# Listen for and receive UDP datagrams in C

**Content**



**Tested on**

Debian (Lenny)
Ubuntu (Lucid)

## Objective

To listen for and receive inbound UDP datagrams in C

## Scenario

Suppose that you wish to write a server that implements the UDP-based variant of the Daytime Protocol, as defined by RFC 867

This is a very simple protocol whereby the client sends a datagram to the server, then the server responds with a datagram containing a human-readable copy of the current date and time. The datagram from the client is not required to have any particular content.

## Method

### Overview

The method described here has four steps:

1. Construct the local socket address.
2. Create the socket.
3. Bind the local address to the socket.
4. Receive and handle datagrams as they arrive.

This is the appropriate procedure when listening for unsolicited datagrams, as in the scenario described above. See below for how it can be adapted to:

- listening for a reply to a datagram that you have sent, or
- exchanging many datagrams with a particular remote host.

The following header files will be needed:

```
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

and if using `recvmsg` to receive datagrams:

```
#include <sys/uio.h>
```

## Construct the local socket address

In order to listen for UDP datagrams it is necessary to choose a port number and, optionally, a local IP address on which to listen. The combination of these two values is treated as a single entity called the socket address, which is represented by a `struct sockaddr_in` for IPv4 or a `struct sockaddr_in6` for IPv6.

Most common network services have an assigned port number on which they are normally expected to listen. While it makes sense to use this as the default, it is good practice to make the port number configurable. Possible reasons for wanting to override the assigned port number include:

- running multiple instances of a network service on the same machine,
- running a network service that would normally use a well-known port number from a non-root account, or
- making port scanning more time-consuming than it would be if the standard port number were used.

The local IP address should normally default to either the the wildcard address or the loopback address, but like the port number it is good practice to make it configurable. When a service is bound to a particular IP address it will only accept connections directed to that address, whereas when bound to the wildcard address it will accept connections to any local address. Binding to the loopback address has the effect of prohibiting connections from other machines.

For most purposes the best way to construct the socket address is by calling `getaddrinfo`. This takes a string containing the IP address and a string containing the port number, and converts them into a `sockaddr_in` or a `sockaddr_in6` as appropriate. It is also able to resolve hostnames and service names:

```
const char* hostname=0; /* wildcard */
const char* portname="daytime";
struct addrinfo hints;
memset(&hints,0,sizeof(hints));
hints.ai_family=AF_UNSPEC;
hints.ai_socktype=SOCK_DGRAM;
hints.ai_protocol=0;
hints.ai_flags=AI_PASSIVE|AI_ADDRCONFIG;
```

```
struct addrinfo* res=0;
int err=getaddrinfo(hostname,portname,&hints,&res);
if (err!=0) {
    die("failed to resolve local socket address (err=%d)",err);
}
```

The `hints` argument contains additional information to help guide the conversion. In this example:

- The address family has been left unspecified so that both IPv4 and IPv6 addresses can be returned. In principle you could receive results for other address families too: you can either treat this as a feature, or filter out any unwanted results after the call to `getaddrinfo`.
- The socket type has been constrained to `SOCK_DGRAM`. This allows UDP but excludes TCP.
- The protocol has been left unspecified because it is only meaningful in the context of a specific address family. If the address family had been set to `AF_INET` or `AF_INET6` then this field could have been set to `IPPROTO_UDP` (but it is equally acceptable to leave it set to zero).
- The `AI_PASSIVE` flag has been set because the address is intended for use by a server. It causes the IP address to default to the wildcard address as opposed to the loopback address.
- The `AI_ADDRCONFIG` flag has been set so that IPv6 results will only be returned if the server has an IPv6 address, and similarly for IPv4.

The `res` argument is used to return a linked list of `addrinfo` structures containing the address or addresses that were found. If the network service daemon has the ability to listen on multiple sockets then it should open one for each address in the list. Otherwise it is considered acceptable to use the first result and discard the remainder.

The memory occupied by the result list should be released by calling `freeaddrinfo` once it is no longer needed, however this cannot be done until after the socket has been created and bound.

## Create the socket.

The socket that will be used to listen for inbound datagrams should be created using the `socket` function. This takes three arguments:

1. the domain (AF_INET or AF_INET6 in this case, corresponding to IPv4 or IPv6 respectively),
2. the socket type (SOCK_DGRAM in this case, meaning that the socket should provide connectionless and potentially unreliable transfer of datagrams), and
3. the protocol (IPROTO_UDP in this case, corresponding to UDP).

A value of 0 for the protocol requests the default for the given address family and socket type, which for `AF_INET` or `AF_INET6` and `SOCK_DGRAM` would be `IPPROTO_UDP`. It is equally acceptable for the protocol to be deduced in this manner or specified explicitly.

Assuming you previously used `getaddrinfo` to construct the remote address then the required values can be obtained from the `addrinfo` structure:

```
int fd=socket(res->ai_family,res->ai_socktype,res->ai_protocol);
if (fd==-1) {
    die("%s",strerror(errno));
}
```

## Bind the local address to the socket

As noted previously, the server socket must be bound to a local address before it can listen for inbound datagrams. This should be done using the `bind` function:

```
if (bind(fd,res->ai_addr,res->ai_addrlen)==-1) {
    die("%s",strerror(errno));
}
```

The first argument is the socket descriptor. The second and third arguments are the local address and its length.

If the local address was constructed using `getaddrinfo` then the memory occupied by the address list can now be released:

```
freeaddrinfo(res);
```

(If the address list has been searched or filtered then take care that it is the head of the list that is released, not the address that you have chosen to use.)

## Receive and handle datagrams as they arrive

Datagrams can be received using any function that is capable of reading from a file descriptor, however if you are listening for unsolicited datagrams (as in this example) then you will normally want to know where each datagram originated from so that it can be replied to. This information is provided by the functions `recvfrom` and `recvmsg`. Of these `recvmsg` is the more flexible option, but at the cost of a significantly more complex interface. Details for each function are given below.

Regardless of which function you choose you will need to supply a buffer to receive the data. If this is too small to accommodate a complete datagram then any excess is discarded. That means you need not be concerned about tracking datagram boundaries, because the first byte returned by a read operation will always be the start of a datagram. However it does raise two issues: how the buffer size should be chosen, and how any overflow can be detected.

UDP-based application-layer protocols often limit the size of datagram that can be sent in order to provide an solution to the first issue. For example, TFTP and DNS each have a fixed maximum payload size of 512 bytes. For DHCP the limit defaults to 548 bytes, but a larger value can be negotiated if both parties are willing to support it.

In the absence of such guidance it is necessary to consider what the transport, network and link layer protocols are likely to support. The maximum payload size for UDP over IPv4 is 65507 bytes, and for IPv6 with jumbogram support it is close to 4 gigabytes. However, the largest payload that an implementation is required to support is 548 bytes for IPv4 and 1452 bytes for IPv6. On an Ethernet with the standard MTU of 1500 bytes, the largest payload that can be sent without fragmentation is 1472 bytes. On this basis, 1472 bytes would be a reasonable choice if you have no reason to believe that a larger buffer is needed or that a smaller buffer would suffice.

It is possible to receive arbitrary-length datagrams with assistance from the `MSG_PEEK` option, however if you choose to do this then it would be prudent to set an upper limit in order to prevent denial of service attacks.

The `recvmsg` function explicitly reports truncation by setting the `MSG_TRUNC` flag in the `msg_flags` member

of the message header. Alternatively, truncation can be detected when using any of the available functions by providing a buffer that is one byte longer than the largest payload that you actually wish to receive, then interpreting a full buffer as a truncated datagram.

## Receive and handle datagrams as they arrive using recvfrom

To call `recvfrom` you need a buffer for the datagram and a buffer for the remote address:

```
char buffer[549];
struct sockaddr_storage src_addr;
socklen_t src_addr_len=sizeof(src_addr);
ssize_t count=recvfrom(fd,buffer,sizeof(buffer),0,(struct sockaddr*)&src_addr,&src_addr_len);
if (count==-1) {
    die("%s",strerror(errno));
} else if (count==sizeof(buffer)) {
    warn("datagram too large for buffer: truncated");
} else {
    handle_datagram(buffer,count);
}
```

The fourth argument is for specifying flags which modify the behaviour of `recvfrom`, none of which are needed in this example.

The value returned by `recvfrom` is the number of bytes received, or -1 if there was an error. Truncation is detected in this example using the technique described above of providing a slightly over-sized datagram buffer.

## Receive and handle datagrams as they arrive using recvmsg

To call `recvmsg`, in addition to buffers for the datagram and remote address you must also construct an `iovec` array and a `msghdr` structure:

```
char buffer[548];
struct sockaddr_storage src_addr;

struct iovec iov[1];
iov[0].iov_base=buffer;
iov[0].iov_len=sizeof(buffer);

struct msghdr message;
message.msg_name=&src_addr;
message.msg_namelen=sizeof(src_addr);
message.msg_iov=iov;
message.msg_iovlen=1;
message.msg_control=0;
message.msg_controllen=0;

ssize_t count=recvmsg(fd,&message,0);
if (count==-1) {
    die("%s",strerror(errno));
} else if (message.msg_flags&MSG_TRUNC) {
    warn("datagram too large for buffer: truncated");
} else {
    handle_datagram(buffer,count);
}
```

The purpose of the `iovec` array is to provide a scatter/gather capability so that the datagram payload need not be stored in a contiguous region of memory. In this example the entire payload is stored in a single buffer, therefore only one array element is needed.

The `msghdr` structure exists to bring the number of arguments to `recvmsg` and `sendmsg` down to a managable number. On entry to `recvmsg` it specifies where the source address, the datagram payload and any ancillary data should be stored. In this example no ancillary data has been requested, therefore no provision has been made for receiving any.

The `msg_flags` field of the `msghdr` structure is used by `recvmsg` to return flags to the caller. These include the `MSG_TRUNC` flag, which on exit will be set if the datagram was truncated or clear if it was not. If you wish to pass any flags into `recvmsg` then this cannot be done using `msg_flags`, which is ignored on entry. Instead you must pass them using the third argument to `recvmsg` (which is zero in this example).

# Variations

## Listening for a reply

When listening for a reply to a datagram that you have sent then three of the four steps listed above may be omitted:

- You can (and normally should) listen for the reply using the same socket from which the request was sent.
- The act of sending the request will have bound the socket to an unused port number. This will have been used as the source of the request, so should match the destination of the reply. The socket is therefore correctly bound to receive the reply.

## Connecting to a remote host

When exchanging many datagrams from a particular remote host it may be beneficial for a UDP socket to be connected to that host. This removes the need for the remote address to be explicitly checked every time a datagram is received, and for the address to be specified every time one is sent. The connection is made using the `connect` function:

```
if (connect(fd,remote_addr,sizeof(remote_addr))==-1) {
    die("%s",strerror(errno));
}
```

This is superficially identical to the call that would be made to establish a TCP connection, however unlike TCP there is no handshake. This has two notable consequences:

- Calling `connect` on a UDP socket does not (by itself) result in any network activity.
- The call to `connect` will succeed even if the remote machine is unreachable or nonexistant.

A UDP socket in the connected state will only receive datagrams that originate from the given remote address. It is therefore feasible to use functions such as `read` or `recv` in place of `recvfrom`. Similarly the given remote address becomes the default for outgoing datagrams, therefore it is feasible to use `write` or `send` in place of `sendto`. (Being connected does not, however, prevent you from sending datagrams to arbitrary destinations using `sendto` if you so wish.)

## Determining the local address

When replying to a datagram on a multihomed host, RFC 1123 recommends that the source address of the

reply should match the destination address of the corresponding request. Unfortunately the POSIX API does not provide a satisfactory way to achieve this in a portable manner. Briefly, the available options include:

- using a non-portable mechanism to obtain the address, such as `IP_RECVDSTADDR` or `IP_PKTINFO`, if one is available,
- binding a separate socket to each local IP address, having non-portably obtained a list of addresses using a mechanism such as `SIOCGIFCONF`, or
- sending the response from the wildcard address in cases where use of a maching address is non-mandatory, accepting that there are some use cases in which this will fail.

This is a substantial topic in its own right and will be the subject of a future microHOWTO.

## See also

- Send a UDP datagram in C
- Listen for and accept TCP connections in C

## Further Reading


- W. Richard Stevens *et al*, *Unix Network Programming*, Volume 1: The Sockets Networking API, 3rd edition, Addison-Wesley, 2003
- The Open Group, `recvfrom`, *Base Specifications Issue 6*
- The Open Group, `recvmsg`, *Base Specifications Issue 6*


Tags: c | posix | socket