EMBECOSM®

Open Source in Embedded System Development

Jeremy Bennett, Embecosm

Presented at the Embedded Masterclass, Cambridge 5th October 2011.

Abstract

This paper introduces the huge range of free and open source software available to the embedded software developer. Hardware modeling, software tool chains, operating systems (RTOS and Linux), middleware and applications are all covered. Today open source is spreading to the hardware world.

The paper addresses the advantages and risks associated with using free and open source software, including the issues of quality, support and licensing.

History

First a little history. Where did the open source philosophy originate? One answer is the MIT Model Railroad Society in the 1950s. It might be folklore, but when the University acquired its first computer, members of the society would sneak in at night to play with the equipment. Their exploits or "hacks" (a student term for a prank) were shared openly, with members competing to show what they could achieve.

This culture of innovative "hacking", and the associated openness spread across campuses as Universities acquired computers in the 1960s, 70s and 80s. Some of that early free software is still widely used. MIT's X window system and Donald Knuth's TeX being examples.

What is "open source software"

A key step was the GNU Manifesto published by Richard Stallman in 1983, as a protest against the growing commercialization and proprietary nature of software, which he believed was destroying innovation. The name identifies his primary culprit. What's GNU: **G**NU's **N**ot **U**nix.

Originally such software was referred to as "free" software, but in English "free" has two meanings. It can mean "not paid for" as in "Jeremy would you like a free beer?" Or it can mean "freedom".

However software that is "free as in beer" does not necessarily make you "free as in freedom". I can give you my proprietary software free of charge until you have your entire corporation adopting it. I can then decide to charge you for the next version—you are being held to ransom. If you don't pay up, you have to start again with your entire organization.

We are interested in "freedom". With software this can be achieved by giving the user the source code to a program, so they can independently develop it. No longer can they be held to ransom. To make this distinction clear, the term "open source " was introduced in 1998 by Eric S Raymond and others, along with the Open Source Initiative, which aimed to define what was meant by open source software.

The Open Software Initiative came up with a set of criteria, know as the Open Source Definition which any software project must meet. There are 10 criteria in all, but they can be summarized into just three principles.

1. The principle of free distribution. There must be no restriction on any party giving away the software freely, either standalone or as part of another program.
2. The right of access to the source code, and this is important because it allows anyone to make what are known as "derivative works". These may be bug patches, or they

may be completely new programs, but they are central to the true value of open source software.

3. Finally there is the principle of non-discrimination. By this is meant discrimination against who may use the software, discrimination against the technology on which this software may run, discrimination against what the software can be used for, and discrimination over whether you can use the software commercially.

Today many people refer to “free and open source software” or FOSS to make it absolutely clear what they mean. By contrast “freeware” refers to software that you don't pay for, but for which source code is not available.

Why does anyone develop open source commercially?

It is important to understand that it has always been perfectly acceptable to make money out of open source software. However it should also be immediately clear that if the source code is freely available, then simply selling the software is not going to work.

Yet most open source software today is developed by well paid commercial developers. While hobbyists and academic “hackers” are still important, they are only a minor part of what takes place.

And you can make a lot of money. MySQL, the open source database company was sold for a billion dollars. Red Hat, the Linux and services provider is a multi-billion dollar company. IBM, the world's largest holder of patents makes more money from open source than it does from its patents.

The reason is that this free software underpins other very valuable commercial products. Websites like Google and Facebook depend on open source infrastructure. Chip providers like ARM and Intel depend on robust tool chains for their processors. All these companies have a vested interested in contributing effort for the common good.

Other companies like IBM, Red Hat and my own company, Embecosm, make their money through services. You have the source and you can of course maintain and develop the software yourself. But invariably it is cheaper and quicker to ask an expert to do it for you.

The bottom line is that today, open source software is often very robust and commercially supported. The main advantage is that, unlike proprietary software, it is lower risk. You cannot be held to ransom by your supplier.

The Legal Situation

Open source software is not (usually) “public domain”. It is protected by copyright. Copyright law has the merit of being largely the same around the world. It is usually free, and in most countries automatically granted to an author. Most importantly the courts have for many years held that software is a creative act covered by copyright law. Open source software works by granting the user a license to the copyrighted software, in exchange for which the user must follow the rules of open source software.

