CONVEX POSIX Impacts

Application Note
Document No. 710-002130-000

November 1989

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX POSIX Impacts
Order No. DSW-313

Printed in the United States of America

CONVEX COMPUTER CORPORATION (214) 497-4000
P.O. BOX 833851 RICHARDSON, TEXAS 75083-3851
3000 WATERVIEW PKWY RICHARDSON, TEXAS 75080

With the upcoming release of ConvexOS V8.0, CONVEX has updated its system call interface for the operating system to comply with the IEEE Std 1003.1-1988 (POSIX.1). This standard is the first in the POSIX set, all of which are based largely on UNIX Seventh Edition, UNIX System III, UNIX System V, BSD V4.2, and BSD V4.3 documentation.

To provide you with an overview of POSIX.1 and its enhancements in a timely manner, CONVEX has produced the attached *CONVEX POSIX Impacts* document. While this document may not answer every question you have, it provides a detailed summary of POSIX.1 and how it has been integrated into ConvexOS V8.0.

The scope of this document is limited to POSIX.1 and how it might affect your applications. It includes a discussion of ANSI C, as POSIX is currently defined in terms of the C programming language. CONVEX will offer an ANSI C-based implementation with the release of CONVEX C V4.0. At that time, and not before, ConvexOS V8.0 will be POSIX.1 compliant.

This document does *not* cover modifications and enhancements to ConvexOS or CONVEX C that are not POSIX-related. Please consult the appropriate product documentation or release notices for detailed information that is not covered here.

CONVEX is committed to maintaining POSIX compliance as each POSIX working committee completes and ratifies its group of standards.

If you have a current software maintenance agreement, you will be sent ConvexOS V8.0 automatically when it becomes available; there is no need to place an order. If you are unsure of your maintenance status, please contact your sales representative.

It is important that you familiarize yourself with the issues discussed in the *CONVEX POSIX Impacts* document so you understand what is occurring. If you have any questions, please address them to the CONVEX Technical Assistance Center (TAC) via the "contact" utility, by phone at 1-800-952-0379, or by FAX at 1-214-497-4560.

Document No. 710-002130-000

Table of Contents

Appendices

Chapter 1
Overview

After the IEEE ratified the Portable Operating System Interface for Computing Environments (POSIX.1) standard, CONVEX redefined the system call interface for the operating system. ConvexOS V8.0 will be an implementation of the Berkeley UNIX operating system containing POSIX.1 functionality with extensions for supercomputer environments. ConvexOS V8.0 retains backward compatibility for existing binaries and sources.

Recently, the ANSI standard X3J11 was also approved. It clarifies some ambiguities and modifies other current practices. Previously, *The C Programming Language*, First Edition by Kernighan and Ritchie, served as the "standard" for C. Now, with the ANSI standard, there are formal requirements for all aspects of the C language.

CONVEX will be offering this enhanced version of ANSI C with the release of CONVEX C V4.0. At that time, and not before, ConvexOS V8.0 will be POSIX.1 compliant. ANSI C is fully compatible with POSIX.1. The two interact without any problems because POSIX.1 was standardized after the ANSI C library specifications.

CONVEX is committed to maintaining POSIX compliance as each POSIX working committee completes and ratifies their group of standards.

This document describes POSIX.1 and ANSI C enhancements and how they might affect your applications. Where possible, examples are given.

The term "POSIX.1" refers specifically to IEEE Std 1003.1-1988.

The term "POSIX" refers to the general group of proposed standards sponsored by various working committees of the IEEE.

The term "ConvexOS V8.0" refers to the CONVEX implementation of the Berkeley UNIX operating system containing POSIX.1 functionality with extensions for supercomputer environments.

The term "CONVEX C V4.0" refers to the CONVEX C compiler that supports ANSI C.

Chapter 2
POSIX

2.1 What Is POSIX?

The Portable Operating System Interface for Computer Environments (POSIX.1) is a standard operating system interface and environment based on the UNIX operating system. It is intended to support application portability at the source code level. Actually, POSIX is a group of proposed standards sponsored by various working committees of the IEEE. The IEEE Std 1003.1-1988 is the first of the POSIX standards to be adopted and represents a standard system call interface.

