

Creating Energy Efficient Apps Using Qt

This article is designed to help developers create power efficient Qt code for use on Symbian devices.

Introduction

The Symbian platform has long been the leading mobile operating system in terms of resource management and is highly optimised for efficient use of battery power. A poorly coded application, however, can adversely impact the power efficiency of the whole system.

The introduction of Qt as the recommended development framework for applications on Symbian does not lessen the need for applications to be power efficient. Many of the Symbian C++ strategies for power efficiency still apply, but the mechanisms to implement them may differ. In addition, Qt will bring many developers who previously focused on desktop applications to Symbian. These developers may not have been concerned about power efficiency, and so may benefit from some additional guidance.

With this in mind, the goals of this article are to:

- Highlight the key areas of Qt code design where developers can minimise power consumption
- Give desktop Qt developers an appreciation of the issues to consider when coding for power efficiency.

The article achieves these goals by providing a collection of power conservation tips for application design and coding.

Code examples

This article is accompanied by three code examples which provide fully working apps that illustrate some key principles of energy efficient design and Qt coding. The examples are:

- iteration — this example shows how different mechanisms for traversing a list affect an app's performance.
- animations — this example shows two aspects of power efficient design: suspending processing when an app goes into the background and efficiently implementing drawing.
- accelerometer — this example shows how to use the Qt APIs for Mobility to make use of a device's accelerometers in a power efficient way.

The example apps are available in the [QtSymbianPowerEfficiency repository on Gitorious](#). In addition, archives of the code can be downloaded from the links at the top of this page.

Strategies for creating energy efficient apps

React appropriately to device and application status

Perhaps the most important aspect of conserving power is to ensure your application does not do any unnecessary processing. The most obvious case is when it's not visible to the user. It's therefore essential that your app monitors the device to detect inactivity, and stops unnecessary processing (such as animations) until it becomes visible again.

You should therefore consider suspending your application when:

- it's sent to the background.
- the device enters power save or standby mode.

The *accelerometer* and *animations* examples demonstrate a number of methods for monitoring whether an app is active. Specifically, to detect inactivity they listen for the following events:

- `QEvent::WindowDeactivate` — issued when an application goes to the background. This event occurs also when the screen is locked, the screen saver starts or the screen is turned off and the device goes into standby mode.

and to detect when to resume normal operation they listen for:

- `QEvent::WindowActivate` — issued when an application comes to the foreground. This event occurs also when the screen is unlocked or the device comes out of standby and the screen is turned on.

 Note: All Qt apps should listen for `QEvent::WindowDeactivate` and `QEvent::WindowActivate`, however a bug in Qt 4.6 on Maemo 5 - the Nokia N900 with firmware version 10.2010.19-1 (aka PR 1.2) - means `QEvent::Leave` and `QEvent::Enter` have to be used instead. The examples show how to use these alternative events, once the bug is fixed the code can be removed.

There are other events that provide signals that you can use to detect changes in your application's state or visibility. These include:

- `QEvent::Show`

- `QEvent::Hide`

These events are sent when the widget is explicitly shown or hidden (using `QWidget::show()` or `QWidget::hide()`). If the widget is hidden, there is no need to update the widget.

- `QEvent::FocusIn`

- `QEvent::FocusOut`

These events are sent when the widget is either gaining or losing edit focus. This can be used to activate and deactivate a blinking cursor used to indicate that input will go into a particular widget.

- `QEvent::WindowUnblock`

- `QEvent::WindowBlocked`

These events are sent when the window is blocked by a modal dialogue and when the blocking dialogue is closed. If the app shows a modal dialogue, updates could be suspended even if the window is visible behind the dialog.

