported from Fifth Edition devices onwards. `IdList` or `LocalIdList` is a list of Ids of the entries to be exported. Specify either `IdList` or `LocalIdList`. It exports all the entries from the specified calendar file if the list is not specified. Also, it exports only for valid Ids and ignores the remaining Ids. Entries are exported to the file if `FileName` is specified else, 8-bit Data is returned as output.

Table 6.48: Input parameters Export

Output Parameters

Output contains the requested information `ReturnValue`. It also contains `ErrorCode`, and an `ErrorMessage`, if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue` (Applicable in case `FileName` is not specified in data)	8-bit data	NA	Contains the exported entries in the specified format. It is applicable if `FileName` is not specified in input parameters.

Table 6.49: Output parameters Export

Errors

The following table lists the errors and their values:

Error code value	Description
`1000`	Invalid service argument
`1004`	Service not supported

Table 6.50: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`Calendar:Export:Type is invalid`	Export called with invalid Type
`Calendar:Export:Data is missing`	Export called without passing input `Data`
`Calendar:Export:Data is invalid`	Invalid type is passed for input `Data` parameter
`Calendar:Export:Format is missing`	Export Format not passed in `Data`.
`Calendar:Export:FileName is invalid`	Invalid type for `FileName` parameter or, `FileName` exceeds 239 characters
`Calendar:Export:IdList is invalid`	Invalid type for input `IdList` parameter
`Calendar:Export:LocalIdList is invalid`	Invalid type for input `LocalIdList` parameter
`Calendar:Export:CalendarName is invalid`	Invalid type is passed for `CalendarName`

Table 6.51: Error messages

Example

The following sample illustrates how to export a calendar entry:

```
# Load Calendar service
calendar_handle = scriptext.load('Service.Calendar', 'IDataSource')

try:
    calendar_handle.call('Export', {'Type': u'CalendarEntry', 'FileName': u'C:\\Data\\import
except scriptext.ScriptextError:
    print 'Error in servicing the request'
else:
    print "Export request successfully complete!"
```

6.3.6 RequestNotification

`RequestNotification` is used to notify the registered client when events such as entry creation, updation, or deletion occurs in a specified calendar. If no calendar is specified, the default calendar is used. This is an asynchronous method.

The following is an example for using `RequestNotification`:

```
event_id = calendar_handle.call("RequestNotification", {'Type': u'CalendarEntry'}, callback=
```

The following table summarizes the specification of `RequestNotification`:

Interface	`IDataSource`
Description	Notifies when add, delete, or modify is performed on the entries in the calendar store.
Response Model	Asynchronous
Pre-condition	`IDataSource` interface is loaded.
Post-condition	Nil
Note	The specified calendar must exist.

Input Parameters

Input parameter specifies the Type and its details to perform operation.

Name	Type	Range	Description
Type	unicode string	`CalendarEntry`	Performs the operation on calendar entries.
[Filter]	map	**[CalendarName]**: unicode string **[LocalIdList]**: List of unicode strings **[EndRange]**: datetime **[IncludeUndatedTodos]**: bool	If this entry is not specified then, notifies changes to the default calendar. `CalendarName` must be in the format **Drivexxx:FileNamexxx**. `LocalIdList` specifies Ids for notification. These are obtained by a call to `Getlist`. If it is not specified all the entries are considered. The `StartRange` and `EndRange` fields specify the time range during which notifications are required. `IncludeUndatedTodos` specifies whether notifications are required for `ToDo` entries that have no date.

Table 6.52: Input parameters RequestNotification

Output Parameters

Output parameter contains the type of modification performed on the entries in the Calendar store and the `LocalId` of that entry. It also contains `ErrorCode`, and an `ErrorMessage`, if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue`	Iterator (map)	**ChangeType**: string: `Add` `Delete` `Modify` `Unknown` **LocalId**: string	The `ChangeType` field indicates the type of modification made to the entries in the calendar store. The `LocalId` gives the Id of the entry that is modified, added, or deleted.

Table 6.53: Output parameters RequestNotification

Errors

The following table lists the errors and their values:

Error code value	Description
`1000`	Service argument out of range

Table 6.54: Error codes

Error Messages

the following table lists the error messages and their description:

Error messages	Description
`Calendar:RequestNotification:CalendarName is invalid`	Invalid type is passed for `CalendarName`
`Calendar:RequestNotification:Type is invalid`	`RequestNotification` called with invalid Type
`Calendar:RequestNotification:StartRange is invalid`	Invalid type for Filter:StartRange parameter
`Calendar:RequestNotification:EndRange is invalid`	IInvalid type for Filter:EndRange parameter
`Calendar:RequestNotification:IncludeUndatedTodos is invalid`	Invalid type for Filter: IncludeUndatedTodos parameter.
`Calendar:RequestNotification:FileName is invalid`	Invalid type for `FileName` parameter or, `FileName` exceeds 239 characters
`Calendar:RequestNotification:LocalIdList is invalid`	Invalid type for Filter:LocalIdList parameter or, `LocalIdList` contains invalid data
`Calendar:RequestNotification:Filter is invalid`	Invalid type for Filter parameter

Table 6.55: Error messages

Example

```
import scriptext
import e32

lock = e32.Ao_lock()
calendar_handle = scriptext.load('Service.Calendar', 'IDataSource')

def calendar_callback(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message is: " + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "Modification is: " + str(input_params["ReturnValue"]["ChangeType"])
    lock.signal()

# Make a request to get notification
event_id = calendar_handle.call("RequestNotification", {'Type': u'CalendarEntry'}, callback=

lock.wait()
```

6.3.7 Key Values

Calendar Entry

Fields applicable for a particular 'Type' of entry are mentioned in **Add** Section, all other fields are ignored.

Repeat Rule Structure

Most the fields in the following table are applicable for specific 'Type'.

Note:

- Specify only Type and `UntilDate` for a 'Daily' repeat.
- Specify Type, `UntilDate`, and `DaysInWeek` for a 'Weekly' repeat rule.
- Specify Type, `UntilDate`, and `MonthDays` / `DaysOfMonth` for a 'Monthly' repeat rule. The `WeekNum` parameter can take values 1, 2, 3, 4 for the first, second, third, and fourth week of the month, or -1 for the last week of the month.
- Specify Type, `UntilDate`, `DaysOfMonth`, and `Month` for a 'Yearly' repeat rule. Only first entry in `DaysOfMonth` is taken. If specified, `DaysOfMonth` and `Month` must be given together.
- If `DaysInWeek`, `MonthDays`, `DaysOfMonth`, or `Month` (whichever applicable for repeat rule 'Type') is not specified, it is calculated from Entry `StartTime`.
- Interval is an optional parameter for all types.
- `UntilDate` parameter is set to be the same value as specified for Third Edition and Third Edition FP1 onwards, it is modified internally to be the start time of the last instance of the repeat rule.
  If `UntilDate` is not specified, it is taken as the maximum time.

Attendee Structure

Most the fields in the following table are applicable for specific 'Type'.

Key	Description
Type	Specifies whether the entry is a meeting, to-do item, reminder, event or anniversary.
`CalendarName`	Specifies Calendar Name. It must be given in the format **Drivexxx:Filenamexxx**.
`Summary`	Holds the summary for the calendar entry.
`SeqNum`	Holds the sequence number for the calendar entry, used in group scheduling. The default value is 0.
`StartTime`	Holds the start time for the calendar entry.
`EndTime`	Holds the end time for the calendar entry.
Replication	Specifies replication status of the entry **Open**: No restriction on access, this is the default value. **Private**: Data is private, no access. **Restricted**: Data is confidential, restricted access.

Method	The method property of an entry (only for `ICalendar` entry).
	`None`: This is the default value if not specified.
	`Publish`
	`Request`
	`Reply`
	`Add`
	`Cancel`
	`Refresh`
	`Counter`
	`DeclineCounter`

Description	Holds the description for the calendar entry.
Priority	Specifies the priority for the calendar entry (range is 0-255, default value is 0).
`AlarmTime`	Holds the alarm time for the calendar entry, must be before `StartTime` entry. For entry type `ToDo`, it must be before `EndTime`.
Location	Holds the location name for an entry of type `Meeting`.

Status	Specifies the status for the calendar entry.
	`Tentative`
	`Confirmed`
	`TodoNeedsAction`
	`TodoCompleted`
	`TodoInProcess`
	`Cancelled`
	`NullStatus`: This is the default value, if not specified.

`RepeatDates`	Contains a list of out-of-sequence dates on which the calendar entry repeats.
`ExDates`	Contains a list of exception dates that is, occurrences in the original schedule that have been removed and may be replaced with a different occurrence.
`PhoneOwner`	Holds the details of the phone owner.
Organizer	Holds the organizer information, applicable for an entry of type `Meeting`.
Attendees	Holds the attendee information, applicable for an entry of type `Meeting`. For more information, see **Attendee structure**.
`RepeatRule`	Contains name-value pairs. For more information, see **Repeat Rule Structure**.

Table 6.56: Key value- Calendar Entry

Key	Type	Description
Type	int	Specifies the type of repeat rule: (Daily) (Weekly) (Monthly) (Yearly)
[StartDate]	datetime	Start Time. If not specified Entry StartTime is taken.
[UntilDate]	datetime	Holds the end date until which this entry will repeat.
[Interval]	int	Specifies the interval between instances of a repeating entry.
[DaysInWeek]	List	List of integers. Specifies on what days of the week the rule must repeat. Values are 0(Monday) to 6(Sunday).
[MonthDays]	List	List of integers. Specifies on what days of the month (0-30) the rule must repeat.
[DaysOfMonth]	List	List of maps each having the format: Day (0 - 6): 32 bit int WeekNum: 32 bit int
[Month]	int	Specifies the month for a yearly repeat rule. Values are 0(January) to 11(December).

Table 6.57: Key value- Repeat Rule Structure

Key	Type	Description
[CommonName]	string	Holds the common name for group scheduling.
[Role]	string	Specifies the role of a meeting participant. The possible values are: Required Optional NonParticipant Chair The default value is Required.
Address	string	Specifies the email address of a meeting participant.
[Status]	string	Specifies the status of an attendee. The possible values are: NeedsAction Accepted Tentative Confirmed Declined Completed Delegated InProcess The default value is NeedsAction
[Rsvp]	Boolean	Specifies whether or not a response is requested for this attendee. Default value is 0(False).

Table 6.58: Key value- Attendee Structure

6.4 Contacts

The Contacts service enables Python applications to access and manage contacts information. This information can reside in one or more contacts databases stored on a device or, in the SIM card database.
It enables applications to perform the following operations on the Contacts Database:

- Retrieve contact or group information
- Add contact or group
- Edit a particular contact or group
- Import and export a contact
- Delete a contact or group item

The following sample code is used to load the provider:

```
import scriptext
contact_handle = scriptext.load('Service.Contact', 'IDataSource')
```

The following table summarizes the Contacts Interface:

Service provider	`Service.Contact`
Supported interfaces	`IDataSource`

The following table lists the services available in Calendar:

Services	**Description**
`GetList` 6.4.1	Retrieves a list of contacts or groups in the default or specified database, also used to get the list of existing databases.
`Add` 6.4.2	Adds contact or group to the specified or default contacts database.
`Delete` 6.4.3	Deletes an array of contacts or groups from the specified or default contacts database.
`Import` 6.4.4	Imports contact to the specified contacts database.
`Export` 6.4.5	Exports the selected item from the contacts database specified as VCard.
`Organise` 6.4.6	Associate or Disassociate a list of contacts in a database to and from a group.

6.4.1 GetList

`GetList` retrieves a list of contacts, contact groups, or contacts databases. Contacts and contact groups are retrieved from the specified contacts database. If no database is specified, from the default one. This method can be called both in synchronous and asynchronous mode.

Note: Calls that retrieve a list of databases must be synchronous.

The following are the examples for using `GetList`:

Synchronous

```
list_contacts = contacts_handle.call('GetList', {'Type': u'Contact', 'Filter': {'SearchVal':
```

Asynchronous

```
event_id = contacts_handle.call('GetList', {'Type': u'Contact', 'Filter':{'SearchVal': u'Cra
```

where, `get_list` is a user defined function.

The following table summarizes the specification of `GetList`:

Interface	`IDataSource`
Description	Retrieves a list of contacts or groups in the default or specified database, also used to get the list of existing databases.
Response Model	Synchronous and asynchronous in case of Third Edition FP2 and Fifth Edition devices, except for `GetList` with Type as **Database**, which will always be synchronous. In case of Third Edition and Third Edition FP1 devices: Synchronous for Get Single Contact and Group. Asynchronous and synchronous for the rest of the functionality.
Pre-condition	`IDataSource` interface is loaded.
Post-condition	Nil

Input Parameters

`GetList` retrieves a list of contacts objects and metadata from the S60 messaging center based on Search or Sort inputs. This is an object that specifies what contact information is returned and how the returned information is sorted.

Output Parameters

Output parameter contains `ReturnValue`. It also contains `ErrorCode` and an `ErrorMessage` if the operation fails. `ReturnValue` contains complete contact item, group, or database information requested by `GetList`.

Errors

The following table lists the errors and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to list full name of contact matched by last name in asynchronous mode:

Name	Type	Range	Description

Type	unicode string	`Contact`
		`Group`
		`Database`
			Operation is performed on the specified type.

[Filter]	**Contact** (map) `[DBUri]`: unicode string `[id]`: unicode string `[SearchVal]`: unicode string **Group** (map) [DBUri]: unicode string [Id]: unicode string **Database** No map required.	`DBUri`: Database on which search must be performed. `Id`: Id is the unique identifier of the contact item or group to be retrieved. If `Id` is specified, `SearchVal` and `DBUri` are not required, and they will be ignored. `SearchVal`: Value searched for in the given `DBUri`. It cannot exceed 255 characters. If Filter is not supplied and Type is **Contact**, then it gets all the contacts of the default database. If Filter is not supplied and Type is **Group**, then it gets all the groups of the default database.	`SearchVal`: Value searched for in the given `DBUri` (If default database is not specified). If `SearchVal` is not specified then, it loads all the contacts in the database. `SearchVal` is looked for in first name and last name fields in case of Third Edition FP2 and Fifth Edition devices and it looks in all fields in case of Third Edition and Third Edition FP1 devices. With Type as **Contact**, it retrieves the list of contacts based on the Filter map input (if provided). With Type as **Group**, it gets a list of all the groups in the default database, if Filter is not specified. If Filter is specified, and `Id` is given, it fetches the group that the `Id` represents (DBUri is ignored in this case). Searching for a group by its name is not supported. With Type as **Database**, it gets the list of all the open databases.

Table 6.59: Input parameters Getlist

Name	Type	Range	Description
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.
`ReturnValue`	iterable list of maps	**Contact** (map) 6.61 **Group** (map) 6.62 **Database** (map) `DBUri`: string	Every **Next** operation on the iterator returns a map. Each map contains complete contact item/group/database information. Every **Next** operation on a contact gives a map with: `id`: It is a unique identifier for the contact that the map represents. *Key1, Key2..*: Gets as many keys available for a particular contact. For more information on keys, refer the section Key Values 6.4.7. Label and value give the information for the key. **Next**: In case, the key has multiple values, then it is added as another map. Every **Next** operation on group gives a map with: `id`: It is a unique identifier for the group that the map represents. `GroupLabel`: Label to the group. Contents: List of `ids` of the contacts that belong to the particular group. For example, *Contact Id1*, *Contact Id2*. Every **Next** operation on database gives a map with: `DBUri`: Uri of the database that is represented by the particular map.

Table 6.60: Output parameters for GetList

Key	**Value**	NA	NA
`id`	string	NA	NA
Key1	map	NA	NA
NA	Label	NA	string
NA	Value	NA	string
Key2	map	NA	NA
NA	Label	string	NA
NA	Value	string	NA
NA	Next	map	NA
NA	NA	Label	string
NA	NA	Value	string
NA	NA	Next	map

Table 6.61: Contact(map)

Key	Value
id	string
GroupLabel	string
Contents	List
NA	*Contact id1*
NA	*Contact id2*
NA	....

Table 6.62: Group(map)

Error code value	Description
0	Success
1002	Bad argument type

Table 6.63: Error codes

Error messages	Description
Contacts:GetList:Type is missing	Indicates Type is missing
Contacts:GetList: Invalid value for Type, Must be Contact/Group/Database	Indicates invalid value for Type
Contacts:GetList:Invalid Sort Type, Map is required	Indicates that the sort order type passed is invalid, map is expected
Contacts:GetList:Sort Order Value is not a String	Indicates that the value for order must be a string
Contacts:GetList:Invalid Type of Filter, Map is required	Indicates that the value for Filter must be a map
Contacts:GetList:Wrong Type of Sort Order value	Indicates that sort order value is not ascending or descending
Contacts:GetList:Wrong Type of Search value	Indicates that search value is not a string
Contacts:GetList:Wrong Type of ContentType	Indicates that the Type is not a string.

Table 6.64: Error messages

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def get_list(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
        print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The contacts matching are"
        for i in input_params["ReturnValue"]:
            print i["FirstName"]["Value"] + i["LastName"]["Value"]
    lock.signal()

# Load contacts module
contacts_handle = scriptext.load("Service.Contact", "IDataSource")

event_id = contacts_handle.call('GetList', {'Type': u'Contact', 'Filter':{'SearchVal': u'Cra

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.4.2 Add

`Add` is used to add a contact or contact group to a contacts database. If the contact or contact group already exists in the database, it is replaced with the new entry.

You can use this method to both add and edit contacts and contact groups. The data is added to the specified database. When no database is specified, the data is added to the default database. If the default database does not exist, `Add` creates a new database. This method can be called both in synchronous and asynchronous mode.

The following is an example for using `Add`:

Synchronous

```
contacts_handle.call('Add', {'Type': u'Contact', 'Data':
            {'FirstName': {'Label': u'first name', 'Value': u'Daniel'},
             'LastName': {'Label': u'last name', 'Value': u'Craig'},
             'MobilePhoneGen': {'Label': u'mobile', 'Value': u'9008025211'},
             'EmailHome': {'Label': u'email', 'Value': u'dcraig@ford.com'}}})
```

The following table summarizes the specification of `Add`:

Input Parameters

Input parameter contains the contact information to add or edit and also the target database.

Output Parameters

The output contains `ErrorCode` and an `ErrorMessage` if the operation fails.

Interface	`IDataSource`
Operation	Adds contact/group to the specified/default contacts database.
Response Model	Synchronous and asynchronous for Third Edition FP2 and Fifth Edition devices. Synchronous for Third Edition and Third Edition FP1 devices.
Pre-condition	`IDataSource` interface is loaded. For editing an existing contact/group, the specified Id must exist. You must use `GetList` to retrieve the Id for editing.
Post-condition	Adds a new contact item to the database in case of add and updates an existing contact in case of edit.

Name	Type	Range	Description
Type	unicode string	`Contact` `Group`	Operation performed on the specified Type.
Data	**Contact** (map) 6.66 **Group** (map) `[DBUri]`: string `[id]`: string `GroupLabel`: string	All string values in the map are unicode. *Key 1, Key 2, and so on* are based on the keys supported. `id`: For Type `Contact`, `Id` is the unique identifier for the contact to be modified. `id`: for Type `Group`, `Id` is the unique identifier for the group to be modified. `GroupLabel`: Label for the group being added or modified.	Information about the contact/group to be added to the contacts database. You must not set the Id/Id field to add a new entry and also, must not modify the value of the Id/Id field when editing an existing entry. You must use the id that is given to it by `GetList` sapi. In case of editing an existing contact, it overwrites the existing entry of that id completely. Edit operation is not editing of selected fields, it is replacement of the entire contact/group. If string value given for Value key is empty, then that field is not added to the contact. Rest of the fields are still added. If the database does not support Label then, the Label is ignored if it is given (sim database does not support Label).

Table 6.65: Input parameters for Add

Key	Value	NA	NA
`[DBUri]`	string	NA	NA
`[id]`	string	NA	NA
Key1	map	NA	NA
NA	Label	NA	string
NA	Value	NA	string
Key2	map	NA	NA
NA	Label	string	NA
NA	Value	string	NA
NA	Next	map	NA

Table 6.66: Contact(map)

Name	Type	Range	Description
ErrorCode	int	NA	Contains the SAPI specific error code when the operation fails.
ErrorMessage	string	NA	Error Description in Engineering English.

Table 6.67: Output parameters for Add

Errors

The following table lists the errors and their values:

Error code value	Description
0	Success
1002	Bad argument type
1004	Service not supported
1005	Service in use
1011	Access denied

Table 6.68: Error codes

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to add a contact information:

```
# Load contacts module
contacts_handle = scriptext.load('Service.Contact', 'IDataSource')
try:
    contacts_handle.call('Add', {'Type': u'Contact', 'Data':
             {'FirstName': {'Label': u'first name', 'Value': u'Daniel'},
                 'LastName': {'Label': u'last name', 'Value': u'Craig'},
                 'MobilePhoneGen': {'Label': u'mobile', 'Value': u'9008025211'},
                 'EmailHome': {'Label': u'email', 'Value': u'dcraig@ford.com'}}})
    print "Contact added Successfully"

except scriptext.ScriptextError, err:
    print "Error adding the contact : ", err
```

6.4.3 Delete

Delete is used to delete one or more contacts or contact groups from a contact database. It deletes data from a specified database or from the default database if you do not specify a database. This method can be called both in synchronous and asynchronous mode.

The following is an example for using Delete:

Asynchronous

```
event_id = contacts_handle.call('Delete', {'Type': u'Contact', 'Data': {'IdList': [req_id]}}
```

Error messages	Description
`Contacts:Add:Type is missing`	Indicates Type is missing
`Contacts:Add:Invalid Type, must be Contact/Group`	Indicates invalid value for Type, can have values Contact/Group only.
`Contacts:Add:Invalid Sort Type, Map is required`	Indicates that the sort order type passed is invalid, map is expected
`Contacts:Add:Add Data is Missing`	Indicates that the key `Data` is missing.
`Contacts:Add:Add data Map is Missing`	indicates that the value of the `Data` is missing.
`Contacts:Add:Group Label is Missing`	Indicates that the label for group is missing.
`Contacts:Add:Mandatory Argument is not present`	Indicates not all mandatory parameters are present.
`Contacts:Add:Type of Contact Id is wrong`	Indicates that Contact Id value is not a string.
`Contacts:Add:Invalid Type of Data, Map is required`	Indicates that `Data` value must be of Type map.
`Contacts:Add:Invalid Type of Field value, Map is required`	Indicates that value of a given Key (for example: *Key1*, *Key2*), is not a Map.
`Contacts:Add:Invalid Type of NextField value, Map is required`	Indicates that value of Next field is not a Map.
`Contacts:Add:Invalid Type of Id`	Indicates that value Group Id is not a string.
`Contacts:Add:Invalid Type of GroupLabel`	Indicates that value Group Label is not a string.
`Contacts:Add:Wrong Type of ContentType`	Indicates that the value Type is not a string.
`Contacts:Add:Atleast one field is required`	Indicates that atleast one field must be specified.
`Contacts:Add:Group Label is Empty`	Indicates that the mandatory input parameter `GroupLabel` is an empty string.
`Contacts:Add:Invalid Field Key:`*`fieldkey`*	Indicates that the key *fieldkey*, is not a valid.
`Contacts:Add:Field Key Not Supported on this Database:`*`fieldkey`*	Indicates that the *fieldkey* is not supported on the given database.
`Contacts:Add:Field Value too long for key:`*`fieldkey`*	Indicates that the *fieldkey* has a greater length than the maximum allowed for the particular key.

Table 6.69: Error messages

where, `del_contact` is a user defined callback function.

The following table summarizes the specification of `Delete`:

Interface	`IDataSource`
Operation	Deletes an array of contacts/groups from the specified or default contacts database.
Response Model	Asynchronous and synchronous for Third Edition FP2 and Fifth Edition devices. Synchronous for Third Edition and Third Edition FP1 devices.
Pre-condition	`IDataSource` interface is loaded. Contact must exist in the contacts database. The `IDs` can be obtained from `GetList`.
Post-condition	Nil

Input Parameters

The following table describes input parameter. The default contacts database is **cntdb://c:contacts.cdb**. The SIM card database is **sim://global_adn**. The contacts or contacts groups to be deleted must exist in the specified database. You must use `GetList` to retrieve the IDs of the entries you want to delete.

Name	Type	Range	Description
Type	unicode string	`Contact` `Group`	Operation is performed on the specified type.
Data	**Contact** or **Group** (map) `[DBUri]`: unicode string `IdList`: List *[id1, id2, id3]*	All string values in the map are unicode.	`IdList` is a mandatory field. You must specify the contact ids or group ids to delete a set of contacts/groups from the contacts database given by the list. For example, *id1*, *id2*, *id3* are the ids of the contacts/groups. `DBUri` is optional, it operates on the specified database or on the default database, if specified.