POSIX.1 describes external characteristics and facilities that are important to application developers rather than internal construction techniques used to achieve these capabilities. Special emphasis is placed on those functions and facilities needed in a wide variety of commercial applications. POSIX.1's objective is for a strictly conforming application to compile on a new host without any code modifications.

POSIX.1 is currently defined in terms of a C programming language interface to the operating system, though interfaces for other languages will soon be defined. CONVEX will offer an ANSI C-based implementation with the release of CONVEX C V4.0. At that time, and not before, ConvexOS V8.0 will be POSIX.1 conforming. POSIX.1 and ANSI C interact seamlessly because POSIX.1 was developed after the ANSI C library specification, which is the only point of interaction between the two.

POSIX is based largely on UNIX Seventh Edition, UNIX System III, UNIX System V, BSD V4.2, and BSD V4.3 documentation.

2.2 Types of Conformance

POSIX.1 requires the following conformance criteria to be defined:

- Implementation
- Application
- C Language

2.2.1 Implementation

A conforming implementation meets the following criteria:

- The system supports all required interfaces and functional behavior described in IEEE Std 1003.1-1988.
- Additional extensions are allowed but, if used, must be identified as such. If using them changes the defined functions, system documentation must describe an environment in which the program can be run according to the IEEE standard.

A group that claims their implementation conforms to the POSIX.1 standard must provide a conformance document stating so. Specifications for this document are contained within the IEEE standard.

2.2.2 Application

All applications that claim conformance use only language-dependent services for the C programming language and are classified as one of the following:

- Strictly Conforming Application—requires only facilities described in the IEEE standard and applicable language standards.
- Conforming Application—uses facilities described in the IEEE standard and an approved conforming language binding.
- Conforming Application Using Extensions—differs from a conforming application only in that it uses nonstandard facilities that are consistent with the IEEE standard.

2.2.3 C Language

Implementations conforming to the POSIX.1 standard with the C language will claim one of the following:

- C Standard Language-Dependent System Support—to be used with the offered CONVEX C compiler that complies with the ANSI standard X3J11 (released with CONVEX C V4.0).
- C Language-Dependent System Support—to be used with the current C compiler.

Chapter 3
Backward Compatibility

3.1 Binaries

Because binary backward compatibility is supported, a method has been devised to distinguish between POSIX.1 and non-POSIX.1 binaries. To determine the POSIX.1 mode of a binary, a new flag bit in the *a.out* file header indicates a POSIX.1 application. This bit is zero in all existing binaries and is now set by *ld* at the direction of the compiler (for example, *cc*). It is accessed by the *-E posix* option. With the release of CONVEX C V4.0, and not before, the C compiler specifies the *-E posix* option by default.

At run time, the kernel then determines the behavior expected by an application. Different behavior is provided for the following:

- Error number (*errno*) values
- Default signal masks
- Signal delivery permissions
- Interrupted system calls

Binaries created previous to ConvexOS V8.0 do not have the new bit set. This way, the kernel can determine the behavior the application expects (with respect to the above items).

3.1.1 What Is Not Compatible

At this time, the only areas where it is difficult to accommodate existing binaries are those applications that deal with job control. This includes process group membership and the notion of the terminal's controlling process group. Typically they are categorized as either login shells or applications that start shells. Examples of applications that required modification are:

- *csh*
- *emacs*
- *init*
- *login*
- *ps* and libvm.a
- *script*
- *stty*
- *window*
- *xterm*

If you run old versions of programs on the ConvexOS V8.0 kernel, one of two things will happen:

1. If the application checks the return value from system calls, it will complain about something failing and let you know where the failure is.

2. If an application that starts a shell sets up the terminal process group incorrectly but does not notice that something failed and executes the shell, the following diagnostic from the shell appears:

```
Warning: no access to tty; thus no job control in this shell...
```

   The interrupt (CTRL-C) or quit (CTRL-\) signals will not work correctly.