- `QEvent::ApplicationActivate`

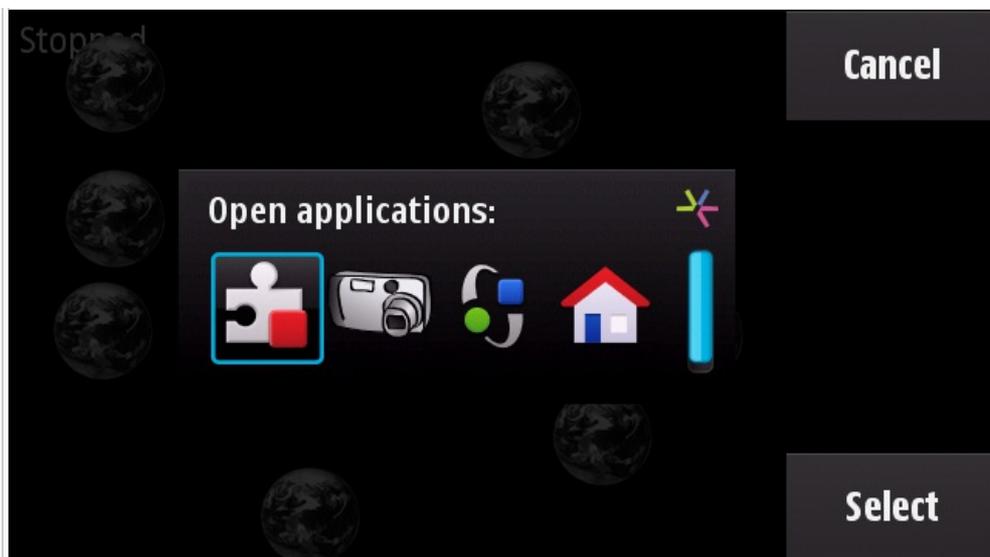
- `QEvent::ApplicationDeactivate`

These events are sent on Symbian devices when the user switches the app to the background and when the screen-dimmer activates.

For more information, see the [QEvent](#) documentation.

The *accelerometer* and *animations* examples illustrate how an application can be suspended when the app window receives a `QEvent::WindowDeactivate` event. They also contain (commented out) code to show how an app is restarted when it is moved to the foreground and receives the `QEvent::WindowActivate` event. This section of code from the *animations* example shows how these events are handled:

```
// The first show event will start the animation.
case QEvent::Show:
// Tap the screen to restart the animation.
case QEvent::MouseButtonPress:
// To restart the animation automatically, uncomment the following lines:
//case QEvent::Enter: // needed by Maemo
//case QEvent::WindowActivate: // needed by Symbian
    timer.start(20, this);
    text->setText("Started");
    break;
case QEvent::Leave: // needed by Maemo
case QEvent::WindowDeactivate: // needed by Symbian
    timer.stop();
    text->setText("Stopped");
    break;
```



The "animations" example, showing the animation stopped as a result of opening the task switcher.

In the example, processing is stopped by halting the timer used to trigger the updating of the balls' positions (and in the "accelerometer" example, to obtain the accelerometer values). Because the position of the ball is no longer updated the `QGraphicsScene` object stops painting to the screen as there are no changes to paint. As a result, all significant processing in the application stops. The effect is illustrated below. On the left is a profile of the energy consumption (obtained with [Nokia Energy Profiler](#)) of a version of "accelerometer" without the stop code — so it continues to run and update the screen even when the device is locked. Compare this with the example version that stops drawing when the device is locked. The "without stop" version reduces its power consumption by about half when the device is locked (because the screen is no longer lit) whereas the "with stop" version reverts to standby power consumption.



The effect on power consumption when a device is locked if the "stop" code is not included

These actions are generally applicable to any application. If you are using hardware, stop it (you may need to close and destroy the object that is accessing the hardware). If you are drawing to the screen, stop the processing that defines how the content of the screen changes and, if you are manually updating the screen content, stop the painting process too. See [Keep redrawing to a minimum](#) for more information on how to control drawing.

Use events instead of polling

If your application needs to react to a change of state within the device, such as the availability of a WiFi connection or receipt of an email, you have two implementation options:

- poll the resource each time a timer expires in order to see if there has been a change, or
- wait for a signal.

Polling consumes CPU cycles and battery life whether or not a change has occurred, and can result in a delayed reaction depending on the frequency of the polling cycle. More importantly, polling timers may prevent the device from entering its low-power or sleep mode. The event-driven paradigm overcomes the drawbacks of polling. In this paradigm, an event signal is issued only when something changes. The application threads are dormant while waiting for the signal. If all of the code in the system is dormant and waiting for some event, the device can enter its low power mode.

Fortunately the architecture of Qt is fundamentally event driven and you can find more information on the signalling mechanism in the Qt article on [signals and slots](#). Qt's event driven architecture means that by following good Qt programming practices you should end up making use of event signals where they are available.

There are many circumstances where timers are the most practical solution to design requirements. The key point here is not that you can't use timers, but that you ensure that timers are stopped when the application is in the background or there has been a period of user inactivity. If all the application is doing is waiting for timers to expire, then this will prevent the device going into a low power mode.

The *accelerometer* example app includes a slightly contrived illustration of this principal. The accelerometer does not provide a signal when its state changes, so a timer is used to check whether the device has been moved. However, if the user has put the device down there is no need to continually poll for accelerometer data, as there won't be any. So the example uses two timers, one to check for changes in the accelerometer data and another to kill this timer (and close the connection to the sensor) if the device appears stationary for a relatively long period. Setting this "inactivity" timer is shown below:

```
QAccelerometerReading *reading = accelerometer->reading();
qreal xacceleration = 0.0f;
qreal yacceleration = 0.0f;
if (reading) {
    xacceleration = reading->x();
    yacceleration = reading->y();

    // check if user inactive
    if (qAbs(xacceleration) < 1.0f && qAbs(yacceleration) < 1.0f) {
        if (!inactiveTimer.isActive())
            inactiveTimer.start(3000);
    } else {
        inactiveTimer.stop();
    }
}
```

Interestingly, the app uses the mechanism of signals and slots to send a signal from the expiring inactivity timer to the `stopAccelerometer` slot. `stopAccelerometer` then stops the timer that polls for accelerometer data and stops the connection to the accelerometer. Of course, the example also runs `stopAccelerometer` when the application is no longer in the foreground.

