-- --

IEEE P1003.0 Draft 13 – September 1991

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

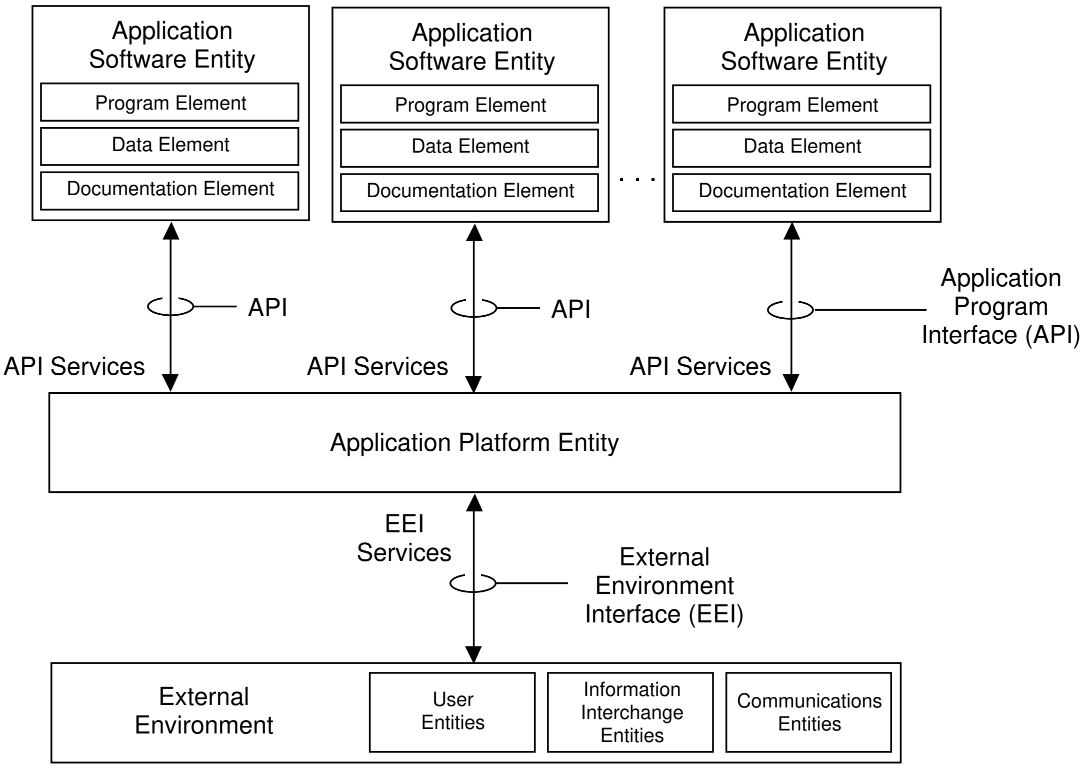

Figure 3-2 – POSIX OSE Reference Model — Entities

standards or other open specifications.

An application program may be divided into two parts:

— An *invariant* portion of source code, requiring no change when ported, and

— A *variant* portion of source code, which requires changes when ported.

The objective of any effective application software portability method should be to minimize the ‘‘variant’’ portion of the application software via creation and use of API standards. This would ideally allow application software components to be moved to a different (but portability-standard compliant) system and run without source code modification. However, since standards exist for which strictly conforming application software requires modification (e.g., memory requirements, processor-specific COBOL statements), this can only be approximated.

Separate but related standards may be required to support the portability of each of the elements listed above. Examples of application software are the familiar word-processing, spreadsheet, or accounting packages, as developed by the consumer or a commercial application software developer. Each of these packages appears as an application software entity when executed on an application platform.

One or more applications may run on a given application platform simultaneously, as represented by the boxes at the top of Figure 3-2. Each application can be thought of as an independent application entity, communicating and synchronizing with other applications, if necessary, via a variety of communications mechanisms.

The Application Platform is defined (see 2.2.2.2) as the set of resources that support the services on which an application or application software will run. It provides services at its interfaces that, as much as possible, make the implementation-specific characteristics of the platform transparent to the application software.

In order to assure system integrity and consistency, application software entities competing for application platform resources must access all resources via service requests across the API. Examples of application platform elements could include an operating system kernel, a realtime monitor program, and all hardware and peripheral drivers.

The application platform concept does not imply or constrain any specific implementation beyond the basic requirement to supply services at the interfaces. For example, the platform might be a single processor shared by a group of applications, or it might be a large distributed system with each application dedicated to a single processor. (See 3.2.4.)

The application platform for systems built to the POSIX OSE will differ greatly depending upon the requirements of the system and its intended use. It is expected that application platforms defined to be consistent with the POSIX OSE will not necessarily provide all the features discussed here, but will use tailored subsets for a particular set of application software.

The External Environment contains the external entities with which the application platform exchanges information. These entities are classified into the general categories of human users, information interchange entities, and communications entities.

