-- --

IEEE P1003.2 Draft 11.2 – September 1991

Section 3: Shell Command Language

The shell is a command language interpreter. This section describes the syntax of that command language as it is used by the `sh` utility and the functions in 7.1 [such as *system* () and *popen* () in the C binding].

The shell operates according to the following general overview of operations. The specific details are included in the cited clauses and subclauses of this section. The shell:

(1) Reads its input from a file (see `sh` in 4.56), from the `−c` option, or from one of the functions in 7.1. If the first line of a file of shell commands starts with the characters #!, the results are unspecified.

(2) Breaks the input into tokens: words and operators. (See 3.3.)

(3) Parses the input into simple (3.9.1) and compound (3.9.4) commands.

(4) Performs various expansions (separately) on different parts of each command, resulting in a list of pathnames and fields to be treated as a command and arguments (3.6).

(5) Performs redirection (3.7) and removes redirection operators and their operands from the parameter list.

(6) Executes a function (3.9.5), built-in (3.14), executable file, or script, giving the name of the command (or, in the case of a function within a 1
script, the name of the script) as the ''zero'th'' argument and the remain- 1
ing words and fields as parameters (3.9.1.1).

(7) Optionally waits for the command to complete and collects the exit status (3.8.2).

3.0.1 Shell Command Language Rationale. *(This subclause is not a part of P1003.2)*

The System V shell was selected as the starting point for this standard. The BSD C-shell was excluded from consideration, for the following reasons:

(1) Most historically portable shell scripts assume the Version 7 ''Bourne'' shell, from which the System V shell is derived.

(2) The majority of tutorial materials on shell programming assume the System V shell.

Despite the selection of the System V shell, the developers of the standard did not limit the possibilities for a shell command language that was upward-compatible.

The only programmatic interfaces to the shell language are through the functions in 7.1 and the `sh` utility. Most implementations provide an interface to, and processing mode for, the shell that is suitable for direct user interaction. The behavior of this interactive mode is not defined by this standard; however, places where historically an interactive shell behaves differently from the behavior described here are noted.

(1) Aliases are not included in the base POSIX.2 because they duplicate functionality already available to applications with functions. In early drafts, the search order of simple command lookup was ‘‘aliases, built-ins, functions, file system,’’ and therefore an alias was necessary to create a user-defined command having the same name as a built-in. To retain this capability, the search order has changed to ‘‘special built-ins, functions, built-ins, file system,’’ and a built-in, called `command`, has been added, which disables the looking up of functions. Aliases are a part of the POSIX.2a UPE because they are widely used by human users, as differentiated from applications.

(2) All references to job control and related commands have been omitted from the base POSIX.2. POSIX.2 describes the noninteractive operation of the shell; job control is outside the scope of this standard until the UPE revision is developed. Apparently it is not widely known that traditionally, even in a job control environment, the commands executed during the execution of a shell script are not placed into separate process groups. If they were, one could not stop the execution of the shell script from the interactive shell, for example. This standard does not require or prohibit job control; it simply does not mention it.

(3) The conditional command (double bracket `[[ ]]`) was removed from an earlier draft. Objections were lodged that the real problem is misuse of the `test` command (`[`), and putting it into the shell is the wrong way to fix the problem. Instead, proper documentation and a new shell reserved word (`!`) are sufficient. Tests that require multiple `test` operations can be done at the shell level using individual invocations of the `test` command and shell logicals, rather than the error prone `−o` flag of `test`.

(4) Exportable functions were removed from an earlier draft. See the rationale in 3.9.5.1.

The construct #! is reserved for implementations wishing to provide that extension. If it were not reserved, the standard would disallow it by forcing it to be a comment. As it stands, a conforming application shall not use #! as the first line of a shell script.

3.1 Shell Definitions

The following terms are used in Section 3. Because they are specific to the shell, they do not appear in 2.2.2.

3.1.1 control operator: A token that performs a control function.

It is one of the following symbols:

`&`	`)`	`<newline>`
`&&`	`;`	`|`
`(`	`;;`	`||`

The end-of-input indicator used internally by the shell is also considered a control operator. See 3.3.

On some systems, the symbol `((` is a control operator; its use produces unspecified results. 1

3.1.2 expand: When not qualified, the act of applying all the expansions described in 3.6.

3.1.3 field: A unit of text that is the result of parameter expansion (3.6.2), arithmetic expansion (3.6.4), command substitution (3.6.3), or field splitting (3.6.5).

During command processing (see 3.9.1), the resulting fields are used as the command name and its arguments.

3.1.4 interactive shell: A processing mode of the shell that is suitable for direct user interaction.

The behavior in this mode is not defined by this standard.

NOTE: The preceding sentence is expected to change following the eventual approval of the UPE supplement.

3.1.5 name: A word consisting solely of underscores, digits, and alphabetics from the portable character set (see 2.4).

The first character of a name shall not be a digit.

3.1.6 operator: Either a control operator or a redirection operator.

3.1.7 parameter: An entity that stores values.

There are three types of parameters: variables (named parameters), positional parameters, and special parameters. Parameter expansion is accomplished by introducing a parameter with the `$` character. See 3.5.

3.1.8 positional parameter: A parameter denoted by a single digit or one or more digits in curly braces.

See 3.5.1.