Table 6.70: Input parameters for Delete

Output Parameters

The output is an object, which contains `ErrorCode` and an `ErrorMessage` if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails and `SErrNone` on success.
`ErrorMessage`	string	NA	Error Description in Engineering English.

Table 6.71: Output parameters for Delete

Errors

The following table lists the errors and their values:

Error Messages

Error code value	Description
0	Success
1002	Bad argument type
1004	Service not supported
1005	Service in use
1011	Access denied

Table 6.72: Error codes

Error messages	Description
Contacts:Delete:Type is missing	Indicates Type is missing
Contacts:Delete:Invalid Type, must be Contact/Group	Indicates invalid value for Type, if it is not Contact or Group.
Contacts:Delete:Delete data Missing	Indicates that the key Data is missing.
Contacts:Delete:Invalid Type of Data, Map is required	Indicates that the value of the Data is not present and Map is expected.
Contacts:Delete:List of Ids is Missing	Indicates that the list of Contact Ids to be deleted is missing.
Contacts:Delete:Type of IdList is wrong, List is required	Indicates that value of IdList is not a List.
Contacts:Delete:Invalid Type of Id	Indicates that Contact/Group Id is not a string.
Contacts:Delete:Wrong Type of ContentType	Indicates that the value for Type is not a string.
Contacts:Delete:Mandatory Argument is not present	Indicates that Type is not a string.

Table 6.73: Error messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to delete a contact:

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def del_contact(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in deleting the contact"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
           print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The contact is deleted"
        lock.signal()

# Load contacts module
contacts_handle = scriptext.load("Service.Contact", "IDataSource")

list_contacts = contacts_handle.call('GetList', {'Type': u'Contact', 'Filter': {'SearchVal':
for i in list_contacts:
     req_id = i['id']

event_id = contacts_handle.call('Delete', {'Type': u'Contact', 'Data':{'IdList': [req_id]}},

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.4.4 Import

`Import` is used to import a contact to a contacts database. The information must be imported from a VCard file. This API can be called both in synchronous and asynchronous mode.

The following is an example for using `Import`:

Asynchronous

```
event_id = contacts_handle.call('Import', {'Type': u'Contact','Data':{'SourceFile':u'c:\\Dat
```

where, `get_import` is a user defined callback function.

The following table summarizes the specification of `Import`:

Input Parameters

Input parameter specifies the contact to import and optionally the target database.

Example of vcard format:

Interface	IDataSource
Operation	Imports contact to the specified contacts database.
Response Model	Asynchronous and synchronous for Third Edition FP2 and Fifth Edition devices. Synchronous for Third Edition and Third Edition FP1 devices.
Pre-condition	IDataSource interface is loaded.
Post-condition	Updates database with imported contact.

Name	Type	Range	Description
Type	unicode string	Contact	Operation is performed on the specified type.
Data	map [DBUri]: unicode string SourceFile: unicode string	All string values in the map are unicode. **Note:** SourceFile can have any extension or no extension. SourceFile has to be in Vcard format. Example of vcard format is given below	DBUri: Imports contact to the specified database or to the default database if not specified. SourceFile: Imports contact from the specified file. SourceFile is the complete path to the file. It cannot be greater than 256 characters.

Table 6.74: Input parameters Import

```
BEGIN:VCARD
VERSION:2.1
N:Kent; Clark
FN:Clark Kent
ORG:Superman Inc.
TITLE:Super Man
TEL;WORK:VOICE:(111) 556-9898
TEL;HOME;VOICE:(090) 556-6767
ADR;WORK:;;3rd Rock from Sun;Solar System;Milky Way
LABEL;WORK;ENCODING=QUOTED-PRINTABLE:3rd Rock from Sun=0D=0ASolar System=0D=0AMilky Way
ADR;HOME:;;Krypton
LABEL;HOME;ENCODING=QUOTED-PRINTABLE:Krypton
EMAIL;PREF;INTERNET:clarkkent@krypton.com
REV:2008042T195243Z
END:VCARD
```

Output Parameters

The output is an object, which contains ErrorCode and an ErrorMessage if the operation fails.

Name	Type	Range	Description
ErrorCode	int	NA	Contains the SAPI specific error code when the operation fails and SErrNone on success.
ErrorMessage	string	NA	Error Description in Engineering English.

Table 6.75: Output parameters for Import

Errors

The following table lists the errors and their values:

Error code value	Description
1002	Bad argument type
1011	Access denied
1017	Path not found

Table 6.76: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
Contacts:Import:Type is missing	Indicates Type is missing
Contacts:Import:Invalid Type, it must be Contact	Indicates invalid value for Type, it can be only Contact.
Contacts:Import:Import data Missing	Indicates that the key Data is missing.
Contacts:Import:Invalid Type of Data, Map is required	Indicates that the value of the Data is not a Map.
Contacts:Import:Import Source Filename is Missing	Indicates the argument to signify the filename of the imported file is missing.
Contacts:Import:Import Source File is not a String	Indicates that the filename specified is not a string.
Contacts:Import:Wrong Type of ContentType	Indicates that the value for Type is not a string.
Contacts:Import:Mandatory Argument is not present	Indicates that not all mandatory parameters are present.
Contacts:Import:Import DataBaseUri is not a String	Indicates that the uri specified is not a string.
Contacts:Import:Filename too long	Indicates that filename has exceeded 256 characters.

Table 6.77: Error messages

Example

The following sample code illustrates how to import contacts from a VCard format file:

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def get_import(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The contacts are imported"
        lock.signal()

# Load contacts module
contacts_handle = scriptext.load("Service.Contact", "IDataSource")

event_id = contacts_handle.call('Import', {
                        'Type': u'Contact',
                        'Data':{'SourceFile': u'c:\\Data\\python\\VCARD.txt'}}, callback=get

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.4.5 Export

`Export` is used to export a contact from a contacts database. The information is exported to a vCard file. This API can be called both in synchronous and asynchronous mode.

The following is an example for using `Export`:

Asynchronous

```
event_id = contacts_handle.call('Export', {'Type': u'Contact','Data': {'DestinationFile': u'
```

where, `export_contact` is a user defined callback function.

The following table summarizes the specification of `Export`:

Interface	`IDataSource`
Operation	Exports the selected item from the contacts database specified as VCard.
Response Model	Asynchronous and synchronous for Third Edition FP2 and Fifth Edition devices. Synchronous for Third Edition and Third Edition FP1 devices.
Pre-condition	Valid `IDataSource` interface is loaded. Valid contact store must exist. The specified contact ID, retrieved using `GetList`, must be available.
Post-condition	Exports contact to the specified file and creates a file in the specified location.

Input Parameters

The following table describes input parameter properties:

Name	Type	Range	Description
Type	unicode string	`Contact`	Operation is performed on the specified type.
Data	map `[DBUri]`: unicode string `DestinationFile`: unicode string `id`: unicode string	All string values in the map are unicode. **Note:** `DestinationFile` can have any extension or no extension, usually it is **.vcf** `DestinationFile` is of Vcard format. Example of vcard formatis given below If complete path is not specified, the file is created in private folder of the process.	`DBUri`: Exports contact from the specified database or to the default database if not specified. `DestinationFile`: Exports contact to the specified file. It cannot be greater than 256 characters. `id`: Exports the contact item with the specified id.

Table 6.78: Input parameters Export

```
BEGIN:VCARD
VERSION:2.1
N:Kent; Clark
FN:Clark Kent
ORG:Superman Inc.
TITLE:Super Man
TEL;WORK:VOICE:(111) 556-9898
TEL;HOME;VOICE:(090) 556-6767
ADR;WORK:;;3rd Rock from Sun;Solar System;Milky Way
LABEL;WORK;ENCODING=QUOTED-PRINTABLE:3rd Rock from Sun=0D=0ASolar System=0D=0AMilky Way
ADR;HOME:;;Krypton
LABEL;HOME;ENCODING=QUOTED-PRINTABLE:Krypton
EMAIL;PREF;INTERNET:clarkkent@krypton.com
REV:2008042T195243Z
END:VCARD
```

Output Parameters

The output is an object, which contains `ErrorCode` and an `ErrorMessage` if the operation fails.

Errors

The following table lists the errors and their values:

Error Messages

Name	Type	Range	Description
ErrorCode	int	NA	Contains the SAPI specific error code when the operation fails and SErrNone on success.
ErrorMessage	string	NA	Error Description in Engineering English.

Table 6.79: Output parameters for Export

Error code value	Description
1002	Bad argument type
1010	Entry exists
1012	Item not found

Table 6.80: Error codes

The following table lists the error messages and their description:

Error messages	Description
Contacts:Export:Type is missing	Indicates Type is missing
Contacts:Export:Invalid Type, it must be Contact	Indicates invalid value for Type, it can be only Contact.
Contacts:Export:Export data Missing	Indicates that the key Data is missing.
Contacts:Export:Invalid Type of Data, Map is required	Indicates that the value of the Data is not a Map.
Contacts:Export:Export Destination Filename is Missing	Indicates the argument to signify the filename to which contact is to be exported is missing.
Contacts:Export:Contact Id to be exported is missing	Indicates that the id of the contact to be exported is missing.
Contacts:Export:Wrong Type of ContentType	Indicates that the value for Type is not a string.
Contacts:Export:Destination Filename is of wrong Type	Indicates that the filename is not a string.
Contacts:Export:Id is of wrong Type	Indicates that the id is not a string.
Contacts:Export:Mandatory Argument is not present	Indicates that not all mandatory arguments are present.
Contacts:Export:Export DataBaseUri is not a String	Indicates that the uri specified is not a string.
Contacts:Export:Filename too long	Indicates that filename has exceeded 256 characters.

Table 6.81: Error messages

Example

The following sample code illustrates how to export contacts to a file in VCard format:

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def export_contact(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:   # Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The contact is exported"
        lock.signal()

# Load contacts module
contacts_handle = scriptext.load("Service.Contact", "IDataSource")

list_contacts = contacts_handle.call('GetList', {'Type': u'Contact', 'Filter': {'SearchVal':
for i in list_contacts: req_id = i['id']

event_id = contacts_handle.call('Export', {'Type': u'Contact', 'Data': {'DestinationFile': u

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.4.6 Organise

`Organise` is used to add contacts to a contact group (association) or remove contacts from a contact group (disassociation). The operation is performed on the specified database or, if no database is specified, on the default one.

This method can be called in both synchronous and asynchronous mode.

The following is an example for using `Organise`:

Asynchronous

```
event_id = contacts_handle.call('Organise', {'Type': u'Group','Data': {'id': unicode(req_gro
```

where, `export_contact` is a user defined function.

The following table summarizes the specification of `Organise`:

Input Parameters

Input parameter specifies which contact group to organize.

Output Parameters

The output is an object, which contains `ErrorCode` and an `ErrorMessage` if the operation fails.

Errors

Interface	`IDataSource`
Operation	Associates or disassociates a list of contacts in a database to and from a group.
Response Model	Asynchronous and synchronous for Third Edition FP2 and Fifth Edition devices. Synchronous for Third Edition and Third Edition FP1 devices.
Pre-condition	Valid `IDataSource` interface is loaded. The IDs specified for group and contact must exist and can be retrieved using `GetList`.
Post-condition	Contacts from the default or specified contacts database are associated/ disassociated to and from a group.

Name	Type	Range	Description
Type	unicode string	`Group`	Operation is performed on the specified type.
Data	map `[DBUri]`: unicode string `id`: unicode string `IdList`: List *id1* *id2* and so on	All string values in the map are unicode. *id1*, *id2*, ... are strings. These are obtained by calling `GetList`.	`DBUri`: Organise groups in the specified database or to the default database if not specified. `id`: Associate or disassociate contacts to the particular `Id`. `IdList`: Organise a particular list of contacts.
`OperationType`	unicode string	**OperationType:** Associate Disassociate	NA

Table 6.82: Input parameters Organise

Name	Type	Range	Description
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails and `SErrNone` on success.
`ErrorMessage`	string	NA	Error Description in Engineering English.

Table 6.83: Output parameters for Organise

The following table lists the errors and their values:

Error code value	Description
1002	Bad argument type
1011	Access denied

Table 6.84: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
Contacts:Organise:Type is missing	Indicates Type is missing
Contacts:Organise:Invalid Content Type, it must be Group	Indicates invalid value for Type, it can be only Contact.
Contacts:Organise:Organise Data Missing	Indicates that the key Data is missing.
Contacts:Organise:Invalid Type of Data, Map is required	Indicates that the value of the Data is not a Map.
Contacts:Organise:List of Ids is missing	Indicates that Contact id list is missing.
Contacts:Organise:Id is missing	Indicates that Group id is missing.
Contacts:Organise:OperationType is Missing	Indicates that OperationType is missing.
Contacts:Organise:Operation Type is Wrong	Indicates that OperationType is not a string.
Contacts:Organise:Invalid Operation Type	Indicates that the Operation type is neither associate nor disassociate.
Contacts:Organise:Id type is wrong	Indicates that the id is not a string.
Contacts:Organise:IdList type is wrong	Indicates that IdList is not a of type List.
Contacts:Organise:Wrong Type of ContentType	Indicates that the value for Type is not a string.
Contacts:Organise:Mandatory Argument is not present	Indicates that not all mandatory arguments are present.
Contacts:Organise:Id List is empty	Indicates that the mandatory Idlist is given but is empty.

Table 6.85: Error messages

Example

The following sample code illustrates how to associate or disassociate a contact from a group:

```
import scriptext
import e32
# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()
req_groupid = []

# Callback function will be called when the requested service is complete
def export_contact(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:   # Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The contact is organised"
        lock.signal()

# Load contacts module
contacts_handle = scriptext.load("Service.Contact", "IDataSource")

list_contacts = contacts_handle.call('GetList', {'Type': u'Contact', 'Filter': {'SearchVal':

for i in list_contacts: req_id = i['id']

list_groups = contacts_handle.call('GetList', {'Type': u'Group'})

for j in list_groups:
    req_groupid.append(j['id'])

event_id = contacts_handle.call('Organise', {'Type': u'Group', 'Data': {'id': unicode(req_gr

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.4.7 Key Values

For uri- **cntdb://c:contacts.cdb**:

- All keys are supported on S60 3rd Edition FP2 and S60 5th Edition devices.
- Keys documented in green are not supported in S60 3rd Edition and S60 3rd Edition FP1.
- Keys documented in blue are not supported in S60 3rd Edition only.

For uri- **sim://global_adn**, which is supported only on **3.2** and **5.0**:

- Keys documented in red are only supported.

Keys supported are dependent on the accessing database and not platform dependent.

The keys listed in the following table are a superset of all the keys supported on all Third Edition and Fifth Edition platforms and different databases altogether. If you try to add an unsupported key on a given database and a given platform, the API returns an error message.

Note: SyncClass field is added to the contact by default, with a **Synchronisation** label and **private** value. (unless specified as **public**). All values other than **private** or **public** are stored as **private**.

Key	Description
LastName	Last name field
FirstName	First name field
Prefix	Name prefix field
Suffix	Name suffix field
SecondName	Second name field
LandPhoneHome	Home land phone number
MobilePhoneHome	Home mobile phone number
VideoNumberHome	Home video number field
FaxNumberHome	Home FAX number
VoipHome	Home VOIP phone number
EmailHome	Home Email address
URLHome	Home URL
AddrLabelHome	Home Address label
AddrPOHome	Home address post office
AddrEXTHome	Home address extension
AddrStreetHome	Home address street
AddrLocalHome	Home address local
AddrRegionHome	Home address region
AddrPostCodeHome	Home address post code
AddrCountryHome	Home address country
JobTitle	Job title field
CompanyName	Company name field
LandPhoneWork	Work land phone number
MobilePhoneWork	Work mobile phone number
VideoNumberWork	Work video number field
FaxNumberWork	Work FAX number
VoipWork	Work VOIP
EmailWork	Work email id
URLWork	Work URL field
AddrLabelWork	Work address label
AddrPOWork	Work address post office
AddrEXTWork	Work address extension
AddrStreetWork	Work address street
AddrLocalWork	Work address local field
AddrRegionWork	Work address region
AddrPostCodeWork	Work address post code
AddrCountryWork	Work address country
LandPhoneGen	General land phone number
MobilePhoneGen	General mobile phone number
VideoNumberGen	General video number
FaxNumberGen	General FAX number
VOIPGen	General VOIP
POC	POC field (Push to Talk Over Cellular)
SWIS	SWIS field (See What I See).
SIP	SIP Identity field
EmailGen	General Email id
URLGen	General URL field
AddrLabelGen	General address label
AddrPOGen	General address post office
AddrExtGen	General address extension
AddrStreetGen	General address street
AddrLocalGen	General address local field
AddrRegionGen	General address region
AddrPostCodeGen	General address post code
AddrCountryGen	General address country
PageNumber	Pager number
DTMFString	DTMF String
Date	Date field
Note	Note field
Ringtone	Ring tone field
MiddleName	Middle name field

6.5 Landmarks

The Landmark service enable run-time client applications to manage landmarks in a consistent manner, within a terminal. Landmarks can be stored in one or more databases within a terminal. You can manage landmarks and landmark categories within a database using this service.

A category is characteristic of a landmark. Categories denote a class of geographical or architectural interest, attraction or activity-related types of objects. Categorization is very useful when searching for landmarks by type. The classification of Landmark categories is as follows:

- **Local category**: You can create the local category. It does not have global IDs, which distinguish it from the global categories.
- **Global category**: Each global landmark category has a unique global identifier associated with it.

Landmarks and Categories are stored in landmark databases. The classifications of Landmark databases are as follows:

- **Local database**: Resides in the phone or in a device mapped to the file system of the phone.
- **Remote database**: Stored in third party servers, accessed using a specific protocol. Currently remote database and associated operations are not supported.

The `Landmarks` service provides the user facilities to access, create, add, delete, import, export, and organize landmarks.

The following sample code is used to load the provider:

```
import scriptext
landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
```

The following table summarizes the Landmarks Interface:

Service provider	`Service.Landmarks`
Supported interfaces	`IDataSource`

The following table lists the services available in Landmarks:

Services	**Description**
`New` 6.5.1	Get a template of landmark or category.
`Getlist` 6.5.2	Get a list of landmark databases, landmarks or landmark categories based on given criteria.
`Add` 6.5.3	Add or update a landmark and landmark category.
`Delete` 6.5.4	Delete a landmark and landmark category.
`Import` 6.5.5	Imports landmark(s).
`Export` 6.5.6	Launches application based on the Document.
`Organise` 6.5.7	Associate or disassociate a landmark category with a set of landmarks.

6.5.1 New

`New` method is used to create an empty landmark or landmark category item. You can use the new item as a template. It is available only in synchronous mode.

The following is an example for using `New`:

```
new_output = landmark_handle.call('New', {'Type': u'Landmark'})
```

The following table summarizes the specification of `New`:

Interface	`IDataSource`
Operation	Creates an empty landmark or category item.
Response Model	Synchronous
Pre-condition	IDataSource interface is loaded.
Post-condition	Nil

Input Parameters

Input parameter specifies the content type to create.

Name	Type	Range	Description
Type	unicode string	`Landmark` `Category`	Specifies the content type to create.

Table 6.86: Input parameters for New

Output Parameters

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue`	map Landmark or category, as discussed in the Key Values 6.5.8 section.	NA	For content template description, see Landmark and Category in the Key Values 6.5.8 section.

Table 6.87: Output parameters for New

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to create an empty landmark/category item:

Error code value	Description
1007	No memory

Table 6.88: Error codes

Error messages	Description
Landmarks:New:Type is missing	Indicates Type is missing or data type of Type is mismatched.
Landmarks:New:Type is invalid	Indicates that Type is not a value in the given range.

Table 6.89: Error messages

```
import scriptext

landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
try:
    new_output = landmark_handle.call('New', {'Type': u'Landmark'})
    error = new_output['ErrorCode']
    if error != 0:
        print "Error in creating the landmark item"
    else:
        print "The Landmark item is created"

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.5.2 GetList

`GetList` is used to retrieve information about landmarks, landmark categories, or landmark databases. Landmarks and landmark categories are retrieved from the specified landmark database or, if no database is specified, from the default one.

The following are the examples for using `GetList`:

Synchronous

```
landmarkinfo = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                 'Filter':{'uDatabaseURI':  u'dataBaseUri',
                                                           'LandmarkName': u'AnyLandMarkNm'},
                                                 'Sort' :{'Key': u'LandmarkName',
                                                          'Order': u'Descending'}})
```

Asynchronous

```
event_id = landmark_handle.call('GetList', {'Type': u'Landmark',
                                            'Filter':{'uDatabaseURI':u'dataBaseUri',
                                                      'LandmarkName':u'AnyLandMarkNm'},
                                            'Sort':{'Key':u'LandmarkName',
                                                    'Order':u'Descending'}},
                                callback=get_list)
```

where, `get_list` is a user defined callback function.

The following table summarizes the specification of `GetList`:

Interface	`IDataSource`
Description	Retrieves an iterable on items qualified by search criteria.
Response Model	Synchronous and asynchronous, depending on the criteria and use case.
Pre-condition	`IDataSource` interface is loaded.
Post-condition	The iterable points to the first element in the list from an active or specified database. The default or active database opened for reading landmarks and categories. Creates the database, if it does not exist and is set as active.

Input Parameters

Input parameter specifies what landmark information is returned and how the returned information is sorted.

Output Parameters

Output parameters contain the requested information. They also contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to query a list of Landmarks with search criteria, in asynchronous mode:

Name	Type	Range	Description

Type	unicode string	`Landmark`
		`Category`
		`Database`
			Performs operation based on the specified content types.

[Filter]	map
	Landmark search criteria: For more information, see Key Values 6.5.8 section in Landmarks.
	Category search criteria: For more information, see Key Values 6.5.8 section in Landmarks.
	Database search criteria `[DbProtocol]`: unicode string
		Landmark search criteria:
		It is the map containing the landmark search fields for setting the search criteria.
		
		Text Criterion:
		The following fields can be specified:
		`LandmarkName`
		`LandmarkDesc`
		
		Nearest Criterion:
		The following fields need to be specified:
		`LandmarkPosition`
		`CoverageRadiusOption`
		`MaximumDistance`
		
		Category Criterion:
		The following field needs to be specified:
		`CategoryName`
		
		Area Criterion:
		The following field needs to be specified:
		`BoundedArea`
		
		Category search criteria:
		It is the map containing the landmark category search fields for setting the search criteria.
		
		Database search criteria:
		`DbProtocol`: Search criteria are the protocol string.
			Optional Parameter. If filter is not specified, an iterator to all entries of the specified type is returned.
			
			Landmark search criteria:
			Specify one or more search criteria to retrieve a list of landmarks.
			
			`CoverageRadiusOption` and `MaximumDistance` are required only when landmark Position is specified.
			
			Category Search Criteria:
			Specify text with wild cards to iterate through the list of categories.
			
			Database Search Criteria:
			If you do not specify protocol then all available databases will be listed.

[Sort]	map
	[Key]: unicode string
	Order: unicode string
		Key:
		Possible Values for the types:
		Landmark:
		`LandmarkName`
		Category:
		`CategoryName`
		Database:
		`DatabaseURI`
		Order:
		Ascending
		Descending
			Optional Parameter.
			
			Default Value for Order
			Type Landmarks: Ascending
			Type Category: None
			Type Database: Ascending
			
			Sorts qualified list based on sort key and sort order.

Table 6.90: Input parameters for Getlist

Name	Type	Range (Type: string)	Description
ReturnValue	Iterable To maps of requested type	map: Landmark Category Database	Iterator to the retrieved list of items of the requested type. For map of type, see Landmark, Category, and Database in the Key Values 6.5.8 section.
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.

Table 6.91: Output parameters for GetList

Error code value	Description
-304	General Error
0	Success
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1012	Item not found

Table 6.92: Error codes

Error messages	Description
Landmarks:GetList:Type is missing	Indicates Type is missing or data type of Type is mismatched.
Landmarks:GetList:Type is invalid	Indicates that Type is not a value in the given range.
Landmarks:GetList:Data is missing	Indicates Data is missing or data type of Data is mismatched.
Landmarks:GetList:LandmarkPosition is missing	Indicates LandmarkPostion is missing or data type of LandmarkPostion is mismatched.
Landmarks:GetList:Latitude is missing	Indicates Latitude is missing or data type of Latitude is mismatched.
Landmarks:GetList:Longitude is missing	Indicates Longitude is missing or data type of Longitude is mismatched.
Landmarks:GetList:BoundedArea is missing	Indicates BoundedArea is missing or data type of BoundedArea is mismatched.
Landmarks:GetList:NorthLatitude is missing	Indicates NorthLatitude is missing or data type of NorthLatitude is mismatched.
Landmarks:GetList:SouthLatitude is missing	Indicates SouthLatitude is missing or data type of SouthLatitude is mismatched.
Landmarks:GetList:EastLongitude is missing	Indicates EastLongitude is missing or data type of EastLongitude is mismatched.
Landmarks:GetList:WestLongitude is missing	Indicates WestLongitude is missing or data type of WestLongitude is mismatched.
Landmarks:GetList:MaximumMatches is invalid	Indicates MaximumMatches value provided is invalid that is, equal or less than 0.
Landmarks:GetList:Sort is missing	Indicates Sort is missing or data type of Sort is mismatched.

