Apple Color Emoji

Posix-Nexus C Core

Canine-Table

April 22, 2025

Contents

I Definitions

I Definitions

- **Constants**: Includes mathematical, physical, astronomical, and engineering constants.
- **Basic Operations**: Contains macros for arithmetic, bitwise operations, and geometry.
- **Variable Types**: Definitions of pointers, floating-point types, and integers.
- **Endian Macros**: Definitions for byte order detection and endian swapping.
- **Architecture and Alignment**: Macros and types for architecture detection and memory alignment.
- **CPU Features**: Detection of instruction sets, branch prediction, and prefetching capabilities.

I Constants

I Constants

This section defines constants for various domains, including mathematical, physical, astronomical, and engineering applications.

- **Mathematical Constants**: Includes values like Pi, Tau, and Euler's number.
- **Physical Constants**: Includes speed of light, Planck's constant, and more.
- **Astronomical Constants**: Includes values like AU, parsec, and solar mass.
- **Engineering Constants**: Includes data size units and floating-point precision limits.

I Mathematical Constants

I Mathematical Constants

Mathematical constants include commonly used values for calculations, such as the golden ratio, Pi, and Euler's number.

Mathematical Constants

```c
/* Mathematical Constants */
#define NX_GOLDEN_RATIO 1.618033988749895     /* Golden Ratio (phi) */
#define NX_PI 3.141592653589793             /* Value of Pi */
#define NX_TAU 6.283185307179586         /* Tau (2 * Pi) */
#define NX_E 2.718281828459045              /* Euler's number */
#define NX_SQRT2 1.414213562373095          /* Square root of 2 */
#define NX_LN2 0.6931471805599453          /* Natural logarithm of 2 */
#define NX_LN10 2.302585092994046          /* Natural logarithm of 10 */
```

/* Mathematical Constants */ define NX_GOLDEN_RATIO 1.618033988749895 /* Golden Ratio (phi) */ define NX_PI 3.141592653589793 /* Value of Pi */ define NX_TAU 6.283185307179586 /* Tau (2 * Pi) */ define NX_E 2.718281828459045 /* Euler's number */ define NX_SQRT2 1.414213562373095 /* Square root of 2 */ define NX_LN2 0.6931471805599453 /* Natural logarithm of 2 */ define NX_LN10 2.302585092994046 /* Natural logarithm of 10 */

I Physical Constants

I Physical Constants

Physical constants include fundamental values such as the speed of light, Planck's constant, and Boltzmann constant.

Physical Constants

```c
/* Physical Constants */
#define NX_LIGHT_SPEED 299792458        /* Speed of light in vacuum (m/s) */
#define NX_GRAVITY 9.80665            /* Standard gravity (m/s^2) */
#define NX_PLANCK 6.62607015e-34        /* Planck's constant (Js) */
#define NX_BOLTZMANN 1.380649e-23        /* Boltzmann constant (J/K) */
#define NX_AVOGADRO 6.02214076e23        /* Avogadro's number (1/mol) */
#define NX_GAS_CONSTANT 8.314462618        /* Ideal gas constant (J/(mol· K)) */
#define NX_ELECTRON_MASS 9.10938356e-31        /* Electron mass (kg) */
#define NX_PROTON_MASS 1.67262192369e-27    /* Proton mass (kg) */
#define NX_ELEM_CHARGE 1.602176634e-19        /* Elementary charge (C) */
#define NX_PERMITTIVITY 8.854187817e-12        /* Vacuum permittivity (F/m) */
#define NX_PERMEABILITY 1.2566370614e-6        /* Vacuum permeability (H/m) */
```

/* Physical Constants */ define NX_LIGHT_SPEED 299792458 /* Speed of light in vacuum (m/s) */ define NX_GRAVITY 9.80665 /* Standard gravity (m/s^2) $*/defineNX\_PLANCK6.62607015e-34/*Planck'sconstant(Js)*/defineNX\_BOLTZMANN1.380649e-23/*Boltzmannconstant(J/K)*/defineNX\_AVOGADRO6.02214076e23/*Avogadro'snumber(1/mol)*/defineNX\_GAS\_CONSTANT8.314462618/*Idealgasconstant(J/(molK))*/defineNX\_ELECTRON\_MASS9.10938356e-31/*Electronmass(kg)*/defineNX\_PROTON\_MASS1.67262192369e-27/*Protonmass(kg)*/defineNX\_ELEM\_CHARGE1.602176634e-19/*Elementarycharge(C)*/defineNX\_PERMITTIVITY8.854187817e-12/*Vacuumpermittivity(F/m)*/defineNX\_PERMEABILITY1.2566370614e-6/*Vacuumpermeability(H/m)*/$

