Elementary Socket Calls

Socket Address Structure (SAS)

IPV4;

IPV6;

UNIX domain socket;

Generic socket address structure

Value Result Argument

Socket Address Structure (SAS)

- Various structures are used in Unix Socket Programming to hold information about the address and port, and other information.
- Most socket functions require a pointer to a socket address structure as an argument. Each supported protocol suite defines its own socket address structure.
- The basic purpose of socket address structures for IP based protocols is to specify an IP address and port.
- Socket address structures are passed to connect () – to specify the IP address and port to connect to.
- Socket address structures are passed to bind () – to specify the IP address and/or port to bind to.
- The names of these structures begin with ”sockaddr_ ” and end with a unique suffix for each protocol suite.

Socket Address Structure (SAS)

- All socket address structures start with “sockaddr_”
  - IPV4 socket address structure: sockaddr_in
  - IPV6 socket address structure: sockaddr_in6
  - Generic socket address structure: sockaddr
  - UNIX Domain socket address structure: sockaddr_un

The first structure is *sockaddr* that holds the socket information −

```
struct sockaddr {
   unsigned short    sa_family;
   char              sa_data[14];
};
```

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sa_data	Protocol-specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address.

- **IPv4 Socket Address Structure**
  - commonly called an "Internet socket address structure," is named sockaddr_in and is defined by including the <netinet/in.h> header.
- **IPv6 Socket Address Structure**
  - The IPv6 socket address is defined by including the <netinet/in.h> header

Datatypes required by the POSIX specification.

Datatype	Description	Header
int8_t	Signed 8-bit integer	<sys/types.h>
uint8_t	Unsigned 8-bit integer	<sys/types.h>
int16_t	Signed 16-bit integer	<sys/types.h>
uint16_t	Unsigned 16-bit integer	<sys/types.h>
int32_t	Signed 32-bit integer	<sys/types.h>
uint32_t	Unsigned 32-bit integer	<sys/types.h>
sa_family_t	Address family of socket address structure	<sys/socket.h>
socklen_t	Length of socket address structure, normally uint32_t	<sys/socket.h>
in_addr_t	IPv4 address, normally uint32_t	<netinet/in.h>
in_port_t	TCP or UDP port, normally uint16_t	<netinet/in.h>

sockaddr in

The second structure that helps you to reference to the socket's elements is as follows −

```
struct sockaddr_in {
   short int          sin_family;
   unsigned short int sin_port;
   struct in_addr     sin_addr;
   unsigned char      sin_zero[8];
};
```

Here is the description of the member fields −

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sin_port	Service Port	A 16-bit port number in Network Byte Order.
sin_addr	IP Address	A 32-bit IP address in Network Byte Order.
sin_zero	Not Used	You just set this value to NULL as this is not being used.

in addr

This structure is used only in the above structure as a structure field and holds 32 bit netid/hostid.

```
struct in_addr {
   unsigned long s_addr;
};
```

Here is the description of the member fields −

Attribute	Values	Description
s_addr	service port	A 32-bit IP address in Network Byte Order.

Socket Address Structure (SAS) for IPv4

- An IPv4 socket address structure, commonly called an "Internet socket address structure", is named sockaddr_in
- Defined in <netinet/in.h> header.
- POSIX only requires the following 3 fields : sin_family, sin_port & sin_addr

Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`.

```
struct in_addr {
  in_addr_t   s_addr;           /* 32-bit IPv4 address */
                                /* network byte ordered */
};

struct sockaddr_in {
  uint8_t         sin_len;      /* length of structure (16) */
  sa_family_t     sin_family;   /* AF_INET */
  in_port_t       sin_port;     /* 16-bit TCP or UDP port number */
                                /* network byte ordered */
  struct in_addr  sin_addr;     /* 32-bit IPv4 address */
                                /* network byte ordered */
  char            sin_zero[8];  /* unused */
};
```

Socket Address Structure (SAS) for IPv4

- sin_len: the length field.
- POSIX requires only three members in the structure: sin_family, sin_addr, and sin_port. Almost all implementations add the sin_zero member so that all socket address structures are at least 16 bytes in size.
  - ***in_addr_t*** datatype must be an **unsigned integer** type of at least 32 bits,
  - ***in_port_t*** must be an **unsigned integer** type of at least 16 bits, and
  - ***sa_family_t*** can be any **unsigned integer** type.
- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in **network byte order**.
- ***sin_zero*** member is unused.
- Socket address structures are used only on a given host.