In summary, use events where they are available and if you use timers, kill them when they no longer serve a useful purpose.

If you wish to make use of timers the blog post [When being wrong is right](#) is well worth a read, to be aware of some improvements expected in Qt 4.8.

Optimise your use of graphic content and effects

Graphics, animations, and video can be expensive to process, so avoid unnecessary complexity.

Using graphics in Qt

Implement just enough UI graphics, but no more

Users expect increasingly rich UI features such as animations, transparency, gradients and other similar UI effects. These effects have an impact on power consumption and you should think carefully about how and when you implement them:

- Does an effect add to the overall user experience? Don't use effects simply because you can. Ask yourself whether they inform the user of something or simply look "pretty". For example, a button which is animated after it's pressed or while a process starts is helpful, a button with an animation that runs permanently might not be.
- Ensure that any effects are turned off when they are not visible to the user, such as when the device is locked or the app is in the background.

Consider offering users the option of turning off dynamic UI effects so that they can decide whether appearance or longer battery life is their priority.

Understand how Qt draws graphics

The first step to creating efficient graphics in Qt is, rather obviously, understanding how Qt handles graphics. An excellent discussion of graphics in Qt is provided by [Gunnar Sletta's blog](#) series, which comprises:

- [Qt Graphics and Performance - Whats Hot and whats Not](#)
- [Qt Graphics and Performance - An Overview](#)
- [Qt Graphics and Performance - The Raster Engine](#)
- [Qt Graphics and Performance - The Cost of Convenience](#)
- [Qt Graphics and Performance - Fast Text](#)
- [Qt Graphics and Performance - Generating Content in Threads](#).

Make the best use of hardware accelerated graphics

Hardware graphics acceleration has the potential to offer power efficiencies compared to graphics processing in the CPU. This is because a GPU requires fewer cycles to perform graphics rendering. Although the purpose of GPU hardware is to enable more sophisticated graphics, rather than to reduce power consumption, ensuring your graphics make use of this hardware can offer power efficiencies in apps with low or modest use of graphics.

Qt uses [QPainter](#) for all graphics operations. This in turn uses a platform specific engine implemented in the [QPaintEngine](#) API to do the actual painting when a GPU is present, or [QRasterPaintEngine](#) otherwise. When [QPaintEngine](#) encounters paint commands not handled by the GPU it falls back to using [QRasterPaintEngine](#).

On Symbian devices Qt automatically selects the best approach to graphics rendering. However, if a GPU is present on the device you can use the [QOpenGL](#) module to access it directly.

Keep redrawing to a minimum

Redrawing your user interface can be a costly exercise, particularly when not much of it has changed. To minimise unnecessary energy consumption your app should make the best use of the Qt features for efficient redrawing.

You can use the [ViewportUpdateMode](#) flag to control how [QGraphicsView](#) updates its viewport when the content of a scene changes or the content is exposed. The flag provides four automatic modes and manual one. In the automatic modes [QGraphicsView](#) assesses the nature of the changes and updates the view accordingly. The options range from simply redrawing the view after any change to a smart option where [QGraphicsView](#) looks for the optimal update strategy. The challenge when selecting which option to chose is balancing the processing required to decide what to redraw with the processing saved by not redrawing everything. The wrong choice could offset any advantage from minimising redrawing with an increase in processing to determine what and when to draw. You can achieve more granular drawing with [QGraphicsItem::paint\(\)](#) — which paints the contents of an item using local coordinates — and [QGraphicsItem::update\(\)](#) — which schedules a redraw of the area covered by a rectangle.

The *animation* and *accelerometer* example apps make use of [QGraphicsView](#) to handle drawing.

In some cases the best strategy is to control the redrawing of your widget manually, based on knowledge of your app's functionality. Reimplement [QWidget::paintEvent\(\)](#) in a subclass to receive paint events — requests to repaint all or part of a widget - and then use [QWidget::update\(\)](#) to update a rectangle inside the widget.

 Tip: You can use the environment variable `QT_FLUSH_PAINT` to help you optimise your screen drawing. This feature highlights in yellow the areas of the screen that are updated. It's only available on applications built for the desktop development

environment. To use the feature set `QT_FLUSH_PAINT = 1` and start the app from that environment or enter `set QT_FLUSH_PAINT = 1` myapp at the command line.

Use of transparent windows

For power efficiency avoid semi-transparent windows, as updating and resizing them is significantly more processor intensive than opaque windows. If you wish to use semi-transparent windows refer to the Qt documentation on [Transparency and Double Buffering](#) and [Window Opacity](#).

