KSH

KULICH SHELL

REFERENCE MANUAL

PUBLIC INFORMATION

This document contains technical specifications
for the Kulich Shell (KSH) system.

Assembly Language Implementation
x86-64 Architecture

Document Version 1.0

January 1976

CONTENTS

1 INTRODUCTION

1.1 Purpose

The Kulich Shell (KSH) is a command-line interpreter written entirely in x86-64 assembly language using the Netwide Assembler (NASM) syntax. This manual provides comprehensive documentation of the internal structure, register usage patterns, system call interfaces, and operational characteristics of KSH.

1.2 System Requirements

KSH requires the following environment:

- x86-64 compatible processor
- Linux operating system (kernel 2.6 or later)
- NASM assembler (version 2.0 or later)
- GNU ld linker
- Minimum 128 bytes of buffer space

1.3 Document Conventions

Throughout this manual, the following conventions are observed:

- Register names appear in lowercase: rax, rbx, rcx, etc.
- Hexadecimal values are prefixed with 0x
- System call numbers are shown in decimal
- Memory addresses are shown in hexadecimal
- Assembly mnemonics appear in monospace typeface

2 ARCHITECTURE OVERVIEW

2.1 Module Organization

The KSH system is organized into three primary modules:

main.asm Main control loop and command parsing logic

io.asm Input/output operations and buffer management

utils.asm Utility functions for path manipulation

2.2 Memory Layout

Segment	Size	Purpose
.data	Variable	Static strings and constants
.bss	512 bytes	Uninitialized data buffers
.text	Variable	Executable code

2.3 Buffer Allocations

Buffer	Size	Description
buf	128 bytes	Primary input buffer
bff	128 bytes	Secondary buffer (utils)
pth	256 bytes	Path construction buffer
argv	64 bytes	Argument vector (8 pointers)

3 SYSTEM CALL INTERFACE

3.1 System Call Mechanism

KSH utilizes the Linux x86-64 system call interface. System calls are invoked using the `syscall` instruction with parameters passed in specific registers according to the System V AMD64 ABI.

3.2 System Call Register Convention

Register	Purpose
rax	System call number (input), return value (output)
rdi	First argument
rsi	Second argument
rdx	Third argument
r10	Fourth argument
r8	Fifth argument
r9	Sixth argument

3.3 System Calls Employed

Number	Name	Description
0	sys_read	Read from file descriptor
1	sys_write	Write to file descriptor
57	sys_fork	Create child process
59	sys_execve	Execute program

Number	Name	Description
60	sys_exit	Terminate process
61	sys_wait4	Wait for process state change

4 REGISTER USAGE ANALYSIS

4.1 General Purpose Registers

This section documents the specific usage of x86-64 general purpose registers within KSH functions.

4.1.1 RAX Register

Primary Functions:

- System call number specification
- System call return value
- Temporary calculations
- Loop counters (specific contexts)

Usage Examples:

```
mov rax, 1      ; sys_write
mov rax, 0      ; sys_read
mov rax, 60     ; sys_exit
```

4.1.2 RBX Register

Primary Functions:

- Argument vector index counter
- Temporary pointer storage
- General purpose calculations

Critical Note: RBX must be preserved across function calls as per ABI.
Usage Example:

```
push rbx        ; Save
mov rbx, 1      ; Index = 1
; ... operations ...
pop rbx         ; Restore
```

4.1.3 RCX Register

Primary Functions:

- Primary loop counter
- Buffer offset calculator
- String manipulation index

Usage Example:

```
xor rcx, rcx    ; Initialize counter
.loop:
    ; operations
    inc rcx
    cmp rcx, 128
    jne .loop
```

4.1.4 RDX Register

Primary Functions:

- Buffer length specification
- Third system call parameter
- Temporary storage

4.1.5 RSI Register

Primary Functions:

- Source buffer pointer
- Second system call parameter
- String operation source

4.1.6 RDI Register

Primary Functions:

- Destination buffer pointer
- First system call parameter
- File descriptor specification

4.1.7 R10 Register

Primary Functions:

- Fourth parameter for wait4 syscall
- Temporary storage

4.2 Register Preservation

The following registers must be preserved (caller-saved):

- RBX, RBP, R12-R15

The following registers are scratch (callee-saved):

- RAX, RCX, RDX, RSI, RDI, R8-R11

5 I/O SUBSYSTEM

5.1 Print Function

Function Signature:

`void print(char* buffer, size_t length)`

Register Usage:

Register	I/O	Purpose
rsi	Input	Pointer to output buffer
rdx	Input	Length of data to write
rax	Modified	System call number (1), return value
rdi	Modified	File descriptor (1 = stdout)

Implementation:

```
print:
    mov rax, 1      ; sys_write
    mov rdi, 1      ; stdout
    syscall
    ret
```

5.2 Read Function

Function Signature:

`size_t read(void)`

Register Usage:

Register	I/O	Purpose
rax	Output	Number of bytes read
rcx	Modified	Loop counter for buffer clearing
rsi	Modified	Buffer pointer (buf)
rdx	Modified	Maximum bytes to read (128)
rdi	Modified	File descriptor (0 = stdin)

Algorithm:

1. Clear input buffer (128 bytes)
2. Execute sys_read system call
3. Remove trailing newline character
4. Return number of bytes read

Implementation Details:

The read function performs a critical operation: it removes the trailing newline (ASCII 10) that results from the user pressing Enter. This is accomplished by:

1. Testing if any bytes were read (test rax, rax)
2. Decrementing the count to point to last character
3. Comparing byte at buf+rax with 10
4. Setting byte to 0 if newline detected

6 COMMAND PROCESSING

6.1 Main Loop Structure

The main loop implements the following algorithm:

1. Display prompt (" > ")
2. Read user input
3. Parse and validate command
4. Execute built-in or external command
5. Return to step 1

6.2 Built-in Commands

6.2.1 exit

Terminates the shell process.

Syntax: `exit`

Implementation: Executes sys_exit (60) with return code 0.

Byte Comparison Sequence:

```
cmp byte [buf], 'e'
jne .check_clear
cmp byte [buf+1], 'x'
jne .check_clear
cmp byte [buf+2], 'i'
jne .check_clear
cmp byte [buf+3], 't'
jne .check_clear
cmp byte [buf+4], 0
je _exit
```

6.2.2 clear

Clears the terminal screen using ANSI escape sequences.

Syntax: `clear`

ANSI Sequence: ESC[2J ESC[H

Byte Representation:

```
27, '[', '2', 'J', 27, '[', 'H'
```

6.2.3 info

Displays shell information with color formatting.

Syntax: `info`

ANSI Color Sequence:

```
ESC[1;38;5;2;49m ... ESC[0;39;49m
```

Color Breakdown:

- 1 = Bold
- 38;5;2 = Foreground green (256-color mode)
- 49 = Default background
- 0;39;49 = Reset to defaults

6.3 External Command Execution

6.3.1 Command Parsing Algorithm

The parser implements a state machine to convert space-delimited input into an argv array:

1. Set argv[0] to start of buffer
2. Scan for space characters
3. Replace spaces with null bytes
4. Skip consecutive spaces
5. Set argv[n] to next non-space character
6. Terminate with NULL pointer

Register Usage During Parsing:

Register	Purpose
rdi	Pointer to argv array
rsi	Pointer to current position in input buffer
rcx	Offset counter within buffer
rbx	Index of current argument (1, 2, 3, ...)

