

Архангельский государственный технический университет  
Кафедра вычислительных систем и телекоммуникаций

---

Шайдо Павел Иванович  
Использование интерфейса сокетов

Архангельск 2001

## Содержание

<b>Введение</b>	<b>3</b>
<b>1 Сокеты</b>	<b>3</b>
<b>2 Структура sockaddr_in</b>	<b>4</b>
<b>3 Установка параметров сокета</b>	<b>5</b>
<b>4 Установление TCP соединения</b>	<b>6</b>
<b>5 Передача данных через TCP соединение</b>	<b>7</b>
<b>6 Обмен данными при помощи протокола UDP</b>	<b>8</b>
<b>7 Операции с сетевой базой данных</b>	<b>10</b>
7.1 Протоколы . . . . .	10
7.2 Номера портов . . . . .	10
7.3 Имена хостов . . . . .	10
<b>Приложение А. Реализация протокола daytime с использованием UDP</b>	<b>12</b>
<b>Приложение В. Реализация протокола daytime с использованием TCP</b>	<b>16</b>
<b>Приложение С. Практические задания</b>	<b>19</b>
<b>Источники</b>	<b>20</b>

## Введение

В настоящее время широко распространены технологии основанные на модели клиент-сервер. Данная технология предполагает взаимодействие процессов выполняющихся на разных системах соединенных средой передачи данных. При передаче данных по сети могут использоваться транспортные протоколы с установлением соединения, либо датаграммные протоколы. Для того, чтобы процессы могли взаимодействовать между собой, необходим механизм адресации. Данный механизм использует сетевой адрес для идентификации компьютера на котором выполняется процесс и номер порта для идентификации конкретного процесса, выполняющегося на данном компьютере. Сетевой адрес совместно с номером порта образует сокет.

Интерфейс сокетов впервые появился в операционной системе 4.2BSD. С тех пор он получил широкое распространение и реализован практически во всех операционных системах. Интерфейс сокетов обеспечивает возможность взаимодействия между процессами независимо от того, выполняются они на одном компьютере или на разных. Кроме того, интерфейс предоставляет единый набор функций для работы с различными стеками протоколов.

В данном документе описываются основные функции, предназначенные для работы с сокетами, и методы их использования. Последняя версия документа доступна по адресу <http://www.arh.ru/~zwon>.

## 1 Сокеты

Для хранения информации о сокете существует стандартная структура:

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[];
};
```

На практике, в зависимости от используемого сетевого протокола, используются другие структуры. Сокеты для использования с протоколом IP определены следующим образом:

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

Структура `in_addr` определена следующим образом:

```
struct in_addr {
    in_addr_t s_addr;
};
```

где `in_addr_t` это целый беззнаковый тип длиной 32 бита.

Сокеты создаются при помощи системного вызова `socket`:

```
int socket(int domain, int type, int protocol);
```

Аргументы функции `socket` имеют следующее значение:

`domain`

определяет коммуникационный домен. Для использования протоколов стека TCP/IP следует присвоить этому параметру значение `AF_INET`.

**type** определяет тип создаваемого сокета. Значение **SOCK\_STREAM** указывается при создании сокета работающего в режиме соединения, значение **SOCK\_DGRAM** указывается при создании сокета работающего в датаграммном режиме.

**protocol** определяет используемый протокол. Обычно этот параметр задается равным нулю, при этом используется протокол принятый по умолчанию для данного типа сокетов (TCP для сокетов типа **SOCK\_STREAM** и UDP для сокетов типа **SOCK\_DGRAM**).

Функция **socket** возвращает дескриптор файла сокета, используемый в дальнейшем для работы с сокетом. В случае возникновения ошибки функция возвращает значение -1.

Функция **socket** создает "безымянный" сокет, т.е. не связанный ни с локальным адресом, ни с номером порта. Связать сокет с адресом компьютера и номером порта можно при помощи функции **bind**:

```
int bind(int socket, const struct sockaddr *address,
        socklen_t address_len)
```

Аргументы функции **bind**:

**socket**  
является дескриптором файла сокета, который будет связан с адресом.

**address**  
указывает на структуру в которой хранится информация о сокете. Длина и формат структуры зависят от используемого семейства сетевых протоколов. В случае использования протокола IP, это структура типа **sockaddr\_in**.

**address\_len**  
содержит размер структуры **address**.

Если вызов функции **bind** завершается успешно, то возвращаемое значение равно нулю. В случае возникновения ошибки возвращается значение -1. Код ошибки содержится в переменной **errno**.

## 2 Структура **sockaddr\_in**

Структура **sockaddr\_in** описывает сокет для работы с протоколами TCP/IP. Значение поля **sin\_family** всегда равно **AF\_INET**. Поле **sin\_port** содержит номер порта который намерен занять процесс. Если значение этого поля равно нулю, то операционная система сама выделит свободный номер порта для сокета. Поле **sin\_addr** содержит IP адрес к которому будет привязан сокет. Структура **in\_addr** содержит поле **s\_addr**. Этому полю можно присвоить 32х битное значение IP адреса. Для перевода адреса в целое число из строкового представления можно воспользоваться функцией **inet\_addr**, которой в качестве аргумента передается указатель на строку содержащую IP адрес в виде четырех десятичных чисел разделенных точками. Можно, также, воспользоваться одной из следующих констант:

**INADDR\_ANY**  
все адреса локального хоста (0.0.0.0);

**INADDR\_LOOPBACK**  
адрес *loopback* интерфейса (127.0.0.1);

**INADDR\_BROADCAST**  
широковещательный адрес (255.255.255.255).

При присвоении значений номеру порта и адресу следует учитывать, что порядок следования байтов на разных архитектурах различен. При передаче данных по сети общепринятым является представление чисел в формате big-endian, в котором самый старший байт целого числа имеет наименьший адрес, а самый младший байт имеет наибольший адрес. Компьютеры построенные на архитектуре Intel x86 используют схему представления целых чисел little-endian, в которой наименьший адрес имеет самый младший байт, а наибольший адрес имеет самый старший байт. Для преобразования числа из той схемы которая используется на компьютере к той которая используется в сети, и наоборот, применяются функции:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

### 3 Установка параметров сокета

Для управления параметрами, связанными с сокетом, используют функции `setsockopt` и `getsockopt`:

```
int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
int getsockopt(int socket, int level, int option_name,
               void *option_value, socklen_t *option_len);
```

Функция `setsockopt` устанавливает параметр, заданный аргументом `option_name` на уровне протокола определенного аргументом `level`, в значение на которое указывает параметр `option_value`. Для присвоения параметра на уровне библиотеки сокетов, аргументу `level` присваивается значение `SOL_SOCKET`. Для установки параметра на другом уровне, аргументу `level` присваивается номер соответствующего протокола. На уровне библиотеки сокетов допустимыми являются следующие параметры:

**SO\_DEBUG**

Включить запись отладочной информации. Параметр имеет логическое значение.

**SO\_BROADCAST**

Разрешить отправку широковещательных пакетов (если данная возможность поддерживается используемым протоколом). Параметр имеет логическое значение.

**SO\_REUSEADDR**

Разрешает повторное использование локальных адресов (если данная возможность поддерживается используемым протоколом). Параметр имеет логическое значение.

**SO\_KEEPAIVE**

Сохраняет установленные соединения путем периодической передачи сообщений (если данная возможность поддерживается используемым протоколом). Если удаленный сокет не отвечает на сообщение, то соединение считается разорванным, процессу, осуществляющему запись в сокет, посылается сигнал `SIGPIPE`. Параметр имеет логическое значение.

**SO\_SNDBUF**

Устанавливает размер буфера отправки. Параметр имеет целое значение.

**SO\_RCVBUF**

Устанавливает размер буфера приема сообщений. Параметр имеет целое значение.

#### SO\_SNDTIMEO

Устанавливает максимальный интервал времени в течение которого функция вывода ждет завершения. Если функция, отправляющая данные, не завершается в течение указанного интервала, то она либо возвращает частичный ответ, либо, если данные отправлены не были, присваивает переменной errno значение EAGAIN или EWOULDBLOCK. По умолчанию параметр равен нулю, что означает отсутствие таймаута. Параметру присваивается значение типа `struct timeval`.

#### SO\_RCVTIMEO

Параметр аналогичен предыдущему, но устанавливает таймаут для функций ввода.

На уровне протокола TCP допустимы следующие параметры:

#### TCP\_NODELAY

Не задерживать отправку данных. Если данный параметр установлен, то отключается алгоритм буферизации. Параметр имеет логическое значение.

#### TCP\_MAXSEG

Устанавливает максимальный размер сегмента данных. Параметр имеет целое значение.

#### TCP\_NOPUSH

Не использовать проталкивание. Параметр имеет логическое значение.

#### TCP\_NOOPT

Не использовать параметры TCP. Параметр имеет логическое значение.

Параметры имеющие логическое значение являются целыми. Значение 0 обозначает, что соответствующий параметр будет отключен, значение 1 обозначает, что параметр будет включен. В случае успешного завершения функция возвращает ноль, если возникли ошибки, то результат равен -1.

Функция `getsockopt` возвращает значение указанного параметра. Помимо вышеперечисленных параметров могут использоваться следующие:

#### SO\_ERROR

Возвращает информацию о коде ошибки. Параметр имеет целое значение.

#### SO\_TYPE

Возвращает тип сокета. Параметр имеет целое значение.

## 4 Установление TCP соединения

При использовании протокола TCP в обмене данными участвуют две стороны. Одна сторона является пассивной, а другая активной. Пассивная сторона открывает некоторый TCP порт и ждет установления соединения. Схема действий пассивного процесса выглядит следующим образом:

```
socket() // Создание сокета
bind()   // Привязка сокета к номеру порта
listen() // Создание очереди соединений

while(){
    accept() // Прием запроса на установление соединения
    ...     // Обмен данными
    close()  // Закрытие соединения
}
```

Вызов `listen` включает прием соединений и ограничивает очередь входящих соединений.

```
int listen(int socket, int backlog)
```

Параметр `socket` содержит идентификатор сокета, который будет принимать соединения. Параметр `backlog` содержит длину очереди входящих запросов на установление соединения.

После приема запроса на установление соединения сервер создает новый сокет для работы с соединением. Для создания этого сокета используется вызов `accept`:

```
int accept(int socket, struct sockaddr *address,  
           socklen_t *address_len);
```

Функция `accept` извлекает первый запрос из очереди ожидающих соединений, создает новый сокет, с тем же протоколом и семейством адресов что и исходный, и возвращает дескриптор файла для этого сокета. Аргумент `socket` определяет дескриптор сокета который принимает запросы на установление соединений. Аргумент `address` либо `NULL`, либо указатель на структуру, в которую будет помещен адрес удаленного сокета после возврата из функции. `address_len` – указатель на переменную, в которой хранится длина структуры `address`.

Функция возвращает дескриптор файла сокета для установленного соединения или `-1` в случае ошибки. Полученный в результате вызова функции `accept` дескриптор файла сокета используется, в дальнейшем, для работы с установленным соединением, он не может использоваться для установления других соединений.

Схема действий активной стороны выглядит следующим образом:

```
socket()    // Создание сокета  
connect()   // Установление соединения  
...         // Обмен данными  
close()     // Закрытие соединения
```

После создания сокета пассивная сторона сразу устанавливает соединение. Для установки соединения используется функция `connect`:

```
int connect(int socket, const struct sockaddr *address,  
            socklen_t *address_len);
```

Аргумент `socket` определяет сокет, который будет использоваться для установки соединения. `address` указывает на структуру содержащую адрес сервера. `address_len` содержит длину структуры `address`. Если сокет еще не был привязан к локальному номеру порта, то функция `connect` сделает это сама. Возвращаемое значение равно нулю в случае успеха и `-1` в противном случае.

## 5 Передача данных через TCP соединение

Для обмена данными при помощи протокола TCP используются функции `send` и `recv`. Функция `send` предназначена для отправки данных:

```
ssize_t send(int socket, const void *buffer,  
             size_t length, int flags);
```

Функция выполняет передачу данных через указанный сокет партнеру. Аргумент `socket` определяет дескриптор файла сокета, через который отправляются данные. `buffer` указывает на буфер, содержащий данные для передачи. Длина передаваемых данных определяется аргументом `length`. Аргумент `flags` определяет тип передачи данных. Значение `flags` является результатом логического ИЛИ нуля или большего числа следующих констант:

**MSG\_OOB**

передать срочные данные.

**MSG\_DONTROUTE**

игнорировать параметры маршрутизации.

В случае успешного завершения **send** возвращает число переданных байт. В противном случае возвращаемое значение равно -1.

Для приема данных используется функция **recv**.

