Class #427 at the Embedded Systems Conference – San Francisco 2004

Designing Portable Software

Steven A. Stolper

An approach for designing multi-platform software

We live in a "disposable" society. We take pictures with disposable cameras and see by the light of disposable flashlights. We diaper our children, wipe up spills, and eat our meals without thought to the products we discard. Even our relationships don't endure as of old. According to the National Center for Health Statistics, 43% of all first marriages are discarded.

But, in the embedded software industry, software is becoming precious. Using the same software across different product lines, migrating it to different platforms, and leveraging it over time has become a business imperative. However, little is said about design approaches that address the special challenges and hidden costs of engineering portable software.

The Growing Importance of Portability

In the past, developers were not afflicted with the burdens of writing code that executed on a wide variety of platforms. A company that markets I/O cards, for example, might supply example programs or device drivers for a select group of real-time operating systems. But usually these were distinct software components that were maintained as separate products. Until relatively recently, entire embedded applications were not expected to execute on different hardware platforms, under different operating systems.

At a growing number of companies, developers are being asked to design embedded systems that can run on more than one hardware platform, and sometimes, under more than one operating system. The driving force behind this is the decreasing time between consecutive generations of a product. Not only has the duration of development cycles decreased, but the time between development cycles has decreased as well!

A highly competitive marketplace demands that the next generation of a product be designed even as the previous one heads out the door. Developers find they must leverage the software from the previous generation to accelerate the next generation's time to market. But advances in technology frustrate this goal. System designers select different devices (i.e.: FPGA's), subsystems, and CPUs to improve the next generation product. As a result, developers often face a porting issue that they never anticipated.

The economic downturn slowed the pace of development. But it has also brought into play other forces that promote multi-platform efforts. Companies now strive to cut costs by applying existing designs to a wider range of applications. A board that was the heart of a media gateway may now also become part of a wireless base-station.

Hidden Dangers

When software is not designed to anticipate these possibilities, the result can be more costly than launching a new design. Major overhauls to the code, constant support calls to the original software developers, duplication of testing effort, and major redesigns all combine to confound the original goals of cost saving and accelerated time to market.

The specter of multi-platform development is ever eager to snare an unwary development team. Its jaws are multifarious code changes and it claws eternal technical support. Its greatest weapon: The lack of development strategies that address multi-platform software development.

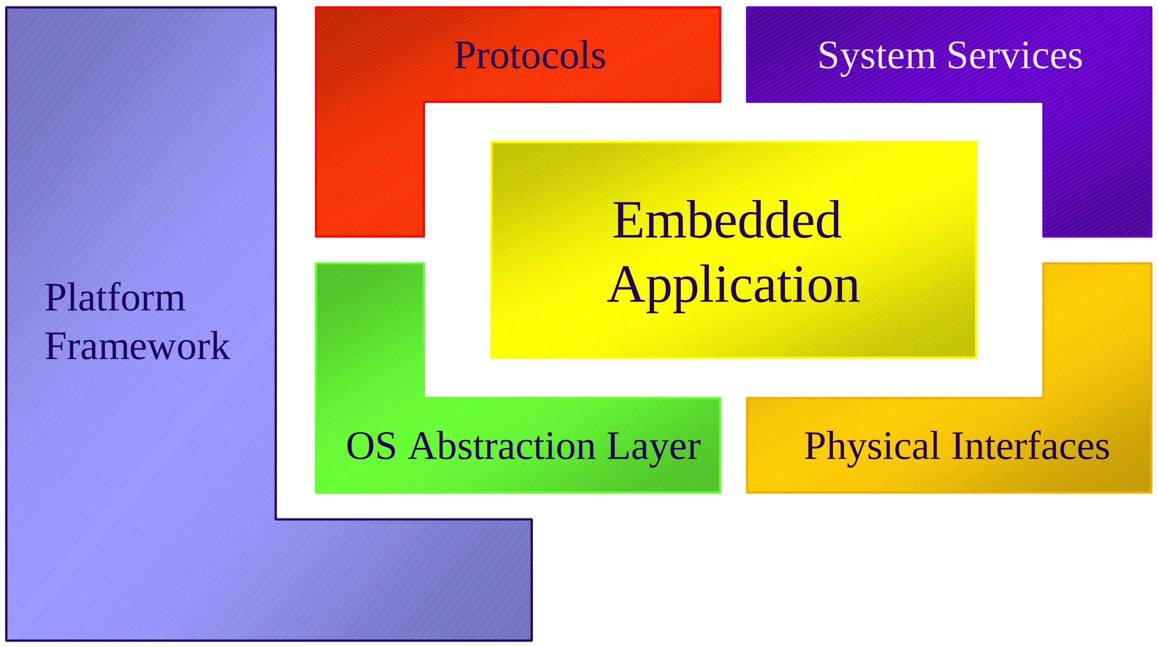
Designing Portable Software using Abstraction