3.1.2 Determining Compatibility

To determine if an application will behave differently with POSIX.1 functionality, *grep* for the following keywords in the application source:

getpty()	Programs that call *getpty()* go on to create interactive shells associated with *pty*. Examples are *xterm* and *window*.
TIOCNOTTY	Once you discard your controlling terminal on a POSIX.1 kernel, you can only regain it by calling the *setsid()* function.
TIOCSPGRP	Look for the sequence of calls to *setpgrp()* and *ioctl()*. An application can no longer TIOCSPGRP to a process group that does not exist.

3.2 Sources

Several interface functions in POSIX.1 have different semantics from those prior to the release of ConvexOS V8.0, while a few functions have similar semantics but yield either different return or *errno* values. Because the ultimate goal is ease of use, include files will work in backward compatible mode by default for existing compilers and use the old definitions of defines, macros, and return types where the old and new usages are not compatible.

With the release of CONVEX C V4.0, and not before, maintaining source code backward compatibility will affect <signal.h> and <sys/wait.h>.

3.2.1 <signal.h>

SIG_DFL and SIG_IGN have traditionally been pointers to functions returning int (i.e., *int(*)()*). For POSIX.1, all signal handling functions should be updated to those returning void (i.e., *void(*)()*), as illustrated in the following example:

Previous code:

```
int sighan(int num) {
      /* signal handled here */
}
void foo() {
int (*oldval)();
      oldval = signal(SIGINT,sighan);
      ...
      (void)signal(SIGINT,oldval);
}
```

Preferred code:

```
#define sigfunc void      /* change to int on old style system */
typedef sigfunc (*sigfunc_ptr)();
sigfunc sighan(int num) {
      /* signal handled here */
}
foo() {
sigfunc_ptr oldval;
      oldval = signal(SIGINT,sighan);
      ...
      (void)signal(SIGINT,oldval);
}
```

3.2.2 <sys/wait.h>

Macros WIFSTOPPED, WIFSIGNALED, and WIFEXITED previously operated on *union wait* arguments. For POSIX.1, they operate on int arguments. An example follows.

Previous code:

```
void foo() {
      union wait bar;
      if ( wait(&bar) != -1
      &&  WIFEXITED(bar) )
            (void)puts("foobar!");
}
```

Preferred code:

```
void foo() {
      union wait bar;
      if ( wait(&bar.w_status) != -1
      && WIFEXITED(bar.w_status) )
            (void)puts("new foobar!");
}
```

In the above example, the argument to *wait* was changed to a pointer to type int. Note that the w_status field of the *union wait* may be used.

3.3 Permissions and Protections

POSIX.1 functionality means that permissions and protections will change slightly. Most of the differences are in using a new user identification (UID) field. In addition to the traditional effective and real UID and group identification (GID) values, POSIX.1 functionality carries with it the new saved UIDs and GIDs, which are the same as the effective IDs at the time *exec()* is called. This facility allows an application to use *setuid()* or *setgid()* to change between real and effective IDs. The reason for this feature is that many programs currently requiring installation in the file system as *set-user-ID* to root can now be set to some other user.

setregid() and *setreuid()* always change the *saved set-group-ID* and *saved set-user-ID* when the real ID is changed, regardless of POSIX.1 modifications. This may make a difference when "*exec*'ing" a POSIX.1 application from a non-POSIX.1 one.

kill() — POSIX.1 says process *S* can kill process *R* either if *S*'s effective UID is root or if *S*'s real or effective UID matches the real or saved UID of *R*. BSD V4.2, on the other hand, says *S*'s effective UID must either be root or match *R*'s effective UID.

setgid() — There is a difference in supplementary group handling. For POSIX.1, *setuid()* and *setgid()* leave the supplementary group list unchanged. When changing the real GID, BSD V4.2's version of *setgid()* removes the old real GID from the group list and inserts the new real GID.

