Porting to ANDF

Stavros Macrakis
Open Software Foundation

Version of January 22, 1993

From strictly portable applications, ANDF produces strictly portable distributions. Most applications are not strictly portable, and must be adapted to ANDF before distribution. ANDF provides abstraction mechanisms which support portability.

1. Introduction

OSF's Architecture-Neutral Distribution Format (ANDF) allows distributing a single binary application to multiple platforms. To do this successfully, the application must be portable.

A strictly portable application does not depend on any platform-specific behavior, but only on standard language semantics and standard programming interfaces. It can be compiled and run with no modifications on any conforming platform. It can also be compiled by an ANDF producer and installed and run on any conforming platform.

Very few applications are strictly portable: Only recently have the necessary standards been published and widely implemented. Application writers exploit platform-specific features. Verifying portability is not always easy. And sometimes, portability has not been a goal during development.

Still, many applications are easily portable. This paper gives some advice on how to move them onto the ANDF technology. Some applications are portable by being parameterized by platform, that is, they use the preprocessor to compile a somewhat different program for each platform. We show how to handle this situation in ANDF. Some applications expect extensions to standard interfaces. We show how to handle this within ANDF.

With these techniques, most applications can be ported once to ANDF and thereafter be installed on a wide range of conforming platforms with no additional porting effort.

2. Degrees of Application Portability

Before open systems, applications could only be made to run on many platforms by creating a different variant for each platform, that is, *porting* it. Applications which can be or have been ported to many systems are called *portable*.

With open systems, applications can run on many platforms without variants if they rely only on industry-standard languages and application programming interfaces (APIs). We call such applications *strictly portable*. For instance, an application written in portable ANSI C, using only XPG/3 operating system calls and Motif user interface calls, is strictly portable to all open system platforms which support these standards—as most now do.[1]

— *Strictly portable source yields strictly portable ANDF*

The source of a strictly portable application can be compiled and run without modification on any compliant platform. It can also be compiled by an ANDF producer on one platform into the ANDF form, and installed by an ANDF installer on any other compliant platform. That is, if the source is strictly portable, the ANDF form is also strictly portable.

1. Both application and platform must strictly respect the semantics and the syntax of the interfaces. Also, applications must rely on standard header files, and not attempt to declare interfaces themselves.

So, any strictly portable application can be distributed using ANDF by simply substituting the ANDF producer and installer for the various native compilers.

A good example of a strictly portable application is the ANDF C producer itself (tcc). We have successfully installed it on all platforms with ANDF installers with no modifications.

— *New applications can easily be strictly portable*

Applications written today can easily be strictly portable. By using standard and well-documented languages and interfaces, and following widely-understood portability guidelines, they will be strictly portable, and thus installable as source or as ANDF on all conforming open system platforms.

With a disciplined use of ANDF features, portable applications can also take advantage of platform-specific features.

— *Many legacy applications are portable, but not strictly portable*

Unfortunately, few *existing* applications are strictly portable, for many reasons.

Many existing applications were written before open system standards were published. After all, ANSI C was only standardized in 1989. Motif's specification was published in 1989. And in both cases, there was some lag between publication of the specification and full implementation by a wide range of vendors. Existing applications often rely on versions of Unix which do not completely conform to current industry standards.

Some programs have a continuing requirement to be backwards compatible with pre-ANSI compilers or non-standard APIs, because an important part of their market does not yet support open systems standards.

Programs often exploit platform-specific features for additional functionality or performance. Sometimes, necessary functionality is not covered by the standards, and must be provided in a different way on each

platform. This was especially true in earlier versions of open systems API standards.

Many legacy applications are, however, portable with a modicum of effort to ANDF, and once ported to ANDF, are strictly portable to all platforms supporting ANDF.

An example of such a portable commercial legacy application is Wingz, which was ported to ANDF with little effort, and now runs on all ANDF platforms.