Reducing graphics quality

Another strategy for optimising graphics to reduce power usage is to reduce the quality. Qt enables you to control the method of antialiasing, and all the functions for drawing primitives have both floating point and integer versions.

There are also [OptimizationFlags](#) which enable you to control some subtle paint-state and antialiasing features.

Reducing graphics quality, however, should be used with care as the negative impact on usability may not be worthwhile for small power savings.

For more information see the [Rendering Quality](#) section of the `QPainter` documentation and the Qt [Concentric Circles Example](#).

Match content quality to device and use case

Rendering or playing back audio, Flash, SVG, video or similar content can be particularly power intensive, especially where the device needs to scale the content in some way to match its capabilities.

Bitmap images

If an image is only ever displayed at 70 x 70 pixels there is no point in delivering a larger version and then relying on the app to scale it down. Similarly, where thumbnail versions of large images are used to navigate some form of image gallery or list, create and store thumbnail images rather than creating them on-the-fly each time they are needed.

SVG graphics

You may assume that an SVG graphic is inherently efficient, however there are hidden dangers.

Firstly, if an SVG file is used to create an image that is used repeatedly at the same size, it's worth converting it to a bitmap and caching the result for subsequent draws.

In addition, there is a potentially serious hidden danger in the content of SVG files. Depending on the mechanism you use to draw and convert your source graphic to SVG, there is the possibility that the resulting file may contain hidden or invisible elements. When the rendering engine processes the file it processes all the elements in the file, whether or not they contribute to the final visual effect. SVG files have been found to contain hundreds of ultimately invisible elements, all processed with a resulting impact on performance and battery life.

Therefore, you should always review your SVG files and optimise them so that only the elements essential to creating the final graphic are included. There are some tools available to help with this process, such as [Scour](#), though, as far as we are aware, none have been specifically tested with SVG content for Symbian devices. The only other option is to review the SVG file by hand and remove unnecessary elements.

Video

When including video in your application optimise the video resolution to match the screen and consider whether the frame rate can be reduced.

Audio

Consider the user experience associated with the content and reduce quality to match the desired effect. For example, in a game a sound with a low sampling rate may be as effective as full HiFi quality sound.

Reserve hardware resources when they are needed

The most significant impact on power consumption comes from the many hardware features found on a typical Symbian smartphone. The display is an obvious power drain, as is any hardware which transmits or receives radio signals. Newer features such as accelerometers and magnetometers have a power impact too, particularly when handled inefficiently. Other less-obvious resources can also have an impact on power consumption. For example, allocating more memory than needed may cause the device to keep banks of memory active that it could otherwise power down.

To put things in perspective, the following table shows some typical power consumption requirements for key hardware components.

Hardware	Typical power consumption (mW)
LCD display	190-360
OLED display	40-500
GPS	≈50
WLAN	1000-1600
Bluetooth	<1mW for sniff mode 10s of mW for active mode
Accelerometer	
Magnetometer	

A large capacity battery stores approximately 4.5 watt-hours of energy. From the table above we can see that inefficient use of a WLAN connection could drain a battery in under three hours. With the screen active too the battery life could be as little as two hours. Even hardware with apparently modest power requirements can have a large impact: the GPS only draws tens of milliwatts but if your application updates the screen each time the GPS reports a new position the resulting battery drain could be much higher.

Therefore from a power efficiency standpoint it makes sense to only turn on hardware when it's needed, and to turn it off again when it's no longer needed. The following sections discuss this approach in more detail.

Just-in-time resource use

Create (construct, load etc.) resources just before they are required and close or destroy objects as soon as they are no longer required, even if the same resource might be required again shortly. Opening a device's accelerometer, for example, powers on the hardware and leaves it running even if your app is not actively using accelerometer information.

Releasing resources — particularly hardware components, such as Bluetooth radio, cellular radio and hardware accelerators — ensures that power management features can be triggered as early as possible.

Time out idle resources

We've [already discussed](#) how you can use events to trigger power saving actions when the device or app changes status.

On Symbian a device the background event triggered by the screen saver is an important way to identify the need to close hardware, as it provides a reliable method of identifying that the user is no longer actively using the device. However, it's important to consider whether you could close hardware resources before receiving this trigger. This is particularly relevant if you are creating cross-platform apps.

For example, an app could provide a view that offers information on share prices. The app may initialise radio communication as the view opens and close communication when the view closes, on the assumption that the user will only remain in the view for a short period of time. However, if the user remains in this view for long periods, without needing the content of the view updated, the app could consume power unnecessarily by keeping the network connection open. Suspending communications may therefore be appropriate once the information for the view's contents has been created.

This strategy is used in the *accelerometer* example app. For more information, see [Use events instead of polling](#).

Override screen, screen saver and backlight only when necessary

In general the best strategy is to not override the device's backlight and screen saver settings, because they work automatically according to preferences set by the user.