IPv6 Socket Address Structure

- The IPv6 socket address is defined by including the <netinet/in.h> header:

Figure 3.4 IPv6 socket address structure: `sockaddr_in6`.

```
struct in6_addr {
  uint8_t  s6_addr[16];          /* 128-bit IPv6 address */
                                 /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t         sin6_len;      /* length of this struct (28) */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* flow information, undefined */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
  uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};
```

IPv6 Socket Address Structure (SAS)

- The ***SIN6_LEN*** constant must be defined if the system supports the length member for socket address structures.
- The IPv6 family is ***AF_INET6,*** whereas the IPv4 family is ***AF_INET***
- The members in this structure are ordered so that if the ***sockaddr_in6*** structure is 64-bit aligned, so is the 128-bit sin6_addr member.
- The ***sin6_flowinfo*** member is divided into two fields:
  - The low-order 20 bits are the flow label
  - The high-order 12 bits are reserved
- The sin6_scope_id identifies the scope zone in which a scoped address is meaningful, most commonly an interface index for a link-local address

Comparison of Socket Address Structures

- Socket address structures all contain a one-byte length field
- The family field also occupies one byte
- Any field that must be at least some number of bits is exactly that number of bits

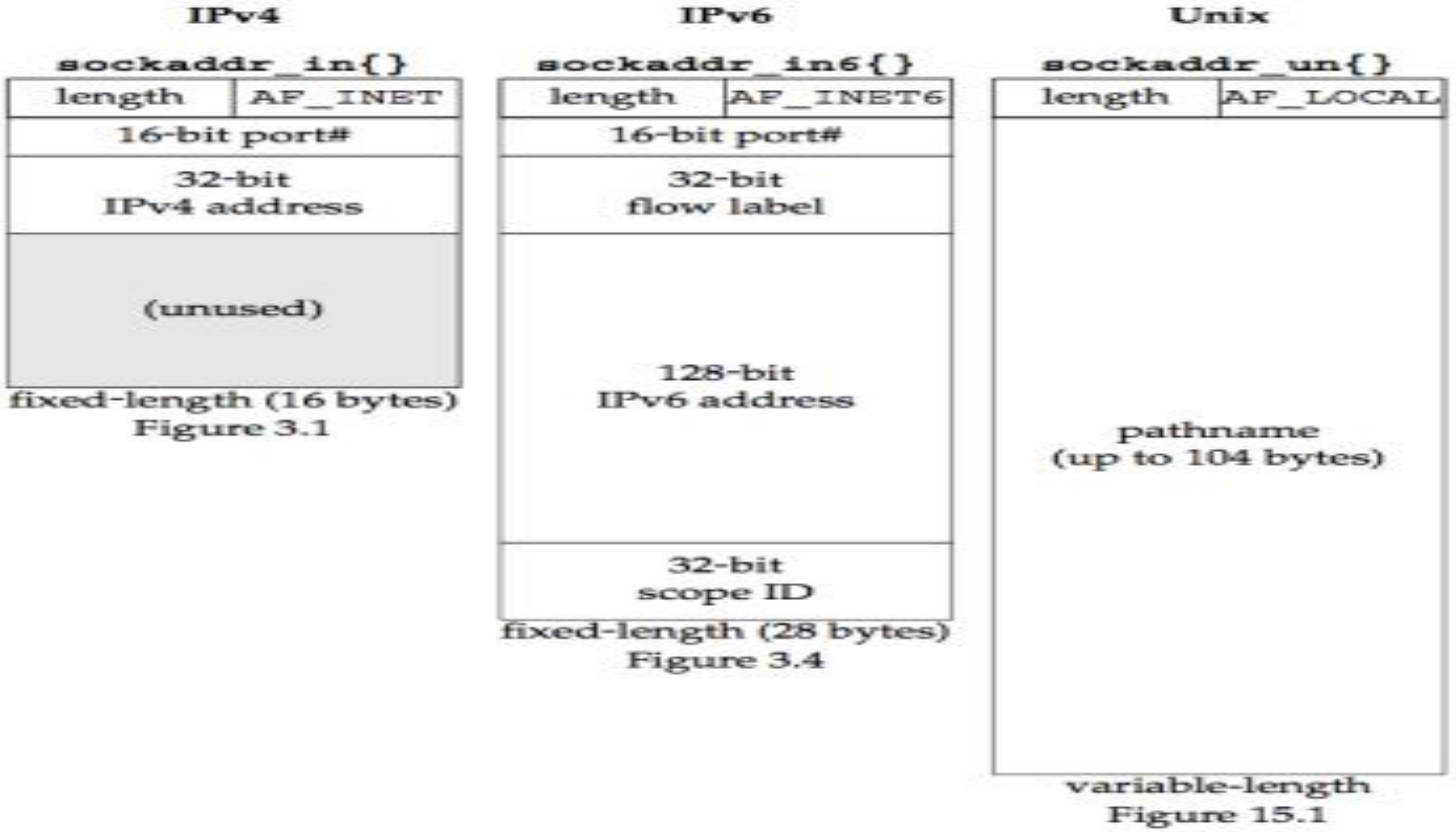


Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`.

```
struct in_addr {
  in_addr_t   s_addr;           /* 32-bit IPv4 address */
                                /* network byte ordered */
};

struct sockaddr_in {
  uint8_t         sin_len;      /* length of structure (16) */
  sa_family_t     sin_family;   /* AF_INET */
  in_port_t       sin_port;     /* 16-bit TCP or UDP port number */
