The best of both worlds

Mixing open source and closed software can prove to be an effective – and profitable – corporate strategy. Philips is one company that has come to understand this

By **Arnoud Engelfriet**

Over recent years, the amount of software in products has risen dramatically. Not only do devices today contain more of it, but software also realises a larger part of the functionality. This explosive growth has driven up the cost of software development. Outsourcing some or parts of this development provides a partial solution. And the use of open source for the right parts can provide significant additional cost savings and a reduction in product development times.

Open source

Software packages such as the Linux operating system, the Firefox web browser or the Apache web server software are the most well-known examples of open source. More generally speaking, the term open source refers to a software development model by which the source code to a computer program is made available publicly under a licence that gives users the right to modify and redistribute the program. Users are expected (although not required) to make their modifications and improvements available for inclusion in the official distribution.

The use of open source software is not just restricted to software companies or consultancy firms (see "A sharing, caring IBM" in the April/May 2006 issue of *IAM*). Consumer electronics products and other devices also increasingly use open source. For example, Linux is the most popular choice as embedded operating system today. A good introduction to the business benefits open source software can bring is Martin Fink's book *The Business and Economics of Linux and Open Source* (Prentice Hall, 2002).

Understanding open source risks

The licence conditions applicable to open source can be quite peculiar. For example, some licences require one to release one's own software as open source software itself, if that software incorporates the open source software. If open source software under such a licence is used throughout the software stack in a product, all software for that product may have to be published as open source. For patent holders, there is an additional risk that one may have to give a free patent licence when distributing third-party open source software that infringes on one's patent. This patent licence could be limited only to the open source in question, but in some cases even to the software stack as a whole.

Because of these risks, a company may be tempted to avoid open source software altogether. This, however, is not a realistic option from a business point of view. The use of open source software is gaining more and more popularity in commercial environments and even in commercial products. By ignoring this, a company locks itself out from all the available high-quality software and does not benefit from the reduced costs and time to market that open source software implies. In some cases it is simply not feasible for a commercial product to keep up feature-wise with open source alternatives. The only viable option, therefore, is to understand the risks and how to manage them.

There are over 40 different open source licences, each with its own conditions and implications. Roughly they can be classified into these three categories:

- Free-for-all licences: these licences only require licensees to give credit to the

original authors. Derivative works can be kept proprietary. Sometimes these licences are referred to as "academic licences". Examples are the so-called BSD and MIT licences, as well as the licence used for the Apache Web server.

- Keep-open licences: modifications to software under these licences have to be made available as open source as well. Larger works incorporating such software can be kept proprietary. The GNU Lesser GPL (used for Linux system libraries) and the Mozilla Public Licence (used for the Firefox Web browser) are keep-open licences.
- Share-alike licences: when software under such a licence is modified or extended, the result as a whole has to be made available as open source. The term 'copyleft' is sometimes used to characterise this kind of licence. The most famous example is the GNU GPL, which applies to the Linux operating system. Another example is the Open Software Licence (OSL).

A company may be tempted to avoid having to share software it considers proprietary by disallowing use of share-alike and even keep-open licensed open source. This will severely limit the ability to use open source at all. About 65% of all open source software is covered by the GNU GPL, with an additional 20% being covered by the GNU Lesser GPL. These two licences thus cover virtually all key components of devices with embedded software. Hence an open source policy should be about where and how, not if, the different types of open source can be used in a product.

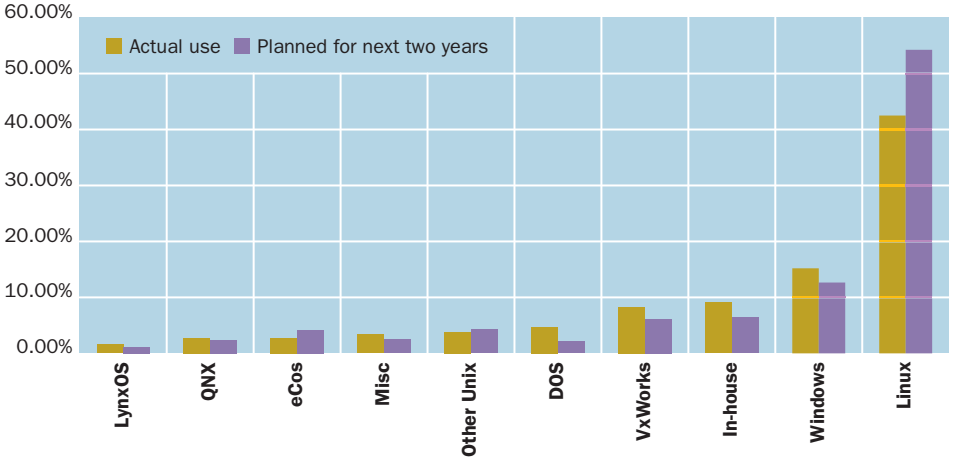
Open or closed?