Table 6.93: Error messages

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def get_list(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The landmarks are"
        for i in input_params["ReturnValue"]:
            print "Name of Landmark"
            print i["LandmarkName"]
            print "Description of Landmark"
            print i['LandmarkDesc']

    lock.signal()

# Async Query a list of Landmarks with search criteria
landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
event_id = landmark_handle.call('GetList', {'Type': u'Landmark',
                                            'Filter': {'uDatabaseURI':u'dataBaseUri',
                                                       'LandmarkName':u'AnyLandMarkNm'},
                                            'Sort': {'Key':u'LandmarkName',
                                                     'Order':u'Descending'}},
                                callback=get_list)

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.5.3 Add

`Add` is used to add or modify an object to the active or specified landmark database. It accepts a set of input parameters that define the Type and its details to add. It is available only in synchronous mode.

The following is an example for using `Add`:

```
add_output = landmark_handle.call('Add', {'Type': u'Landmark',
                                          'Data': {'LandmarkName': u'land1'}})
```

The following table summarizes the specification of `Add`:

Input Parameters

Input parameter specifies what landmark information is returned and how the returned information is sorted.

Output Parameters

Output parameters contain the requested information. They also contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Interface	`IDataSource`
Description	Adds or Modifies an object to the active or specified landmark database.
Response Model	Synchronous
Pre-condition	`IDataSource` interface is loaded. Update is performed only on an existing landmark or category. You must provide ID of the landmark or category to update. The ID is retrieved by calling `GetList`.
Post-condition	The default or active database is opened for reading landmarks and categories. A default database is created, if it does not exist and is set as active. Landmark/category is added or edited in the specified database or the active databases, in case database is not specified. A new database is created within a terminal, in case of addition of a new database.

Name	Type	Range	Description
Type	unicode string	`Landmark` `Category`	Performs operation based on the specified content types.
Data	**Landmark** map (`LandmarkMap`) `[DatabaseURI]`: unicode string **Category** map (`CategoryMap`) `[DatabaseURI]`: unicode string	**Landmark** `DatabaseURI`: The Uri of the database to which the landmark must be added or edited. If this is not specified then, landmark or category is added to default database. **LandmarkMap**: The map containing the landmark fields which are added or edited. **Category** `DatabaseURI`: The Uri of the database to which the category must be added or edited. **CategoryMap**: The map containing the category fields which are added or edited.	Data Fields contain information about the object to be added. Do not set the ID field to add a new landmark/category. For adding landmark/category you can make use of `New`. Do not modify the ID field when editing an existing landmark/category which is retrieved from `GetList`.

Table 6.94: Input parameters for Add

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue`	string	NA	ID of the landmark/category that was added or modified.

Table 6.95: Output parameters for Add

Errors

The following table lists the error codes and their values:

Error code value	Description
-304	General Error
0	Success
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1006	Service not ready
1011	Access denied
1012	Item not found

Table 6.96: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
Landmarks:Add:Type or Data is missing	Indicates Type is missing or data type of Type is mismatched.
Landmarks:Add:Type is invalid	Indicates that Type is not a value in the given range.
Landmarks:Add:Data is missing	Indicates Data is missing or data type of Data is mismatched.
Landmarks:Add:LandmarkPosition is missing	Indicates LandmarkPostion is missing or data type of LandmarkPostion is mismatched.
Landmarks:Add:CategoryInfo is missing	Indicates CategoryInfo is missing or data type of CategoryInfo is mismatched.
Landmarks:Add:IconIndex is missing	Indicates IconIndex is missing or data type of IconIndex is mismatched.
Landmarks:Add:LandmarkFields is missing	Indicates LandmarkFields is missing or data type of LandmarkFields is mismatched.
Landmarks:Add:CategoryName is missing	Indicates CategoryName is missing or data type of CategoryName is mismatched.
Landmarks:Add:DatabaseURI is missing	Indicates DatabaseURI is missing or data type of DatabaseURI is mismatched.

Table 6.97: Error messages

Example

The following sample code illustrates how to add or modify an object to the active / specified landmark database:

```
import scriptext

landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
try:
    add_output = landmark_handle.call('Add', {'Type': u'Landmark',
                                              'Data': {'LandmarkName': u'land1'}})
    error = add_output['ErrorCode']

    if error != 0:
        print "Error in adding the landmark"
    else:
        print "Landmark added"

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.5.4 Delete

`Delete` is used to delete the user specified object or data from the active or specified landmark database. It accepts a set of input parameters that define the Type and data for performing the delete operation. It is available only in synchronous mode.

The following is an example for using `Delete`:

```
getlist_output = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                   'Filter': {'LandmarkName': u'land1'}})
```

The following table summarizes the specification of `Delete`:

Interface	`IDataSource`
Description	Deletes an object from the active or specified landmark database.
Response Model	Synchronous
Pre-condition	`IDataSource` interface is loaded.
Post-condition	The default or active database opened for reading landmarks and categories. A default database is created, if it does not exist and is set as active. A Landmark/category is deleted from an active or specified database. Deleting a database, deletes it from terminal.

Input Parameters

Input parameter specifies the Type Landmark/category to delete and details of the particular Type.

Output Parameters

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

Name	Type	Range	Description
Type	unicode string	`Landmark` `Category`	Performs operation based on the specified content types.
Data	**Landmark** map `[DatabaseURI]`: unicode string `ID`: unicode string **Category** map `[DatabaseURI]`: unicode string `ID`: unicode string	NA	the Type Landmark/Category to delete and details of the particular Type. `ID` is a mandatory field for deleting a landmark and category object.

Table 6.98: Input parameters for Delete

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.99: Output parameters for Delete

Error code value	Description
`1002`	Bad argument type
`1003`	Missing argument
`1004`	Service not supported
`1006`	Service not ready
`1011`	Access denied

Table 6.100: Error codes

The following table lists the error messages and their description:

Error messages	Description
`Landmarks:Delete:Type or Data is missing`	Indicates Type is missing or data type of Type is mismatched.
`Landmarks:Delete:Type is invalid`	Indicates that Type is not a value in the given range.
`Landmarks:Delete:Data is missing`	Indicates Data is missing or data type of Data is mismatched.
`Landmarks:Delete:Id is missing`	Indicates `ID` is missing or data type of `ID` is mismatched.
`Landmarks:Delete:DatabaseURI is missing`	Indicates `DatabaseURI` is missing or data type of `DatabaseURI` is mismatched.

Table 6.101: Error messages

Example

The following sample code illustrates how to delete an object from the active / specified landmark database:

```
import scriptext

landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
try:
    getlist_output = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                       'Filter': {'LandmarkName': u'land1'}})
    getlist_error = getlist_output['ErrorCode']
    if getlist_error != 0:
        print "GetList error"
    else:
        retval = getlist_output['ReturnValue']
        id = retval['id']
        delete_output = landmark_handle.call('Delete',{'Type': u'Landmark',
                                                        'Data': {'id': unicode(id)}})
        delete_error = delete_output['ErrorCode']
        if delete_error != 0:
            print "Error in deleting landmark"
        else:
            print "Landmark deleted"
except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.5.5 Import

`Import` is used to import a set of Landmarks. It accepts a set of input parameters that define the Type and data for performing the operation. It is available only in synchronous mode.

The following is an example for using `Import`:

```
getlist_output = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                   'Filter': {'LandmarkName': u'land1'}})
```

The following table summarizes the specification of `Import`:

Input Parameters

Interface	`IDataSource`
Description	Imports a set of Landmarks.
Response Model	Synchronous
Pre-condition	`IDataSource` interface is loaded.
Post-condition	The default or active database is opened for reading landmarks and categories. A default database is created, if it does not exist and is set as active. The iterator points to the first item in the list of imported objects. Updates the Database with the list of imported landmarks.

Input parameter specifies the Type and Data of the particular landmark to import.

Name	Type	Range	Description
Type	unicode string	`Landmark`	Performs operation based on the specified content types.
Data	map `[DatabaseURI]`: unicode string `SourceFile`: unicode string `MimeType`: unicode string	NA	**DatabaseURI**: Import landmarks to the database. If this is not specified landmarks / categories is imported to the default database. **SourceFile**: Import landmarks from this file. **MimeType**: Encoding algorithm. You must specify the Mime type of the landmark content that must be parsed. Mime enables the inclusion of media other than plain text and the inclusion of several entities in one single message. Supported Mime types: **application/vnd.nokia.landmarkcollection+xml**.

Table 6.102: Input parameters for Import

Output Parameters

Output parameters contain `ReturnValue`. It also `ErrorCode`, and `ErrorMessage` if the operation fails. `ReturnValue` is an iterator to an array of Landmarks.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to import a set of landmarks:

Name	Type	Range (Type: string)	Description
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.
ReturnValue	Iterable To maps of imported Landmark.	map **Landmark**. For more information, refer Key Values 6.5.8 section.	Iterator to the list of imported landmarks. For map, see **Landmark** in the Key Values 6.5.8 section.

Table 6.103: Output parameters for Import

Error code value	Description
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1010	Entry exists
1011	Access denied
1012	Item not found
1013	Unknown format
1017	Path not found

Table 6.104: Error codes

Error messages	Description
Landmarks:Import:Type or Data is missing	Indicates Type is missing or data type of Type is mismatched.
Landmarks:Import:Type is invalid	Indicates that Type is not a value in the given range.
Landmarks:Import:Data is missing	Indicates Data is missing or data type of Data is mismatched.
Landmarks:Import:MimeType is missing	Indicates MimeType is missing or data type of MimeType is mismatched.
Landmarks:Import:SourceFile is missing	Indicates SourceFile is missing or data type of SourceFile is mismatched.

Table 6.105: Error messages

```
import scriptext

landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
try:
    import_output = landmark_handle.call('Import', {'Type': u'Landmark',
                                         'Data': {'SourceFile': u'c:\data\land_import.txt',
                                         'MimeType':
                                          u'application/vnd.nokia.landmarkcollection+xml'}})
    error = import_output['ErrorCode']
    if error != 0:
        print "Error in importing landmark"
    else:
        print "Landmark imported"

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.5.6 Export

`Export` is used to exports a specified set of Landmarks. It is available only in synchronous mode.

The following is an example for using `Export`:

```
getlist_output = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                   'Filter': {'LandmarkName': u'land1'}})
```

The following table summarizes the specification of `Export`:

Interface	`IDataSource`
Description	Imports a set of Landmarks.
Response Model	Synchronous
Pre-condition	`IDataSource` interface is loaded.
Post-condition	The default or active database is opened for reading landmarks and categories. A default database is created, if it does not exist and is set as active. Landmarks is exported to the specified file.

Input Parameters

Input parameter specifies the Type and Data for performing the operation.

Output Parameters

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Name	Type	Range	Description
Type	unicode string	`Landmark`	Performs operation based on the specified content types.
Data	map `[DatabaseURI]`: unicode string `DestinationFile`: unicode string `IdList`: List (Lmid1, Lmid2) `MimeType`: unicode string	NA	**DatabaseURI**: Export landmarks from this database. If this is not specified landmarks/categories is exported from default database. **DestinationFile**: Export landmarks to this file. Complete file path must be specified. **IdList**: List of landmark Ids. **MimeType**: Encoding algorithm. You must specify the Mime type of the landmark content. Mime enables the inclusion of media other than plain text and the inclusion of several entities in one single message. Supported Mime types: **application/vnd.nokia.landmarkcollection+xml**.

Table 6.106: Input parameters for Export

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.107: Output parameters for Export

Error code value	Description
`-301`	No Service
`1002`	Bad argument type
`1003`	Missing argument
`1004`	Service not supported
`1010`	Entry exists
`1017`	Path not found

Table 6.108: Error codes

Error messages	Description
`Landmarks:Export:Type or Data is missing`	Indicates Type is missing or data type of Type is mismatched.
`Landmarks:Export:Type is invalid`	Indicates that Type is not a value in the given range.
`Landmarks:Export:Data is missing`	Indicates Data is missing or data type of Data is mismatched.
`Landmarks:Export:MimeType is missing`	Indicates `MimeType` is missing or data type of `MimeType` is mismatched.
`Landmarks:Export:DestinationFile is missing`	Indicates `DestinationFile` is missing or data type of `DestinationFile` is mismatched.
`Landmarks:Export:IdList is missing`	Indicates `IdList` is missing or data type of `IdList` is mismatched.
`Landmarks:Export:IdList is empty`	Indicates `IdList` is empty.

Table 6.109: Error messages

Example

The following sample code illustrates how to export a set of landmarks:

```
import scriptext

landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
try:
    getlist_output = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                       'Filter': {'LandmarkName': u'land1'}})
    getlist_error = getlist_output['ErrorCode']
    if getlist_error != 0:
        print "GetList error"
    else:
        retval = getlist_output['ReturnValue']
        id_val = retval['id']
        export_output = landmark_handle.call('Export', {'Type':  u'Landmark',
                                                        'Data': {'DestinationFile':
                                                                   u'c:\data\export_land.txt
                                                                 'idList': [id_val],
                                                                 'MimeType':
                                                                   'application/vnd.nokia.la
        export_error = export_output['ErrorCode']
        if export_error != 0:
            print "Export unsuccessful"
        else:
            print "Landmark ecported"

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.5.7 Organise

`Organise` is used to associate or disassociate a list of landmarks in a database to a category. It accepts a set of parameters that defines the Type, data, and operation type for performing the operation. It is available only in synchronous mode.

The following is an example for using `Organise`:

```
org_output = landmark_handle.call('Organise', {'Type': u'Landmark',
                                                'Data': {'id': unicode(cat_id),
                                                'idList': [id_val1,id_val2]},
                                                'Operation Type': 'Associate'})
```

The following table summarizes the specification of `Organise`:

Interface	`IDataSource`
Description	Associates or disassociates a list of landmarks in a database to a category.
Response Model	Synchronous
Pre-condition	`IDataSource` interface is loaded.
Post-condition	The default or active database is opened for reading landmarks and categories. A default database is created, if it does not exist and is set as active. Landmarks is exported to the specified file.

Input Parameters

Input parameter specifies the type, data, and type of operation for performing the operation.

Name	Type	Range	Description
Type	unicode string	`Landmark`	Performs operation based on the specified content types.
Data	map `[DatabaseURI]`: unicode string `Id`: unicode string `IdList`: List (Id1, Id2)	NA	**DatabaseURI**: Organise landmarks in this database. **Id**: Associate or disassociate landmarks to the category `Id`. **IdList**: List of landmarks need to be organized.
`OperationType`	unicode string	**OperationType**: `Associate` `Disassociate`	NA

Table 6.110: Input parameters for Organise

Output Parameters

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.111: Output parameters for Organise

Errors

The following table lists the error codes and their values:

Error code value	Description
1002	Bad argument type
1003	Missing argument
1011	Access denied
1012	Item not found

Table 6.112: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
Landmarks:Organise:Type or Data or OperationType is missing	Indicates Type is missing or data type of Content Type is mismatched.
Landmarks:Organise:Type is invalid	Indicates that Type is not a value in the given range.
Landmarks:Organise:Data is missing	Indicates Data is missing or data type of Data is mismatched.
Landmarks:Organise:Id is missing	Indicates category Id is missing or data type of category Id is mismatched.
Landmarks:Organise:IdList is missing	Indicates IdList is missing or data type of IdList is mismatched.
Landmarks:Organise:IdList is empty	Indicates IdList is empty.
Landmarks:Organise:OperationType is missing	Indicates OperationType is missing or data type of OperationType is mismatched.
Landmarks:Organise:OperationType is invalid	Indicates OperationType is not a value in the given range.

Table 6.113: Error messages

Example

The following sample code illustrates how to associate or disassociate list of landmarks in a database to a category:

```
import scriptext

landmark_handle = scriptext.load('Service.Landmarks', 'IDataSource')
try:
    getlist_cat_output = landmark_handle.call('GetList', {'Type': u'Category'})
    retval_cat = getlist_cat_output['ReturnValue']
    cat_id = retval_cat['id']

    getlist_land1_output = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                            'Filter': {'LandmarkName': u'land1'}}
    retval1 = getlist_land1_output['ReturnValue']
    id_val1 = retval['id']

    getlist_land2_output = landmark_handle.call('GetList', {'Type': u'Landmark',
                                                            'Filter': {'LandmarkName': u'land2'}}
    retval2 = getlist_land2_output['ReturnValue']
    id_val2 = retval['id']

    org_output = landmark_handle.call('Organise', {'Type': u'Landmark',
                                                   'Data': {'id': unicode(cat_id),
                                                            'idList': [id_val1,id_val2]},
                                                   'Operation Type': 'Associate'})
    error = org_output['ErrorCode']
    if error != 0:
        print "Error in organising contacts"
    else:
        print "Conatcs organised"

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.5.8 Key Values

Landmark

Category

Database

Position Information of a Landmark

LandmarkPositionFields

Landmark Search Criteria

BoundedArea

Bounded area is the area enclosed at the intersection of the latitudes and longitudes as mentioned in the following table:

Category Search Criteria

Key	Type	Description
[LandmarkName]	string	Specifies a name for the landmark. Landmark name is not unique in a database. Maximum string Length is 255.
[id]	string	A unique identifier created in the database on addition of a new landmark. This field must not be specified when a new landmark is added to the database.
[CategoryInfo]	List of strings	List of Category IDs to which a landmark belongs.
[LandmarkDesc]	string	Description about the landmark. Maximum string Length is 4095.
[LandmarkPosition]	map	map describes latitude, longitude, altitude of a landmark. For more information, see **Position Information of a Landmark**.
[CoverageRadius]	Double	Radius from a position defined in landmark.
[IconFile]	string	Specifies Icon associated with landmark. Maximum string Length is 255.
[IconIndex]	int	Index of icon within the Icon file.
[IconMaskIndex]	int	Index of the icon mask within the Icon file.
[LandmarkFields]	map	This is a name-value pair. For more information, see **LandmarkPositionFields**.

Table 6.114: Key values- Landmark

Key	Type	Description
[CategoryName]	string	Specifies a name for the category. Category name is unique in a database. Maximum string Length is 124.
[id]	string	A unique identifier created in the database on addition of a new category to a database. This field must not be specified when a new landmark is added to the database.
[GlobalId]	string	Specifies global category ID. This field is a non-modifiable field. It is ignored if passed as input.
[IconFile]	string	Specifies Icon associated with landmark. Maximum string Length is 255.
[IconIndex]	int	Index of icon within the Icon file.
[IconMaskIndex]	int	Index of the icon mask within the Icon file.

Table 6.115: Key values- Category

Key	Type	Description
[DatabaseName]	string	Specifies a name for a database. Database name need not be unique.
DatabaseURI	string	Database file name defined in specific format: **¡protocol¿://¡filename¿**. For example: **file://c:landmark.ldb** [local database]. **protocol://location** [Remote database].
[DbDrive]	string	Specifies drive in which database is stored. For example, **C**.
[DbProtocol]	string	Specifies protocol by which database can be accessed.
[DbMedia]	int	0: MediaNotPresent 1: MediaUnknown 2: MediaFloppyDisk 3: MediaHardDisk 4: MediaCdRom 5: MediaRam 6: MediaFlash 7: MediaRom 8: MediaRemote 9: MediaNANDFlash 10: MediaRotatingMedia
[DbSize]	int	Specifies the size of the database in bytes.
[DbActive]	bool	Indicates if this database is opened by default (that is, device default database). **True**: Default database. **False**: Not a default database.

Table 6.116: Key values- Database

Key	Type	Description
Latitude	Double	Specifies latitude of a location in WGS-84 format. This needs to be specified in decimal degrees. Normal range of values [-90,+90]. Out of range values will be normalized to range as per standards. Negative degrees indicate west latitude and positive value indicated east latitude.
Longitude	Double	Specifies longitude of a location in WGS-84 format. This needs to be specified in decimal degrees. Normal range of values [-180 ,+180]. Out of range values will be normalized to range as per standards. Negative value indicates south longitude and positive value indicates north longitude.
[Altitude]	Double	Specifies altitude of a location in WGS-84 format, in meters.
[HAccuracy]	Double	Error estimate of horizontal accuracy to Latitude, Longitude, and Altitude in meters.
[VAccuracy]	Double	Error estimate of vertical accuracy to Latitude, Longitude, and Altitude in meters.

Table 6.117: Key values- Position Information of a Landmark

Key	Type	Description
[Street]	string	Address of the landmark. Maximum string Length is 255.
[BuildingName]	string	Address of the landmark. Maximum string Length is 255.
[District]	string	Address of the landmark. Maximum string Length is 255.
[City]	string	Address of the landmark. Maximum string Length is 255.
[AreaCode]	string	Address of the landmark. Maximum string Length is 255.
[Telephone]	string	Contact number. Maximum string Length is 255.
[Country]	string	Address of the landmark. Maximum string Length is 255.

Table 6.118: Key values- LandmarkPositionFields

Key	Type	Description
[DatabaseURI]	string	Search is performed on the specified database. If database is not specified then, search is performed on default database. Maximum string Length is 255.
[LandmarkName]	string	Text is case insensitive, wild cards supported- '?' for single character, '*' for zero or more characters. Maximum string Length is 255.
[LandmarkPosition]	map	Map describes latitude, longitude, altitude of a landmark. For more information, see **Position Information of a Landmark**. Only Latitude and Longitude fields are considered for search.
[CoverageRadiusOption]	bool	**True**: The coverage radius of landmarks is considered in the distance calculation. For example, if the circular search area and centre coordinates, which are mentioned in **LandmarkPosition** and radius mentioned in **MaximumDistance** intersects the landmark circular area, centre coordinates specified by the coordinates of the landmark and radius specified by CoverageRadius then, such landmark will be returned. **False**: The CoverageRadius of the landmark is not considered in the distance calculation. The default value is **False**.
[MaximumDistance]	Double	It is the distance from centre coordinate if CoverageRadius option is **False** else it is the effective distance calculated as landmark centre minus the coverage radius.
[CategoryName]	string	Search results only for landmarks that belong to this category. Maximum string Length is 124. If specified **False** then, only unlisted landmarks are listed.
[LandmarkDesc]	string	Text is case insensitive, wild cards supported- '?' for single character, '*' for zero or more characters. Maximum string Length is 4095.
[BoundedArea]	map	Area specified within NSEW latitudes and longitudes. For more information, see **Bounded area**.
[MaximumMatches]	int	The maximum number of items retrieved when provided with criteria information. If not mentioned then all landmarks are returned.
[PreviousMatchesOnly]	bool	You can request to search within previous search results only. **True**: Searches in previous search results. **False**: A new search will be carried out on database. If you do not specify this option then, **False** will be taken as default.

Table 6.119: Key values- Landmark Search Criteria

Key	Type	Description
NorthLatitude	Double	The northern-most latitude of the bounded area in WGS-84 format.
SouthLatitude	Double	The southern-most latitude of the bounded area in WGS-84 format.
EastLongitude	Double	The eastern longitude of the bounded area in WGS-84 format.
WestLongitude	Double	The western longitude of the bounded area in WGS-84 format.

Table 6.120: Key values- BoundedArea

Key	Type	Description
[DatabaseURI]	string	Search is performed on a specified database. If the database is not specified then, search is performed on default database. Maximum string Length is 255.
[CategoryName]	string	Text is case insensitive, wild cards supported- '?' for single character, '*' for zero or more characters. Maximum string Length is 124.
[MaximumMatches]	int	The maximum number of items retrieved when provided with criteria information. If not mentioned then all landmarks are returned.
[PreviousMatchesOnly]	bool	You can request to search within previous search results only. **True**: Searches in previous search results. **False**: A new search will be carried out on a database. If you do not specify this option then **False** will be taken as default.

Table 6.121: Key values- Category Search Criteria

6.6 Location

The Location Service enables Python applications to retrieve information on the physical location of an S60 device. It also enables to perform calculations based on location information.

For the location services to function in S60 device, the device must be location aware. It must include some location information provider, that is, a positioning system in the form of GPS, AGPS, or Bluetooth.

The following sample code is used to load the provider:

```
import scriptext
location_handle = scriptext.load('Service.Location', 'ILocation')
```

The following table summarizes the Location Interface:

Service provider	Service.Location
Supported interfaces	ILocation

The following table lists the services available in Application Manager:

Services	Description
GetList 6.6.1	Retrieves the current location of the user.
Trace 6.6.2	Informs the consumer of any change in current location.
CancelNotification 6.6.3	Cancels the registered listeners with the service provider.
MathOperations 6.6.4	Performs specific calculations on user provided data.

6.6.1 GetList

GetList is used to retrieve the current location of the device.

The following are the examples for using GetList:

Synchronous