3.4 Empty Path Names

When dealing with path names (such as *open()* and *exec()*), POSIX.1 requires an *errno* return of ENOENT when the path name is a pointer to the empty string "".

This is in contrast with BSD V4.2 behavior that maps "" to ".". Refer to the *intro*(2) man page.

3.5 *exec()*

ConvexOS V8.0 has the capability, through the use of *-o nosuid*, to mount a file system that does not allow the execution of set UID or set GID programs. If a user attempts to execute a set UID program from such a file system, the *exec()* call fails and *errno* is set to EPERM.

Because POSIX.1 does not deal with the concept of disallowing set UID execution, *exec* fails and returns EPERM. Note that POSIX.1 does not include EPERM as an *exec*-specific *errno*, so this extension can be used without becoming nonconforming.

Performing an *exec()* of a POSIX.1 image causes all signals to interrupt system calls by default. The application may clear the SV_INTERRUPT bit and thus restart system calls after a signal. This action makes an application *not* strictly POSIX.1 conforming.

3.6 Signals

3.6.1 *abort()*

The *abort()* library call has been changed. The SIGABRT signal now corresponds to the old SIGIOT signal.

3.6.2 System Call Restart

POSIX.1 requires that, by default, system calls not be restarted after a signal. The following system calls have EINTR in their list of *errno* values and are affected:

- *fcntl()*
- *getdirentries()*—not required by POSIX.1; will be used to implement the POSIX.1 *readdir()* function
- *msleep()*—not required by POSIX.1
- *open()*
- *read()*
- *recv()*—not required by POSIX.1
- *write()*
- *wait()*

3.6.3 Inheritance

When dealing with inheritance, POSIX.1 requires that pending signals and alarms be cleared. This is in contrast with BSD V4.2 behavior where the child inherits pending signals at *fork()* time.

3.6.4 Permissions

The POSIX.1 permission to send a signal (*kill()*) is based on the saved UID of the sender. Where the sender and the receiver differ in terms of POSIX.1, signal rules for the sender are enforced.

3.7 Use of *vfork()*

Using the *vfork()* system call, which is not a POSIX.1 function, can affect job control behavior not defined in POSIX.1. For example, the terminal driver will not deliver the SIGTTOUT signal to an application that has done a *vfork()* but not yet performed an *exec()*. Using *vfork()* makes an application *not* strictly POSIX.1 conforming.

3.8 tty Driver

POSIX.1 has defined a set of functions for interfacing to the tty driver that are more portable than current system calls. The tty line disciplines, therefore, have changed such that both line discipline 0 (#defined as OTTYDISC in <sys/ioctl.h>) and line discipline 2 (#defined as NTTYDISC in <sys/ioctl.h>) have similar behavior. The only difference is that using line discipline 0 now disables the following special control characters:

werase word erase (CTRL-W)

rprnt reprint (CTRL-R)

flush flush (CTRL-O)

lnext literal next (CTRL-V)

susp suspend (CTRL-Z)

dsusp delayed suspend (CTRL-Y)

These control characters can then be explicitly enabled, on an individual basis, by a user with *stty* or by an application issuing the appropriate *ioctl()* system calls. In these cases, after enabling the control characters, there is no difference between the line disciplines. Switching to line discipline 2 will also enable these control characters. Setting the line discipline to the value of the current discipline has no effect.

Some applications used the different pre-POSIX.1 line disciplines to avoid receiving the signals SIGTTIN, SIGTTOU, and SIGTSTP. This no longer works reliably because applications now running under line discipline 0 may receive these job-control-related signals in the unlikely event that an application explicitly sets a suspend character. These applications should explicitly ignore the signals with code like the following example:

```
signal(SIGTTIN, SIG_IGN);
signal(SIGTTOU, SIG_IGN);
signal(SIGTSTP, SIG_IGN);
```

Very few, if any, user applications are affected by this change (it is reflected in */etc/init* and */bin/login*). For more on the tty driver, refer to the *stty*(1), *termios*(4), and *tty*(4) man pages.