I Astronomical Constants

I Astronomical Constants

Astronomical constants include values used for celestial calculations, such as the astronomical unit (AU), parsec, and solar mass.

Astronomical Constants

```c
/* Astronomy Constants */
#define NX_AU 149597870700          /* Astronomical Unit (meters) */
#define NX_PARSEC 3.085677581e16     /* Parsec (meters) */
#define NX_SOLAR_MASS 1.989e30        /* Mass of the Sun (kg) */
#define NX_EARTH_MASS 5.972e24        /* Mass of the Earth (kg) */
#define NX_LUNAR_MASS 7.342e22        /* Mass of the Moon (kg) */
#define NX_EARTH_RADIUS 6371000        /* Earth's mean radius (meters) */
#define NX_EARTH_ORBITAL_PERIOD 365.25     /* Earth's orbital period (days) */
#define NX_MOON_DISTANCE 384400000    /* Average Earth-Moon distance (meters)
↪*/
```

/* Astronomy Constants */ define NX_AU 149597870700 /* Astronomical Unit (meters) */ define NX_PARSEC 3.085677581e16 /* Parsec (meters) */ define NX_SOLAR_MASS 1.989e30 /* Mass of the Sun (kg) */ define NX_EARTH_MASS 5.972e24 /* Mass of the Earth (kg) */ define NX_LUNAR_MASS 7.342e22 /* Mass of the Moon (kg) */ define NX_EARTH_RADIUS 6371000 /* Earth's mean radius (meters) */ define NX_EARTH_ORBITAL_PERIOD 365.25 /* Earth's orbital period (days) */ define NX_MOON_DISTANCE 384400000 /* Average Earth-Moon distance (meters) */

I Engineering Constants

I Engineering Constants

Engineering constants include values such as data size units and floating-point precision limits.

Engineering Constants

```c
/* Engineering and Computer Science Constants */
#define NX_KILOBYTE 1024            /* Bytes in a kilobyte */
#define NX_MEGABYTE (1024 * NX_KILOBYTE)    /* Bytes in a megabyte */
#define NX_GIGABYTE (1024 * NX_MEGABYTE)    /* Bytes in a gigabyte */
#define NX_FLOAT_EPSILON 1.19209290e-7        /* Smallest float difference
↪(32-bit) */
#define NX_DOUBLE_EPSILON 2.22044605e-16    /* Smallest double difference
↪(64-bit) */
#define NX_MAX_INT 2147483647           /* Maximum value of a 32-bit int */
#define NX_MIN_INT -2147483648           /* Minimum value of a 32-bit int */
```

/* Engineering and Computer Science Constants */ define NX_KILOBYTE 1024 /* Bytes in a kilobyte */ define NX_MEGABYTE (1024 * NX_KILOBYTE) /* Bytes in a megabyte */ define NX_GIGABYTE (1024 * NX_MEGABYTE) /* Bytes in a gigabyte */ define NX_FLOAT_EPSILON 1.19209290e-7 /* Smallest float difference (32-bit) */ define NX_DOUBLE_EPSILON 2.22044605e-16 /* Smallest double difference (64-bit) */ define NX_MAX_INT 2147483647 /* Maximum value of a 32-bit int */ define NX_MIN_INT -2147483648 /* Minimum value of a 32-bit int */

I Basic Operations

I Basic Operations

This section includes macros for basic arithmetic operations, bitwise operations, range checks, and geometry-related formulas.