There are a wide range of open source licenses in use, and one of the downsides of using open source software is the need to understand the differences. We can categorize the various open source licenses in two ways.

- First whether the license is permissive or non-permissive. Permissive licenses like the MIT and BSD licenses grant you access to the source code, but make few demands on what you then do with the code. By contrast, non-permissive licenses like the GNU license require that if you pass the software on, you must also pass on the source code, including any changes you have made and all the license obligations.
- Secondly whether the license is viral or non-viral. If you include open source code with a viral license in your own code, then the entire body of code becomes covered by that viral license. By contrast non-viral licenses only cover the open source code yourself.

If you are just using open source software as an application, then the details of the license are not hugely important.

However if you are going to incorporate software into your own developments, then you must take care. In particular including a library with a viral license could make the entirety of your code open source. That is fine if you wish to be open source, but otherwise is a problem. Even if you wish to open source your own software, you may also be linking in a proprietary library, which would then have to be made open.

Many users are initially attracted to non-viral, permissive licenses. They allow the user to take maximum advantage without the need to contribute back. The downside is that the communities associated with such licenses can be much weaker. There is little incentive to contribute to a project where the users give nothing back.

So the most widely used open source license is in fact the GNU General Public License (GPL) which is both viral and non-permissive. It is very demanding on users, but conversely generates very strong developer communities. The success of Linux shows that GPL need not be a barrier to adoption.

In the embedded community there is widespread use of non-viral licenses such as the GNU Lesser General Public License (LGPL) or the permissive BSD and MIT licenses.

The bottom line is that open software offers advantages, but you must follow two key engineering principles.

1. Understand the licenses you are using.
2. Maintain strict software management, so you know what software you incorporate.

With good engineering management, incorporating open source offers high value at low risk to the developer.

Fear, Uncertainty and Doubt

There are some myths which surround open source software and some pitfalls for the unwary. We have already encountered some of these earlier, but to summarise.

1. The first myth is that open source software is full of bugs. But any significant open source project has far more eyes looking for bugs and their fixes than a commercial product. Apache wouldn't have obtained its dominant position in the web server market if it was not extremely robust.
2. The second myth is that open source software is unsupported. But how many in this audience can say that every bug, or feature request they have submitted to a proprietary tool provider has been dealt with promptly and accurately? As we have seen there are plenty of commercial enterprises supporting tools, and in addition there is the wider community who are a source of support. And with open source you always have the final option of fixing it yourself.
3. The third myth is that free in the sense of “unpaid for” software is the same as open source software. But as we see aove, such software just leaves you at the mercy of ransom demands from the supplier.

The pitfalls are all to do with licensing.

1. As a user, check what you are getting. A supplier may advertise a tool as being open source and based on Eclipse. However the Eclipse license is permissive and non-viral. So all too often you'll find the interesting parts are proprietary and secret. As an end user your best friend is the viral license.
2. The biggest danger for developers is when incorporating virally licensed code in their products. As noted, good engineering discipline can easily avoid undesired “contamination”.

3. Constructing open source programs by bringing together open source components from various sources requires care, since not all licenses are compatible. For example you can combine GPL and BSD licensed code, but only if the combined code is licensed under the GPL. This must be planned right at the start of the project.

Warranties and certification

Many industry sectors (automotive, medical, aerospace for example) require suppliers to use certified tools and operating systems. Invariably such projects will require warranties, guaranteeing the code.

Warranties can be problematic for open source software, since in its nature, it is not possible to know the full provenance of the code. Invariably, open source software licenses explicitly provide NO WARRANTY.

A number of companies offer specific versions of tools and operating systems, which they have checked, and for which they provide warranties. That warranty costs money (the supplier will have to back it with insurance) and almost invariably is voided if you modify the software.

Certification is harder, because of the difficulty of charging for open source software itself (you can do so, but why would anyone buy more than one copy). Where a company owns all the rights to the software it can choose to issue a proprietary licensed version that is certified and not open source.

Alternatively companies will take software through certification as a service. That is hard for one company to justify, but it is possible for governments. The UK has for example certified one particular Linux distribution for some military and security applications.

Suggestions of open source software for use in embedded systems

Now the interesting part. What open source software might you use in an embedded system? The range is huge, and this paper is necessarily selective.

