

Introduction to the Camera API

Mark Shackman

Published by the Symbian Developer Network

Version: 1.02 – August 2008

1 INTRODUCTION TO THE ONBOARD CAMERA API	2
2 ONBOARD CAMERA API FACILITIES	2
2.1 FUNCTIONALITY	2
2.2 BASE CLASSES	2
3 CODE EXAMPLES	3
3.1 GAINING ACCESS TO THE CAMERA	3
3.1.1 <i>The Observer Class</i>	3
3.1.2 <i>Checking Camera Capability</i>	4
3.1.3 <i>Releasing the Camera</i>	4
3.2 SETTING CAMERA OPTIONS	4
3.2.1 <i>Basic Settings</i>	4
3.2.2 <i>Advanced Settings</i>	5
3.2.3 <i>Presets</i>	5
3.3 TAKING A PICTURE	6
3.3.1 <i>Showing the Viewfinder</i>	6
3.3.2 <i>Taking the Picture</i>	7
3.3.3 <i>Capturing the Image</i>	8
4 CONCLUSION	8
5 WEB RESOURCES	9
6 AUTHOR PROFILE	9

1 Introduction to the Onboard Camera API

As cameras became more widespread on phones, different APIs were introduced by the phone manufacturers to allow developers to access the camera functionality. Fragmentation began to occur, meaning that different camera-handling code had to be written depending on the phone platform and the individual phone's capabilities.

To resolve this, Symbian defined a common Camera API, called ECAM, which was first released in Symbian OS v7.0s. Additional functionality was added in subsequent versions of Symbian OS to cover the largest variety of camera hardware available and to provide a common API for developers to work with, irrespective of the hardware available on any particular Symbian OS-based device.

This paper outlines the facilities provided by the ECAM API, including some code examples to show how they are used. Note that the ECAM API provides only the definitions – it is still up to each manufacturer to implement the code for the API, and hence there may be some differences in behavior between manufacturers.

2 Onboard Camera API Facilities

The Onboard Camera API, ECAM, is an open, extensible, generic API for controlling simple camera devices on board a device. The API defines basic common operations, but can be extended by device manufacturers to include proprietary properties.

Documentation for the ECAM API is available in the Symbian Developer Library, as detailed in the Web resources section.

2.1 Functionality

ECAM offers the following main functions, which can be used by client applications (note that the availability of a function depends on the hardware capabilities of the device):

- camera settings: flash mode, contrast, brightness, zoom level
- image capture: transfer image to the client, image format, image size, clipping rectangle
- video capture: number of buffers, frames per buffer.

2.2 Base Classes

The key classes that make up the ECAM API are as follows:

- [CCamera](#) provides the base class for camera devices. This is the interface that the client application uses to communicate with the camera. It provides all the functions necessary to control and acquire images from the camera.
- [TCameraInfo](#) provides information about the camera. This includes: the version number and name of camera hardware, camera orientation and whether the camera supports still image capture.
- [MCameraObserver2](#) is an observer class which permits access to picture data as descriptors, bitmaps or chunks. It defines an event-handling interface that is called when events are generated from use of the camera APIs, or from external sources such as the camera being plugged in or unplugged.

- [MCameraBuffer](#) is the buffer class used by MCameraObserver2 to pass back view finder data, images and video frames. It presents an API for the client to choose in which way it accesses the picture data (descriptor, bitmap or kernel chunk).
- [TECAMEvent](#) is a simple event class, which contains a UID to identify the event and a status code. It is passed to MCameraObserver2 on completion of requests, when control of the camera switches, when camera hardware settings change, or due to external events such as the camera being plugged in. Some features in the extended API require extra information to be passed to the API client along with the error/event code. In order to do that, a derived version of TECAMEvent, called TECAMEvent2, is used.

3 Code Examples

Note that platform security requires that applications have UserEnvironment capability to create a Camera object.

3.1 Gaining Access to the Camera

The camera object class is CCamera, which provides the interface that an application uses to control, and to acquire images from, the camera.

To ensure that the phone supports camera hardware, use:

```
TIInt noOfCamerasAvailable = CCamera::CamerasAvailable();
```

Assuming that at least one camera is supported, a camera object must be created and the camera reserved for the application's use. With a member variable declared as:

```
CCamera* iCamera;
```

the object can be created and reserved as follows:

```
iCamera = CCamera::NewL(*observer, 0, 0);
iCamera->Reserve();
```

The parameters to NewL() specify, in order, a pointer to the observer class (see section 3.1.1), the index number of the camera (to allow for devices with multiple cameras) and a client priority value.

If camera functionality is not supported by the phone or emulator, the call to the NewL() method will leave with the KErrNotSupported error code.