— *Many applications are parameterized by platform*

Many applications are portable by being parameterized, that is, they use makefiles and preprocessor conditionals and macros to compile a somewhat different program for each platform. This is often true of applications which started out on one platform, and were ported successively to others. Parameterization is also useful when applications want to take advantage of platform-specific features while remaining portable.

Parameterized portability is handled in the ANDF framework by conditional compilation and the token mechanism, which are described in greater detail below.

An example of a heavily-parameterized portable application is Gnu Emacs. Gnu Emacs uses a variety of tricks to ensure efficient application on a wide range of platforms. Also, it adds functionality whose implementation is highly platform-dependent (notably unexec). Gnu Emacs has been successfully ported to ANDF.

— *Some applications are not very portable*

Some applications are hard to port.

This may be because the developers were unaware of portability techniques, or because they considered portability unnecessary. Some developers may have considered “tuning” for a particular platform more important than

portability. Of course, applications that depend critically on non-standard interfaces cannot be portable.

The marks of a hard-to-port application are extensive reliance on proprietary features or enhancements to standard interface, or on the details of its implementation (“piercing the veil of abstraction”).

Strict portability

The ANDF model of strict portability is very simple. The program should be written in ANSI C or another language supported by ANDF, and use only standard interfaces. In additional, all external procedures must be declared before use.[2]

ANDF can support any standard interface, since interface definitions are written in a special sublanguage—the token mechanism—available to all ANDF users; they are not built into the technology. Currently, the following interfaces are supplied with the technology: POSIX, XPG/3, System V.4, and the OSF Application Environment Specification (currently incomplete). Other interfaces, such as Windows NT, Macintosh OS, or VMS could equally well be defined, with no modifications to the producer or installer.

When compiling a program using ANDF, ANDF header files are used rather than normal header files. Using the token mechanism described in this paper, ANDF header files declare only the visible part of the interface while deferring its implementation. This not only produces an ANDF form which can be installed on any platform, but also provides a high level of API interface checking. Applications could also use the token mechanism to provide more abstract interfaces—and therefore better type checking—between program modules.

2. Implicit declarations, although tolerated by ANSI C, are known to be a common source of error. See Harbison and Steele or any of the portability handbooks.

Partial portability

There are two main ways in which a program can be partially- or non-portable: it can be written using non-portable language constructs; or it can use non-standard programming interfaces (or use standard ones incorrectly).

— *Language problems*

Most programming languages have dialects and implementation dependencies. Dialects are different versions of the language, with non-standard extensions or restrictions. Implementation dependencies are features of a language which are not guaranteed to have the same meaning in different implementations.

In the C world, two important dialects are the original C language as presented by Kernighan and Ritchie, (K&R C), and ANSI Standard C. Since there is in fact considerable variation in pre-standard C compilers, which usually go somewhat beyond K&R C, ANSI C is more portable. ANSI C also provides additional mechanisms (notably function prototypes) which permit stricter static portability and interface checking. (ANDF C can give the K&R or the ANSI interpretation of a program.)

Still, ANSI C has not resolved all the implementation dependencies of C. For instance, C integer declarations remain imprecise. C allows declarations of integers as “short”, “int”, or “long”. Shorts must be at least 16 bits, longs must be at least 32 bits, and ints must be no smaller than shorts and no larger than longs. Thus portable programs cannot assume anything more than this. The ANDF technology contains additional mechanisms for specifying ranges of integers more precisely, but C programs cannot take advantage of them without adding ANDF-dependent constructs (pragmas).

Worse, many compilers are permissive, and although they may implement conforming C programs correctly, they also accept nonconforming C and, rather than give errors, generate interpretations that programmers find useful (this is not an accident; it typically comes from the programmer being aware of “typical” compiler behavior). This means that even code that compiles and runs correctly on many compilers may contain non-portable constructs.[3]

3. See “Porting Postgres...”, Watt, 1992.

— Programming interface problems

Standards documents define standard application programming interfaces by specifying the abstract interface, that is, the characteristics that an application programmer can depend on.