                                /* network byte ordered */
  struct in_addr  sin_addr;     /* 32-bit IPv4 address */
                                /* network byte ordered */
  char            sin_zero[8];  /* unused */
};
```

Figure 3.4 IPv6 socket address structure: `sockaddr_in6`.

```
struct in6_addr {
  uint8_t  s6_addr[16];           /* 128-bit IPv6 address */
                                  /* network byte ordered */
};

#define SIN6_LEN      /* required for compile-time tests */

struct sockaddr_in6 {
  uint8_t         sin6_len;      /* length of this struct (28) */
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* transport layer port# */
                                 /* network byte ordered */
  uint32_t        sin6_flowinfo; /* flow information, undefined */
  struct in6_addr sin6_addr;     /* IPv6 address */
                                 /* network byte ordered */
  uint32_t        sin6_scope_id; /* set of interfaces for a scope */
};
```

Unix Domain Socket Address Structure

- defined by including the <sys/un.h> header.

Figure 15.1 Unix domain socket address structure: `sockaddr_un`.

```
struct sockaddr_un {
  sa_family_t sun_family;     /* AF_LOCAL */
  char        sun_path[104];  /* null-terminated pathname */
};
```

- The pathname stored in the sun_path array must be null-terminated.

Generic Socket Address Structure

- A socket address structures is always passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.
- A generic socket address structure in the ***<sys/socket.h>*** header:

```
struct sockaddr {
  uint8_t      sa_len;
  sa_family_t  sa_family;    /* address family: AF_xxx value */
  char         sa_data[14];  /* protocol-specific address */
};
```

- The socket functions are then defined as taking a pointer to the generic socket address structure
  - int bind(int, struct sockaddr *, socklen_t);