```
GetList_Output = location_handle.call('GetList', {'LocationInformationClass': u'BasicLocatio
```

Asynchronous

```
event_id = location_handle.call('GetList', {'LocationInformationClass':  u'BasicLocationInfo
```

where, `get_list` is a user defined function.

The following table summarizes the specification of `GetList`:

Interface	`ILocation`
Description	Retrieves the current location of the device.
Response Model	Synchronous and asynchronous
Pre-condition	Device must be Location aware (that is, it must have some location information provider in form of GPS, AGPS, or Bluetooth). ILocation interface loaded.
Post-condition	Nil

Input Parameters

Input parameter specifies the category of location information and the update option used for retrieving location information.

Output Parameters

Output parameters contain the requested information. They also contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to retrieve a list with location information, in asynchronous mode:

Name	Type	Range	Description
[Location Information Class]	unicode string	Basic Location Information Generic Location Info	This specifies category of location information. You will receive detailed location estimations on specifying Generic Location Info. Default value for this argument is BasicLocationInformation. Refer to Updateoptions description to know more about what are the output that are guaranteed to be in the location estimates for each of the LocationInformationClass provided.
[Updateoptions]	map **Name**: **Type** [UpdateInterval] (Microseconds): int32 [UpdateTimeOut] (Microseconds): int32 [UpdateMaxAge] (Microseconds): int32 [PartialUpdates] (Microseconds): bool	NA	This specifies update option used while retrieving location estimation. Default values are used if no argument is specified as part of input argument list. UpdateInterval specifies the time interval between two consecutive location estimates. If location server is not able to give location estimates within specified UpdateTimedOut, you will receive SErrTimedOut error. UpdateMaxAge specifies the expiry time for the position information cache. It means that when a position request is made the position information can be returned from the cache, (Managed by location server) as long as the cache is not older that the specified maximum age. The default value is zero that is, the position information will never be returned from cache. Setting PartialUpdates to **FALSE** ensures that you will get at least BasicLocationInformation (Longitude, Latitude, and Altitude.) By default, following values (in seconds) are used for these input parameters. UpdateInterval = 1 UpdateTimeOut = 15 UpdateMaxAge = 0 PartialUpdates = FALSE **note:** In case the following order is not maintained when you supply value for updateoption, it returns the error SErrArgument. UpdateTimeOut¿UpdateInterval¿MaxAge

Table 6.122: Input parameters for GetList

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.
`ReturnValue`	For more information, refer table map: GetList 6.124	NA	It contains location estimations. In case you specify `BasicLocationInformation` in the input list only longitude, latitude and altitude will return. **note:** If `PartialUpdates` is set to **FALSE** you must get longitude, altitude and latitude. The WGS-84 datum is used to refer co-ordinates. Also representation is in decimal degree. In case generic information is requested, there is no guarantee that all information mentioned here will be obtained as it depends on the underlying GPS technology and other factor like number of satellites, which are available when location fix is obtained. **note:** Not all GPS technology are capable of retrieving all information listed here. For example, if you select network based positioning technology it does not have capability to retrieve satellites information. In situation where a particular field can not be retrieved from the underlying GPS technology, it will not be present in the output list mentioned here.

Table 6.123: Output parameters for GetList

Data	Type	Description
Longitude	Double	This is the longitudinal data. Degree value is in the range [+180, -180].
Latitude	Double	This is the latitudinal data. Degree value is in the range [+90, -90].
Altitude	Double	Altitude data, height in meters.
SatelliteNumView	Double	Number of field satellite currently in view.
SatelliteNumViewUsed	Double	Number of satellites used.
HorizontalSpeed	Double	Horizontal speed, value in meters per second.
HorizontalSpeedError	Double	Horizontal speed error, value in meters per second.
TrueCourse	Double	This is the information about the current direction in degrees to true north.
TrueCourseError	Double	This is the true course error in degrees.
MagneticHeading	Double	This is the current direction in degrees to magnetic north.
MagneticHeadingError	Double	True magnetic course error in Degrees.
Heading	Double	This is the current instantaneous direction of travel in degrees to the true north.
HeadingError	Double	Heading error, value in degrees.
MagneticCourse	Double	This is the information about the current direction in degrees to magnetic north.
MagneticCourseError	Double	Magneticcourser error.

Table 6.124: map: GetList

Error code value	Description
-302	No Interface
0	Success
1007	No memory
1009	Server busy
1011	Access denied
1016	Service timed-out

Table 6.125: Error codes

Error messages	Description
Location:GetList:Wrong category info should be BasicLocationInformation/ GenericLocationInfo	Indicates argument supplied for category information is wrong.
Location:GetList:BadArgument - Updateoptions	Indicates argument supplied for Updateoptions is wrong.
Location:GetList:Negative Time Interval	Indicates time interval supplied is negative.
Location:GetList:Updateoptions Type Mismatch	Indicates a wrongly supplied type for Updateoptions.

Table 6.126: Error messages

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def get_list(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The landmarks are"
        for i in input_params["ReturnValue"]:
            print "Longitude"
            print i["Longitude"]
            print "Latitude"
            print i['Latitude']
            print "Altitude"
            print i['Altitude']
            print "SatelliteNumView"
            print i['SatelliteNumView']
            print "SatelliteNumViewUsed"
            print i['SatelliteNumViewUsed']
            print "HorizontalSpeed"
            print i['HorizontalSpeed']

    lock.signal()

# Async Query a location with search criteria
location_handle = scriptext.load('Service.Location', 'ILocation')
event_id = location_handle.call('GetList', {'LocationInformationClass': u'BasicLocationInfor

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.6.2 Trace

`Trace` method is used to retrieve periodic updates on the current location of the device. You can use this method to track the movements of the device.

The following is an example for using `Trace`:

```
event_id = location_handle.call('Trace', {'LocationInformationClass': u'GenericLocationInfo'
```

The following table summarizes the specification of `Trace`:

Input Parameters

Input parameter specifies the type of device location information returned and how it is returned.

Output Parameters

Interface	`ILocation`
Description	Tracks the movements of the device.
Response Model	Asynchronous
Pre-condition	Device must be Location aware (that is, it must have some location service provider in form of GPS, AGPS, or Bluetooth). ILocation interface loaded. No other instance of `Trace` is presently pending or is in use.
Post-condition	Nil

Output parameters contain the requested information. They also contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to inform the consumer of any change in current location, in asynchronous mode:

Name	Type	Range	Description
[Location Information Class]	unicode string	Basic Location Information Generic Location Info	This specifies category of location information. You will receive detailed location estimations on specifying Generic Location Info. Default value for this argument is BasicLocationInformation. Refer to Updateoptions description to know more about what are the output that are guaranteed to be in the location estimates for each of the LocationInformationClass provided.
[Updateoptions]	map **Name**: **Type** [UpdateInterval] (Microseconds): int32 [UpdateTimeOut] (Microseconds): int32 [UpdateMaxAge] (Microseconds): int32 [PartialUpdates] (Microseconds): bool	NA	This specifies update option used while retrieving location estimation. Default values are used if no argument is specified as part of input argument list. UpdateInterval specifies the time interval between two consecutive location estimates. If location server is not able to give location estimates within specified UpdateTimedOut, you will receive SErrTimedOut error. UpdateMaxAge specifies the expiry time for the position information cache. It means that when a position request is made the position information can be returned from the cache, (Managed by location server) as long as the cache is not older that the specified maximum age. The default value is zero that is, the position information will never be returned from cache. Setting PartialUpdates to **FALSE** ensures that you will get at least BasicLocationInformation (Longitude, Latitude, and Altitude.) By default, following values (in seconds) are used for these input parameters. UpdateInterval = 1 UpdateTimeOut = 15 UpdateMaxAge = 0 PartialUpdates = FALSE **note:** In case the following order is not maintained when you supply [Updateoptions] it returns the error SErrArgument. UpdateTimeOut¿UpdateInterval¿MaxAge

Table 6.127: Input parameters for Trace

Name	Type	Range (Type: string)	Description
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.
ReturnValue	For more information, refer table map: Trace 6.129	NA	It contains location estimations. In case you specify BasicLocationInformation in the input list only longitude, latitude and altitude will return. **note:** If PartialUpdates is set to **FALSE** you must get longitude, altitude and latitude. The WGS-84 datum is used to refer co-ordinates. Also representation is in decimal degree. In case generic information is requested, there is no guarantee that all information mentioned here will be obtained as it depends on the underlying GPS technology and other factor like number of satellites, which are available when location fix is obtained. **note:** Not all GPS technology are capable of retrieving all information listed here. For example, if you select network based positioning technology it does not have capability to retrieve satellites information. In situation where a particular field can not be retrieved from the underlying GPS technology, it will not be present in the output list mentioned here.

Table 6.128: Output parameters for Trace

Data	Type	Description
Longitude	Double	This is the longitudinal data. Degree value is in the range [+180, -180].
Latitude	Double	This is the latitudinal data. Degree value is in the range [+90, -90].
Altitude	Double	Altitude data, height in meters.
SatelliteNumView	Double	Number of field satellite currently in view.
SatelliteNumViewUsed	Double	Number of satellites used.
HorizontalSpeed	Double	Horizontal speed, value in meters per second.
HorizontalSpeedError	Double	Horizontal speed error, value in meters per second.
TrueCourse	Double	This is the information about the current direction in degrees to true north.
TrueCourseError	Double	This is the true course error in degrees.
MagneticHeading	Double	This is the current direction in degrees to magnetic north.
MagneticHeadingError	Double	True magnetic course error in Degrees.
Heading	Double	This is the current instantaneous direction of travel in degrees to the true north.
HeadingError	Double	Heading error, value in degrees.
MagneticCourse	Double	This is the information about the current direction in degrees to magnetic north.
MagneticCourseError	Double	Magneticcourser error.

Table 6.129: map: Trace

Error code value	Description
0	Success
1011	Access denied
1012	Item not found

Table 6.130: Error codes

Error messages	Description
Location:Trace:Invalid LocationInformationClass	Indicates argument supplied for category information is wrong.
Location:Trace:Updateoptions Type Mismatch	Indicates wrong type for Updateoptions.
Location:Trace:Badargument - updateoptions	Indicates wrongly supplied updateoptions.
Location:Trace:Negative Time Interval	Indicates wrongly supplied time interval as part of Updateoptions.

Table 6.131: Error messages

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def Trace(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message:" + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "The location change are as "
        for i in input_params["ReturnValue"]:
            print "Longitude"
            print i["Longitude"]
            print "Latitude"
            print i['Latitude']
            print "Altitude"
            print i['Altitude']
            print "SatelliteNumView"
            print i['SatelliteNumView']
            print "SatelliteNumViewUsed"
            print i['SatelliteNumViewUsed']
            print "HorizontalSpeed"
            print i['HorizontalSpeed']

    lock.signal()

# Async Query a location with search criteria
location_handle = scriptext.load('Service.Location', 'ILocation')
event_id = location_handle.call('Trace', {'LocationInformationClass': u'GenericLocationInfo'

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.6.3 CancelNotification

`CancelNotification` method is used to cancel an outstanding asynchronous call.

The following is an example for using `CancelNotification`:

```
cancel_output = location_handle.call('CancelNotification', {'CancelRequestType': u'GetLocCan
```

The following table summarizes the specification of `CancelNotification`:

Input Parameters

The parameters specify whether to cancel a `GetList` call or a `Trace` call. The object must contain the `CancelRequestType` property (unicode string) that is used to specify the type of call to cancel.

Output Parameters

Interface	`ILocation`
Description	Cancels the registered listeners with the service provider.
Response Model	Synchronous
Pre-condition	Device must be Location aware (that is, it must have some location service provider in form of GPS, AGPS, or Bluetooth). ILocation interface loaded.
Post-condition	Nil

Name	Type	Range	Description
`CancelRequestType`	unicode string	`TraceCancel` `GetLocCancel`	Contains specific information about the type of notification expected to be canceled.

Table 6.132: Input parameters for CancelNotification

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.133: Output parameters for CancelNotification

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to cancel the registered listeners with the service provider:

```
import scriptext
location_handle = scriptext.load('Service.location', 'ILocation')

try:
    cancel_output = location_handle.call('CancelNotification', {'CancelRequestType':  u'GetL
    errorcode = cancel_output["ErrorCode"]
    if errorcode != 0:
        print "Error in cancelling the request"
    else:
        ret_val = cancel_output["ReturnValue"]

        print "The cancellation request is successful"

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

Error code value	Description
1000	Invalid service argument
1012	Item not found

Table 6.134: Error codes

Error messages	Description
Location:Cancel:BadArgument - cancel type	Indicates error in supplied cancel type.
Location:Cancel:Missing cancel type	Indicates missing cancel type in input.
Location:Cancel:Wrong cancel type	Indicates cancel type supplied is wrong.

Table 6.135: Error messages

6.6.4 MathOperations

MathOperations performs mathematical calculations based on a source location and a target location.

The following is an example for using MathOperations:

```
Distance_measured = location_handle.call('MathOperations', {'MathRequest': u'FindDistance',
```

The following table summarizes the specification of MathOperations:

Interface	ILocation
Description	Performs mathematical calculations based on a source location and a target location.
Response Model	Synchronous
Pre-condition	Device must be Location aware (that is, it must have some location service provider in form of GPS, AGPS, or Bluetooth). ILocation interface loaded.
Post-condition	Nil

Input Parameters

Input parameter specifies the mathematical operation such as FindDistance, FindBearingTo and so on, and position co-ordinates for performing the mathematical operation.

Output Parameters

Output parameter contains ReturnValue. It also contains ErrorCode, and ErrorMessage if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Name	Type	Range	Description
MathRequest	string	FindDistance FindBearingTo MoveCoordinates	Specifies the mathematical operation.
DistanceParamSource	map. For more information, refer table map-DistanceParamSource 6.137	NA	This specifies the position co-ordinates. Note that expected datum here is WGS-84 with decimal degree representation. Also note that altitude supplied does not effect the result of calculation. It is used to maintain a uniform input argument, which makes it easy to use.
DistanceParamDestination	map. For more information, refer table map-DistanceParamDestination 6.138	NA	Specifies co-ordinates of another position. It is not required when value specified in the first parameter is MoveCoordinates. Note that expected datum here is WGS-84 with decimal degree representation.
MoveByThisDistance (only if MathRequestType is MoveCoordinates)	double	NA	Move source position by the specified the distance.
(Only if MathRequestType is MoveCoordinates)	double	NA	Move the source position by the specified bearing.

Table 6.136: Input parameters for MathOperations

Key	Type	Description
Longitude	double	Longitude data
Latitude	double	Latitude data
Altitude	double	Altitude data

Table 6.137: map- DistanceParamSource

Key	Type	Description
Longitude	double	Longitude data
Latitude	double	Latitude data
Altitude	double	Altitude data

Table 6.138: map- DistanceParamDestination

Name	Type	Range (Type: string)	Description
ReturnValue	The table 6.140 describes output obtained for various input combination	NA	Resultant calculation. In case you request to Move coordinates, map described in column 2 will be returned. Note that if distance between two coordinate is requested, it is returned in meters while FindBearingTo returned is in degrees counting clockwise relative to true north.
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.

Table 6.139: Output parameters for MathOperations

MathRequest type in input	Obtained output type	Description
FindDistance	double	Contains the calculated distance in meters.
FindBearingTo	double	Bearing between two points.
MoveCoordinates map	Map described in the table 6.141 is returned, which represents the translated coordinate.	

Table 6.140: map- Resultant output

Key	Type	Description
Longitude	double	Longitude data
Latitude	double	Latitude data
Altitude	double	Altitude data

Table 6.141: map- MoveCoordinates

Error code value	Description
1002	Bad argument type

Table 6.142: Error codes

Error messages	Description
Location:MathOperations:Missing argument- MathRequest	Indicates missing Mathrequest argument.
Location:MathOperations:Wrong argument- MathRequest	Indicates supplied MathRequest argument is wrong.
Location:MathOperations:Missing argument- locationcoordinate	Indicates missing locationCoordinate in input.
Location:MathOperations:Missing argument- MoveByThisDistance	Indicates missing MoveByThisDistance in input.
Location:MathOperations:Missing argument- MoveByThisBearing	Indicates missing MoveByThisBearing in input.
Location:MathOperations:TypeMismatch- MoveByThisDistance	Indicates type for Movebydistance is wrong.
Location:MathOperations:TypeMismatch- MoveByThisBearing	Indicates type for Movebythisbearing is wrong.

Table 6.143: Error messages

Example

The following sample code illustrates how to perform specific calculations on user provided data:

```
import scriptext
location_handle = scriptext.load('Service.location', 'ILocation')

try:
    Distance_measured = location_handle.call('MathOperations', {'MathRequest': u'FindDistanc
    errorcode = Distance_measured["ErrorCode"]
    if errorcode != 0:
        print "Error in retrieving the Distance covered"
    else:
        ret_val = Distance_measured["ReturnValue"]
        if ret_val["distance covered"]["Value"] == '50':
            print "The distance covered is  retrieved"

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.7 Logging

The Logging service allows Python applications to integrate logging functionality of S60 device. It is used to add, read, and delete logging events such as call logs, messaging logs, data logs, and so on in the device.

It also provides a simple interface to the application developer to add, read, and delete events occurring in the device.

The following sample code is used to load the provider:

```
import scriptext
msg_handle = scriptext.load('Service.Logging', 'IDataSource')
```

The following table summarizes the Logging Interface:

Service provider	`Service.Logging`
Supported interfaces	`IDataSource`

The following table lists the services available in Logging:

Services	Description
`Add` 6.7.1	Adds a new event to the event log.
`Getlist` 6.7.2	Retrieves an event from the event log as specified by the input filter.
`Delete` 6.7.3	Deletes an event from the Event Log.
`RequestNotification` 6.7.4	Requests for notification for the updates occurring in the log.

6.7.1 Add

`Add` is used to add a new event to the event log. It takes a set of input parameters that define the Type and properties of the event log to add.

The following is an example for using `Add`:

```
log_id = logging_handle.call('Add',{'Type': u'Log', 'Item': {'EventType': 3, 'Direction': 1,
```

The following table summarizes the specification of `Add`:

Interface	`IDataSource`
Description	Adds a new event to the event log.
Response Model	Synchronous and asynchronous
Pre-condition	IDataSource interface is loaded.
Post-condition	A new event is added to the event log.

Input Parameters

Input parameter specifies the Type, and the properties of the event log. Input parameter properties are Type and Item.

Output Parameters

Name	Type	Range	Description
Type	unicode string	`Log`	Specifies the Content Type.
Item	map For more information, refer map 6.145	**EventType** (Indicates the type of log Event) `EKLogCallEventType`: 0 `EKLogDataEventType`: 1 `EKLogFaxEventType`: 2 `EKLogShortMessageEventType`: 3 `EKLogPacketDataEventType`: 4 **Direction** `EIncomingEvent`: 0 `EOutgoingEvent`: 1 `EIncomingEvent Alternateline`: 2 `EOutgoingEvent Alternateline`: 3 `EFetchedEvent`: 4 `EMissedEvent`: 5 `EMissedEvent Alternateline`: 6 **DeliveryStatus** `EStatusPending`: 0 `EStatusSent`: 1 `EStatusFalied`: 2 `EStatusNone`: 3 `EStatusDone`: 4 `EStatusNotSent`: 5 `EStatusScheduled`: 6 **LogFlags** `EKLogEventContactSearched`: 0 `EKLogEventRead`: 1 **note:** Flag `EKLogEventRead` is set when you read the event in the log database. Flag `EKLogEventContactSearched` is set when you search through contact database for performing any operation, like voice call or sms.	Adds the given event to the event log.

Table 6.144: Input parameters for Add

Key	Value	Description
EventType	32 bit int	Unique Id to identify event type.
[RemoteParty]	unicode string	Describes the destination of the out going event or source of incoming event. If the length of the specified string is greater than 64 characters, then the data is truncated.
[Direction]	32 bit int	The direction of a call means incoming, outgoing, and so on.
[EventDuration]	32 bit int	Time in seconds.
[DeliveryStatus]	32 bit int	Delivered, pending, or failed.
[Subject]	unicode string	Describes the subject for the event. If the length of the specified string is greater than 64 characters, then the data is truncated.
[PhoneNumber]	unicode string	The phone number is associated with the event. This is used when the number cannot be stored in any other field. If the length of the specified string is greater than 100 characters, then the number is truncated.
[EventData]	8-bit Data	Specific data associated with the event.
[Link]	32 bit int	Link is used to relate this event to an entity in other application. For example, it can be used to associate the event with the call ID or the message ID for emails and faxes.
[LogFlags]	32 bit int	Sets the specified flags for this event. The function does not change any of the other flag bit settings.
[RepeatDates]	List of dates	NA

Table 6.145: map

Output contains `ReturnValue`. It also contains `ErrorCode` and an `Error Message` if the operation fails. `ReturnValue` contains the identifier in the event log corresponding to the user specified input parameters.

Name	Type	Range	Description
`ErrorCode`	32 bit int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.
`ReturnValue`	string	Unique identifier for the particular event in the log.	

Table 6.146: Output parameters for Add

Errors

The following table lists the error codes and their values:

Error code value	Description
`1000`	Invalid service argument
`1002`	Bad argument type
`1003`	Missing argument
`1004`	Service not supported

Table 6.147: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`Logging:Add:TypeInvalid`	Invalid Type is passed to `contenttype` parameter.
`Logging:Add:TypeMissing`	Content Type missing in the inputparam list.
`Logging:Add:ItemInvalid`	Invalid Type is passed to Item parameter.
`Logging:Add:ItemMissing`	Item map missing in the inputparam list.
`Logging:Add:EventTypeInvalid`	Invalid Type is passed to `EventType` parameter.
`Logging:Add:EventTypeMissing`	`EventType` field is missing in the Item map.
`Logging:Add:RemotePartyInvalid`	Invalid Type is passed to `RemoteParty` parameter.
`Logging:Add:EventDurationInvalid`	Invalid Type is passed to `EventDuration` parameter.
`Logging:Add:DeliveryStatusInvalid`	Invalid Type is passed to `DeliveryStatus` parameter.
`Logging:Add:SubjectInvalid`	Invalid Type is passed to `Subject` parameter.
`Logging:Add:PhoneNumberInvalid`	Invalid Type is passed to `PhoneNumber` parameter.
`Logging:Add:EventDataInvalid`	Invalid Type is passed to `EventData` parameter.
`Logging:Add:LinkInvalid`	Invalid Type is passed to `Link` parameter.
`Logging:Add:LogFlagsInvalid`	Invalid Type is passed to `LogFlags` parameter.
`Logging:Add:DirectionInvalid`	Invalid Type is passed to `Direction` parameter.

Table 6.148: Error messages

Example

The following sample code illustrates how to add a log entry:

```
import scriptext
logging_handle = scriptext.load('Service.Logging', 'IDataSource')
log_id = logging_handle.call('Add', {'Type': u'Log', 'Item': {'EventType': 3, 'Direction': 1
```

6.7.2 GetList

`GetList` retrieves an event from the event log as specified by the input filter. It takes a set of input parameters that define Type and Filter for retrieving information.

The following are the examples for using `GetList`:

Synchronous

```
logging_info = logging_handle.call('GetList', {'Type': u'Log',})
```

Asynchronous