Basic Operations

```c
/* Basic Operations */
#define NX_SQUARE_N(N) ((N) * (N))                          /* Calculates the square of a number */
#define NX_CUBE_N(N) (NX_SQUARE_N(N) * (N))                      /* Calculates the cube of a number */
#define NX_MIN(A, B) ((A) < (B) ? (A) : (B))                      /* Returns the minimum of two numbers */
#define NX_MAX(A, B) ((A) > (B) ? (A) : (B))                      /* Returns the maximum of two numbers */
#define NX_AVG(A, B) (((A) + (B)) / 2)                          /* Calculates the average of two numbers */
#define NX_ODD_N(N) ((N) % 2 == 1)                          /* Checks if a number is odd */
#define NX_EVEN_N(N) ((N) % 2 == 0)                          /* Checks if a number is even */
#define NX_ABS(N) ((N) < 0 ? -(N) : (N))                         /* Returns the absolute value of a number */
#define NX_IS_POWER_OF_TWO(N) ((N) && !((N) & ((N) - 1)))        /* Checks if N is a power of 2 */
```

/* Basic Operations */ define NX_SQUARE_N(N) ((N) * (N)) /* Calculates the square of a number */ define NX_CUBE_N(N) (NX_SQUARE_N(N) * (N)) /* Calculates the cube of a number */ define NX_MIN(A, B) ((A) < (B) ? (A) : (B)) /* Returns the minimum of two numbers */ define NX_MAX(A, B) ((A) > (B) ? (A) : (B)) /* Returns the maximum of two numbers */ define NX_AVG(A, B) (((A) + (B)) / 2) /* Calculates the average of two numbers */ define NX_ODD_N(N) ((N) define NX_EVEN_N(N) ((N) define NX_ABS(N) ((N) < 0 ? -(N) : (N)) /* Returns the absolute value of a number */ define NX_IS_POWER_OF_TWO(N) ((N) !((N) ((N) - 1))) /* Checks if N is a power of 2 */

I Endian Macros

I Endian Macros

Endian macros are used to detect the byte order (endianness) of the platform and provide utilities for converting between big-endian and little-endian formats.

Why does Intel use little-endian while older processors use big-endian?

Intel's x86 architecture, which became dominant in personal computers, adopted little-endian because it made certain operations easier. Many older systems and RISC-based processors (like PowerPC, SPARC, and older ARM designs) started with big-endian for compatibility with older networking and data transmission standards. Big-endian was originally favored because it aligns with how humans typically write numbers—most significant digit first (like 1234, not 4321). Little-endian, on the other hand, makes certain low-level memory operations more efficient, like reading variable-sized numbers without needing adjustments.

Pros and Cons of Each Endianness

- **Big-Endian (Most Significant Byte First)**:
  - ✓ Easier to read for humans (matches how numbers are written).
  - ✓ Often used in networking protocols (big-endian is standard in IP/TCP).
  - ✗ Can require extra steps when handling certain memory operations.
- **Little-Endian (Least Significant Byte First, Used by Intel)**:
  - ✓ Makes accessing multi-byte numbers more efficient in some cases.
  - ✓ Easier to handle variable-length data structures.
  - ✗ Feels counterintuitive when reading raw memory data (since the bytes are in "reverse" order).

I Endian Macros

Why is it called "Little-Endian"? The term "Little-Endian" comes from Jonathan Swift's *Gulliver's Travels*, where people argued over which end of an egg to break first: the Big End or the Little End. Computer scientists borrowed the term to describe data storage formats.

How Does This Relate to Two's Complement? Two's complement is a way computers store negative numbers, and while endianness doesn't affect the math of two's complement itself, it does influence how the bytes are arranged in memory.

- **Big-Endian:** The sign bit is stored at the beginning.
- **Little-Endian:** The sign bit is stored at the end.

