-- --

IEEE P1003.2 Draft 11.2 – September 1991

Since `/dev/null` is not a regular file, no error is generated by redirecting to it in *noclobber* mode.

The case of a missing delimiter at the end of a here-document is not specified. This is considered an error in the script (one that sometimes can be difficult to diagnose), although some systems have treated end-of-file as an implicit delimiter.

Tilde expansion is not performed on a here-document because the data is treated 1
as if it were enclosed in double-quotes. 1

3.8 Exit Status and Errors

3.8.1 Consequences of Shell Errors

For a noninteractive shell, an error condition encountered by a special built-in (see 3.14) or other type of utility shall cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

	Special Built-in	**Other Utilities**
Shell language syntax error	shall exit	shall exit
Utility syntax error (option or operand error)	shall exit	shall not exit
Redirection error	shall exit	shall not exit
Variable assignment error	shall exit	shall not exit
Expansion error	shall exit	shall exit
Command not found	n/a	may exit
`dot` script not found	shall exit	n/a

An ‘‘expansion error’’ is one that occurs when the shell expansions defined in 3.6 are carried out (e.g., `${x!y}`, because `!` is not a valid operator); an implementation may treat these as syntax errors if it is able to detect them during tokenization, rather than during expansion.

If any of the errors shown as ‘‘shall (may) exit’’ occur in a subshell, the subshell shall (may) exit with a nonzero status, but the script containing the subshell shall not exit because of the error.

In all of the cases shown in the table, an interactive shell shall write a diagnostic message to standard error without exiting.

3.8.2 Exit Status for Commands

Each command has an exit status that can influence the behavior of other shell commands. The exit status of commands that are not utilities are documented in this subclause. The exit status of the standard utilities are documented in their respective clauses.

If a command is not found by the shell, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. See 3.9.1.1. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

If a command fails during word expansion or redirection, its exit status shall be greater than zero.

Internally, for purposes of deciding if a command exits with a nonzero exit status, the shell shall recognize the entire status value retrieved for the command by the equivalent of the POSIX.1 {8} *wait*() function WEXITSTATUS macro. When reporting the exit status with the special parameter ?, the shell shall report the full eight bits of exit status available. The exit status of a command that terminated because it received a signal shall be reported as greater than 128.

3.8.3 Exit Status and Errors Rationale. *(This subclause is not a part of P1003.2)*

There is a historical difference in `sh` and `ksh` noninteractive error behavior. When a command named in a script is not found, some implementations of `sh` exit immediately, but `ksh` continues with the next command. Thus, POSIX.2 says that the shell "may" exit in this case. This puts a small burden on the programmer, who will have to test for successful completion following a command if it is important that the next command not be executed if the previous was not found. If it is important for the command to have been found, it was probably also important for it to complete successfully. The test for successful completion would not need to change.

Historically, shells have returned an exit status of 128+*n*, where *n* represents the signal number. Since signal numbers are not standardized, there is no portable way to determine which signal caused the termination. Also, it is possible for a command to exit with a status in the same range of numbers that the shell would use to report that the command was terminated by a signal. Implementations are encouraged to chose exit values greater than 256 to indicate programs that terminated by a signal so that the exit status cannot be confused with an exit status generated by a normal termination.

Historical shells make the distinction between "utility not found" and "utility found but cannot execute" in their error messages. By specifying two seldomly used exit status values for these cases, 127 and 126 respectively, this gives an application the opportunity to make use of this distinction without having to parse an error message that would probably change from locale to locale. The POSIX.2 `command`, `env`, `nohup`, and `xargs` utilities also have been specified to use this convention.

When a command fails during word expansion or redirection, most historical implementations exit with a status of 1. However, there was some sentiment that this value should probably be much higher, so that an application could distinguish this case from the more normal exit status values. Thus, the language "greater than zero" was selected to allow either method to be implemented.

3.9 Shell Commands

This clause describes the basic structure of shell commands. The following command descriptions each describe a format of the command that is only used to aid the reader in recognizing the command type, and does not formally represent the syntax. Each description discusses the semantics of the command; for a formal description of the command language, consult the grammar in 3.10.

A *command* is one of the following:

— *simple command* (see 3.9.1)

— *pipeline* (see 3.9.2)

— *list* or *compound-list* (see 3.9.3)

— *compound command* (see 3.9.4)

— *function definition* (see 3.9.5).

Unless otherwise stated, the exit status of a command is that of the last simple command executed by the command. There is no limit on the size of any shell command other than that imposed by the underlying system (memory constraints, {ARG_MAX}, etc.).

3.9.0.1 Shell Commands Rationale. *(This subclause is not a part of P1003.2)*

A description of an ‘‘empty command’’ was removed from an earlier draft because it is only relevant in the cases of `sh -c ""`, `system("")`, or an empty shell-script file (such as the implementation of `true` on some historical systems). Since it is no longer mentioned in POSIX.2, it falls into the silently unspecified category of behavior where implementations can continue to operate as they have historically, but conforming applications will not construct empty commands. (However, note that `sh` does explicitly state an exit status for an empty string or file.) In an interactive session or a script with other commands, extra `<newline>`s or semicolons, such as

```
$ false
$
$ echo $?
1
```

would not qualify as the empty command described here because they would be consumed by other parts of the grammar.

3.9.1 Simple Commands

A *simple command* is a sequence of optional variable assignments and redirections, in any sequence, optionally followed by words and redirections, terminated by a control operator.

When a given simple command is required to be executed (i.e., when any conditional construct such as an AND-OR list or a `case` statement has not bypassed the