SUPPORTING ARTICLE

The POSIX Family of Standards

Stephen R. Walli
SRW SOFTWARE, KITCHENER, ONT.

■ **The IEEE POSIX family of standards has been developing since 1988, when the original base system interface standard was ratified. Having been at the core of a number of vendor-based consortia specifications, largely due to U.S. government support in purchasing, it is widely implemented.**

Despite its history, POSIX still suffers from a misunderstood scope, bad press, and an apparent lack of use. However, when POSIX is viewed as a programmer's specification and simple explanations for using specific POSIX standards are given, a very different picture emerges, and the nature of the poor perceptions can be placed in context.

POSIX is a family of IEEE standards that supports portable programming. There are actually more than 20 standards and draft documents under the POSIX umbrella, at different stages in the standards development process. In this article each piece of work will be referred to as "POSIX.n," with "n" denoting the document or its working group, and enough other context will be given to identify the work being discussed. If the document is a completed formal standard, its official number will be cited. Official numbers look like "IEEE Std 1003.n-yyyy," where 1003 is the IEEE POSIX project, "n" denotes the document, and "yyyy" is the year the standard was completed or last amended.

The term "POSIX" is really quite vague, which probably contributes to the confusion surrounding the standards. To some it's the POSIX.1 system interface standard; to others it's the POSIX.1 standard plus the POSIX.2 shell and utilities standard. From the IEEE standards administration's point of view, POSIX is the entire body of work, draft documents, standards, working groups and all.

The first standard, POSIX.1 (IEEE Std 1003.1-1990), defines the C language interface to operating system services. The POSIX.1 standard describes its own scope best:

> [POSIX.1] defines a standard operating system interface and environment to support application portability at the source-code level. It is intended to be used by both application developers and system implementors. [IEEE 1990]

POSIX.1 describes interfaces to operating system services at the source-code level. It describes syntax (in the C language) and behavior, not the implementation of the interface by the operating system. This allows operating systems as diverse as the Digital VMS, IBM MVS, Unisys 2200, and CTOS architectures, HP MPE/iX, and Sun Solaris to implement the interface specification in whatever manner is appropriate.

POSIX.1 is defined to support application portability. It defines the interface syntax and the behavior of system services for such fundamental tasks as process creation and termination, process environment

Despite all the good work that has been accomplished, there is a perception that POSIX is failing.

manipulation, file and directory access, and simple I/O, and it extends part of the standard C library.

POSIX.1 was originally ratified in 1988, and was amended in 1990 to become the internationally accepted standard ISO/IEC 9945-1:1990. The IEEE 1990 edition is functionally the same as the IEEE 1988 edition, although it was reformatted and certain informational appendices were changed. The 1990 ISO and IEEE standards are identical, except that the ISO book lacks the colorful cover of the IEEE book.

Another important POSIX standard, POSIX.2 (IEEE Std 1003.2-1992), describes a programmable shell (command interpreter) and related utilities. Similar in scope to POSIX.1, POSIX.2 exists to support portable shell programming and a portable program development and user environment.

The creation of POSIX.1 and POSIX.2 was strongly influenced by existing UNIX practice and experience. The work began as an effort in the early 1980s to standardize the rapidly diverging UNIX operating system interface. In 1985 the IEEE standards board sponsored the work and the first IEEE POSIX committee was formed. With these two standards, plus the ANSI/ISO C standard, a wealth of applications can be developed in a portable way.

Within the IEEE POSIX committee, there are approximately 22 POSIX-related standards projects, sponsored and assigned to a dozen working groups. Like POSIX.1 and POSIX.2, other standards have also completed the IEEE standards process: POSIX.1 interfaces in Ada syntax (POSIX.5, IEEE Std 1003.5-1992); FORTRAN-77 syntax (POSIX.9, IEEE Std 1003.9-1992); a test description methodology (POSIX.3, IEEE Std 1003.3-1991); and test methods for POSIX.1 (POSIX.3.1, IEEE Std 2003.1-1992); POSIX.4 (IEEE Std 1003.4-1993), describes additional system interfaces to support the portability of real-time applications, and will eventually be published as an amendment of POSIX.1.