Value-Result Arguments

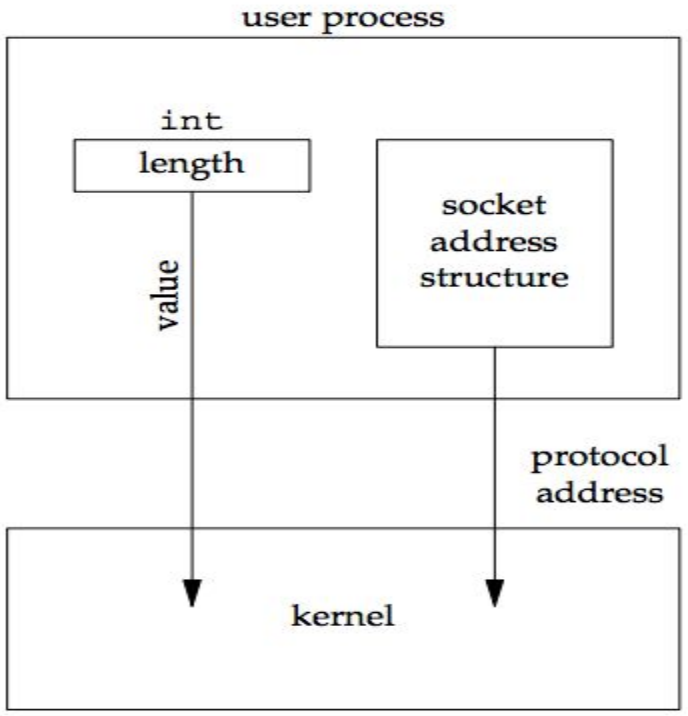
- When a socket address structure is passed to any socket function, it is always passed by reference (a pointer to the structure is passed). The length of the structure is also passed as an argument.
- The way in which the length is passed depends on which direction the structure is being passed:
  - From the **process to the kernel**
  - From the **kernel to the process**
-

From process to kernel

- **bind, connect,** and **sendto** functions pass a socket address structure from the process to the kernel.
- Arguments to these functions:
  - The pointer to the socket address structure
  - The integer size of the structure

```
struct sockaddr_in serv;

/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```



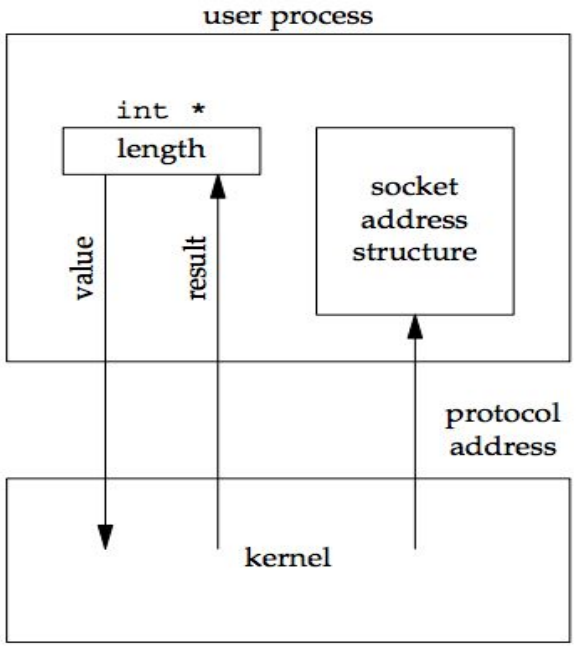
The datatype for the size of a socket address structure is actually ***socklen_t*** and not int, but the POSIX specification recommends that ***socklen_t*** be defined as ***uint32_t.***

From kernel to process

- ***accept, recvfrom, getsockname,*** *and* ***getpeername*** functions pass a socket address structure from the kernel to the process.
- Arguments to these functions:
  - The pointer to the socket address structure
  - The pointer to an integer containing the size of the structure.

```
struct sockaddr_un  cli;   /* Unix domain */
socklen_t  len;

len = sizeof(cli);         /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```