Is software development an art or a science? Significantly, Donald Knuth's books were titled *The Art of Computer Programing,* not the *science* of computer programming. If software development is an art, then its highest expression is the creative way in which developers solve daunting technical problems, such as multi-platform development. One of the most creative ways to approach multi-platform development is through the strategic use of abstraction.

The elegance of abstraction is that it distils the application to its root concepts while isolating it from the details of the system implementation. The key is to compartmentalize the system such that the application's algorithms are decoupled from the implementation environment. A camera pointing system, for example, should operate on velocities, coordinates, and distances to target. It is irrelevant whether a radar device or an optical tracking subsystem generated the ranging information. Does the algorithm really need to know that some of the rate information was produced by a Hall effect sensor?

This approach is effective over a wide variety of applications. Just as the mechanical components of a system can be abstracted, so can communications interfaces, CPU resources, RTOS facilities, and storage devices. Deft use of abstraction simplifies the application into a compact, reusable code base. It is a powerful tool when applied against the problems of multi-platform development.

The following sections provide a guideline to follow when choosing abstractions to aid multi-platform development. As always, creativity, elegance, simplicity, and pragmatism should guide the selection of abstractions.



Components of a multi-platform software system

OS Abstraction

Never call OS functions directly. Divorce the application from the underlying RTOS by encapsulating operating system functions in an "OS abstraction" library. This permits developers to migrate the application to different operating systems simply by porting the OS abstraction layer. The application code remains intact.

Testing and quality assurance performed previously on the application is not wasted because the application code remains unchanged. The "hardening" that the software acquired in the field will carry over to the new platform and accelerate the transition to the new environment. Also, when bugs are discovered, the OS abstraction library provides a ready starting point to focus the debugging effort.

The OS abstraction library should export function prototypes and guarantee specific behaviors for them, regardless of the underlying operating system. For example, many operating systems have semaphores that are safe to *take* more than once from the same task context. Other operating systems will block the caller if a "recursive" semaphore take is attempted. If the OS abstraction library exports a function, **`OS_SemTake()`**, then it must proscribe a standard behavior for the function independent from the underlying operating system. In the case of a "recursive" semaphore take function, developers might have to implement the "recursive" behavior themselves. The implementation for some operating systems might involve comparing the ID of the task that last acquired the semaphore to the ID of the task attempting to take it. If they are the same, the function should increment a counter for that semaphore and not call the OS semaphore take function. The corresponding "give" function would have to decrement the counter each time it was called until the counter reaches zero. At that time, the OS semaphore give function is called to relinquish the semaphore.

Write wrappers even for functions you would expect to behave similarly on all operating systems. This protects the application from idiosyncrasies and even bugs in the underlying operating system implementations. "Standard" functions often aren't, and "well known" functions can make fortunes writing tell-all books about their secret lives.

The socket library function, **`select()`**, provides a good example of why encapsulation is so valuable. The types of devices that can be "select-ed" vary considerably from operating system to operating system. Some allow only sockets to be "select-ed," while others permit sockets, pipes, and message queues. Abstracting the underlying implementations protect the application against an unwanted, and often messy, redesign for another operating system. In one case, an operating system failed to implement the "timeout" feature of **`select()`** properly. Fortunately, the work-around could be confined to the abstraction library. Otherwise significant architectural changes to the application may have been required.

Logical Interfaces

Convert physical interfaces to logical ones. As an example, consider a system that has an engineering bus to which peripherals are attached. Create a set of functions that access the bus, such as an **`EngBus_SendTo()`** and **`EngBus_ReadFrom()`**, that effectively hides the implementation of the bus from the application. Whether the bus implementation is PCI, VME, 1553B or a serial port, is separated from the application logic.

This approach works extremely well for networked applications. If an embedded client engages in a "session" with a networked server over Ethernet-TCP/IP, abstract the concept of "session" and partition it from the IP connection. Allow the application to invoke functions that reflect the logical session, such as **`Session_Open()`**, **`Session_Close()`**, **`Session_Send()`**, **`Session_Receive()`**, and **`Session_BlockForSessionData()`**. This allows the application to migrate to a platform where the physical interface may be a paltry serial port. The concept of "session" has not changed, only the medium over which the session is conducted. If abstracted judiciously, the application can be embedded in different types of equipment with a variety of physical interfaces.