Wikipedia (www.wikipedia.org) has a number of lists of open source software, and is a good starting point when considering open source software. While most large projects have their own websites, smaller projects make use of one of a number of hosting sites. Two of the most popular are SourceForge (sourceforge.net) and GitHub (github.com).

Hardware modeling

When it comes to hardware modeling, EDA has been a bastion of proprietary software—much of it costing $50,000 per seat per year or more. However open source has some credible modeling tools now available. For high level modeling, **SystemC** has always had an open source reference implementation, while for cycle accurate modeling of synthesizable Verilog RTL designs, **Verilator**, on its third major release cycle, is well established and robust. For event driven simulation, **Icarus Verilog** handles Verilog, while the newer **GHDL** handles VHDL. Both event driven simulators are significantly slower than the fastest proprietary products. However for many designs—with hundreds of thousands of gates or so, they are quite adequate.

More generally the **Free Electronics Lab** (formally the Fedora Electronics Lab) is a collection of open source EDA tools, that forms part of the standard Fedora Linux distribution.

Compiler tool chains

Whether you have physical hardware or only a model, you will need a compiler tool chain and associated CASE tools.

For compilation, the GNU Compiler Collection (**GCC**) has dominated for the best part of 25 years, and supports nearly 40 architectures in its standard distribution, with a wide range of languages (C, C++, Java, ObjectiveC/C++, Fortran and Ada as standard).

However GCC is showing its age, and **LLVM**, with a more modern architecture is now becoming a very popular alternative. Currently only C and C++ are well supported, with only a handful of official targets ported. However many chip manufacturers have teams working on LLVM versions for their processor. Apple and ARM's backing ensures LLVM will continue to move forward rapidly.

Neither GCC nor LLVM is really aimed at "small" processors. For C support for microcontrollers, the Small Device C Compiler (**SDCC**) is a good alternative.

Java took a while to come into the open source fold. The GCC Java compiler generally suffers from compatibility issues. Fortunately Sun eventually made available the **OpenJDK** system. It is not as complete as the full commercial system, but allows compatible Java development in an open source environment.

Libraries

Since almost all software can be used as a library, we only consider the main C/C++ system support libraries. These libraries are generally characterized by having more non-viral and more permissive licensing.

For small embedded systems, the GNU tool chain is often used with **newlib**, a very small library suitable for bare metal and RTOS environments. The main GNU library is **glibc**, but too large for most embedded applications. A smaller library is **uClibc**, representing a good compromise between newlib and glibc, and rich enough to be used with Linux.

C++ demands a comprehensive library, and both GCC and LLVM supply a version. However an alternative is the **STLport C++ library**.

Graphical CASE

GCC, LLVM and SDCC are all command line tools. The pre-eminent graphical CASE environment is **Eclipse**. Its configurability and relatively permissive licensing mean that it is used as the graphical front end to a huge range of tools. Both GCC and LLVM have very good integration in Eclipse.

Source code debugging

For source code debugging, the GNU Debugger (**GDB**) has been the main open source option, and includes a very powerful integration in Eclipse. The LLVM project is working on its own debugger (**LLDB**), but is not yet mature.

Version control

For source code version control, there is a wide choice. For a traditional centralized repository there is **CVS** and **Subversion** (which is newer, and tending to replace CVS). For distributed repositories, **git**, best known for being used by the Linux project is pre-eminent, although there are alternatives in **mercurial** and **bazaar**.

Build systems

The longest standing build tool is **make**, a standard part of any Linux distribution. There are many implementations (not all originally open source), but the GNU version is both common and powerful. Make does not really scale well. The GNU autotools (autoconf, automake and libtool) are a long established solution building on make, but need some care in their use. A newer alternative is **cmake**. For Java systems **Ant** is commonly used.

Operating systems

Many RTOS are open source, or at least offer open source variants. Examples include **RTEMS**, **FreeRTOS** and **eCos**. In most cases the suppliers offer proprietary variants, which either offer support and warranties, or are certified for use in particular applications.

Linux is a poster child for open source software development, and widely used in larger embedded systems. Some companies offer commercial versions of Linux for embedded use. While the kernel is open source, there is no reason companies cannot add proprietary drivers and applications.

For embedded applications, **BusyBox** is often used as a lightweight shell to run on the kernel (for example in Netgear routers).

Android is a Linux derivative, and while Google have been criticized for keeping some parts proprietary, much of the code is fully open source.