Companies that use open source often make a purely black or white decision: either a product is open source or it is not. Much greater benefits can be obtained by using a more subtle approach. A company can use open source for certain features and use closed, in-house developed or commercially licensed software for other features if the above-mentioned licence implications are properly managed. The ultimate goal should be to ensure the product offers the most value. Therefore, a proprietary feature or solution should be replaced with an open source alternative when doing so represents a larger value than keeping the feature proprietary or closed. Of course, instead of using an existing open source implementation, a company could also release its own implementation as open source.

Value can be obtained through different strategies. When a feature is kept closed or proprietary, the feature can serve to differentiate the product from competing products. Alternatively, the feature can be licensed to others for a royalty fee for example, as a software library or a chip with embedded software. Patents related to the feature can be licensed royalty-bearing as well. A consequence of this approach is that the feature now must be written in-house or be licensed-in from a third party, which increases product cost. Using open source software for a feature does not carry software licence costs, although in some cases, for example when the feature implements a royalty-bearing standard, a patent royalty may still be due. The software can be adapted if necessary and there is no dependence on a particular vendor. But licensing the software, or any relevant patents, for a royalty is no longer possible.

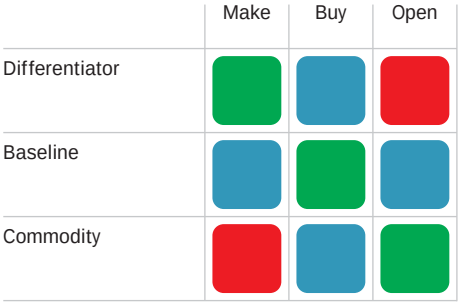
To find the right strategy, keep in mind an old rule in economics: make the complement of your own product a commodity. A complement is needed to make your own product useful. Digital music complements portable music players, for example. Commoditising your complement actually stimulates demand for your own product. If digital music becomes easier to purchase online, demand for music players goes up. This not only applies to complete products, but also to components or features of a single product. An advanced digital television may have the ability to generate

Embedded operating system usage



Source: LinuxDevices.com Embedded Linux Market Survey 2005

Figure 1
The make / buy / open source decision for different product features



automatically one-minute summaries of available television programmes. The ability to decode and play digital video streams is essential to make this feature useful, and so video decoding is a complement to automatic summary generation.

Often commoditisation goes hand in hand with standardisation. Once a feature is standardised, anyone can build an implementation by following the standard. Soon afterwards the feature becomes a commodity, as there now are many more or less interchangeable implementations available. In the examples above, it is in the interest of the television and music player manufacturers to ensure digital content formats are standardised.

In principle, open source is appropriate for those features that complement the differentiating features of a product. But using open source for all non-differentiating components can be a step too far. Some non-differentiating features provide more value when closed than when open, and so should remain closed. For instance, if a feature is being licensed to others for money, then open sourcing that feature cuts off a stream of revenue. If the component is closely linked with a differentiating feature and the only available open source implementations are licensed under share-alike licences, the differentiators might also have to be made available as open source.

Product feature categories

There are three categories of features that need to be considered:

- Differentiator: these features provide added value to product. They give an edge over the competition and provide a reason why customers want to buy the product.
- Baseline: these features are necessary and expected by customers. They provide value, but no added value to the product. For example, today no one buys a portable music player because it supports the MP3 format. On the other hand, no one will buy such a player if it does not support MP3.
- Commodity: these are hidden, uninteresting features. They are needed to make the product work but the customer does not care about them.

For each category, the classic decision of whether to make or buy now becomes one of deciding whether to make, buy or open source. To answer this question for a particular feature, the first step is to classify it as either a differentiator, a baseline feature or a commodity feature. Second, for each feature the impact of each option (make, buy or open source) should be determined. The diagram in Figure 1 shows the impact of each.

Generally speaking, differentiators should be developed in-house to ensure maximum market advantage. Outsourcing (buying) such features is an option, but that means a company relies on a third party for the most critical parts of its product. This creates a significant risk should the third party run into problems. And, of course, open sourcing such a feature is usually not a good idea, as doing so would turn the feature into a commodity.

The opposite holds true for commodity features. A company should not spend time and effort in-house to develop or maintain such features. Using open source here saves time, effort and money and thus usually is the right choice. An example is using Linux as an embedded operating system.

In the middle are the baseline features. Here a case-by-case approach to decide whether to make, buy or use open source is most appropriate. A company might choose a rule of thumb, such as “we use open source unless the feature is being licensed for money to others” to help make the decision. Technical considerations may block a decision to open source a particular baseline feature. For instance, if open sourcing the baseline feature would also require open sourcing a differentiating feature, it is clear that the baseline feature should remain closed.