In special cases, however, overriding the settings is understandable (for example, in a navigation app). However, even with the settings overridden, consider turning the display and backlight off during lengthy operations, for example, during downloading (partial display mode can be used to show a progress bar without backlight) or other processing that takes a long time. In the case of a navigation app, consider whether the screen should be dimmed or turned off between navigation points.

Optimisation

Following good power efficient practices in your application design and coding should deliver a power efficient application.

However, the complexity of the underlying hardware and software and the way in which the application is used can result in

unexpected issues. Should this occur you might consider optimising your application by looking for power issues and resolving them on a case-by-case basis.

There is a danger, however, that in optimising your implementation for one device you de-optimize it for another. The best advice is therefore, beyond ensuring you make sensible use of the hardware and follow good general programming practices, to optimise only when a significant power issue is found.

To assist with this optimisation work, this section provides some guidance on things to consider.

Hardware specific considerations

One of the challenges in creating power efficient code is the diversity of hardware used in Symbian devices. Not only do different manufacturers use different hardware, but even within a single manufacturer's range there may be different radio, sensor and GPS modules. And even when the hardware is the same there may be different drivers.

As a result it's impossible to offer fine grained advice on the most power efficient use of hardware. For example, when using a GPS it's possible to make either a one-off request for location or open the GPS channel and set it to provide periodic updates. To choose which method is the more efficient it's necessary to know how a long periodic update compares to issuing one-off requests timed by some other means. Unfortunately the answer depends on the hardware and how it has been implemented.

Radio receivers and transmitters

These include cellular radio, WLAN, and Bluetooth hardware. (GPS is strictly a radio receiver but is discussed separately.) In addition to ensuring your app opens these resources when they are needed and closes them when idle or no longer required, consider:

- optimising the size of data transmission, for example to ensure that only necessary data is transmitted
- caching data that may be reused and which changes infrequently
- segmenting data items so that only the most commonly used data is always transmitted, and optional or infrequently use data is only transmitted on request. A typical example is email where details of the sender and message title are always transmitted, the message body is transmitted only when the email is opened, and attachments are only transmitted on request
- avoiding protocols that require polling
- TCP rather than UDP since it requires fewer keep-alive messages. This is especially important for always-on apps
- bursty data transfer rather than constant "drop-by-drop" transfer.
- enabling WLAN power save in the device and avoiding unnecessary background scanning.

GPS

Location based apps can be particularly problematic from a power consumption point of view. Not only are the GPS modules themselves relatively power hungry, but these applications are often provided for real-time tracking and therefore can keep the display active for long periods. They may even share the information over the network - a triple hit in power consumption.

Location information is provided to Qt apps by `QGeoPositionInfoSource`. You can specify whether its `PositioningMethod` as satellite, non-satellite, or any positioning method. If your app does not require particularly accurate location positions the non-satellite option can offer a power efficiency benefit, as the GPS hardware is not involved. However, if another app has already started making location requests with the device's GPS, it will be used in your app too. If your app does not require particularly frequent location updates consider turning off GPS between requests.

Sensors

Sensors, such as accelerometers and magnetometers are relatively new hardware innovations in mobile devices. Generally their power optimisation is not as advanced as that for radio transmitters and GPS. As a result, it's best to assume that opening a channel to such a sensor powers on the sensor hardware and the hardware remains powered on regardless of whether sensor information is being requested or not.



Tip: Many Symbian devices are designed to be used with the screen in landscape and portrait modes. In general, an application that is set to run in full screen mode will automatically adjust if the device is able to report changes in orientation or device configuration, such as sliding out a keyboard. However, you may wish to manually control the screen layout for landscape and portrait modes. In this case the using the accelerometer is not the best method of detecting orientation changes. The more effective and power efficient method is to react to use `resized` signal of [QDesktopWidget](#).

The behaviour of the sensor channels means that it does make sense to instantiate, start, and stop the sensors in a timely manner.

In the *accelerometer* example the accelerometer is started when the app first starts or when the screen is tapped (after the

accelerometer has stopped) and the code can be modified to automatically restart when the app comes to the foreground as follows:

```
bool View::event(QEvent *event)
{
    switch (event->type()) {
        // The first show event will start the animation.
        case QEvent::Show:
            // Tap the screen to restart the animation.
            case QEvent::MouseButtonPress:
                // To restart the animation automatically, uncomment the following lines:
                //case QEvent::Enter:          // needed by Maemo
                //case QEvent::WindowActivate: // needed by Symbian
                    startAccelerometer();
                    break;
            case QEvent::Leave:          // needed by Maemo
            case QEvent::WindowDeactivate: // needed by Symbian
                stopAccelerometer();
                break;
        case QEvent::Resize: {
            QSize s = static_cast<QResizeEvent*>(event)->size();
            bounds->setRect(0, 0, s.width(), s.height());
            updateBallPositions();
            break;
        }
        case QEvent::Timer:
            if (static_cast<QTimerEvent*>(event)->timerId() == timer.timerId())
                updateBallPositions();
            break;
        default:
            break;
    }
    return QGraphicsView::event(event);
}
```

For more information on detecting when to suspend an application, see [React appropriately to device and application status](#).

