NEK Tech Shell Development

Functional Requirements Specification

Version: 1.0
Date: 04.07.13

DOCUMENT APPROVAL

Name		Position Title		Signature		Date
Pankaj Saraf		Technical Head – Product Development				
Anil Dahiya		Network Development Engineer – Network Protocol Expert				
Ashish Pandey		OS Development Engineer – Filesystem Expert				

Version History

Version #	Implemented By	Revision Date	Approved By	Approval Date	Reason
1.0	*Pallavi Gadge*	*05/07/2013*	*Pankaj Saraf*	*08/07/2013*	Approved

1. PURPOSE OF THIS DOCUMENT

This Document is targeted to identify the initial requirements of Unix Shell development. The Product/Software is expected to be a part of learning exercise and understanding the operating system concepts. There would be some further enhancements in the product will be planned by company's technical team.

The Functional Requirements Specification will:

- Development of Linux Shell for Educational purpose and developing understanding to Operating Systems.
- Linux Shell Development will impart all the functionalities of Traditional UNIX Shell.
- Further Enhancements will be expected, as covering all the functionalities in a short time, will not be possible.

2. INTRODUCTION

A command-line interface (CLI) is a means of interacting with a computer program where the user (or client) issues commands to the program in the form of successive lines of text (command lines).

The CLI was the primary means of interaction with most popular operating systems in the 1970s and 1980s, including MS-DOS, CP/M, Unix, and Apple DOS. The interface is usually implemented with a command line shell, which is a program that accepts commands as text input and converts commands to appropriate operating system functions.

Command-line interfaces to computer operating systems are less widely used by casual computer users, who favor graphical user interfaces (GUI). Command-line interfaces are often preferred by more advanced computer users, as they often provide a more concise and powerful means to control a program or operating system.

The general pattern of an OS command line interface (CLI) is:

prompt command param1 param2 param3 ... paramN

Prompt - generated by the program to provide context for the client.
Command - provided by the client. Commands are usually one of three classes:
Internal - recognized and processed by the command line interpreter itself and not dependent upon any external executable file.
Included - A separate executable file generally considered part of the operating environment and always included with the OS.
External - External executable files not part of the basic OS, but added by other parties for specific purposes

and applications.
param1 ...paramN - Optional parameters provided by the client. The format and meaning of the parameters depends upon the command issued. In the case of Included or External commands, the values of the parameters are delivered to the process (specified by the Command) as it is launched by the OS. Parameters may be either Arguments or Options. In this example, the delimiters between command line elements are whitespace characters and the end-of-line delimiter is the newline delimiter. This is a widely used (but not universal) convention for command-line interfaces.

A CLI can generally be considered as consisting of syntax and semantics. The syntax is the grammar that all commands must follow. In the case of operating systems (OS), MS-DOS and Unix each define their own set of rules that all commands must follow. In the case of embedded systems, each vendor, such as Nortel, Juniper Networks or Cisco Systems, defines their own proprietary set of rules that all commands within their CLI conform to. These rules also dictate how a user navigates through the system of commands. The semantics define what sort of operations are possible, on what sort of data these operations can be performed, and how the grammar represents these operations and data—the symbolic meaning in the syntax.

Two different CLIs may agree on either syntax or semantics, but it is only when they agree on both that they can be considered sufficiently similar to allow users to use both CLIs without needing to learn anything, as well as to enable re-use of scripts.

A simple CLI will display a prompt, accept a "command line" typed by the user terminated by the Enter key, then execute the specified command and provide textual display of results or error messages. Advanced CLIs will validate, interpret and parameter-expand the command line before executing the specified command, and optionally capture or redirect its output.

The *shell* is a command programming language that provides an interface to the UNIX operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while,ifthenelse,case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.
The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through `pipes' can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

Unix Command Line Interface executes commands by forking (Creating new process) and Execing (initiate new program within a program):

The fork() system call will spawn a new child process, which is an identical process to the parent except that

has a new system process ID. The process is copied in memory from the parent and a new process structure is assigned by the kernel. The return value of the function is which discriminates the two threads of execution. A zero is returned by the fork function in the child's process.

The environment, resource limits, umask, controlling terminal, current working directory, root directory, signal masks and other process resources are also duplicated from the parent in the forked child process.

The exec() family of functions will initiate a program from within a program. They are also various front-end functions to execve(). It initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. /bin/ls) and arguments are passed to the function. The functions return an integer error code. (0=Ok/-1=Fail).

3. REFERENCE DOCUMENTS

Document	Link
Command Line Interfaces (CLI)	*http://en.wikipedia.org/wiki/Command-line_interface*
Anatomy of Shell	*http://en.wikipedia.org/wiki/Command-line_interface#Anatomy_of_a_shell_CLI*
Introduction to Processes and Unix Shell	*http://www.ucblueash.edu/thomas/Intro_Unix_Text/Process.html*
Intro to Processes, Fork & Exec	*http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html*
fork() System call	*http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#fork*
exec() System call	*http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#exec*
wait() System call	*http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#wait*
getpid() System call	*http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#getpid*
getgrp() System call	*http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#getpgrp*

4. SCOPE OF THE FUNCTIONAL REQUIREMENTS SPECIFICATION

"NEK Tech Shell Development" for Linux will be submitted to Open Source by Next year, with all the Traditional UNIXShell Capabilities.

Following are the Functionalities will be covered in this version of Development:

In Scope	Out of Scope
Taking command and arguments as input, storing and calling (Current FRS)	Logical operators
New Process creation and executing new command	Pipes and redirections in shell
Change directory handling by the shell	Special Character of Shell Programming

5. FUNCTIONAL REQUIREMENTS

This document will cover the functional requirement of Shell's basic architecture and features provided by the "NEK Tech Shell":

- *"NEK Tech Shell" takes a user input from "Command Line Interface".*
- *It parses, handles modifies and arranges commands/arguments according to "Linux System call interface" and provide to Kernel to get the services from Linux Kernel.*
- *It also handles some specific conditions, for examples change directory (cd is not an another binary to exec, it has to be handled with the internal system call interface.)*
- *This document will cover run-cmd(), which is responsible for creating new process and executing binaries.*
- *This run_cmd() is responsible for creating new processes and exec() it to new binaries. Process creation is done using fork() and exec() system call to replace the text of the running process.*