```
ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

Функция **recv** принимает данные из сокета, заданного первым аргументом. Аргумент **buffer** указывает на буфер в который будут помещены принятые данные. **length** определяет длину буфера. Аргумент **flags** определяет параметры получения данных. Значение **flags** является результатом логического ИЛИ нуля или большего числа следующих констант:

**MSG\_PEEK**

данные не удаляются из буфера приема. Следующий вызов функции **recv** прочитает те же данные.

**MSG\_OOB**

принять срочные данные.

**MSG\_WAITALL**

блокировать функцию, пока не будет принят полный объем данных, определенный аргументом **length**. Функция может вернуть меньший объем данных в случае обрыва соединения, ошибки, связанной с сокетом, использования флага **MSG\_PEEK**.

В случае успешного завершения функция возвращает число принятых байт. В противном случае возвращается -1.

## 6 Обмен данными при помощи протокола UDP

При работе с UDP сокетом для приема и передачи данных используются функции **recvfrom** и **sendto**.

```
ssize_t sendto(int socket, const void *message,
               size_t length, int flags,
               const struct sockaddr *dest_addr,
               socklen_t dest_len);
```

Функция **sendto** предназначена для отправки данных. Аргументы функции имеют следующее значение:

**socket**

сокет, через который будут отправлены данные.

**message**

указатель на буфер, содержащий данные для отправки.

**length**

определяет длину сообщения в байтах.

**flags** определяет параметры передачи сообщения. Значение **flags** является результатом логического ИЛИ нуля или большего числа следующих констант:

**MSG\_OOB**

передать срочные данные (не поддерживается протоколом UDP).



**MSG\_DONTROUTE**

игнорировать параметры маршрутизации.

**dest\_addr**

указатель на структуру, содержащую адрес получателя.

**dest\_len**

определяет длину структуры, на которую указывает **dest\_addr**

Функция возвращает число переданных байт в случае успешного завершения и -1 в противном случае. Следует заметить, что успешное выполнение функции **sendto** не гарантирует доставку данных получателю. Возврат значения -1 происходит только в случае локально обнаруженных ошибок.

```
ssize_t recvfrom(int socket, void *buffer,  
                 size_t length, int flags,  
                 struct sockaddr *address,  
                 socklen_t *address_len);
```

Функция **recvfrom** принимает данные из сокета. Аргументы функции имеют следующее значение:

**socket**

сокет из которого производится чтение данных.

**buffer**

указатель на буфер, в который будут помещены данные.

**length**

определяет длину буфера, на который указывает аргумент **buffer**.

**flags** определяет параметры приема данных. Значение **flags** является результатом логического ИЛИ нуля или большего числа следующих констант:

**MSG\_PEEK**

оставить принятые данные в буфере приема. Следующий вызов **recvfrom** получит те же данные.

**MSG\_OOB**

принимать только срочные данные (не поддерживается протоколом UDP).

**MSG\_WAITALL**

блокировать функцию, пока не будет принят полный объем данных, определенный аргументом **length**. Функция может вернуть меньший объем данных в случае обрыва соединения, ошибки, связанной с сокетом, использования флага **MSG\_PEEK**.

**address**

указатель на структуру, в которую будет помещен адрес отправителя.

**address\_len**

определяет длину структуры, на которую указывает **address**.

Функция возвращает количество данных, записанных в буфер. Если при выполнении функции возникли ошибки, то возвращается значение -1. Для протокола UDP, данные, пришедшие в одном пакете, должны быть прочитаны одним вызовом функции **recvfrom**. Если длина буфера недостаточна для размещения всех данных, то лишние байты отбрасываются.

## 7 Операции с сетевой базой данных

IP адреса, номера портов, идентификаторы протоколов являются числами. Для человека более удобно иметь дело с символьными обозначениями. Для преобразования чисел в соответствующие им символьные строки и наоборот имеется ряд стандартных функций.

### 7.1 Протоколы

Для работы с протоколами используется структура `protoent`:

```
struct protoent {
    char    *p_name; /* Название протокола */
    char    **p_aliases; /* Массив указателей на альтернативные
                           имена протокола */
    int     p_proto; /* Номер протокола */
};
```

Для получения информации о протоколе по его названию или номеру используются функции:

```
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
```

В случае возникновения ошибок функции возвращают `NULL`.