State Machine Diagram:

```
START -> SCAN_WORD -> FOUND_SPACE -> SKIP_SPACES -> SCAN_WORD
                    -> FOUND_NULL -> FINISH
```

6.3.2 Process Creation

External commands are executed using the fork-exec pattern:

1. Fork: sys_fork (57) creates child process
2. Parent: sys_wait4 (61) waits for child completion
3. Child: sys_execve (59) replaces process image

Fork Implementation:

```
mov rax, 57     ; sys_fork
syscall
test rax, rax   ; Check return value
jz .child       ; 0 = child process
jl main_loop    ; < 0 = error
```

Parent Wait Implementation:

```
mov rdi, rax    ; PID of child
mov rax, 61     ; sys_wait4
xor rsi, rsi    ; status = NULL
xor rdx, rdx    ; options = 0
xor r10, r10    ; rusage = NULL
syscall
```

Child Exec Implementation:

```
.child:
    mov rdi, [argv]     ; Program path
    lea rsi, [argv]     ; Argument vector
    xor rdx, rdx        ; Environment = NULL
    mov rax, 59         ; sys_execve
    syscall
    ; If exec fails, exit with status 1
    mov rax, 60
    mov rdi, 1
    syscall
```

7 UTILITY FUNCTIONS

7.1 make_path Function

Purpose: Concatenates "/usr/" prefix with a relative path.
Function Signature:

```
void make_path(char* buf, char* path)
```

Parameters:

- rdi: Pointer to source path
- rsi: Pointer to destination buffer

Algorithm:

1. Load "/usr/" string address
2. Copy prefix to destination using lodsb/stosb
3. Decrement destination to overwrite null terminator
4. Copy source path to destination
5. Return with concatenated string

Register Usage:

Register	Purpose
rbx	Saved on stack, used for "/usr/" pointer
rsi	Source pointer for lodsb
rdi	Destination pointer for stosb
al	Byte transfer register

String Instructions:

- `lodsb`: Load byte from [rsi] into al, increment rsi
- `stosb`: Store al into [rdi], increment rdi

8 DATA STRUCTURES

8.1 Static String Table

Label	Length	Content
msg	16 bytes	Welcome message
shs	3 bytes	Shell prompt " > "
kms	43 bytes	Info message with ANSI codes
clr	7 bytes	Clear screen ANSI sequence
nln	1 byte	Newline character
usr	6 bytes	"/usr/" path prefix

8.2 BSS Segment Layout

```
+-------------------+
| argv (64 bytes)   | <- 8 quadword pointers
+-------------------+
| buf (128 bytes)   | <- Main input buffer
+-------------------+
| bff (128 bytes)   | <- Utility buffer
+-------------------+
| pth (256 bytes)   | <- Path buffer
+-------------------+
```

9 ERROR HANDLING

9.1 System Call Failures

System calls return negative values on error. KSH implements minimal error checking:

- Fork failure: Returns to main loop

- Read failure (EOF): Continues main loop
- Exec failure: Child exits with status 1

9.2 Invalid Commands

If a command is not recognized as a built-in and cannot be executed as an external program, the execve system call will fail and the child process terminates with exit code 1.

10 BUILD SYSTEM

10.1 Makefile Structure

The build system uses GNU Make with the following configuration:

Variables:

- ASM = nasm
- LD = ld
- ASMFLAGS = -f elf64 -g
- LDFLAGS = (empty)
- TARGET = ksh

Build Process:

1. Assemble .asm files to .o object files
2. Link object files into executable

Targets:

- all: Build executable
- run: Build and execute
- clean: Remove build artifacts

10.2 Compilation Commands

Assembly:

```
nasm -f elf64 -g src/io.asm -o build/io.o
```

Linking:

```
ld build/io.o build/main.o build/utils.o -o ksh
```

11 COMPLETE SOURCE CODE LISTINGS

11.1 io.inc - Interface Definitions

```
extern read
extern print
extern buf
extern make_path
```

11.2 io.asm - Input/Output Module

```
section .bss
global buf
buf: resb 128

section .text
global print
global read

print:
    mov rax, 1
    mov rdi, 1
    syscall
    ret

read:
    xor rcx, rcx
.clear_buf:
    cmp rcx, 128
    je .done_clear
    mov byte [buf+rcx], 0
    inc rcx
    jmp .clear_buf
.done_clear:
    mov rax, 0
    mov rdi, 0
    mov rsi, buf
    mov rdx, 128
    syscall
    ; remove \n
    test rax, rax
    jz .done
    dec rax
    cmp byte [buf+rax], 10
    jne .done
    mov byte [buf+rax], 0
```

```
.done:
    ret
```