Human users are not further classified, but are treated as an abstract, or average, person. Information interchange entities include removable disk packs, floppy disks, and security badges. Communications entities include phone lines, local area networks, and packet switching equipment

3.2.2 Reference Model Interfaces

Figure 3-3 expands Figure 3-1 to identify the services available at the reference model interfaces.

Between these three classes of entities there are two types of interface where standards and other open system specifications are required to enable application software portability and interoperability. These two interface types are labeled as the Application Program Interface (API) and the External Environment Interface (EEI).

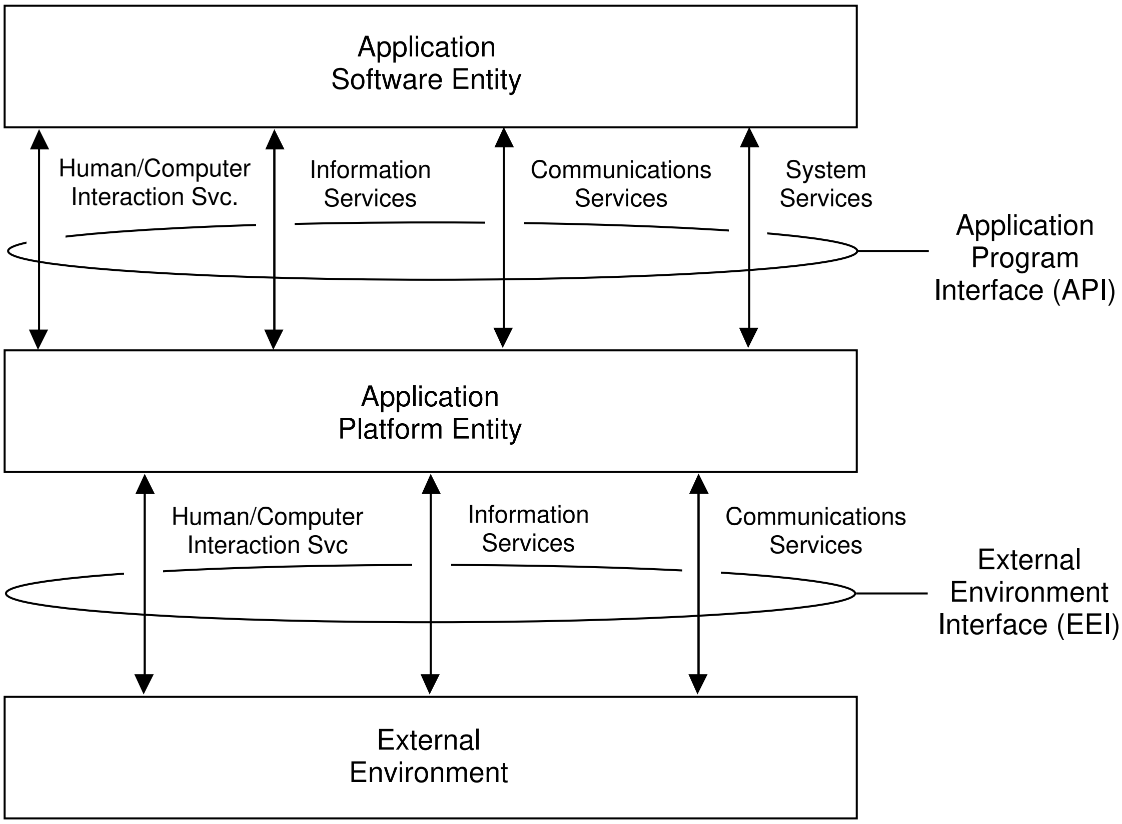

Figure 3-3 – POSIX OSE Reference Model — Interfaces

3.2.2.1 External Environment Interface (EEI)

The External Environment Interface is defined (see 2.2.2.9) as the interface between the application platform and the external environment across which information is exchanged. It is defined primarily in support of system and application software interoperability. User and data portability are directly provided by the EEI, but application software portability also is indirectly supported by reference to common concepts linking specifications at both interfaces. The services available at the EEI comprise:

— Human/Computer Interaction Services

— Information Services

— Communications Services

The Human/Computer Interaction EEI is the boundary across which physical interaction between the human being and the application platform takes place. Examples of this type of interface include CRT displays, keyboards, mice, and audio input/output devices. Standardization at this interface will allow users to access the services of compliant systems without costly retraining.

The Information Services EEI defines a boundary across which external, persistent storage service is provided, where only the format and syntax is required to be specified for data portability and interoperability.

The Communications Services EEI provides access to services for interaction between internal application software entities and application platform external entities, such as application software entities on other application platforms, external data transport facilities, and devices. The services provided are those where protocol state, syntax, and format all must be standardized for application interoperability.

3.2.2.2 Application Program Interface (API)

The Application Program Interface (API) is defined (see 2.2.2.3) as the interface between the application software and the application platform across which all services are provided. It is defined primarily in support of application portability, but system and application software interoperability also are supported via the communications services API.