In certain situations, open source can be inappropriate. A frequent issue is that the open source licence conditions could be too cumbersome or costly for a particular product. There could also be technical reasons to use a closed source alternative. For example, in a small embedded device Linux might not be the best choice because of its relatively large system requirements. Another reason is when a customer demands that no open source is used.

Legal software design

While designing software legally may seem straightforward enough in theory, the practice can be difficult. Building a software-based product is not a matter of putting components together like Lego blocks. Source code can be copied and pasted, libraries can be linked or accessed through remote procedure calls. With object-oriented languages such as Java and web-accessible services, the possibilities grow even larger. This may create complex interactions and dependencies between the software

components. Assessing the legal implications of open source code in such a complex software stack can be a difficult task.

To ensure these implications can be properly managed, IP professionals should get involved in the product design at an early stage. When an IP assessment has to be carried out on a fully completed software stack, making changes based on IP or legal considerations can be very difficult or costly. IP and legal design goals should be formulated at the beginning of the project, so that these can be taken into account during the design and implementation of the software.

Into the ABISS

A concrete example of how to put the above into practice is the so-called Active Block IO Scheduling System scheduler – ABISS for short. This Philips contribution to the Linux operating system enables real-time reading and writing of streams of data from a hard disk. This is an important topic, especially for audio or video playback. With ABISS, Linux applications can request real-time delivery of video or audio streams from a hard disk at a certain playback speed. ABISS either delivers the stream at the requested speed, guaranteed, or it tells the application that the request is not possible. Such a scheduler significantly enhances the design of a hard disk recorder or video player.

A key element of ABISS is its scheduler. The scheduler balances requests to make sure the desired rate can be delivered. The choice for a particular scheduling algorithm, or even the policies and rules to be used with the scheduler, can have a significant impact on the system's performance. A policy covers items such as the maximum number of files that can be open at once. Philips has spent a significant amount of research developing and fine-tuning fast and efficient algorithms and associated policies.

For Philips, the reason to make ABISS open source was to establish the technology as a *de facto* standard. Further development of ABISS would be done jointly with the Linux community. This has several advantages. To name one, if the Linux hard disk access subsystem were to change, the community could change ABISS accordingly if it were part of Linux.

In a straightforward implementation, ABISS would be implemented by modifying the Linux operating system directly. However, as Linux is licensed under the GNU GPL, that would mean ABISS as a whole would have to be made available under the GPL as well. Because that would commoditise the valuable know-how related to the scheduling algorithms and the associated scheduling policy choices, this was not desirable.

In addition, such an implementation could make it difficult for third parties to use ABISS for their applications. To drive wide acceptance of ABISS, it was important that ABISS could be used with proprietary (closed source) third-party applications.

Legal design goals

The above was translated into three IP-related design goals:

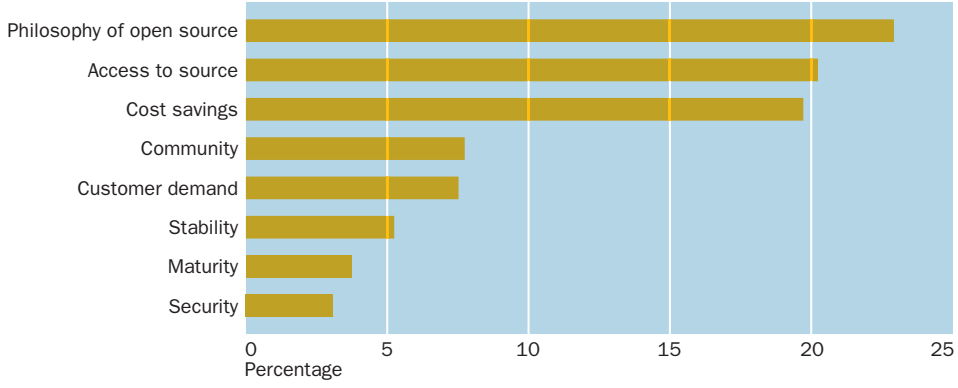
- Allow any application to use the ABISS functionality.
- Allow any party to use proprietary scheduling algorithms in ABISS and to tune them using proprietary know-how.
- Ensure changes to ABISS itself become available as open source.

The engineers worked with the IP department to redesign the ABISS architecture based on these goals. This resulted in several material changes to the ABISS architecture. The end result is illustrated in Figure 2.

ABISS was redesigned from one big block of code to a framework into which scheduling algorithms and policy settings could be included as separate components. The framework was realised as a set of modifications to the Linux kernel. The scheduler was implemented as a so-called Loadable Kernel Module. Such a module is a separate component that interacts with the framework using a standardised Linux interface.