Chapter 4
CONVEX C

4.1 Compilers

CONVEX C V4.0 will be shipped to all customers. In its default mode, the compiler uses an extended version of the ANSI C language and libraries. A command line option provides backward compatibility with previous compilers, include files, and libraries.

Operating with ConvexOS V8.0 and CONVEX C V4.0, you can no longer invoke the pcc-based compiler (hereafter referred to as common C) with the command "cc" as in V7.1, and earlier, of the operating system. The common C compiler that was previously shipped to all customers and installed as */bin/cc* is now called */bin/pcc*. CONVEX C V4.0 will be installed as *cc*.

Those who buy an optional license for CONVEX C (formerly known as Vector C) will receive a version of the compiler that supports vectorization and parallelization. Customers who do not buy the optional license will receive a version of CONVEX C that does not perform vectorization or parallelization.

4.2 Compatibility Modes

The C compiler offers four modes of compatibility that provide different language features, libraries, and include-file contents:

- Default
- Conforming
- Strict
- Backward Compatible

The modes differ mainly in language specifications (for example, keywords and data types) and system functions used. Default, conforming, and strict modes represent varying levels of adherence to ANSI C and POSIX.1 standards. Each mode is attained by using a specific option on the compiler command line.

4.2.1 Default

The default mode (no option necessary) provides ANSI C language compatibility and POSIX.1 functions along with CONVEX extensions.

4.2.2 Conforming

The conforming mode (-std option) provides ANSI C language compatibility and access to ANSI C and POSIX.1 libraries. Extensions like the long long int data type and the asm keyword are not permitted.

4.2.3 Strict

The strict mode (-str option) provides ANSI C compatibility and accepts the same language features as those in the conforming mode. This mode, though, attempts to detect usage that prevents a program from being a strictly conforming implementation of ANSI C. Only functions defined by the ANSI C standard are automatically available in this mode. Other functions are accessible by specifying the relevant library on the command line. Refer to section 4.3, "Libraries."

4.2.4 Backward Compatible

The backward compatible mode (-pcc option) is compatible with the common C compiler, although minor differences exist in areas not well-defined by the language. This mode causes include files to be non-ANSI C and non-POSIX.1 conforming. In this mode, the compiler generates errors for all new ANSI C keywords. A program that behaves differently with CONVEX C than it does with common C probably depends on some particular behavior of the compiler that the language does not define. Some examples follow.

Example 1:

```
int a, b;
main() {
   a = 1;
   b = 3;
   f(a++, b = a);
}
```

In this example, the first argument to *f*() is 2. The second argument is either 1 or 2, depending on the order in which the compiler chooses to perform the increment of *a* and the assignment to *b*. The language leaves this undefined.

Example 2:

```
main() {
   unsigned int i;
   (unsigned) i = (-3);
}
```

Here, the language disallows casts from appearing on the left-hand side of the assignment, but common C allows it in some cases. Often, *lint* will help in finding such errors.

4.3 Libraries

The four compatibility modes not only select language specifications but also libraries that define system functions for the following:

- CONVEX extensions
- POSIX.1 functions
- ANSI C language specifications
- Backward compatible features

The following options allow you to link with the specified libraries:

default	CONVEX extensions, POSIX.1, and ANSI C libraries
-std	POSIX.1 and ANSI C libraries
-str	ANSI C library
-pcc	Backward compatible library

These libraries are searched automatically in the correct order for object code that will be linked into the executable program. The libraries are only part of the solution to using system functions in an application; include files also contain information that is pertinent.

4.4 Include Files

CONVEX C V4.0 provides include files that can be interpreted in several ways. By default, include files are interpreted to define only ANSI C features. Defining certain conditional compilation symbols causes additional features to be defined by include files. These symbols can be defined by any of these equivalent methods:

- #define directive in the program text
- -D switch on the command line
- -D switch in the CCOPTIONS environment variable

Defining the symbol _POSIX_SOURCE causes the include files to be interpreted to define all POSIX.1 and ANSI C features. The symbol _CONVEX_SOURCE may be defined for CONVEX extensions *if, and only if,* the _POSIX_SOURCE symbol is defined. It is an error to define _CONVEX_SOURCE if _POSIX_SOURCE has not been defined.