Using an interface means including a header file of definitions, which are required to be fully specified. Header files typically include data type definitions, function definitions, macro definitions, and variable declarations. Although the function definitions are abstract (that is, they contain no more information than that needed to use them), type and macro definitions are not.

Applications programmers thus can take advantage of the internal structure of these interfaces, intentionally or unwittingly. For instance, although the internal structure of a `stdio` buffer is not part of `stdio`'s interface, it is typically present in the header file, since it is used by macros such as `putchar`.

Checking for these dependencies is difficult because many implementations' concrete definitions are similar or identical, so even porting to multiple platforms will not detect non-standard use of APIs.

The ANDF technology introduces a mechanism for defining only the abstract interface, and then later checking for the conformance of the concrete to the abstract interface. This is part of the token mechanism (*q.v.*).

Further reading on portability in C

There are several books available on writing portable code in C. They present both general principles and particular traps.

The general principles are simple: never say the same thing in two different places; only rely on the standard properties of data types; don't use obscure features. The detail is what makes these books valuable. We have found Rabinowitz and Schaap particularly useful for C language issues, and Horton for reference material on portability within Unix variants.

Harbison and Steele has excellent discussion of machine dependencies and traps in the C language, but portability advice is less prominent than a complete description of language semantics.

3. Support for Portability in ANDF C

ANDF C supports portability in two ways: interface abstraction and conditional compilation.

Interface abstraction is also a well-known technique, but is often thought of primarily as a software engineering method rather than a portability technique. ANDF C adds a general-purpose interface abstraction mechanism to C, called tokens. Tokens' interfaces are declared when the program is compiled by the producer, but their definitions are only supplied when the program is installed.

Conditional compilation is a traditional way of parameterizing applications. ANDF C supports it fully. In particular, the conditional can depend on tokens which are only defined at install time, thus allowing install-time conditional compilation.

Tokens are abstract interfaces

Tokens are ANDF's key abstraction and therefore portability mechanism.[4] Tokens define abstract interfaces which are later instantiated by concrete interfaces. The abstract interfaces are used by the producer, and the concrete interfaces by the installer. The process of compiling token definitions guarantees that the concrete interfaces are type-correct implementations of the abstract interfaces.

For instance, consider `getc`. Abstractly, it is simply a function of a file pointer returning an integer. It is almost always implemented as a macro for efficiency, and that macro refers to the internal structure of a file structure. But that internal structure differs from implementation to implementation,

4. Some writers use "token" as a synonym for "lexeme" or "lexical element." The ANDF use of token is completely different: it is an interface specification whose implementation is deferred to installation.

and should not be visible to the user of getc. Yet although C provides function declarators, it does not provide macro declarators—that is, there is no way to separate the declaration of a `getc` macro from its implementation. An ANDF function token is precisely what is needed: a macro declarator.

— *Tokens abstract program elements*

Tokens can abstract many kinds of program element, including identifiers, types, and macros.

Tokens must be declared before use. Tokens can replace any part of a C program, not just a statement or an expression. For instance, tokens can represent a type or a structure field.

Uses of tokens can be checked for type correctness from just their declaration, without looking at the body. This is critical to their usefulness, since they typically have different bodies on different targets.

These abstract interfaces are a useful facility for modern software engineering practice, and become crucial for the definition of portable interfaces: the abstract interface is the portable interface, and the concrete interface is a particular implementation of the interface.

In the C producer, tokens are an extension to C's definition facilities. Tokens allow separating the declaration of types, macros, constants, and other entities from the actual values they take. This is necessary because types, macros, and constants are visible parts of library interfaces. Although standards such as POSIX specify these interfaces abstractly, there is no provision in the C language for declaring them abstractly, that is, without committing to a particular implementation. The `time_t` type is a good example of the problem.