11.3 main.asm - Main Control Module

```
%include "include/io.inc"

global _start

section .data
msg: db "welcome to ksh!", 10
msl: equ $ - msg
shs: db " > "
shl: equ $ - shs
kms: db 27, "[1;38;5;2;49m", "ksh stands for kulich shell", 27,
   "[0;39;49m", 10
mkl: equ $ - kms
clr: db 27, '[', '2', 'J', 27, '[', 'H'
cll: equ $ - clr
nln: db 10

section .bss
argv resq 8

section .text
_start:
    mov rsi, msg
    mov rdx, msl
    call print

main_loop:
    mov rsi, shs
    mov rdx, shl
    call print
    call read
    cmp rax, 0
    jle main_loop

    cmp byte [buf], 'e'
    jne .check_clear
    cmp byte [buf+1], 'x'
    jne .check_clear
    cmp byte [buf+2], 'i'
    jne .check_clear
    cmp byte [buf+3], 't'
    jne .check_clear
    cmp byte [buf+4], 0
    je _exit

.check_clear:
    cmp byte [buf], 'c'
    jne .check_info
    cmp byte [buf+1], 'l'
    jne .check_info
    cmp byte [buf+2], 'e'
    jne .check_info
    cmp byte [buf+3], 'a'
    jne .check_info
    cmp byte [buf+4], 'r'
    jne .check_info
    cmp byte [buf+5], 0
    je .do_clear

.check_info:
    cmp byte [buf], 'i'
    jne .parse
    cmp byte [buf+1], 'n'
    jne .parse
    cmp byte [buf+2], 'f'
    jne .parse
    cmp byte [buf+3], 'o'
    jne .parse
    cmp byte [buf+4], 0
    je .do_info

.parse:
    lea rdi, [argv]
    lea rsi, [buf]
    mov [rdi], rsi
    xor rcx, rcx
    mov rbx, 1

.build:
    cmp byte [buf + rcx], 0
    je .finish
    cmp byte [buf + rcx], ' '
    jne .next
    mov byte [buf + rcx], 0
    inc rcx

.skip_spaces:
    cmp byte [buf + rcx], ' '
    jne .set_arg
    inc rcx
    jmp .skip_spaces

.set_arg:
    cmp byte [buf + rcx], 0
    je .finish
    lea rsi, [buf + rcx]
    mov [argv + rbx*8], rsi
    inc rbx
    jmp .build

.next:
    inc rcx
    jmp .build

.finish:
    mov qword [argv + rbx*8], 0
    mov rax, 57
    syscall
    test rax, rax
    jz .child
    jl main_loop
    mov rdi, rax
    mov rax, 61
    xor rsi, rsi
    xor rdx, rdx
    xor r10, r10
    syscall
    jmp main_loop

.child:
    mov rdi, [argv]
    lea rsi, [argv]
    xor rdx, rdx
    mov rax, 59
    syscall
    mov rax, 60
    mov rdi, 1
    syscall

.do_clear:
    mov rsi, clr
    mov rdx, cll
    call print
    jmp main_loop

.do_info:
    mov rsi, kms
    mov rdx, mkl
    call print
    jmp main_loop

_exit:
    mov rax, 60
    xor rdi, rdi
    syscall
```

11.4 utils.asm - Utility Functions

```
section .data
usr db "/usr/", 0

section .bss
global bff
bff: resb 128

global pth
pth: resb 256

section .text
global make_path
global _start

; ------------------------------------------------
; void make_path(char *buf, char *path)
; rdi = pointer path
; rsi = pointer buf
; ------------------------------------------------
make_path:
    push rbx
    lea rbx, [usr]

.copy_usr:
    lodsb
    stosb
    test al, al
    jne .copy_usr
    dec rdi     ; remove extra null

.copy_bff:
    lodsb
    stosb
    test al, al
    jne .copy_bff
    pop rbx
    ret
```