The POSIX OSE API is a combination of a number of standards-based interfaces. It can be thought of as a bookshelf containing several standards-based APIs, with each API a separate book on the bookshelf.

The POSIX OSE API specifies a complete interface between the application software and the underlying application platform, and may be divided into the following parts:

- Human/Computer Interaction Services API
- Information Services API
- Communications Services API
- System Services API

The first three APIs listed are required to provide the application software with access to services associated with each of the external environment entities.

The fourth API is required to provide access to services associated with the application platform internal resources, identified as the System Services API. This interface may be divided into two types of specifications; i.e., Language Service and System Services API specifications.

Definitions of services at the API take the form of programming-language specifications, language-independent service specifications, and language bindings for the service specifications. These specifications may be described as follows:

(1) Those traditionally associated with the language specifications, such as program control (if . . . then . . . else), math functions, string manipulation, etc., defined as *the programming language API*, and

(2) Services provided by the underlying application platform defined independent of language, such as interprocess communications,

interobject messages, access to the user interface, and data storage. Specifications of for these services are defined independently of any programming language, and are identified as *language-independent service specifications*.

(3) The language-independent service specifications are translated into language-specific specifications used by programmers in writing applications. These specifications provide access to the services using methods consistent with a specific programming language. Such language-specific specifications are called *language-binding APIs*.

Creation of a *language-independent service specification* facilitates the management and development of consistent language binding standards. The language-binding specifications are used directly by programmers and application platform suppliers in implementing application software and platforms.

The ''programming language''/''language binding'' dichotomy may be a result of the way Information Technology standards are currently developed. Programming language specifications are developed with the goal of being ''system independent'' (e.g., C, COBOL, FORTRAN, etc.). Language Binding specifications (e.g., POSIX.1 {2}, MOSI, etc.) are being translated into ''language-independent'' specifications, with one or more bindings for specific languages.

3.2.3 EEI-API Service Relationships

The relationships between similarly named services provided at the API and the EEI are not simple one-to-one relationships. For example, a data storage service interface may provide an application with transparent access to a remote file via network services. In this case, the completion of the data storage service provided at the API is dependent upon, and can be thought of as having been ''translated'' into, communication services provided at the EEI.

Fortunately, it is not essential for the purpose of satisfying the requirements of the POSIX OSE to specify these relationships in detail. In fact, a detailed definition could unnecessarily constrain the implementation. A given implementation of the application platform will define the relationship between the API and EEI in different ways.

3.2.4 POSIX OSE-Based Distributed Systems

In a distributed environment, multiple application platforms may interact by way of a network external to the platforms, but connected to them via the communications EEI, as in Figure 3-4. For an application software entity to gain access to the EEI services, communications services are requested at the API. The implementation of the application platform translates these API requests into appropriate action at the EEI.

Communication occurs between application platforms via external entities that implement the data transport function. These can use a wide variety of implementation methods and protocols, providing access to distributed data and

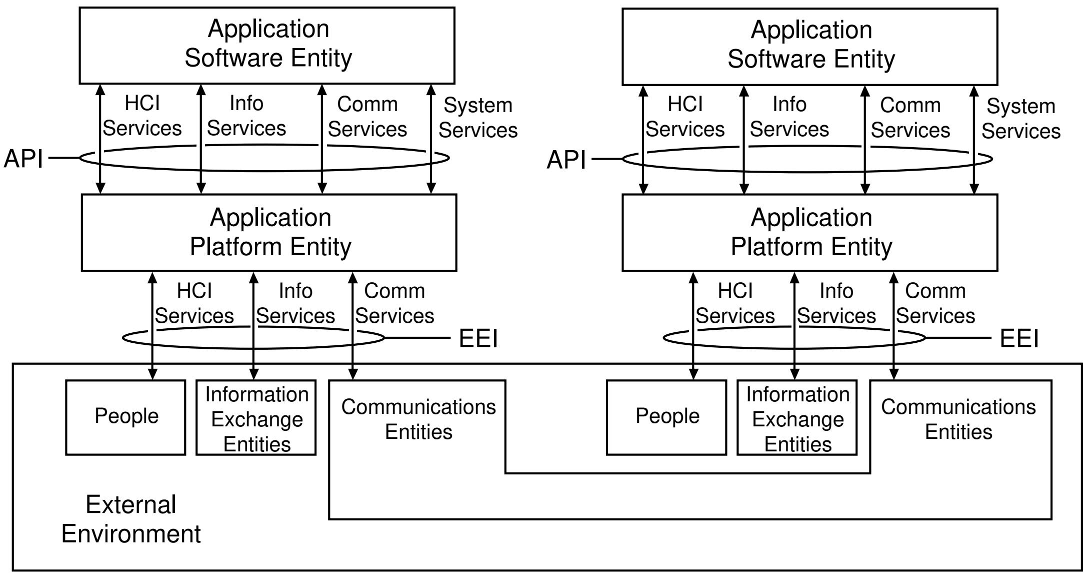

Figure 3-4 – POSIX OSE Reference Model — Distributed Systems

services via the network.