Endian Macro Definitions

```c
/* Endian Macros */
#if defined(__BYTE_ORDER__) && (__BYTE_ORDER__ == __ORDER_BIG_ENDIAN__)
    #define NX_BIG_ENDIAN 1
    #define NX_LITTLE_ENDIAN 0
#elif defined(__BYTE_ORDER__) && (__BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__)
    #define NX_BIG_ENDIAN 0
    #define NX_LITTLE_ENDIAN 1
#elif defined(__BIG_ENDIAN__) || defined(_BIG_ENDIAN) || defined(__ARMEB__) ||
↪defined(__MIPSEB__) || defined(__POWERPC__) || defined(__sparc__)
    #define NX_BIG_ENDIAN 1
    #define NX_LITTLE_ENDIAN 0
#else
    #define NX_BIG_ENDIAN 0
    #define NX_LITTLE_ENDIAN 1
#endif

/* Conditional Endian Swap Based on Architecture */
#if NX_BIG_ENDIAN
    #define NX_TO_LITTLE16(X) nx_swap16(X)
    #define NX_TO_LITTLE32(X) nx_swap32(X)
    #define NX_TO_LITTLE64(X) nx_swap64(X)
    #define NX_TO_BIG16(X) (X)  /* Already big-endian */
    #define NX_TO_BIG32(X) (X)
    #define NX_TO_BIG64(X) (X)
#else
    #define NX_TO_LITTLE16(X) (X)  /* Already little-endian */
    #define NX_TO_LITTLE32(X) (X)
    #define NX_TO_LITTLE64(X) (X)
    #define NX_TO_BIG16(X) nx_swap16(X)
    #define NX_TO_BIG32(X) nx_swap32(X)
    #define NX_TO_BIG64(X) nx_swap64(X)
#endif
```

/* Endian Macros */ if defined(__BYTE_ORDER__) (__BYTE_ORDER__ == __ORDER_BIG_ENDIAN__) define NX_BIG_ENDIAN 1 define NX_LITTLE_ENDIAN 0 elif defined(__BYTE_ORDER__) (__BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__) define NX_BIG_ENDIAN 0 define NX_LITTLE_ENDIAN 1 elif defined(__BIG_ENDIAN__) || defined(_BIG_ENDIAN) || defined(__ARMEB__) || defined(__MIPSEB__) || defined(__POWERPC__) || defined(__sparc__) define NX_BIG_ENDIAN 1 define NX_LITTLE_ENDIAN 0 else define NX_BIG_ENDIAN 0 define NX_LITTLE_ENDIAN 1 endif
/* Conditional Endian Swap Based on Architecture */ if NX_BIG_ENDIAN define NX_TO_LITTLE16(X) nx_swap16(X) define NX_TO_LITTLE32(X) nx_swap32(X) define NX_TO_LITTLE64(X) nx_swap64(X) define NX_TO_BIG16(X) (X) /* Already big-endian */ define NX_TO_BIG32(X) (X) define NX_TO_BIG64(X) (X) else define NX_TO_LITTLE16(X) (X) /* Already little-endian */ define NX_TO_LITTLE32(X) (X) define NX_TO_LITTLE64(X) (X) define NX_TO_BIG16(X) nx_swap16(X) define NX_TO_BIG32(X) nx_swap32(X) define NX_TO_BIG64(X) nx_swap64(X) endif

I Endian Swap Functions

I Endian Swap Functions

These inline functions provide utilities for swapping the byte order (endianness) of 16-bit, 32-bit, and 64-bit values. They are useful for ensuring compatibility across architectures with different endianness.

Endian Swap Functions

```c
/* Swap 16-bit endian */
static inline nx_u16_t nx_swap16(nx_u16_t v)
{
    return (v >> 8) | (v << 8);
}

/* Swap 32-bit endian */
static inline nx_u32_t nx_swap32(nx_u32_t v)
{
    return ((v >> 24) & 0x000000FF) |
        ((v >> 8)  & 0x0000FF00) |
        ((v << 8)  & 0x00FF0000) |
        ((v << 24) & 0xFF000000);
}

/* Swap 64-bit endian */
static inline nx_u64_t nx_swap64(nx_u64_t v)
{
    return ((v >> 56) & 0x00000000000000FF) |
        ((v >> 40) & 0x000000000000FF00) |
        ((v >> 24) & 0x0000000000FF0000) |
        ((v >> 8)  & 0x00000000FF000000) |
        ((v << 8)  & 0x000000FF00000000) |
        ((v << 24) & 0x0000FF0000000000) |
        ((v << 40) & 0x00FF000000000000) |
        ((v << 56) & 0xFF00000000000000);
}
```