Other work continues on such items as security interfaces, sockets, and TLI interfaces for interprocess communications and profiling projects that attempt to describe application domains in terms of collections of standards.

Why Does POSIX Appear to be Failing?

Despite all the good work that has been accomplished, there is a perception that POSIX is failing or has somehow already failed. A large part of this myth developed because the IEEE did not market its standards, while large vendor consortia, such as the Open Software Foundation (OSF) or X/Open, have special marketing departments to promote their solutions. However, while vendor alliances count on their individual marketing departments to sell their solutions, POSIX standards are developed in an accredited, consensus-based standards process, open to all, and stand on their own. There is no IEEE Standards Office marketing group to promote POSIX as a solution or to deal with perceptions created by countermarketing from other vendor-based groups.

Excuses are often made for not using POSIX in various situations. A number of frequently used pretexts along with their refutations are given below.

POSIX TECHNOLOGY ISN'T STABLE ENOUGH

The base system service interface standard (POSIX.1) was ratified in 1988. There was a U.S. government (NIST) procurement standard (FIPS Pub 151-1), and certification process and test suite in place for POSIX.1 at the time. POSIX.1 is widely implemented, and NIST certification for FIPS Pub 151-1 numbers quite a few different platforms [NIST POSIX 1994].

POSIX.1 was amended in 1990, and became both an IEEE and an ISO international standard. The NIST process for testing and certifying POSIX.1 implementations to the NIST FIPS profile was updated (FIPS Pub 151-2) to point to the 1990 amended standard, and a better test suite was put in place based on experience with the first suite.

The POSIX.2 shell and utilities standard was ratified in 1992, and with minor textual changes was internationally ratified in 1993 by ISO, i.e., IEEE Std 1003.2-1992 is wholly equivalent to ISO/IEC 9945-2:1993. As of this writing, a test suite to measure conformance to POSIX.2 is near completion, sponsored by X/Open, and a NIST FIPS is complete. NIST will likely (though not necessarily) use this test suite for a POSIX.2 certification process.

When X/Open's portability specification evolved to Issue 3 (XPG3) in 1989, its stated direction was to support standards created in a consensus-based process, and POSIX.1 and POSIX.2 were cited. Both of these standards formed the core of Issue 4 (XPG4) of the portability guide when it was published in 1992. Indeed, the U.S. government FIPS options of POSIX.1 were specifically called out. The XPG3/4 branding processes test for POSIX.1 conformance. X/Open quickly points out in its marketing the number of procurements that are based on XPG4.

This is very stable technology. Where POSIX is not stable is within the IEEE working groups, the cascading set of amendment projects, and other less evolved projects that are still under construction. There is, however, a stable core of trustworthy, testable work that has existed for six years.

POSIX ISN'T COMPLETE ENOUGH

Yes, and speaking specifically, POSIX.1 defines the base operating system service API. That is all it is supposed to describe. This stable specification has existed for six years.

Saying that POSIX.1 "isn't complete enough" is like saying one cannot write useful portable applications adhering to ANSI C. POSIX.1 defines the operating system service behavior and syntax for the C language. Using POSIX.1 plus ANSI C, one broadens the boundaries of the portability model to include applications that can manipulate common operating system objects.

If one wants to define a standards-based portability model for applications development that also accounts for other aspects of the application, such as its GUI or its interprocess communications across a network, then one will have to review other specifications for inclusion in one's portability model.

POSIX.1 has a very specific scope. Other POSIX standards that amend POSIX.1 will broaden its scope; however, it is a specification of an interface to operating system services. It is not intended to be a complete specification of a library of every possible desirable functional interface a programmer might want to use.

POSIX IS SO LARGE THAT IT IS TOO COMPLEX

This is the reaction many take when they see a table of POSIX working groups and projects. There are a lot of working groups. Many of these projects amend one another in some fashion. Others are attempting to profile the work of groups that are still drafting documents. International synchronization, test methods synchronization, and the programming-language-independent specification work can confuse an already complex picture.

There is enormous political pressure from users and the international community, attempting to accomplish their own objectives. If allowed, POSIX will become so heavy as to cave in under its own weight, and many vendors would be quite happy with this situation. But again, there are a number of stable mature standards that can be used today.