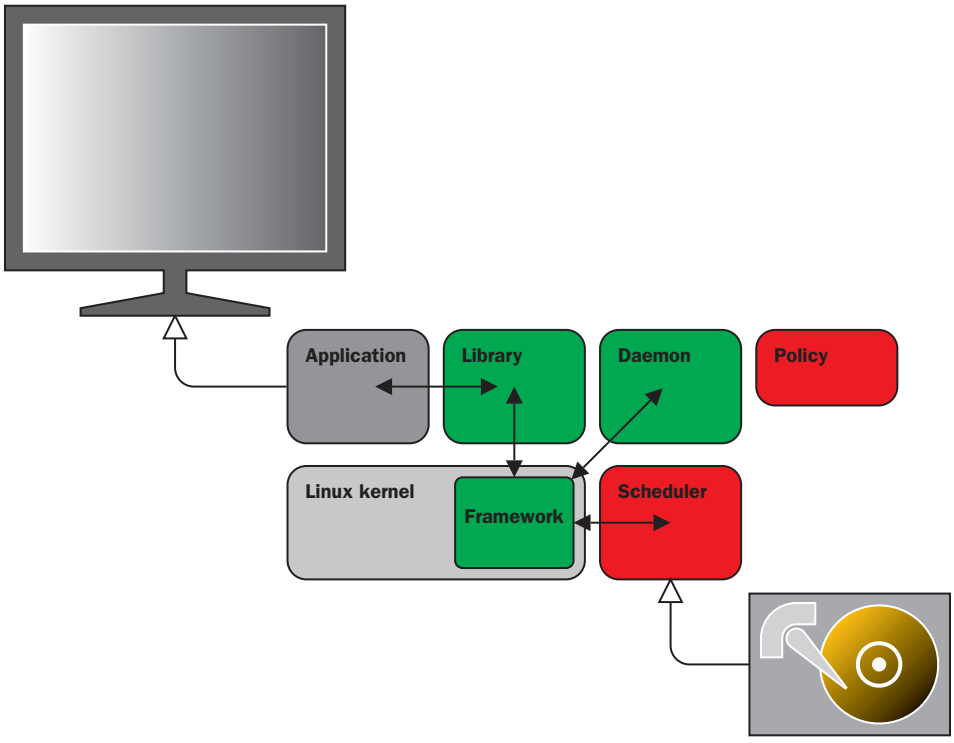
An application that wants to read or write a file with a guaranteed bit rate uses a

Largest motivators to develop open source software



Source: *Evans Data Corporation Open Source/Linux Development Survey, Spring 2006*

Figure 2
ABISS system architecture



system library that provides access to the ABISS functions. For technical reasons, the functionality to manage the scheduling policy and the scheduling algorithm was separated into two components. A background process (called a daemon in Linux terminology) communicates with the ABISS framework to execute the scheduling policy. Policy settings are provided through a small component (a plugin) that is loaded by the daemon.

By designing the architecture in such a way that the scheduler and the policy were implemented as separate components, the above design goal in the second bullet point could be satisfied. The software that interacts with these components was licensed under a keep-open licence, which satisfies the design goal in the third bullet point and allows the combination with proprietary scheduler and policy modules. Similarly, the library for applications that need to use the ABISS functionality was made available under a keep-open licence, satisfying the design goal in bullet point one. The changes to the Linux kernel could only be licensed under a share-alike licence, but because of the ABISS architecture this had no consequences for the other licences.

When ABISS was released, a scheduler with basic functionality and policy settings was included. This basic scheduler shows ABISS is viable, yet it does not contain differentiating know-how. Releasing only a partial implementation (ABISS without scheduler) would have created a very negative impression.

Strategies for effective use

Today many companies regard the use of open source as an *ad hoc* decision, made by individual programmers when trying to meet deadlines. Formal open source policies often treat requests to use open source as a rare exception: "not allowed, unless". Significant business advantages can be obtained by actively selecting open source for certain features and using closed, commercially licensed software for other parts.

Policies regarding open source should not be presented as a prohibition but as part of a strategy to make effective use of open source. A positive attitude from the IP and legal professionals with good knowledge of software development issues is very important to get such a strategy off the ground.

Following such a strategy, the first step is to classify features as differentiating, baseline or commodity and to determine for each whether using open or closed software is the best choice. Second, the software architecture should be designed with these legal design goals in mind. As the ABISS example shows, this requires close cooperation between the software architects and engineers and the IP and legal professionals. When done carefully, this approach achieves the best of both worlds. ■

Arnoud Engelfriet *is a European patent attorney working at Philips Intellectual Property & Standards. As secretary of Philips' Open Source Advisory Board he coordinates the IP aspects of the use of open source software by Philips*
Arnoud.Engelfriet@Philips.com