— *An example: the problem with `time_t`*

POSIX defines several functions which manipulate values of type `time_t`, defined as the number of seconds since midnight, 1 Jan 1970. POSIX guarantees that it will be an arithmetic type, but does not commit to any

particular type. On the other hand, the include files which define the interface for a normal C compilation must commit to a particular type, typically `long`, `double`, or `long double`:

```
typedef long time_t;
```

Thus if an application writer unwittingly assumes that time_t is a long, the native compiler cannot catch the portability error in a piece of code like:

```
time_t clock; ...
if (clock & 1) ... /* Odd second? */
```

since `time_t` is simply another name for `long` to it.

— The token solution for `time_t`

ANDF provides a mechanism to define `time_t` as an arithmetic type without specifying which one:

```
#pragma token ARITHMETIC time_t #
```

This definition is used in the platform-independent header files read by the producer. The token `time_t` can now be used anywhere a type name could be used, for instance in the definition of a structure or the declaration of a function.

On the other hand, the platform-dependent header files contain the corresponding concrete interface, specified as usual with a `typedef`.

The kinds of tokens used for C

Tokens can abstract pure expressions (r-values), object references (l-values), integer constants, types, member selectors, certain kinds of macros, and functions.

A token can be used anywhere the corresponding type of expression could be used in standard C. For instance, a type token can be used not only to declare variables, but also to build up structures and unions, or to define function parameters.

The full syntax and semantics of tokens in the C producer are discussed in the document *The token syntax for the C producer*[5]. The current token definition syntax corresponds closely to the internal structure of tokens in ANDF, rather than to C's syntax; an alternative syntax is being studied which follows C more closely, and thus makes it easier to rewrite existing interfaces in terms of tokens.

Here, we summarize the different kinds of tokens and their uses.

— *Expressions*

A token of type `EXP` can replace a *primary-expression*. All `EXP`s have a specified storage type (lvalue or rvalue) and type. Only lvalue `EXP`s may appear on the left-hand side of an assignment.

Rvalue `EXP`s are often used to define platform-dependent constants such as `MAXFLOAT` and `stdin`. `MAXFLOAT` is of course of type double (a built-in type) but `stdin` is of type `FILE *`, where `FILE` is a type token.

Lvalue `EXP`s are often used for variables such as `errno` or `curscr` (curses.h). Again, `errno` is defined by ANSI as an `int`, but `curscr` is of type `WINDOW *`, where `WINDOW` is a type token.

— *Integer constants*

Constants used for array sizes and bitfield widths must be natural numbers (non-negative integers), and are defined as `NAT` tokens. To allow the use of unmodified platform-dependent header files, the ANDF installer is particularly clever at handling implicit declarations of such constants.

5. Core, 1992.

— *Types*

Types can be tokenized at several levels of specificity.

A completely abstract type is simply a `TYPE`. Variables and parameters can be declared of type TYPE, but no operations which require knowledge of the type may be used. This is useful for opaque types such as FILE, whose internal structure is not defined by standards, and must be free to vary across implementations.

Abstract integer, float, and arithmetic types can be declared, and have the corresponding operations.

Struct and Union types can be declared, and support any defined member selectors (see the next section).

A tokenized type T can be used to build up composed types, such as pointer-to-T, a struct with a field of type T, and so on. Since `stdin` is of type `FILE *`, it is legal to dereference it, but not to examine its internal fields using `->`.

— *Member selectors*

Standards often require that a type be a structure or union containing certain fields. They usually leave the order of the fields undefined, and permit implementations to support additional, internal, fields.

ANDF tokens permit declaring individual fields' names and types without specifying the complete structure or union, and without specifying the order.

— *Functions and certain kinds of macros*

Tokens can replace functions and certain kinds of macros. To be precise, they can replace macros which expand to expressions or to sequences of statements. They cannot replace arbitrary sequences of lexemes (*e.g.* `#define begin {`). (However, that kind of macro is rarely if ever platform-dependent.)