When both _CONVEX_SOURCE and _POSIX_SOURCE are defined, the include files are interpreted to define all ANSI C features, POSIX.1 functions, and CONVEX extensions.

Libraries are structured to support these facilities.

Compiler usage and examples are discussed in the *CONVEX C User's Guide.* Compatibility modes and language features are discussed in the *CONVEX C Language Reference Manual.*

Chapter 5
ANSI C

5.1 Background and Benefits

In the past, *The C Programming Language*, First Edition by Kernighan and Ritchie, served as the "standard" for C. However, it left parts of the language unclear and did not specify the C library. Now, with the ANSI standard X3J11 approved, there are formal requirements for all aspects of C including the library, the preprocessor, and the language itself. This standard is significant with regard to enhancing C program portability. Knowing ANSI C, you can more easily port C programs from one machine to another. As long as a program strictly follows only ANSI C functions and constructs, it will compile in every conforming implementation.

Other benefits are that ANSI C

- Provides a new way to declare functions that helps reduce errors by allowing type checking between the call and declaration
- Allows greater optimization because of new rules for floating-point expression evaluation and aliasing
- Requires that parentheses enforce the order of evaluation of expressions, which is important in numerical programming

5.2 Differences

If you decide to employ ANSI C, use the default mode of the compiler.

Note	Upgrading your code to be compatible with the ANSI C standard is not required. Refer to section 4.2.4, "Backward Compatible."

The following differences exist between traditional C and ANSI C:

- Preprocessing is more carefully defined and explicitly token based:
  - ## for token concatenation (replaces /**/ hack)
  - # for string creation of a formal macro parameter (#x becomes "x")
  - #elif control for better nesting of conditional compilation directives
  - #pragma control
  - Splicing lines ending with backslash permitted everywhere

  - — Parameters inside strings are not replaced in ANSI C (use #x instead)
  - — Adjacent string literals are concatenated—"abc""def" is equivalent to "abcdef"

- Keyword "void" is clearly defined.
- Keyword "const" allows you to declare objects constant.
- Keyword "enum" is less strongly typed (ANSI C's definition is identical to CONVEX C V3.0's and forward compatible with scalar C's).
- New escape sequences include "\a" for alert (bell), "\v" for vertical tab, "\?" for question mark, and "\xhh" for hex numbers.
- Numerals 8 and 9 are no longer octal digits.
- Suffixes for constants include "U" for unsigned, "L" for long, and "L" or "F" for floats; the types of unsuffixed constants are more rigorously defined.
- Characters and other integral types may be declared as carrying a sign with the keyword "signed."
- Using "long float" as a synonym for "double" is withdrawn; "long double" can be used to declare extra-precision floating points ("long double" has the same representation as "double" in CONVEX C V4.0).
- The "void *" type is now available as a generic pointer type and explicit rules have been defined for pointer casts. It is useful for data abstraction.
- <limits.h> and <float.h> give minimum implementation-defined numeric limits.
- Literal strings are no longer modifiable and can be placed in read-only memory. CONVEX C V4.0 places them in the text segment.
- Type conversions change from "unsigned always wins" to "promote to the smallest capacious-enough type" (this may present or prevent surprises in sign extension because the old standard was "implementation defined").
- Old assignment operators such as "=+" are gone, though they will be supported in the backward compatible mode.
- Assignment operators are single tokens; no space may come between them. For example, "+=" is allowed but "+ =" is not.
- A compiler can no longer treat mathematically associative operators as computationally associative. For example, "a+(b+c)" is not the same as "(a+b)+c." Parentheses enforce the order in which floating point operators are evaluated. The compiler may still choose which operator to do first in "a+b+c."
- "Unary +" available for symmetry with "unary -."
- The "size of" operator returns an unsigned quantity; the exact type is given by unsigned integer or char *size_t*, which is declared in <stddef.h> and other include files. *size_t* is unsigned int in CONVEX C V4.0.
- The difference of two pointers returns the type *ptr_diff* that is declared in <stddef.h>. This type is required by the language to be a signed integer or char type; it will be signed int in CONVEX C V4.0.