```
logging_handle.call('GetList', {'Type': u'Log',}, callback=logging_callback)
```

where, `logging_callback` is the callback function.

The following table summarizes the specification of `GetList`:

Interface	`IDataSource`
Description	Gets the specified event details from event log.
Response Model	Synchronous and asynchronous
Pre-condition	IDataSource interface is loaded.
Post-condition	Nil

Input Parameters

Input parameter specifies the Type and Filter.

Output Parameters

Output contains `ReturnValue`. It also contains `ErrorCode` and an `ErrorMessage` if the operation fails. `ReturnValue` contains an iterable wrapper of entries containing all relevant fields of the user specified Item.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to get the logging details on the phone:

Name	Type	Range	Description
Type	unicode string	`Log`	Specifies the Content Type.
[Filter]	map For more information, refer map 6.150	**RecentList** `EKLogNullRecentList`: -1 `EKLogRecentIncomingCalls`: 1 `EKLogRecentOutgoingCalls`: 2 `EKLogRecentMissedCalls`: 3 A value of `EKLogNullRecentList` indicates to include events from all of the recent event lists. **EventType** `EKLogCallEventType`: 0 `EKLogDataEventType`: 1 `EKLogFaxEventType`: 2 `EKLogShortMessageEventType`: 3 `EKLogPacketDataEventType`: 4 **Direction** `EIncomingEvent`: 0 `EOutgoingEvent`: 1 `EIncomingEvent Alternateline`: 2 `EOutgoingEvent Alternateline`: 3 `EFetchedEvent`: 4 `EMissedEvent`: 5 `EMissedEvent Alternateline`: 6 **DeliveryStatus** `EStatusPending`: 0 `EStatusSent`: 1 `EStatusFalied`: 2 `EStatusNone`: 3 `EStatusDone`: 4 `EStatusNotSent`: 5 `EStatusScheduled`: 6 **LogFlags** `EKLogEventContactSearched`: 0 `EKLogEventRead`: 1 **note:** Flag `EKLogEventRead` is set when you read the event in the log database. Flag `EKLogEventContactSearched` is set when you search through contact database for performing any operation, like voice call or sms.	This is an optional parameter. Filter contains the search criteria for specific event search in the log database. When an empty filter is passed, it gets all the events from the database by default.

Table 6.149: Input parameters for GetList

Key	Value	Description
`Id`	unicode string	Unique Id for the particular event in the log. **note:** To specify values for other fields in the map are not valid, when the Id field is used for querying. If specified, then the values are ignored.
`RecentList`	32 bit int	Gets the 20 recent call list. Initialises or refreshes the view for the specified recent event list, conforming to the specified filter. On successful completion, the view is positioned at the first that is, most recent event in the recent event list. Recent lists also have a mechanism for duplicate event handling, set up via the recent list configuration. for example, two calls to the same number will only take up one entry on the outgoing calls list. **note:** To specify values for other fields in the map are not valid, when the `RecentList` field is used for querying. If specified then, the values are ignored.
`EventType`	32 bit int	Unique Id to identify event type.
`[PhoneNumber]`	unicode string	The phone number is associated with the event. This is used when the number cannot be stored in any other field. If the length of the specified string is greater than 100 characters, then the number is truncated.
`[RemoteParty]`	unicode string	Describes the destination of the out going event or source of incoming event. If the length of the specified string is greater than 64 characters, then the data is truncated.
`[Direction]`	32 bit int	The direction of a call means incoming, outgoing, and so on.
`[DeliveryStatus]`	32 bit int	Delivered, pending, or failed.
`[EndTime]`	datetime	Sets the specified EndTime to be used by the filter.
`[LogFlags]`	32 bit int	Sets the specified flags for this event.

Table 6.150: map

Name	Type	Range	Description
ReturnValue	**Iterator** (map) EventType: 32 bit int RemoteParty: String Direction: 32 bit int EventTime: datetime EventDuration: 32 bit int DeliveryStatus: 32 bit int Subject: string PhoneNumber: string Description: string EventData: 8-bit Data Link: 32 bit int id: string LogFlags: 32 bit int	NA	The output is an iterable list of entries, which on each invocation returns a map containing all relevant fields of an Item. **note:** Description and EventTime are automatically set by the system for a created event. For example, The Description for CallEvent is **voice call**, SMS Event is **short message**. EventTime is the time at which the event is created. This should be a datetime object.
ErrorMessage	string	NA	Error Description in Engineering English.
ErrorCode	32 bit int	NA	Contains the SAPI specific error code when the operation fails.

Table 6.151: Output parameters for GetList

Error code value	Description
0	Success
1002	Bad argument type
1003	Missing argument
1004	Service not supported

Table 6.152: Error codes

Error messages	Description
Logging:GetList:TypeInvalid	Invalid Type is passed to contenttype parameter.
Logging:GetList:TypeMissing	Content Type missing in the inputparam list.
Logging:GetList:FilterInvalid	Invalid Type is passed to Filter parameter.
Logging:GetList:IdInvalid	Invalid Type is passed to Id parameter.
Logging:GetList:RecentListInvalid	Invalid Type is passed to RecentList parameter.
Logging:GetList:PhoneNumberInvalid	Invalid Type is passed to PhoneNumber parameter.
Logging:GetList:DirectionInvalid	Invalid Type is passed to Direction parameter.
Logging:GetList:DeliveryStatusInvalid	Invalid Type is passed to DeliveryStatus parameter.
Logging:GetList:LogFlagsInvalid	Invalid Type is passed to LogFlags parameter.
Logging:GetList:EndTimeInvalid	Invalid Type is passed to EndTime parameter.
Logging:GetList:RemotePartyInvalid	Invalid Type is passed to RemoteParty parameter.
Logging:GetList:EventTypeInvalid	Invalid Type is passed to EventType parameter.

Table 6.153: Error messages

```
import scriptext
# Load the desired SAPI
logging_handle = scriptext.load('Service.Logging', 'IDataSource')
try:
    logging_info = logging_handle.call('GetList', {'Type': u'Log',})
    for item in logging_info:
        print item['EventType']
        print item['RemoteParty']
        print item['Direction']
        print item['EventTime']
        print item['Subject']
        print item['PhoneNumber']
        print item['Description']
        print item['EventData']
except scriptext.ScriptextError, err:
        print "Error getting the list of Installed Application : ", err
```

6.7.3 Delete

`Delete` is used to delete a specified event from the event Log.

The following is an example for using `Delete`:

```
logging_handle.call('Delete', {'Type': u'Log', 'Data': {'id': log_id,}})
```

The following table summarizes the specification of `Delete`:

Interface	`IDataSource`
Description	Deletes events in the event log.
Response Model	Synchronous and asynchronous
Pre-condition	IDataSource interface is loaded.
Post-condition	Nil

Input Parameters

Input parameter specifies the Type and its Id for performing the delete operation.

Name	Type	Range	Description
Type	unicode string	`Log`	Specifies the Content Type.
Data	map `Id`: unicode string	NA	Deletes the event specified by the Id.

Table 6.154: Input parameters for Delete

Output Parameters

Output contains `ErrorCode` and `ErrorMessage`, if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

Name	Type	Range	Description
ErrorMessage	string	NA	Error Description in Engineering English.
ErrorCode	32 bit int	NA	Contains the SAPI specific error code when the operation fails.

Table 6.155: Output parameters for Delete

Error code value	Description
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1012	Item not found

Table 6.156: Error codes

The following table lists the error messages and their description:

Error messages	Description
Logging:Delete:TypeInvalid	Invalid Type is passed to contenttype parameter.
Logging:Delete:TypeMissing	Content Type missing in the inputparam list.
Logging:Delete:DataInvalid	Invalid Type is passed to Data parameter.
Logging:Delete:DataMissing	Data map missing in inputparam list.
Logging:Delete:idInvalid	Invalid Type is passed to Id parameter.
Logging:Delete:idMissing	Id field is missing in the Data map.

Table 6.157: Error messages

Example

The following sample code illustrates how to delete an entry from the log:

```
import scriptext

logging_handle = scriptext.load('Service.Logging','IDataSource')
logging_handle.call('Delete', {'Type': u'Log', 'Data': {'id': log_id,}})
```

6.7.4 RequestNotification

RequestNotification is used to request notification for the updates occurring to log and register for any changes happening to event log.
It is used in asynchronous mode only.

The following is an example for using RequestNotification:

```
logging_id = logging_handle.call('RequestNotification', {'Type': u'Log', 'Filter': {'DelayTi
```

where, logging_app_callback is user defined function.

The following table summarizes the specification of RequestNotification:

Input Parameters

Input parameter specifies the Type and delay time in microseconds, which elapses before the notification request

Interface	`IDataSource`
Description	Registers for the changes occurring to the event log.
Response Model	Asynchronous
Pre-condition	IDataSource interface is loaded.
Post-condition	Nil

can complete.

Name	Type	Range	Description
Type	unicode string	`Log`	Specifies the Content Type.
Filter	map `DelayTime`: 32 bit int	NA	The minimum time in microseconds, which elapses before the notification request can complete.

Table 6.158: Input parameters for RequestNotification

Output Parameters

Output contains `ErrorCode` and `ErrorMessage`, if the operation fails.

Name	Type	Range	Description
`ErrorMessage`	string	NA	Error Description in Engineering English.
`ErrorCode`	32 bit int	NA	Contains the SAPI specific error code when the operation fails.

Table 6.159: Output parameters for RequestNotification

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to register changes in the event log, in asynchronous mode:

Error code value	Description
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1012	Item not found

Table 6.160: Error codes

Error messages	Description
Logging:RequestNotification:TypeInvalid	Invalid Type is passed to contenttype parameter.
Logging:RequestNotification:TypeMissing	Content Type missing in the inputparam list.
Logging:RequestNotification:FilterMissing	Filter map missing in inputparam list.
Logging:RequestNotification:FilterInvalid	Invalid Type is passed to Filter parameter.
Logging:RequestNotification:DelayTimeMissing	DelayTime is missing in the Filter Map.
Logging:RequestNotification:DelayTimeInvalid	Invalid Type is passed to DelayTime parameter.

Table 6.161: Error messages

```
import scriptext
import e32
lock = e32.Ao_lock()

def logging_app_callback(trans_id, event_id, input_params):
    if trans_id != logging_id and event_id != scriptext.EventCompleted:
        print "Error in servicing the request"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message is: " + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "Changes in the Log Event Notified "

    lock.signal()

# Load appmanage service
# Load the desired SAPI
logging_handle = scriptext.load('Service.Logging', 'IDataSource')
logging_id = logging_handle.call('RequestNotification',
    {'Type': u'Log', 'Filter': {'DelayTime': 600000}},
        callback=logging_app_callback)

print "Waiting for the request to be processed!"
lock.wait()

print "Request complete!"
```

6.8 Messaging

The Messaging service enables Python applications to integrate messaging services of S60 device. It is used either to retrieve message information or use the messaging services, or both.

Using the Messaging service, you can access/iterate inbox, send messages (SMS/MMS), register for new message notification, status changes of messages and delete messages.

The following sample code is used to load the provider:

```
import scriptext
messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')
```

The following table summarizes the Application Manager Interface:

Service provider	`Service.Messaging`
Supported interfaces	`IMessaging`

The following table lists the services available in Application Manager:

Services	Description
`GetList` 6.8.1	Retrieves list of messaging objects from messaging center based on the search/sort inputs.
`Send` 6.8.2	Sends message.
`RegisterNotification` 6.8.3	Registers for new message notification.
`CancelNotification` 6.8.4	Cancels notification for incoming messages.
`ChangeStatus` 6.8.5	Changes status for the message.
`Delete` 6.8.6	Deletes message.

6.8.1 GetList

`GetList` is used to retrieve a list of messaging objects from messaging center based on the search / sort inputs. Each object contains messaging information that is, data and metadata about a single message. It is available only in synchronous mode.

The following is an example for using `GetList`:

```
sms_iter = messaging_handle.call('GetList', {'Type': u'Inbox'})
```

The following table summarizes the specification of `GetList`:

Interface	`IMessaging`
Description	Retrieves an iterable message list based on the search / sort inputs.
Response Model	Synchronous
Pre-condition	Valid instance of `IMessaging` interface is instantiated.
Post-condition	Nil

Input Parameters

Input parameter specifies the folder from which the messages are retrieved, also the Filter criteria and sort order for the returned list. Input parameter has three properties: Type, Filter and SortOrder.

Name	Type	Range	Description
Type	unicode string	`Inbox`	Performs operation based on the specified content types.
[Filter]	map `[MessageTypeList]`: List of unicode strings `[MessageId]`: 32 bit int `[SenderList]`: List of unicode strings `[Subject]`: unicode string `[StartDate]`: Date `[EndDate]`: Date	`MessageTypeList`: SMS MMS	It specifies the search information. If `StartDate` alone is specified, all messaging from the data will be returned and if `EndDate` alone is specified, all messages before the end date will be returned. And if both are specified then all the messages within the two bounds will be returned. An exception will be raised if `EndDate` is earlier than `StartDate`.

[SortOrder]	map
	Key: unicode string
	Order: unicode string
		Key:
		`Date`
		`Size`
		`Sender`
		`Subject`
		`MessageId`
		
		Order:
		Ascending
		Descending
			Sort Information. If not specified sorting is done on `Date` in ascending order.

Table 6.162: Input parameters for Getlist

Output Parameters

Output parameters contain the requested information. They also contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to iterate through inbox and print the SMS 'Sender' IDs:

```
import scriptext

messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')
# This 'GetList' request returns all the SMS in the inbox as an iterable map
sms_iter = messaging_handle.call('GetList', {'Type': u'Inbox'})
sender_list = []
for sms_dict in sms_iter:
    if sms_dict['MessageType'] == 'SMS':
sender_list.append(sms_dict['Sender'])
print "ID list :", sender_list
```

Name	Type	Range (Type: string)	Description
ReturnValue	Iterable List (of maps) map: MessageType: string Sender: string Subject: string Time: Time Priority: string Attachment: bool Unread: bool MessageId: 32 bit int BodyText: string To: List of strings Cc: List of strings Bcc: List of strings AttachmentList: List of map AttachmentList contains a list of Map (Attachment): AttachmentMap: FileName: string FileHandle: FileBuffer MimeType: string FileSize: int	MessageTypeList: SMS MMS Unknown Priority: Low Medium High	An iterable list of the resultant messages. Current implementation recognizes only SMS and MMS, other types of messages are unknown. SMS does not support subject, so it returns first few characters of the body text. Priority is applicable for Email type of messages. For SMS and MMS, it gives default value set by underlying messaging server. **Note:** Cc and Bcc fields are not applicable for SMS. Also, in case MMS has body text in it ; it appears as attachment. So the output value for body text field in case of MMS will be empty.
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.

Table 6.163: Output parameters for GetList

Error code value	Description
0	Success
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1007	No memory

Table 6.164: Error codes

Error messages	Description
`Messaging:GetList: Type Type Invalid`	Specifies if the type of `Type` parameter is invalid.
`Messaging:GetList:Type Value Incorrect`	Specifies if the value of `MessageType` parameter is incorrect.
`Messaging:GetList:Type Missing`	Specifies if the `MessageType` parameter is missing.
`Messaging:GetList:Filter Type Invalid`	Specifies if the type of Filter parameter is invalid.
`Messaging:GetList:SenderList Type Invalid`	Specifies if the type of `SenderList` parameter is invalid.
`Messaging:GetList:SenderList Element Value Incorrect`	Specifies if the value of element of `SenderList` parameter is incorrect.
`Messaging:GetList:SenderList Element Type Invalid`	Specifies if the type of element of `SenderList` parameter is invalid.
`Messaging:GetList:MessageTypeList Type Invalid`	Specifies if the type of `MessageTypeList` parameter is invalid.
`Messaging:GetList:MessageTypeList Element Value Incorrect`	Specifies if the value of element of `MessageTypeList` parameter is incorrect.
`Messaging:GetList:MessageTypeList Element Type Invalid`	Specifies if the type of element of MessageTypeList parameter is invalid.
`Messaging:GetList:MessageId Type Invalid`	Specifies if the type of `MessageId` parameter is invalid.
`Messaging:GetList:Subject Type Invalid`	Specifies if the type of `Subject` parameter is invalid.
`Messaging:GetList:StartDate Type Invalid`	Specifies if the type of `StartDate` parameter is invalid.
`Messaging:GetList:StartDate Value Incorrect`	Specifies if the value of `StartDate` parameter is incorrect.
`Messaging:GetList:EndDate Type Invalid`	Specifies if the type of `EndDate` parameter is invalid.
`Messaging:GetList:EndDate Value Incorrect`	Specifies if the value of `EndDate` parameter is incorrect.
`Messaging:GetList:SortOrder Type Invalid`	Specifies if the type of `SortOrder` parameter is invalid.
`Messaging:GetList:SortOrder Value Incorrect`	Specifies if the value of `SortOrder` parameter is incorrect.
`Messaging:GetList:Key Type Invalid`	Specifies if the type of `Key` parameter is invalid.
`Messaging:GetList:Order Type Invalid`	Specifies if the type of `Order` parameter is invalid.
`Messaging:GetList:Asynchronous Operation not supported`	Specifies if `GetList` is called asynchronously.

Table 6.165: Error messages

6.8.2 Send

`Send` is used to send an SMS or MMS message. It takes a set of input parameters that specifies the message type, and the message details associated with that particular message type.

The following are the examples for using `Send`:

Synchronous

```
messaging_handle.call('Send', {'MessageType': u'SMS', 'To': u'12345678',
                               'BodyText': u'Hi'})
```

Asynchronous

```
messaging_handle.call('Send', {'MessageType': u'SMS', 'To': u'12345678', 'BodyText': u'Hi'},
                      callback=callback_function)
```

where, `callback_function` is a user defined callback function.

The following table summarizes the specification of `Send`:

Interface	`IMessaging`
Description	Sends the message.
Response Model	Synchronous and asynchronous
Pre-condition	Valid instance of `IMessaging` interface is instantiated.
Post-condition	Nil

Input Parameters

Input parameter specifies the type of messaging object and its details.

Output Parameters

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to send SMS message to the specified phone number:

Name	Type	Range	Description
MessageType	unicode string	SMS MMS	This specifies the message type.
To	unicode string	NA	Multiple recipients can be passed using `MessageParam`.
[BodyText]	unicode string	NA	Body text for the message.
[Subject]	unicode string	NA	Message subject. Not applicable for SMS.
[Attachment]	unicode string	`FileName` with complete path.	Attachment Name (Full path). Valid for MMS only. Additional attachments can be passed using `MessageParam`.
[MimeType]	unicode string	NA	Mime type of the attachment mentioned above.
[MessageParam]	map [To]: List (unicode string) [Cc]: List (unicode string) [Bcc]: List (unicode string) [AttachmentList]: List (map) [TemplateId]: 32 bit int [LaunchEditor]: bool `AttachmentList` map elements contains: `FileName`: unicode string `[MimeType]`: unicode string	`FileName`: `FileName` with complete path. `MimeType` example: image/gif, image/jpeg and so on. `MimeType` is searched in system for the given file name. It is used if found or, user provided `MimeType` is taken.	This parameter specifies the full details of the message depending on its type. It adds the body text as text attachment in case of MMS. Template Id is the message id, which is the template for new message. `Bcc` is supported for email only. (Currently email is not supported in the service) In case of template id, if the body text of message is specified and exists for the given template Id then both the body text specified will be appended to the template Id message body text and sent. If Launch Editor Flag is set to **ETrue** then, the Message Editor will be popped up over the application expecting you to act, by default it is **EFalse**.

Table 6.166: Input parameters for Send

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.167: Output parameters for Send

Error code value	Description
`0`	Success
`1000`	Invalid service argument
`1002`	Bad argument type
`1003`	Missing argument
`1004`	Service not supported
`1007`	No memory
`1012`	Item Not found

Table 6.168: Error codes

Error messages	Description
Messaging:Send:MessageType Type Invalid	Specifies if the type of MessageType parameter is invalid.
Messaging:Send:MessageType Value Incorrect	Specifies if the value of MessageType parameter is incorrect.
Messaging:Send:MessageType Missing	Specifies if the MessageType parameter is missing.
Messaging:Send:To Type Invalid	Specifies if the type of To parameter is invalid.
Messaging:Send:To Value Incorrect	Specifies if the value of To parameter is incorrect.
Messaging:Send:To Missing	Specifies if the To parameter is missing.
Messaging:Send:BodyText Type Invalid	Specifies if the type of BodyText parameter is invalid.
Messaging:Send:Subject Type Invalid	Specifies if the type of Subject parameter is invalid.
Messaging:Send:Attachment Type Invalid	Specifies if the type of Attachment parameter is invalid.
Messaging:Send:Attachment Value Incorrect	Specifies if the value of Attachment parameter is incorrect.
Messaging:Send:MimeType Type Invalid	Specifies if the type of MimeType parameter is invalid.
Messaging:Send:MimeType Value Incorrect	Specifies if the value of MimeType parameter is incorrect.
Messaging:Send:MessageParam Type Invalid	Specifies if the type of MessageParam parameter is invalid.
Messaging:Send:TemplateId Type Invalid	Specifies if the type of TemplateId parameter is invalid.
Messaging:Send:LaunchEditor Type Invalid	Specifies if the type of LaunchEditor parameter is invalid.
Messaging:Send:To Type Invalid	Specifies if the type of To parameter is invalid.
Messaging:Send:To List Element Type Invalid	Specifies if the type of element of To List parameter is invalid.
Messaging:Send:To List Element Value Incorrect	Specifies if the value of element of To List parameter is incorrect.
Messaging:Send:Cc Type Invalid	Specifies if the type of Cc parameter is invalid.
Messaging:Send:Cc List Element Type Invalid	Specifies if the type of element of Cc List parameter is invalid.
Messaging:Send:Cc List Element Value Incorrect	Specifies if the value of element of Cc List parameter is incorrect.
Messaging:Send:Bcc Type Invalid	Specifies if the type of Bcc parameter is invalid.
Messaging:Send:Bcc List Element Type Invalid	Specifies if the type of element of Bcc List parameter is invalid.
Messaging:Send:Bcc List Element Value Incorrect	Specifies if the value of element of Bcc List parameter is incorrect.
Messaging:Send:AttachmentList Type Invalid	Specifies if the type of AttachmentList parameter is invalid.
Messaging:Send:AttachmentList Element Type Invalid	Specifies if the type of element of AttachmentList parameter is invalid.
Messaging:Send:FileName Type Invalid	Specifies if the type of FileName parameter is invalid.
Messaging:Send:FileName Value Incorrect	Specifies if the value FileName parameter is incorrect.
Messaging:Send:MimeType Type Invalid	Specifies if the type of MimeType parameter is invalid.
Messaging:Send:MimeType Value Incorrect	Specifies if the value MimeType parameter is incorrect.

Table 6.169: Error messages

```
import scriptext

messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')

try:
    messaging_handle.call('Send', {'MessageType': u'SMS', 'To': u'12345678',
                                   'BodyText': u'Hi'})
except scriptext.ScriptextError, err:
    print "Error sending SMS : ", err
else:
    print "SMS sent successfully"
```

6.8.3 RegisterNotification

`RegisterNotification` method registers the widget to receive notifications of new incoming messages. For each new message, the method returns the header information of that message. It is available only in asynchronous mode.

The following is an example for using `RegisterNotification`:

Asynchronous

```
sms_id = messaging_handle.call('RegisterNotification', {'Type': u'NewMessage'},
                               callback=new_sms_callback)
```

where, `new_sms_callback` is a user defined callback function.

The following table summarizes the specification of `RegisterNotification`:

Interface	`IMessaging`
Description	Registers for getting notification for new messages.
Response Model	Asynchronous
Pre-condition	Valid instance of `IMessaging` interface is instantiated.
Post-condition	Nil

Input Parameters

Input parameter specifies the request for notification of new messages. The object must contain the `NotificationType` property (unicode string), and this property must contain the value `NewMessage`.

Name	Type	Range	Description
Type	unicode string	`NewMessage`	Performs operation based on the specified content types.

Table 6.170: Input parameters for RegisterNotification

Output Parameters

Output parameters contain the requested information. They also contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Name	Type	Range (MessageType: string)	Description
ReturnValue	map: MessageType: string Sender: string Subject: string Time: Time Priority: string Attachment: bool Unread: bool MessageId: 32 bit int	MessageType: SMS MMS Unknown Priority: Low Medium High	It contains the list of message header fields. SMS does not support subject, so it returns first few characters of body text.
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.

Table 6.171: Output parameters for RegisterNotification

Error code value	Description
0	Success
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1010	Entry exists

Table 6.172: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
Messaging:RegisterNotification:Type Type Invalid	Specifies if the type of Type parameter is invalid.
Messaging:RegisterNotification:Type Value Incorrect	Specifies if the value of Type parameter is incorrect.
Messaging:RegisterNotification:Type Missing	Specifies if the Type parameter is missing.
Messaging:RegisterNotification:Synchronous Operation not supported	Specifies if RegisterNotification is called synchronously.

Table 6.173: Error messages

Example

The following sample code illustrates how to register for a new message notification, send a SMS and then cancel the notification request, asynchronously:

```
import scriptext
import e32

lock = e32.Ao_lock()
messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')

def new_sms_callback(trans_id, event_id, output_params):
    if trans_id == sms_id and event_id == scriptext.EventCompleted:
        print "SMS received from" + output_params['ReturnValue']['Sender'])
    else:
        print "Error in callback"
    # Cancel notification request
    messaging_handle.call('CancelNotification', {'Type': u'NewMessage'})
    lock.signal()

# The callback 'new_sms_callback' will be called when a sms is received
sms_id = messaging_handle.call('RegisterNotification', {'Type': u'NewMessage'},
                               callback=new_sms_callback)

# Send SMS to self so that the notification callback is hit
messaging_handle.call('Send', {'MessageType': u'SMS', 'To': u'12345678',
                               'BodyText': u'Hi self'})
lock.wait()
```

6.8.4 CancelNotification

`CancelNotification` method cancels the registration for notification of new messages. It is available only in synchronous mode.

The following is an example for using `CancelNotification`:

Synchronous