Value-Result Arguments

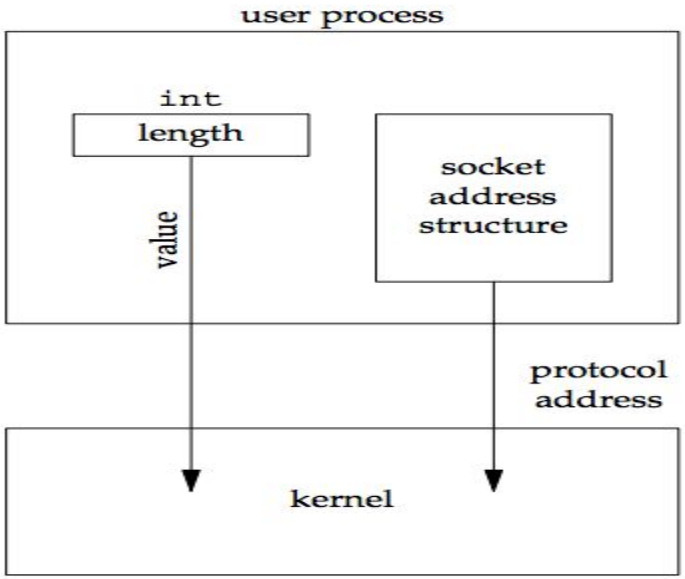
From the **process to the kernel**	From the **kernel to the process**
bind, connect, and sendto functions pass a socket address structure	accept, recvfrom, getsockname, and getpeername functions pass a socket address structure

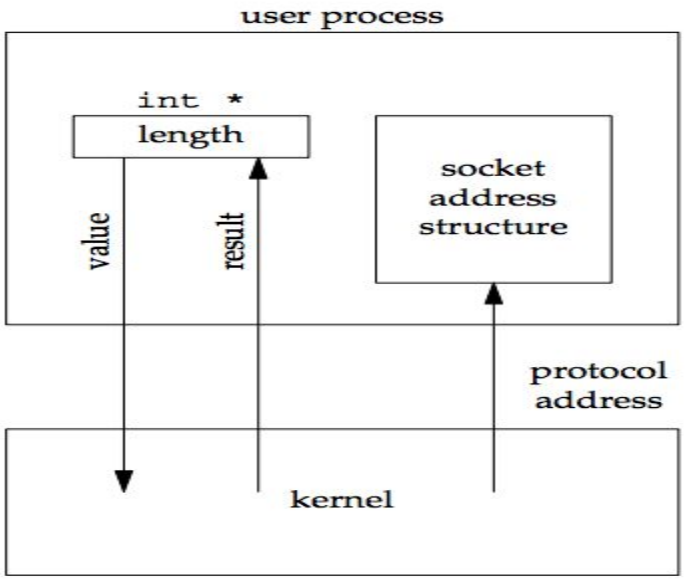
```
struct sockaddr_in serv;

/* fill in serv{} */
connect (sockfd, (SA *) &serv, sizeof(serv));
```

```
struct sockaddr_un  cli;    /* Unix domain */
socklen_t  len;

len = sizeof(cli);          /* len is a value */
getpeername(unixfd, (SA *) &cli, &len);
/* len may have changed */
```





