IEEE P1003.2 Draft 11.2 – September 1991

2.3 Built-in Utilities

Any of the standard utilities may be implemented as *regular built-in* utilities within the command language interpreter. This is usually done to increase the performance of frequently-used utilities or to achieve functionality that would be more difficult in a separate environment. The utilities named in Table 2-2 are frequently provided in built-in form. All of the utilities named in the table have special properties in terms of command search order within the shell, as described in 3.9.1.1.

Table 2-2 – Regular Built-in Utilities

`cd`	`false`	`kill`	`true`	`wait`
`command`	`getopts`	`read`	`umask`	

However, all of the standard utilities, including the regular built-ins in the table, but not the special built-ins described in 3.14, shall be implemented in a manner so that they can be accessed via the POSIX.1 {8} *exec* family of functions (if the underlying operating system provides the services of such a family to application programs) and can be invoked directly by those standard utilities that require it (`env`, `find`, `nohup`, `xargs`).

Since versions shall be provided for all utilities except for those listed previously, an application running on a system that conforms to both POSIX.1 {8} and Section 7 of this standard can use the *exec* family of functions, in addition to the shell command interface in 7.1 [such as the *system* () and *popen* () functions in the C binding] defined by this standard, to execute any of these utilities.

2.3.1 Built-in Utilities Rationale. *(This subclause is not a part of P1003.2)*

In earlier drafts, the table of built-ins implied two things to a conforming application: these may be built-ins and these need not be executable. The second implication has now been removed and all utilities can be *exec*-ed. There is no requirement that these be actually built into the shell itself, but many shells will want to do so because 3.9.1.1 requires that they be found prior to the **PATH** search. The shell could satisfy its requirements by keeping a list of the names and directly accessing the file-system versions regardless of **PATH**. Providing all of the required functionality for those such as `cd` or `read` would be more difficult.

There were originally three justifications for allowing the omission of *exec*-able versions:

(1) This would require wasting space in the file system, at the expense of very small systems. However, it has been pointed out that all nine in the table can be provided with nine links to a single-line shell script:

```
$0 "$@"
```

(2) There is no sense in requiring invocation of utilities like `cd` because they have no value outside the shell environment or cannot be useful in a child process. However, counter-examples always seemed to be available for even the strangest cases:

```
find . -type d -exec cd {} ; -exec foo {} ;
```
(which invokes `foo` on accessible directories)

```
ps . . . | sed . . . | xargs kill
```

```
find . -exec true ; -a . . .
```
(where `true` is used for temporary debugging)

(3) It is confusing to have something such as `kill` that can easily be in the file system in the base standard, but requires built-in status for the UPE (for the `%` job control job ID notation). It was decided that it was more appropriate to describe the required functionality (rather than the implementation) to the system implementors and let them decide how to satisfy it.

On the other hand, there were objections raised during balloting that any distinction like this between utilities was not useful to applications and that the cost to correct it was small. These arguments were ultimately the most effective.

There were varying reasons for including utilities in the table of built-ins:

`cd`, `getopts`, `read`, `umask`, `wait`
: The functionality of these utilities is performed more simply within the context of the current process. An example can be taken from the usage of the `cd` utility. The purpose of the utility is to change the working directory for subsequent operations. The actions of `cd` affect the process in which `cd` is executed and all subsequent child processes of that process. Based on the POSIX.1 {8} process model, changes in the process environment of a child process have no effect on the parent process. If the `cd` utility were executed from a child process, the working directory change would be effective only in the child process. Child processes initiated subsequent to the child process that executed the `cd` utility would not have a changed working directory relative to the parent process.

`command`
: This utility was placed in the table primarily to protect scripts that are concerned about their **PATH** being manipulated. The ‘‘secure’’ shell script example in 4.12.10 would not be possible if a **PATH** change retrieved an alien version of `command`. (An alternative would have been to implement `getconf` as a built-in, but it was felt that it carried too many changing configuration strings to require in the shell.)

`kill`
: Since common extensions to `kill` (including the planned User Portability Extension) provide optional job control functionality using shell notation (`%1`, `%2`, etc.), some implementations would

find it extremely difficult to provide this outside the shell.

`true`, `false`

These are in the table as a courtesy to programmers who wish to use the '`while true`'' shell construct without protecting `true` from **PATH** searches. (It is acknowledged that '`while :`'' also works, but the idiom with `true` is historically pervasive.)

All utilities, including those in the table, are accessible via the functions in 7.1.1 or 7.1.2 [such as *system* () or *popen* ()]. There are situations where the return functionality of *system* () and *popen* () is not desirable. Applications that require the exit status of the invoked utility will not be able to use *system* () or *popen* (), since the exit status returned is that of the command language interpreter rather than that of the invoked utility. The alternative for such applications is the use of the *exec* family. (The text concerning conformance to POSIX.1 {8} was included because where *exec* is not provided in the underlying system, there is no way to require that utilities be *exec*-able).

2.4 Character Set

Table 2-3 – Character Set and Symbolic Names

Symbolic Name	Glyph	Symbolic Name	Glyph	Symbolic Name	Glyph
<NUL>		<colon>	:	<circumflex>	^
<alert>		<semicolon>	;	<circumflex-accent>	^
<backspace>		<less-than-sign>	<	<underscore>	_
<tab>		<equals-sign>	=	<low-line>	_
<newline>		<greater-than-sign>	>	<grave-accent>	`
<vertical-tab>		<question-mark>	?	<a>	a
<form-feed>		<commercial-at>	@	<b>	b
<carriage-return>		<A>	A	<c>	c
<space>		<B>	B	<d>	d
<exclamation-mark>	!	<C>	C	<e>	e
<quotation-mark>	"	<D>	D	<f>	f
<number-sign>	#	<E>	E	<g>	g
<dollar-sign>	$	<F>	F	<h>	h
<percent-sign>	%	<G>	G	<i>	i
<ampersand>	&	<H>	H	<j>	j
<apostrophe>	´	<I>	I	<k>	k
<left-parenthesis>	(	<J>	J	<l>	l
<right-parenthesis>	)	<K>	K	<m>	m
<asterisk>	*	<L>	L	<n>	n
<plus-sign>	+	<M>	M	<o>	o
<comma>	,	<N>	N	<p>	p
<hyphen>	-	<O>	O	<q>	q
<hyphen-minus>	-	<P>	P	<r>	r
<period>	.	<Q>	Q	<s>	s
<full-stop>	.	<R>	R	<t>	t
<slash>	/	<S>	S	<u>	u
<solidus>	/	<T>	T	<v>	v
<zero>	0	<U>	U	<w>	w
<one>	1	<V>	V	<x>	x
<two>	2	<W>	W	<y>	y
<three>	3	<X>	X	<z>	z
<four>	4	<Y>	Y	<left-brace>	{
<five>	5	<Z>	Z	<left-curly-bracket>	{
<six>	6	<left-square-bracket>	[	<vertical-line>	|
<seven>	7	<backslash>	\	<right-brace>	}
<eight>	8	<reverse-solidus>	\	<right-curly-bracket>	}
<nine>	9	<right-square-bracket>	]	<tilde>	~

Conforming implementations shall support one or more coded character sets. Each supported coded character set shall include the *portable character set* specified in Table 2-3. The table defines the characters in the portable character set and the corresponding symbolic character names used to identify each character in a character set description file. The names are chosen to correspond closely with character names defined in other international standards. The table contains more than one symbolic character name for characters whose traditional