If a higher priority client is already using the camera, the call to the Reserve() method will be unsuccessful – the HandleEvent() callback will return an error code.

Once the camera is available, the camera power must be switched on:

```
iCamera->PowerOn();
```

The callback interface is notified when power on is complete. (To avoid draining the battery, it is recommended that the camera be switched off when not in use.)

3.1.1 The Observer Class

The pointer parameter passed to the NewL() method must be provided, and is a callback to an observer mixin class. This callback is called when an asynchronous event completes, such as when an image has been captured.

The observer class provides callback functions to handle the following completion events:

- CCamera::Reserve(), when the camera is accessible
- CCamera::PowerOn(), when the camera has powered on

- the transfer of view finder data, indicating that a bitmap is ready for access
- `CCamera::CaptureImage()`, to transfer the image from the camera to the client
- `CCamera::StartVideoCapture()`, when a buffer has been filled with the required number of video frames.

The recommended callback interface to implement is `MCameraObserver2`, introduced in Symbian OS v9.4, with a number of additional methods to notify the client of new data (viewfinder data, captured image or captured video) or to indicate that a camera event (defined in `TECAMEvent`) has completed. Note that the older callback interface, `MCameraObserver`, is still supported, but which new clients do not need to use. (If only the `MCameraObserver` class is supported but the callback is to the newer `MCameraObserver2` class, the `NewL()` method will leave with the `KErrNotSupported` error code.)

3.1.2 Checking Camera Capability

The capabilities of the camera can be checked using the `TCameraInfo` class:

```
TCameraInfo cameraInfo;
iCamera->CameraInfo(cameraInfo);
TCameraInfo::TCameraOrientation orientation = cameraInfo.iOrientation;
if (orientation == TCameraInfo::EOrientationOutwards)
    { // camera points outward, use for pictures rather than video telephony
    ...
    }
```

3.1.3 Releasing the Camera

Once the application completes, any subclass objects must be released (in the destructor) as follows:

```
delete iCameraPresets;
delete iCameraAdvancedSettings;
```

(Presets and advanced settings are discussed in sections 3.2.3 and 3.2.2 respectively.)

Finally, the camera object must be released (in the destructor) as follows:

```
iCamera->Release();
delete iCamera;
```

3.2 Setting Camera Options

The camera can be configured in a number of ways.

3.2.1 Basic Settings

A number of more basic settings are available from `CCamera`.

Brightness and contrast, if available, are set in a similar way, in the range -100 to +100 or using a flag to use an automatic value:

```
TBool brightnessSupported = cameraInfo.iOptionsSupported &
TCameraInfo::EBrightnessSupported;
if (brightnessSupported)
    {
    iCamera->SetBrightnessL(CCamera::EBrightnessAuto);
    }
```

The camera's optical zoom level, if supported, can be set to any value between the maximum and minimum:

```

TInt minimumZoom = cameraInfo.iMinZoom;
TInt maximumZoom = cameraInfo.iMaxZoom;
if (minimumZoom != 0) // if supported
{
    iCamera->SetZoomFactorL(maxZoom);
}

```

The digital zoom value is set similarly, using `TCameraInfo`'s `iMaxDigitalZoom` and `iMinDigitalZoom` member data, and calling the `SetDigitalZoomFactor()` method.

Exposure can be changed as follows:

```
iCamera->SetExposureL(CCamera::EExposureNight);
```

`EExposureAuto` is the default value and is always supported; other exposure presets are given by the `CCamera::TExposure` enum, with `TCameraInfo::iExposureModesSupported` giving the ones that are supported.

The flash is set similarly – presets are given by the `CCamera::TFlash` enum and supported ones given in `TCameraInfo::iFlashModesSupported`. The default is `EFlashNone`.

```
iCamera->SetFlashL(CCamera::EFlashRedEyeReduce);
```

3.2.2 Advanced Settings

A number of camera hardware settings can be accessed using the `CCameraAdvancedSettings` subclass. Settings are identified by UID value, to allow for flexibility.

With a member variable declared as:

```
CCamera::CCameraAdvancedSettings* iCameraAdvancedSetting
```

the variable is initialized as follows:

```
iCameraAdvancedSetting = CCamera::CCameraAdvancedSettings::NewL(*iCamera);
```

Settings can then be applied. For example, to set the camera into burst shot mode:

```

if (CCamera::CCameraAdvancedSettings::EDriveModeBurst &
    iCameraAdvancedSettings->SupportedDriveModes())
{
    iCameraAdvancedSettings->
        SetDriveMode(CCamera::CCameraAdvancedSettings::EDriveModeBurst);
}
else
{
    User::Leave(KErrNotSupported);
}

