USING POSIX FOR EMBEDDED DEVELOPMENT

Steve Furr (QNX Software Systems Ltd., Ottawa, Ontario, Canada, K2M 1W8)

ABSTRACT

More than ever, commercial, government, and military organi-zations are demanding that developers use POSIX interfaces. The question is, how much of the POSIX standard must your project support? POSIX is so large, and has so many optional components, that few applications need everything it offers. This session introduces basic concepts of application porta-bility and explores the benefits of using POSIX as a standard for achieving greater levels of portability. It examines how to: identify which POSIX APIs your system requires; determine whether your OS can support those APIs; achieve portability without sacrificing performance; and weigh the relative merits of POSIX conformance, POSIX compliance, and POSIX certification. Examples related to portability are examined from the perspective of various editions of the POSIX specifi-cations.

1. TODAY'S ENVIRONMENT

In today's realtime and embedded environments, the diversity of operating environments (e.g. RTOS, general-purpose operating system, realtime kernel) poses a real challenge to organizations deploying software for multiple programs or across multiple product lines.

Product perspective

A typical product company has multiple product lines with substantial software investments incorporated into the products. There is a strong desire to preserve these software investments when migrating features across product lines, or when creating subsequent generations of a product, possibly on new hardware and software platforms. All too frequently, this may involve a significant amount of recoding. Such recod-ing can substantially add to the development costs of the product and add delays in development that increase the time to market.

Unfortunately, there is often limited interoperability of the software used to implement important product features and functions. In many cases, different product lines or even models within a line may run on entirely different operating environments. This tends to limit the choices in building new products, or migrating features from one product line to another. Each product is susceptible to vendor lock, in which the product and most of its features become tied to the opera-ting environment on which its software is implemented.

In this environment, migrating features or upgrading a product to new hardware may often require extensive recoding when a change of operating environment is considered by necessity (e.g. where the hardware capabilities can't be sup-ported) or by design (e.g. advanced performance or reliability features offered by the new candidate).

Program perspective

From a program perspective, the issues are largely the same as for products, but often on a grander scale, since more deployments may be involved, and the complexity of each deployment may often be greater.

Most programs have a large software component — a component that, in many cases, is based on existing software repurposed for the new program. For this reason, software reuse has been a focus for many organizations over the last decade and even longer. There is a strong desire among system integrators and even their customers to build libraries of reusable software algorithms for use in new program deployments. To avoid substantial recoding for each new deployment program, portable software is required to make this strategy a success.

2. PORTABILITY

Portability is a characteristic of software (e.g. an application, library, framework, or other base of code) that indicates the extent to which the software is reusable without modification. A portable application is one that can be reused in different operating environments, whether they come from a different vendor or different versions of the same vendor's products.

Consequently, portability can be difficult to measure in any quantifiable fashion — so many variables may be involved. Even the fact that software has successfully migrated from one platform to another cannot be taken as a reliable metric of whether or not it is portable. Any assumptions that may have been built into the system interfaces may prove to be invalid the next time a port is attempted, even to a subsequent version of the same platform.

Portable software is most easily created when the system interfaces provided by the underlying operating environments have sufficient commonality. Greater amounts of commonality translate into fewer contingencies that must be taken into account to make code portable to different operating environ-ments.

3. API CONTRACT

Standardized application programming interfaces (API) represent one means to address the portability issue. If you think of the API as a contract between the implementation vendor and the application developer, then you can begin to measure portability by the extent to which the application stays within the letter of the contract. In this case, provided that the application developer only uses the API as documented, and the implementation vendor continues to produce operating environments that behave as described, the code will continue to be portable to all versions of the same platform.

Industry-standard API specifications can preserve software investments by extending the contract across a broader base of implementations. If the application developer creates *conforming applications* that stick to the API contract, then their code will be portable to any *conformant implementation* of that specification.

Applicable industry-standard API specifications preserve software investments by enabling the creation of code that is portable to multiple platforms from different vendors. While an incremental level of effort is required to develop portable software, recoding isn't necessary to migrate to other conformant implementations, reducing the development risks, costs, and time-to-market. By choosing a widely used, industry-standard API, product companies can also reduce training costs, increase productivity, and leverage a larger pool of programming expertise.