Distributed Systems are manifest in this model primarily through the use of the distributed system network services API. As can be seen in Figure 3-5, distributed systems are a refinement of the POSIX Network Environment Model shown in Figure 4-4. As such, a perceived Application Platform may in fact be comprised of several (or many) individual application platforms. However, in the distributed environment, they operate and are viewed as a single entity by the using applications. Within this extended application platform are the embedded network services necessary for the elements of a distributed environment to function.

Within the distributed environment, network access between the platforms that make up the ‘‘perceived’’ application platform are handled using the Distributed Systems Network Services APIs. Network services for access between ‘‘perceived’’ application platforms will use the Network Services EEI between the platforms.

3.3 POSIX Open System Environment Services

This guide defines a uniform set of standard services provided to users of application platforms in support of POSIX objectives of application portability and system interoperability. These services are available to users across specified interfaces keyed to the POSIX reference model defined in 3.2.

The POSIX OSE services are divided into categories described by the clauses in Section 4. Each category begins by defining a more detailed and specialized version of the OSE reference model (see 3.2) to provide context for service

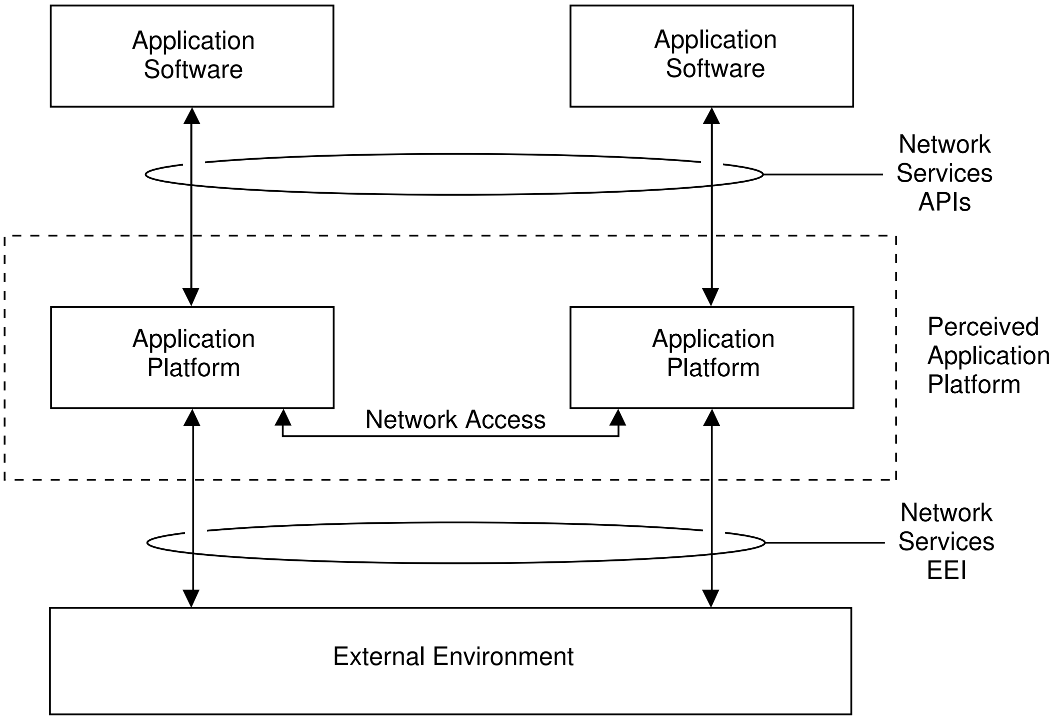

Figure 3-5 – Distributed System Environment Model

specification. Services and associated standards are then defined for each category. Finally, POSIX OSE Cross-Category Services affecting each category are discussed.

The service descriptions for each category are intended to be complete and not merely representative. Further refinement through successive releases of this document will lead to a complete specification.

3.4 POSIX Open System Environment Standards

The identification of a complete, consistent suite of standards for the POSIX OSE will, by necessity, draw from many forums. One of the criteria for judging completeness is the satisfaction of the full range of services required by the application platform user. The factors used to select standards will be described followed by the selection precedence.

Note that while the services are stated with a clear partitioning in mind, the standards reflect the current partitioning. These standards were created within disparate organizations and projects, which were in many cases carried out in isolation from the others. As a result, mapping of services to standards is not a simple relationship.

3.4.1 Factors in Standards Selection

The selection criteria for standards to be included in the POSIX OSE are based upon four concepts. Those concepts are Those concepts are openness, Stage of Completion, stability, Geographic Scope of Consensus, Functional Scope Addressed within this guide, Consistency with POSIX.1 {2}, and Availability for Unencumbered Implementation.

(1) Openness

Standards development organizations can differ from one another by virtue of their ‘‘openness.’’ That is, some standards development bodies utilize an open forum for the development of standards while other bodies use a closed forum. The result is a varying degree of consensus in the technical content of the standards across development bodies.