```
messaging_handle.call('CancelNotification',{'Type': u'NewMessage'})
```

The following table summarizes the specification of `CancelNotification`:

Interface	`IMessaging`
Description	Cancels registration for notification of new messages.
Response Model	Synchronous
Pre-condition	Valid instance of `IMessaging` interface is instantiated.
Post-condition	Stop getting new message notifications.

Input Parameters

Input parameter specifies the request for canceling notification of new messages. This must contain the Notification Type, and this property must contain the value `NewMessage`.

Name	Type	Range	Description
Type	unicode string	`NewMessage`	Performs operation based on the specified content types.

Table 6.174: Input parameters for CancelNotification

Output Parameters

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Name	Type	Range (Type: string)	Description
`ErrorCode`	int	NA	Service specific error code on failure of the operation.
`ErrorMessage`	string	NA	Error description in Engineering English.

Table 6.175: Output parameters for CancelNotification

Errors

The following table lists the error codes and their values:

Error code value	Description
`0`	Success
`1000`	Invalid service argument
`1002`	Bad argument type
`1003`	Missing argument

Table 6.176: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`Messaging:CancelNotification:Type Type Invalid`	Specifies if the type of `Type` parameter is invalid.
`Messaging:CancelNotification:Type Value Incorrect`	Specifies if the value of `Type` parameter is incorrect.
`Messaging:CancelNotification:Type Missing`	Specifies if the `Type` parameter is missing.
`Messaging:CancelNotification:Asynchronous Operation not supported`	Specifies if `CancelNotification` is called asynchronously.

Table 6.177: Error messages

Example

The following sample code illustrates how to cancel a notification:

```
import scriptext
import e32

lock = e32.Ao_lock()
messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')

def new_sms_callback(trans_id, event_id, output_params):
    if trans_id == sms_id and event_id == scriptext.EventCompleted:
        print "SMS received from" + output_params['ReturnValue']['Sender'])
    else:
        print "Error in callback"
    # Cancel notification request
    messaging_handle.call('CancelNotification', {'Type': u'NewMessage'})
    lock.signal()

# The callback 'new_sms_callback' will be called when a sms is received
sms_id = messaging_handle.call('RegisterNotification', {'Type': u'NewMessage'},
                               callback=new_sms_callback)

# Send SMS to self so that the notification callback is hit
messaging_handle.call('Send', {'MessageType': u'SMS', 'To': u'12345678',
                               'BodyText': u'Hi self'})
lock.wait()
```

6.8.5 ChangeStatus

`ChangeStatus` method changes the read status of a message. The status can be `Read`, `Unread`, `Replied`, or `Forwarded`. It is available only in synchronous mode.

The following is an example for using `ChangeStatus`:

Synchronous

```
messaging_handle.call('ChangeStatus', {'MessageId': message_id, 'Status': u'Unread'})
```

The following table summarizes the specification of `ChangeStatus`:

Interface	`IMessaging`
Description	Sets a given value for the given flag.
Response Model	Synchronous
Pre-condition	Valid instance of `IMessaging` interface is instantiated.
Post-condition	Message status changed to new status.

Input Parameters

Input parameter specifies the message ID, and message status to be set.

Output Parameters

Output parameters contain `ErrorCode`, and `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Name	Type	Range	Description
MessageId	32 bit int	NA	Message Id
Status	unicode string	Read Unread Replied Forwarded	Message status to be set. Replied and Forwarded are applicable for email type of messages.

Table 6.178: Input parameters for ChangeStatus

Name	Type	Range (Type: string)	Description
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.

Table 6.179: Output parameters for ChangeStatus

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to set SMS status as Unread:

```
import scriptext
import appuifw

messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')

sms_iter = messaging_handle.call('GetList', {'Type': u'Inbox'})
id_list = []
body_list = []
for sms_dict in sms_iter:
    if sms_dict['MessageType'] == 'SMS':
        id_list.append(sms_dict['MessageId'])
        body_list.append(sms_dict['BodyText'])

message_index = appuifw.selection_list(body_list)
try:
    messaging_handle.call('ChangeStatus', {'MessageId': id_list[message_index],
                                           'Status': u'Unread'})
except scriptext.ScriptextError, err:
    print "Error setting message status to Unread"
else:
    print "Message status changed to Unread"
```

Error code value	Description
0	Success
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1012	Item Not found

Table 6.180: Error codes

Error messages	Description
Messaging:ChangeStatus:MessageId Type Invalid	Specifies if the type of MessageId parameter is invalid.
Messaging:ChangeStatus:MessageId Value Incorrect	Specifies if the value of MessageId parameter is incorrect.
Messaging:ChangeStatus:MessageId Missing	Specifies if the MessageId parameter is missing.
Messaging:ChangeStatus:Status Type Invalid	Specifies if the type of Status parameter is incorrect.
Messaging:ChangeStatus:Status Value Incorrect	Specifies if the range of Status parameter is exceeded.
Messaging:ChangeStatus:Status Missing	Specifies if the Status parameter is missing.
Messaging:ChangeStatus:Asynchronous Operation not supported	Specifies if ChangeStatus is called asynchronously.

Table 6.181: Error messages

6.8.6 Delete

Delete method is used to delete a message. It is available only in synchronous mode.

The following is an example for using Delete:

Synchronous

```
messaging_handle.call('Delete', {'MessageId': message_id})
```

The following table summarizes the specification of Delete:

Interface	IMessaging
Description	Delete deletes the message.
Response Model	Synchronous
Pre-condition	Valid instance of IMessaging interface is instantiated.
Post-condition	Message no more exists in database.

Input Parameters

Input parameter specifies the MessageId of the message to delete.

Name	Type	Range	Description
MessageId	32 bit int	MessageId	Deletes message with the specified message ID.

Table 6.182: Input parameters for Delete

Output Parameters

Output parameters contain ErrorCode, and ErrorMessage if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

Name	Type	Range (Type: string)	Description
ErrorCode	int	NA	Service specific error code on failure of the operation.
ErrorMessage	string	NA	Error description in Engineering English.

Table 6.183: Output parameters for Delete

Error code value	Description
0	Success
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1012	Item Not found

Table 6.184: Error codes

The following table lists the error messages and their description:

Error messages	Description
Messaging:Delete:MessageId Type Invalid	Specifies if the type of MessageId parameter is mismatched.
Messaging:Delete:MessageId Value Incorrect	Specifies if the value of MessageId parameter is negative.
Messaging:Delete:MessageId Missing	Specifies if the MessageId parameter is missing.
Messaging:Delete:Asynchronous Operation not supported	Specifies if Delete is called asynchronously.

Table 6.185: Error messages

Example

The following sample code illustrates how to delete a particular SMS:

```
import scriptext
import appuifw
import e32

lock = e32.Ao_lock()
sms_iter = None
messaging_handle = scriptext.load('Service.Messaging', 'IMessaging')

sms_iter = messaging_handle.call('GetList', {'Type': u'Inbox'})

id_list = []
body_list = []
for sms_dict in sms_iter:
    if sms_dict['MessageType'] == 'SMS':
        id_list.append(sms_dict['MessageId'])
        body_list.append(sms_dict['BodyText'])

# Select the message to be deleted
message_index = appuifw.selection_list(body_list)
try:
    messaging_handle.call('Delete', {'MessageId': id_list[message_index]})
except scriptext.ScriptextError, err:
    print "Error deleting SMS :", err
else:
    print "Message deleted successfully"
```

6.9 Media Management

The Media Management service allows Python applications to retrieve information from the media files stored in the media gallery of an S60 device.

It is used to access information about different types of media including music, sounds, images, video, and streaming.

You can create applications like custom photo viewer or audio player that displays or, otherwise incorporate media, using the Media Management service.

The following sample code is used to load the provider:

```
import scriptext
msg_handle = scriptext.load('Service.MediaManagement', 'IDataSource')
```

The following table summarizes the Media Management Interface:

Service provider	`Service.MediaManagement`
Supported interfaces	`IDataSource`

The following table lists the services available in Media Management:

Services	Description
`Getlist` 6.9.1	Retrieves information from a given service or data source on S60 device.

6.9.1 GetList

`GetList` takes a set of input parameters that define filter and sort criteria, and retrieves the metadata of media files based on media and metadata type.

`GetList` implements the main functionality of Media Management service. It is available only in asynchronous mode.

The following is an example for using `GetList`:

```
media_handle.call('GetList'[{'Type': string, 'Filter': map, 'Sort': map},
                            callback=callback_function])
```

where, `callback_function` is user defined function.

The following table summarizes the specification of `GetList`:

Interface	`IDataSource`
Description	Retrieves the metadata of media files based on media and metadata type.
Response Model	Asynchronous
Pre-condition	Valid service object representing the provider and interface.
Post-condition	Nil

Input Parameters

Input parameter specifies the Type, the metadata of media file to fetch and the criteria for sorting. Input parameter has three properties: Type, Filter, and Sort.

Name	Type	Range	Description
Type	unicode string	`Fileinfo`	Operation performed on the specified type. This field is mandatory.
Filter	map	For more information, refer table 6.187	It specifies the type of media file to fetch, and key for filtering the media files with their range. `FileType` field is mandatory. If key is specified, then it is mandatory to specify the range. You must only mention the StartRange for keys, where EndRange is not applicable. For example, if key is `FileName` then, mention the desired file name in the StartRange and leaving the EndRange empty.
[Sort]	map	For more information, refer table 6.188	It specifies the key name on which the resulting output will be sorted and can be one of the values mentioned in the **Value** column. By default, sorting is done in ascending order based on the `FileName`.

Table 6.186: Input parameters for GetList

Output Parameters

Output contains `ReturnValue`. It also contains `ErrorCode` and an `ErrorMessage` if the operation fails.

Errors

The following table lists the error codes and their values:

Error Messages

The following table lists the error messages and their description:

Example

The following sample code illustrates how to get a list of all MP3s:

Key	Value	Type
FileType	Music Sound Image Video StreamingURL	unicode string
[Key]	FileName FileExtension Drive FileSize FileDate MimeType FileNameAndPath SongName Artist Album Genre TrackNumber Composer LinkFirstURL	unicode string
[StartRange]	Valid for all keys	unicode string
[EndRange]	Valid for the following keys: FileSize(bytes) FileDate(YYYYMMDD:HHMMSS) For example, 20070303:010101	unicode string

Table 6.187: Media file type

Key	Value	Type
[Key]	FileName FileExtension Drive FileSize FileDate MimeType FileNameAndPath SongName Artist Album Genre TrackNumber Composer LinkFirstURL	unicode string
[Order]	Ascending or descending	unicode string

Table 6.188: Key name

Name	Type	Range	Description
ErrorCode	int	NA	Contains the SAPI specific error code when the operation fails.
ErrorMessage	string	NA	Error Description in Engineering English.
ReturnValue	Iterable maps	Type: string FileName: string FileExtension: string Drive: string FileSize: int FileDate: datetime MediaType: int FileNameAndPath: string SongName: string Artist: string Album: string Genre: string TrackNumber: string Composer: string MimeType: string LinkFirstURL: string	The output is an iterable which on each invocation returns a map, which will be filled by the service provider. Map stores the key names and its values. The key-value pair that is, Property name and Value in the output map depends upon the file type in the input -Filter map.

Table 6.189: Output parameters for GetList

Error code value	Description
1002	Bad argument type
1003	Missing argument

Table 6.190: Error codes

Error messages	Description
MediaMgmt:GetList:Server busy	Indicates provider is busy in processing another request.
MediaMgmt:GetList:Type Missing	Indicates Type parameter is missing.
MediaMgmt:GetList:Type not supported(should be FileInfo)	Indicates that the content type is incorrect.
MediaMgmt:GetList:Filter parameter missing	Indicates that the Filter parameter which is mandatory is missing.
MediaMgmt:GetList:Filter parameter type mismatch	Indicates that the type of Filter parameter is incorrect.
MediaMgmt:GetList:Sort parameter type mismatch	Indicates that the type of Sort parameter is incorrect.
MediaMgmt:GetList:FileType missing in Filter map	Indicates that FileType parameter is not present in Filter map or, FileType parameter type is incorrect.

Table 6.191: Error messages

```
import scriptext
import e32

def media_callback(trans_id, event_id, output_params):
    if trans_id == media_trans_id:
        if event_id == scriptext.EventCompleted:
            song_list = []
            for item in output_params['ReturnValue']:
                song_list.append(item['FileName'])
            print "List of files retrieved:", song_list
        else:
            print "Event ID was not EventCompleted"
    else:
        print "Invalid Transaction ID"
    lock.signal()

lock = e32.Ao_lock()
media_handle = scriptext.load('Service.MediaManagement', 'IDataSource')

# Request for the list of mp3s in ascending order
media_trans_id = media_handle.call('GetList', {'Type': u'FileInfo',
                                               'Filter': {'FileType': u'Music',
                                                          'Key': u'FileExtension',
                                                          'StartRange': u'.mp3'},
                                               'Sort': {'Key': u'FileName',
                                                        'Order': u'Ascending'}},
                                    callback=media_callback)

lock.wait()
```

6.9.2 Key Values

File Types

Key	Description	Supported Metadata
`Music`	Retrieves media files of Music type.	Artist, SongName, TrackNumber, Album, Genre, Composer, FileNameAndPath, FileName, FileExtension, Drive, MimeType, FileSize, FileDate.
`Sound`	Retrieves media files of Sound type.	FileNameAndPath, FileName, FileExtension, Drive, MimeType, FileSize, FileDate.
`Image`	Retrieves media files of Image type.	FileNameAndPath, FileName, FileExtension, Drive, MimeType, FileSize, FileDate.
`Video`	Retrieves media files of Video type.	FileNameAndPath, FileName, FileExtension, Drive, MimeType, FileSize, FileDate.
`StreamingUrl`	Retrieves media files of Link type.	LinkFirstURL, FileNameAndPath, FileName, FileExtension, Drive, MimeType, FileSize ,FileDate.

Keys

Output maps for various values of FileType

Key	Description
Type	Always a media file.
FileName	Filter/sort the result as per file name.
FileExtension	Filter/sort the result based on file extension.
Drive	Filter/sort the result as per file drive.
FileSize	Filter/sort the result as per file size.
FileData	Filter/sort the result as per file date.
MediaType	Filter/sort the result as per file type. **Value**: **Description** 0: Unknown media type 1: Music media type 2: Sound media type 3: Image media type 4: Video media type 5: Streaming URLs
FileNameAndPath	Filter/sort the result as per full path of file.
SongName	Filter/sort the result as per song name.
Artist	Filter/sort the result as per artist name.
Album	Filter/sort the result as per album name.
MimeType	Filter/Sort the result based on mime type.

6.10 Sensors

The Sensor service enables access to the various methods provided by S60 sensor channel subsystem. provides abstraction of various physical sensors that exist in the device. You can map data from one physical sensor to several channels. These include the following:

- Finding available sensor channels.
- Registering to receive notification on data from various sensors.
- Getting channel properties.

The following sample code is used to load the provider:

```
import scriptext
sensor_handle = scriptext.load('Service.Sensor', 'ISensor')
```

The following table summarizes the Sensor Interface:

The following table lists the services available in Sensor:

6.10.1 FindSensorChannel

`FindSensorChannel` performs a search operation for sensor channels in a S60 device based on the specified search criteria.

The client application specifies the search parameters and queries to the Sensor services, which returns a list containing channel information matching the search parameters.

The following is an example for using `FindSensorChannel`:

```
sensor_handle.call('FindSensorChannel', {'SearchCriterion': u'Orientation'})
```

FileType	Output map	
`Image`	**Key**:	**Value**
	Type:	string
	`FileName`:	string
	`FileExtension`:	string
	`Drive`:	string
	`FileSize`:	int
	`FileDate`:	datetime
	`FileNameAndPath`:	string
	`MimeType`: string	
`Sound`	**Key**:	**Value**
	Type:	string
	`FileName`:	string
	`FileExtension`:	string
	`Drive`:	string
	`FileSize`:	int
	`FileDate`:	datetime
	`FileNameAndPath`:	string
	`MimeType`: string	
`Video`	**Key**:	**Value**
	Type:	string
	`FileName`:	string
	`FileExtension`:	string
	`Drive`:	string
	`FileSize`:	int
	`FileDate`:	datetime
	`FileNameAndPath`:	string
	`MimeType`: string	
`Music`	**Key**:	**Value**
	Type:	string
	`FileName`:	string
	`FileExtension`:	string
	`Drive`:	string
	`FileSize`:	int
	`FileDate`:	datetime
	`MimeType`:	string
	`FileNameAndPath`:	string
	`SongName`:	string
	`Artist`:	string
	`Album`:	string
	`TrackNumber`:	string
	`Genre`:	string
	`Composer`: string	
`StreamingUrl`	**Key**:	**Value**
	Type:	string
	`FileName`:	string
	`FileExtension`:	string
	`Drive`:	string
	`FileSize`:	int
	`FileDate`:	datetime
	`FileNameAndPath`:	string
	`LinkFirstURL`:	string
	`MimeType`: string	

Service provider	`Service.Sensor`
Supported interfaces	`ISensor`

Services	**Description**
`FindSensorChannel` 6.10.1	Searches for sensor channels in the device based on a given search criteria.
`RegisterForNotification` 6.10.2	Registers for notification with a sensor channel to receive channel data.
`GetChannelProperty` 6.10.3	Gets the channel property of the specified sensor channel.

The following table summarizes the specification of `FindSensorChannel`:

Interface	`ISensor`
Description	Performs a search operation for sensor channels in an S60 device based on the specified search criteria.
Response Model	Synchronous
Pre-condition	ISensor interface is loaded.
Post-condition	Receives list of sensor channel which can be used to open channels.

Input Parameters

Input parameter is a string that specifies the search criteria for performing the search operation.

Name	Type	Range	Description

`SearchCriterion`	unicode string	All
		`AccelerometerAxis`
		`AccelerometerDoubleTapping`
		Orientation
		Rotation
			Specifies the search criterion.
			
			You can select from the list provided and specify it as an argument.

Table 6.192: Input parameters for FindSensorChannel

Output Parameters

Add API which misses out on some mandatory input Output contains `ReturnValue`. It also contains `ErrorCode` and an `ErrorMessage` if the operation fails. `ReturnValue` is an array of objects, which contains the sensor channel information requested by `FindSensorChannel`.

Name	Type	Range	Description
`ErrorCode`	32 bit int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.
`ReturnValue`	Lists of maps. Each map in this document is referred as **ChannelInfoMap**. For more information, refer table **ChannelInfoMap** 6.194	**ContextType** 0: Not defined 1: Ambient sensor. For example, to measure pressure. 2: Gives information on device itself. 3: Measures user initiated stimulus. **Quantity** 0: Not defined 10: Acceleration 11: Tapping 12: Orientation 13: Rotation 14: Magnetic 15: Tilt	`ReturnValue` consists of a list of maps, each map of which holds the key-value pair for each of sensor channel that satisfy the search criterion.

Table 6.193: Output parameters for FindSensorChannel

Type	Name	Description
32 bit int	`ChannelId`	Unique ID representing the channel.
32 bit int	`ContextType`	Defines the context where the channel is available.
32 bit int	`Quantity`	Defines the quantity being sensed.
32 bit int	`ChannelType`	Defines a unique type ID for each channel.
string	`Location`	Location of the sensor related to channel.
string	`VendorId`	Vendor ID of the sensor related to channel.
32 bit int	`DataItemSize`	Data item size delivered in the channel.
32 bit int	`ChannelDataTypeId`	Unique data type identifier for the data being sensed.

Table 6.194: ChannelInfoMap

Errors

The following table lists the error codes and their values:

Error code value	Description
`0`	Success
`1000`	Invalid service argument
`1002`	Bad argument type
`1003`	Missing argument

Table 6.195: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`Sensors:FindSensorChannel: Search Criterion Missing`	Indicates that channel search criterion is missing from the input parameter list.
`Sensors:FindSensorChannel: Invalid Search Criterion`	Indicates that the channel search criterion is invalid and does not fall within the specified range of search criterion strings.
`Sensors:FindSensorChannel: Channel search param type invalid`	Indicates that the datatype of the parameter passed for the channel search criterion is invalid.

Table 6.196: Error messages

Example

The following sample code illustrates how to query a list of channel information matching the search parameters:

```
try:
    result = sensor_handle.call('FindSensorChannel', {'SearchCriterion': u'Orientation'})
    count_items = len(result)
    print count_items
    print "ChannelId : ", result[0]['ChannelId']
    print "ContextType : ", result[0]['ContextType']
    print "Quantity : ", result[0]['Quantity']
    print "ChannelType : ", result[0]['ChannelType']
    print "Location : ", result[0]['Location']
    print "VendorId : ", result[0]['VendorId']
    print "DataItemSize : ", result[0]['DataItemSize']
    print "ChannelDataTypeId : ", result[0]['ChannelDataTypeId']

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.10.2 RegisterForNotification

`RegisterForNotification` is used to register for notification with a sensor channel to receive channel data or channel property changes. This is associated with the transaction ID of an asynchronous request. These notifications are continuous in nature and are stopped by invoking the `Cancel` command on the retrieved transaction ID.

The following is an example for using `RegisterForNotification`:

Asynchronous

```
sensor_handle.call('RegisterForNotification',
                   {'ListeningType': u'ChannelData',
                    'ChannelInfoMap': {'ChannelId': result['ChannelId'],
                                       'ContextType': result['ContextType'],
                                       'Quantity': result['Quantity'],
                                       'ChannelType': result['ChannelType'],
                                       'Location': result['Location'],
                                       'VendorId': result['VendorId'],
                                       'DataItemSize': result['DataItemSize'],
                                       'ChannelDataTypeId': result['ChannelDataTypeId']}},
                    callback=sensor_callback)
```

where, `sensor_callback` is the user defined callback function.

The following table summarizes the specification of `RegisterForNotification`:

Interface	`ISensor`
Description	Registers for notification with a sensor channel to receive channel data.
Response Model	Asynchronous
Pre-condition	ISensor interface is loaded.
Post-condition	Client application receives an array of channel information to open channels.

Input Parameters

Input parameter is a set of arguments that specifies the `Listening` Type and the `ChannelInfoMap`.

Name	Type	Range	Description
`ListeningType`	unicode string	Range for `ListeningType` `ChannelData`: Data listening	Determines the type of notification that needs to be registered for.
`ChannelInfoMap`	map as mentioned in `FindSensorChannel`. Refer **ChannelInfoMap** 6.194	NA	The map is obtained by invoking `FindSensorChannel`.

Table 6.197: Input parameters for RegisterForNotification

Output Parameters

Output contains `ReturnValue`. It also contains `ErrorCode` and an `ErrorMessage` if the operation fails. `ReturnValue` is an object, which contains output parameter details depending on the listening type and channel selected.

Name	Type	Range	Description
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.

`ReturnValue`	The output consists of one of the following maps depending on the listening type and channel selected:
	
	For listening type - `ChannelData` and channel information corresponding to `AccelerometerAxis`:
	Type: **Name**
	string: `DataType`
	Time: `TimeStamp`
	32 bit int: `XAxisData`
	32 bit int: `YAxisData`
	32 bit int: `ZAxisData`
	
	For listening type - `ChannelData` and channel information corresponding to `AccelerometerDoubleTapping`:
	Type: **Name**
	string: `DataType`
	Time: `TimeStamp`
	32 bit int: `DeviceDirection`
	
	For listening type - `ChannelData` and channel information corresponding to `Orientation`:
	Type: **Name**
	string: `DataType`
	Time: `TimeStamp`
	string: `DeviceOrientation`
	
	For listening type - `ChannelData` and channel info corresponding to `Rotation`:
	Type: **Name**
	string: `DataType`
	Time: `TimeStamp`
	32 bit int: `XRotation`
	32 bit int: `YRotation`
	32 bit int: `ZRotation`
		`DataType` for `AccelerometerAxis` is `AxisData`
		
		`DataType` for `AccelerometerDoubleTapping` is `DoubleTappingData`
		
		`DataType` for `Orientation` is `OrientationData`
		Range for `DeviceOrientation`:
		Undefined
		DisplayUp
		DisplayDown
		DisplayLeftUp
		DisplayRightUp
		DisplayUpwards
		DisplayDownwards
		
		`DataType` for `Rotation` is `RotationData`
			A map is returned in case notification is received.

Table 6.198: Output parameters for RegisterForNotification

Errors

The following table lists the error codes and their values:

Error code value	Description
0	Success
1000	Invalid service argument
1002	Bad argument type
1003	Missing argument
1005	Service in use

Table 6.199: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`Sensors:RegisterForNotification: Listening type missing`	Indicates that the listening type for receiving notification is missing.
`Sensors:RegisterForNotification: Listening type is invalid`	Indicates that the datatype of Listening type is invalid.
`Sensors:RegisterForNotification: ChannelInfoMap missing`	Indicates that the channel information map is not provided as input parameter.
`Sensors:RegisterForNotification: Incomplete input param list`	Indicates that the input parameter list is incomplete.
`Sensors:RegisterForNotification: Listening type is out of allowed range`	Indicates that the Listening type falls outside the allowed range of listening types.
`Sensors:RegisterForNotification: Callback missing`	Indicates that the callback function is missing.
`Sensors:RegisterForNotification: Notification is already registered on this channel`	Indicates that the notification is already registered from the same user on the same channel.