The abstraction provided by tokenized macros is valuable not only for portability, but also for static checking. In effect, the arguments and result of ANDF tokenized macros are tagged for type and storage class, so error messages can in many cases be given in terms of the macro application rather than the macro expansion.

There are actually two kinds of function token, `FUNC` and `MACRO`. The distinction is necessary because `FUNC`s cannot return lvalues, and `MACRO`s' names cannot be passed as function parameters. On the other hand, `FUNC`s can be implemented as macros and `MACRO`s by functions (as long as they don't return lvalues).

Conditional compilation

ANDF C supports ANSI C conditional compilation using C's `#if` and `#ifdef` constructs. Although `#ifdef` is reserved for traditional preprocessing, `#if` has additional functionality.

If the condition can be fully evaluated at production time, standard C preprocessor semantics are applied. In particular, the contents of each branch of the `#if` can be arbitrary lexeme sequences. Syntactic analysis then happens after the branch of the `#if` is chosen.

On the other hand, if the condition cannot be fully evaluated at production time because it contains deferred constants or expressions in the form of tokens, its evaluation is deferred to installation time. In this case, ANDF C compiles both branches of the `#if`, and evaluates the conditions and selects a branch at installation time.

For this to work, the branches of the `#if` must be syntactically complete program fragments. Moreover, the branches must be statically consistent. In particular, this means that inconsistent declarations cannot be contained within target-dependent conditions. Thus, complicated conditional declarations are best handled using the token mechanism.

— Applications of conditional compilation

Programs often contain configuration constants. ANDF deferred constants (tokens) can replace these constants where they are unresolved until install time.

Install-time conditional compilation is typically used for target-dependent constructs. But it can also be used to configure an application for a user's needs. For instance, the user could turn security features on or off, or preallocate fixed-size buffers corresponding to the expected load or available memory.

4. Defining libraries

ANDF supports defining libraries which are only linked to an application on installation. System service libraries are of course implemented this way, but other common libraries such as graphic user interfaces and numerical packages can be, as well.

As always, interface definitions used by the applications programmer and the library programmer must agree. If the interface consists only of functions, static variables, and explicit types, it may be defined simply using C prototypes and declarations.

If the interface also includes macros or opaque types, ANDF's token mechanism should be used to define them abstractly.

One of ANDF's special features is a name space of unique identifiers, similar to Internet addresses, so that there is no possibility of namespace clashes.

Thus, ANDF supports separately distributed libraries.

Advantages of tokenized libraries

Target-dependent compilation is powerful, but tokens have many advantages over it.

Target-dependent compilation in fact gives source which has many variants. Both static analysis and dynamic testing of such code is much harder than with single-source code.

Tokens force a clean definition of an interface.

Tokens improve error-checking because they are abstract.

Tokens make it more likely that the interface will be robust.

5. Suggested procedures

In our porting experience at OSF, we have developed procedures for porting existing software to ANDF.

The general outline of this procedure is:

- check-out of the ANDF compiler;
- survey of the application's portability;
- modification of the application to increase its portability;
- validation of the result;
- tuning on particular platforms.

The first step is simply a "sanity check".

The portability survey, the second step, is critical. It usually allows estimating the amount of porting effort necessary. For some applications, this effort may be prohibitive. For others, it may be negligible. The portability survey should give sufficient information to permit an informed go/no-go decision on the ANDF port.

The third step modifies the application to make it portable and represents the bulk of the time investment.

Validation, the fourth step, recognizes that no technology is perfect. There may be installer bugs, or hidden platform dependencies in the application.

Finally, the ANDF technology does allow tuning for particular platforms through the use of platform-specific library definitions.

Compiler check-out

The first step of porting to ANDF is checking that the application's language usage is accepted by the ANDF compiler. (We assume that the application compiles successfully using the standard compiler available on the machine.) The possible problems here are that the application uses non-standard constructions, or that the compiler has bugs.