12 PERFORMANCE CHARACTERISTICS

12.1 Time Complexity

Operation	Complexity	Notes
Buffer clear	O(n)	Linear scan, n=128
Command parse	O(n)	Single pass, n=input length
Built-in check	O(1)	Fixed comparisons
Path concat	O(n+m)	n=prefix, m=path length

12.2 Space Complexity

Total static allocation: 512 bytes (BSS) + 100 bytes (data)

12.3 System Call Overhead

Each command execution incurs:

- 1 sys_read (input)
- 1 sys_write (prompt)
- 1 sys_fork (external commands)
- 1 sys_execve (external commands)
- 1 sys_wait4 (external commands)

13 LIMITATIONS AND KNOWN ISSUES

13.1 Buffer Overflow Vulnerability

The input buffer is fixed at 128 bytes with no bounds checking. Input exceeding this limit will cause buffer overflow.

13.2 No Environment Variables

The execve call passes NULL for the environment pointer, so child processes have no environment.

13.3 No PATH Resolution

Commands must be specified with full path or relative to current directory. The shell does not search standard PATH locations.

13.4 No Redirection or Pipes

Standard I/O redirection and pipeline operations are not implemented.

13.5 No Job Control

Background processes, job suspension, and process groups are not supported.

14 FUTURE ENHANCEMENTS

14.1 Proposed Features

- PATH environment variable support
- Command history buffer
- Tab completion
- Signal handling (SIGINT, SIGTSTP)
- Wildcard expansion
- I/O redirection (>, <, »)
- Pipeline support (|)
- Command substitution

14.2 make_path Integration

The make_path function is currently unused in the main codebase. Future versions should integrate this for automatic /usr/ prefix resolution.

15 APPENDIX A: ASCII REFERENCE

15.1 Control Characters

Decimal	**Hex**	**Character**
0	0x00	NUL (null terminator)
10	0x0A	LF (line feed)
27	0x1B	ESC (escape)
32	0x20	SPACE

15.2 Command Characters

Character	Decimal	Hex
'a'	97	0x61
'c'	99	0x63
'e'	101	0x65
'f'	102	0x66
'i'	105	0x69
'l'	108	0x6C
'n'	110	0x6E
'o'	111	0x6F
'r'	114	0x72
't'	116	0x74
'x'	120	0x78

16 APPENDIX B: X86-64 REGISTER REFERENCE

16.1 General Purpose Registers (64-bit)

Register	ABI Status	Description
RAX	Volatile	Accumulator, syscall number/return
RBX	Non-volatile	Base register (must preserve)
RCX	Volatile	Counter register
RDX	Volatile	Data register
RSI	Volatile	Source index
RDI	Volatile	Destination index
RBP	Non-volatile	Base pointer (must preserve)
RSP	Non-volatile	Stack pointer (must preserve)
R8-R11	Volatile	Additional general purpose
R12-R15	Non-volatile	Additional (must preserve)

16.2 Special Purpose Registers

Register	Description
RIP	Instruction pointer (program counter)
RFLAGS	Status flags (ZF, SF, CF, OF, etc.)

17 APPENDIX C: ANSI ESCAPE SEQUENCES

17.1 Color Codes

Sequence	**Effect**
ESC[0m	Reset all attributes
ESC[1m	Bold / increased intensity
ESC[38;5;Nm	Set foreground color (N = 0-255)
ESC[39m	Default foreground color
ESC[49m	Default background color

17.2 Screen Control

Sequence	**Effect**
ESC[2J	Clear entire screen
ESC[H	Move cursor to home (0,0)
ESC[K	Clear line from cursor to end

18 INDEX

END OF DOCUMENT

KULICH SHELL REFERENCE MANUAL
VERSION 1.0 - JANUARY 2026