

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**



ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №1

по дисциплине: Компьютерные сети
тема: «Протокол сетевого уровня IPX»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Белгород 2025 г.

Лабораторная работа №1

Протокол сетевого уровня IPX

Вариант 6

Цель работы: изучить протокол сетевого уровня IPX, основные функции API драйвера IPX и разработать программу для приема/передачи данных.

Краткие теоретические сведения

Протокол **IPX** – это протокол сетевого уровня модели взаимодействия открытых систем (OSI), реализующий передачу пакетов (сообщений) между станциями сети на уровне датаграмм.

Датаграмма – это сообщение, доставка которого получателю не гарантируется.

Номер сети - это номер сегмента сети, определяемого системным администратором.

Адрес станции - это число, которое является уникальным для каждой рабочей станции. При использовании адаптеров Ethernet уникальность обеспечивается изготовителем сетевого адаптера. Специальный адрес FFFFFFFFh используется для рассылки данных всем станциям данной сети одновременно.

Идентификатор программы на рабочей станции (сокет) - число, которое используется для адресации определенной программы, работающей на компьютере. В среде мультзадачных операционных систем, на каждой рабочей станции в сети одновременно может быть запущено несколько приложений.

Формат передаваемых с использованием протокола IPX по сети пакетов представлен на рис. 1.1.

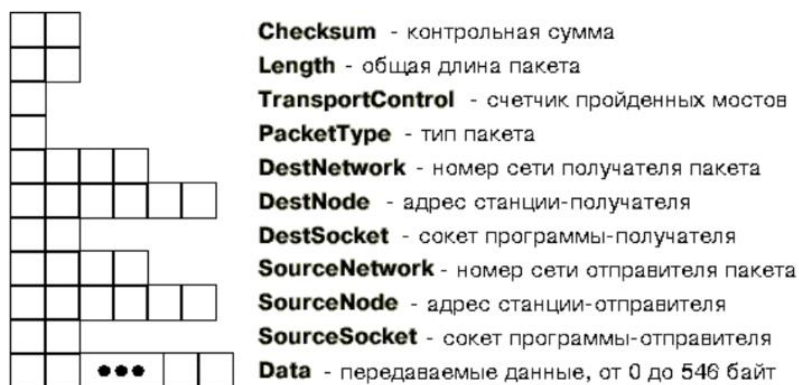


Рис. 1.1. Структура пакета IPX

- **Checksum** предназначено для хранения контрольной суммы передаваемых пакетов. При формировании пакетов не нужно заботиться о содержимом этого поля, так как проверка данных по контрольной сумме выполняется драйвером сетевого адаптера.
- **Length** определяет общий размер пакета. Длина заголовка фиксирована и равна 30 байт. Размер передаваемых в поле Data данных может составлять от 0 до 546 байт. При формировании пакетов не нужно проставлять длину пакета в поле Length, протокол IPX делает это сам.
- **TransportControl** является счетчиком мостов, которые проходит пакет на своем пути от передающей станции к принимающей. Каждый раз, когда пакет проходит

через мост, значение этого счетчика увеличивается на единицу. IPX перед передачей пакета сбрасывает содержимое этого поля в нуль. Так как IPX сам следит за содержимым этого поля, при формировании пакетов не нужно изменять или устанавливать его в какое-либо состояние.

- **PacketType** определяет тип передаваемого пакета. Для IPX следует установить значение равное 4.
- **DestNetwork** определяет номер сети, в которую передается пакет.
- **DestNode** определяет адрес рабочей станции, которой предназначен пакет.
- **DestSocket** предназначено для адресации программы, запущенной на рабочей станции, которая должна принять пакет.
- Поля **SourceNetwork**, **SourceNode** и **SourceSocket** содержат соответственно номер сети, из которой посылается пакет, адрес передающей станции и сокет программы, передающей пакет.
- **Data** содержит передаваемые данные.

Прикладные программы все свои запросы на прием и передачу пакетов направляют драйверу IPX, который, в свою очередь, обращается к драйверу сетевого адаптера. Для приема или передачи пакета прикладная программа должна подготовить пакет данных, сформировав его заголовок, и построить так называемый блок управления событием ECB (Event Control Block).

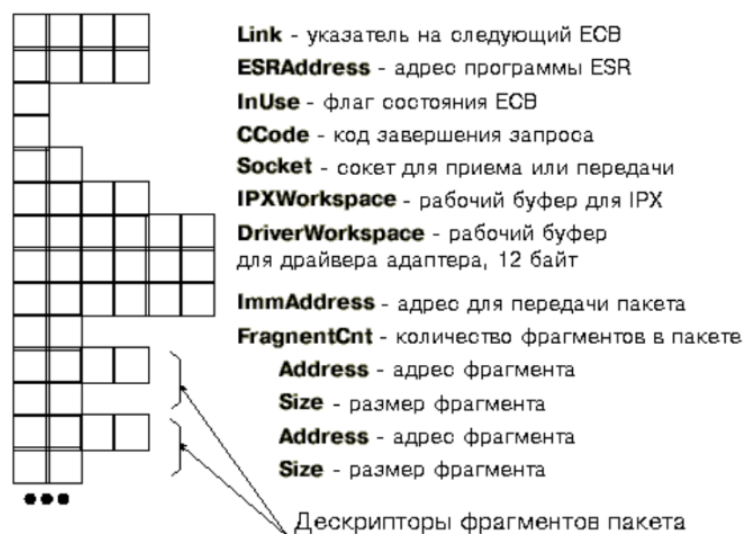


Рис. 1.2. Формат блока ECB

Блок ECB состоит из фиксированной части размером 36 байт и массива дескрипторов, описывающих отдельные фрагменты передаваемого или принимаемого пакета данных.

- **Link** предназначено для организации списков, состоящих из блоков ECB. Драйвер IPX использует это поле для объединения переданных ему блоков ECB в списки, записывая в него полный адрес в формате [сегмент: смещение]. После того, как IPX выполнит данную ему команду и закончит все операции над блоком ECB, программа может распоряжаться полем Link по своему усмотрению.
- Поле ESRAddress содержит полный адрес программного модуля (в формате [сегмент: смещение]), который получает управление при завершении процесса чтения или передачи пакета IPX. Этот модуль называется программой обслуживания события ESR (Event Service Routine). Если программа не использует

ESR, она должна записать в поле ESRAddress нулевое значение. В этом случае о завершении выполнения операции чтения или передачи можно узнать по изменению содержимого поля InUse. **Важно! Внутри ESRAddress недоступны прерывания – значит нельзя будет вызвать другие функции. Кроме того, программа по ESRAddress не должна заботиться о сохранении регистров и поддержания их целостности.**

- **InUse** может служить индикатором завершения операции приема или передачи пакета. Перед тем как вызвать функцию IPX, программа записывает в поле InUse нулевое значение. Пока операция передачи данных, связанная с данным ECB, не завершилась, поле
 - **FFh - ECB** используется для передачи пакета данных;
 - **FEh - ECB** используется для приема пакета данных, предназначенного программе с конкретным сокетом;
 - **FDh - ECB** используется функциями асинхронного управления событиями AES (Asynchronous Event Sheduler), ECB находится в состоянии ожидания истечения заданного временного интервала;
 - **FBh** - пакет данных принят или передан, но ECB находится во внутренней очереди IPX в ожидании завершения обработки.
- Программа может постоянно опрашивать поле InUse, ожидая завершения процесса передачи или приема данных. Как только в этом поле окажется нулевое значение, программа считает, что запрошенная функция выполнена. Результат выполнения можно получить в поле **CCode**, где после выполнения функции IPX содержится код результата выполнения.
 - **00** - пакет был принят без ошибок;
 - **FFh** - указанный в ECB сокет не был предварительно открыт программой;
 - **FDh** - переполнение пакета: либо поле количества фрагментов в пакете FragmentCnt равно нулю, либо буферы, описанные дескрипторами фрагментов, имеют недостаточный размер для записи принятого пакета;
 - **FCh** - запрос на прием данного пакета был отменен специальной функцией драйвера IPX. Если ECB использовался для передачи пакета, в поле CCode после завершения передачи могут находиться следующие значения:
 - **00** - пакет был передан без ошибок (но это не означает, что пакет был доставлен по назначению и успешно принят станцией-адресатом);
 - **FFh** - пакет невозможно передать физически из-за неисправности в сетевом адаптере или в сети;
 - **FEh** - пакет невозможно доставить по назначению, так как станция с указанным адресом не существует или неисправна;
 - **FDh** - пакет сбойный, т.е. либо имеет длину меньше 30 байт, либо первый фрагмент пакета по размеру меньше размера стандартного заголовка пакета IPX, либо поле количества фрагментов в пакете FragmentCnt равно нулю;
 - **FCh** - запрос на передачу данного пакета был отменен специальной функцией драйвера IPX.
- **Socket** содержит номер сокета, связанный с данным ECB. Если ECB используется для приема, это поле содержит номер сокета, на котором выполняется прием пакета. Если же ECB используется для передачи, это поле содержит номер сокета передающей программы.
- **IPXWorkspace** зарезервировано для использования драйвером IPX. Приложение не должно инициализировать или изменять содержимое этого поля, пока обработка ECB не завершена.

- **DriverWorkspace** зарезервировано для использования драйвером сетевого адаптера. Программа не должна инициализировать или изменять содержимое этого поля, пока обработка ECB не завершена.
- **ImmAddress** (Immediate Address - непосредственный адрес) содержит адрес узла в сети, в который будет направлен пакет. Если пакет передается в пределах одной сети, поле ImmAddress будет содержать адрес станции-получателя (такой же, как и в заголовке пакета IPX). Если же пакет предназначен для другой сети и будет проходить через мост, поле ImmAddress будет содержать адрес этого моста в сети, из которой передается пакет.
- **FragmentCnt** содержит количество фрагментов, на которые нужно разбить принятый пакет, или из которых необходимо собрать передаваемый пакет. Механизм фрагментации позволяет избежать пересылок данных или непроизводительных потерь памяти. Можно указать отдельные буферы для приема данных и заголовка пакета.

Если принимаемые данные имеют какую-либо структуру, можно рассредоточить отдельные блоки по соответствующим буферам.

Значение, записанное в поле FragmentCnt, не должно быть равно нулю. Если в этом поле записано значение 1, весь пакет вместе с заголовком записывается в один общий буфер.

- Далее располагаются дескрипторы фрагментов, состоящие из указателя в формате [сегмент : смещение] на фрагмент **Address** и поля размера фрагмента **Size**.

Основные функции API для работы с протоколом IPX

Для того чтобы проверить, загружен ли драйвер IPX и доступно ли API, необходимо загрузить в регистр AX значение 7A00h и вызвать мультиплексное прерывание INT 2Fh. Если после возврата из прерывания в регистре AL будет значение FFh, то драйвер IPX загружен. Адрес точки входа для вызова API драйвера при этом будет находиться в регистровой паре ES:DI. Если же содержимое регистра AL после возврата из прерывания INT 2Fh будет не равно FFh, то драйвер IPX/SPX не загружен. Это означает, что на данной рабочей станции не загружены резидентные программы ipx.exe или ipxodi.exe, обеспечивающие API для работы с протоколами IPX и SPX.

Для работы с протоколом необходимо создать сервер IPX на рабочей станции, работающей на ОС DOS (или в эмуляторе среды DOS), командой терминала ipxnet startserver. Далее на других компьютерах или в других эмуляторах системы, которые находятся в одной сети с созданным сервером (даже если в дальнейшем будет запущена программа-сервер), необходимо выполнить команду терминала ipxnet connect <адрес сервера>. В случае работы на одном компьютере адресом сервера является localhost (или 127.0.0.1).

Для вызова API в регистр BX необходимо загрузить код выполняемой функции. Значения, загружаемые в другие регистры, зависят от выполняемой функции. Для работы с IPX используются следующие основные функции.

- **IPXOpenSocket** предназначена для получения сокетов.

На входе BX = 00h, AL = Тип сокета: 00h - короткоживущий; FFh - долгоживущий,

DX = Запрашиваемый номер сокета или 0000h, если требуется получить динамический номер сокета. Байты номера сокета находятся в перевернутом виде.

На выходе **AL** = Код завершения (00h - сокет открыт; FFh – этот сокет уже был открыт раньше; FEh - переполнилась таблица сокетов), **DX** = Присвоенный номер сокета.

- **IPXCloseSocket** предназначена для освобождения сокетов, т.к. сокеты являются ограниченным ресурсом.

На входе **BX** = 01h, **DX** = Номер закрываемого сокета.

На выходе регистры не используются.

- **IPXGetLocalTarget** применяется для вычисления значения непосредственного адреса, помещаемого в поле **ImmAddress** блока **ECB** перед передачей пакета.

На входе **BX** = 09h, **ES:SI** = Указатель на буфер длиной 10 байт, в который будет записан адрес станции, на которой работает данная программа. Адрес состоит из номера сети и адреса станции в сети.

На выходе регистры не используются.

- Станция-получатель может находиться в другой сети, поэтому прежде чем достигнуть цели, пакет может пройти один или несколько мостов. Поле непосредственного адреса **ImmAddress** блока **ECB** должно содержать либо адрес станции назначения (если передача происходит в пределах одной сети), либо адрес моста (если пакет предназначен для рабочей станции, расположенной в другой сети). Используя, указанный в буфере размером 12 байт, полный сетевой адрес, состоящий из номера сети, адреса станции в сети и сокета приложения, функция **IPXGetLocalTarget** вычисляет непосредственный адрес, т.е. адрес той станции в данной сети, которая получит передаваемый пакет.
- **IPXListenForPacket** предназначена для начала процесса приема пакетов данных из сети.

На входе **BX** = 04h, **ES:SI** = Указатель на заполненный блок **ECB**. На выходе регистры не используются.

Сразу после вызова функции **IPXListenForPackets** в поле **InUse** блока **ECB** устанавливается значение FEh, которое означает, что для данного блока **ECB** ожидается прием пакета. После прихода пакета в поле **InUse** блока **ECB** устанавливается значение 0h и вызывается программа **ESR**. Если ее адрес был задан перед вызовом функции **IPXListenForPacket**, в поле **CCode** блока **ECB** драйвер **IPX** записывает код результата приема пакета, а в поле **ImmAddress** - непосредственный адрес станции, из которой пришел пакет. Если пакет пришел из другой сети, в этом поле будет стоять адрес моста.

- **IPXSendPacket** подготавливает блок **ECB** и связанный с ним заголовок пакета для передачи пакета по сети.

На входе **BX** = 03h, **ES:SI** = Указатель на заполненный блок **ECB**.

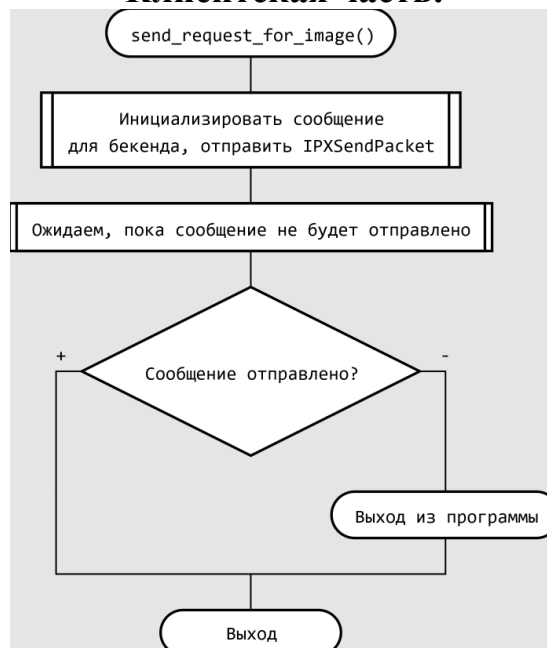
На выходе регистры не используются.

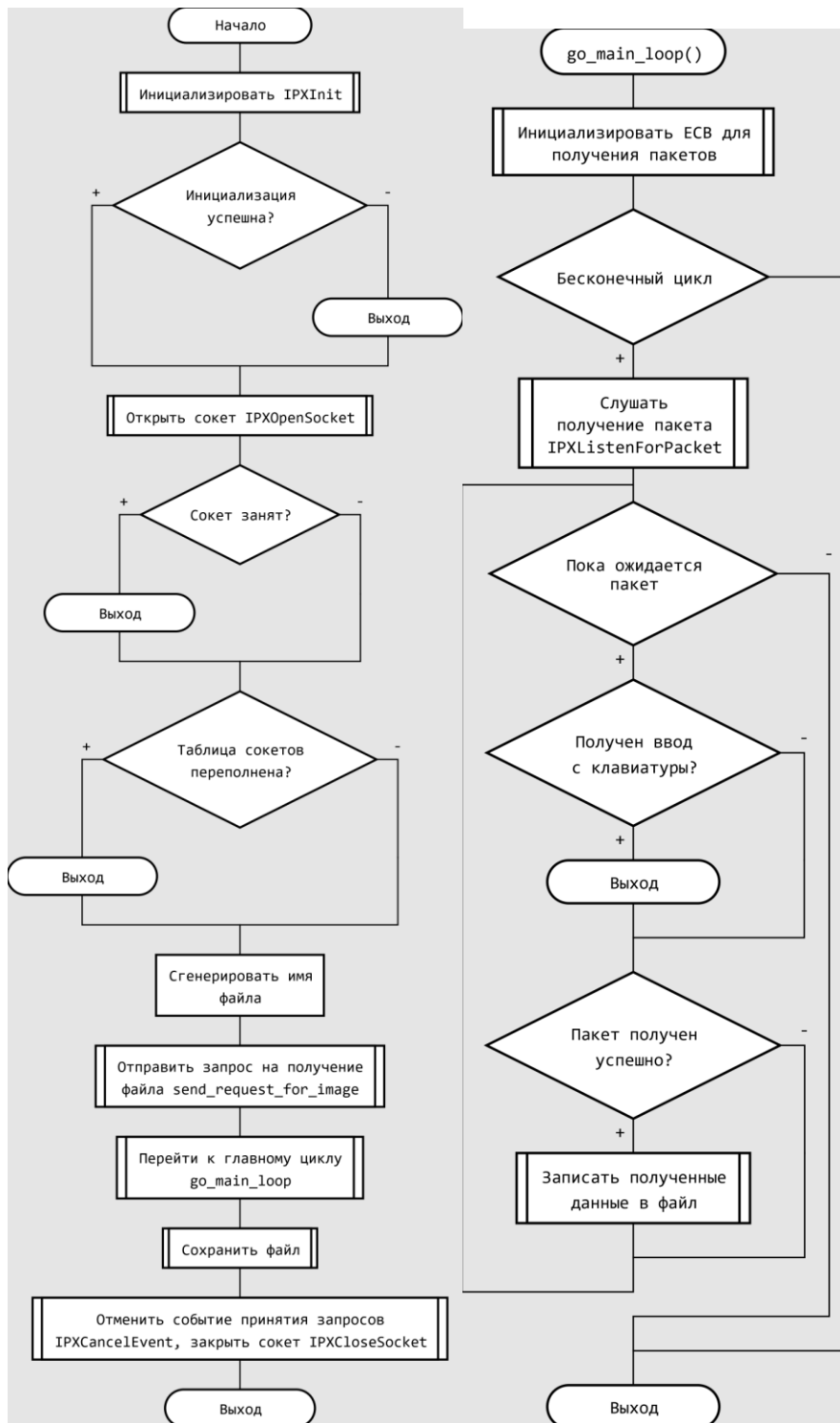
Сразу после вызова функции `IPXSendPacket`, IPX-служба ставит блок ECB в очередь на передачу и возвращает управление вызвавшей ее программе, не дожидаясь прихода пакета, в поле `InUse` блока ECB устанавливается значение `FFh`. Сама передача пакета происходит асинхронно по отношению к вызывавшей ее программе. После завершения процесса передачи пакета в поле `InUse` записывается значение `0h`, в поле `CCode` блока ECB – код результата приема пакета. Результат выполнения передачи пакета можно узнать, проанализировав поле `CCode` блока ECB.

- **IPXRelinquishControl** (`BX = 0Ah`) выделяет драйверу IPX процессорное время, необходимое для его правильной работы. Если программа не использует ESR, она, должна в цикле опрашивать поле `InUse` блока ECB, для которого выполняется ожидание завершения процесса приема или передачи пакета. Однако для правильной работы драйвера IPX в цикл ожидания необходимо вызывать эту функцию.
- **IPXCancelEvent** отменяет ожидание приема или отправки пакета, связанного с блоком ECB. Ее рекомендуется выполнять при аварийном выходе из программы для каждого подготовленного, но ещё неиспользованного блока ECB.

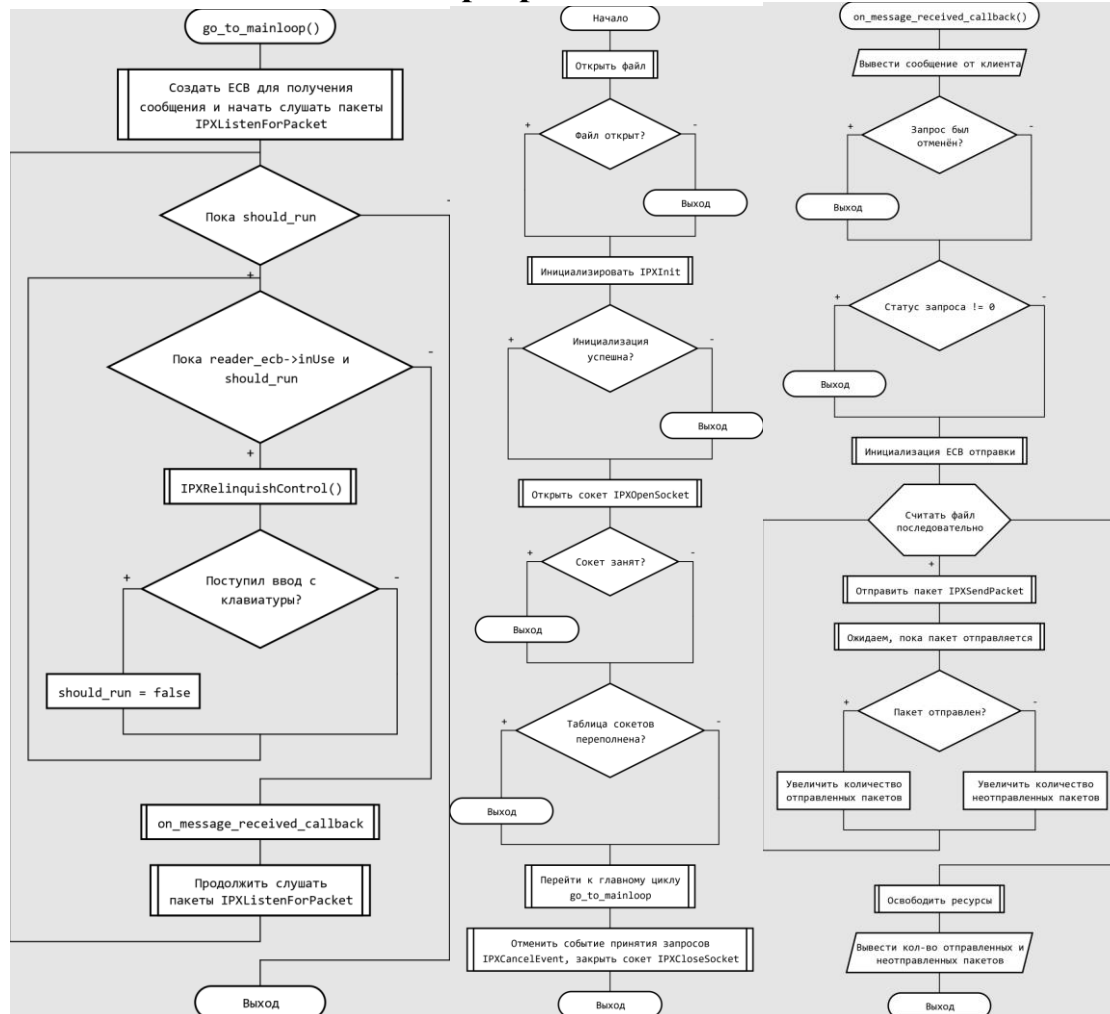
Разработка программы. Блок-схемы программы.

Клиентская часть:





Серверная часть:



Анализ функционирования программ

Для отправки изображения размером 1.51 Мб потребовалось отправить 3949 пакета, каждый из пакетов имеет 400 байт полезной информации – самого изображения. Для отправки одного изображения потребовалось около 12-13 секунд (что довольно много по сегодняшним меркам).

Сервер обладает возможностью повторной отправки файла, что позволяет не перезапускать его каждый раз.

Протокол IPX не гарантирует доставки сообщений. Так как тестирование велось в локальной сети в пределах одного компьютера, упущенных пакетов не было допущено. Был проведён эксперимент, позволяющий просимулировать нестабильную сеть: что если с сервера пропускать каждый 3 или каждый 4 пакет? Ожидается, изображения были доставлены с ошибками и в них присутствовали артефакты. Протокол IPX не подходит для передачи файлов без потерь.



Вывод: в ходе лабораторной изучили протокол сетевого уровня IPX, основные функции API драйвера IPX и разработать программу для приема/передачи данных.

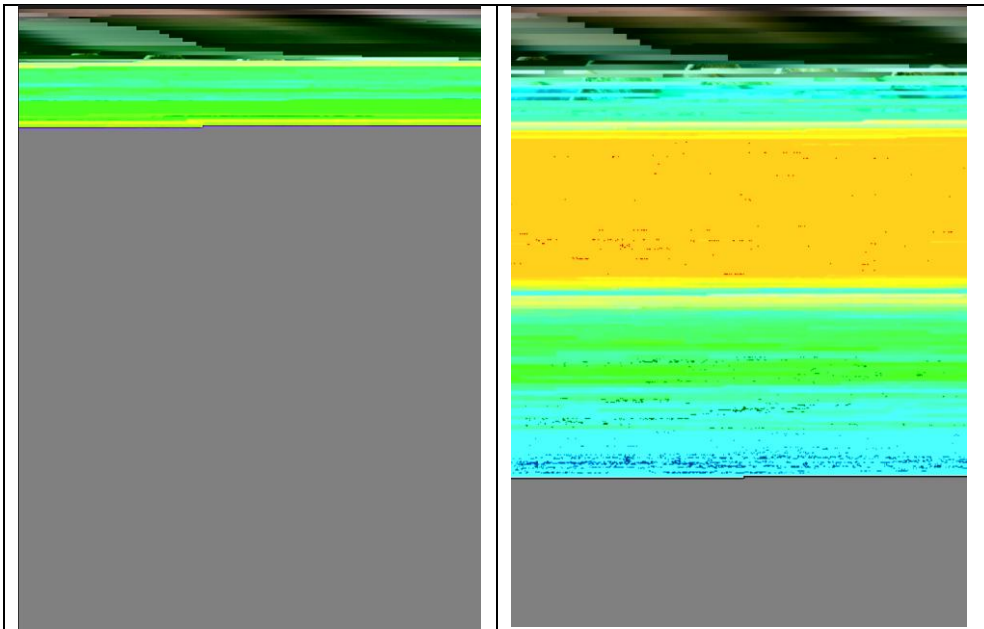
Текст программ. Скриншоты программ.

Ссылка на репозиторий с кодом: https://github.com/IAmProgrammist/comp_net/tree/lab1

```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Fram...
C:\LAB1>server
Socket successfully opened at: 16387 (4003 in hex)
Awaiting requests...
A new request for an image received
A message: I need cat picture pls ...
Package sending set up complete
Request was obtained
Total packages: 3949
Succeeded packages: 3949
Failed packages: 0
Awaiting requests...
A new request for an image received
A message: I need cat picture pls ...
Package sending set up complete
Request was obtained
Total packages: 3949
Succeeded packages: 3949
Failed packages: 0
Awaiting requests...

DOSBox 0.74-3, Cpu speed: 3000 cycles, Fram...
Socket successfully opened at: 16386 (4002 in hex)
Input server socket: 16386
Sending request to server...
A request is accepted successfully
Starting file accepta file was saved as img84.jpg
C:\LAB1>
C:\LAB1>client
Socket successfully opened at: 16386 (4002 in hex)
Input server socket: 16387
Sending request to server...
A request is accepted successfully
Starting file accepta file was saved as img99.jpg
C:\LAB1>client
Socket successfully opened at: 16386 (4002 in hex)
Input server socket: 16387
Sending request to server...
A request is accepted successfully
Starting file accepta file was saved as img37.jpg
C:\LAB1>
C:\LAB1>
```

Исходное изображение	Изображение, переданное без потерь
	
Потеря каждого 3 пакета	Потеря каждого 4 пакета



server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include "IPX.H"
#include "SHARED.H"

// Картинка с котиком (очень нужна клиенту)
FILE *source;

// Порт или сокет
unsigned int socket_num = 0;

// Должен ли работать мейнлуп
unsigned char should_run = 1;

// ECB для чтения сообщений
ECB *reader_ecb = NULL;

// Функция для создания ECB для получения сообщений
ECB *construct_receiver_ecb();

// Очистка ECB для получения сообщений
void destroy_ecb(ECB *ecb_to_del);

// Мейнлуп, создает ECB-читалку и ожидает пользовательского
// ввода для прекращения работы сервера
void go_to_mainloop();

// Функция-коллбек (ESR) для обработки сообщений на получение изображения.
// Здесь нам, на самом деле, важен сам факт того что мы получили сообщение.
// Теперь можем отправлять по полученному адресу картинку
void on_message_received_callback();

ECB *construct_receiver_ecb()
{
    ECB *new_esb = (ECB *)malloc(sizeof(ECB));
    new_esb->ESRAddress = NULL;
    new_esb->socket = _REVERSE_BYTES(socket_num);
    memset(new_esb->immAddress, 0xff, 6);
    new_esb->fragmentCnt = 2;
    new_esb->descs[0].addr = malloc(sizeof(IPXHeader));
    new_esb->descs[0].size = sizeof(IPXHeader);
}
```

```

new_esb->descs[1].addr = malloc(CLIENT_BUFFER_MESSAGE_SIZE);
new_esb->descs[1].size = CLIENT_BUFFER_MESSAGE_SIZE;

return new_esb;
}

void on_message_received_callback()
{
    int bytes_read;
    IPXHeader header;
    int packages_sent = 0, failed_packages_send = 0;
    ECB *construct_image_ecb = (ECB *)calloc(1, sizeof(ECB));
    printf("A new request for an image received\n");
    printf("A message: %s\n", reader_ecb->descs[1].addr);
    if (reader_ecb->ccode == 0xFC)
    {
        free(construct_image_ecb);
        printf("A request was canceled\n");

        return;
    }
    else if (reader_ecb->ccode != 0)
    {
        free(construct_image_ecb);
        printf("A request was damaged during transportation");

        return;
    }

    // Сделаем маленькую задержку чтобы клиент успел перевести своё состояние в прослушку.
    // лучше чем ничего
    delay(100);

    // Инициализация ECB
    construct_image_ecb->ESRAddress = NULL;
    construct_image_ecb->socket = _REVERSE_BYTES(socket_num);
    memset(construct_image_ecb->immAddress, 0xff, 6);
    construct_image_ecb->fragmentCnt = 2;
    construct_image_ecb->descs[0].addr = &header;
    construct_image_ecb->descs[0].size = sizeof(IPXHeader);
    construct_image_ecb->descs[1].addr = malloc(IMG_BUFFER_SIZE);
    construct_image_ecb->descs[1].size = IMG_BUFFER_SIZE;

    // Инициализация IPXHeader
    header.packetType = 4;
    memset(header.destNetwork, 0, 4);
    memset(header.destNode, 0xff, 6);
    header.destSocket = ((IPXHeader *)reader_ecb->descs[0].addr)->sourceSocket;

    fseek(source, 0, SEEK_SET);

    printf("Package sending set up complete\n");

    delay(100);

    while ((bytes_read = fread(construct_image_ecb->descs[1].addr, 1, IMG_BUFFER_SIZE, source)))
    {
        IPXSendPacket(construct_image_ecb);

        while (construct_image_ecb->inUse)
        {
            IPXRelinquishControl();
        }

        if (!construct_image_ecb->ccode)
        {
            packages_sent++;

```

```

    }
    else
    {
        failed_packages_send++;
    }
}

free(construct_image_ecb->descs[1].addr);
free(construct_image_ecb);

printf("Request was obtained\nTotal packages:      %d\nSucceeded packages: %d\nFailed
packages:      %d\n\n",
        packages_sent + failed_packages_send,
        packages_sent,
        failed_packages_send);
}

void go_to_mainloop()
{
    reader_ecb = construct_receiver_ecb();

    IPXListenForPacket(reader_ecb);

    while (should_run)
    {
        printf("Awaiting requests...\n");
        while (reader_ecb->inUse && should_run)
        {
            IPXRelinquishControl();
            if (kbhit())
            {
                should_run = 0;
            }
        }
        on_message_received_callback();
        IPXListenForPacket(reader_ecb);
    }

    printf("Have a good day!\n");
    exit(0);
}

int main()
{
    unsigned char socket_opening_result = 0;
    int i = 0;

    source = fopen(".\\cat.jpg", "rb");
    if (!source)
    {
        printf("Cat image not found :(\n");
        return 1;
    }

    // Проверяем, доступен ли IPX
    if (IPXInit() != 0xFF)
    {
        printf("IPX is not found");

        return 1;
    }

    // Пытаемся открыть сокет
    if ((socket_opening_result = IPXOpenSocket(SOCKET_SHORT_LIVING, &socket_num)) == 0)
    {
        printf("Socket succesfully opened at: %hu (%hx in hex)\n", socket_num, socket_num);
    }
}

```

```

else if (socket_opening_result == 0xFF)
{
    printf("Socket is already taken\n");
    return 1;
}
else if (socket_opening_result == 0xFE)
{
    printf("Socket table overflow\n");
    return 1;
}

go_to_mainloop();

IPXCancelEvent(reader_ecb);
IPXCloseSocket(socket_num);

fclose(source);

return 0;
}

```

client.c

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include "IPX.H"
#include "SHARED.H"

unsigned int socket_num = 0;
unsigned int server_num = 0;

// ECB для чтения сообщений
FILE *source;
ECB *reader_ecb;
ECB *to_send_esb;

// Функция-коллбек (ESR) для обработки сообщений на получение изображения.
// Здесь нам, на самом деле, важен сам факт того что мы получили сообщение.
// Теперь можем отправлять по полученному адресу картинку
void on_message_received_callback()
{
    printf("A new request for a message received\n");
    if (reader_ecb->ccode == 0xFC)
    {
        printf("A request was canceled\n");
    }
    else if (reader_ecb->ccode != 0)
    {
        printf("A request was damaged during transportation");
    }
}

// Функция для создания ECB для получения сообщений
ECB *construct_receiver_ecb()
{
    ECB *new_esb = (ECB *)malloc(sizeof(ECB));
    new_esb->ESRAddress = NULL;
    new_esb->socket = _REVERSE_BYTES(socket_num);
    memset(new_esb->immAddress, 0xff, 6);
    new_esb->fragmentCnt = 2;
    new_esb->descs[0].addr = malloc(sizeof(IPXHeader));
    new_esb->descs[0].size = sizeof(IPXHeader);
    new_esb->descs[1].addr = malloc(IMG_BUFFER_SIZE);
    new_esb->descs[1].size = IMG_BUFFER_SIZE;
}

```

```

    return new_esb;
}

ECB *construct_request_ecb(int chosen_socket_num, char *chosen_immediate_address, IPXHeader
*chosen_ipx_header)
{
    ECB *new_esb = (ECB *)calloc(1, sizeof(ECB));
    new_esb->ESRAddress = NULL;
    new_esb->socket = _REVERSE_BYTES(socket_num);
    new_esb->fragmentCnt = 2;
    new_esb->descs[0].addr = chosen_ipx_header;
    new_esb->descs[0].size = sizeof(IPXHeader);
    new_esb->descs[1].addr = malloc(CLIENT_BUFFER_MESSAGE_SIZE);
    new_esb->descs[1].size = CLIENT_BUFFER_MESSAGE_SIZE;
    sprintf(new_esb->descs[1].addr, "I need cat picture pls ._.");

    return new_esb;
}

void send_request_for_image()
{
    IPXHeader header;
    printf("Sending request to server...\n");

    header.packetType = 4;
    memset(header.destNetwork, 0, 4);
    memset(header.destNode, 0xff, 6);
    header.destSocket = _REVERSE_BYTES(server_num);

    to_send_esb = construct_request_ecb(header.destSocket, header.destNode, &header);

    IPXSendPacket(to_send_esb);

    while (to_send_esb->inUse)
    {
        IPXRelinquishControl();
    }

    if (to_send_esb->ccode == 0)
    {
        printf("A request is accepted successfully\n");
    }
    else
    {
        printf("Unable to send a request to a server. No cat image for you, sorry :(\n");
        exit(1);
    }

    return;
}

void go_main_loop()
{
    ECB *accept_ecb = construct_receiver_ecb();
    printf("Starting file accept");

    while (1)
    {
        IPXListenForPacket(accept_ecb);

        while (accept_ecb->inUse)
        {
            IPXRelinquishControl();
            if (kbhit())
            {
                IPXCancelEvent(accept_ecb);
            }
        }
    }
}

```

```

        return;
    }
}

if (!accept_ecb->ccode)
{
    fwrite(accept_ecb->descs[1].addr, 1, accept_ecb->descs[1].size, source);
}
}

destroy_ecb(accept_ecb);
}

int main()
{
    char filename[20];
    unsigned char socket_opening_result = 0;

    // Проверяем, доступен ли IPX
    if (IPXInit() != 0xFF)
    {
        printf("IPX is not found");

        return 1;
    }

    // Пытаемся открыть сокет
    if ((socket_opening_result = IPXOpenSocket(SOCKET_SHORT_LIVING, &socket_num)) == 0)
    {
        printf("Socket succesfully opened at: %hu (%hx in hex)\n", socket_num, socket_num);
    }
    else if (socket_opening_result == 0xFF)
    {
        printf("Socket is already taken\n");
        return 1;
    }
    else if (socket_opening_result == 0xFE)
    {
        printf("Socket table overflow\n");
        return 1;
    }

    printf("Input server socket: ");
    fflush(stdout);

    scanf("%hu", &server_num);

    srand(time(NULL));
    sprintf(filename, "img%d.jpg", rand() % 100 + 1);
    source = fopen(filename, "wb");

    send_request_for_image();
    go_main_loop();

    printf("A file was saved as %s\n", filename);

    fflush(source);
    fclose(source);
    IPXCloseSocket(socket_num);

    return 0;
}

```

shared.c

#ifndef SHARED_H


```

#define SHARED_H

#define CLIENT_BUFFER_MESSAGE_SIZE 64
#define SOCKET_SHORT_LIVING 0
#define IMG_BUFFER_SIZE 400

#include "IPX.H"

void destroy_ecb(ECB *ecb_to_del)
{
    if (ecb_to_del == NULL)
        return;

    free(ecb_to_del->descs[0].addr);
    free(ecb_to_del->descs[1].addr);
    free(ecb_to_del);
}

#endif

```

ipx.h

```

#ifndef _INC_IPX_H
#define _INC_IPX_H
/*Коды функций*/
#define _IPXOpenSocket 0x00
#define _IPXCloseSocket 0x01
#define _IPXListenForPacket 0x04
#define _IPXSendPacket 0x03
#define _IPXRelinquishControl 0x0A
#define _IPXCancelEvent 0x06
/*Инициализация сокета*/
#define _IPX_CHECK_DRIVER 0x7A00
#define _IPX_DRV_OK 0xFF
#define _IPX_MULTIPLEX_INT 0x2F
/*Открытие сокета*/
#define _IPX_SHORT_LIVED 0x0
#define _IPX_LONG_LIVED 0xFF
#define _IPXOpenSocket_OPEN 0x0
#define _IPXOpenSocket_BUSY 0xFF
#define _IPXOpenSocket_OVERFLOW 0xFE
/*Флаг InUse*/
#define _ECB_SND 0xFF
#define _ECB_REC 0xFE
#define _ECB_AES 0xFD
#define _ECB_QUE 0xFB
/*Прием*/
#define _ECB_CCODE_OK 0x0
#define _ECB_CCODE_NOSOCKET 0xFF
#define _ECB_CCODE_POVERFLOW 0xFD
/*Передача*/
#define _ECB_CCODE_CNC 0xFC
#define _ECB_CCODE_PHYS 0xFF
#define _ECB_CCODE_NOSTATION 0xFE
#define _ECB_CCODE_BADPACKET 0xFD
/*Отмена события*/
#define _IPXCancelEvent_OK 0x0
#define _IPXCancelEvent_FAILED 0xF9
#define _IPXCancelEvent_NOTBUSY 0xFF
/*Другие константы и макросы*/
#define _IPX_SAME_NETWORK 0x0
#define _IPX_BROADCAST 0xFF
#define _IPX_PCT_TYPE 0x4
#define _REVERSE_BYTES(x) (((x) & 0x00FF) << 8 | ((x) & 0xFF00) >> 8)

/*Точка входа*/

```

```

extern void (far *IPXDrvInvoke) (void);

typedef struct IPXHeader {
    unsigned int checksum;
    unsigned int length;
    unsigned char transportControl;
    unsigned char packetType;
    unsigned char destNetwork[4];
    unsigned char destNode[6];
    unsigned int destSocket;
    unsigned char sourceNetwork[4];
    unsigned char sourceNode[6];
    unsigned int sourceSocket;
} IPXHeader;

int IPXInit(void);

int IPXOpenSocket(int socketType, int *socketNum);
void IPXCloseSocket(int socketNum);
void IPXListenForPacket(void far *ECB);
void IPXSendPacket(void far *ECB);
void IPXRelinquishControl(void);
void IPXGetLocalTarget(void far *imm);
int IPXCancelEvent(void far *ECB);

typedef struct ECBfrag{
    void far *addr;
    unsigned int size;
} ECBfrag;

typedef struct ECB {
    void far *link;
    void (far *ESRAddress) (void);
    unsigned char inUse;
    unsigned char ccode;
    unsigned int socket;
    void far *IPXWorkspace;
    unsigned char driverWorkspace[12];
    unsigned char immAddress[6];
    unsigned int fragmentCnt;
    ECBfrag desc[4];
} ECB;

#endif

```

ipx.c

```

#include "ipx.h"
#include <dos.h>
#include <stdio.h>

void (far *IPXDrvInvoke) (void);

int IPXInit(void){
    int status = 0;

    _AX = _IPX_CHECK_DRIVER;
    geninterrupt(_IPX_MULTIPLEX_INT);
    status = _AL;

    IPXDrvInvoke = MK_FP(_ES, _DI);
    return status;
}

int IPXOpenSocket(int socketType, int *socketNum){
    int status = 0;

```

```

    _DX = _REVERSE_BYTES(*socketNum);
    _AL = socketType;
    _BX = _IPXOpenSocket;
    IPXDrvInvoke();
    status = _AL;
    *socketNum = _DX;
    *socketNum = _REVERSE_BYTES(*socketNum);
    return status;
}

void IPXCloseSocket(int socketNum){
    _BX = _IPXCloseSocket;
    _DX = _REVERSE_BYTES(socketNum);
    IPXDrvInvoke();
}

void IPXListenForPacket(void far *ECB){
    _BX = _IPXListenForPacket;
    _ES = FP_SEG(ECB);
    _SI = FP_OFF(ECB);
    IPXDrvInvoke();
}

void IPXSendPacket(void far *ECB){
    _BX = _IPXSendPacket;
    _ES = FP_SEG(ECB);
    _SI = FP_OFF(ECB);
    IPXDrvInvoke();
}

void IPXRelinquishControl(void){
    _BX = _IPXRelinquishControl;
    IPXDrvInvoke();
}

void IPXGetLocalTarget(void far *imm) {
    _BX = 0x09;
    _ES = FP_SEG(imm);
    _SI = FP_OFF(imm);
    IPXDrvInvoke();
}

int IPXCancelEvent(void far *ECB){
    _BX = _IPXSendPacket;
    _ES = FP_SEG(ECB);
    _SI = FP_OFF(ECB);
    IPXDrvInvoke();
    return _AL;
}

```