Middleware and applications

The range is huge, so this paper will only cover a handful that are of particular interest to embedded engineers.

Open source code for many protocol stacks is available, particularly under Linux.

For timing synchronization between machines, **NTP** has always been open source.

For databases, the two big open source options are **MySQL** and **PostgreSQL**. There are a wide range of other open source databases catering to specific needs.

Many embedded systems need to run a web server. **Apache** is used by 70% of all web servers worldwide, but is quite suitable for embedded use as well (it is available with BusyBox for example).

The following table summarizes all the tools discussed in this section, together with a link to their source.

Tool	*Source*
Hardware modeling	
SystemC	www.systemc.org
Verilator	www.veripool.org/wiki/verilator
Icarus Verilog	iverilog.icarus.com
GHDL	ghdl.free.fr
Free Electronics Lab	spins.fedoraproject.org/fel
Compiler tool chains	
GNU Compiler Collection (GCC)	gcc.gnu.org
LLVM	llvm.org
Small Device C Compiler (SDCC)	sdcc.sourceforge.net
OpenJDK	openjdk.java.net
System C/C++ libraries	
newlib	sourceware.org/newlib
glibc	www.gnu.org/s/libc
uClibc	uclibc.org
STLport C++	www.stlport.org
Graphical CASE	
Eclipse	www.eclipse.org
Source code debugging	
The GNU Debugger (GDB)	www.gnu.org/s/gdb
The LLVM Debugger (LLDB)	lldb.llvm.org
Version control	
CVS	www.nongnu.org/cvs
Subversion	subversion.tigris.org
Git	git-scm.com
Mercurial	mercurial.selenic.com
Bazaar	bazaar.canonical.com/en

Tool	*Source*
Build systems	
GNU make	www.gnu.org/software/make
GNU autoconf	www.gnu.org/s/autoconf
GNU automake	www.gnu.org/software/automake
GNU libtool	www.gnu.org/software/libtool
cmake	www.cmake.org
Ant	ant.apache.org
Operating systems	
RTEMS	www.rtems.com
FreeRTOS	www.freertos.org
eCos	ecos.sourceware.org
Linux	www.kernel.org
BusyBox	www.busybox.net
Android	www.android.com
Middleware and applications	
NTP	www.ntp.org
MySQL	www.mysql.com
PostgreSQL	www.postgresql.org
Apache	www.apache.org

Open source hardware

Open source now extends to hardware as well as software. Licensing is more problematic, since copyright law generally does not apply to manufactured items.

There are a number of board level designs which are open source, of which **Arduino** is probably the best known. **Milkymist** is an open source video system based on an open source processor.

There is an increasing range of open source silicon IP in Verilog and VHDL. Such IP is best suited to FPGAs, although there have been open source ASICs produced. Perhaps best known is the **OpenCores** project, which includes the **OpenRISC 1000** 32-bit RISC processor and the reference Systems-on-chip **ORPSoC** and **MinSoC**.

Other open source hardware designs are hosted by the **Open Hardware Repository** at CERN (although they tend to be designs useful in particle accelerators).

Lattice Semiconductor have made their **LM32** processor available in open source form. This processor is used in the Milkymist project and a version of Arduino based on the LM32 has also been announced.

The following table summarizes the open source hardware available.

Project	*Source*
Board level designs	
Arduino	http://www.arduino.cc/
Milkymist	http://milkymist.org/
Silicon IP	
OpenCores	opencores.org
OpenRISC 1000	opencores.org/or1k/Main_Page
ORPSoC	opencores.org/or1k/ORPSoC
MinSoC	www.minsoc.com
Open Hardware Repository	www.ohwr.org
Lattice Semiconductor	www.latticesemi.com

Summary

This paper has looked at all the issues concerned with using open source software, particularly as it applies to embedded systems. The information should provide a useful starting point for any embedded engineer considering adopting open source.

About the Author

Dr Jeremy Bennett is Chief Executive of Embecosm Limited. Embecosm (www.embecosm.com) provides open source services, tools and models to facilitate embedded software development with complex systems-on-chip.

Dr Bennett is also the embedded systems champion for the Electronics, Sensors and Photonics Knowledge Transfer Network. The KTNs are government funded and charged with increasing the competitiveness of UK industry by improving the flow of ideas between industry and academia.

Contact him at jeremy.bennett@embecosm.com.