Table 6.200: Error messages

Example

The following sample code illustrates how to receive notification for channel data, on registering:

```
import scriptext
import e32

# Using e32.Ao_lock() to make main function wait till callback is hit
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def register_operation(trans_id, event_id, input_params):
    if trans_id != scriptext.EventCompleted:
        print "DataType: ", input_params["ReturnValue"]["DataType"]
        print "TimeStamp: ", input_params["ReturnValue"]["TimeStamp"]
        print "X-Axis Rotation: ", input_params["ReturnValue"]["XRotation"]
        print "Y-Axis Rotation: ", input_params["ReturnValue"]["YRotation"]
        print "Z-Axis Rotation: ", input_params["ReturnValue"]["ZRotation"]
```

6.10.3 GetChannelProperty

`GetChannelProperty` is used to get the channel property of the specified sensor channel.

The following is an example for using `GetChannelProperty`:

```
sensor_handle.call('GetChannelProperty',
                   {'ChannelInfoMap':{'ChannelId': result['ChannelId'],
                                      'ContextType': result['ContextType'],
                                      'Quantity': result['Quantity'],
                                      'ChannelType': result['ChannelType'],
                                      'Location': result['Location'],
                                      'VendorId': result['VendorId'],
                                      'DataItemSize': result['DataItemSize'],
                                      'ChannelDataTypeId': result['ChannelDataTypeId']},
                   'propertyId': u'DataRate'})
```

The following table summarizes the specification of `GetChannelProperty`:

Interface	`ISensor`
Description	Gets the specified property of a sensor channel.
Response Model	Synchronous
Pre-condition	ISensor interface is loaded.
Post-condition	Client application receives the requested property details.

Input Parameters

Input parameters define the `ChannelInfoMap` and `PropertyId`.

Name	Type	Range	Description
`ChannelInfoMap`	map as mentioned in `FindSensorChannel`. Refer **ChannelInfoMap** 6.194	NA	The map is obtained by invoking `FindSensorChannel`.

`PropertyId`	unicode string	Range for `PropertyId`:
		`DataRate`
		`Availability`
		`MeasureRange`
		`ChannelDataFormat`
		
		`ChannelAccuracy`
		`ChannelScale`
		`ScaledRange`
		`ChannelUnit`
		`SensorModel`
		`ConnectionType`
		`Description`
			The property ID string for which the property is being queried.

Table 6.201: Input parameters for GetChannelProperty

Output Parameters

Output contains `ReturnValue`. It also contains `ErrorCode` and an `ErrorMessage` if the operation fails. `ReturnValue` contains the requested channel property.

Name	Type	Range	Description
ErrorCode	int	NA	Contains the SAPI specific error code when the operation fails.
ErrorMessage	string	NA	Error Description in Engineering English.
ReturnValue	Channel property map: **Type**: **Name** string: PropertyId 32 bit int: PropertyDataType 32 bit Integer: ItemIndex bool: ReadOnly 32 bit int/ double/ string: PropertyValue	The channel property can either be of type integer, double, or string. Range for PropertyDataType: 0: For Integer datatype 1: For Double datatype 2: For String datatype	ReturnValue contains a map of key-value pair for channel property.

Table 6.202: Output parameters for GetChannelProperty

Errors

The following table lists the error codes and their values:

Error code value	Description
0	Success
1002	Bad argument type
1003	Missing argument
1012	Item not found

Table 6.203: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
Sensors:GetChannelProperty: Property id missing	Indicates that the property ID input parameter is missing.
Sensors:GetChannelProperty: Invalid property id	Indicates that the input property ID is invalid and does not fall within the specified range of property ID strings.
Sensors:GetChannelProperty: ChannelInfoMap missing	Indicates that the channel information map is not provided as input parameter.
Sensors:GetChannelProperty: Channel property not supported	Indicates that the channel property is not supported hence no value is returned.
Sensors:GetChannelProperty: Incomplete input param list	Indicates that the input param list is incomplete.

Table 6.204: Error messages

Example

The following sample code illustrates how to get the specified property of sensor channel:

```
try:
    result = sensor_handle.call('FindSensorChannel',
                                {'SearchCriterion': u'Rotation'})

    ChannelId = result[0]['ChannelId']
    ContextType = result[0]['ContextType']
    Quantity = result[0]['Quantity']
    ChannelType = result[0]['ChannelType']
    Location = result[0]['Location']
    VendorId = result[0]['VendorId']
    DataItemSize = result[0]['DataItemSize']
    ChannelDataTypeId = result[0]['ChannelDataTypeId']

    result = sensor_handle.call('GetChannelProperty',
                    {'ChannelInfoMap':{'ChannelId': ChannelId,
                                       'ContextType': ContextType,
                                       'Quantity': Quantity,
                                       'ChannelType': ChannelType,
                                       'Location': Location,
                                       'VendorId': VendorId,
                                       'DataItemSize': DataItemSize,
                                       'ChannelDataTypeId': ChannelDataTypeId},
                                       'PropertyId': u'DataRate'})
    print "Property Id: ", result["PropertyId"]
    print "PropertyDataType: ", result["PropertyDataType"]
    print "ItemIndex: ",  result["ItemIndex"]
    print "ReadOnly: ", result["ReadOnly"]
    print "PropertyValue: ",result["PropertyValue"]

except scriptext.ScriptextError, err:
    print "Error performing the operation : ", err
```

6.11 Sys Info

The SysInfo service provides Read or Write access to system information of a terminal. SysInfo service allows registering to system events identified by System Attributes (SAs). Some of the SAs are modifiable and supports notifications.

An Object with an entity and a key represents a System Attribute. An entity broadly represents a component in the device. A key is an attribute of an entity. For example, battery is an entity where, `ChargingStatus`, `BatteryStrength`, and `BatteryLevel` are the Keys of the entity.

The following sample code is used to load the provider:

```
import scriptext
msg_handle = scriptext.load('Service.SysInfo', 'ISysInfo')
```

The following table summarizes the SysInfo Interface:

Service provider	`Service.SysInfo`
Supported interfaces	`ISysInfo`

The following table lists the services available in SysInfo:

Services	**Description**
`GetInfo` 6.11.1	Reads system attributes value.
`SetInfo` 6.11.2	Modifies system attributes value.
`GetNotification` 6.11.3	Register for notifications.

6.11.1 GetInfo

`GetInfo` retrieves the value of a system attribute. It can be used in both synchronous and asynchronous mode.

The following are the examples for using `GetInfo`:

Synchronous

```
self.sysinfo_handle.call("GetInfo", {"Entity": u"General",
                                     "Key": u"VibraActive",
                                     "SystemData": {"Status": 1}})
```

Asynchronous

```
event_id = sysinfo_handle.call("GetInfo", {"Entity": u"Network",
                                           "Key": u"LocationArea"},
                               callback=print_location_area)
```

where, `print_location_area` is user defined function.

The following table summarizes the specification of `GetInfo`:

Interface	`ISysinfo`
Description	Retrieves the value of a system attribute.
Response Model	Synchronous and asynchronous
Pre-condition	ISysInfo Interface is loaded.
Post-condition	Returns an object on success.

Input Parameters

Input parameter specifies the Entity of system attribute information returned.

Name	Type	Range	Description
Entity	unicode string	For complete list of supported Entities, refer Key Values 6.11.4 section.	Entity of system attribute. For example, `Battery Network` and so on.
Key	unicode string	For complete list of supported Keys, refer Key Values 6.11.4 section.	Key of system attribute. For example, `BatteryStrenth HomeNetwork` and so on.
[SystemData]	map	`DriveInfo` Drive: unicode string	This is an optional parameter from API definition point of view. For some system attributes, you need to specify input. This map must contain one of the input data specifiers defined in System Data. For more information on input specifier refer the section Key Values 6.11.4.

Table 6.205: Input parameters for GetInfo

Output Parameters

Output parameter returns an object that contains the requested information. It also contains `ErrorCode` and an `ErrorMessage`, if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.
`ReturnValue`	map(System Data) Entity: string Key: string	For complete range of keys for the particular map, refer to **System Data** in **Key Value** section.	Output map always contains Entity and Key. Rest of the elements in the map depends on requested system attribute (Entity-Key). It will be one of the data specifiers defined in System Data. On requesting drive information using system attribute (for example: `Memory`, `DriveInfo`), `ReturnValue` map will contain Keys defined in `DriveInfo` Map.

Table 6.206: Output parameters for GetInfo

Errors

The following table lists the error codes and their values:

Error code value	Description
`-304`	General Error
`1002`	Bad argument type
`1003`	Missing argument
`1004`	Service not supported
`1010`	Entry exists
`1012`	Item not found

Table 6.207: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`SysInfo:GetInfo: Insufficient Arguments to process`	At least two input arguments are expected to process `GetInfo` service request.
`SysInfo:GetInfo: Entity:Input Parameter Missing`	Indicates mandatory parameter Entity is missing in the service request.
`SysInfo:GetInfo: Key:Input Parameter Missing`	Indicates mandatory parameter Key is missing in the service request.
`SysInfo:GetInfo: Incorrect SystemData Type, SystemData Must be a Map`	Indicates that either the optional parameter `SystemData` specified is not a map or content of the map is inappropriate to process request.
`SysInfo:GetInfo: CallBack and CmdOptions not matching`	Indicates that the situation where user specified callback and `CmdOptions` is set to Synchronous and vice-versa.

Table 6.208: Error messages

Example

The following sample code illustrates how to retrieve the current location area in synchronous mode:

```
import scriptext
import e32

# Using e32.Ao_lock() so that the main function can wait
# till the callback is hit.
lock = e32.Ao_lock()

# Callback function will be called when the requested service is complete
def print_location_area(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message is: " + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "Current Location Area is: " + input_params["ReturnValue"]["Status"]

    lock.signal()

# Load sysinfo service
sysinfo_handle = scriptext.load("Service.SysInfo", "ISysInfo")

# Make a request to query the required information
event_id = sysinfo_handle.call("GetInfo", {"Entity": u"Network", "Key": u"LocationArea"}, ca

print "Waiting for the request to be processed!"
lock.wait()
print "Request complete!"
```

6.11.2 SetInfo

`SetInfo` modifies the value of a system attribute. It takes a set of input parameters that define entity and key of `SystemAttribute` to modify the value of system attribute.

It is available in only synchronous mode.

The following is an example for using `GetInfo`:

Synchronous

```
sysinfo_handle.call("SetInfo", {"Entity": u"General",
                                "Key":u"VibraActive",
                                "SystemData" {"Status": 1}})
```

The following table summarizes the specification of `GetInfo`:

Interface	`ISysinfo`
Description	Modifies the value of a system attribute.
Response Model	Synchronous
Pre-condition	ISysInfo Interface is loaded.
Post-condition	Changes the system attribute on success

Input Parameters

Input parameter specifies an entity and key of system attribute.

Name	**Type**	**Range**	**Description**
Entity	unicode string	For complete list of supported Entities, refer Key Values 6.11.4 section.	Entity of system attribute. For example, `Connectivity` `Display` and so on.
Key	unicode string	For complete list of supported Keys, refer Key Values 6.11.4 section.	Key of system attribute. For example, `Bluetooth` `Wallpaper` and so on.
SystemData	map	Status information Status: int Wallpaper path `StringData`: unicode string	This map must contain one of the input data specifiers defined in System Data. For more information on input specifier refer the section Key Values 6.11.4.

Table 6.209: Input parameters for SetInfo

Output Parameters

Output parameter contains `ErrorCode` and an `ErrorMessage`, if the operation fails.

Name	**Type**	**Range**	**Description**
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.

Table 6.210: Output parameters for GetInfo

Errors

The following table lists the error codes and their values:

Error code value	Description
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1011	Access denied
1012	Item not found
1014	General error
1017	Path not found

Table 6.211: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
SysInfo:SetInfo: Insufficient Arguments to process	At least two input arguments are expected to process SetInfo service request.
SysInfo:SetInfo: Entity:Input Parameter Missing	Indicates mandatory parameter Entity is missing in the service request.
SysInfo:SetInfo: Key:Input Parameter Missing	Indicates mandatory parameter Key is missing in the service request.
SysInfo:SetInfo: Incorrect SystemData Type, SystemData Must be a Map	Indicates that either the optional parameter SystemData specified is not a map or content of the map is inappropriate to process request.
SysInfo:SetInfo: SystemData Argument Missing	Indicates that mandatory parameter SystemData is not specified in input argument list.
SysInfo:SetInfo: ASync Version Not Supported	This message is given when SetInfo is requested by specifying callback or CmdOptions set to Asynchronous request type.

Table 6.212: Error messages

Example

The following sample code illustrates how to set Vibra mode:

```
# Synchronous example: Setting Vibra mode

import scriptext

# Load sysinfo service
sysinfo_handle = scriptext.load('Service.SysInfo', 'ISysInfo')

# Make a request to set vibra mode
try:
    sysinfo_handle.call("SetInfo", {"Entity": u"General", "Key": u"VibraActive", "SystemData
    print "Request complete!"
except scriptext.ScriptextError:
    print 'Error in servicing the request'
```

6.11.3 GetNotification

`GetNotification` method registers a callback function to receive notifications of system data. It takes a set of input parameters that specifies entity and key of System Attribute.
It is available in only asynchronous mode.

The following is an example for using `GetNotification`:

Asynchronous

```
event_id = sysinfo_handle.call("GetNotification",
                               {"Entity": u"Battery", "Key":  u"ChargingStatus"},
                               callback=sysinfo_callback)
```

where, `sysinfo_callback` is user defined function.

The following table summarizes the specification of `GetNotification`:

Interface	`ISysinfo`
Description	Registers a callback function for listening to notifications.
Response Model	Asynchronous
Pre-condition	ISysInfo Interface is loaded.
Post-condition	Returns the generic parameter system data on success.

Input Parameters

Input parameter specifies the Entity and Key of system attribute, and system data.

Name	Type	Range	Description
Entity	unicode string	For complete list of supported Entities, refer Key Values 6.11.4 section.	Entity of system attribute. For example, `Battery` `Network` and so on.
Key	unicode string	For complete list of supported Keys, refer Key Values 6.11.4 section.	Key of system attribute. For example, `BatteryStrenth` `CurrentNetwork` and so on.
[SystemData]	map	Status information Status: int `DriveInfo` Drive: unicode string CriticalSpace: int	This is an optional parameter from API definition point of view. For some system attributes, you need to specify input. This map must contain one of the input data specifiers defined in System Data. For more information on input specifier refer the section Key Values 6.11.4. Here are some system attributes for which status information is used as input specifier. `DriveNumber` and critical space to be specified for drive critical memory notifications. For example, Battery- `BatteryStrength` (Threshold Strength value). Network- `Signal` (Threshold Signal value).

Table 6.213: Input parameters for GetNotification

Output Parameters

Output parameter returns an object that contains the requested information. It also contains `ErrorCode` and an `ErrorMessage`, if the operation fails.

Name	Type	Range	Description
`ErrorCode`	int	NA	Contains the SAPI specific error code when the operation fails.
`ErrorMessage`	string	NA	Error Description in Engineering English.
`ReturnValue`	map(System Data) Entity: string Key: string	For complete range of keys for the particular map, refer to **System Data** in **Key Value** section.	Output map always contains Entity and Key. Rest of the elements in the map depends on requested system attribute (Entity-Key). It will be one of the data specifiers defined in System Data. On requesting drive information using system attribute (for example: `Memory`, `DriveInfo`), `ReturnValue` map will contain Keys defined in `DriveInfo` Map.

Table 6.214: Output parameters for GetNotification

Errors

The following table lists the error codes and their values:

Error code value	Description
`1002`	Bad argument type
`1003`	Missing argument
`1006`	Service not ready
`1010`	Entry exists
`1012`	Item not found

Table 6.215: Error codes

Error Messages

The following table lists the error messages and their description:

Error messages	Description
`SysInfo:GetNotification: Insufficient Arguments to process`	At least two input arguments are expected to process `GetNotification` service request.
`SysInfo:GetNotification: Entity:Input Parameter Missing`	Indicates mandatory parameter Entity is missing in the service request.
`SysInfo:GetNotification: Key:Input Parameter Missing`	Indicates mandatory parameter Key is missing in the service request.
`SysInfo:GetNotification: Incorrect SystemData Type, SystemData Must be a Map`	Indicates that either the optional parameter `SystemData` specified is not a map or content of the map is inappropriate to process request.
`SysInfo:GetNotification: Sync Version Not Supported`	This message is given when `GetNotification` is requested without specifying callback or `CmdOptions` set to Synchronous request type.

Table 6.216: Error messages

Example

```
import scriptext
import e32

lock = e32.Ao_lock()
messaging_handle = scriptext.load('Service.SysInfo', 'ISysInfo')

def sysinfo_callback(trans_id, event_id, input_params):
    if event_id != scriptext.EventCompleted:
# Check the event status
        print "Error in retrieving required info"
        print "Error code is: " + str(input_params["ReturnValue"]["ErrorCode"])
        if "ErrorMessage" in input_params["ReturnValue"]:
            print "Error message is: " + input_params["ReturnValue"]["ErrorMessage"]
    else:
        print "Current Battery charging value: " + str(input_params["ReturnValue"]["Status"]
    lock.signal()

# Make a request to get notification
event_id = sysinfo_handle.call("GetNotification", {"Entity": u"Battery", "Key": u"ChargingSt
lock.wait()
```

6.11.4 Key Values

This section details the key values used in the context of SysInfo Service API invocation.

System Attributes

Entity

Key	DataType	Description
Entity	string	Part of system attribute represents an Entity.

Table 6.217: Entity

Key

Key	DataType	Description
Key	string	Part of system attribute represents a key with in Entity.

Table 6.218: Key

System Data

`SystemData` is a map whose keys are defined by one of the following data specifiers described in this section. These are added to the `ReturnValue` map. Typically, `SystemData` is status information that is represented using integer. In some cases, it is a map of network details, or a list having connection details of each connection. This section covers all the possible `SystemData` types.

Key	DataType	Description
Status	int	This Key provides status information of system attribute. For example, `BatteryLevel` (0-7), `NetworkMode`, and `BTPower` (0-OFF, 1-ON) and so on.

Table 6.219: Status

Key	DataType	Description
StringData	string	This Key provides data of type string to specify. For example, WallpaperPath, IMEI Number, and PhoneModel and so on.

Table 6.220: StringData

Key	DataType	Description
NetworkName	string	This Key provides name of the network.
NetworkStatus	int	**Status**: **Description** -1: Unknown 0: Available. A network that the ME is allowed to register to. 1: Current. This is the currently registered network. 2: Forbidden. A network that the ME is not allowed to register to.
NetworkMode	int	**Mode**: **Description** -1: Unknown 0: Unregistered 1: Global System for Mobile communications (GSM) 2: Advanced Mobile Phone System (AMPS) 3: Code Division Multiple Access (CDMA95) 4: Code Division Multiple Access (CDMA2000) 5: Wideband Code Division Multiple Access (WCDMA) 6: Time Division, Code Division Multiple Access (TD-CDMA)
CountryCode	string	Mobile Country Code (MCC).
NetworkCode	string	Mobile Network Code (MNC).
LocationStatus	bool	True: Location Area Code (LAC), CellId are valid. False: Location Area Code (LAC), CellId are invalid.
AreaCode	int	Location Area Code LAC.
CellId	int	CellId.

Table 6.221: NetworkInfo

Key	DataType	Description
ConnectionList	Iterator	This Iterator points to the list of available active data connections. ConnectionInfo map represents a data connection.

Table 6.222: ConnectionList

Key	DataType	Description
ConnectionStatus	int	0: DisConnected 1: Connected
IAPID	int	Access point ID
ConnectionType	int	Coloured connection types are not supported. **Mode**: **Description** -1: Unknown 0: Circuit Switch Data (CSD) 1: WCDMA 2: LAN [Emulator] 3: CDMA2000 4: General Packet Radio Service (GPRS) 5: High Speed Circuit Switched Data (HSCSD) 6: Enhanced Data rates for Global Evolution GPRS(EdgeGPRS) 7: Wireless Local Area Network(WLAN) 8: Bluetooth 9: Virtual VPN
IAPName	string	Access Point Name. For example, www.airtelgprs.com.
NetworkName	string	Network name applicable for WLAN networks.
IAPConnectionName	string	The access point connection name that is, MobileOffice, MyGprs.

Table 6.223: ConnectionInfo

Key	DataType	Description
AccessoryType	int	-1: Unknown 0: HeadSet 1: BTHeadSet 2: CarKit 3: BTCarKit
AccessoryState	int	-1: Unknown 0: Disconnected 1: Connected

Table 6.224: AccessoryInfo

Key	DataType	Description
AccessoryList	Iterator	This Iterator points to the list connected accessories. AccessoryInfo map represents an accessory.

Table 6.225: AccessoryList

Key	DataType	Description
LanguageList	List of int	This List points to the list of supported language enumerations, which are defined in S60.

Table 6.226: LanguageList

Key	DataType	Description
MajorVersion	string	This List points to the list of supported language enumerations, which are defined in S60.
MinorVersion	string	Minor number of the version. For example, for 3.1, 1 will be minor.

Table 6.227: Version

Key	DataType	Description
Drive	string	Drive is a string. For example, **c:¿**\\, **d:¿**\\ and so on.
TotalSpace	string	Total Space in bytes.
FreeSpace	string	Free Space in bytes.
CriticalSpace	int	This is the critical free space in bytes.
MediaType	int	0: MediaNotPresent 1: MediaUnknown 2: MediaFloppyDisk 3: MediaHardDisk 4: MediaCdRom 5: MediaRam 6: MediaFlash 7: MediaRom 8: MediaRemote 9: MediaNANDFlash 10: MediaRotatingMedia
BatteryState	int	0: BatNotSupported 1: BatGood 2: BatLow
DriveName	string	Drive name

Table 6.228: DriveInfo

Key	DataType	Description
XPixels	int	X-Pixels
YPixels	int	Y-Pixels

Table 6.229: Resolution

Key	DataType	Description
DriveList	List of strings	This List points to the list of drives in the terminal. Drives are represented as strings. For example, **c:¿**\\ and so on.

Table 6.230: DriveList

Key	DataType	Description
StringList	List of strings	This List points to the list of available USB modes.

Table 6.231: StringList

6.12 Appendix

6.12.1 Platform Service API Error Codes and Description

SErrorcode	Error Description
-306	Error in processing version information.
-305	Undefined data type is passed as input.
-304	General Error
-302	Interface is not found.
-301	Service is not found.
0	Success
1000	Invalid service argument
1001	Unknown argument name
1002	Bad argument type
1003	Missing argument
1004	Service not supported
1005	Service in use
1006	Service not ready
1007	No memory
1008	Hardware not available
1009	Server busy
1010	Entry exists
1011	Access denied
1012	Not found
1013	Unknown format
1014	General error
1015	Cancel success
1016	Service timed-out
1017	Path not found

Table 6.232: Platform Service API Error Codes

6.12.2 EventID

EventID	Description
EventStarted	Asynchronous service informs the user to prepare for action.
EventCompleted	Asynchronous service request completed.
EventCanceled	Asynchronous service request cancelled.
EventError	Error during asynchronous service request.
EventStopped	Service no longer available or stopped.
EventQueryExit	Specifies if exit is possible.
EventInProgress	Asynchronous service execution in progress.
EventOutParamCheck	This relates to KLiwOptOutParamCheck.
EventInParamCheck	This relates to KLiwOptInParamCheck.

Table 6.233: EventID

CHAPTER

SEVEN

Module Repository

Introduction

Starting with PyS60 1.9.x series, the Python core is upgraded to 2.5.4. With this, PyS60 1.9.x is loaded with much more core Python modules than the previous PyS60 releases that were based on Python 2.2.2 core. This also means that the size of the runtime SIS file increases by many folds. Bigger runtime SIS means longer time to download it and also longer time to install it on to a device. This also results in slowing down the interpreter load time.

The following were the main ideas behind this packaging tool:

- Reduce the runtime SIS file size, with no compromise on the number of modules supported by the Python runtime
- Easier SIS packaging for Python applications
- Easier and more robust ways to use and distribute extension modules with your application

These have been achieved in PyS60 1.9.x by:

- Reducing the runtime SIS size by including only the most essential modules in it.
- Providing a repository for those modules that are not included in the runtime SIS. This repository is a part of the PyS60 Application packager installed on to the host system.
- At the time of packaging a Python application it into a SIS, the application is scanned to find the dependency modules and packaged them along with the application files.

Module repository (hereafter module-repo) is the name given to the directory structure where all the Python modules are placed. This can be extended by the users by adding their own modules in it. This also includes information required to package all the Python modules on which a given Python script is dependent on. This module-repo is installed on the host machine along with the PyS60 Application Packager.