/* Swap 16-bit endian */ static inline nx_u16_t nx_swap16(nx_u16_t v) return (v » 8) | (v « 8);
/* Swap 32-bit endian */ static inline nx_u32_t nx_swap32(nx_u32_t v) return ((v » 24) 0x000000FF) | ((v » 8) 0x0000FF00) | ((v « 8) 0x00FF0000) | ((v « 24) 0xFF000000);
/* Swap 64-bit endian */ static inline nx_u64_t nx_swap64(nx_u64_t v) return ((v » 56) 0x00000000000000FF) | ((v » 40) 0x000000000000FF00) | ((v » 24) 0x0000000000FF0000) | ((v » 8) 0x00000000FF000000) | ((v « 8) 0x000000FF00000000) | ((v « 24) 0x0000FF0000000000) | ((v « 40) 0x00FF000000000000) | ((v « 56) 0xFF00000000000000);

I Endian Macros

What Are These Functions Doing?

These functions take a number in memory and flip the order of its bytes. This is necessary when converting data between big-endian and little-endian systems.

How does nx32 work? (32-bit swap)

A 32-bit integer is made up of 4 bytes. Imagine it like this:

Byte0 | Byte1 | Byte2 | Byte3

In big-endian, it would be stored like this:

[AB] [CD] [EF] [GH] (Most significant byte first)

In little-endian, it would be:

[GH] [EF] [CD] [AB] (Least significant byte first)

The nx32 function rearranges the bytes by shifting them left or right, then using bitwise operations (&) to isolate each chunk before placing it into its new position.

How does nx64 work? (64-bit swap)

This does the same thing, but for a 64-bit number (which has 8 bytes). It follows the same pattern:

Byte0 | Byte1 | Byte2 | Byte3 | Byte4 | Byte5 | Byte6 | Byte7

Just like before, it shifts and isolates bytes to reverse their order.

Why Is This Important?

If you're working on cross-platform applications or network protocols, you might need to convert data between formats to ensure all systems interpret it correctly. That's why these functions exist—to swap the byte order and avoid misinterpretation.

I Endian Macros

What happens when you use » (bit shifting)?

When you use », you're shifting bits to the right. Each shift moves everything one position to the right, and depending on the system, new bits are filled with either zeros or sign bits.
Example with [AB] [CD] [EF] [GH] (assuming a 32-bit number):
v » 8 means everything moves 8 bits to the right:

Before: [AB] [CD] [EF] [GH]
After »8: [00] [AB] [CD] [EF] (GH gets pushed out)

If you shift by 24:

Before: [AB] [CD] [EF] [GH]
After »24: [00] [00] [00] [AB] (Only the most significant byte remains)

What happens when you use & (bit masking)?

The & operator compares two numbers bit by bit. If both bits are 1, the result is 1; otherwise, it's 0.
Example: v & 0x000000FF keeps only the last byte, because 0x000000FF in binary is:
00000000 00000000 00000000 11111111

[AB] [CD] [EF] [GH] & 00000000 00000000 00000000 11111111
= [00] [00] [00] [GH]

Essentially, & helps extract specific parts of the number, while » moves things around.

How this applies to swapping endianness

The swap functions use:

- » to move bytes into the correct position
- & to extract only the parts we want before combining them into a new order

Bitmasking in CIDR notation

CIDR (Classless Inter-Domain Routing) uses bitmasking to define IP address ranges and subnet sizes.

I Endian Macros

How Bitmasking Works in CIDR

CIDR notation looks like this: `192.168.1.0/24`
The "/24" part is the subnet mask, which means:

- The first 24 bits of the IP address are fixed.
- The remaining 8 bits can vary (allowing for 256 possible addresses).

The subnet mask for /24 in binary:

```
11111111 11111111 11111111 00000000 = 255.255.255.0
```

Using a bitwise AND (&) operation, you can filter out the network portion of an IP address. **Example:**

```
IP:    192.168.1.73 ->  11000000 10101000 00000001 01001001
MASK:  255.255.255.0   ->  11111111 11111111 11111111 00000000
---------------------------------------------------------------
Result:  192.168.1.0  ->  11000000 10101000 00000001 00000000
```

This operation ensures that any device within the subnet keeps the same "network" part of the IP, while only the last portion changes.