POSIX STANDARDS TAKE TOO LONG

The IEEE standards board sponsored the first POSIX project in 1985. The POSIX.1 standard was ratified three years later. The Ada POSIX community created and ratified their language mapping to POSIX.1 in four years. The FORTRAN 77 POSIX community accomplished a similar goal in a similar time frame.

In a number of situations events have conspired to keep individual working groups from accomplishing their stated objectives on time. POSIX.2 (shell and utilities) spent entirely too long in ballot; but it is a standard of remarkable proportions. The blind agreement to synchronize with the ISO working group responsible for POSIX standardization was already in the process. The POSIX.4 (real-time) working group went to ballot late in 1989. It didn't complete ballot until 1993, primarily because the volunteers from the working group on ballot resolution shifted positions in the real world, so many of them were unable to participate once the draft document was in ballot. POSIX is a consensus-based process of *volunteer* individual professionals.

Another way to look at this criticism is that creating a consensus standard *should* take a certain amount of time. The people actually volunteering their time at POSIX meetings need time to present, debate, and digest proposals that will or will not end up in a standard specification. A good standard should have a certain perceived long-term value. If the standard is irrelevant a year after it is created, then it doesn't address the problem it was intended to solve.

There is tremendous pressure to speed up the process, as if more standards faster will some how solve the applications development problems in our industry. This false hope is truly confusing. Creating more specifications seems pointless, since the industry hasn't taken the time to understand and use the standards it already has.

POSIX WILL NO LONGER BE RELEVANT

This perception is a variation on the "POSIX isn't relevant because it takes too long" theme. The IEEE POSIX project-sponsoring committee attempts to use existing practice as criteria to determine whether or not a project should be sponsored. Existing practice and experience is a reasonable indication that the technology that is about to be captured in a textual specification is mature and stable enough to become a useful long-term standard. Attempts to develop standards based on technology that is scarcely developed are driven by vendor marketing hype.

Technology is changing remarkably quickly. POSIX.1 and its amendments exist to provide a portability model that allows an application developed against the POSIX specification to be portable to new platforms that support the same POSIX model. Thus, the investment in developing that application is maintained over a longer period of time. Software is the expensive part of the IS budget. Portability standards should not be viewed as a solution to unborn applications, but as a way to preserve the investment in applications that already exist. Porting applications to new platforms preserves the investment. Instead of porting within a line of computer architecture (such as moving from one VAX running VMS to a newer-faster-cheaper VAX, with a new version of VMS supporting new large-scale media), ANSI C, POSIX.1, and their kin address the desire to expand beyond a single vendor's borders.

If a new technology that will give you some benefit (tangible or otherwise) is discovered, the economics of applications development says that it will be used to best advantage. This might be a new way to develop graphic user interfaces faster and more cheaply, or a new set of functional library calls sup-

The commitment to portability is not a commitment to history.

plied on a new release of a vendor's architecture. This situation has not changed since the time applications were developed on a single architecture.

The decision to break an established portability model deliberately under appropriate circumstances has nothing to do with the future economics of porting the application to new platforms. If immediate deployment of a new application is a critical benefit with a know cost to a corporation, it will be deployed, and the corporation will live with the less tangible, less known, future costs and concerns.

X.H 1. The Practical Application of POSIX Standards

A wealth of useful applications can be written using ANSI/ISO C and its definition of the standard library interfaces. These applications can be compiled on any system with a compiler that supports the ANSI C specification, making them more portable. POSIX.1 extends the boundary of "useful applications" and "portability" by defining a standard interface to operating system services.

To use POSIX standards to advantage, a number of issues need to be understood. A portability model must be envisioned, including its use and limitations. If POSIX is to form the basis of the portability model, there needs to be an understanding of what conformance means and how to use specific certifications and branding programs to advantage. There also needs to be an understanding of how multiple platforms will affect source management and the construction environments.

PORTABILITY MODELS

An organization needs a model for portable applications design and development. This model must fit the objectives of the organization with respect to the platforms it already has, what applications will be moved, when, where—i.e., to what platforms.

Developing applications that are portable means having an idea of where they will be ported, which becomes more difficult to predict the further into the future one goes. Hardware technology is changing rapidly, and new software technologies are appearing all the time. Developing applications is still an expensive task, however, and an organization should have confidence that the technological base for the source code will be stable and as long-term as possible.

This commitment to portability is not a commitment to history. Indeed, POSIX as an interface specification is powerful precisely because, while it is based on historical practice (i.e., proven ways of accomplishing some end), it is an interface. As vendors deliver new platforms that support POSIX interfaces to market, they encourage an organization to utilize those platforms while protecting its investment in its developed applications.

Because the portability model determines where platform dollars are spent in the future, an organization should, whenever possible, ensure that the portability model is conveyed to its primary vendors so they know where it wants to go. If an organization wants to restrict itself to certain IBM architectures, IBM's SAA would be a perfectly acceptable portability model. If vendor neutrality is an important aspect of the portability model, POSIX is the broadest choice available.

A number of specifications are often mentioned as portability models. The System V Interface Definition (SVID) for example, defines the interfaces of System V UNIX, although it tends to be limited to platforms implementing a System V UNIX derivative.

X/Open has developed the X/Open Portability Guide, currently in its fourth issue (XPG4). X/Open formed in 1984 when a group of European vendors defined a specification for portable application development based on the UNIX System V interface definition. By the time it matured to its third edition (XPG3), it included a broad spectrum of functional interfaces.

Because X/Open recognized the importance of accredited consensus-based specifications such as POSIX early, they participated closely in the development of POSIX standards. When the fourth edition of the portability guide (XPG4) was released in 1992, POSIX.1 and POSIX.2 figured prominently at its core. Vendors of a number of non-UNIX operating systems are working to extend their POSIX.1 and POSIX.2 environments to be XPG4 conformant.

Vendors of UNIX workstations and other desktop solutions began forming joint market initiatives and alliances to combat the perceived threat of MicroSoft's NT project. In the spring of 1993, a group of hardware vendors moved to "unify" UNIX and bring an end to the "Unix Wars."

This work centered around a specification known as Spec 1170, a grab bag of programming interfaces from several different flavors of UNIX. Some interfaces from one platform perform the same functions as differently named interfaces from another platform. The *intent* of this specification is to make it possible for the currently successful third-party software houses to quickly port their applications to all of the hardware platforms, so that the platform vendors could be seen as providing application-rich platforms.

X/Open turned Spec 1170 into the Single UNIX Specification, a five-volume superset of the XPG4 specification. X/Open is now the exclusive licensor

of the UNIX trademark, a privilege it obtained from Novell, which bought both the trademark and the UNIX source code from AT&T.

POSIX.1 and POSIX.2 remain at the core of both XPG4 and the Single UNIX Specification. An organization runs the risk of developing code that is less portable than originally planned, however, if it does not clearly identify its portability model.

Microsoft suggests that NT is also a portability specification, since the NT operating system runs on more than one architecture. However, this is confusing "product implementations" with "programming specifications." The UNIX vendor community demonstrated that providing an operating system on multiple architectures controlled and licensed by a single vendor (AT&T) is not what the consumer wants as a definition of an "open systems" specification or a portability model.

POSIX.1, however, offers a broadly implemented portability specification for operating system services. Many applications today also support a GUI and communicate across a network. Indeed, with the push to client/server application models, software portability models and applications architectures can change to suit servers running on multiple different platforms and clients running on still different platforms.

To extend the portability model to account for other functional aspects of an application requires diligent care. There is no magic answer to certain hard problems. Clues can be found in POSIX, if there is a commitment to consensus-based, vendor-neutral standards. With respect to network communications, POSIX.12 is working on specifications for Berkeley sockets and X/Open's XTI (based on System V TLI). Sockets are supported on almost every platform from PCs to MVS.

The choice of POSIX.1 as a base for a portability model means that an organization understands certain ideas and terminology with respect to conformance, testing and certification, and legal variances allowed by the standard.

CONFORMANCE TO POSIX

Each POSIX standard has a conformance statement at the front that describes what criteria an implementation must meet to claim conformance to the specification. It is a surprisingly simple statement. All the required interfaces must be implemented so that they behave as specified, and all decisions must be documented. This documentation is captured in the POSIX conformance document, and its format is also described in the standards.

Nothing else need be done. "Conformance" is the important term. Vendors often say they "comply with POSIX" or "follow POSIX." These statements are meaningless. A vendor "conforms to IEEE Std 1003.1-1990," documenting its conformance in a POSIX.1 conformance document.

The IEEE requires no test suites to guarantee conformance. First, it is not a funded testing laboratory. Second, an implementation of an operating system service interface standard is extremely difficult to test completely, and so guarantees of conformance are impossible. (A number of testing alternatives will be discussed later.) However, it is a good idea to always ask for the conformance documents, since they demonstrate that the vendor has at least done its homework.

OPTIONS AND LIMITS

There are points in the consensus process where two technical camps cannot reach an agreement based on existing practice and experience about the single best way something should be specified. There are a number of ways a working group manages this impasse.

A specific option can be created to call attention to behavior, which would allow an implementor to support or not support the option, while still claiming conformance to the standard. The implementor must document that fact in the conformance document. Support for job control is an option in POSIX.1.

Because constants are defined for certain items with certain minimums set in the standard, an implementor of the standard needs to identify values in the conformance document. For example, the minimum length of a filename in POSIX.1 is set to 14 characters.

In certain areas, the creation of a very concise language allows two existing behaviors to conform, or at least doesn't disallow it. However, this language cannot be counted on in a perfectly portable way.

These areas of legal variance exist due to a consensus process that is geared to serve the majority of industry interested in the technology being standardized. It is better to have a standard with points of known legal variance than no standard at all. While all of these techniques weaken the standard's specification, there are not many of these weak points, and for the most part they are along the "edges" of the specification where they should not badly damage the portability model.

NIST, FIPS PUB 151-2, AND CERTIFICATION

The National Institute of Standards and Technology (NIST, formerly the National Bureau of Standards) is the U.S. federal organization responsible for developing U.S. government procurement standards. These procurement specifications are called FIPS (Federal Information Processing Standards), and FIPS Pub 151-2 describes the use of POSIX.1 for U.S. federal agency computer procurement.

It is a slim few pages, most of it *pro forma* boiler plate, that point to POSIX.1 for developing applications, and describe which options and limits need to be provided by the implementation. A similar document for POSIX.2 (FIPS 189) was published in October 1994.

NIST builds certification programs for its FIPS, and one exists for the POSIX.1 FIPS. Essentially, a vendor submits its POSIX.1 implementation to be tested by an accredited testing laboratory, which uses the offi-

It is better to have a standard with points of known legal variance than no standard at all.

cial NIST POSIX.1 Conformance Test Suite. The results of the test suite and the POSIX Conformance Document are submitted to NIST.

If all is correct, NIST issues a FIPS 151-2 certificate for that platform. Waivers can be issued with the certificate, if the implementation failed certain test cases due to errors in the test suite. The certificate states that the implementation passed a test suite. It cannot be a guarantee of conformance, due to the complexity of testing an operating system. The FIPS certificate does act as a reasonable seal of approval.

FIPS Pub 151-2 specifies that all implementation options in POSIX.1 must be implemented, and sets certain reasonable limits, with the intent of describing the most functional implementation of POSIX.1 possible. When NIST updated the POSIX.1 FIPS from 151-1 to 151-2, it improved the testing process. While the U.S. government provided this certification process for POSIX.1 for its own benefit, there is nothing to prevent any other consumer from insisting on the existence of the FIPS 151 certificate. If a vendor does not have one, the consumer can check the register of certified platforms by email. (See email instructions in the references section.)

X/Open Branding

As a reasonable alternative to FIPS 151-2 certification, a vendor might demonstrate that is has been XPG3 or XPG4 branded, according to X/Open's certification process. However, XXPG brands are not the same as FIPS certificates with respect to POSIX.1 and POSIX.2. The X/Open branding process for XPG4 did not start as rigorously as XPG3(!) so consumers should ask to see the POSIX conformance documents for POSIX.1 or POSIX.2 implementations, depending on their requirements.

Education and Process

As obvious as it sounds, application architects and developers need to understand how to use the implementation and the specification to best develop applications. Developers who are very knowledgeable about one architecture may have no idea how to design and write code in a more portable fashion. Developers need to understand a number of important issues:

—*The overall objectives.* Without the knowledge of an organization's long-term goals with respect to application development and application portability, developers may not have the information they need to judge when and how to design a piece of code *less* portably. For all its benefits, portability is not without its costs.

—*The portability model.* Developers porting code to a new architecture, or developing code for two or more architectures simultaneously, need (i) copies of the relevant standards and specifications that form the basis for the portability model, and (ii) the relevant POSIX conformance documents for the architectures in question. These documents are not tutorial learning guides. They are the ultimate authority on how something should behave.

—*How to use the portability model.* Developers should take advantage of tutorials and guides in designing and developing portable code. The model should be researched and documented so that all current and future employees understand it. For example, because developers new to porting issues often fall into the classic #ifdef trap in the C language, the model should describe when and where it is acceptable to use #ifdef.

—*How the model is implemented* on the architectures upon which the application will be distributed. Some vendors have provided their POSIX implementations as a separate environment, with tools to move data between the native- and standards-based environments. Others have integrated it as much as possible into the native operating system. The single best tool is another platform that fits the portability model. For example, developers working on applications for a DEC VMS environment using POSIX.1 and ANSI C could be provided with an inexpensive UNIX machine that supports the same standards and is networked to the VMS machine.

Application source code can be quickly built to confirm portability and to catch built-in architectural assumptions. Even if the application will never be made to work completely on the secondary architecture, due to the deliberate desire to make limited use of certain architecture-specific features of the primary environment, it will tune and improve code that is designed to be portable, and more rapidly educate the developers.

Educating developers is important, but it is equally important to educate those responsible for the selection of the platforms. An organization's portability objectives need to be understood. Conformance, testing and certification, and the need for POSIX conformance documents should also be understood. Even if application designers are not directly responsible for platform selection, they should be involved at some point to review the conformance documents.

THE SOFTWARE MANAGEMENT PROBLEM

Those who think that having a portability model and knowledgeable developers is the complete answer overlook one of the most critical aspects of developing portable applications, which is the ability to manage the application's source code and construction process on all the platforms upon which the application will run. This issue has nothing to do with the choice of POSIX or any other specification as a base for a portability model, but it can dramatically affect the ability to implement multiplatform applications. We will introduce it in simple terms.

Surprisingly, many organizations are still "discovering" source-code versioning tools. As soon as those organizations begin using versioning tools, they realize the need for organizing and managing collections of source files at different revision levels.

Software porting requires the ability to manage the identification and tracking of the source code components of an application. It also requires planning and forethought to manage the construction problem on multiple platforms. This is often overlooked; although a decision might be made to architect new applications to a new portability model, the test application is still developed and released on a single architecture.

When managing software, a few other things need to be considered: (1) how application executables are delivered on different platforms from identifable source-code modules; (2) how construction information for different platforms is managed so that one doesn't fall into the trap of moving the #ifdef's out of the source code and into the construction process; (3) how platform-specific code will be managed; and (4) how new platforms will be added to the software management process.

Conclusions

Architecting applications software to be more portable has a number of long-term benefits with respect to the ability to move applications to new platforms that are cheaper, faster, and smaller.

The IEEE POSIX family of standards provides a sound stable base for a portability model, if one understands certain basic points before approaching it. The appearance that POSIX is failing is created by hardware-vendor marketing organizations intent on selling their own solutions to a problem.

Developing portable software is still a complex task that requires long-term planning and commitment. Simply identifying a portability model, POSIX-based or otherwise, is not enough. Education is essential, as is documentation, configuration, and construction support. **sv**

References

IEEE 1990. ISO/IEC 9945-1:1990(E), IEEE Std 1003.1-1990, information technology—Portable Operating System Interface (POSIX)—part 1: System Application Program Interface (API) [C Language] Institute of Electrical and Electronics Engineers, New York.

NIST POSIX. 1994. Electronic mail from the NIST POSIX electronic mail server. Send Internet email to <posix@nist.gov>, with a message body:
path your-internet-email-address
send help
send posix/index