As a general rule, standards developed by accredited standards development organizations (all of which use an open forum) are preferred over those standards developed by bodies using a closed forum.

(2) Stage of Completion

Another factor involved in the selection of standards for inclusion in the POSIX OSE is ‘‘stage of completion.’’ That is, there is a standards development life cycle process whose effects need to be taken into account. Most standards follow a sequence from approved development, through draft, and on to approved standard.

As a general rule, where choices were made among standards, the more complete standards were favored.

(3) Stability

A third factor in determining which standards are included in the POSIX OSE is stability. This factor refers to anticipated change in the standard over time. This change may expand or contract the technical coverage of the standard.

As a general rule the more stable standards are preferred over those subject to change.

(4) Geographic Scope of Consensus

There are differences among standards development bodies with respect to the scope of their geographic consensus. Some among those bodies are formal standards bodies (i.e., accredited as standards developers by a recognized body). It is typical for those bodies to be authorized to develop standards for a particular technical topic and have their standards applicable to some defined geographic area. Formal standards development bodies are typically empowered to develop standards for either international, regional or national standards coverage.

The general rule applied in the selection of standards for inclusion in the POSIX Open System Environment is to select standards developed by those bodies that have the greatest scope of coverage. This results in a precedence for standards selection of international, followed by regional, followed by national body developed standards.

(5) Functional Scope Addressed within this guide

A specification is listed only if it addresses some service requirement listed in this guide. Standards and/or specifications listed are not, however, limited to one per set of services.

(6) Consistency with POSIX.1 {2}

Standards listed in this guide are suitable for inclusion in a profile with POSIX.1 {2}, and do not contradict that standard in any way.

(7) Availability for Unencumbered Implementation

A standard or specification is listed only if it is available for implementation to the specification and distribution of that implementation is unencumbered. The specification qualifies for inclusion in the guide even if the document itself is a salable item.

3.4.2 Selection Precedence

The list below shows the precedence of standards and specifications as used for inclusion in the POSIX OSE. The order from top to bottom is from most to least preferred.

(1) Approved standards developed by accredited international bodies

(2) Approved standards developed by accredited regional bodies

(3) Approved standards developed by accredited national bodies

(4) Draft standards developed by accredited international bodies

(5) Draft standards developed by accredited regional bodies

(6) Draft standards developed by accredited national bodies.

(7) Recognized de facto standards and specifications developed by nonaccredited bodies using an open forum

(8) Approved standards and specifications developed by nonaccredited international standards bodies using a closed forum

(9) Approved standards and specifications developed by nonaccredited national standards bodies using a closed forum.

Standards projects for which there is no draft or approved standard are never selected for inclusion in the POSIX OSE.

''Equivalent'' or ''derived'' standards lower in the precedence hierarchy are correlated in an annex. *FES : Do we still want to do this?* Only the highest precedence specification is listed or discussed in the main text.

This guide only cites government and de facto standards and specifications in discussion of gaps in available standards.

3.5 POSIX Open System Environment Profiles

The results of Open System specification projects are collected into an expanding set of ‘‘Base Standards,’’ addressing a growing subset of functional requirements.

Profile projects then select among these base standards to create a tailored, consistent set of standards addressing a more specific type (or instance) of system or set of application software. Profiles satisfy the requirements of application ‘‘domains’’ such as office or industrial automation, transaction processing, or real-time control systems.

This framework provides a way to characterize the functionality of profile activities. The current OSI profiles tend to focus strictly on the communications EEI. Other profiles might focus on a single component or span multiple interface types.

3.6 Application Platform Implementation Considerations

HLJ : All instances of ‘‘System Integration Interface’’ in this clause have been changed to ‘‘System Internal Interface’’ without further diff marks.

Profile writers need to be aware that in an open system environment, the application platform can be decomposed into independently procurable components. While standards are interface specifications, and as such are independent of implementation, there are aspects of platform implementation or construction that may affect the specification of standards, and that profile writers may want to address.

For each case, the portion of the application platform that implements any particular independently procurable service is described as the service component. Figure 3-6 shows an application platform made up of several service components. If components interact, the specification of the interface between service components within the application platform may be standardized or nonstandard (including proprietary).

An intercomponent interface is labeled in Figure 3-6 as ‘‘System Internal Interface’’ because it may be used to assemble an application platform from multiple components. Figure 3-6 shows how a System Internal Interface is shown in the reference model.

A standards-based SII between the application platform service components addresses portability and interoperability of the application platform service components, not portability and interoperability of application software and systems.

