Fear and loathing of porting embedded software

Robert Cravotta - August 08, 2002

AT A GLANCE

• Breaking portability is usually a matter of technical necessity.
• Define the system headroom and results you need for a successful port before you start porting.
• Implicit coding assumptions are significant sources of porting pitfalls.

Sidebars:
Striving for silicon independence

Using legacy software in your project can minimize your development time because you can avoid duplicating effort for design specifications, test cases, and performing implementation trades from earlier development efforts. However, legacy software, even when written in a portable language, often requires you to perform some porting. Software porting is the engineering process that transforms software so that it will operate within a new target environment. Porting differs from development or maintenance efforts in that the software probably does not specifically anticipate the
The need to port software is not limited to reusing legacy software from other projects. A given application can change its target environment as it undergoes generational upgrades. The types of changes that can force you to port your software include using a different processor, modifying the system board, migrating to a different operating system, and adopting a new development tool set. Depending on the nature of your software and the differences in the original and target environments, the porting process can be as simple as recompiling your code and verifying that everything works properly or as complex as rewriting and reoptimizing large sections of the application.

Given the significant variability in embedded-processor architectures, the key to the simplest porting efforts is using portable code. Using high-level languages, such as C and C++, can abstract the processor-architecture details, provided that the compiler is aware of and can effectively use the features and resources of your target processor. Operating systems and drivers can abstract the interaction with peripherals and real-world interfaces. In these cases, the development tools and support files abstract the complexity of the porting effort. In an ideal world, this is as complex as porting would get. However, the realities of an embedded environment often dictate otherwise. Embedded software often consists of various levels of portable and nonportable code (Figure 1).

The myriad processor architectures available are a testament to the wide range of efficiency vectors and constraints that embedded applications must consider. Some of these vectors are cost, processing capacity, device size, peripheral set, code size, and power consumption. Each type of processor makes trades between these constraints, maximizing some features and minimizing others (Reference 1). Eliminate the sensitivity to enough of these constraints, and general-purpose processors make more sense to use.

Standard programming languages can support higher portability when processor architectural variability is limited. Operating systems help limit the visibility of architectural variability, but this abstraction rarely extends beyond how to interact with peripherals, board-level resources, and other software modules. Many processors include features that emphasize a balance between resource constraints that operating systems and standard programming languages do not accommodate, such as implementing low-power modes, fixed-point data types, and parallel resources. To fully benefit from these nonstandard features, you have to break your code's portability and use assembly language or compiler-specific directives and intrinsics.

Why would you purposely break your code's portability if it will cost you time and money to port it when you migrate your application to another target? Breaking portability is usually a matter of technical necessity that overrides company portability mandates. Breaking portability entails optimizing one or more of your efficiency vectors and maintaining a sufficient level of headroom. When breaking portability, compare the impact of a future port with changing your target environment today. Consider your future algorithm and headroom needs, as well as the bill-o-material and engineering costs over the product life cycle and volumes between generations.

Breaking portability does not mean that you cannot minimize the complexity of a future port. Isolate your architecture-specific-optimized code from the portable code to make it easier to identify what needs manual porting. Keep a modular structure and minimize the number of access interfaces with your optimized code to simplify the testing complexity. Using a processor whose road map grows appropriately with your application needs will allow you to stay with the same instruction-set architecture and compiler family, which will allow you to continue to leverage your intellectual momentum with your current platform and reduce your effort to realign with the new processor and tool idiosyncrasies.
Your ability to anticipate and abstract hardware features in your software structure will ease your porting efforts. It requires you to remain aware of processor-architecture trends for your application space. You can abstract the hardware in your software by using globally used compile-time constructs in your code. You can use #include and header files that explicitly declare typedefs and constants to abstract hardware-architecture details, such as the length of int, long data types, and maximum values. Using parameterized macros can abstract endian-ness, data packing, and data-alignment features, but be aware that you can incur significant runtime performance overhead if you blindly apply these macros. You can use conditional assembly and compilation directives to explicitly identify which language standardizations you are using and to maintain a single source tree to avoid source divergence when software supports multiple configurations.

Porting scenarios

As your application evolves from generation to generation, it may reach a point at which your current platform becomes insufficient or at which it is too risky to try squeezing one more generation out of your workhorse because there is too little headroom. You might change processor families because the road map for your current device will not meet your future needs, you may need higher device reliability, or you are receiving poor technical support from the device vendor. As your application matures, you may change your target environment to reduce system costs. Many processor families include dozens of members with various configurations of integrated memory, peripheral, and controller support, so that you can better match a device to your application’s most sensitive constraints.

However, changing your processor is not the only reason to perform a software port. Changing your system board to support a new capability, such as USB or Ethernet connectivity, will require you to rewrite or replace some peripheral drivers. You may decide to change your operating system (Reference 2). You may choose to rewrite your legacy code in a different language to improve its portability and maintainability (see sidebar “Striving for silicon independence”). Changing your tool set, even for the same processor, can cause you some porting effort, because the new tool set may be slightly different.