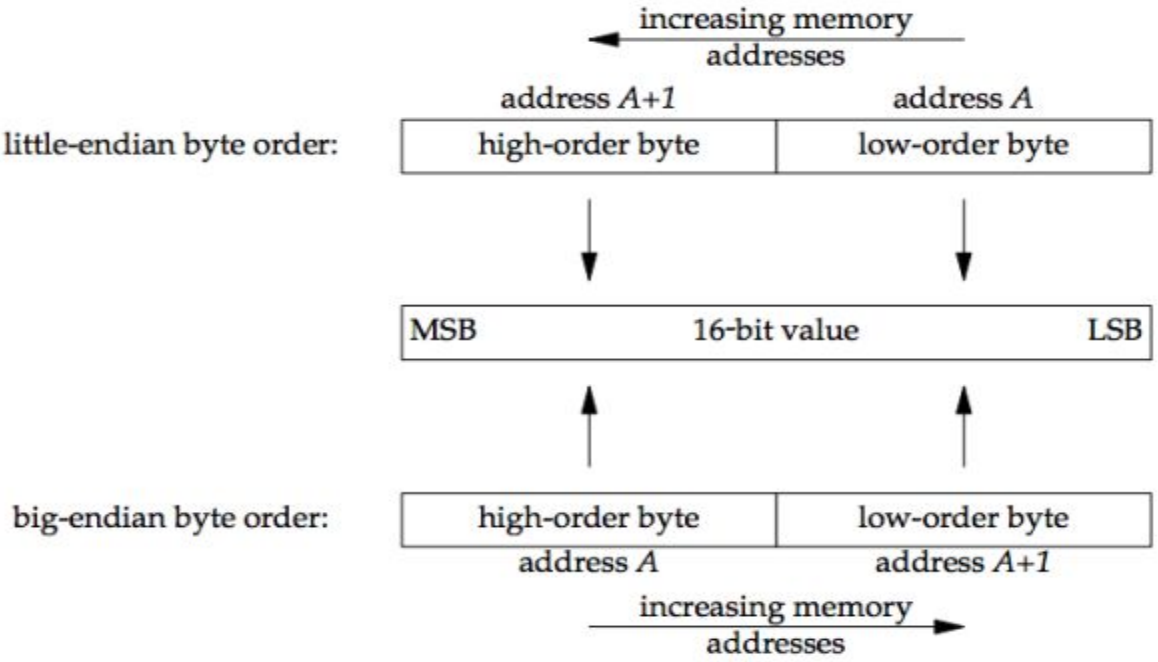
- **Value-result argument** : the size changes from an integer to be a pointer to an integer because the size is both a value when the function is called and a result when the function returns.
  - As a **value**: it tells the kernel the size of the structure so that the kernel does not write past the end of the structure when filling it in
  - As a **result**: it tells the process how much information the kernel actually stored in the structure
- For two other functions that pass socket address structures, recvmsg and sendmsg, the length field is not a function argument but a structure member.
- If the socket address structure is fixed-length, the value returned by the kernel will always be that fixed size: ***16 for an IPv4 sockaddr_in*** and ***28 for an IPv6 sockaddr_in6.***

Byte Ordering and manipulating function

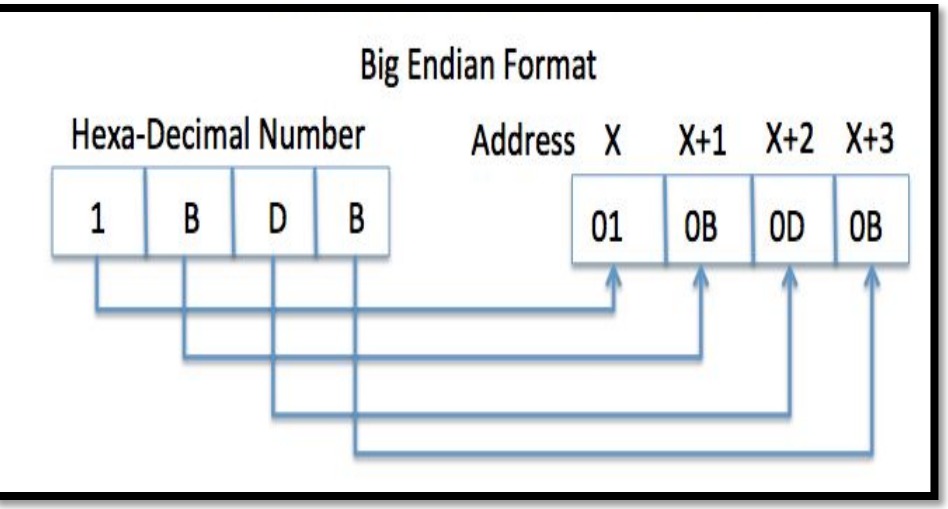
htonl(), htons(), ntohl(), ntohs(), inet_addr(), inet_aton(), inet_ntoa(), inet_pton()