4. TYPES OF PORTABILITY

Consideration of portability as a contract between the application and the implementation introduces additional factors that determine whether the application can be considered to be a conforming application. These take into account the two elements of portability that a conforming application must possess — intrinsic portability and conceptual portability.

Intrinsic portability

Intrinsic portability refers to the portability of the application in terms of its syntactic use of the interfaces specified by the API specification. Intrinsic portability can be readily verified against the source code using static analysis tools.

It is a relatively straightforward matter to verify whether the application makes use of any facilities that aren't specified by some component of the API specification. This can be considered a breach of the API contract. In most cases, an alternate standard facility should be used in place of the violating construct indicating that the application is non-conforming, and that the code is non-portable.

Taking the POSIX specification as an example, there are two primary aspects to intrinsic portability: the facilities actually used by the application through function calls, and the members referenced in structures associated with any of the facilities of the system interface.

For portable POSIX code, the UNIX *getdents*() system call should be eschewed in favor of the more standard POSIX directory reading mechanisms.

The second aspect deals with the use of structures passed to system interface functions. For example, two structures commonly used in conjunction with the file management services in UNIX and POSIX are the `dirent` and `stat` structures.

POSIX mandates only one element for the `dirent` structure: `d_name`. The rationale is that directory access, for which the structure is used, is intended to be implementation independent. The historical UNIX `d_fileno` or `d_ino` member is excluded because it normally represents the i-node number or file serial number for a UNIX file system. As such, it is specific to a particular file system implementation and would be irrelevant in other contexts, such as when applied to a FAT-based file system on an Intel machine running Win32. Since only system programs need to access the file serial number, it is deliberately left out of the POSIX specification and portable programs should not use it.

Similarly, the `stat` structure contains the `st_rdev` field under UNIX. This field is specific to the implementation of character and block special device files under UNIX and has no meaning outside of this context.

As with the use of non-standard facilities, use of non-standard structure members of this form constitute a breach of the API contract, indicating that the code is not intrinsically portable.

These two examples clearly illustrate how these small violations represent a larger portability problem regarding the implementation-defined interpretation of the data. This is essentially the problem of conceptual portability.

Conceptual portability

Conceptual portability is concerned with the manner in which the system interface facilities are actually used. The main focus here is to determine whether a code construct in an application makes any assumptions about the behavior of facilities in a manner that is unspecified, undefined, or implementation defined.

The descriptions of these terms are representative of the way they are generally defined in most API specifications and industry standards:

- Unspecified — Correct program constructs whose semantics aren't completely specified, but which must be performed correctly by the implementer, leaving some latitude for the method chosen. In standard C, for example, the most notable unspecified behavior is the order of evaluation of some expressions.

- Undefined — Constructs that are illegal but may be difficult to detect. API specifications don't impose any restrictions on these, so the implementations are free to handle them in any way the implementer sees fit.
- Implementation-defined — Legal constructs whose exact behavior is left to the discretion of the implementer using any appropriate approach, *providing that approach is explained to the user*.

Undefined behaviors are often the most pernicious of these, since there is no indication of how an implementation will respond to their use. You might expect applications to behave normally and continue to execute with unexpected results or errors, but this isn't always the case. Such behavior may actually result in abnormal termination of the application.

Conceptual portability is harder to determine than intrinsic portability. It involves a careful examination of the use of a sequence of code constructs to determine if it makes assumptions about implementation behavior that isn't specified by the API specification. Code reviews are an important tool for identifying portions of code that aren't conceptually portable. They can pinpoint constructs that are specifically identified by the API specification as unspecified, undefined or implementation-defined behavior.

A good example of this is in the area of directory operations. POSIX provides a directory stream corresponding to a named directory for examining the contents of that directory. The directory stream is obtained by calling *opendir*() and destroyed by calling *closedir*(). Successive calls to the *readdir*() function return a representation of each entry in the directory and *rewinddir*() restores the directory stream to its initial state. The directory stream may be implemented in any suitable manner and the standard doesn't specify whether all entries have to be on the same device.