Camera

In general most apps will only use the camera as it's needed to capture images or video. Therefore the general guidelines in terms of resource usages should be applicable to most apps.

There is, however, an increasing interest in offering forms of gesture control in apps based on, for example, the user passing a hand over the camera or detecting changes in light level. Where possible, it's recommended that the ambient light sensor is used for these tasks.

Screen

Mobile device broadly use two screen technologies: LCD (Liquid Crystal Display) or OLED (Organic Light Emitting Diode). LCD is the older technology, with OLED displays becoming common on higher end devices. The technology trend will probably see OLED as the prevalent technology within the next couple of years.

The principal difference between the two technologies is that OLEDs create light directly while LCDs require a backlight to make them usefully visible. Because OLEDs generate light the amount of power they consume varies greatly, ratios of 1:14 have [been observed](#) between the power consumption of a black screen compared to a white one.

LCDs exhibit variations in power consumption between similar black and white screens, however the variation is closer to 1:2.

Fortunately this means the same rule can be applied to the creation of UIs: the use of darker colours will minimise power consumption on any device.

In general, you are encouraged to implement your apps so that they use the device's default theme as much as possible. This is done automatically by [QS60Style](#). This means that some of the impact of a UI design on power consumption is devolved to the theme and in turn to the user. However, for apps where the screen may be active for long periods, such as a game or navigation app the following UI and app design strategies should be considered:

- Overriding the default theme to use dark backgrounds and light foregrounds in app views. This is best achieved by [creating a new palette](#) (Try to avoid [Qt Style Sheets](#) as they are known to be power inefficient.)
- Switching the display off when it's not providing useful information, for example, between turns in a navigation app.
- Ensuring any graphics, such as those used to implement a game, use predominantly dark colours with light colours used as highlights.

Create efficient code

Good programming practices are the foundation of power efficient applications. The fewer CPU cycles your code requires to achieve a particular outcome the less power it consumes.

In addition to good C/C++ programming practice, there are some Qt-specific aspects of best practice that you should follow. There are also coding approaches that you should avoid.

Some typical examples include:

- Traverse lists and access list items appropriately:
 - Prefer the most power efficient iterator — Qt provides several different methods for traversing a list, such as STL style iterators and a `foreach` macro. However, the performance of the `foreach` macro can be significantly [slower than other methods](#).
 - Prefer [QList::at\(\)](#) for read-only access to list items. Using `at()` can be faster than [QList::operator \[\]](#), because it does not cause a deep copy (duplication of the list data) to occur.
 - Use `const` iterators where elements are not modified.

Consider the two code fragments below. In both examples the code iterates from the beginning to the end of the vector and assigns the value of `qreal` to either `foo` or `bar`. The first fragment uses a non-const iterator as follows:

```
QVector<qreal>::iterator it = container.begin();
while (it != container.end()) {
    foo += *it;
    ++it;}

```

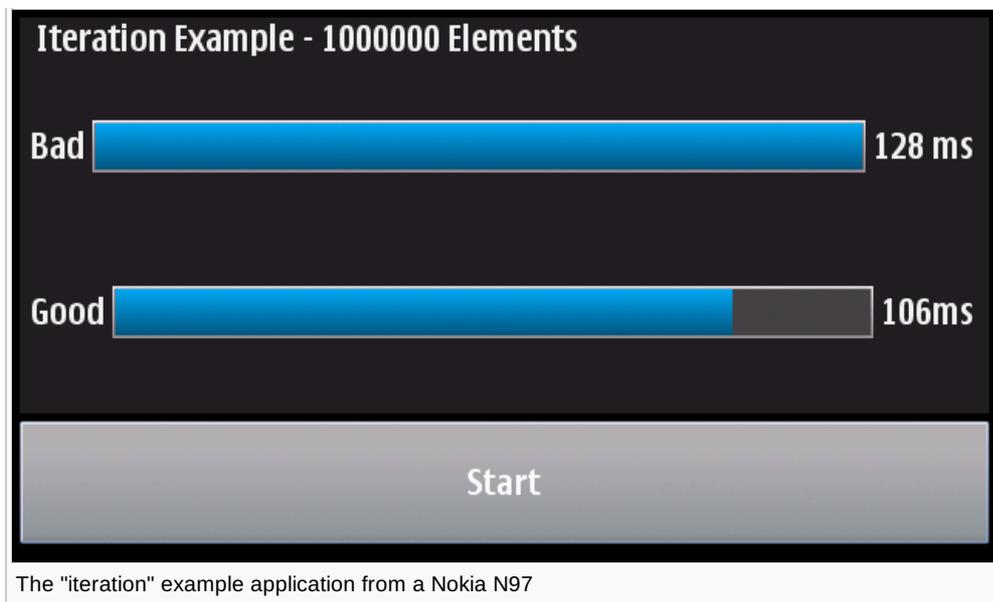
The second method uses a const iterator:

```
QVector<qreal>::const_iterator it = container.constBegin();
while (it != container.constEnd()) {
    bar += *it;
    ++it;
}

```

The difference between the execution time for the two methods can be quite significant, as illustrated by the table below which shows the approximate execution time of the iterator example on various devices. The compiler is able to optimise more effectively when it's explicitly told to access the elements using a `const`.