Development of an SII would also require a consensus to emerge on the ‘‘best’’ design and implementation of system software/hardware. Very little consensus has developed on the partitioning of the platform into components and consequent

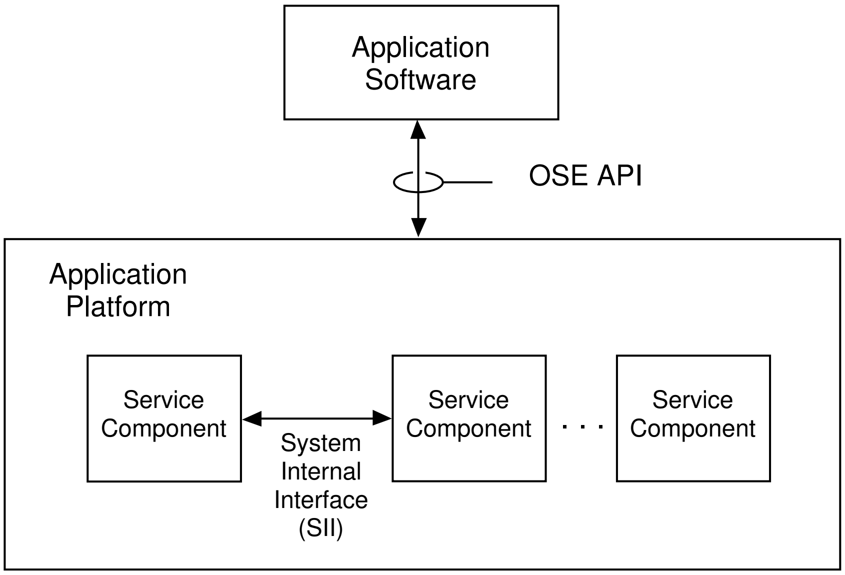

Figure 3-6 – Service Components and Interfaces

allocation of function to each. In fact, this aspect of system design has been in a constant and accelerating state of innovation for decades. One of the major objectives of the API is to provide a more stable interface that decouples application software from the constantly changing platform. This enables the migration of application software to platforms based on constantly upgraded technology. (See 3.1 ‘‘Accommodation of New Information System Technology’’.)

The relationship and services exchanged among the components may be quite complex and varied in different implementations. This complexity and variety would, of necessity, be reflected in an SII. It would not, however, be visible to the application software at the API, since one of the major objectives of the API is to hide this complexity. (See 3.1 ‘‘Implementation Transparency’’.)

Since SII specifications

— do not affect application portability and interoperability, and

— do not affect specification of the API and EEI, and

— are primarily driven by specific implementations of the application platform,

SII specification is beyond the scope of this guide. D

Specification of SII in this guide would represent an unnecessary constraint on the implementation of the application platform, and are unnecessary for the specification of the API and EEI.

There are a number of ways which the Application Platform can be divided into separate service components. The main decomposition methods are division, layering, and redirection. These methods are indistinguishable to the application software and external entities, in that they all interface to the application platform via the API and EEI, respectively. They assume a starting base application platform, which provides a subset of the required services.

3.6.1 Subdivision

In this commonly used method, the application platform is simply subdivided into a base and one or more service components. See Figure 3-7.

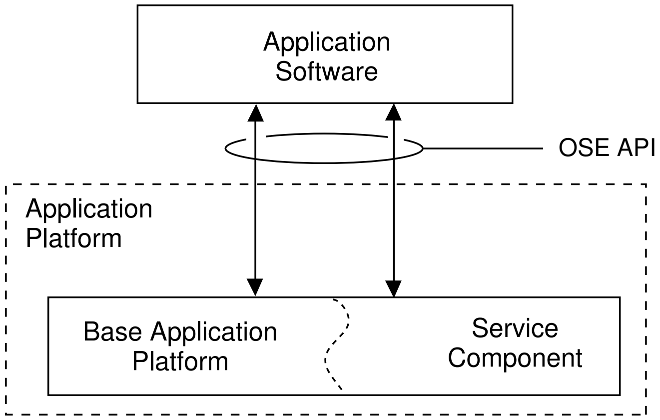

Figure 3-7 – Application Platform Implementation — Subdivision

One possible implementation of this is to link the appropriate service modules directly into the system kernel.

The internal interfaces used in this method are normally proprietary, and hence normally imply that both components will come from the same vendor.

In this case the Application Platform and the Application Platform Base are the same entity.

3.6.2 Layering

In layering, the service is interposed as a layer between the application software and the base application platform. See Figure 3-8.

This is the most common method of supplying a service component that is independent of the base. One possible implementation is to provide the service component as a set of library routines.

Whether the interface between the service layer and the base application platform conforms to any standards affects the portability of the service component. Note that specifying a standard API for this interface guarantees only that this