```

Details of other advanced settings are in the Symbian Developer Library.¹

3.2.3 Presets

To simplify setting up the camera, the `CCameraPresets` subclass uses a single parameter setting, such as 'NightPortrait,' to set various advanced camera hardware settings:

```
iCameraPresets = CCamera::CCameraPresets::NewL(*iCamera);
```

The list of presets supported by the camera can be obtained as follows:

¹ See www.symbian.com/developer/techlib/v9.3docs/doc_source/reference/reference-cpp/ECAM/CCameraClass.html#::CCamera::CCameraAdvancedSettings.

```
RArray<TUi d> supportedPresets;
i CameraPresets->GetSupportedPresetsL(supportedPresets);
```

To check if a preset is supported, and set it if it is supported:

```
TUi d presetUID = KUi dECamPresetNi ghtPortrai t;
i f (supportedPresets. Fi nd(presetUID) == -1)
{
    User:: Leave(KErrNotFound);
}
el se
{
    i CameraPresets->SetPreset(KUi dECamPresetNi ghtPortrai t);
}
```

Changes to a preset cause an event notification, containing the UID of the preset, to be sent to the `MCameraObserver2` callback interface.

Details of other presets can be found in the Symbian Developer Library.²

3.3 Taking a Picture

There are a number of stages to actually taking a photograph. Initially the user has to be able to see through the viewfinder; then the image parameters need to be specified; and finally, the image needs to be captured. The following sections discuss these stages in greater depth.

3.3.1 Showing the Viewfinder

Once the camera power has been switched on, the user must be able to see the image that the camera sees, to decide when to take the photograph. This is achieved with a viewfinder. The camera implementation may include support for transferring viewfinder images into the client's display buffer, or the application may have to do this. (It's also possible that there's no support for either method.)

To determine which option to use, there are two settings in the `i Opti ons` member of `TCameraI nfo`:

- `EVi ewFi nderDi rectSupported` – frames are transferred directly from the camera to the display
- `EVi ewFi nderBi tmapsSupported` – the application must draw the viewfinder.

The following code determines which option to select, and calls the appropriate API (discussed below):

```
TRectscreenRect screenRect = i AppVi ew->Rect();
TSi ze si ze = screenRect. Si ze();

TCameraI nfo = cameraI nfo;
i Camera->cameraI nfo(cameraI nfo);

i f (cameraI nfo. i Opti onsSupported & TCameraI nfo:: EVi ewFi nderDi rectSupported)
{
    i Camera->StartVi ewFi nderDi rectL(. . . );
}
```

² See www.symbian.com/developer/techlib/v9.3docs/doc_source/reference/reference-cpp/ECAM/ecamadvssettings.hVariables.html#2.87.

```

else if (cameraInfo.optionsSupported & TCameraInfo::EViewfinderBitmapSupported)
{
    iCamera->StartViewfinderBitmapL(size);
}

```

3.3.1.1 Direct screen access

This is the most efficient method of displaying the viewfinder, as frames are transferred directly from the camera to the display by the camera subsystem, and can make use of hardware acceleration, if available. The application specifies the location of the display memory as the last parameter to the API call, as follows:

```

iCamera->StartViewfinderDirectL(iCoeEnv->WsSession(),
                               *iCoeEnv->ScreenDevice(),
                               *iAppView->DrawableWindow(), screenRect);

```

Once this call has been made, the viewfinder is shown to the user; even if the application loses focus whilst some of the viewfinder is visible, it will continue to be updated.

Use `iCamera->StopViewfinder()` to stop displaying the viewfinder.

3.3.1.2 Bitmap-based

If the direct screen access method isn't available, a less efficient method is to have the camera pass bitmap images of the viewfinder to the application at regular intervals, for the application to draw. The interval is usually sufficiently short to allow a fairly smooth display.

The API call `StartViewfinderBitmapsL()`, shown above, is fairly simple, but the application also needs to implement a callback method to receive and handle the bitmaps. The required callback depends on which observer class is used.

- `ViewfinderFrameReady(CFbsBitmap &aFrame)`
This method is the callback for `MCameraObserver`, and the bitmap is a standard `CFbsBitmap`. Note that the viewfinder frame must be drawn in the callback method if the bitmap isn't stored for later use, as the API reuses the bitmap.
- `ViewfinderReady(MCameraBuffer &aCameraBuffer, TInt aError)`
This method is the callback for `MCameraObserver2`. The bitmap is obtained from the `MCamera` class's buffer using the `BitmapL()` method as a single, unencoded frame in the form of a `CFbsBitmap`.

Use `iCamera->StopViewfinder()` to stop displaying the viewfinder.

3.3.2 Taking the Picture

Taking a photo involves specifying the format and size of the photo, and capturing a still image from the camera and transferring it to the application.