We have already seen that the historical UNIX `d_fileno` member has been excised from the POSIX.1 specification for the `dirent` structure that is used to represent directory entries when reading directories. UNIX programs often perform comparisons using the file serial number in the directory entry returned by *readdir*() or *getdents*() to locate a particular file, or to identify if the directory entry represents a link to the same on-disk file as another entry. The portable way to make this comparison is to perform a *stat*() operation on the file represented by the directory entry. In this case both the `st_dev` and `st_ino` file characteristics should be compared with the target file because the files represented by the entries may reside on different devices. For implementations or file systems that don't support links, the comparison will be superfluous and won't affect the application's behavior.

The directory reading operation doesn't specify that "." or ".." must be returned by a *readdir*() or in what order directory entries will be returned. Portable code has to be able to cope with any ordering of the directory entries and should be able to deal with "." and ".." but not require them. Furthermore, there is no guarantee that the same ordering will be used in any subsequent reading of the directory. To allow for this possibility with unconventional file system implementations, seeking to a particular position in a directory isn't part of the standard. Portable POSIX.1 applications shouldn't use the UNIX *seekdir*() or *telldir*() system calls. The *rewinddir*() func-tion could be used, but is nonportable in this context.

Directory operations are just one of the many areas your coding standards must deal with. Each type of facility your application requires should have a set of guidelines regarding their use and highlighting these types of unspecified behaviors.

5. APROACHES TO PORTABILITY

A number of approaches may be taken to ensure that a code base is portable. Two important ones are discussed here, relating to judgments of how well the application compares against the API contract(s) that apply to all the API specifications.

Thinking of them in contractual terms, they generally take either a normative approach (which assesses portability from a course of performance perspective) or an empirical approach (which assesses portability from a course of practice perspective).

Course of performance considers whether the application is in strict conformance with the rules of the API specification (the contract). Course of practice considers how well the application conforms relative to industry norms, or to some acceptable level of expectations.

Normative — conformance

The normative approach dictates that all portable code will qualify as a *strictly conforming application* — one that makes use only of facilities fully defined by **one** of the API specifications that govern the implementation on the target platform.

A benefit to this approach is that it provides a simple pass/fail metric of code portability. An application or code base can be defined as either fully portable or not, based on whether it has been verified as strictly conforming.

If the normative approach is to provide a verifiable metric of portability, three premises must hold true:

1. There must be an *application environment profile* (AEP) completely defining the set of API specifications that the code is tested against as strictly conforming.
2. The code itself must be verifiable as strictly conforming.

3. The implementation must be conformant to the application environment profile — it must implement all of the chosen API specifications as defined.

Pragmatically, the normative approach is useful only if a suitable AEP can be selected — or alternatively, a scalable set of profiles — for which there is a suitable variety of implementations. Only then can the code be retargeted to all of the types of platforms required to fit its intended use.

Empirical — contingency

The empirical approach is somewhat harder to directly measure but it takes into account the diversity of implementations. It recognizes that no API specification may be able to account for all of the facilities or implementation-dependent behavior that may be required by an application.

It loosely allows for extensions that rely on platform-specific facilities, or deviations from specifications — that rely on unspecified or implementation-defined behavior — provided that the behavior is well documented for a target platform implementation.

Portable code designed for this approach will usually operate conditionally based on the contingencies introduced by variation in behavior for different implementations of unspecified, implementation-defined behavior or facilities outside of specifications. In many cases, there may be conditional compilation of platform-specific pieces of code.

One advantage to this approach is the ability to leverage the knowledge base of an organization or the industry at large where the different contingencies for various implementations have been codified in some form.

Hybrid approach

A hybrid combining elements of the two approaches is viable. The hybrid approach adopts the AEP definition or selection from the normative approach and imposes conformance criteria on applications. The conformance criteria are more lax than those for a strictly normative approach. They may require that an application be a *conforming* application, but not necessarily a *strictly conforming* application. A conforming application is allowed to make use of unspecified or implementation-defined behavior under certain rules.

A hybrid model may also allow for extension — use of features not defined by the AEP — by a *conforming application using extensions*. A hybrid methodology, if implemented appropriately, will provide a variance plan that gives guidelines or conformance rules for using extensions in a manner consistent with the AEP.