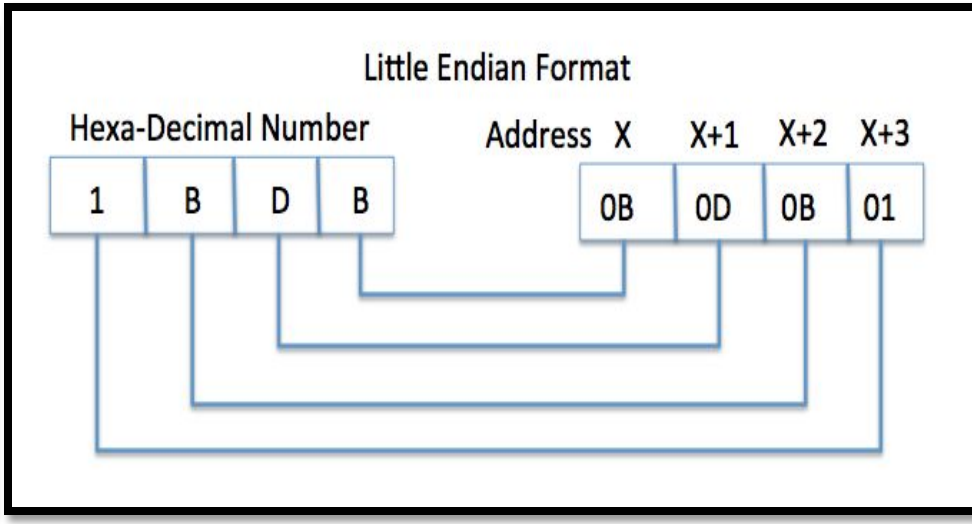
Byte Ordering Functions

- For a 16-bit integer that is made up of 2 bytes, there are two ways to store the two bytes in memory:
  - **Little-endian** order: low-order byte is at the starting address.
  - **Big-endian** order: high-order byte is at the starting address.



- While most machines handle data in **Little Endian Format** the network transmission and the **Internet** is using **Big Endian Format**.
- Hence, before sending data to the network and after receiving data from the network, the translation to Network Order ie, to Big Endian Order and vice versa is a routine task.
- In **Little Endian Format**, **least significant byte(LSB)** will be placed in the lowest address and the **most significant byte(MSB)** will be placed in the highest address.
- For example, if the starting address is X, the **least significant byte** will be placed in X, and the next least significant byte will be placed in X+1, the next least significant byte at X+2 and the **most significant byte** will be placed in X+3. In Little Endian, the **LSB** is stored at the smallest address.





Byte Ordering Functions

- The network byte order is defined to always be big-endian, which may differ from the host byte order on a particular machine.
- Using network byte ordering for data exchanged between hosts allows hosts using different architectures to exchange address information
  - htonl() translates an unsigned long integer into network byte order.
  - htons() translates an unsigned short integer into network byte order.
  - ntohl() translates an unsigned long integer into host byte order.
  - ntohs() translates an unsigned short integer into host byte order.

- **unsigned short htons(unsigned short hostshort) −**
  - This function converts 16-bit (2-byte) quantities from host byte order to network byte order.
- **unsigned long htonl(unsigned long hostlong) −**
  - This function converts 32-bit (4-byte) quantities from host byte order to network byte order.
- **unsigned short ntohs(unsigned short netshort) −**
  - This function converts 16-bit (2-byte) quantities from network byte order to host byte order.
- **unsigned long ntohl(unsigned long netlong) −**
  - This function converts 32-bit quantities from network byte order to host byte order.

- POSIX specification say that certain fields in the socket address structures must be maintained in network byte order.

unp_htons.h

```
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue);
uint32_t htonl(uint32_t host32bitvalue);

/* Both return: value in network byte order */

uint16_t ntohs(uint16_t net16bitvalue);
uint32_t ntohl(uint32_t net32bitvalue);

/* Both return: value in host byte order */
```

- `h` stands for *host*
- `n` stands for *network*
- `s` stands for *short* (16-bit value, e.g. TCP or UDP port number)
- `l` stands for *long* (32-bit value, e.g. IPv4 address)

Function Name	Description	Translation Direction
`socket.ntohl(Int32bit_postive)`	• **`ntohl()`** function converts a 32 bit integer from network order to host order.	Big Endian to Little Endian
`socket.ntohs(Int16bit_postive)`	• **`ntohs()`** function of socket module converts a 16 bit integer from network format to host format. •	Big Endian to Little Endian
`socket.htonl(Int32bit_postive)`	• **`htonl()`** function converts a 32 bit positive integer from host byte order to network byte order.	Little Endian to Big Endian
`ocket.htons(Int16bit_postive)`	• **`htons()`** function converts a 16 bit positive integer from host byte order to network byte order.	Little Endian to Big Endian