- The "&" operator may not be applied to a register variable even if that variable was not bound to a register by the implementation.

- The result of a shift operator has the type of the left operand; the right operand's type does not affect the type of the result. For example:

```
main() {
   int i = 1;
   long long int j = 32;
   printf("%lld\n", i<<j);
}
```

  prints 2^32 (2 raised to 32nd power) using CONVEX C V3.0 (or scalar C). This is the long long int result of shifting 1 left 32 places. In ANSI C, the expression *i<<j* would have the result type int so the *printf()* statement would have to be modified to

```
printf("%lld\n", ((long long int)i)<<j);
```

  to get the same result. The original program compiled in ANSI mode would pass an int (4 byte) 0 to *printf()* instead of an 8-byte representation of 2^32.

- Function prototypes are introduced, including implicit type conversion for actual parameters to appropriate formal parameter types in function calls. The method for writing variadic functions is modified slightly (a variadic function is one with varying numbers of arguments).

- Empty declarations that have no declarators and do not declare at least a structure, union, or enumeration are not allowed; however, a declaration with just a structure or union tag redeclares that tag even if it was declared in an outer scope.

- External declarations without any specifiers or qualifiers (only a naked declarator) are not allowed.

- An "extern" declaration within an inner block is not visible outside that block.

- The scope of parameters is injected into a function's compound statement so that variable declarations at the top level of the function cannot hide the parameters.

- Unions may be initialized; the initializer refers to the first member.

- Automatic structures, unions, and arrays may be initialized with compile time constant values.

5.3 Name Space Issues

The ANSI standard requires that neither the compiler nor the library define names, other than those specified by the standard, in the user's name space. This causes the names of many support routines in the C library to change. This change is beneficial because it allows users to choose names that do not conflict with those in the library. Those who have code that calls undocumented parts of the library must modify their code to use only the standard library interfaces.

The ANSI C and POSIX.1 standards provide a number of rules that split the available identifiers into two classes or name spaces:

- Those available for use by applications
- Those available for use by implementations (i.e., compilers, libraries, etc.)

By confining the implementation to use a known set of names, the user is assured that the program will not contain names that conflict with those used, for example, as support routines to implement the I/O system. The programmer has a responsibility to ensure that the application does not use names in the name space reserved for the implementation. Such use can cause the application to fail at compile, load, or run time.

Rules that separate application and implementation name spaces are quite complex because they have been designed (by the ANSI and POSIX committees) to reflect existing practice. The following rules define a name space that is slightly more restrictive than the application name space defined by the ANSI C standard:

- Do not use an identifier beginning with an underscore.
- Do not use an identifier containing the currency symbol ($).
- Do not use an identifier for which either the ANSI C or POSIX.1 standard defines a meaning, regardless of whether the header pertaining to the identifier has been included (except, of course, with the meaning defined by the standard).

A good description of the complete set of rules and listings of ANSI C predefined identifiers is available in *Standard C*, First Edition by Plauger and Brodie. Also, refer to both the ANSI C and POSIX.1 standards.

APPENDIX A
Bibliography

American National Standard for Information Systems—Programming Language C. X3J11. New York: The American National Standards Institute, 1988.

Barkakati, Naba. *The Waite Group's Essential Guide to ANSI C.* 1st ed. Indianapolis: Howard W. Sams & Company, 1988.

CONVEX C Compiler V4.0 documentation.

Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language.* 2d ed. Englewood Cliffs: Prentice-Hall, Inc., 1988.

Plauger, P. J. and Brodie, Jim. *Standard C.* 1st ed. Redmond: Microsoft Press, 1988.

Portable Operating System Interface for Computer Environments. 1003.1. New York: The Institute of Electrical and Electronics Engineers, Inc., 1988.