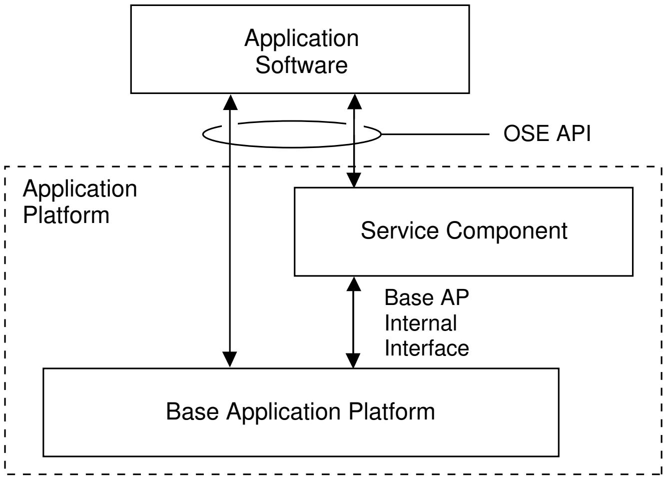

Figure 3-8 – Application Platform Decomposition II — Layering

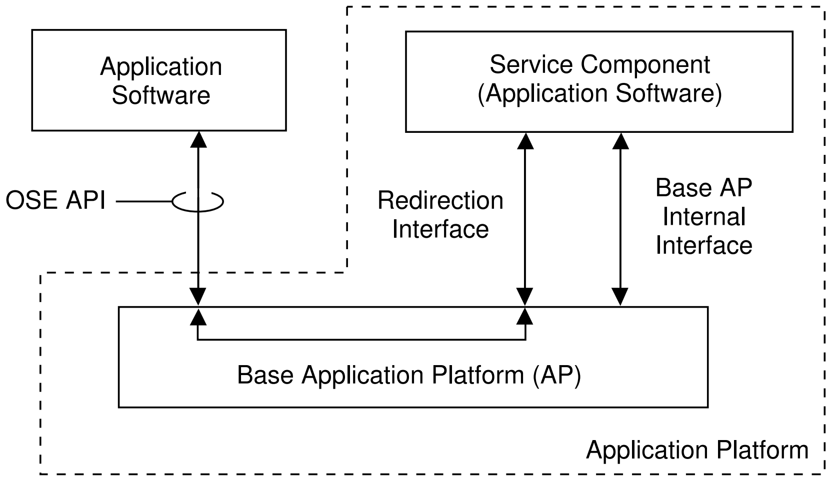

Figure 3-9 – Application Platform Decomposition III — Redirection

component will be portable at the source level.

3.6.3 Redirection

Redirection allows a service component to ask the base application platform to redirect all requests for that type of service to the service component. See Figure 3-9. Possible examples of such services are device drivers, network protocol handlers, and database engines.

In actual implementation, the service component may or may not be a separate process. Possible implementations are: dynamically loadable kernel modules, library routines layered over IPC, and lightweight kernel processes.

Note that there are three interfaces. The application software normally sees a complete, standard API to the base. The service component has two interfaces— one to effect the redirection, and one to provide base services to the service application software entity. Considerations for portability discussed under Layering also apply here.

Note also that no POSIX standardization activity currently exists for the redirection interface.

Section 4: POSIX Open System Environment Services

Responsibility: Fritz Schulz

This section describes the services required in support of the objectives identified in this guide. The services are grouped in major categories defined in Section 3, with more detailed breakdowns within each category as appropriate. These categories are:

- Fundamental System Services
  - 4.1 Language Services
  - 4.2 System Services
- Communications Services
  - 4.3 Network Services
- Information Services
  - 4.4 Database Services
  - 4.5 Data Interchange Services
- Human-Computer Interaction Services
  - 4.6 Windowing System Services D
  - 4.7 Graphic Services
  - 4.8 Character-Based User Interface Services
  - 4.9 User Command Interface Services
- Domain Services
  - 4.10 Transaction Processing Services D
  - 4.11 Software Development Environment Services

Criteria used to partition services are outlined in 3.2, and discussed at the beginning of each clause. The discussion for each of the service category subclauses follows the same outline, and is as follows:

4.*n*.1
4.*n*.2 Overview and Rationale, Scope

This text introduces the scope of this service category, and the criteria used to identify the services within it.

4.*n*.3 Reference Model

This subclause builds on the model of clause 3.2 and gives additional detail related to the interfaces and services discussed there. An optional subclause may discuss implementation considerations, similar to the discussion of 3.6.

4.*n*.4 Service Requirements

This text provides the definition of service requirements within the scope described in 4.*n*.2.

4.*n*.5 Standards, Specifications, and Gaps

A table lists the standards and specifications available to meet the service requirements listed in 4.*n*.4. This is followed by a brief discussion of services for which standards are not available.

4.*n*.5.1 Current Standards

The following subclauses cite existing specifications that have been approved as standards by accredited standards bodies, in the order of precedence identified in 3.4.2. When service requirements are satisfied at a higher precedence level, specifications at a lower level are not listed.

4.*n*.5.2 Emerging Standards

The following subclauses list specifications and/or activities that address the functional areas within the 4.*n* section, but which have not yet been completed. Where a group or activity is cited, the charter of the group may address the functionality, but it is possible that a draft may not be available. Only those services not currently addressed by existing standards are to be discussed in this subclause. It is expected that documents will migrate from 4.*n*.5.2 to 4.*n*.5.1 as they complete the consensus process.