3.3.2.1 Setting the Image Format

It is possible to capture the image from the camera in a number of formats. The available formats are determined by `TCameraInfo::ImageFormatsSupported`, which is a bit field of values from `CCamera::TFormat`. These formats include:

- `EFormatFbsBitmapColorXxx` – an uncompressed format
- `EFormatJpeg` and `EFormatExif` – encoded formats that can be decoded using the ICL framework
- `EFormatXxBitsRGBXxx` and `EFormatYUVXxx` – raw data formats contained in a descriptor.

It's possible that a camera may support a number of formats, in which case the application should select one.

Having selected the required format, the application can enumerate the image sizes available and select the best one, as follows:

```
CCamera::TFormat imageFormat = CCamera::TFormat::EFormatFbsBimapColorXxx;

TInt si zel ndex;
RArray<TSi ze> si zeArray;
for (TInt i=0; i < cameraInfo. i NumImageSizesSupported; ++i)
{
    TSi ze si ze;
    i Camera->EnumerateCaptureSizes(si ze, i, imageFormat);
    si zeArray. AppendL(si ze);
}
si zel ndex = SelectBestSize(si zeArray);

i Camera->PrepareImageCaptureL(imageFormat, si zel ndex);
```

3.3.3 Capturing the Image

After successfully calling `PrepareImageCaptureL()`, the camera is ready to capture images. This is achieved via an asynchronous call to `iCamera->CaptureImage()`. When the image is available, the `MCameraObserver2` callback method `ImageBufferReady()` will be called.

The callback is passed an `MCameraBuffer` object, which can store the image data in a number of ways, depending on the image format selected:

- `EFormatFbsBimapColorXxx` – `MCameraBuffer::BitmapL()` is used to return a handle to a `CFsbBimap` containing the image data
- `EFormatJpeg` and `EFormatExif` – `MCameraBuffer::DataL()` is used to return a handle to a descriptor which can be saved directly to a JPEG file or decoded to a `CFsbBimap` using `CImageDecoder`
- for other formats, retrieve the data as a descriptor using `MCameraBuffer::DataL()` and process as required

Once the image has been processed, call the `MCameraBuffer`'s `Release()` method to release any memory it used and to allow it to be reused by the camera subsystem.

It is the responsibility of the application to display the captured image, if required. The viewfinder will not automatically stop – this is done by calling `iCamera->StopViewfinder()`.

To capture another image, call `CaptureImage()` again, either after the `ImageBufferReady()` callback has been received, or after the ongoing image capture has been cancelled by the `CancelImageCapture()` method

If `MCameraObserver2` is unsupported, the `MCameraObserver ImageReady()` callback method will be called. The image format used will determine whether the `CFsbBimap` or the `HBufC8` pointer parameter is valid; the pointer must be deleted after use.

4 Conclusion

Use of the camera hardware can provide some exciting opportunities for developers. This paper has outlined the basics; further details of single image capture and video capture using ECAM are available in the Symbian Developer Library and in Symbian Press's forthcoming [Multimedia on Symbian OS](#) book.

5 Web Resources

The Symbian Developer Library contains both guide and reference documentation, at:

- www.symbian.com/developer/techlib/v9.3docs/doc_source/guide/Multimedia-subsystem-guide/OnboardCameraGuidePA/index.html
- www.symbian.com/developer/techlib/v9.3docs/doc_source/reference/reference-cpp/ECAM/index.html.

There is a section on using the Camera API in the book *UIQ 3: The Complete Guide* by Holloway and Wright [Wiley, 2008]. For online content and more details, see:

- books.uiq.com/index.php/Main_Page.
- developer.symbian.com/main/documentation/books/developer/uiplatforms.jsp.

[Multimedia on Symbian OS](#) by Rome and Wilcox [Wiley, 2008] is forthcoming.

6 Author Profile



Mark graduated with a first class honours degree in Computing Studies, followed by a Masters in Digital Systems and finally a Postgraduate Certificate of Education. After six years of teaching and a spell at Morgan Stanley, he joined Psion Software in 1997 as a Technical Author working on SDK content and installation technologies.

After the formation of Symbian, Mark joined the Connectivity Engineering group, with sole responsibility for authoring, producing, delivering and supporting the Connectivity SDK. He also wrote a chapter in Symbian's first book, *Professional Symbian Programming*. In 2001 Mark moved to the Kits team, becoming Technical Architect shortly afterwards.

Mark transferred to the Symbian Developer Network in 2004, and is now responsible for the technical content on the SDN web site. He provides technical support to developers in the form of presentations, papers, books and tools and has given presentations in the USA, Brazil, Israel, India, the Philippines, Singapore, at 3GSM & at the Smartphone Shows.

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter](#).

Every month, the Symbian Community Newsletter brings you the latest news and resources for Symbian OS.