Several general strategies are available for porting your code. The most common is to recompile your portable source code with a compiler for your target environment and translate the nonportable software, whether it’s low- or high-level code. For most processors, you must manually perform this translation, because few tools support mapping processor- or compiler-specific features to another. Although there is complexity in mapping target-specific features to different targets, the small number of automated translation tools is more due to the huge scale and low return for the effort to provide significant translation coverage among the many possible processor-architecture pairings. MicroAPL offers several tools for translating source code, from 680x0 and 80x86 assembly to PowerPC, 80x86, and ColdFire architectures. These types of translators analyze the program flow of the original code, perform static optimizations, and accommodate incidental side effects, such as with condition codes, before translating it to source code for the new target.

If you can absorb the memory footprint and processing overhead of a runtime translation engine in your system, you can retain and emulate your original machine code on your target processor. In this scenario, the translator reads each original machine instruction, decodes it, and emulates it on the target architecture. This approach requires no changes to the original software. You might consider using runtime translation if source code is unavailable or you do not control the source code, such as when you have purchased object code from a vendor. This method is especially viable if you cannot persuade the vendor to port, or allow you to port, the code to the new target. Even if you have the source, it may be in a language that has no assembler or compilers available for your
Although runtime translators, such as those offered by MicroAPL and Transitive Technologies, incur a translating overhead in your system, they can allow you to maintain a single version of your source code while using the same machine code on disparate architectures. Runtime translation is similar to virtual machines in that both engines emulate your code to operate with nontargeted processor architectures. However, virtual-machine implementations, such as Java, differ from runtime translators by defining and using a standardized byte-code instruction set that is architecture-independent instead of translating native machine code from one architecture to another.

**Preparation is key to success**

You can improve the success of a software port if you perform a requirements analysis and establish your criteria for a successful effort before you start. Defining success criteria provides a basis for your porting decisions and allows you to objectively determine when the port is complete. If you will be continuing to support and enhance the software for the original platform, you may reconsider how to perform the port, including how to bridge the source code between multiple tool sets, so that you need only a single source tree supporting both platforms. Your software may rely on support files, such as operating systems, device drivers, exception-handling facilities, and user-interface libraries, to operate properly. You need to identify all of these required support files and how you will address them in your target environment.

Define the amount of system headroom you need the port to finish with to be considered successful. You are probably not performing this port just for the fun of it. Your original platform is lacking sufficient headroom to some sensitivity vector. You might need more processor capacity for your new algorithms, or you might need a system that consumes less power. You risk another premature porting effort if you have insufficient headroom after completing the port. Remember that, although you may need to optimize the ported code, the main goal of the porting effort is to transfer a function to a new platform. Balance your project time frame with the system-headroom and -performance requirements. This effort will help you complete a good enough port without expending more time and resources than necessary.

Before beginning the port, determine what test cases and procedures are in place or need to be developed. You should know how to apply your test vectors and measure the results on both the source and the destination systems for comparison. Define results in terms of functional behavior and know beforehand what the "golden" and acceptable result ranges are. Commit to using the same tests before and after the port. Develop and use scripts, where possible, to automate testing and improve your regression-testing regimen. At this point, you will have established your assumptions, evaluated your alternatives, determined the functional requirements, and defined the completion criteria so that you can decide whether to continue with the project as is or change the scope to mitigate conflicting requirements.

You can get the porting effort under way in earnest after you determine the target environment and tool options, specify the porting goals and requirements, and determine how to measure your success. Porting is generally a trial-and-error, iterative process, because you do not know ahead of time where all of your problems will be in the code. One way to approach a port is to test only after you have fully analyzed all of the source code for problem areas. For many ports, though, performing trial-code porting followed by repeated testing and refinement is a better approach, because the problems you encounter throughout the software tend to be scattered but similar.

It is important that you understand the differences between your source environment and the new target architecture.
development tools and processor architecture so that you can identify, during code analysis, where you need to transform your nonportable source code to a higher abstraction layer and more clearly express the functional behavior of the code. However, the code may experience some performance loss when you implement it on the new target, forcing you to follow up with an optimization effort. This situation should not be a surprise, because the code in question most likely broke portability to better meet some performance constraint.

As you build the software for the target platform, just like when you develop new software, you will need to iteratively debug and fix the problems. After you have a functional port, optimize those software modules that are not meeting their performance requirements. Confirm the porting effort with a system validation against your predefined test vectors and compare for acceptable results. Complete the process by updating relevant documentation to reflect the changes and ensure that you integrate the new software into your software-building tree for archiving and revision control. If everything goes well, you’ll have a system that performs properly and meets the headroom goals you set.

**Pitfalls and challenges**

Software porting is still significantly a manual effort. The success of a porting effort depends greatly on your team's porting experience, their knowledge of the source and target architectures and tools, and the quality of the source code. Accompanying the challenge of getting the code to operate on a new processor with different tools and modified requirements is the fact that the person performing the port is often not the author of the code. Transposition errors are a risk for projects in which a programmer's attention may wander because he or she is implementing a generic or similar change many times throughout the code. Why are there not more publicly supported tools to automate these types of changes? There are just too many possible processor pairings for a general tool to be useful.