This check is performed by compiling the application with the ANDF C compiler in "native mode". In this mode, it uses the libraries available on the local platform rather than the tokenized libraries. Thus issues of correct interface usage are separated from issues of correct language usage.

Compiling and testing the application compiled in native mode is a sanity check for the application, the C producer, and the local installer and often detects trivial errors.

— Other tools

There are stand-alone C portability checkers on the market. The tcc (ANDF) compiler itself is, however, an excellent static portability checker and strict ANSI C compiler, and generally gives good error messages.

Portability survey

The portability survey, the second step, is critical. It usually allows estimating the amount of porting effort necessary. For some applications, this effort may be prohibitive. For others, it may be negligible. The portability survey should give sufficient information to permit an informed go/no-go decision on the ANDF port.

The primary emphasis of this step is on correct usage of the API, including the effects that an abstract interface has on types used within the program.

— Tools

As for interfaces, the ANDF header files for the various APIs are maximally abstract, that is, they contain no information other than that specified by the specification of the interface. Thus any dependence on internal fields is detected at production time, even if every actual implementation supplies consistent additional information.

One area where style and portability checkers such as lint, and OSPC,[6] and ProQA are stricter than tcc is inter-module consistency. This is useful whenever applications do not use common header files, but instead declare externals explicitly. Although it is considered to be poor practice, it is allowed by ANSI C and tcc.

Also, tcc performs no dynamic checking as do tools such as CenterLine C[7] and the Model Implementation C compiler.[8] These tools can be useful for development and debugging.

Portability engineering

In the third step, target dependencies are eliminated.

There are two main strategies to follow:

One is rewriting code to make it respect the standard APIs. This can be a long and arduous process.

Another possible approach is to create a “translation” layer. A set of tokens is defined in place of the interfaces the application expects, and target-dependent token definition libraries translate these tokens to appropriate operations.

Our experience is that translation layers are cost-effective when a large application has been written to a non-standard interface. They are also a

6. Open Systems Portability Checker from Knowledge Software, 62 Fernhill Road, Farnborough, Hants GU14 9RZ England; OSPC@knosof.uucp.
7. CenterLine C (formerly Saber C) from CenterLine Software, 10 Fawcett St., Cambridge, Mass. 02138, USA.
8. Also from Knowledge Software.

cost-effective way of emulating widely-used *de facto* standards such as BSD header files.

In both the rewriting and the translation strategies, it is often the case that a parameterized application has one target platform which is much closer to standard interfaces than others. This is often a good starting point for porting to the standard APIs.

Validation

Validation, the fourth step, recognizes that no technology is perfect. There may be installer bugs, or hidden platform dependencies in the application. We therefore recommend that an application ported to ANDF be validated on all convenient platforms. Our experience so far is that producer and installer bugs are rare. They should become rarer, and confidence in producers and installers should increase, once the ANDF validation suites are put in place. Still, per-platform validation gives additional confidence, and we would not recommend shipping software without such testing.

Platform-specific tuning

Finally, the ANDF technology does allow tuning for particular platforms through the use of platform-specific library definitions.

The two main areas where tuning can be effective are data declarations and use of system services. As always, the tradeoff of tuning is harder debugging and validation: naturally, validation procedures should be performed after tuning on all variants of the code.

— *Tuning data declarations*

The ANDF installer usually does a very good job of selecting layouts for data. However, it must respect the semantics of the original C program. This means for instance that struct fields must be in the same order as in their declaration. It is sometimes advantageous to pack structs differently on different machines. This can be done cleanly in ANDF by tokenizing the

struct members, and only committing to a particular layout in a platform-dependent header file. Alternatively, different declarations can be compiled conditionally.

The same techniques can be used for applications which want packed field widths to vary so as to exploit a target machine most efficiently. An example of this is Gnu Emacs's Lisp_Object, which is used to represent every object visible to its extension language. In this case, the effort required to support parameterized code is worthwhile.