### 7.2 Номера портов

Для преобразования номеров портов используется структура `servent`:

```
struct servent {
    char    *s_name; /* Имя сервиса */
    char    **s_aliases; /* Альтернативные имена сервиса */
    int     s_port; /* Номер порта занимаемый сервисом */
    char    *s_proto; /* Имя протокола используемого сервисом */
}
```

Для получения информации о сервисе по номеру порта и наоборот используются, соответственно, функции:

```
struct servent *getservbyport(int port, const char *proto);
struct servent *getservbyname(const char *name, const char *proto);
```

В случае возникновения ошибок функции возвращают `NULL`.

### 7.3 Имена хостов

Для получения информации о хостах определена структура `hostent`:

```
struct hostent {
    char    *h_name; /* Официальное имя хоста */
    char    **h_aliases; /* Массив псевдонимов хоста */
    int     h_addrtype; /* Тип адреса (обычно AF_INET) */
    int     h_length; /* Длина адреса в байтах */
    char    **h_addr_list; /* Список адресов хоста */
}
```

Функция `gethostbyname` позволяет получить адрес хоста по его имени:

```
struct hostent *gethostbyname(const char *name);
```

Функция `gethostbyaddr` позволяет определить имя хоста по его адресу. В качестве аргументов функции передаются указатель на адрес хоста, длина адреса и его тип (`AF_INET` для IPv4):

```
struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
```

В случае возникновения ошибок функции возвращают `NULL`. Код ошибки помещается в переменную `h_errno`.

## Приложение А. Реализация протокола Daytime с использованием UDP

Протокол daytime определен в документе RFC867. Протокол может использоваться в качестве транспортного протокола UDP и TCP. В случае использования UDP сервер занимает 13-й порт и ожидает поступления датаграмм. После получения датаграммы он отправляет назад строку содержащую текущие дату и время в произвольном формате.

Ниже приведен пример реализации серверной части daytime:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <time.h>
#include <string.h>

main(){
    int s, clen, rd, proto;
    struct sockaddr_in saddr, caddr;
    struct sockaddr *sa, *ca;
    struct hostent *rhost;
    time_t itime;
    char buf[2048], *tstr, *host;

    sa=&saddr;
    ca=&caddr;

    // Получаем номер протокола UDP
    proto=getprotobyname("udp")->p_proto;

    // Создаем сокет
    s=socket(PF_INET, SOCK_DGRAM, proto);
    if(s<0) {
        perror("udps: не удастся создать сокет");
        exit(1);
    }

    // Резервируем порт 13
    saddr.sin_family=AF_INET;
    saddr.sin_addr.s_addr=INADDR_ANY;
    saddr.sin_port=htons(13);

    if(bind(s, sa, sizeof(saddr))== -1) {
        perror("udps: не удастся занять порт");
        exit(1);
    }

    caddr.sin_family=AF_INET;
    clen=sizeof(caddr);
```

```

while(1) {

    // Ожидаем поступления запроса
    rd=recvfrom(s, buf, 1, 0, ca, &clen);
    if(rd==-1){
        perror("udps: ошибка при получении данных");
        exit(1);
    }

    // Преобразуем адрес хоста отправителя в его имя
    rhost=gethostbyaddr((char*)&caddr.sin_addr,
        sizeof(caddr.sin_addr), AF_INET);
    if(h_errno){
        printf("gethostbyaddr error: %d\n", h_errno);
        host=inet_ntoa(caddr.sin_addr);
    }
    else{
        host=rhost->h_name;
    }

    // Получаем строку содержащую дату и время
    itime=time(NULL);
    tstr=ctime(&itime);

    // Выводим время поступления запроса,
    // адрес и порт отправителя
    printf("%s request from %s:%d\n", tstr, host,
        htons(caddr.sin_port));

    // Отправляем дату и время клиенту
    sendto(s, tstr, strlen(tstr), 0, ca, sizeof(caddr));

}
}

```

Реализация клиентской части приведена ниже. Клиент устанавливает ограничение на время ожидания поступления данных, посылает широковещательный запрос и ожидает поступления ответа. Получив ответ клиент выводит его на экран и ожидает поступления других ответов. Если функция `recv` завершается с ошибкой превышения времени ожидания ответа, то клиент считает что все ответы получены и завершает выполнение.