Why Bitmasking is Important in Networking

- It helps routers quickly determine which subnet an IP belongs to.
- Makes it easy to allocate specific IP ranges to different parts of a network.
- Prevents overlapping IP addresses in large-scale networking.

So, in essence, CIDR uses bitmasks to group IP addresses together efficiently. That means networking and endianness both rely on shifting and masking bits, but for different reasons!

I Endian Macros

The Trick: Shifting and Masking

The trick is in shifting (» and «) and masking (&) to move bytes around and extract them properly.

Step-by-Step: How We Flip Bytes

Let's say we have a 32-bit number like this (big-endian format):

```
[AB] [CD] [EF] [GH] (stored as big-endian)
```

Now, we want to convert it to little-endian, so it becomes:

```
[GH] [EF] [CD] [AB]
```

Step 1: Using » to Isolate Bytes
Shifting moves bytes to the correct positions.

v » 24 moves [AB] all the way to the least significant byte:
[00] [00] [00] [AB] (only [AB] remains)
v » 8 moves [CD] down two slots, but we need to remove extra bits, so we use & to keep only [CD].
Step 2: Using & to Mask Out Unwanted Bytes
The & operation clears unnecessary data.

v » 8 & 0x0000FF00 makes sure only [CD] remains.
v » 24 & 0x000000FF ensures we just get [AB].
Step 3: Using « to Put Bytes in the New Order
Now that we've extracted each byte separately, we shift them into their new locations:

- [GH] should move left 24 bits (v « 24).
- [EF] should move left 8 bits (v « 8).
- [CD] stays where it is.
- [AB] was isolated earlier and now moves to the least significant byte.

Final Assembly Using | (Bitwise OR)
After shifting, we combine everything:

```
(nx_swap32) = (v >> 24 & 0x000000FF) | (v >> 8 & 0x0000FF00) |
              (v << 8 & 0x00FF0000) | (v << 24 & 0xFF000000);
```

I Endian Macros

Summary

Now, everything is reversed, so:

Before (Big-Endian): [AB] [CD] [EF] [GH]
After (Little-Endian): [GH] [EF] [CD] [AB]

- » moves bytes right to extract them.
- & masks out unwanted bits to keep the correct ones.
- « moves bytes left to their new positions.
- | combines the results to form the reversed number.

I Architecture and Alignment

I Architecture and Alignment

These macros define memory alignment attributes for variables and structures in C. The __attribute__((aligned(N))) directive tells the compiler to ensure that the specified variable or data structure is aligned to an N-byte boundary in memory.

- **NX_ALIGN_4, NX_ALIGN_8, NX_ALIGN_16, NX_ALIGN_32**: Align data to 4, 8, 16, or 32 bytes, respectively.
- **NX_CACHE_ALIGN_64, NX_CACHE_ALIGN_128, NX_CACHE_ALIGN_256**: Align data to cache line sizes of 64, 128, or 256 bytes, respectively.
- **NX_IS_64BIT**: Set to 1 if the system is 64-bit, otherwise set to 0.
- **nx_size_t**: Defines an unsigned integer type for sizes, appropriate for the architecture.
- **nx_ptrdiff_t**: Defines a signed integer type for pointer differences, appropriate for the architecture.

Types of Alignments

- NX_ALIGN_4, NX_ALIGN_8, NX_ALIGN_16, NX_ALIGN_32:
  - Align data to 4, 8, 16, or 32 bytes, respectively.
  - Ensures variables are stored efficiently in memory.
  - Improves CPU performance by avoiding penalties for unaligned memory access.
- NX_CACHE_ALIGN_64, NX_CACHE_ALIGN_128, NX_CACHE_ALIGN_256:
  - Align data to 64-, 128-, or 256-byte boundaries.
  - Optimizes cache performance by aligning data to cache line sizes.
  - Reduces cache thrashing and improves CPU efficiency.

I Architecture and Alignment

Why is Alignment Important?

Alignment is crucial for performance in systems where memory access is optimized for specific boundaries, particularly in:

- Embedded systems, where memory and processing resources are constrained.
- High-performance computing, where unaligned memory access can significantly impact throughput.
- SIMD (Single Instruction Multiple Data) operations, which require tightly aligned memory for vectorized processing.

