```c
/*

An example of an Iterative Connectionless Server

This server will accept a UDP message consisting of a text string sent to it by
a client process. The sever will take the string and send it back to the client
in reverse character order.

This "echo" server receives messages on a user specified port, and will respond
to up to "max_iterations" client messages before shutting itself down.

Compile with: cc UDP_revEchod.c −o echo_serv

Usage: ./echo_serv port max_iterations

You should specify the server port number as the last four digits of your
student nubmer (as long as it is above 1024 add 1024 if it is not). This should
minimise server port number clashes with other students sharing the machine.


Note: there is minimal error checking in this example, in order to improve its
readability. Production quality code would be filled with error checking and
recovery code to make it as robust as possible. Real code doesn’t "bail out" at
the first sign of an error.

*/
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<netdb.h>
#include<stdio.h>
#include<string.h>
#include<errno.h>

#define BUF_LEN 48

/* takes instr reverses it and stores the reversed string in outstr */
int string_reverse(char *instr, char *outstr)
{
        int i, len;

        len=strlen(instr);

        for(i=0;i<len;i++)
                outstr[i]=instr[len−1−i];

        outstr[len]=’\0’;

        return len;
}



main(int argc, char *argv[])
{
        int ssd;                    /* server socket descriptor               */
        struct sockaddr_in server;  /* server address structure              */
        struct sockaddr_in client;  /* client address structure              */
        int client_len;             /* size of above client structure         */
        short echo_port;            /* servers port number                    */
        int max_iterations;         /* maximum iterations to perform          */
```

```c
        int out_cnt, in_cnt;        /* byte counts for send and receive       */
        int recv_cnt, i;            /* more counters                          */
        char client_string[BUF_LEN];/* buffer to hold send string             */
        char server_reversed_string[BUF_LEN];
                                    /* buffer to hold recieve string          */
        int ret_code;               /* generic return code holder             */


        /* Check for correct command line usage */
        if(argc!=3)
        {
                fprintf(stderr,"Usage: %s Port max_iterations\n",argv[0]);
                exit(EXIT_FAILURE);
        }

        /* Grab the command line arguments and decode them */

        echo_port=atoi(argv[1]);
        max_iterations=atoi(argv[2]);

        /* create the socket
           a socket descriptor that identifies the socket is returned,
           the socket descriptor is anagolous to a file descriptor.

           PF_INET: The Internet (TCP/IP) family.

           SOCK_DGRAM: the type of service required − datagram

           17: the UDP protocol (see /etc/protocols)
               this parameter can be used to specify which protocol in the
               family to use for the service, but for the Internet Protocol
               family only the UDP protocol supports the datagram service,
               so a 0 could have been used.

        */
        ssd=socket(PF_INET, SOCK_DGRAM, 17);
        /* if there’s a problem, report it and exit */
        if(ssd<0)
        {
                perror("While calling socket()");
                exit(EXIT_FAILURE);
        }

        /* set up the server address details in order to bind them to a
           specified socket:
            *  use the Internet (TCP/IP) address family;
            *  use INADDR_ANY, this allows the server to receive messages sent
               to any of its interfaces (Machine IP addresses), this is useful
               for gateway machines and multi−homed hosts;
            *  convet the port number from (h)ost (to) (n)etwork order, there
               is sometimes a difference.
        */

        server.sin_family=AF_INET;
        server.sin_addr.s_addr=htonl(INADDR_ANY);
        server.sin_port=htons(echo_port);

        /* bind the details in the server sockaddr_in structure to the socket */

        ret_code=bind(ssd, (struct sockaddr *)&server, sizeof(server));
        if(ret_code<0)
        {
                perror("While calling bind()");
                exit(EXIT_FAILURE);
        }

        /* Normally a server will serve forever, but this example puts a limit
           on the number of requests (max_iterations) to limit its lifetime
           so typically the for loop would be for(;;) or while(1) instead of
           whats below.
        */

        for(i=0;i<max_iterations;i++)
        {

                fprintf(stderr,"Iteration %d of %d. Waiting for client...\n",
                        i+1, max_iterations);
                client_len=sizeof(client);
                /* The following recvfrom() system call will block until a
                    message arrives from a client. The details of the client
                    will be stored in the UDP datagram will be put into the
                    client address structure, and can be used for later
                    replies to the client.
                */
                in_cnt=recvfrom(ssd, client_string, BUF_LEN, 0,
                                (struct sockaddr *)&client,
                                (socklen_t *)&client_len);
                if(in_cnt<0)
                {
                        perror("While calling recvfrom()");
                        exit(EXIT_FAILURE);
                }

                fprintf(stderr,"Message received is %d bytes long\n", in_cnt);
                fprintf(stderr,"Message received is \"%s\"\n", client_string);
                /* reverse the string */
                recv_cnt=string_reverse(client_string, server_reversed_string);


                fprintf(stderr,"Reversed string is %d bytes long\n", recv_cnt);
                fprintf(stderr,"Reversed string is \"%s\"\n",
                        server_reversed_string);

                /* send the processed data back to the client,
                   to send a string we need to include the nul on the end,
                   hence the +1
                */

                out_cnt=sendto(ssd, server_reversed_string, recv_cnt+1, 0,
                                (struct sockaddr *)&client, sizeof(client));
                if(out_cnt<0)
                {
                        perror("While calling sendto()");
                        exit(EXIT_FAILURE);
                }
                fprintf(stderr,"Client request now seviced reply sent.\n");

        }

        close(ssd);

        fprintf(stderr,"Server has shut down\n");

        return 0;
```

```
}
```