4.*n*.5.3 Gaps in Available Standards

This subclause identifies those service requirements that have not been satisfied by existing or emerging standards. If all service requirements in this category have been met by existing or emerging standards, this subclause will be empty. Text in this subclause will be minimal.

4.*n*.5.3.1 Public Specifications

This subclause lists any specification outside of the formal standards community that is available to anyone (e.g., no membership required) for implementation and distribution (including sale) without restriction, including all government and de facto standards.

4.*n*.5.3.2 Unsatisfied Service Requirements

This subclause lists the services for which no specification has been cited in this guide. Products may be cited here to illustrate capabilities that are not addressed by standards.

4.*n*.6 POSIX OSE Cross-Category Services

This subclause contains any discussion of the Cross-Category Services in Section 5 that is specific to subclause 4.*n*.

4.*n*.7 Related Standards

This subclause is optional and may identify interdependencies among standards that should be taken into account when selecting among them.

4.*n*.8 Open Issues

This subclause is optional and may identify issues under discussion in the open systems community.

4.1 Language Services

Responsibility: Don Folland

4.1.1 Overview and Rationale

While a consistent interface to the operating system is essential for applications portability, the application will have been developed using language and system development tools that, in turn, require to be supported by standards to achieve source code portability.

Those responsible for system or software development will wish to write programs in code supported by an international standard and compile the code using a compiler that has a certificate of conformance issued by an accredited test center. Noncompliant extensions must be avoided if applications portability is to be maintained. Compilers should identify nonstandard-compliant code.

The languages that have been identified in this document are those seen to be in most popular use today for software development. The POSIX.2 shell command language is discussed in 4.9. The standards identified are the most widely recognized today, with significant use in the Information Technology industry on a broad range of processors, or where a large installed base of a particular version is known to exist.

4.1.2 Scope

The services described in this clause cover the most widely used third-generation computer languages in use today for the development of applications; i.e., the languages used to write application programs. Fourth-generation languages are not currently addressed in this guide. In order for a program to address an API to the services described in other clauses of this guide, an appropriate language binding to that interface is required. References to those bindings will be found in the clause describing the relevant service.

4.1.3 Reference Model

This subclause identifies the entities and interfaces supporting language services. The reference model based on the reference model in Figure 3-1 is illustrated in Figure 4-1, but because the language services directly support the binding of the applications to the API, there is no EEI. However, the EEI is shown in Figure 4-1 for consistency.

At the simplistic level, the programmer developing an application that requires only basic operating system services will use a compiler that meets both the fundamental language standard (e.g., ISO 1989: 1985 for COBOL, ISO 1359: 1990 for Fortran) and the binding established for the relevant system calls in POSIX.1 {2}.

As identified in 4.10, an application program may also require database services that will be provided by the Database Manager API. The database vendor will

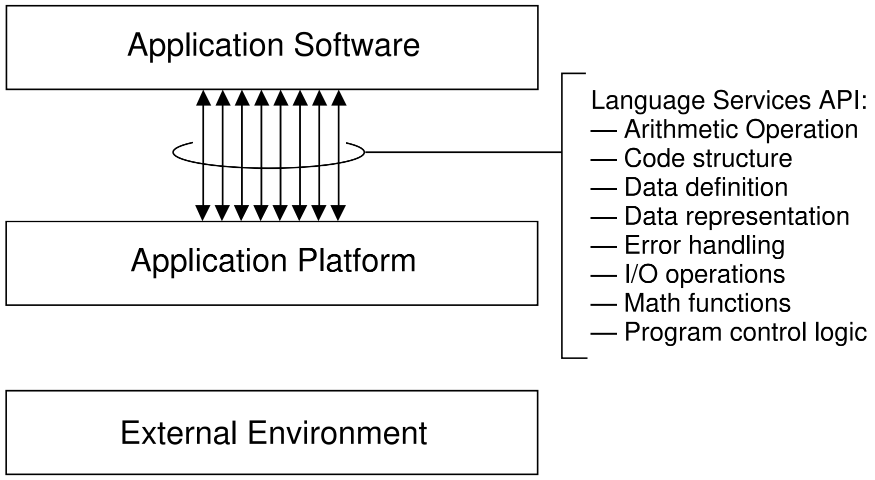

Figure 4-1 – Language Service Reference Model

offer an API to meet the requirements for the popular programming languages.

In a POSIX Open System Environment the intention is that support is provided for all languages identified in 4.1.4.

4.1.4 Service Requirements

Programming language services provide the basic syntax and semantic definition for use by a software developer to describe the desired application software function. While most clauses in this guide provide a comprehensive list of services, in the case of languages many services are a unique function of the language specification. Rather than extend the size of this guide, the detail is more appropriately found in the relevant language manuals and supporting standards.

4.1.4.1 Application Program Services

Programmers require the ability to compile a program in the language of their D
choice. The selection of a particular programming language for the development of an application may depend on a variety of factors, including the capability to provide some of the functions listed here:

— Arithmetic operation

— Code structure

— Data definition

— Data representation

— Error handling