— *Tuning use of system services*

Another area where platform-specific tuning can be appropriate is in the use of system services. For instance, an application which makes heavy use of file directories may wish to work on their internal representation. This is clearly a non-portable technique, but may be worthwhile for important target platforms.

Another example is interprocess communication and locking mechanisms. Different platforms may have vastly different performance characteristics for these operations, so it may be appropriate to provide for target platform parameterization of these operations.

6. Conclusion

Strictly portable applications can be as easily distributed in ANDF form as in source form, probably more so.

Taking existing portable, but not *strictly* portable, applications and moving them onto the ANDF technology requires removing target-dependent language constructs and limiting non-standard API use. It is possible, however, to interface to non-standard APIs, since ANDF's definition mechanism (tokens) is available to the library writer.

ANDF is a technology for distributing applications. Portable applications can be moved onto ANDF through a well-defined porting process, which then makes them installable on a wide range of platforms. New applications,

and existing strictly portable applications, can benefit from ANDF with almost no modification.

The ANDF technology makes easily portable applications into easily distributed applications.

7. Bibliography

C Language

Samuel P. Harbison, Guy L. Steele, Jr., *C: A Reference Manual* (3rd edition). Prentice-Hall, 1991.

X3 Accredited Standards Committee, *Programming Language C* (American National Standard for Information Systems), Doc. No. X3J11/90-012, February 14, 1990.

Portability in C

L.W. Cannon et al., *Recommended C Style and Coding Standards*. Available by anonymous FTP from cs.washington.edu as ~ftp/pub/cstyle.tar.Z.

A. Dolenc, A. Lemmke, D. Keppel, G.V. Reilly, *Notes on Writing Portable Programs in C*. Available by anonymous FTP from sauna.hut.fi as ~ftp/pub/CompSciLib/doc/portableC. {tex, sty, bib, ps.Z} or from cs.washington.edu as ~ftp/pub/cport.tar.Z.

Mark R. Horton, *Portable C Software*. Prentice Hall, 1990.

Rex Jaeschke, *Portability and the C Language*. Hayden Books, 1989.

Andrew Koenig, *C Traps and Pitfalls*. Addison-Wesley, 1989.

Henry Rabinowitz and Chaim Schaap, *Portable C*. Prentice-Hall, 1990.

Thomas Plum, *C Programming Guidelines* (2nd edition). Plum Hall, Inc. (1 Spruce Ave., Cardiff, NJ 08232), 1989.

Application Programming Interfaces

Open Software Foundation, *Application Environment Specification (AES)*, multiple volumes, Prentice Hall, 1990.

X/Open Company, Ltd., *X/Open Portability Guide* (Issue 3), 7 vols., Prentice Hall, 1988.

IEEE Std 1003.1-1988, *IEEE Standard Portable Operating System for Computer Environments*, IEEE, 1988.

Donald Lewine, POSIX *Programmer's Guide: Writing Portable* UNIX *Programs*. O'Reilly and Associates, Inc. (103 Morris St., Suite A, Sebastopol, Calif. 95472), 1991.

ANDF Tokens

P.W. Core, *The token syntax for the C producer*. DRA Malvern, 1992.

P.W. Core, *#pragma extensions to the C producer*, DRA Malvern, 1992.

Thomas J. Watt, Jr., *Tokenizing Applications in the ANDF Prototype*. OSF Research Institute, February 1990.

Application porting to ANDF

Andrew Johnson, *Application Development Using ANDF*. (slide presentation) OSF Research Institute, 1992.

Thomas J. Watt, Jr., *Porting Postgres with the Research Prototype ANDF Technology*. OSF Research Institute, December 1992.

For further information please contact:

Stavros Macrakis
macrakis@osf.org
(617) 621-7356

Copyright 1993 by Open Software Foundation, Inc.

All Rights Reserved

Permission to reproduce this document without fee is hereby granted, provided that the copyright notice and this permission notice appear in all copies or derivative works. OSF MAKES NO WARRRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. OSF shall not be liable for errors contained herein or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.