```

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

main(){
    int s, so, clen, rd, proto;

```

```

struct sockaddr_in saddr, caddr;
struct sockaddr *sa, *ca;
struct hostent *rhost;
struct timeval timeout;
char buf[100], *host;

sa=&saddr;
ca=&caddr;

// Получаем номер протокола UDP
proto=getprotobyname("udp")->p_proto;

// Создаем сокет
s=socket(AF_INET, SOCK_DGRAM, proto);
if(s<0) {
    perror("udpc: не удается создать сокет");
    exit(1);
}

// Разрешаем отправку широковещательных пакетов
so=1;
rd=setsockopt(s, SOL_SOCKET, SO_BROADCAST, &so, sizeof(so));
if(rd== -1) {
    perror("udpc: не удается установить параметры сокета");
    exit(1);
}

// Устанавливаем предельное время ожидания ответа
timeout.tv_sec=3;
timeout.tv_usec=0;
rd=setsockopt(s, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));
if(rd== -1) {
    perror("udpc: не удается установить параметры сокета");
    exit(1);
}

// Резервируем порт
caddr.sin_family=AF_INET;
caddr.sin_addr.s_addr=INADDR_ANY;
caddr.sin_port=0;

if(bind(s, ca, sizeof(caddr))== -1) {
    perror("udpc: не удается занять порт");
    exit(1);
}

// Задаем адрес получателя
saddr.sin_family=AF_INET;
saddr.sin_port=htons(13);
saddr.sin_addr.s_addr=INADDR_BROADCAST;
clen=sizeof(saddr);

```

```

// Отправляем запрос
rd=sendto(s, buf, 1, 0, sa, clen);
if(rd==-1){
    perror("udprc: ошибка при отправке запроса");
    exit(1);
}

while(1){
    // Ожидаем ответ
    rd=recvfrom(s ,buf ,99 ,0 ,sa , &clen);
    if(rd==-1){
        // Если превышено время ожидания ответа, то выход
        if(errno==EAGAIN) break;
        // Иначе ошибка
        perror("udprc: ошибка при получении ответа");
        exit(1);
    }
    buf[rd]=(char)0;

    // Преобразуем адрес хоста отправителя в его имя
    rhost=gethostbyaddr((char*)&saddr.sin_addr,
        sizeof(saddr.sin_addr), AF_INET);
    if(h_errno){
        printf("gethostbyaddr error: %d",h_errno);
        host=inet_ntoa(caddr.sin_addr);
    }
    else{
        host=rhost->h_name;
    }

    // Выводим информацию о поступившем ответе
    printf("%s - reply from %s\n", buf, host);
}
}

```

## Приложение В. Реализация протокола Daytime с использованием TCP

Теперь рассмотрим реализацию протокола daytime при помощи протокола TCP. Как и в UDP, сервер занимает порт 13 и ожидает поступления запросов на установление соединения. После установления соединения, сервер отправляет клиенту строку, содержащую дату и время, и закрывает соединение. Реализация сервера приведена ниже:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

main() {
    int s, c, sz;
    struct sockaddr_in ssa, csa;
    struct sockaddr *sp, *cp;
    struct hostent *rhost;
    char *host, *tstr;
    time_t itime;

    sp=(struct sockaddr *)&ssa;
    cp=(struct sockaddr *)&csa;
    sz=sizeof(ssa);

    // Создаём сокет
    s=socket(AF_INET, SOCK_STREAM, 0);
    if(s == -1){
        perror("Невозможно создать сокет");
        exit(1);
    }

    // Резервируем порт 13
    ssa.sin_family = AF_INET;
    ssa.sin_port = htons(13);
    ssa.sin_addr.s_addr = INADDR_ANY;

    if(bind(s, sp, sz) == -1){
        perror("Невозможно занять порт");
        exit(1);
    }

    // Переводим сокет в режим ожидания соединения
    if(listen(s, 0) == -1){
        perror("Невозможно перейти в режим ожидания");
        exit(1);
    }

    while(1){
        // Принимаем соединение
```



```

        if((c = accept(s, cp, &sz)) == -1) {
            perror("Ошибка при выполнении accept");
            exit(1);
        }

        // Преобразуем адрес хоста отправителя в его имя
        rhost=gethostbyaddr((char*)(&csa.sin_addr),
                            sizeof(csa.sin_addr), AF_INET);
        if(h_errno){
            printf("gethostbyaddr error: %d\n", h_errno);
            host=inet_ntoa(csa.sin_addr);
        } else {
            host=rhost->h_name;
        }

        // Получаем строку, содержащую дату и время
        if((itime = time(NULL)) < 0){
            perror("Не удалось получить время");
            exit(1);
        }
        tstr = ctime(&itime);

        // Выводим время поступления запроса,
        // адрес и порт отправителя
        printf("%s request from %s:%d\n", tstr, host,
              htons(csa.sin_port));

        // Отправляем дату и время клиенту
        send(c, tstr, strlen(tstr), 0);

        // Закрываем соединение
        close(c);
    }
}

```

Теперь рассмотрим реализацию клиента. Обратите внимание, что клиент не должен выполнять вызов функции bind, порт выделяется автоматически при выполнении connect.