Analogies for Clarity

- NX_ALIGN_4: Think of this as saying, "Put this data in a spot that's a multiple of 4 bytes." It's akin to arranging chocolates in rows of 4 for uniformity and easy access.
- NX_CACHE_ALIGN_64: This is like saying, "Make sure this data fits into a special box that's 64 bytes big." This helps the computer grab the data faster since it matches the size of its "grabber" (the cache line).

Cache Thrashig

Cache thrashing happens when a computer keeps swapping data in and out of its cache too often, instead of using it efficiently. This slows things down because the CPU spends more time moving data around than doing actual work.

Think of it like a cluttered desk

Imagine you have a small desk and need a few papers to work on. If you keep shuffling papers on and off the desk because there's not enough space, you spend more time rearranging than actually working. That's cache thrashing—the CPU keeps replacing data in the cache because it doesn't fit, making everything slower.

Why does this happen?

- Not enough cache space: If a program frequently accesses large amounts of data that don't fit in the cache, it keeps replacing items.
- Poor memory access patterns: Some algorithms keep switching between memory locations instead of accessing them efficiently.
- Conflicting cache lines: If multiple pieces of data keep landing in the same cache slot, they overwrite each other too quickly.

I Architecture and Alignment

How to Fix It?

- Optimize data structures so that frequently accessed data stays close together.
- Use cache-friendly algorithms that avoid unnecessary swaps.
- Increase cache size (if possible) to fit more data at once.

Basically, cache thrashing is the CPU equivalent of reorganizing your room so much that you never actually get to relax in it!

Architecture and Alignment Macros

```c
/* Architecture Detection */
#if defined(__x86_64__) || defined(_M_X64) || defined(__aarch64__) ||
↪defined(__PPC64__) || defined(__mips64)
    #define NX_IS_64BIT 1
    typedef unsigned long long nx_size_t;
    typedef long long nx_ptrdiff_t;
#else
    #define NX_IS_64BIT 0
    typedef unsigned long nx_size_t;
    typedef long nx_ptrdiff_t;
#endif

/* Memory Alignment Macros */
#define NX_ALIGN_4 __attribute__((aligned(4)))   /* Align to 4 bytes */
#define NX_ALIGN_8 __attribute__((aligned(8)))   /* Align to 8 bytes */
#define NX_ALIGN_16 __attribute__((aligned(16))) /* Align to 16 bytes */
#define NX_ALIGN_32 __attribute__((aligned(32))) /* Align to 32 bytes */
#define NX_CACHE_ALIGN_64 __attribute__((aligned(64))) /* Align to 64-byte
↪cache line */
#define NX_CACHE_ALIGN_128 __attribute__((aligned(128))) /* Align to 128-byte
↪cache line */
#define NX_CACHE_ALIGN_256 __attribute__((aligned(256))) /* Align to 256-byte
↪cache line */
```

/* Architecture Detection */ if defined(__x86_64__) || defined(_M_X64) || defined(__aarch64__) || defined(__PPC64__) || defined(__mips64) define NX_IS_64BIT 1 typedef unsigned long long nx_size_t; typedef long long nx_ptrdiff_t; else define NX_IS_64BIT 0 typedef unsigned long nx_size_t; typedef long nx_ptrdiff_t; endif
/* Memory Alignment Macros */ define NX_ALIGN_4 __attribute__((aligned(4))) /* Align to 4 bytes */ define NX_ALIGN_8 __attribute__((aligned(8))) /* Align to 8 bytes */ define NX_ALIGN_16 __attribute__((aligned(16))) /* Align to 16 bytes */ define NX_ALIGN_32 __attribute__((aligned(32))) /* Align to 32 bytes */ define NX_CACHE_ALIGN_64 __attribute__((aligned(64))) /* Align to 64-byte cache line

```
*/ define NX_CACHE_ALIGN_128 __attribute__((aligned(128))) /* Align to 128-byte cache line */ define
NX_CACHE_ALIGN_256 __attribute__((aligned(256))) /* Align to 256-byte cache line */
```

I CPU Features

I CPU Features

This section includes macros for detecting CPU instruction sets, enabling branch prediction, and optimizing memory access through prefetching.

