Bare Metal Functions Documentation

This document provides a detailed technical explanation of the core utility functions implemented in `kernel/string.c` and `kernel/common.c`. These functions form the low-level foundation for basic string manipulation, memory operations, and formatted output within the Novix kernel.

File: `kernel/string.c`

Function: `int strcmp(const char *s1, const char *s2)`

Purpose:

Compares two null-terminated strings lexicographically.

Parameters:

- `s1` — Pointer to the first string.
- `s2` — Pointer to the second string.

Return Value:

- `0` if the strings are equal.
- A positive value if `s1` is greater than `s2`.
- A negative value if `s1` is less than `s2`.

Description:

Iterates through both strings character by character. The loop continues until a mismatch is found or the null terminator (`'\0'`) is reached. Returns the numerical difference between the first mismatched characters.

Function: `int strncmp(const char *s1, const char *s2, unsigned int n)`

Purpose:

Compares up to `n` characters of two strings.

Parameters:

- `s1`, `s2` — Strings to compare.
- `n` — Maximum number of characters to compare.

Return Value:

Same as `strcmp`.

Description:

Performs a bounded comparison of two strings. Stops when either `n` characters are compared, or a null terminator is found. Used for safe comparisons in limited buffers.

Function: `size_t strlen(const char *str)`

Purpose:

Returns the length of a string (excluding the null terminator).

Parameters:

- `str` — Pointer to the null-terminated string.

Return Value:

Number of characters before the null terminator.

Description:

Counts each character until `'\0'` is found.

Function: `size_t strnlen(const char *s, size_t maxlen)`

Purpose:

Returns the length of a string, up to a maximum limit.

Parameters:

- `s` — String pointer.
- `maxlen` — Maximum number of bytes to scan.

Return Value:

Length of the string or `maxlen`, whichever is smaller.

Function: `char *strcpy(char *dst, const char *src)`

Purpose:

Copies a string from `src` to `dst`.

Parameters:

- `dst` — Destination buffer.
- `src` — Source string.

Return Value:
Pointer to `dst`.

Description:

Copies characters one by one until `'\0'`. The destination string is null-terminated.

Function: `char *strncpy(char *dest, const char *src, size_t n)`

Purpose:

Copies up to `n` characters from `src` to `dest`.

Parameters:

- `dest`, `src` — Pointers to destination and source buffers.
- `n` — Maximum number of bytes to copy.

Return Value:
Pointer to `dest`.

Description:

If `src` is shorter than `n`, the remainder of `dest` is filled with `'\0'`.
Prevents buffer overflows by limiting copy size.

Function: `char *strstr(const char *haystack, const char *needle)`

Purpose:

Finds the first occurrence of substring `needle` in `haystack`.

Return Value:

Pointer to the first match or `NULL` if not found.

Description:

Sequentially checks all starting positions in `haystack` for a match with `needle`.

Function: `char *strcat(char *dest, const char *src)`

Purpose:

Appends `src` string to the end of `dest`.

Return Value:

Pointer to `dest`.

Description:

Finds the end of `dest` and copies `src` to that position. Ensures final null termination.

Function: `char *strchr(const char *s, int c)`

Purpose:

Finds the first occurrence of character `c` in string `s`.

Return Value:

Pointer to character in `s`, or `NULL` if not found.

Function: `char *strtok(char *str, const char *delim)`

Purpose:

Splits a string into tokens using delimiters.

Parameters:

- `str` — Input string (or `NULL` to continue previous tokenization).
- `delim` — Set of delimiter characters.

Return Value:

Pointer to next token, or `NULL` when no more tokens.

Description:

Uses a static pointer `saved` to remember the next starting position.
Replaces delimiters with `'\0'` to isolate tokens.

Function: `void *memcpy(void *dest, const void *src, size_t n)`

Purpose:

Copies `n` bytes from `src` to `dest`.

Return Value:

Pointer to destination buffer.

Description:

Simple byte-by-byte copy, safe for non-overlapping memory regions.

Function: `void *memset(void *dest, int val, size_t n)`

Purpose:

Fills a memory block with a given byte value.

Parameters:

- `dest` — Pointer to memory.
- `val` — Value to set.
- `n` — Number of bytes to write.

Return Value:

Pointer to `dest`.

File: `kernel/common.c`

Function: `void itoa_unsigned(unsigned int value, char *str, int base)`

Purpose:

Converts an unsigned integer to a string representation in the given base (2–16).

Parameters:

- `value` — Number to convert.
- `str` — Output buffer.
- `base` — Numerical base (binary, decimal, hex, etc.).

Return Value:

None.

Description:

Repeatedly divides the number by the base and collects digits in reverse order using a temporary buffer. Then reverses them for correct output.

Function: `void itoa_signed(int value, char *str, int base)`

Purpose:
Converts a signed integer to string, handling negative values.

Behavior:

If the value is negative and base = 10, a minus sign is prefixed.

Function: `int kvnprintf(char *buf, size_t size, const char *fmt, va_list args)`

Purpose:

Implements a minimal kernel-safe version of `vsnprintf`.

Parameters:

- `buf` — Destination buffer.
- `size` — Maximum buffer length.
- `fmt` — Format string.
- `args` — Variable argument list.

Return Value:

Number of characters written.

Description:

Processes format specifiers like `%s`, `%d`, `%x`, `%p`, and `%c`.
Performs no dynamic allocation and ensures buffer safety by checking `i < size - 1`.

Function: `int ksnprintf(char *buf, size_t size, const char *fmt, ...)`

Purpose:

Wrapper for `kvnprintf`, providing the standard variadic function interface.

Description:

Initializes a `va_list`, forwards it to `kvnprintf`, and returns the result.

Summary

These functions provide: - Core **C library** equivalents for the Novix kernel. - Deterministic behavior and **no dynamic memory usage**. - Strict buffer safety for low-level environments.

Tip:

These utilities form the foundation for higher-level kernel modules like UART logging, ELF loading, and system calls.
Understanding them ensures you can safely manipulate data structures without relying on standard libraries.