Note: The following codecs related modules _codecs_cn, _codecs_hk, _codecs_jp, _codecs_kr and _codecs_tw are not packaged automatically. If the application depends on any of them, then they can be packaged by explicitly specifying them using the –extra-modules option of ensymble. (Use 'Additional options' field in the GUI)

Extending Module-repo

Developers can extend the module-repo by adding the new modules that they develop or receive from other extension developers. Module-repo can be extended by the steps mentioned below:

- Place all the Python modules in a directory, named with the module name and copy it to module-repo\dev-modules directory.
- The module directory should also contain a configuration file named as module_config.cfg. This file contains the information about the module dependencies. It should be a dictionary with key 'type' which

has a value 'repo' indicating that it is part of module-repo and 'deps' which is a list of Python modules on which the module is directly dependent. The packager will automatically scan the application(only .py files) for dependencies related to Python core modules. If the application is dependent on any dev-module then it should be mentioned here.

```
{'type': 'repo',  'deps': ["socket", "btsocket", "my_mod1"]}
```

Directory structure on the PC

```
<PythonForS60>\
                module-repo\
                        standard-modules\
                            ...
                        dev-modules\
                            module_search_path.cfg
                            mod1\
                                module_config.cfg
                                mod1.py
                            mod2\
                                module_config.cfg
                                kf_mod2.pyd
                            mod3\
                                module_config.cfg
                                mod3\
                                    __init__.py
                                    mod3_run.py
                            mod4\
                                module_config.cfg
                                kf_mod4core.pyd
                                kf_mod4base.pyd
                                mod4\
                                    __init__.py
                                    mod4_wrapper.py
```

- Dev module type-1 : mod1 - A single py file
  For developer modules that have a single Python file, the directory should be named after the file and placed under dev-modules folder. If mod1.py imports any third party PYD modules then it has to explicitly mention the dev module name so that it is packaged along with the application. This can be mentioned in the module_config.cfg file.
- Dev module type-2 : mod2 - A single PYD
  The PYD should be directly under the module folder and should be prefixed with 'kf_'. If the PYD is dependent on some other dev module then it should be mentioned in the module_config.cfg file.
- Dev module type-3 : mod3 - A Python package
  If the developer module is a Python package then the package folder should be under another folder named after the pacakge. The module_config.cfg should be outside the package and serves the same purpose as mentioned above.
- Dev module type-4 : mod4 - A Python package with PYDs
  This is a mixture of type-2 and type-3 and in this scenario the PYDs should be at the top level of the module directory. If an application imports mod4 then the application packager will find the mod4 directory under dev-modules folder and package the mod4 Python package along with the PYDs(kf_mod4core.pyd and kf_mod4base.pyd).

Distributing extension modules to application developers

Distributing extension modules to application developers is much easier now with the PyS60 application packager scanning for dependencies automatically. The extension module developers should create a zip/rar/tar archive of their module directory in the format mentioned above so that the application developers can directly extract it to the dev-modules folder. After this the application developers just need to use PyS60 application packager to package their script. The application packager will automatically scan the dependencies and package the dev-modules the script is dependent on.

Module search path

A module when being included in the application package, is first searched in the paths specified in the configuration file - module_search_path.cfg unless it is distributed by PyS60(e.g. : messaging, contacts, sysinfo etc.). If it is not found in these locations, then it searches for this module in the module-repo.

The paths in the config file are in a list(['path1', 'path2']). This feature is useful in the scenario when the Python modules (especially the extension modules - PYD files) are not present in the module-repo. The developer can just specify the path to be searched for the modules, instead of copying them to the module-repo.

For example, if the developer is developing an extension module, he can specify the path to \epoc32\release\armv5\urel in the module search path and the module is automatically picked up while packaging. Without this option, the developer needs to copy the module to module-repo every time he compiles the code.

Directory structure on a device

The application SIS will contain all its dependencies packaged along with it. All the py files except default.py are zipped into lib.zip and placed in the application's private directory. A white-list.cfg is maintained which contains entries for all the PYDs packaged with the application which will be used by PyS60's import mechanism.

The files are placed on a device as shown below:

```
!:\
    private\
        [app-UID]\
            white-list.cfg [List of PYDs packaged with the app]
            default.py [application script]
            lib.zip [standard, dev and application py files]

        resource\
            python25\
                core-modules [PY files compiled and zipped - placed by base runtime]

        sys\
            bin\
                core-modules [PYDs - placed by base runtime]
                repo-modules [standard-pyd and dev-pyd files, renamed with application's UID
```

Include additional modules

A new option is added to PyS60 application packager which can be used to include additional modules with the application. If the application packager does not include a module needed by the application automatically, then the user can use the `--extra-modules` option to forcefully include additional modules with the application.

CHAPTER

EIGHT

Extending and Embedding PyS60

Extending and Embedding Python is explained in the following sections with examples wherever necessary.

8.1 Extending PyS60

The general rules and guidelines for writing Python extensions apply in the S60 Python environment as well.

The steps for the implementation of an extension modules include:

- Preparation of the data structures that make the C/C++ coded extensions visible to the Python interpreter and make it possible to perform calls from Python to C/C++ code
- Conversions between C/C++ representations of the Python objects and object types used in the extension code
- Maintenance of the reference counts of the C/C++ representations of the Python objects
- Passing of exceptions between C/C++ code and Python
- Management of interpreter's thread state and the interpreter lock

In addition to the concerns common for all Python C extensions, the following principles should be considered when implementing new Python interfaces in the S60 environment:

- Maximize the usage of Python's built-in types at the interfaces.
- Related to the above: design interfaces in such a way that information can be passed between them with minimal conversions.
- Convert Symbian operating system exceptions or errors to Python exceptions.
- Unicode strings are used at the interfaces to represent text that gets shown on the GUI. They can be passed to and from Symbian operating system without conversions.
- While performing potentially long-lasting or blocking calls from an extension implementation to services outside the interpreter, the interpreter lock must be released and then re-acquired after the call.
- Rather than always implementing a thin wrapper on top of a Symbian OS facility, consider the actual task for which the script writer needs the particular interface. For example, if the task involves interaction with the users using the GUI, the script writer's interest may well be limited to performing the interaction or information exchange in a way that is compatible with the UI style rather than having full control of the low-level details of the GUI implementation.
- The C/C++ implementation of a Python interface should be optimized for performance and covering access to the necessary features of the underlying Platform. Where necessary, the Python programming interface can be further refined by wrapper modules written in Python.

The pyd name should be of the format 'kf_¡module-name¿.pyd'. Note that this change is required only for the pyd name and module name is not required to have this prefix.

The module initialization function must be exported at ordinal 1. The module identification is based on the filename only. As a special feature of PyS60, an optional module finalizer function may be exported at ordinal 2.

The extension modules added by the developer should be placed in the module repo folder of the PyS60 Application Packager so that the compiled PYD's can be picked up while packaging. For information on this topic refer 8.1.3, Distributing Extension Modules.

8.1.1 Example Extension Module

To demonstrate the writing of an extension module the source and MMP contents for an example module , 'elemlist' is given below. This module is about extracting the pointers in a cons cell. A cons cell is composed of two pointers. The car and cdr are primitive operations upon linked lists composed of cons cells. The car operation extracts the first pointer, and the cdr operation extracts the second.

```
#include "Python.h"

/* type-definition & utility-macros */
typedef struct {
    PyObject_HEAD
    PyObject *car, *cdr;
} cons_cell;

staticforward PyTypeObject cons_type;

/* a typetesting macro (we don't use it here) */
#define is_cons(v) ((v)->ob_type == &cons_type)

/* macros to access car & cdr, both as lvalues & rvalues */
#define carof(v) (((cons_cell*)(v))->car)
#define cdrof(v) (((cons_cell*)(v))->cdr)

/* ctor (factory-function) and dtor */
static cons_cell*
cons_new(PyObject *car, PyObject *cdr)
{
    cons_cell *cons = PyObject_NEW(cons_cell, &cons_type);
    if(cons) {
        cons->car = car; Py_INCREF(car);  /* INCREF when holding a PyObject* */
        cons->cdr = cdr; Py_INCREF(cdr);  /* ditto */
    }
    return cons;
}
static void
cons_dealloc(cons_cell* cons)
{
    /* DECREF when releasing previously-held PyObject*'s */
    Py_DECREF(carof(cons)); Py_DECREF(cdrof(cons));
    PyObject_DEL(cons);
}

/* Python type-object */
statichere PyTypeObject cons_type = {
    PyObject_HEAD_INIT(0)    /* initialize to 0 to ensure Win32 portability */
    0,                  /*ob_size*/
    "cons",             /*tp_name*/
    sizeof(cons_cell), /*tp_basicsize*/
    0,                  /*tp_itemsize*/
    /* methods */
    (destructor)cons_dealloc, /*tp_dealloc*/
    /* implied by ISO C: all zeros thereafter */
};

/* module-functions */
static PyObject*
cons(PyObject *self, PyObject *args)    /* the exposed factory-function */
{
    PyObject *car, *cdr;
    if(!PyArg_ParseTuple(args, "OO", &car, &cdr))
        return 0;
    return (PyObject*)cons_new(car, cdr);
}
static PyObject*
car(PyObject *self, PyObject *args)     /* car-accessor */
{
    PyObject *cons;
    if(!PyArg_ParseTuple(args, "O!", &cons_type, &cons))  /* type-checked */
        return 0;
    return Py_BuildValue("O", carof(cons));
}
static PyObject*
cdr(PyObject *self, PyObject *args)     /* cdr-accessor */
{
    PyObject *cons;
```

information about writing an extension module refer *Extending Python* from the mainline Python documentation.

The header file Python.h makes the Python API's accessible in the code. `car()`, `cdr()` and `cons()` are the functions exposed at Python level so that they can be called after importing the `elemlist` module. The C implementation of these functions take arguments as Python objects. To do anything with them in the C function we have to convert them to C values.

`elemlist_methods` is the method table for the module. The method table is passed to the interpreter in the module's initialization function, `initelemlist()`. The initialization function must be named `initname()`, where name is the name of the module, and should be the only non-static item defined in the module file. The PyMODINIT_FUNC declares the function as `void` return type, declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`. When the Python program imports module elemlist for the first time, `initelemlist()` is called.

The MMP file contents for the above source is as follows:

```
TARGETTYPE  dll
TARGET    kf_elemlist.pyd

CAPABILITY LocalServices NetworkServices ReadUserData WriteUserData UserEnvironment

NOSTRICTDEF
DEFFILE   elemlist.def

/* If global data is present in the extension module then this macro should be
 * defined in the mmp file.
 */
EPOCALLOWDLLDATA

SYSTEMINCLUDE    \epoc32\include\python25
SYSTEMINCLUDE    \epoc32\include\stdapis
SYSTEMINCLUDE    \epoc32\include

LIBRARY     python25.lib

SOURCEPATH  ..\src
SOURCE      elemlist.c
```

An example usage of the 'elemlist' extension module can be:

```
from elemlist import *
cell = cons(1, 2)
print "car(cell) :", car(cell)
print "cdr(cell) :", cdr(cell)
```

8.1.2 Compiling the extension module

Requirements

- Series 60 SDK, 3rdEd or higher
- Python for S60 1.9.x SDK package
- Open C/C++ plug-in. Refer the release notes for the version of OpenC to be installed for this release.

Installation
Place the Python for Series 60 SDK 3rdEd under the Symbian SDK installation directory, at the same level as the

”epoc32” directory. Extract the SDK zip package here(On Windows if you have WinZip installed, right-click on the zip and in the menu select : ’Winzip-¿Extract to here’)

Building
Modify the mmp file to include ”Location” capability while building for 3rdEdFp2 and higher devices. A script file build_all.cmd that does all the necessary steps (and some extra cleanup, just to be sure) has been provided for convenience. You can either use that or perform the build manually using these instructions.

- Go to the elemlist directory. Enter:
  bldmake bldfiles
- To build the extension for the device, enter:
  abld build gcce urel
  abld freeze gcce
  abld build gcce urel
  You should find the built module in (path to your SDK)\epoc32\release\gcce\urel\kf_elemlist.pyd
- To build it for the emulator environment, enter:
  abld build winscw udeb
  abld freeze winscw
  abld build winscw udeb
  You should find the built module in (path to your SDK)\epoc32\release\winscw\ubed\kf_elemlist.pyd

Note: The ”freeze” step needs to be done only when you add any function exports. After ”freeze”, just one ”abld build gcce urel” or ”abld build winscw udeb” will rebuild the code properly.

8.1.3 Distributing extension modules

Distributing extension modules to application developers is much easier now with the PyS60 application packager scanning for dependencies automatically. Please refer the topic **Distributing extension modules to application developers** in the Chapter *Module Repository* for information on how this is done.

8.2 Embedding PyS60

There is not much change with respect to embedding Python from what is mentioned in the Python mainline document apart from the custom memory allocator which is explained later in this Section.

The following code snippet, which prints ’Hello World!’ on the screen, demonstrates the embedding of Python interpreter in a C code:

The source file contents are as follows:

```
#include <Python.h>
/* This is a GCCE toolchain workaround needed when compiling with GCCE
   and using main() entry point */
#ifdef __GCCE__
#include <staticlibinit_gcce.h>
#endif

int main(void)
{

    SPy_DLC_Init();
    SPy_SetAllocator(SPy_DLC_Alloc, SPy_DLC_Realloc, SPy_DLC_Free, NULL);
    Py_Initialize();
    PyRun_SimpleString("print 'Hello World!'");
    Py_Finalize();
    SPy_DLC_Fini();
    return 0;
}
```

The basic initialization function is `Py_Initialize()`. This initializes the table of loaded modules, and creates the fundamental modules __builtin__, __main__ and sys. It also initializes the module search path (sys.path). `Py_Finalize()` is called when the application is done with its use of Python and wants to free all memory allocated by Python.

PyS60 provides a DLC custom allocator which can be used instead of Python memory allocator. `SPy_SetAllocator()` is used for redirecting the allocator used by Python. The arguments to this function are the custom functions for allocating, reallocating, freeing the memory and a context pointer in that order. `SPy_DLC_Init()` is used for initializing the DLC custom allocator. `SPy_DLC_Fini()` is used for finalizing the DLC custom allocator and doing a memory cleanup. If you want to use your own custom allocator you will have to define the allocation, reallocation and free memory functions and pass the function names to `SPy_SetAllocator()`.

For more information on embedding Python, refer *Embedding Python in Another Application*

The MMP file contents for the above source is as follows:

```
TARGET        helloworld.exe
TARGETTYPE    exe

SYSTEMINCLUDE    \epoc32\include\python25
SYSTEMINCLUDE    \epoc32\include\stdapis
SYSTEMINCLUDE    \epoc32\include

/* Using main() as entry point */
STATICLIBRARY libcrt0.lib

/* libc and euser are always needed when using main() entry point */
LIBRARY libc.lib
LIBRARY euser.lib
LIBRARY python25.lib

SOURCEPATH      ..\src
SOURCE          helloworld.cpp
```

8.3 Porting 1.4.x to 1.9.x

The changes needed for porting existing native PyS60 extensions are as follows:

- From Symbian 9.1 onwards Symbian allows Writable Static Data in a DLL by making use of EPOCALLOWDLLDATA keyword in the mmp file. Main reason for this one is for porting some non-Symbian applications onto Symbian. Thus TLS functionality is no longer needed. Use EPOCALLOWDLLDATA in the MMP file if the module has initialized static data.
- Use `PyGILState_Ensure()` and `PyGILState_Release()` functions for acquiring and releasing the global interpreter lock, instead of using `PyEval_RestoreThread(PYTHON_TLS->thread_state)` and `PyEval_SaveThread()`.
- The interpreter DLL name is changed to python25.lib. This change has to be reflected in the MMP file so that the module is linked against this DLL instead of python222.lib used in 1.4.x.
- The Python header files are now in \epoc32\include\Python25 and hence the MMP file needs to be updated accordingly.
- The pyd name should be kf_¡module-name¿.pyd.
  Note: This change is required only for the pyd name and module name is not required to have this prefix.
- Packaging an extension module is explained in section 8.1.3, Distributing extension modules.

  Note: The init-function still needs to be exported in the pyd at ordinal 1.

Script related changes

- The main script of the PyS60 applications, default.py is not executed directly, as was the case in PyS60 1.4.x. The wrapper script, launcher.py is first executed which in turn does an execfile on the default.py. Therefore, to exit the application programmatically use `appuifw.app.set_exit()` or `sys.exit()`
- PyS60 1.4.x extension modules socket and calendar are renamed to btsocket and e32calendar due to the conflicting names with Python core modules. Two packaging modes pys60 and pycore have been provided with Application Packager tool to maintain the compatibility with PyS60 1.4.x binaries. The existing scripts dependent on these extension modules need not be modified if it is packaged with pys60 mode.
- Unlike PyS60 1.4.x the module names are case sensitive from PyS60 1.9.x. So the scripts written for 1.4.x will require changes to account for this.

CHAPTER

NINE

Terms and Abbreviations

The following list defines the terms and abbreviations used in this document:

Term	Definition
AAC; Adaptive Audio Coding	AAC provides basically the same sound quality as MP3 while using a smaller bit rate. AAC is mainly used to compress music.
Advertise	Advertise service in Bluetooth makes it known that a certain Bluetooth service is available.
AMR	Adaptive Multi-rate Codec file format.
API	Application Programming Interface
Bluetooth	Bluetooth is a technology for wireless communication between devices that is based on a low-cost short-range radio link.
BPP	Bits Per Pixel
C STDLIB	Symbian OS's implementation of the C standard library
Dialog	A temporary user interface window for presenting context-specific information to the user, or prompting for information in a specific context.
Discovery	Discovery is a process where Bluetooth finds other nearby Bluetooth devices and their advertised services.
DLL	Dynamic link library
GSM; Global System for Mobile communication	GSM is a digital mobile telephone system that uses a variation of time division multiple access. It digitizes and compresses data, then sends it down a channel with two other streams of user data, each in its own time slot.
GUI	Graphical User Interface
I/O	input/output
IP	Internet Protocol
MBM; Multi-BitMap	The native Symbian OS format used for pictures. MBM files can be generated with the `bmconv.exe` tool included in the S60 SDK.
MIDI; Musical Instrument Digital Interface	A protocol and a set of commands for storing and transmitting information about music.
MIF; Multi-Image File	MIF files are similar to MBM files and can contain compressed SVG-T files. This file type can be generated with the `MifConv.exe` tool.
MIME; Multipurpose Internet Mail Extensions	MIME is an extension of the original Internet e-mail protocol that can be used to exchange different kinds of data files on the Internet.
MP3	A standard technology and format for compressing a sound sequence into a very small file while preserving the original level of sound quality when it is played.
OS	Operating System
Real Audio	An audio format developed by Real Networks.
RDBMS	Relational database management system
SMS; Short Message System (within GSM)	SMS is a service for sending messages of up to 160 characters, or 224 characters if using a 5-bit mode, to mobile phones that use GSM communication.

Term	Definition
Softkey	Softkey is a key that does not have a fixed function nor a function label printed on it. On a phone, selection keys reside below or above on the side of the screen, and derive their meaning from what is presently on the screen.
SQL	Structured Query Language
SVG, SVG-T; Scalable Vector Graphics (-Tiny)	XML-based vector graphics format for describing two-dimensional graphics and graphical applications.
Twip	Twips are screen-independent units to ensure that the proportion of screen elements are the same on all display systems. A twip is defined as 1/1440 of an inch, or 1/567 of a centimeter.
UI	User Interface
UI control	UI control is a GUI component that enables user interaction and represents properties or operations of an object.
WAV	A file format for recording sound, especially in multimedia applications.

BIBLIOGRAPHY

[1] G. van Rossum, and F.L. Drake, Jr., editor. [Python] Library Reference. Available at http://www.python.org/doc

[2] G. van Rossum, and F.L. Drake, Jr., editor. Extending and Embedding [the Python Interpreter]. Available at http://www.python.org/doc

[3] G. van Rossum, and F.L. Drake, Jr., editor. Python/C API [Reference Manual]. Available at http://www.python.org/doc

[4] S60 SDK documentation, available at http://www.forum.nokia.com/

[5] Audio & Video section on the *Forum Nokia* Web site (for Nokia devices), http://www.forum.nokia.com/audiovideo

[6] Developers section on the *S60 Platform* Web site (for all S60 devices), http://www.s60.com/

[7] Python for S60 developer discussion board http://discussion.forum.nokia.com/

[8] Scalable Vector Graphics (SVG) 1.1 Specification http://www.w3.org/TR/SVG/

APPENDIX

A

Known Issues

Limitations:

1. On this platform, the stack size is significantly less compared to other platforms and hence deep recursions and high memory consuming operations may cause stack overflow.

2. Calling setsockopt() on UDP sockets requires that the application is signed with NETWORKCONTROL capability.

3. The file access time and modification time are same on Symbian and the same behavior is reflected in the utime module.

4. Using the Listbox widget in large/full screen mode results in an unrefreshed area at the bottom of the screen. This is a S60 platform limitation and as mentioned in the SDK documentation (http://www.forum.nokia.com/infocenter/index.jsp?topic=/S60_3rd_Edition_Cpp_Developers_Library/GUID-759FBC7F-5384-4487-8457-A8D4B76F6AA6/html/classCAknSelectionListDialog.html) it works only in the main pane.

Bugs:

Information about all known bugs can be got from here https://garage.maemo.org/tracker/?group_id=854.

Failing test cases:

Below is the summary of the standard regrtest Python module executed on this platform.

Tests	Comments
test_asynchat	This test failed when tested on a 5th edition device and is under investigation. It works fine on 3rd edition and
test_builtin	test_cmp fails because of Stack Overflow
test_compiler	testCompileLibrary, testLineNo fail because of Stack Overflow. Another reason for this fail is related to the zi
test_copy	test_deepcopy_reflexive_dict, test_deepcopy_reflexive_list and test_deepcopy_reflexive_tuple fail because of Sta
test_datetime	time conversion issues in the dependent libraries
test_doctest	Python bug (http://bugs.python.org/issue1540)
test_exceptions	testInfiniteRecursions fail because of Stack Overflow
test_mailbox	Under investigation
test_richcmp	This fail is because of Stack overflow
test_tarfile	Under investigation
test_time	DST conversion issues in the dependent libraries
test_importhooks	This fails because distutils module not supported.
test_pyclbr	This fail is related to the zipping of all the standard Python files.
test_repr	This fail is related to the zipping of all the standard Python files.
test_urllib2	This fail is related to the zipping of all the standard Python files.
test_urllib2net	Under investigation

Table A.1: Failing test cases

Skipped test cases:

These tests are skipped as one or more modules needed by them are not supported on this platform.

test_aepack	test_al	test_applesingle	test_audioop	test_bsddb	test_bsddb185	test_bsddb3
test_bz2	test_cd	test_cl	test_cmd_line	test_commands	test_crypt	test_ctypes
test_curses	test_dbm	test_distutils	test_dl	test_fork1	test_gdbm	test_gl
test_grp	test_hotshot	test_imageop	test_imgfile	test_ioctl	test_largefile	test_linuxau
test_macfs	test_macostools	test_macpath	test_mhlib	test_mmap	test_nis	test_openpty
test_ossaudiodev	test_pep277	test_plistlib	test_poll	test_popen	test_popen2	test_pty
test_pwd	test_resource	test_rgbimg	test_scriptpackages	test_signal	test_sqlite	test_startfile
test_subprocess	test_sunaudiodev	test_sundry	test_tcl	test_threadsignals	test_wait3	test_wait4
test_winreg	test_winsound	test_zipfile64				

Table A.2: Skipped test cases

These tests are also skipped, but are related to the zipping of all the standard Python files.

test_email	test_email_codecs	test_email_renamed	test_import

Table A.3: Skipped test cases related to zipping of standard Python files

APPENDIX

B

Reporting Bugs

In order to improve the quality of Python for S60 the developers would like to know of any deficiencies you find in Python for S60 or its documentation.

Before submitting a report, you will be required to log into garage.maemo.org; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the project Python for S60 Bug Tracker on garage.maemo.org (https://garage.maemo.org/tracker/?group_id=854). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the ”Bugs: Browse” link present at the top of the page.

If the problem you’re reporting is not already in the bug tracker, then click on the ”Submit New” link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the ”Summary” and ”Detailed descriptioin” fields. For the summary, enter a *very* short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python for S60 you used using the ”Versioin” field, whether any extension modules were involved and what hardware (the S60 device model or emulator) you were using, including version information of the S60 SDK and your device firmware version as appropriate. You can see the device firmware version by entering `*#0000#` on the device keypad - please include all information that is shown by this code.

The only other field that you may want to set is the ”Category” field, which allows you to place the bug report into a broad category (such as ”Documentation” or ”core”).

Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See Also:

How to Report Bugs Effectively
(http://www-mice.cs.ucl.ac.uk/multimedia/software/documentation/ReportingBugs.html)
: Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

Bug Writing Guidelines
(http://www.mozilla.org/quality/bug-writing-guidelines.html)
: Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.

MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W