Device	const iterator	non-const iterator
Nokia N97	106 ms	128 ms
Nokia N97 Mini	109 ms	129 ms
Samsung i8910	60 ms	73 ms



- Avoid instantiating overly large objects — amazingly, examples have been seen of developers trying to instantiate items such as images containing a trillion (10^{12}) pixels each with 32-bit colour depth. Such objects require a lot of processing!
It's worth noting that any excessive memory use, which does not cause your app to fail with an out of memory error, can consume more power than necessary and slow down the device as code needs to be paged in more frequently.
- Cache appropriately — ensure that regularly used items or values, such as image thumbnails or values for items such as font metrics or file info, are cached. Recalculating or recreating them each time they are needed may consume unnecessary processor cycles.
- Use const functions in Qt container classes — using non-const functions may cause the container to detach, that is, cause a copy of the container to be created.

The above list is by no means exhaustive and there are many useful resources that provide guidance on efficient Qt coding. The following list should provide a useful starting point:

- [How-To's and Best Practices](#) — a collection of guides provided as part of the Qt 4.6 documentation.
- [The Little Manual of API Design](#) and [\(Qt\) API Design Principles](#) — these guides discuss the key considerations in designing an effective API, focusing on the Qt API. While much of the content is about API usability, rather than run time efficiency, they provide many useful hints that are worth considering in your own coding.
- [\(Qt\) Coding Conventions](#) — useful information for effective coding of Qt apps.
- [Ensuring Maximum Performance with Qt](#) — this paper provides background information on how Qt behaves and guidance on using `QtBenchLib` to add performance tests to your test suites. Using `QtBenchLib` can provide information that helps you identify where you need to focus your efforts.
- [Qt Developer Network](#) — at the time of writing (July 2010) this resources is still in beta, however it should be a valuable source of information on Qt development.
- The [Qt Quarterly](#) — this publication, containing high-quality technical articles written by Qt experts, is designed to give developers added insight into Qt programming.
- [Qt Labs blog](#) — regular blogs from the experts within Qt. In addition to general information on developments in the Qt framework, articles often offer insight into good programming techniques.

The Qt documentation also includes information on functions that are inherently slow, for example `QWidget::setMask` which can be slow if the masked region is particularly complex. There are also a number of books on Qt development that provide invaluable insight into the best ways to code in Qt. Of these we would recommend:

- [C++ GUI Programming with Qt 4 \(2nd Edition\) - The official C++/Qt book](#), by Jasmin Blanchette and Mark Summerfield.
- [Foundations of Qt Development](#), by Johan Thelin

You can find a longer, regularly maintained list of books on Qt [here](#).

File access

When reading or writing to a file, few larger data transfers will be more efficient than many smaller transfers. Also linear access will be more efficient than random access.

Recommendations:

- Use `QFile` in buffered mode (don't use the `Unbuffered` flag on `open()`). Using the buffered mode will reduce the number of

read/write operations that go to the disk.

- Don't `flush()` a file unnecessarily when writing. A write flush forces data to be written to the disk, which may require a much larger chunk of disk to be erased and rewritten than you may think is being written.
- Try to avoid frequent long `seek()` jumps in a file. This will counteract the benefits of buffering.
- If a file structure does require many `seek()` calls to navigate its contents, where appropriate consider reading the whole file into a `QByteArray` or `QBuffer` and navigating the array or buffer.
- If a file structure does require many `seek()` calls to create its contents, where appropriate consider writing it to a `QBuffer` first, then committing the entire content of the file structure (buffer) to a `QFile`.

Similar considerations should also be made when using databases.

Give the user low battery options

While not a strategy for ensuring your app is power efficient, you may want to consider monitor the battery power status and reacting to changes in the battery's power reserve. You can do this by registering to receive the `QSystemDeviceInfo::batteryLevelChanged` signal. Using this signal you can obtain `QSystemDeviceInfo::BatteryStatus` that will let you know if the devices battery is <3%, <10%, <40% or >40% full.

If the nature of your app means it has a high power consumption, you may consider offering the user the option to close the app or advise them to connect a charger if the battery reserve drops below a particular level.