- **NX_HAS_SSE**: Set to 1 if SSE instructions are available, otherwise set to 0.
- **NX_HAS_AVX**: Set to 1 if AVX instructions are available, otherwise set to 0.
- **NX_HAS_NEON**: Set to 1 if NEON instructions are available, otherwise set to 0.

CPU Features Macros

```c
/* CPU Instruction Set Detection */
#if defined(__SSE__) || defined(__x86_64__) || defined(_M_X64)
    #define NX_HAS_SSE 1  /* SSE Instructions Available */
#else
    #define NX_HAS_SSE 0
#endif

#if defined(__AVX__) || defined(__AVX2__) || defined(__x86_64__) ||
↪defined(_M_X64)
    #define NX_HAS_AVX 1  /* AVX Instructions Available */
#else
    #define NX_HAS_AVX 0
#endif

#if defined(__ARM_NEON) || defined(__ARM_FEATURE_NEON)
    #define NX_HAS_NEON 1 /* NEON Instructions Available (ARM) */
#else
    #define NX_HAS_NEON 0
#endif

/* Branch Prediction Macros */
#if defined(__GNUC__) || defined(__clang__)
    #define NX_LIKELY(x) __builtin_expect(!!(x), 1) /* Likely branch */
    #define NX_UNLIKELY(x) __builtin_expect(!!(x), 0) /* Unlikely branch */
#else
    #define NX_LIKELY(x) (x) /* No prediction available */
    #define NX_UNLIKELY(x) (x)
#endif

/* Instruction Prefetching */
#if defined(__GNUC__) || defined(__clang__)
    #define NX_PREFETCH(addr) __builtin_prefetch(addr)
#elif defined(_MSC_VER)
    #include <mmintrin.h}
    #define NX_PREFETCH(addr) _mm_prefetch((const char *)(addr), _MM_HINT_T0)
```

```c
#else
    #define NX_PREFETCH(addr) /* No prefetch available */
#endif
```

/* CPU Instruction Set Detection */ if defined(__SSE__) || defined(__x86_64__) || defined(_M_X64) define NX_HAS_SSE 1 /* SSE Instructions Available */ else define NX_HAS_SSE 0 endif
if defined(__AVX__) || defined(__AVX2__) || defined(__x86_64__) || defined(_M_X64) define NX_HAS_AVX 1 /* AVX Instructions Available */ else define NX_HAS_AVX 0 endif
if defined(__ARM_NEON) || defined(__ARM_FEATURE_NEON) define NX_HAS_NEON 1 /* NEON Instructions Available (ARM) */ else define NX_HAS_NEON 0 endif
/* Branch Prediction Macros */ if defined(__GNUC__) || defined(__clang__) define NX_LIKELY(x) __builtin_expect(!!(x), 1) /* Likely branch */ define NX_UNLIKELY(x) __builtin_expect(!!(x), 0) /* Unlikely branch */ else define NX_LIKELY(x) (x) /* No prediction available */ define NX_UNLIKELY(x) (x) endif
/* Instruction Prefetching */ if defined(__GNUC__) || defined(__clang__) define NX_PREFETCH(addr) __builtin_prefetch(addr) elif defined(_MSC_VER) include <mmintrin.h define NX_PREFETCH(addr) _mm_prefetch((const char *)(addr), _MM_HINT_T0) else define NX_PREFETCH(addr) /* No prefetch available */ endif

I CPU Core Detection

I CPU Core Detection

This inline function retrieves the number of logical CPU cores on the current system. On macOS, it uses the `sysctlbyname` function to query the `hw.logicalcpu` property.

CPU Core Detection Function

```
#ifndef NX_CPU_CORES
static inline nx_u32_t nx_cpu_cores()
{
    nx_u32_t c;
    nx_size_t s = sizeof(c);
    sysctlbyname("hw.logicalcpu", &c, &s, NX_NULL, 0);
    return c;
}
#define NX_CPU_CORES nx_cpu_cores()
#endif
```

ifndef NX_CPU_CORES static inline nx_u32_t nx_cpu_cores() nx_u32_t c; nx_size_t s = sizeof(c); sysctlbyname("hw.logicalcpu", c, s, NX_NULL, 0); return c; define NX_CPU_CORES nx_cpu_cores() endif