The same approach applies to applications that might be distributed among several processors in later generations of the product. Abstracting inter-process communications (IPC) mechanisms allow the software subsystems of the application to move freely from one processor to another. By partitioning the software in this manner, the application can scale from an “entry level” product to the “behemoth” model by adding additional processors to the system board. The converse occurs when “bridging” between generations of products. Software required to translate between different generations of hardware can be removed in later generations of the product when the older technology has been completely removed.

Commanding the system is abstracted in the same manner. Whether the commands are generated by command line interface (CLI), infrared link, or an embedded SNMP agent does not matter. The mechanism by which the commands enter the system should be divided from the set of functions invoked to execute the commands.

The same is true of event logging and general I/O. By distilling the logical connections from the physical, the application can be “re-wired” into dozens of different platforms. The advantages of this approach are a reusable code base that can be leveraged between different platforms as well as a consistent interface across different product lines from the same manufacturer.

Isolate Protocols

Separate the protocol implementation from both the transport medium and the system-specific details of the application. In our camera-pointing example, the pointing algorithm was isolated from the sources of its input data (the physical sensors). The protocol implementation can be isolated from the transport medium in exactly the same way. For example, implement a packet-based protocol as a code library that accepts packets as input and produces response packets as output. For connection-oriented protocols that maintain state, the library can also accept and modify data structures that preserve state information related to the connection.

This code organization provides the freedom to embed the protocol in any number of devices at a later date. It also permits the testing of the protocol to occur in isolation, which is an invaluable capability. The implementation can be debugged prior to the appearance of hardware and used later as a static test-bed to simulate failures in the field.

It is important not to place application specific knowledge into the protocol implementation. For example, an application-level protocol implementation that produces AAL5 packets cannot easily be placed into a device that uses IP. Similarly, an implementation that accepts a TCP socket descriptor, to which it sends responses, cannot easily be employed across a proprietary backplane.

Use System Services

System services are software components that provide a “service” to the application level software (See “At Your Service”, Steven Stolper, Embedded Systems Programming, April 2001). Developers employ them to abstract the details of the hardware platform into a standard set of capabilities used by the application. They can be used to manage non-volatile storage, provide highly accurate timing through hardware support, and manage processor resources. Similar services can also be used to manage relays, switches, or other peripheral hardware manipulated by the application level.

System services can also provide “software services” that do not depend on the existence of specific physical hardware. Services such as inter-process communications, software health verification, event logging, and time-stamping services can be rendered independently of specialized hardware.

This streamlines multi-platform development by leveraging the existing, well tested, application software. The application migrates to different platforms by porting the lower level services on which the application resides. It also allows the application code-base to be easily ported to more capable hardware when it becomes available simply by porting the underlying services.

Build a Framework

Another way to increase the portability of the application is to place all of the platform-specific initialization into one module. When the system boots, the “root” task invokes a platform-specific *framework* module to initialize and configure the system-specific hardware. Once the platform specific initialization is completed, the framework starts the application. The platform-specific code is executed outside of the application, which can run under any number of frameworks.

When developing applications that might be retargeted to other platforms, it is wise to consider what other development environments might be used to build the software. Companies with heterogeneous environments that consist of both Unix and Windows computers need to keep in mind differences between the systems. Some Windows environments will accept either slashes or backslashes in file path names. Unix environments require forward slashes. Unix environments are also case sensitive. When specifying header files, the capitalization must match the way the file appears in the file system. Also consider how Unix and Windows editors deal with special characters that may be embedded in the application code such as tabs, end-of-line, and carriage return. It could be disastrous to have to modify the source of a critical application after all of the testing has been completed.

Summary

Abstraction is a powerful tool for multi-platform development. Wisely employed, it can transform the potential for multifarious code changes, insidious bugs, and eternal technical support into a triumph of engineering - and even art!

Acknowledgements

Erik Guntvedt, Subbarao Mungara, and Randy Pan contributed valuable ideas and opinions that greatly influenced this paper.

References

The Art of Computer Programming, Volumes 1-3, Donald E. Knuth, Addison-Wesley Pub Co, 1968, 1973, 1997.

“At Your Service”, Steven A. Stolper, *Embedded Systems Programming Magazine*, CMP Media, Inc., April 2001

“Software That Travels”, Steven A. Stolper, *Embedded Systems Programming Magazine*, CMP Media, Inc., October 2002