Improving code migration and reuse

By Robert Day

As the amount of embedded software grows, an organization's ability to generate, debug, and test it becomes more difficult. Software groups must be able to reuse as much code as possible from both their own previous designs and from commercial and open source avenues. Robert considers what tools, methodologies, and practices can be utilized to make reuse and code migration possible and some complex applications where software black boxes *can help.*

The combination of processing power increases and the additional features, functionality, and connectivity of today's embedded devices has expanded software content dramatically. However, time-to-market pressures have also increased, and while the software side has not provided magic answers to generate this new mountain of code, a number of practices, standards, and tools can help.

Often as new features are required in products, *make versus buy* issues must be considered. As an example, consider an embedded product that must be networked in its next generation, requiring a software TCP/IP stack. The dilemma then is whether to:

- Hire networking specialists to write the stack
- Outsource to some networking specialists
- Buy a TCP/IP stack and integrate it
- Find an open source networking stack and integrate it

The options all have cost and time implications, and for most situations, a combination of acquiring some software and gaining some expertise in that arena is the most efficient, timely, and cost-effective route.

When porting applications or creating the next generation of device, the more software that can be reused, the greater chance the software will be ready on time and work properly. Reusable software Intellectual Property (IP) can take many forms, but it essentially equates to software building blocks or components that have been used before and hence have a built-in reliability and modularity that allows for ease of implementation into new designs. It is now common for embedded software designs to have many software IP blocks; some proprietary to the developer's organization and often specific to the embedded application itself; some commercially acquired general purpose IP such as Real-Time Operating Systems (RTOSs) and networking stacks; and some specific applications such as databases and Web browsers. All of this IP must be brought together, integrated, and tested.

Higher-level programming languages

Most software written for embedded systems (especially larger systems with 16-, 32-, and 64-bit architectures) is written in a high-level language, primarily C. While using C language gives an element of portability and reusability by recompilation, much C code in the embedded world has not been written with portability in mind. In fact, code is often tailored and compiler options tweaked to increase performance or optimize memory use, and the result is nonportable. When migrating to a newer environment, perhaps utilizing a different or newer C compiler, the code often breaks.

Using C++ as a programming language can help with reusability. Commonly used and reused sections of code can manifest themselves as objects. These small software IP *black boxes* allow some of the often low-level functions of embedded code to be hidden yet accessible through a high-level interface. When the next system is being designed, these pieces can be ported without breaking the application view, allowing most of the embedded application to be ported through a recompile.

C++ compilers and the language itself tend to be stricter and more clearly defined than C. Today's C++ compilers often have

built-in portability checking and will flag errors for nonportable code, helping to keep code standard and more portable than much of the existing embedded C code. C++ still has some issues when programming tightly memory constrained systems, but for most large embedded systems, its performance is adequate. Some embedded programmers prefer to use a C++ compiler even though their code is written in C because of its better error and portability checking.

A study once showed the average programmer can write 10 lines of fully debugged code per day, irrespective of programming language, in other words, 10 lines of assembler or 10 lines of C or C++. By using high-level languages and tools for modern processors geared toward high-level compilers, the productivity per machine instruction increases dramatically. This begs an interesting question: Is there anything higher than C or C++ that could further increase productivity? The answer is high-level design languages, such as Unified Modeling Language (UML).

UML is a graphical language allowing the software developer to show the software system at a higher level than in traditional high-level languages such as C and C++. These design languages allow the programmer to specify the system's building blocks, the events and triggers that come in and out of those blocks, and the association they have with other building blocks. This becomes a design specification and can be used to document how the system will behave down to the smallest building blocks, which is useful when programmers have to write certain modules interfacing with modules written by other programmers, as it gives a clearly defined reference. However, today's UML tools take this one step further and generate high-level code from UML, meaning that for each line of UML code written, the UML compiler generates a certain number of C or C++ or other high-level language lines.

Since UML defines system level modules that will generate C or C++ code based on a set of predefined and consistent rules, the ability to migrate this code across different applications, hardware platforms, and operating systems is much easier. Using UML can also make code testing simpler and more automatic. With the interactions described for the modules, developers specify a domain for each parameter and representative test values for each parameter, enabling another tool to generate test vectors from the described interactions and supplied data.

Eclipse provides a framework that allows software tools to plug in using a strict set of APIs and a feature-rich Graphical User Interface (GUI).

A highly portable language and environment such as Java aids reuse and migration for embedded applications. Running a Java Virtual Machine (JVM) on an embedded system enables having portable applications not bound by processor architecture. Though more of an application language than an embedded programming language, Java for large applications could play an interesting part in the overall system. The performance of today's processors and JVMs now make this a firm possibility.

Tools move to Eclipse

In the embedded software world, reuse and code migration are also closely tied to software development tools. As previously mentioned, using a high-level language can increase productivity and code migration. But in the embedded world, that migration has often required changing development tools, hampering developers' productivity. Finding a consistent development environment over different processor architectures, RTOSs, and programming languages has been difficult. Chip companies have highly optimized tools for their architectures, and RTOS companies have tools that include detailed awareness of their operating systems, all with their own proprietary Integrated Development Environment (IDE).

Today, a major change sweeping across the embedded tools world marries open source and commercial tools offering a consistent user interface for embedded developers regardless of chip, RTOS, and programming language: Eclipse.

Eclipse provides a framework that allows software tools to plug in using a strict set of APIs and a feature-rich Graphical User Interface (GUI). The framework offers a consistent interface when developing, editing, and building the software, regardless of compiler, making project migration easy. Additional plug-ins for requirements tools, high-level design tools, and version management also allow embedded engineers to take advantage of high-quality products from the enterprise space, all preconfigured in the Eclipse environment.

Even debuggers now have a consistent look and feel under Eclipse. The CDT project in Eclipse provides a modern but standard GUI for debuggers, and comes preconfigured to work with the GNU Project Debugger engine. Many RTOS vendors have chosen to use the CDT debugger and added functionality specific to their RTOS, allowing users to migrate across different operating systems without having to make dramatic changes to tools they use. Figure 1 shows the Luminosity Debugger from LynuxWorks based on the CDT project.

The Eclipse plug-in environment is so flexible that any tools written in-house can also be given an Eclipse plug-in and GUI, meaning tools that have been previously *standalone* can now be brought into the IDE and integrated with the rest of the build, debug, and test tools that are either provided commercially or from the open source community.

Software IP ties to hardware

When building embedded systems, hardware architecture plays a large part in the portability of the software that runs on it. Choosing a processor and high-level tool set is an important step, but the hardware isn't just the processor; it is a myriad of peripherals that must be driven and controlled by the software. Changes in standards, customer requirements, and even protocols can have a dramatic impact on the selection of peripheral devices and the code portability.

Several aspects should be considered, including:

- Choose a processor architecture that supports a wide range of peripheral devices. For example, PowerPC and ARM processors are widely supported by multiple companies for embedded environments, and the popularity of the x86 platform from the PC world has spawned a range of software drivers and IP.
- Instead of writing and porting software, choose peripheral devices where there are commercial or open source software IP providers that provide code and also integration

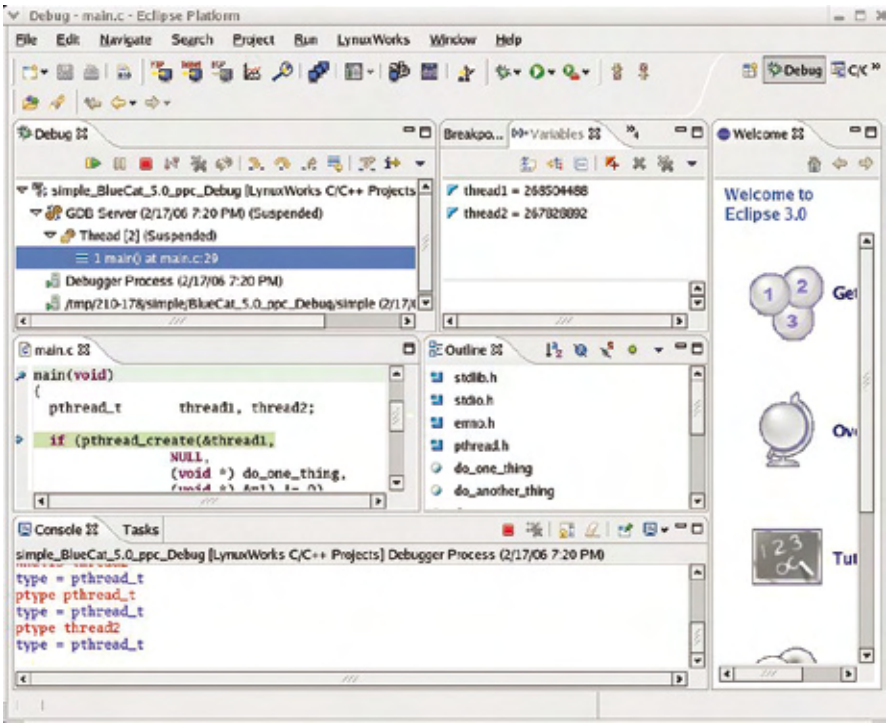

Figure 1

support. This is particularly true of complex communication stacks such as TCP/IP, USB, Bluetooth, Wi-Fi, ZigBee, and similar standards.

- Use reconfigurable hardware. An FPGA's programmable fabric can bring standard hardware IP blocks with software IP to support them into an easily integrated, tested, and prototyped solution upgradeable by reprogramming.

Whatever the choices, integrating the solution presents a significant challenge. Using software IP truly helps speed up development time, as these pieces do not have to be written from scratch. But, this time saved can easily be used up if the software pieces do not work together well.

The answer is to select pieces of software IP that have been preintegrated or at least tested together. The best examples of preintegration are between RTOS and software stacks. Most RTOS vendors also offer a number of software stacks that they maintain and sometimes develop in-house, and a broader selection through partnerships. This gives the assurance that these key pieces of the software architecture and the communication parts work together without major integration work required by the developer, and aids in obtaining support if other integration or testing is necessary.

If support is less of an issue than integration or code writing, then the open source movement could be a good place to obtain preintegrated IP. Often hardware peripheral companies build the underlying chip-specific drivers, integrated with open source stacks and tested with open source operating systems such as Linux, providing a complete integrated software and hardware solution. However, support for this solution is costly, reliant on hardware vendors, or based on the open source community, all of which present risks for software developers.

Certification goes deeper

For certain applications, other considerations make software reuse compelling, such as when the application must be certified. The avionics, transportation, and medical industries all have certification requirements for embedded systems. This can make software IP use difficult because the code will also need to be certified with the application. In this case, new factors come into play when considering the software IP use.

Software maturity and reliability is a very important factor. If the software IP has already been used in a certified application, its credibility for reuse in others is vastly improved. Some commercial RTOS vendors offer artifacts for the different components of the operating systems that can be used by the certification authorities to show its certifiability, reducing risk and potentially speeding up the certification process.

The Federal Aviation Administration (FAA) is trying to speed up the certification process for avionics systems by allowing software IP to be precertified. This precertification produces a *Reusable Software Component* (RSC) and allows these software IP components to be viewed as a software black box when it comes to the certification process. The traditional certification process takes place on the software IP, and thus risks failing certification. With an RSC, the component does not have to go through the same level of certification scrutiny, hence dramatically reducing the risk of failure and cost of certification. To date, only the LynxOS-178 RTOS (Figure 2) from LynuxWorks has been given an RSC by the FAA.

Opening possibilities

Open standards and open source are different things, but both have their place and often apply together in reuse situations. Software standards are generally beneficial for code reusability because adherence to these standards when building code allows portability across architectures, compilers, and when combining software modules together. Conforming to ANSI standard programming rules and using compilers to enforce them makes for better and more portable code.

Using a standard operating system interface is also appealing as it allows for portability across operating systems. Embedded operating systems often have proprietary APIs, and some even resemble other vendors' offerings. This similarity helps developers migrate from one to another, often using their own abstraction layers to insulate the application code from the RTOS API.

POSIX is an open standard API with the idea that code written for one POSIX-based RTOS should be easily portable to another POSIX-based RTOS. However, POSIX is rather large and unwieldy; thus, few operating systems actually fully comply with the POSIX standard. That said, it is still a good practice to use this standard as it is well defined, mature, and well published, and offers a great deal of future proofing for code written to it.

Another API that has emerged in both the enterprise and embedded software worlds, Linux, includes an open source community, an operating system API, a vast collection of open source and commercial applications written for it, and a wide range of Linux operating system ports and distributions. Since its implementation is similar to both UNIX and POSIX, code migration across the three APIs is relatively easy.

Making Linux suitable for embedded and real-time use can be difficult. While the open standards nature of Linux is

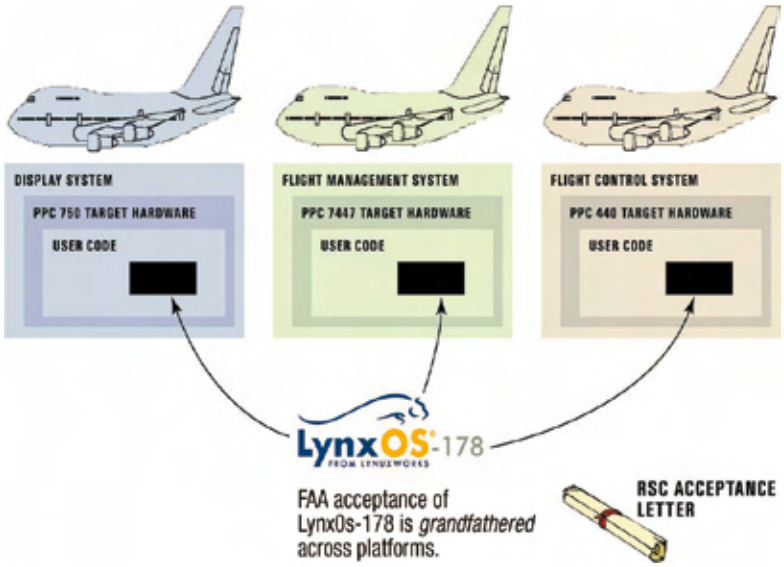

Figure 2

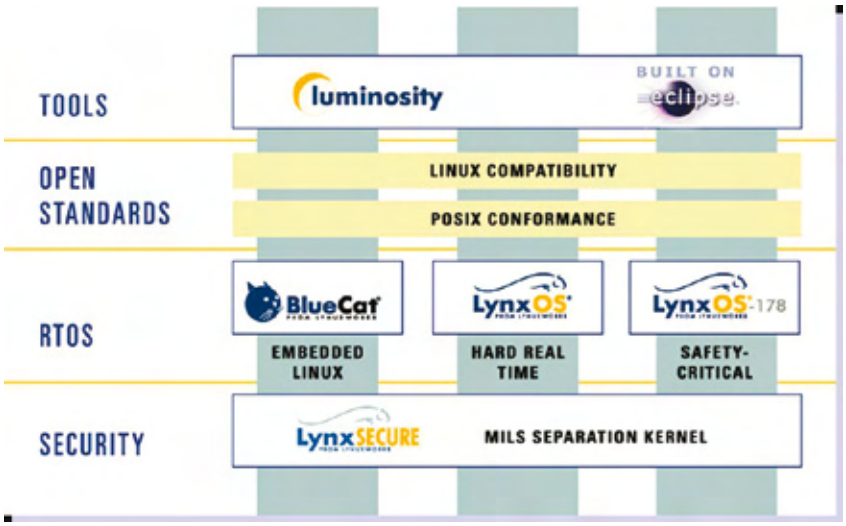

Figure 3

appealing, what developers can download from www.kernel.org or other open sources is not often suitable for running in embedded systems. For example, the ability to run diskless, as many embedded systems do not have access to a hard disk, involves work for the embedded engineer. The ability to run in a hard real-time or deterministic system is also not well supported from open source.

To actually use Linux in a number of embedded systems requires help from an embedded Linux company. There are a number of embedded Linux providers with varied approaches and offerings of technical support and embedded expertise. Most embedded distributions of Linux will run diskless, but the way of dealing with hard real-time and determinism varies. Some embedded Linux offerings modify or patch the Linux kernel to offer better performance and determinism; some even replace the Linux kernel with a more proprietary kernel to achieve the same results. Both of these solutions break away from Linux's open source and open standard nature because they may or may not make it back into the open community in a future version of the Linux kernel. However, they do offer the ability to get Linux running on embedded targets without too much pain for the developer.

LynuxWorks offers a Linux distribution and hard real-time and certifiable operating systems based on POSIX (see Figure 3). BlueCat Linux, a fully embeddable diskless operating system based on the standard Linux distribution, provides the same embedded technical support services as for RTOSs. For users who require hard real-time performance, LynuxWorks gives a unique Linux Application Binary Interface (ABI) that resides above the POSIX API and offers the ability to run Linux applications directly on the LynxOS RTOS. This ABI maintains Linux standards support and helps with code reuse over both soft and hard real-time systems.

These Linux solutions also provide embedded engineers with access to the wealth of applications already written for Linux, hence greatly aiding software IP use. Applications such as networking stacks, Web browsers, databases, and other enterprise applications are more readily available on Linux than any other operating system. This demonstrates how the combination of open source and open standards helps drive software IP availability and aids the reuse of software in embedded systems.

Lifting the burden

As the embedded world moves closer to the enterprise space with applications and software required in embedded devices, the amount of embedded software is growing at an exponential rate. It is clear that without a software reuse and migration strategy, embedded companies will not be able to keep up with the pace and demands of the market, and software will increasingly become the critical path for deliverables.

Fortunately, using standard processors and peripherals, software IP, open standards operating systems, and higher-level programming languages and tools make this job easier and lift the burden for the embedded software developer. ECD

Robert Day *is the vice president of marketing for LynuxWorks. His responsibilities include leading program management teams and driving worldwide marketing initiatives, including corporate communications and brand strategy.*

For more information, contact Robert at:

LynuxWorks
855 Embedded Way
San Jose, CA 95138-1018
Tel: 408-979-3900
E-mail: rday@lynuxworks.com
Website: www.lynuxworks.com