Design vs. Testing

Design for efficient power usage will always be the most cost effective strategy, however, even a well thought out design can have unexpected power consumption consequences. As we mentioned in the section on [Optimisation](#) the variation in hardware between devices can make your app relatively power friendly on one device, but not so on another. The only reasonable strategy is to find the best compromise performance across a range of devices. However, there may be power impact arising from the consequences of unexpected user interaction with the app. This simply reinforces the recommendation to test your app as widely as possible before a full commercial release.

Testing tools

A fairly obvious question, when it comes to testing for poor power performance, is how you identify that a problem exists and the code in your application that is causing the problem. There are a couple of tools you can use, however, at present they are limited to use on Symbian devices from Nokia. These tools are Nokia Energy Profiler and Carbide.c++ Performance Investigator. There are other solutions, but these are mainly designed for device manufacturers.

Nokia Energy Profiler

The Nokia Energy Profiler is the most accessible tool for testing an application's energy usage. In fact it's delivered as a consumer application through [Nokia Store](#). Usable on most Nokia S60 3rd Edition, Feature Pack 1 and later devices, Nokia Energy Profiler provides information on:

- Power consumption, as well as battery voltage and current.
- Cumulative power consumption.
- Processor activity.
- RAM use.
- IP-network speeds.
- Mobile-network signal strengths.
- 3G Timers.
- WLAN signal strength.

To use Nokia Energy Profiler you install and run the application on a device, then start the capture process. With the capture running, exercise your application, then return to Nokia Energy Profiler, stop the capture and examine the information recorded. The application provides graphs that plot the various captured values against time. You can export this information as CSV files, for analysis in a suitable spreadsheet application. In addition, NEP provides for capturing screenshots while it's recording, which is helpful as a way of identifying what your application was doing at the time of excessive resource usage. There are two basic testing scenarios you should consider when using Nokia Energy Profiler:

- Standby power consumption testing. In this test scenario you first use Nokia Energy Profiler to determine a baseline for your test device's standby power consumption. To do this, ensure no other applications are running and network connections closed. Then run a capture in NEP and lock your device. This will provide the baseline standby power consumption. Now run

your application and at various points in its use start a NEP capture and lock the device. Compare these captures with the baseline and look for any significant differences.

- Active use power consumption testing. In this test scenario you make use of your application, exercising various different use cases, and look for any unexpected spikes in power consumption. When issues are identified, the testing use cases should be refined until the code that is causing the problem is identified.

Nokia Energy Profiler can be downloaded to a Nokia device from [Nokia Store](#).

An alternative solution for this type of testing is the Performance Monitor application.

Carbide.c++ Performance Investigator

An option for more rigorous analysis of an application's performance is provided by Performance Investigator Profiler.

Performance Investigator (PI) has the advantage over Nokia Energy Profiler of providing explicit information on the code that is running on a device. This enables you to identify more readily the code that is causing energy consumption or other issues, such as out of memory.

PI however, is part of the Carbide.c++ toolset and requires you to install Carbide.c++ as well as a profiling client on your device. If you are using the Qt tools for development, the additional effort in using PI may be worthwhile only if you encounter problems you cannot resolve using the information provided by Nokia Energy Profiler.

A guide to using PI is provided in the article [Analyzing Application Performance with the Carbide.c++ Performance Investigator](#).

Other options

Performance investigation can be undertaken with tools such as the [Arm Profiler](#), however these tools generally need specific hardware or development boards to enable their use.

Other sources of information

[Symbian C++ Performance Tips](#) — This Symbian wiki article provides tips on coding for app performance in Symbian C++. These tips don't directly relate to power efficient coding, however, in general more efficient code will require less CPU cycles and therefore less power to execute. Many of the tips are general in nature and the principles discussed can be applied to Qt coding.

Code Examples

- [Media:Energy Qt iteration.zip](#)
- [Media:Energy Qt animations.zip](#)
- [Media:Energy Qt accelerometer.zip](#)

Conclusion

Ensuring your application is power efficient is an essential part of the overall design and it could easily be a make or break issue: after all no app, however functional and good looking will find favour if it contributed to draining a device's battery too quickly.

In this article we have looked at a number of strategies for creating a power efficient app. The most important of these is to ensure that your application stops unnecessary processing when it's the background. We also reviewed a wide range of other practices that can help contribute to energy efficiency, such as appropriate use of timers, approaches to graphics and the use of hardware, as well as good C++ programming practices.

By following this advice you will go a long way towards ensuring your app does not get labelled as a battery eater. However, creating a power efficient app is not a precise science. Variation in hardware and user behaviour can have unexpected results. So while good design is the best starting point ultimately testing and optimisation are just as essential.



© 2010 Symbian Foundation Limited. This document is licensed under the [Creative Commons Attribution-Share Alike 2.0](#) license. See

<http://creativecommons.org/licenses/by-sa/2.0/legalcode> for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