What does a programmer do when compiling the original source code generates many warning or error messages? Did the last person ignore these messages at the last compilation with no problems, or are they artifacts of the development system? Sometimes source code cannot compile on the latest version compiler. An example is that some Linux sources cannot compile on the latest Gnu compiler, so they specify an earlier version of the compiler that they can work with. Sometimes, the code contains a bug that is benign on the original processor but fatal on the new processor, such as from using uninitialized variables or pointers.

Although many compilers have compiler- and architecture-specific extensions that are not portable, they are easier to catch during a porting effort than are implicit language assumptions. The C standard defines and supports implementation-specific assumptions, such as how to pack bit fields, the size of basic data types, and endian-ness. It becomes critical for porting programmers to identify where the software relies on the architectural features of the processor it runs on. Other sources of implicit assumption concerns in C are data alignment, properties of pointers, maximum and minimum values for basic data types, precision properties of noninteger values, and how casting values converts among signed, unsigned, and different data types.

Implicit assumptions are not limited to higher order languages. Assembly instructions can set condition codes that the code expects to remain untouched until it does a query later in the logic flow. DSP architectures can include instructions affecting multiple resources that the software may assume will be left untouched until later. These types of assumptions are not always obvious or flagged in the source.

Nonportable architectural differences include moving among 8-, 16-, and 32-bit architectures;
support for different controllers and peripherals; differing instruction/data bus structures; special instructions; and single- versus multiple-instruction issue engines. The programmer must map onto the target processor original code that takes advantage of these features. This mapping can indicate switching a driver module, or it can require a complete rewrite. In some situations, identical code may produce functional equivalence but does not result in comparable performance or resource usage. Differences in data alignment and access can result in different performance for your algorithm. Differences in dynamic-memory-allocation services can result in differences in resource usage and performance.

Supporting low-power modes is an example of functional equivalence extending beyond the source code. However, if you have low-power requirements you should include them in the porting specification and test plan. Differences in memory maps and how data and program code is located in memory is another example of a functional equivalence consideration that extends beyond the source code. One approach to critical, tight loops is to locate them wholly in on-chip memory. You may experience noticeable performance degradation if you fail to note their location or to load and lock the code in a cache block or if the ported code grows so that it cannot fit in on-chip memory.

Scope creep, the process of extending the project beyond the original plan, is another pitfall that can increase the risk to your porting effort. If the project scope changes, it should trigger a change in the project-budget, time-to-complete, testing-criteria, and headroom goals. However, the safest approach for handling scope creep is to continue with the original porting plan and incorporate the change in a subsequent effort after you’ve successfully completed the port.

Going forward

The evolution of contemporary programming languages and tools is emphasizing processor independence and portability for faster development. Often, the more portable the code, the less optimized it is and vice versa. Although portable code may be slower and use more resources than optimized code, it is faster to develop than optimized-assembly code and easier to incorporate into other projects, and it can reduce the time it takes to complete your project. You may be staring at another port effort if you find yourself having to perform heavy optimization during a porting effort; that is, unless you are using a platform that is extreme along some performance vector.

Software prototyping can take much longer than hardware prototyping, even with extensive reuse of existing software. Processor and tool vendors are trying to simplify software-development complexity with their tools and abstracting software designs to be processor-agnostic. This effort encompasses compilers, operating systems, middleware, drivers, chip-support-library APIs, application frameworks, and version-control and building tools. These tools, along with a single source tree that uses portable code and conditional assembly and compilation where necessary, minimizes the effort of managing and maintaining the same software over multiple processing platforms.

Consider using a tool, such as lint, which acts like an overactive compiler syntax-checker. It applies much stronger standards for language correctness for C code that can identify nonportable code; help find logic problems, such as loops not entered at the top; and flag wasteful code, such as unreachable statements. Some lint engines can track abstracted type definitions and identify possible type misuse as the final types are resolved. Lint can also add value to large development projects, because it can examine all of the files in a build and spot inconsistent definitions and usage across files.

An industrywide, standard best practice for porting embedded applications is not apparent. Consider
how engineers and programmers are—or are not—taught to perform a software port. One suggested reason for the lack of formal training is that too many possible processor pairings exist for a standard approach to work. That argument does not apply to software development, so why would it be valid for software porting? Another suggested reason is that embedded applications tend to be small, and their efficiency exists in one or two tight loops. In this case, developers may feel the scale and complexity is not enough for a standard best practice to apply.

Working code is not the only measure of success. The portability and maintainability of your code affects your ability to reuse it, and the ability to reuse software can directly affect the success of your next project. Successful porting rarely happens by accident. Much like any successful software-development project, it requires adequate analysis, planning, and implementing a good trade between code that is perfect and good enough, along with balancing your time, money, engineering effort, and system reliability.

References
2. Cravotta, Robert, "Why tinker with your operating system?" *EDN*, June 27, 2002, pg 45.