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define BUFSZ 128

main(int argc, char *argv[]) {
    int s, sz, i;
    struct sockaddr_in ssa;
    struct sockaddr *sp;
    struct in_addr sip;
    char buf[BUFSZ];

```

```

sp=(struct sockaddr *)&ssa;
sz=sizeof(ssa);

if(argc!=2){
    // Помощь по использованию команды
    printf("Использование: %s ip-адрес\n",argv[0]);
    exit(1);
}
if(inet_aton(argv[1], &sip) != 1){
    printf("Неправильно задан адрес сервера\n");
    exit(1);
}
// Создаём сокет
s=socket(AF_INET, SOCK_STREAM, 0);
if(s == -1){
    perror("Невозможно создать сокет");
    exit(1);
}

// Задаём адрес сервера
ssa.sin_family = AF_INET;
ssa.sin_port = htons(13);
ssa.sin_addr = sip;

// Устанавливаем соединение
if(connect(s, sp, sz) == -1){
    perror("Не удалось установить соединение");
    exit(1);
}

// Получаем данные от сервера
while((i=recv(s, buf, BUFSZ, 0)) > 0)
    write(1, buf, i);
}

```

## Приложение С. Практические задания

1. Напишите программу, которая занимает заданный порт TCP, ожидает установления соединения, принимает данные, переводит все символы латинского алфавита в верхний регистр и отправляет данные обратно.
2. Напишите программу, которая занимает заданный порт UDP, принимает пакет, содержащий имя хоста, переводит имя в IP адрес и отправляет обратно этот адрес в виде текстовой строки.
3. Напишите программу, которая при запуске занимает произвольный порт UDP, запрашивает имя пользователя, а затем принимает вводимые пользователем строки, добавляет в их начало имя пользователя и широковещательно рассылает на порт 5400 UDP.
4. Напишите программу, которая занимает порт 5400 UDP, принимает пакеты и выводит их содержимое на экран.
5. Напишите программу, которая запрашивает у пользователя IP адрес, номер порта и строку текста, отправляет строку по указанному адресу, используя протокол UDP, ожидает поступления ответа и выводит его на экран. Если в течение 5 сек. ответ не получен, то программа начинает своё выполнение сначала.
6. Напишите программу, которая ожидает установления соединения на заданном порту TCP, принимает поступающие данные и сохраняет их в файле, имя которого определяется IP адресом и номером порта клиента.
7. Напишите программу, которая запрашивает у пользователя IP адрес, номер порта и имя файла, устанавливает TCP соединение с указанным адресом, и передает заданный файл.
8. Напишите программу, которая при запуске устанавливает TCP соединение с заданным сервером и открывает в пассивном режиме заданный порт TCP. После подключения к этому порту клиента, программа в цикле принимает данные от клиента, передаёт их серверу, получает ответ сервера и возвращает его клиенту. После отключения клиента программа должна завершиться.
9. Напишите программу, которой в качестве аргументов передаются IP адрес хоста и диапазон портов. Программа должна вывести номера открытых портов TCP из указанного диапазона.
10. Напишите программу, которой в качестве аргументов передаются IP адрес хоста и диапазон портов. Программа должна вывести номера открытых портов UDP из указанного диапазона. Для обнаружения открытого порта UDP следует установить с ним соединение при помощи функции `connect` и отправить данные. Если порт закрыт, то при повторной попытке отправить данные функция `send` завершится с ошибкой, код ошибки `ECONNREFUSED` будет помещён в переменную `errno`.

## **Источники**

1. Кейт Хэвиленд, Дайна Грэй, Бен Салама. Системное программирование в UNIX. Руководство программиста по разработке ПО. Пер. с англ. — М., ДМК Пресс, 2000. — 368 с., ил.
2. The Single UNIX Specification, version 2. The Open Group. 1997.
3. Mark Burgess. The UNIX programming environment. Edition 2.1, Feb 1999.  
(<http://www.iu.hio.no/~mark/unix/unix.html>).