Variance plan

Occasionally, it won't be possible to achieve the desired result on a supported platform using the standard interfaces provided by the AEP. The role of the variance plan is to ensure that these situations are handled consistently. This consistency lets you maximize the likelihood that the same solution can be employed on multiple platforms.

The variance plan should identify the situations that constitute exceptions to the normal usage of the AEP and how the situation should be resolved. The latter should be explicit, identifying an idiom to be employed and the manner in which the application is configured to make use of that idiom. This is an opportunity for the variance plan to document best practices taken from the industry or internal to the organization. Autoconf and configure scripts are a good example of the effective codification of best practices in the open source community. Autoconf and configure provide a knowledge base of programming idioms and tools to enable code portability across a wide variety of platforms.

Common ground

Regardless of the approach taken to portability, its success will rely on finding some common ground between the system interfaces provided by target implementations. Finding a rele-vant set of API specifications to incorporate into an AEP may draw upon industry standards, or follow industry norms by adopting de facto standards or widely available specifications.

The selection of appropriate API specifications can greatly influence the amount of effort that needs to be invested in creating or modifying applications for portability. Even with tools to assist the preparation of contingently portable code, greater standardization allows fewer lines of conditional code to be written to accommodate differences in implementation.

The selection of the system-level interface is one that can have the greatest impact on overall code portability.

6. STANDARDS & PORTABILITY — POSIX

The system interface provides access to fundamental system services such as file management, device control, multi-programming and inter-application communication. The system interface should be your largest source of concern in obtaining code portability to a wide variety of platforms.

The most important portable system interface specification is the Portable Operating System Interface (POSIX).

What is POSIX?

POSIX originally started out as a family of standards that focus on the system interface specification. More recently, the various standards that POSIX comprises have been consolidated into a single specification. The POSIX specification is unique in that it provides an *interface* to operating system services. As such, POSIX doesn't favor any particular implementation mechanism.

The primary goal of the specification is source code portability so it doesn't impose any binary compatibility restrictions, allowing for system-dependent extension. POSIX is designed to be the minimal interface required to provide the set of system services covered. Factored together, these features allow POSIX to be implemented in a number of ways on a wide range of platforms including:

- systems derived from UNIX
- independent implementations designed to the POSIX specifications
- emulations hosted on different operating systems

The POSIX specification emerged in the 1980s from efforts in the user community to obtain a consistent set of system interfaces from the vendors of the many derivatives of UNIX. The POSIX specification was formally developed by the IEEE, and eventually ratified as an ISO standard.

A completely new edition of the specification was introduced in 2001, jointly developed by the IEEE, ISO, and the Open Group. This version of the standard, as amended in 2003, fully incorporates all of the previous POSIX family of standards. It represents the base level of capability for any new POSIX implementation. The POSIX specification is also the base for specifications such as Linux Standard Base and the Embedded Linux Consortium Platform Specification.

Where is POSIX used?

Because it is focused on source compatibility and is largely agnostic to implementation approaches, POSIX can be broadly implemented across a wide range of systems, including:

- current major systems that are ultimately derived from the original UNIX system code (Version 7 or later)
- compatible systems that aren't derived from the original UNIX system code
- emulations hosted on entirely different operating systems
- networked systems
- distributed systems
- systems running on a broad range of hardware

API evolution

As application demands have risen and new facilities have required standardization (e.g. thread programming, synchronization, etc.), the POSIX specification has grown through amendments. The incorporation of the Single UNIX Specification into the POSIX standard has also resulted in the adoption of a large number of standardized UNIX interfaces.

The end result is that the overall scope of the specification has grown from approximately five hundred interfaces in the 1996 edition of POSIX to close to two thousand today.

The specification is scalable in that vendors need only implement a mandatory set of features that represent the base, but they may select which feature sets they wish to implement from the remainder.

7. APPLICATION ENVIRONMENT PROFILES

The definition of an application environment profile takes into account several factors. The class of application imposes demands on the facilities required of the system. Constraints may be imposed by the nature of the target market — as in the embedded market — that might preclude the AEP from specifying complete UNIX facilities. A particular customer or corporate policy may also require conformance with one or more formal standards.

At its simplest, an AEP may consist of nothing more than references to the appropriate set of base specifications (the standard and the edition) as determined by the runtime requirements of the application. This might be the case for a simple application with minimal runtime requirements that could be satisfied using ANSI C, the standard C library. and POSIX, but in practice the AEP will usually be more complex.

One difficulty with POSIX is that a number of facilities are optional within the standard, and other facilities aren't covered at all. A POSIX implementation only has to meet the POSIX system interface specification in order to claim POSIX conformance. POSIX base functionality on its own can't support a broad range of applications unless a suitable profile is specified and other facilities are added.

For example, POSIX doesn't require an implementation to allow a user to belong more than one group of users at any given time, but your profile may require this feature, known as *supplementary groups*, for correct application behavior. If so, your definition of an application conforming to the AEP would take this into account.

Part of the solution is to use a profile that specifies which elements of POSIX are to be treated as mandatory. FIPS-151, a federal information processing standard developed by the National Institute for Standards and Technology (NIST), is one widely adopted profile based on earlier editions of the POSIX specification that did just this. It requires job control, imposes a minimum number of groups to be supported (8), and restricts the use of *chown* to privileged users. A profile like FIPS-151 will cite the specification base and define the feature sets from the base specification that are mandatory

Conforming implementations

As we have seen, conformance to POSIX specifications is largely a matter of degree, since not all features are mandatory. The precise set of features available in an implementation depends on the specifications to which the vendor claims conformance.

In addition to some of the profiles previously discussed, the POSIX minimal application profiles define small subsets of the POSIX specification suitable for different categories of embedded devices. These were originally released as POSIX specification 1003.13. This specification has recently been amended for incorporation into the 1003.1-2001 specification.

POSIX compliance: OS comparison

When selecting a POSIX implementation, it is important to consider the vendor's conformance statement or compliance statement. A vendor's conformance statement may apply to the POSIX base specification for a particular edition of the specification (e.g. 1003.1-1996, 1003.1-2001), or to a specific profile.

There are four profiles defined in the POSIX minimum application profile specification for different categories of embedded devices, each of which mandates implementation of a specific suite of POSIX feature sets. These four profiles are normally identified as PSE51, PSE52, PSE53, and PSE54. You should be aware of what each of these profiles provide, as most embedded operating environments are compliant to one of them.

POSIX specifications: OS comparison

UNIX systems, Linux systems, and a select number of real-time operating systems will be compliant to a larger vendor-defined profile of POSIX. In these instances, it is important to consider the base specification, as well as all of the feature sets they claim to support.

Selection Factors

A number of factors determine whether a profile or choice of operating environment is applicable to a specific application domain. All of these questions are particularly suited to embedded environments where there is considerable variation in the levels of implementation.

Some of the factors to consider here are:

- Realtime — Does this choice offer sufficient support for clocks & timers, priority-based thread scheduling, realtime signals, interprocess communication, and synchronization?
- Reliability — If the target device will run multiple applications concurrently, can the applications be isolated (memory protection) from each other? Do they need to be?
- Persistent storage — Does the choice offer standard file system interfaces?
- Security — Will the device support multiple users? If so, does access to resources need to be controlled on a user-by-user basis?
- Concurrency — Is multiprocessor hardware used, and does it support SMP? If so, additional synchronization primitives, such as spinlocks and barriers, may be needed for maximal performance.

The appropriate definition of these selection criteria, as well as the selection of mandatory feature sets for an AEP — on the basis of the outcomes — can be a major contributor to the usefulness of the AEP.

8. CONCLUSIONS

Developing portable applications is a complicated process that involves careful consideration and planning. A regimented approach using an AEP, coding guidelines, and a variance plan provide a framework for communicating the results of that process to developers, allowing them to produce portable code in a more effective and consistent manner. The material required to produce effective coding guidelines is usually contained within the standards themselves. Automatic configuration tools such as Autoconf provide valuable infor-mation about how to configure software for different target platforms to deal with implementation dependencies.

A solid specifications base enables organizations to maximize their code reuse with the least amount of effort. The POSIX specification is key to enabling application portability across a wide range of systems, from UNIX and Linux servers to deeply embedded devices. Appropriate selection of implementations requires proper consideration of what feature sets may be necessary for the application category.