Byte Manipulation Functions

- Two types functions differ in whether they deal with null-terminated C strings:
- The functions that operate on multibyte fields, without interpreting the data, and without assuming that the data is a null-terminated C string.
- These types of functions ***deal with socket address structures to manipulate fields such as IP addresses***, which can contain bytes of 0, but are not C character strings.
  - The functions whose names begin with ***b*** (for byte)
  - The functions whose names begin with ***mem*** (for memory)
- The functions that deal with null-terminated C character strings (beginning with str (for string), defined by including the ***<string.h>*** header)

The memory pointed to by the ***const*** pointer is read but not modified by the function.

- ***bzero*** sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0.
- ***bcopy*** moves the specified number of bytes from the source to the destination.
- ***bcmp*** compares two arbitrary byte strings. The return value is zero if the two byte strings are identical; otherwise, it is nonzero

unp_bzero.h

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest, size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);

/* Returns: 0 if equal, nonzero if unequal */
```

- ***memset*** sets the specified number of bytes to the value c in the destination
- ***memcpy*** is similar to bcopy, but the order of the two pointer arguments is swapped
- ***memcmp*** compares two arbitrary byte strings

unp_memset.h

```
#include <string.h>

void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t nbytes);
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

/* Returns: 0 if equal, <0 or >0 if unequal (see text) */
```

unp_bzero.h

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);
void bcopy(const void *src, void *dest, size_t nbytes);
int bcmp(const void *ptr1, const void *ptr2, size_t nbytes);

/* Returns: 0 if equal, nonzero if unequal */
```

unp_memset.h

```
#include <string.h>

void *memset(void *dest, int c, size_t len);
void *memcpy(void *dest, const void *src, size_t nbytes);
int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

/* Returns: 0 if equal, <0 or >0 if unequal (see text) */
```

inet_aton, inet_addr, and inet_ntoa

- These functions convert Internet addresses between ASCII strings (what humans prefer to use) and network byte ordered binary values (values that are stored in socket address structures).
- **inet_aton:** converts the C character string pointed to by *strptr* into its 32-bit binary network byte ordered value, which is stored through the pointer *addrptr*
- **inet_addr:** does the same conversion, returning the 32-bit binary network byte ordered value as the return value. It is deprecated and any new code should use inet_aton instead
- **inet_ntoa:** converts a 32-bit binary network byte ordered IPv4 address into its corresponding dotted-decimal string.

unp_inet_aton.h

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);
/* Returns: 1 if string was valid, 0 on error */

in_addr_t inet_addr(const char *strptr);
/* Returns: 32-bit binary network byte ordered IPv4 address; INADDR_NONE if error */

char *inet_ntoa(struct in_addr inaddr);
/* Returns: pointer to dotted-decimal string */
```

inet_pton and inet_ntop

- These two functions are new with IPv6 and work with both IPv4 and IPv6 addresses.
- We use these two functions throughout the text. The letters "p" and "n" stand for *presentation* and *numeric*.
- The presentation format for an address is often an ASCII string and the numeric format is the binary value that goes into a socket address structure.

unp_inet_pton.h

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);
/* Returns: 1 if OK, 0 if input not a valid presentation format, -1 on error */

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
/* Returns: pointer to result if OK, NULL on error */
```

Arguments:

- *family*: is either AF_INET or AF_INET6. If *family* is not supported, both functions return an error with errno set to EAFNOSUPPORT.

Functions:

- **inet_pton**: converts the string pointed to by *strptr*, storing the binary result through the pointer *addrptr*. If successful, the return value is 1. If the input string is not a valid presentation format for the specified *family*, 0 is returned.
- **inet_ntop** does the reverse conversion, from numeric (*addrptr*) to presentation (*strptr*).

- The following figure summarizes the five functions on address conversion functions: