МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

М.А. Сонькин, А.А. Шелупанов, А.А. Шамин

РАСПРЕДЕЛЁННЫЕ МИКРОПРОЦЕССОРНЫЕ СИСТЕМЫ

УЧЕБНОЕ ПОСОБИЕ ПО КУРСОВОМУ ПРОЕКТИРОВАНИЮ

ЧАСТЬ 1

Рекомендовано Сибирским региональным отделением учебно-методического объединения вузов по образованию в области информационной безопасности для межвузовского использования в качестве учебного пособия по специальности 090103 – «Информационная безопасность автоматизированных систем»

В-Спектр
Томск – 2013

УДК 004.41
ББК 32.973.2-04
С 62

С 62 **Сонькин М.А., Шелупанов А.А., Шамин А.А.** Распределённые микропроцессорные системы: учебное пособие по курсовому проектированию. – Томск: В-Спектр, 2013. – 104 с.
ISBN 978-5-91191-299-4

В настоящем учебном пособии изложены научные основы построения распределённых микропроцессорных систем, приведены примеры аппаратных и программных решений с целью их использования в практической работе по выполнению курсового проекта.

Для студентов, магистрантов и аспирантов специальностей 230100 – «Информатика и вычислительная техника», 211000.62 – «Конструирование и технология электронных средств», 05.13.11 – «Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей».

УДК 004.41
ББК 32.973.2-04

***Рецензент* – С.С. Бондарчук**, доктор физ.-мат. наук, профессор ТГПУ

ISBN 978-5-91191-299-4

© Сонькин М.А., Шелупанов А.А.,
Шамин А.А., 2013

ОГЛАВЛЕНИЕ

Список используемых сокращений 5
Введение 6
1. Понятие распределённой микропроцессорной системы 7
1.1. Примеры РМПС 7
1.2. Особенности построения РМПС для труднодоступных и подвижных объектов 9
1.3. Классификация каналов связи 9
1.4. Программная модель РМПС 10
2. Микропроцессорные устройства (МПУ) 13
3. Программное обеспечение МПУ в составе РМПС 18
3.1. Способы реализации программного обеспечения 18
3.1.1. Программирование без использования операционной системы 18
3.1.2. Специализированные операционные системы 19
3.1.3. Операционные системы общего назначения 20
3.2. Примеры построения РМПС 20
3.2.1. РМПС сбора данных 21
3.2.2. РМПС оповещения объектов 22
4. Средства разработки ПО для микропроцессорных устройств 24
4.1. Организация проекта на языках C и C++ 24
4.2. Трансляция программы на языках C и C++ 27
4.3. Утилита make 29
4.3.1. Простейший сборочный скрипт 30
4.3.2. Трансляция проекта по частям 31
4.4. Трансляция проекта под различные платформы 32
4.5. Пример проекта из нескольких модулей 33
5. Примеры типовых программных решений для POSIX-совместимых ОС 39
5.1. Протокол обмена данными между МПУ 39
5.2. Алгоритмы обработки пакетов данных 42
5.2.1. Приём пакета данных 42
5.2.2. Передача пакета данных 43
5.3. Организация ввода-вывода в POSIX-совместимых ОС 44
5.4. Организация обмена данными по последовательному интерфейсу RS232 46
5.4.1. Инициализация порта RS232 в режиме RAW 46
5.4.2. Установка скорости порта RS232 48
5.4.3. Организация приёма информации по порту RS232 с помощью вызова select() 49
5.5. Обмен данными по IP-интерфейсам 51
5.5.1. Обмен данными по протоколу UDP 51
5.5.2. Создание UDP-сокета 52
5.5.3. Отправка UDP-пакета 53
5.5.4. Приём UDP-пакета 54

5.5.5. Обмен данными по протоколу TCP 55
5.5.6. Создание TCP-клиента 56
5.5.7. Создание TCP-сервера 56
5.5.8. Отправка TCP-пакета 58
5.5.9. Приём TCP-пакета 59
6. Пример организации ПО РМПС 60
6.1. Общая структура ПО РМПС (пример) 60
6.2. Модуль обработки событий 63
6.3. Модуль обработки протокола 67
6.4. Модуль управления каналами связи 76
6.4.1. Класс-канал связи по порту RS232 79
6.4.2. Класс-канал связи по сети интернет, протокол UDP 83
6.5. Модуль таблицы маршрутизации 88
6.6. Модуль начальной инициализации 93
Заключение 100
Литература 101

Список используемых сокращений

ТЗ – техническое задание
МПС – микропроцессорная система
РМПС – распределённая микропроцессорная система
МП – микропроцессор (микроконтроллер)
МПУ – микропроцессорное устройство
КС – канал связи
ЦЗ – целевая задача
СПД – системы передачи данных
АСУ – автоматизированные и автоматические системы управления
ССД – системы сбора данных
ПО – программное обеспечение
ОС – операционная система

ВВЕДЕНИЕ

Практически любая современная распределённая система управления, мониторинга или связи относится к классу распределённых микропроцессорных систем (РМПС).

Особое значение имеют РМПС, применяемые для оснащения труднодоступных и подвижных объектов. К таким системам, например, относятся – системы оповещения населения о чрезвычайных ситуациях и техногенных катастрофах, системы системы мониторинга опасных явлений (цунами, наводнения, лесные пожары, штормы и проч.).

Получение теоретических знаний, а так же практических навыков проектирования и реализации РМПС является актуальной задачей при подготовке специалистов в области микропроцессорных систем.

Данное учебное пособие по курсовому проектированию к курсу РМПС охватывает такие теоретические вопросы, как:

- понятие РМПС;
- отличие РМПС от других классов МПС;
- особенности построения РМПС;
- функции и особенности системного и прикладного ПО РМПС;
- операционные системы, применяемые в МПУ различного назначения.

Кроме того, рассмотрены практические вопросы построения РМПС применительно к курсовому проектированию, в том числе:

- приведены примеры стендов для разработки аппаратного и программного обеспечения нескольких вариантов РМПС;
- рассмотрены способы реализации ПО для МПУ в составе РМПС;
- приведён пример разработки протокола обмена данными между МПУ в составе РМПС;
- рассмотрен вопрос выбора средств разработки ПО для МПУ;
- приведены практические рекомендации по работе со средствами разработки.

Особенностью учебного пособия является ориентация на практический опыт авторов в области построения распределённых микропроцессорных систем. В него включены материалы реальных проектов, выполненных в интересах ряда экономики, в том числе:

- элементы систем оповещения на базе П-166 ИТК ОС;
- компоненты сбора метеоданных для аппаратно-программного комплекса АПС-Метео;
- микропроцессорный интеллектуальный информационно-телекоммуникационный терминал ВИП-МК
- и др.

1. ПОНЯТИЕ РАСПРЕДЕЛЁННОЙ МИКРОПРОЦЕССОРНОЙ СИСТЕМЫ

Существуют различные определения распределённой микропроцессорной системы [1]. Воспользуемся теми из них, которые являются наиболее удобными для целей данного учебного пособия.

Несколько определений, сформулированных в терминах распределённых микропроцессорных систем, приведены ниже.

РМПС – это МПС, которая приобрела специфику территориально рассредоточенной системы [2];

РМПС – это совокупность независимых МПУ, взаимодействующих с целью решения задач, не решаемых одним МПУ индивидуально [3];

РМПС – это совокупность независимых МПУ, представляющаяся пользователям единой системой [1].

Исходя приведённых определений, неотъемлемыми частями любой РМПС являются:

ЦЗ – целевые задачи, для решения которых предназначена данная РМПС и которые решаются всей системой в целом, но не её компонентами в отдельности;

МПУ – микропроцессорные устройства, каждое из которых взаимодействует с другими МПУ, входящими в РМПС с целью решения определённой задачи, не решаемой одним МПУ индивидуально;

КС – каналы связи, которые обеспечивают обмен информацией между МПУ, входящими в РМПС.

Таким образом, считаем, что мы имеем дело с РМПС, если в неё входят все три компонента: ЦЗ, МПУ и КС.

По назначению к РМПС относятся:

СПД – системы передачи данных;

АСУ – автоматизированные и автоматические системы управления;

ССД – системы сбора данных и т.п.

1.1. Примеры РМПС

С целью конкретизации понимания задач, стоящих при разработке РМПС, приведём несколько примеров.

Пример 1. Система оповещения населения о чрезвычайных ситуациях [4, 5].

Данная система включает центр-источник сигналов оповещения и объекты оповещения. Задачей системы является доведение сигналов оповещения до объектов оповещения. На уровне целевой задачи определяется:

- максимальное количество сигналов оповещения;
- максимальное количество объектов оповещения;

• способ доведения сигналов – звуковой (сирена, речь и т.п.), визуальный («мигалка», дисплей, бегущая строка и т.п.);

• сроки доведения сигнала от источника до каждого объекта;

• способы подтверждения объектом получения сигнала оповещения;

• дополнительные функции системы, не относящиеся непосредственно к решению основной задачи оповещения;

• другие характеристики, важные для функционирования системы в целом.

Пример 2. Система автоматического сбора метеорологической информации [6, 7].

Данная система включает в себя центр сбора данных и датчики, поставляющие метеорологическую информацию.

Задачей системы является доставка информации от датчиков в центр сбора данных. На уровне целевой задачи определяется:

• максимальное количество датчиков;

• тип и количество информации для каждого датчика;

• сроки сбора информации (например: непрерывный сбор, период сбора и проч.) для каждого датчика и типа информации;

• способ передачи информации для каждого датчика – по инициативе датчика или центра;

• дополнительные функции системы, не относящиеся непосредственно к решению основной задачи сбора метеорологической информации;

• другие характеристики, важные для функционирования системы в целом.

Несмотря на то, что задачи РМПС, рассмотренных в примерах, сильно различаются, структура этих систем одинакова (рис. 1) и представляет собой иерархию, включающую несколько уровней.

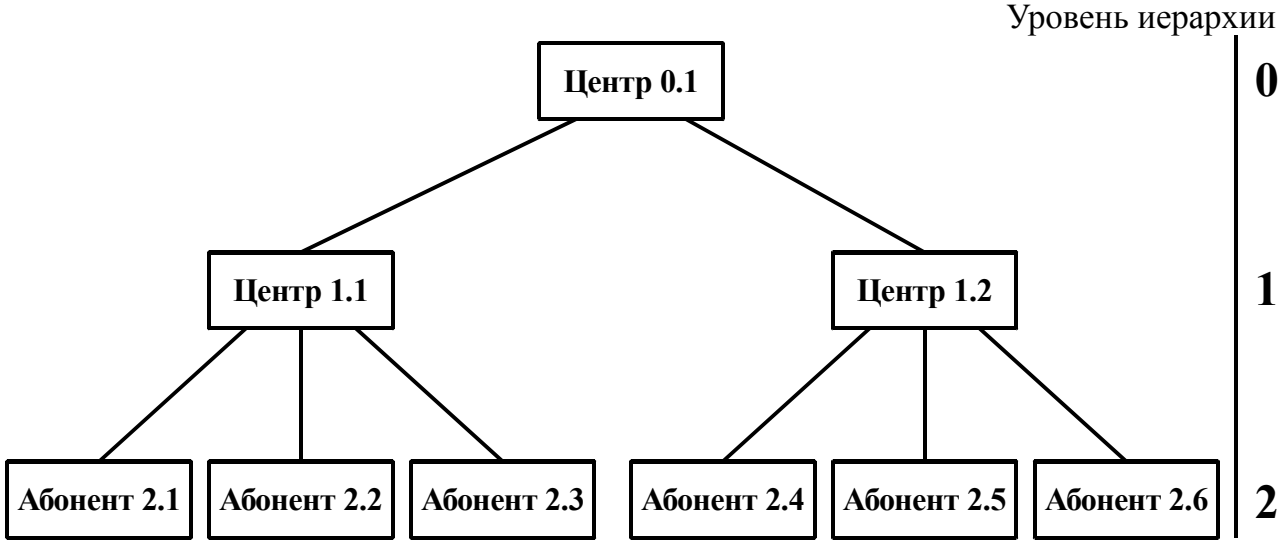

Рис. 1. Типичная структура системы для ТДС и подвижных объектов

1.2. Особенности построения РМПС для труднодоступных и подвижных объектов

Построение РМПС для труднодоступных и подвижных объектов имеет ряд особенностей. Исходя из области применения таких РМПС – системы сбора данных, мониторинга, оповещения, эти РМПС, как правило строятся по иерархическому принципу, т.е. имеют на каждом уровне иерархии явно выраженный «центр» и подчинённых ему абонентов.

Обмен данными между произвольными абонентами помимо центра является возможным, но не обязательным.

Рассмотрим типичную структуру РМПС для ТДС, представленную на рис. 1. Будем считать, что «центр» – это МПУ в составе РМПС, имеющий подчинённые МПУ, а «абонент» – это МПУ в составе РМПС, не имеющее подчинённых МПУ.

В нашем примере система включает три уровня иерархии. Нижний уровень 2 – конечные абоненты, уровень 1 – промежуточные центры и уровень 0 – центр системы.

Иерархическая структура РМПС накладывает определённые ограничения на обмен данными и малопригодна для обмена данными между произвольными МПУ.

Например, если абоненту №2.1 требуется передать данные другому абоненту №2.5, то эти данные – должны пройти по такому маршруту: абонент №2.1 – центр №1.1 – центр №0.1 – центр №1.2 – абонент №2.5.

Проектируя алгоритмы обработки данных, следует учитывать, что наиболее эффективна иерархическая структура, когда весь орбмен данными производится между центром и непосредственно подчинёнными ему абонентами (центрами).

1.3. Классификация каналов связи

Каналы связи различны по своим свойствам. Различны их скоростные характеристики. Различны алгоритмы передачи данных по каналам связи различной природы. Поэтому выбор каналов связи и оценка пригодности их для решения задач, поставленных перед РМПС, – важный этап проектирования РМПС и МПУ. Один из вариантов классификации КС показан на рис. 2.

Классификация произведена по нескольким параметрам – скоростным характеристикам, способу установления соединения и способу обмена данными.

Скоростные характеристики

С точки зрения скоростных характеристик, КС подразделяются на высокоскоростные и низкоскоростные.

Считаем, что **высокоскоростные КС** – это КС, позволяющие реализовать полнофункциональный режим работы РМПС.

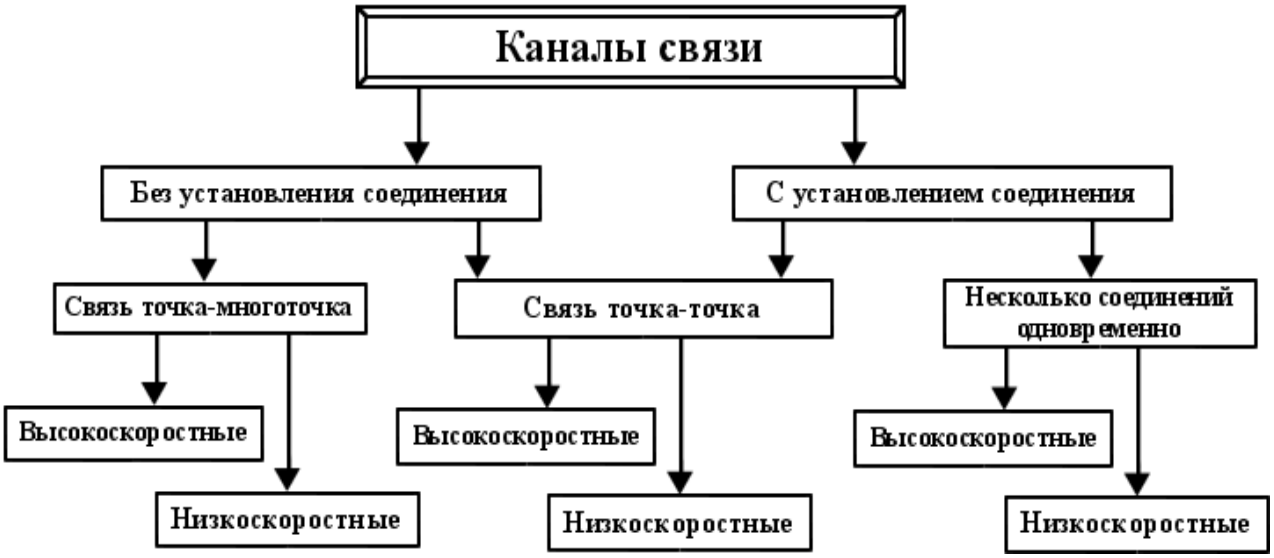

Рис. 2. Классификация каналов связи

Низкоскоростные КС – те КС, которые позволяют реализовать функции РМПС в режиме ограниченного функционала.

Таким образом, понятия «низкоскоростной» и «высокоскоростной» канал связи – зависят от конкретной ЦЗ и требований к конкретной РМПС.

КС, не обеспечивающие работу конкретной РМПС даже в режиме ограниченного функционала, считаем непригодными для использования и не рассматриваем.

Способ установки соединения

По способу установки соединения КС классифицируются следующим образом:

– КС с постоянным соединением. Это однократно устанавливаемое и постоянно контролируемое соединение. При обнаружении разрыва – производится восстановление соединения. Постоянное соединение характерно для каналов связи, использующих протокол TCP/IP;

– КС с соединением на время сеанса связи. В этом случае при передаче сообщения производится процедура установления соединения, затем обмен данными, после чего соединение разрывается. Соединение на время сеанса характерно для прямых модемных соединений (PSTN, GSM);

– КС без установления соединения. Обмен данными производится сразу, без процедуры установки соединения. Такой способ характерен, например, для радиоканала.

1.4. Программная модель РМПС

Существует несколько программных моделей, позволяющих рассматривать взаимодействие МПУ в составе РМПС на различных уровнях представления. Наиболее распространённые из указанных моделей – се-

миуровневая модель OSI [9] и четырёхуровневая модель TCP/IP (DOD) [10]. Программная модель DOD создана для реализации стека протоколов TCP/IP и является по сути упрощённой моделью OSI.

Мы будем рассматривать РМПС исходя из программной модели TCP/IP (DOD), представленной на рис. 3, поскольку модель OSI избыточна для целей данного учебного пособия.

Модель DOD состоит из четырёх уровней (сверху вниз):

Прикладного уровня или уровня приложений (англ. Process / Application), обеспечивающего программный интерфейс для взаимодействия с приложениями пользователя. На этот же уровень возлагаются задачи преобразования протоколов, шифрования/дешифрования данных, поддержание сеанса связи, позволяя приложениям взаимодействовать между собой длительное время, создание и завершение сеанса связи.

Транспортного уровня (англ. Transport), предназначенного для обеспечения надёжной передачи данных от отправителя к получателю.

Рис. 3. Программная модель DOD

Межсетевого уровня (англ. Internet) – отвечает за трансляцию логических адресов и имён в физические, определение кратчайших маршрутов, коммутацию и маршрутизацию, отслеживание неполадок в сети.

Уровня сетевого доступа (англ. Network Access), предназначенного для обеспечения взаимодействия сетей на физическом уровне и контроля за ошибками. На данном уровне данные упаковываются в кадры и осуществляется передача-приём кадров по физическим каналам связи, а также контроль целостности кадров.

Следует помнить, что обе рассмотренные программные модели взаимодействия между МПУ – ISO и DOD – достаточно универсальны, но в то же время не описывают некоторых ограничений, возникающих при проектировании реальных РМПС, например скоростных ограничений физических каналов связи, ограниченного быстродействия МПУ.

Для того чтобы учесть эти ограничения, необходимо рассмотреть РМПС с точки зрения системотехники на различных уровнях абстракции.

2. МИКРОПРОЦЕССОРНЫЕ УСТРОЙСТВА (МПУ)

Микропроцессорное устройство (МПУ) представляет собой функционально и конструктивно законченное изделие, состоящее из нескольких микросхем, в состав которых входит микропроцессор; оно предназначено для выполнения определённого набора функций: получение, обработка, передача, преобразование информации и управление.

Применительно к РМПС МПУ обязательно содержит в своём составе либо средства для передачи данных по каналам связи, либо интерфейсы для подключения таких средств (например, модемов, оптоволоконных линий, спутниковых терминалов и проч.).

Как правило, МПУ в составе РМПС должно сохранять ограниченную функциональность при отсутствии соединений по каналам связи.

Например, в системах сбора данных и системах документированной связи [4–7], а также в большинстве иных РМПС существуют требования доставки всех введённых в МПУ данных без потерь. Но практически в любой РМПС имеются периоды времени, когда связь с центром и всеми другими МПУ на некоторое время прерывается по тем или иным причинам.

С целью выполнения требования доставки всех введённых в МПУ данных без потерь обычно применяется достаточно простой алгоритм: в отсутствие связи по всем доступным каналам МПУ запоминает сформированные сообщения и хранит их до появления связи, а затем передаёт их по назначению.

Отметим, что данный алгоритм накладывает на протокол передачи сообщений требование о наличии в каждом сообщении времени его формирования. Иначе «центру» невозможно будет узнать какому моменту времени какое сообщение соответствует. Кроме этого, очевидно, что все МПУ, входящие в состав РМПС, должны быть синхронизированы по времени.

Итак, сформулируем общие требования, предъявляемые к МПУ в составе РМПС:

– МПУ должно содержать в своём составе либо средства для передачи данных по каналам связи, либо интерфейсы для подключения таких средств;

– МПУ должно обеспечивать ограниченную функциональность в отсутствие связи с другими МПУ;

– все МПУ в составе РМПС должны иметь часы, синхронизированные по времени. Причём часы должны корректно отсчитывать время и при отсутствии связи с «центром» и другими МПУ данной системы;

– иные требования к МПУ, накладываемые решаемыми задачами. Основные требования накладывает целевая задача (ЦЗ). Но, кроме этого, возможны и иные требования к МПУ, не связанные с ЦЗ, но значительно

повышающие функциональные возможности МПУ – например, комфорт оператора при работе с МПУ, удобство диагностирования неполадок в МПУ и т.п.

В качестве примера МПУ, используемого в составе РМПС, рассмотрим многофункциональный микропроцессорный терминал ВИП-МК.

Микропроцессорный терминал ВИП-МК (рис. 4) представляет собой универсальное устройство сбора, обработки и передачи-приёма информации.

Рис. 4. Многофункциональный микропроцессорный терминал ВИП-МК

Основой ВИП-МК является (рис. 5) микропроцессор (центральный процессор ВИП-МК) AT91RM9200. Данный процессор состоит из ядра ARM9 и встроенных периферийных устройств.

Память ВИП-МК состоит из трёх различных по функциональности запоминающих устройств:

1) динамического ОЗУ объёмом 32 Мбайт, подключённого к системной шине процессора, предназначенного для хранения программных компонентов и данных во время их выполнения;

2) NAND-FLASH ПЗУ объёмом не менее 128 Мбайт, используемого в качестве «твёрдого диска» для хранения программ и другой информации в виде файлов;

3) загрузочной FLASH-ПЗУ (Boot Flash) объёмом 8 Мбайт, предназначенного для хранения начального загрузчика U-Boot и ядра ОС Linux.

Интерфейс USB A предназначен для подключения различных внешних периферийных устройств – USB-Flash, USB-аудиокарт (гарнитур) и т.д.

Интерфейс USB B предназначен для эмуляции сетевой карты.

Интерфейс Ethernet предназначен для подключения ВИП-МК по локальной сети.

Интерфейсы RS232 (COM-порты, последовательные порты). ВИП-МК имеет 4 встроенных интерфейса RS232. Каждый из них имеет свои

особенности функционирования. Ниже рассмотрены особенности каждого из последовательных портов.

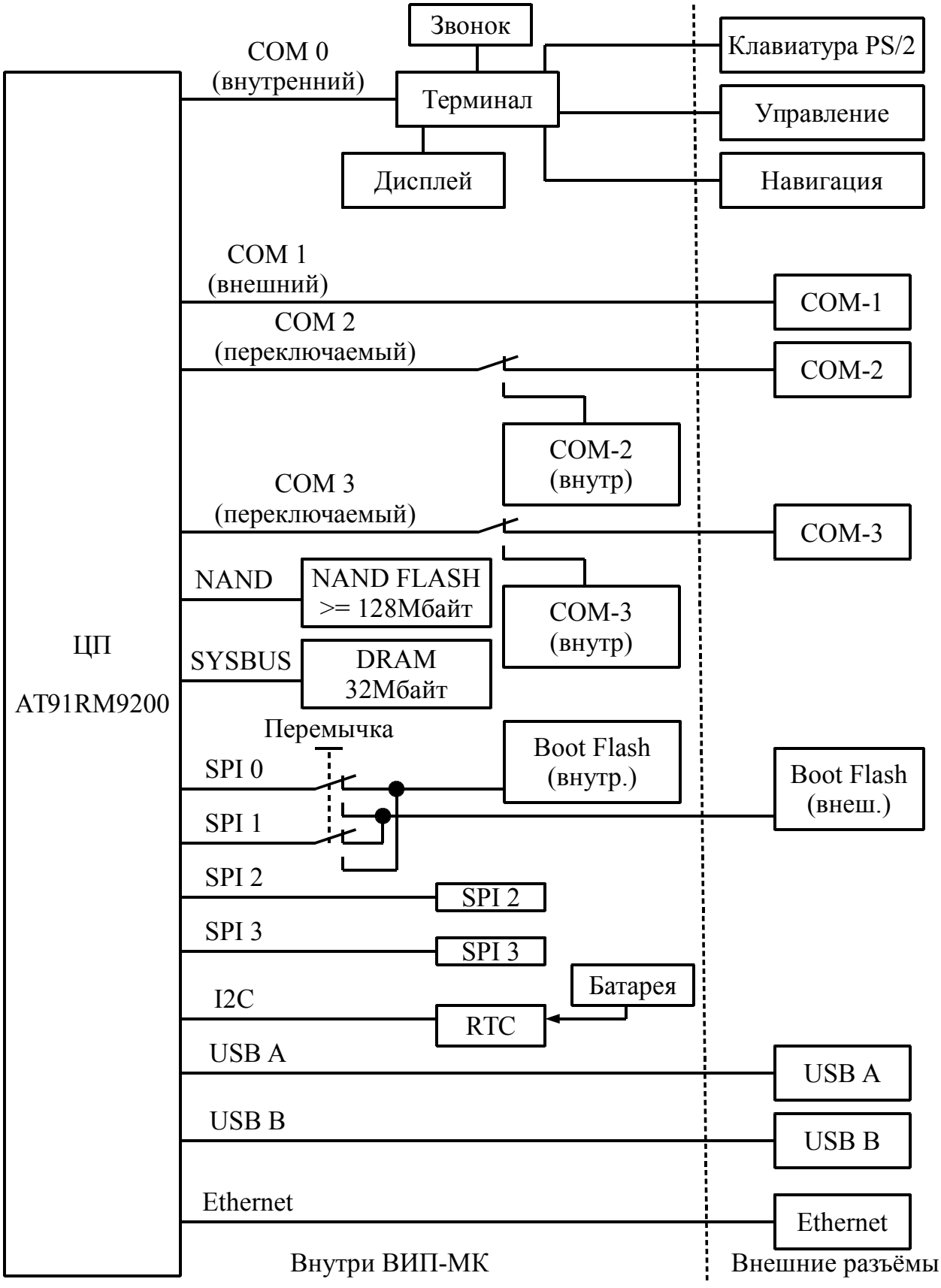

Рис. 5. Функциональная схема ВИП-МК

Порт COM-0 является встроенным и используется для связи центрального процессора ВИП-МК с процессором встроенного терминала ввода-вывода, управляющего выводом на двухстрочный индикатор и вводом с клавиатуры PS/2. Функции терминала рассмотрены ниже.

Порт COM-1 является внешним. Физически данный порт выведен на заднюю панель ВИП-МК. В рабочем режиме данный порт используется для подключения различных периферийных устройств с интерфейсом RS232. В отладочном режиме данный порт может использоваться как системный терминал.

Порт COM-2 является переключаемым. То есть разъём данного порта находится внутри корпуса ВИП-МК, но с помощью переходника может подключаться к разъёму «COM-2» на задней панели ВИП-МК. В зависимости от конкретного назначения терминала ВИП-МК порт используется либо для подключения различных внешних периферийных устройств с интерфейсом RS232, либо для периферийных устройств, встроенных в корпус терминала ВИП-МК.

Порт COM-3 является переключаемым и полностью аналогичен по функциям порту COM-2. Особенностью порта **COM-3** является то, что он также служит для подключения встроенного радиомодема. В режиме, когда порт **COM-3** служит для подключения радиомодема, удаляется специальная перемычка на плате ВИП-МК и шлейф подключения радиомодема соединяется с платой ВИП-МК. В устройствах, где порт COM-3 используется для подключения внешних устройств (аналогично COM-2), перемычка «радиомодем отключён» должна быть установлена, а шлейф подключения радиомодема должен быть отсоединён от платы ВИП-МК.

Встроенный терминал ввода-вывода ВИП-МК является законченным микропроцессорным устройством, включённым в состав платы ВИП-МК. Основой терминала ввода-вывода ВИП-МК является микропроцессор AT91SAM7S256, базирующийся на ядре ARM7. Связь между процессором терминала ввода-вывода и центральным процессором ВИП-МК осуществляется посредством порта **COM-0** центрального процессора.
В функции терминала ввода-вывода входят:

1. Управление двухстрочным символьным индикатором размерами 2×40 символов – очистка индикатора, позиционирование курсора, вывод символов, поступающих на вход терминала в кодировке KOI8-R.
2. Ввод символов с клавиатуры PS/2 – преобразование кодов клавиш в коды символов (кодировка KOI8-R) в соответствии с текущей раскладкой клавиатуры (рус./лат.) и регистром ввода (прописные/строчные буквы). Также специальные ESC-последовательности формируются при нажатии функциональных клавиш (F1-F12) и специальных сочетаний клавиш (CTRL+Fxx, ALT+Fxx и т.д.).

3. Управление режимом работы терминала – программное переключение раскладки клавиатуры, регистра и т.п.
4. Управление индикатором «Почта» передней панели.
5. Управление двумя дискретными выходами разъёма «Управление».
6. Ввод данных с навигационного приёмника GPS или GLONASS.

Управление терминалом ввода-вывода осуществляется с помощью специальных ESC-последовательностей, посылаемых центральным процессором по порту COM-0. Скорость обмена с терминалом фиксирована и составляет 19200 бит/с.

Программное обеспечение терминала ВИП-МК работает под управлением ОС Linux. Применение ОС общего назначения позволяет использовать широкий набор программ, которые доступны в виде исходного кода.

3. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ МПУ В СОСТАВЕ РМПС

Выделяется ряд общих программных компонентов, присутствующих в любом ПО, которое исполняется на МПУ в составе РМПС.

Прежде всего это ПО, обеспечивающее обмен данными между МПУ. Функции данного ПО – реализация алгоритмов обмена данными по каждому из КС, подключённому к МПУ.

Другая составляющая ПО – это программные модули, реализующие целевую функцию.

При разработке ПО студентам – участникам проекта следует определить следующее:

1) общий алгоритм работы всей системы в целом и каждого МПУ в отдельности. Алгоритм составляется на основании задания на проектирование;

2) функции и алгоритм работы МПУ в составе РМПС. Функции МПУ обосновываются исходя из задания на проектирование. Алгоритм работы каждого МПУ описывается исходя из реализуемых им функций и алгоритма работы системы в целом;

3) функции и алгоритм работы каждого программного модуля;

4) способ реализации программных модулей;

5) механизм взаимодействия между различными программными модулями.

3.1. Способы реализации программного обеспечения

В зависимости от назначения и возможностей аппаратного обеспечения микропроцессорных устройств применяется несколько основных методов проектирования и реализации программного обеспечения.

3.1.1. Программирование без использования операционной системы

Данный метод применяется при разработке ПО для тех микропроцессорных устройств, которые не имеют какого-либо системного программного обеспечения. Чаще всего, это микроконтроллеры с низкой производительностью, например семейства AVR или PIC [10], а также младшие представители семейств ARM [11].

Достоинства данного метода – полный доступ программиста к аппаратному обеспечению процессора (микроконтроллера), что позволяет оптимальным образом сконфигурировать периферийные устройства и добиться, например, максимально возможной производительности.

Недостаток данного метода – трудоёмкость, обусловленная отсутствием стандартных средств и необходимостью писать «с нуля» поддержку всех необходимых функций работы с аппаратурой, протоколами и проч.

Для компенсации этого недостатка производители микроконтроллеров поставляют набор программных библиотек для работы с аппаратурой. Такие библиотеки поставляются в виде исходных кодов на языках ассемблера и С и содержат функции управления терминальным вводом-выводом по последовательному порту, функции управления таймерами, линиями ввода-вывода и другим периферийным оборудованием, интегрированным в микроконтроллер.

3.1.2. Специализированные операционные системы

При реализации ПО для микропроцессорных устройств часто встают типовые задачи, такие как переключение процессов, организация буферов ввода-вывода, обработка прерываний и др.

С целью облегчения проектирования и реализации ПО разработаны специализированные операционные системы для встраиваемых устройств. К таким ОС относятся, например, eCos, ChibiOS/RT, FreeRTOS и др. [12]. Обычно специализированные ОС являются системами реального времени.

Данные ОС специализированы в том смысле, что содержат ограниченный набор функций по сравнению с ОС общего назначения. Как правило набор функций специализированной ОС не содержит функций файлового ввода-вывода, не подразумевает разделения прав доступа к ресурсам. Поскольку данные ОС чаще всего применяются в микроконтроллерах и находятся физически в ПЗУ вместе с ПО пользователя, то они не содержат функций, позволяющих загружать и запускать программы с внешних носителей.

Относительная простота организации специализированных ОС позволяет, в случае необходимости, легко добавлять нужные модули, а также избавлять программиста от рутинных операций по переключению процессов, организации функций терминального ввода-вывода и других подобных вещей.

Как правило, специализированные ОС имеют несколько базовых компонентов:

- планировщик задач;
- уровень абстракции оборудования;
- интерфейс пользовательского ПО.

Кроме этого, в состав специализированной ОС могут включаться дополнительные компоненты, расширяющие функциональность ОС, например: математические библиотеки, поддержка многопоточности, отладочные функции и проч.

Рассмотрим назначение различных компонентов специализированной ОС.

Планировщик задач позволяет организовать в пользовательском ПО несколько независимых процессов (задач), выполняющихся параллельно. Такая необходимость возникает, например, при организации обмена данными по нескольким каналам связи одновременно, что актуально для МПУ в составе РМПС.

Часто специализированная ОС содержит несколько планировщиков задач, один из которых можно выбрать при её конфигурации.

Уровень абстракции оборудования представляет собой, по сути, набор драйверов устройств. Каждый драйвер имеет стандартный для данной ОС программный интерфейс, что позволяет улучшить переносимость ПО с одного устройства на другое.

Интерфейс пользовательского ПО представляет собой набор системных функций, с помощью которых ПО пользователя обращается к различным компонентам ОС – планировщику задач, драйверам ввода-вывода (уровню абстракции оборудования) и осуществляет межпроцессное и межпоточное взаимодействие.

При правильном построении ПО всё взаимодействие между специализированной ОС и ПО пользователя происходит только через интерфейс пользовательского ПО.

3.1.3. Операционные системы общего назначения

С ростом производительности и снижением цены микропроцессорной техники в микропроцессорных устройствах всё чаще применяются ОС общего назначения, такие как Linux, QNX, WinCE и т.п.

Создание ПО для таких систем наиболее комфортно с точки зрения проектировщиков и программистов.

Это обусловлено тем, что, во-первых, для данных ОС существует обширнейший инструментарий для написания и отладки программ, а во-вторых, тем, что можно создавать и отлаживать ПО на персональном компьютере с той же ОС, что и на целевом МПУ, а затем путём кросс-компиляции перенести уже отлаженное ПО на требуемое МПУ.

3.2. Примеры построения РМПС

В качестве примеров рассмотрим построение распределённых микропроцессорных систем на базе терминалов ВИП-МК (рис. 6, 7). Системы представляют собой учебные стенды для изучения построения РМПС и экспериментов с ними.

Стенды, рассмотренные ниже, построены на базе трёх различных МПУ – персонального компьютера (ПК), микропроцессорного терминала ВИП-МК и платы расширения STK600 на базе микроконтроллера AVR ATMega2560.

При моделировании различных РМПС стенды могут быть дополнены необходимым оборудованием – например, USB-аудиокартами, USB-видеокартами, датчиками давления, температуры, движения, считывателями идентификационных карт и т.п.

Таким образом, разработка ПО для данных стендов даёт практические навыки программирования различных МПУ и на разных уровнях абстракции: для ПК и ВИП-МК – программы выполняются под управлением ОС Linux; для платы расширения STK600, не имеющей ОС, программы пишутся без использования ОС.

3.2.1. РМПС сбора данных

Система сбора данных (ССД) обеспечивает считывание нескольких датчиков и передачу информации в центр сбора данных.

Основная задача ССД – сбор данных с некоторых МПУ-источников данных и передача собранных данных в центр сбора данных (ЦСД). МПУ получает данные от датчиков автоматически, либо данные вводятся оператором. Учебный стенд, имитирующий систему сбора данных представлен на рис. 6.

В качестве центра системы сбора данных используется ПК под управлением ОС Linux. В качестве МПУ-источников данных – терминалы ВИП-МК с подключёнными к ним имитаторами датчиков на базе микроконтроллеров AVR ATMega2560.

Считаем, что данные передаются по сети Internet, в качестве эмуляции которой используется локальная сеть.

Рассмотрим работу стенда. Режим работы и показания имитаторов датчиков задаются с помощью кнопок, подключённых к микроконтроллерам AVR ATMega2560.

Данные с имитаторов датчиков ИД1 – ИД*N* передаются по интерфейсу RS232 в соответствующие терминалы ВИП-МК1 – ВИП-МКN.

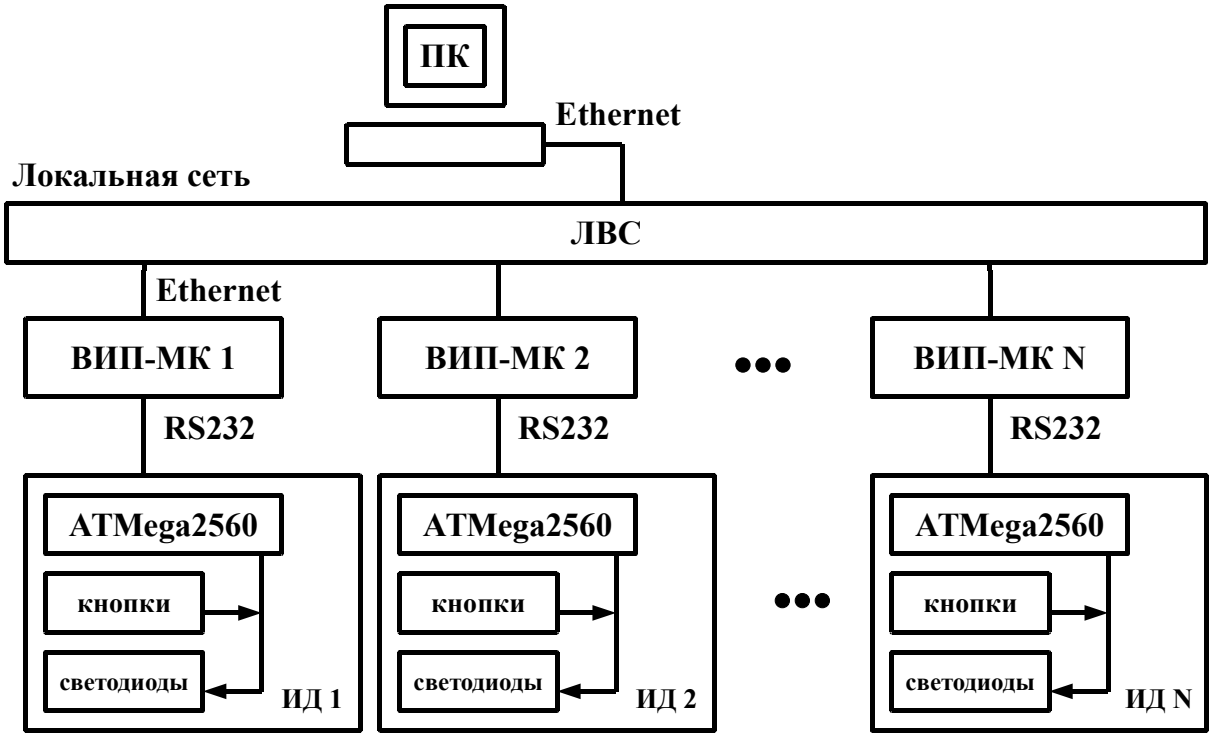

Рис. 6. Система сбора данных (учебный стенд)

ВИП-МК преобразует данные от имитаторов датчиков в форматы, необходимые для передачи их в центр системы сбора данных и передаёт по локальной сети на персональный компьютер.

Управление ВИП-МК осуществляется как с помощью стандартной клавиатуры PS/2 (на рис. 6 не показана), так и удалённо по протоколу **ssh,** например с ПК.

Данный стенд позволяет:

- имитировать различные типы датчиков путём замены ПО для имитаторов датчиков;
- отображать данные с датчиков и другую необходимую информацию на дисплее ВИП-МК;
- организовывать работу по различным протоколам в сети Интернет путём замены ПО для ВИП-МК и ПК;
- организовывать обмен произвольными сообщениями между ВИП-МК и ПК;
- оценивать сетевой трафик системы сбора данных;
- отлаживать программное обеспечение ВИП-МК, ПК и ИД.

Процесс формирования данных и доставки их в центр происходит в два этапа:

1. Получение МПУ данных от датчика (оператора).
2. Передача данных от МПУ в ЦСД.

Алгоритмы, применяемые на каждом этапе формирования данных и доставки их в центр, зависят от конкретной задачи ССД, имеющихся каналов связи, протоколов и других факторов.

Рассмотрим наиболее распространённые алгоритмы каждого этапа.

1. Получение МПУ данных от датчика (оператора):
1) датчик отдаёт текущие данные по моменту их формирования;
2) датчик отдаёт текущие данные с заданными интервалами времени;
3) датчик отдаёт текущие данные по запросу МПУ.

2. Передача данных от МПУ в ЦСД:
1) МПУ передаёт в центр данные по собственной инициативе;
2) МПУ передаёт в центр данные по запросу ЦСД.

3.2.2. РМПС оповещения объектов

Система оповещения объектов представлена на рис. 7. Её основной задачей является доведение специальных сообщений, называемых сигналами оповещения, от центра до абонентов. В сигнал оповещения могут входить текстовое сообщение, звуковой сигнал (сирена), звуковое (речевое) сообщение, световая сигнализация.

В качестве центра системы оповещения используется ПК под управлением ОС Linux. В качестве МПУ-абонентов – терминалы ВИП-МК с подключёнными к ним имитаторами светозвуковой сигнализации и датчиков на базе микроконтроллеров AVR ATMega2560.

ВИП-МК может быть дополнен USB-картой (не показана на рис. 7) для воспроизведения звуковых сообщений, например речи.

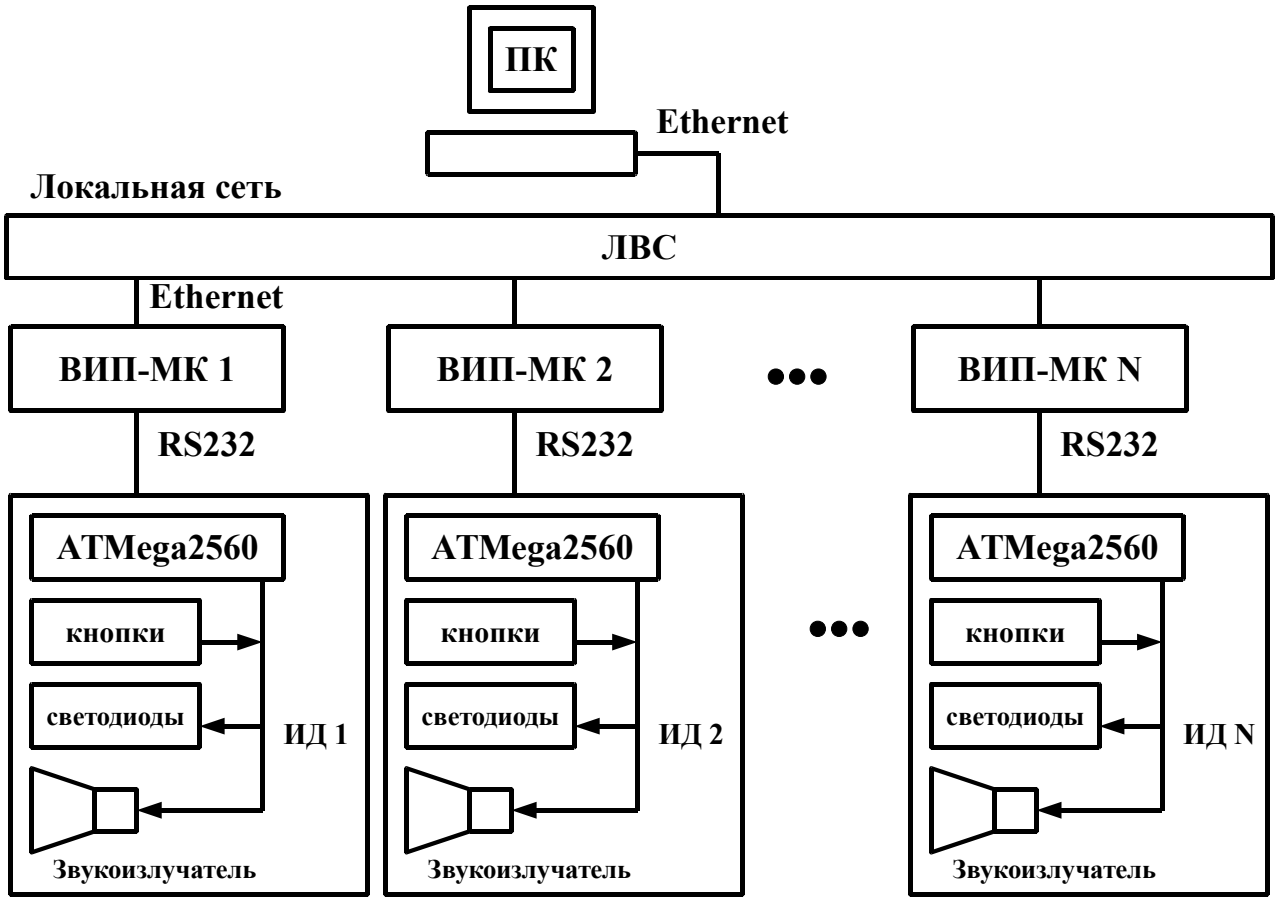

Рис. 7. Система оповещения объектов (учебный стенд)

Считаем, что данные между центром и абонентами передаются по сети Internet, в качестве эмуляции которой используется локальная сеть.

Кроме основной функции – оповещения объектов, система оповещения, как правило, несёт ряд дополнительных функций, например: телефонная связь между объектом оповещения и центром, сбор данных с каких-либо датчиков и т.п.

Имитаторы светозвуковой сигнализации помимо основной функции – подачи световых и звуковых сигналов могут также имитировать различные датчики.

Обмен данными между ВИП-МК и имитаторами светозвуковой сигнализации и датчиков ИД осуществляется по интерфейсу RS232.

Данный стенд позволяет:

- имитировать различные типы сигналов оповещения;
- отображать данные сигналов оповещения и другую необходимую информацию на дисплее ВИП-МК;
- организовывать работу по различным протоколам в сети Интернет путём замены ПО для ВИП-МК и ПК;
- организовывать обмен произвольными сообщениями между ВИП-МК и ПК;
- оценивать сетевой трафик системы оповещения;
- отлаживать программное обеспечение ВИП-МК, ПК и ИД.

4. СРЕДСТВА РАЗРАБОТКИ ПО ДЛЯ МИКРОПРОЦЕССОРНЫХ УСТРОЙСТВ

В настоящее время существует обширный набор средств разработки для создания ПО МПУ – это компиляторы и интерпретаторы языков программирования, отладчики, специализированное ПО для программирования встроенных ПЗУ, средства для автоматизации сборки программного обеспечения и иное специализированное ПО.

Выбор средств разработки зависит от конкретного аппаратного обеспечения МПУ, решаемых задач, наличия или отсутствия операционной системы на выбранном МПУ и т.д.

Для младших моделей микроконтроллеров, имеющих десятки байт встроенного ОЗУ без возможности его расширения, возможно использовать только язык ассемблера.

Для средних и старших моделей микроконтроллеров, имеющих встроенное или внешнее ОЗУ от сотен байт и более, возможно использовать широкий набор компиляторов и интерпретаторов языков программирования, таких как ассемблер, C, C++, Ada, BASIC, FORTH, Java и др.

Самыми популярными из названных являются языки ассемблер, C и C++.

В настоящее время сложилась практика: с выпуском нового типа микроконтроллера фирма-производитель заботится о том, чтобы были доступны средства разработки в виде языка ассемблера, а если позволяют ресурсы микроконтроллера – то и компилятора языка C и C++.

4.1. Организация проекта на языках C и C++

Вопрос правильной организации проекта на языках C и C++ является одним из важнейших при разработке ПО.

Выделяется несколько уровней организации: процедурный уровень (или уровень классов), файловый уровень, уровень библиотек.

Рассматривая программу на уровне процедур или классов, необходимо определить – какую задачу решает та или иная функция (класс), какие входные данные нужны для этого.

Для проекта даже небольшой сложности количество функций и классов может измеряться десятками, поэтому обычно проект обычно состоит из нескольких файлов с исходным текстом.

Файловый уровень проекта определяет, как организованы файлы с исходным текстом, данными и скриптами сборки.

Рассмотрим назначение различных файлов (табл. 1), участвующих в проекте на языках C и C++, собираемых с помощью утилит **make** и **cmake** [13–15].

Таблица 1

Назначение файлов с различными расширениями

№ п/п	Расширение файла	Назначение	Примечание
1	*.c	Исходный текст на языке С	Текстовый файл
2	*.cxx *.cpp *.c++	Исходный текст на языке С++	Текстовый файл
3	*.h	Заголовочный файл	Текстовый файл
4	Makefile makefile	Сборочный скрипт, обрабатываемый утилитой make	Текстовый файл
5	CMakeLists.txt	Сборочный скрипт, обрабатываемый утилитой cmake. Используется для автоматической генерации Makefile	Текстовый файл
6	*.i *.ii	Файл, обработанный препроцессором	Текстовый файл
7	*.s *.S *.asm	Файл на языке ассемблера	Текстовый файл
8	*.o	Несвязанный объектный код.	Бинарный файл
9	*.*	Исполняемый файл. В UNIX-систе-мах не имеет специального расширения. В виндовс – *.com или *.exe	Бинарный файл
10	*.bin	Образ памяти, например для программирования встроенного ПЗУ специальной утилитой	Бинарный файл
11	*.hex	Образ памяти, например для программирования встроенного ПЗУ специальной утилитой	Текстовый файл

Отметим, что в табл. 1 показаны общепринятые расширения файлов, но проектировщик может использовать и другие, более удобные ему расширения, например для отделения файлов по каким-либо признакам.

Настоятельно рекомендуем придерживаться общепринятых расширений файлов, поскольку большинство текстовых редакторов и иных программ ориентированы именно на эти расширения.

На основе обширной практики работы с программами на С и С++ дадим несколько рекомендаций по организации программ, полезных начинающим программистам, на языках С и С++.

1. Один файл – один класс для языка С++ (один файл – сходные по назначению функции для языка С).

Это поможет вам быстро найти в дереве проекта нужную функцию или класс, а также выделить при необходимости в отдельную библиотеку функции сходного назначения.

2. Больше библиотек – проще проект. Сходные взаимосвязанные функции или классы необходимо выделять в функционально законченные библиотеки.

Такой подход особенно эффективен, когда в проекте есть несколько исполняемых файлов, использующих одни и те же функции (классы).

Кроме того, функционально законченные библиотеки с лёгкостью могут быть использованы в других проектах без изменений.

Библиотека, как правило, выделяется в отдельный подкаталог в дереве файлов проекта.

3. В заголовочных файлах должны быть только объявления. Часто встречающаяся ошибка у начинающих – попытка описать тело функции или объявить глобальную переменную в заголовочном файле. Этого делать ни в коем случае нельзя, поскольку такой заголовочный файл можно будет включить только один раз в один из файлов с исходными текстами. Включение такого заголовочного файла в разные файлы с исходным текстом приведёт к ошибке линкера.

4. Всегда делайте защиту заголовочного файла от повторного включения. Если мы более одного раза включим заголовочный файл в файл с исходными текстом, то чаще всего это вызовет ошибку компилятора, который, обнаружив двойные объявления типов и констант, сгенерирует ошибку. Чтобы этого избежать, необходимо использовать условную компиляцию и в каждом заголовочном файле использовать следующую схему. Предположим, у нас есть заголовочный файл **tscr.h**. Тогда в начале файла необходимо проверить – определён ли уже некий уникальный символ. Назовём этот символ **tscr_h**. Если данный символ не определён, значит, файл ещё не включался. Если определён – то файл уже использовался и его содержимое компилировать не надо.

```
/** @file tscr.h Функции экранного вывода.
*/
#ifndef tscr_h        /* Проверка – определён ли сим-
вол tscr_h */
#define tscr_h    /* Символ не определён.
          Определяем символ tscr_h и компилируем тело
          заголовочного файла*/
/* Тело заголовочного файла */
… … … … … … … … … … … … …
… … … … … … … … … … … … …
#endif /* Конец заголовочного файла */
```

Для более глубокого изучения организации проектов рекомендуем изучить [13, 14].

4.2. Трансляция программы на языках C и C++

Практически все проекты на языках C и C++ состоят из нескольких файлов с исходным текстом и подключаемых стандартных или собственных библиотек.

Чтобы ориентироваться в дереве исходных текстов проекта и представлять различные источники ошибок, необходимо не только знать языки C и C++, но и чётко представлять, каким образом происходит трансляция программ от исходного текста до исполняемого файла.

В данном пособии все примеры ориентированы на пакет компиляторов **GCC**.

Пакет компиляторов **GCC** генерирует исполняемые файлы в формате **elf.** В дальнейшем исполняемый файл, если это необходимо, преобразуется в требуемый формат утилитой **objcopy**, входящий в тот же пакет **GCC**.

Рассмотрим стадии трансляции программы от исходного текста до исполняемого кода пакета компиляторов **GCC** (рис. 8).

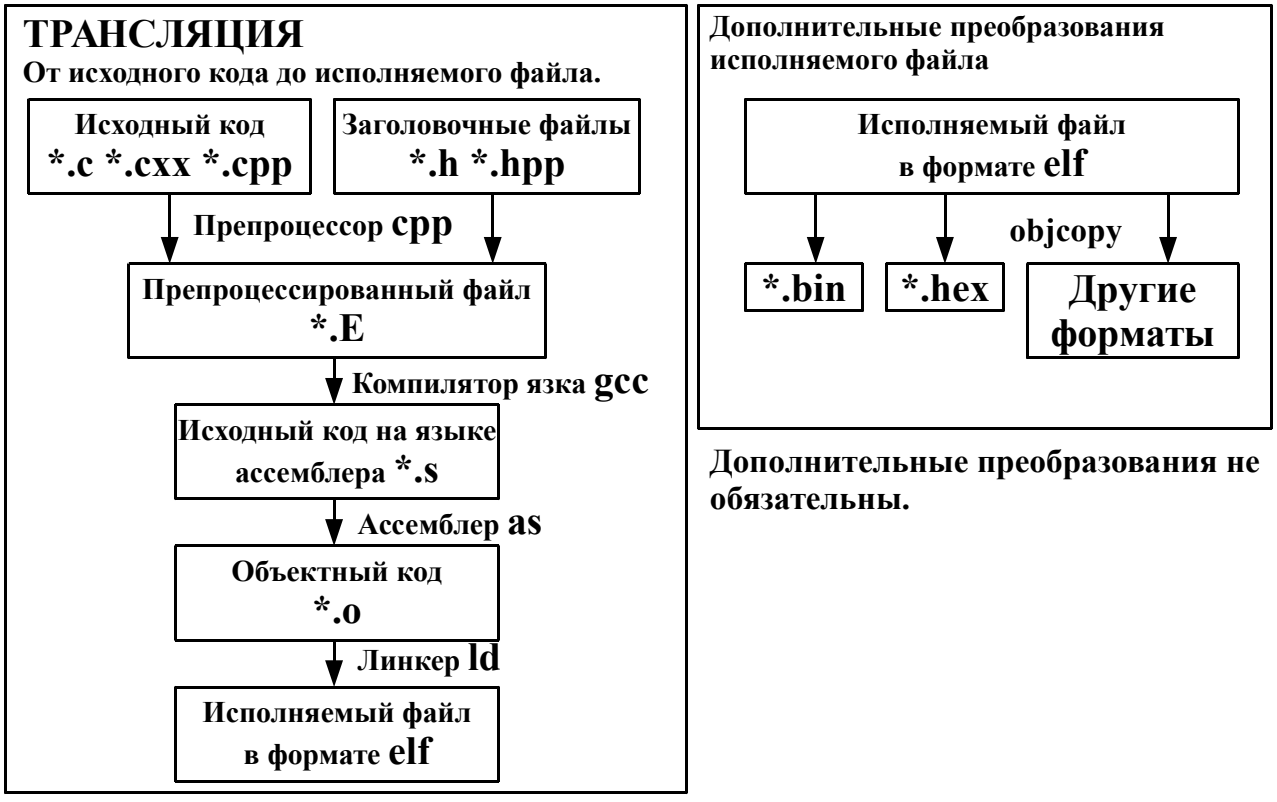

Рис. 8. Трансляция программы на языках C и C++

Препроцессор. Первая стадия трансляции – препроцессинг исходного текста. В пакете компиляторов **GCC** препроцессор называется **cpp**. Задача препроцессора – изменить исходный код программы в соответствии с директивами препроцессора.

Все директивы препроцессора языков C и C++ начинаются с символа # (решётка).

Список основных директив препроцессора с примерами приведён в табл. 2. В различных реализациях препроцессоров возможны различные дополнительные директивы, но с целью улучшения переносимости кода рекомендуем по возможности использовать только директивы, рекомендованные стандартом **ISO/IEC 9899**.

Таблица 2

Директивы препроцессора языков C и C++

1	`#define`	Задаёт макроопределение (макрос) или символическую константу
2	`#undef`	Отменяет макроопределение (макрос) или символическую константу, заданные ранее директивой **#define**
3	`#include`	Вставляет текст из указанного файла
4	`#if`	Осуществляет условную компиляцию при истинности константного выражения
5	`#ifdef`	Осуществляет условную компиляцию, если заданная символическая константа определена
6	`#ifndef`	Осуществляет условную компиляцию, если заданная символическая константа не определена
7	`#else`	Ветка условной компиляции при ложности выражения
8	`#elif`	Ветка условной компиляции, образуемая слиянием #else и #if
9	`#endif`	Конец ветки условной компиляции
10	`#error`	Выдача диагностического сообщения об ошибке и прекращение трансляции программы

Подробное описание директив препроцессора языков C, C++ и примеры их использования можно найти, например, в [16].

Компилятор. Вторая стадия трансляции программы – компиляция. Компиляция – это преобразование препроцессированного исходного кода на языках C и C++ в исходный код на языке ассемблера. Пакет **GCC** имеет различные компиляторы для языков **C** и **C++**: **cc** для языка **C** и **c++** для языка **C++**.

Ассемблер. Исходный код на языке ассемблера, полученные в результате компиляции, преобразуется в объектный код с помощью ассемблера **as**, входящего в пакет **GCC**. Объектный код – это исполняемый двоичный код с указанием секций и неопределёнными абсолютными адресами меток. Вместо меток используются символические имена.

Линкер. Полученные после ассемблирования один или несколько файлов с объектным кодом поступают на вход компоновщика (линкера) **ld**. В соответствии с заданными ключами линкер создаёт объектный файл или разделяемую библиотеку. Задача компоновщика – связать все объектные файлы, т.е. сгенерировать таблицу адресов меток и заменить символические имена меток на абсолютные адреса.

Выходом компоновщика является исполняемый файл или разделяемая библиотека в формате **elf**.

Дополнительные преобразования. Если требуется получить исполняемый код в формате, отличном от **elf**, то его необходимо преобразовать в требуемый формат утилитой **objcopy**. Например, программаторы процессоров **AVR** обычно требуют исполняемый код в форматах **hex** или **bin**.

4.3. Утилита make

Сборка программного обеспечения в общем случае включает в себя не только трансляцию исходных кодов программы, но и большое число дополнительных действий, например:

- конфигурирование глобальных настроек проекта, например путей к файлам, с которыми работает программа, или констант;
- конфигурирование библиотек под конкретный проект;
- генерацию документации к исходному коду, например с помощью таких систем, как Doxygen;
- генерацию версий ПО
- и т.д.

Для автоматизации этих действий применяются программы автоматического преобразования файлов, такие как утилита **make**.

Утилита **make** – это программа, автоматизирующая процесс преобразования файлов из одной формы в другую. Пример такого преобразования – компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Преобразование файлов производится с помощью скрипта, описанного в так называемом «мэйк-файле», который обычно называется **Makefile** или **makefile**. **Makefile** – это текстовый файл специального формата, в котором описаны три вида сущностей: ***цели***, ***зависимости*** и ***команды***. Кроме этого, в **Makefile** могут определяться переменные.

Цель – это то, что должно получиться после выполнения определённой ***команды***. Целью может быть как выходной файл, так и абстрактная цель (например, удаление ненужных более файлов, запуск программы и т.д.).

Зависимость – это что должно быть сделано **до того**, как начнёт выполнятся ***команда***, создающая данную ***цель***.

Команда – действие, которое производит ***цель***. ***Команда*** может быть составной и состоять из любых команд интерпретатора **shell**.

Утилита **make** проверяет, имеются ли зависимости заданной цели. Если зависимости отсутствуют, то эта утилита сначала пытается их создать и только затем приступает к генерации цели.

Для понимания принципов работы утилиты **make** рассмотрим несколько примеров.

4.3.1. Простейший сборочный скрипт

Предположим, имеется программа на языке C++, состоящая из исходных файлов **main.cxx** и **main.h**. Необходимо получить исполняемый файл **main**.

Для компиляции программы создадим сборочный скрипт **Makefile**, который должен находиться в том же каталоге, что и файлы **main.cxx** и **main.h**. Содержимое сборочного скрипта **Makefile** приведено ниже:

```
# Основная цель
all: main.cxx main.h
     g++ -o main main.cxx

# Цель – очистка проекта
clean:
     rm -f main *.o
```

Скрипт содержит две цели – **all** и **clean**. Имя цели заканчивается двоеточием, за которым следует список зависимостей. Список зависимостей может быть пустым, как в описании цели **clean**.

Команды для генерации цели отделяются символом табуляции от начала строки, который выделен цветом.

Для сборки программы необходимо набрать из командного интерпретатора **shell** команду

```
# make all
```

Эта команда говорит, что необходимо вызвать программу **make** с параметром **all**, означающим имя цели, которую необходимо сгенерировать.

Поскольку выполнение **Makefile** начинается с первой встреченной цели, то в данном случае можно набрать просто команду **make**:

```
# make
```

Для того чтобы сгенерировать цель **all**, утилита **make** проверит наличие зависимостей – файлов **main.cxx** и **main.h**.

Если какого-либо из этих файлов не существует в текущем каталоге, то утилита **make** выдаст ошибку и завершится.

Если оба этих файла имеются в текущем каталоге, то утилита **make** выполнит команду для генерации цели, т.е. «**g++ -o main main.cxx**».

Эта команда – вызов транслятора языка C++, который называется **g++**. Ключ **-o main** – говорит о том, что имя выходного исполняемого файла – **main**. Если трансляция прошла без ошибок, то в текущем каталоге появится исполняемый файл **main**.

В проекте часто возникает необходимость удалить все автоматически сгенерированные файлы. Для этого служит цель, обычно называемая **clean** (очистка).

В нашем примере данная цель не имеет зависимостей и вызывает команду удаления исполняемого файла **main** и объектных файлов, имеющих расширение ***.o**. Для очистки проекта необходимо набрать в командной строке.

```
# make clean
```

4.3.2. Трансляция проекта по частям

Большие проекты содержат десятки, сотни и даже тысячи файлов с исходным кодом. Трансляция такого проекта занимает длительное время. С целью экономии времени трансляции был разработан алгоритм, который позволяет транслировать такие проекты не полностью, а по частям. Это достигается путём того, что каждый файл исходного текста транслируется отдельно от остальных до стадии объектного кода, а затем все файлы объектного кода линкуются в исполняемый файл.

Таким образом, можно транслировать не все файлы проекта, а только те, в которых произошли изменения, что существенно сокращает время сборки проекта.

Рассмотрим такой подход на примере. Предположим, что у нас имеются файлы с исходным текстом **module1.cxx, module2.cxx, defs.h** и необходимо создать исполняемый файл **application**. Сборочный скрипт **Makefile** будет выглядеть так:

```
# Основная цель
all:  module1.o module2.o
     g++ -o application module1.o module2.o

module1.o: module1.cxx  defs.h
     g++ -c -o module1.o  module1.cxx

module2.o: module2.cxx  defs.h
     g++ -c -o module2.o  module2.cxx

# Цель – очистка проекта
clean:
     rm -f  *.o  application
```

Рассмотрим процесс исполнения данного сборочного скрипта. Цель **all** зависит от наличия двух файлов – **module1.o** и **module2.o**, которые в свою очередь сами являются целями, зависящими от заголовочного файла **defs.h** и соответствующих файлов с исходным текстом – **module1.cxx** и **module1.cxx**.

Если цель является файлом и такой файл уже существует, то утилита **make** проверит время последнего изменения всех её зависимостей и выполнит команду генерации цели только в том случае, если хотя бы одна зависимость изменилась позже, чем была последний раз сгенерирована цель. Иначе считается, что цель не нуждается в генерации.

Например, если был изменён файл **module2.cxx**, то будет перетранслирован только файл **module2.o**, а файл **module1.o** останется неизменным.

4.4. Трансляция проекта под различные платформы

При разработке ПО для различных МПУ встаёт проблема кросс-трансляции программ. Кросс-трансляция программ – это трансляция программ на одной платформе, в результате которой получается исполняемый код для другой платформы.

Например, на ПК с процессором семейства **x86** можно транслировать один и тот же исходный код на языке **C** в исполняемый код для различных платформ на базе **ARM**, **AVR** и д.

Пакет компиляторов **GCC** имеет поддержку кросс-трансляции программ для множества платформ. При этом все утилиты пакета используются единообразно, независимо от того, под какую платформу идёт трансляция программы.

Трансляторы **GCC** для различных платформ отличаются префиксом в названии. Например, транслятор для ПК обычно называется **gcc**. Транслятор для архитектуры **ARM** под управлением ОС Linux называется **arm-none-linux-gnueabi-gcc**. Транслятор для платформы AVR называется **avr-gcc**.

Таким образом, для трансляции программы под другую платформу часто достаточно изменить название компилятора.

В сборочном скрипте **Makefile**, рассчитанном на трансляцию программы под несколько платформ, обычно задают некоторый префикс, который позволяет оперативно менять название транслятора, как показано в следующем примере:

```
# CROSS – префикс названия транслятора
CXX=$(CROSS)g++

# Основная цель
all:  module1.o module2.o
     $(CXX) -o application module1.o module2.o

module1.o: module1.cxx  defs.h
     $(CXX) -c -o module1.o  module1.cxx

module2.o: module2.cxx  defs.h
     $(CXX) -c -o module2.o  module2.cxx

# Цель – очистка проекта
clean:
     rm -f  *.o  application
```

Если префикс **CROSS** не задан (пустая строка), то сборка проекта будет происходить транслятором gcc.

Если задать префикс **CROSS**:

```
# make  CROSS=arm-none-linux-gnueabi-    all
```

то сборка будет производиться транслятором **arm-none-linux-gnueabi-gcc** для платформы ARM.

В сборочном скрипте **Makefile**, рассчитанном на трансляцию программы под несколько платформ, всегда следует задавать имена утилит, относящихся к транслятору не напрямую, а через переменную, как было показано в примере.

4.5. Пример проекта из нескольких модулей

Создадим универсальный **Makefile**, с помощью которого можно собирать несколько исполняемых файлов и статических библиотек.

Предположим, что у нас имеется проект на языке **C++**, содержащий две библиотеки – **libtcp** и **libudp**, и на выходе должно быть два исполняемых файла – **tcpcs** и **udpcs**. Для удобства исходный код каждой библиотеки и каждого исполняемого файла поместим в отдельный каталог. В этом случае дерево исходных текстов проекта будет выглядеть так, как показано на рис. 9.

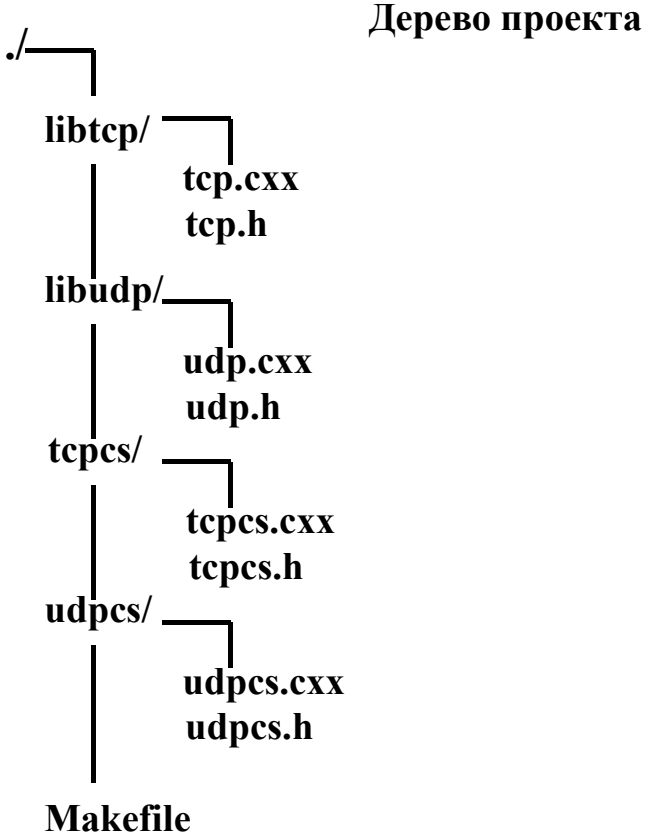

Рис. 9 Дерево проекта (пример)

Пусть имя выходного файла – библиотеки или исполняемого файла – совпадает с именем каталога, в котором он находится. Это позволяет

удобно и быстро расширять проект, добавляя в него новые библиотеки и исполняемые файлы.

Определим требования к создаваемому сборочному скрипту **Makefile**:

- расширяемость проекта. То есть должна иметься возможность добавлять в проект новые библиотеки и программы. Причём необходимо, чтобы добавляемым программам и библиотекам автоматически указывались пути ко всем используемым библиотекам и заголовочным файлам;
- использование сторонних библиотек. Большинство проектов используют стандартные или сторонние библиотеки, поэтому необходимо организовать в Makefile возможность указания пути к заголовочным файлам и библиотекам, не входящим непосредственно в проект;
- возможность кросс-трансляции. Поскольку программы должны исполняться на различных МПУ, то возможность кросс-трансляции необходима.

Условимся, что все библиотеки, входящие в проект, будут находиться в каталогах, начинающихся с префикса **lib***. Это позволит, во-первых, быстро визуально отличать каталоги с исходными кодами библиотек от каталогов с исходными кодами исполняемых файлов, а во-вторых, автоматизировать обработку каталогов с различным содержимым.

В проект могут входить: исходные файлы на языке **C++** (имеют расширение ***.cxx**); заголовочные файлы (имеют расширение ***.h**); дополнительные сторонние библиотеки, не являющиеся частью проекта, но подключаемые к нему; заголовочные файлы, не являющиеся частью проекта, но подключаемые к нему.

Таким образом, **Makefile** должен обеспечить следующие действия:

- сборку библиотек, входящих в проект;
- сборку программ, входящих в проект;
- автоматическое указание пути к заголовочным файлам библиотек, входящих в проект, при сборке библиотек и программ;
- автоматическое подключение библиотек, входящих в проект, при сборке программ;
- автоматическое указание пути к заголовочным файлам, не являющихся частью проекта, при сборке библиотек и программ;
- автоматическое подключение библиотек, не являющихся частью проекта, при сборке программ.

Пример **Makefile**, осуществляющего перечисленные действия, приведён ниже:

```
#
# Настройки проекта
#
```

```
# Префикс для кросс-компиляции
CROSS=

# Исполняемые файлы.
# Их имена совпадают с именами каталогов
APPS=tcpcs udpcs

# Библиотеки пользователя
# Их имена совпадают с именами каталогов без префикса
<lib>
APPLIBS=tcp udp

# Пути к заголовочным файлам, не входящим в проект
STDINC=

# Библиотеки, не входящие в проект
STDLIBS=

#
# Неизменная часть Makefile
#

# Корневой каталог дерева проекта
TOP=$(shell pwd)

CXX=$(CROSS)g++ -c -std=c++0x
LD=$(CROSS)g++
AR=$(CROSS)ar

# Пути к подключаемым *.h – файлам
INCPATH=$(addprefix -I$(TOP)/lib, $(APPLIBS) )
INCPATH+=$(STDINC)

# Подключение библиотек
LIBS=$(addprefix -L$(TOP)/lib, $(APPLIBS) )
LIBS+=$(addprefix -l, $(APPLIBS) )
LIBS+=$(STDLIBS)

all: libs apps
     @echo "- Конец сборки -"

# Сборка библиотек
libs:
     @echo "Пути к заголовочным файлам: $(INCPATH)"
     @for i in $(APPLIBS); do \
         cd "$(TOP)/lib$$i" ; \
         echo "Сборка библиотеки lib$$i.a" ; \
         $(CXX) $(INCPATH) *.cxx ; \
         $(AR) cr "lib$$i.a" *.o ; \
     done
     @cd "$(TOP)"
```

```
apps:
	@echo "Пути к заголовочным файлам: $(INCPATH)"
	@echo "Подключаемые библиотеки: $(LIBS)"
	@for i in $(APPS); do \
	    cd "$(TOP)/$$i" ; \
	    echo "Сборка программы $$i" ; \
	    $(CXX) $(INCPATH) *.cxx ; \
	    $(LD) *.o $(LIBS) -o "$$i" ; \
	    done
	@cd "$(TOP)"

clean:
	@# Удаление библиотек
	@for i in $(APPLIBS); do \
	    cd "$(TOP)/lib$$i" ; \
	    rm -f *.o *.a ; \
	    done
	@cd "$(TOP)"
	@# Удаление программ
	@for i in $(APPS); do \
	    cd "$(TOP)/$$i" ; \
	    rm -f *.o "$$i" ; \
	done
	@cd "$(TOP)"
```

Рассмотрим содержимое представленного **Makefile**. Условно данный **Makefile** можно разбить на две части, помеченные комментариями «Настройки проекта» и «Неизменная часть Makefile».

«Настройки проекта» – это часть Makefile, где указывается, какие именно библиотеки и программы собирать, нужна ли кросс-трансляция проекта и какие не входящие в проект библиотеки и заголовочные файлы должны быть подключены.

«Неизменная часть Makefile» предназначена для автоматической генерации библиотек и программ на основе данных, указанных в настроечных переменных проекта, и в идеале не должна изменяться от проекта к проекту.

Отметим, что представленный Makefile предназначен для сборки небольших проектов, поскольку не оптимален с точки зрения скорости сборки программ.

Настройка проекта осуществляется с помощью настроечных переменных. Рассмотрим назначение каждой из этих переменных.

CROSS – переменная определяет необходима ли кросс-трансляция проекта. В нашем примере данная переменная задана как пустая строка, т.е. будут использоваться стандартные средства трансляции – **g++**, **ar**. Если необходимо осуществить кросс-трансляцию проекта, например для

платформы ARM, то необходимо указать **CROSS**=<префикс утилит для кросс-трансляции>, например:

```
CROSS=arm-none-linux-gnueabi-
```

APPS – указывает, какие программы необходимо собирать в проекте. В нашем примере это программы **tcpcs** и **udpcs**. Заметим, что данный **Makefile** требует того, чтобы имя программы совпадало с именем каталога, содержащего исходные коды для данной программы. Предположим, что нам надо добавить в проект программу **newprogram**. Для этого необходимо создать в дереве проекта каталог **newprogram** с исходными кодами новой программы и добавить в описание переменной **APPS** следующее описание:

```
APPS=tcpcs udpcs  newprogram
```

После этого при исполнении команды **make** программа **newprogram** будет собрана.

APPLIBS – указывает библиотеки, входящие в проект и собираемые совместно с программами. Обратите внимание, что имена библиотек, указанные в **APPLIBS**, не содержат префикса **lib**. Предположим, что нам необходимо добавить в проект новую библиотеку с именем **libnewfunc**. Для этого необходимо создать в дереве проекта каталог **libnewfunc** с исходными кодами новой библиотеки и добавить в описание переменной **APPLIBS** следующее описание:

```
APPLIBS=tcp udp newfunc
```

После этого при исполнении команды **make** библиотека **libnewfunc** будет собрана, пути для подключения заголовочных файлов из каталога **libnewfunc** будут указаны при сборке всех остальных библиотек и программ, а сама библиотека подключена ко всем собираемым программам.

STDINC – указывает пути к заголовочным файлам, которые требуются проекту, но не входят в него. Например, это могут быть заголовочные файлы сторонних библиотек. Предположим, что нам необходимо использовать сторонние библиотеки, заголовочные файлы которых находятся в каталогах **/usr/include/extlib1** и **/usr/include/extlib2**. Для использования этих заголовочных файлов необходимо указать в переменной **STDINC** пути к заголовочным файлам, вместе с ключом **–I** транслятора **g++**:

```
STDINC=-I/usr/include/extlib1 - I/usr/include/extlib2.
```

STDLIBS – указывает пути к библиотекам и библиотеки, требуемые в проекте, но не входящие в сам проект. Например, это могут быть сторонние библиотеки. Предположим, что нам необходимо использовать сторонние библиотеки, находящиеся в каталогах **/usr/lib/extlib1** и

/usr/lib/extlib2 и имеющие названия **libext1** и **libext2**. Для использования этих библиотек необходимо указать в переменной **STDLIBS** пути поиска библиотек совместно с ключом **-L** и имена библиотек без префикса **lib** совместно с ключом **-l**:

```
STDLIBS=-L/usr/lib/extlib1 -L/usr/lib/extlib2 -lext1 -lext2
```

Рассмотрим, как происходит выполнение сборочного скрипта **Makefile** в нашем примере. Приведённый сборочный скрипт имеет несколько целей: **all**, **libs**, **apps**, **clean**.

Цель **libs** – осуществляет трансляцию библиотек, входящих в проект. Трансляция осуществляется для всех каталогов, входящих в список библиотек **APPLIBS**. При этом каждой из создаваемых библиотек доступны заголовочные файлы всех других библиотек, что позволяет сделать библиотеки проекта взаимозависимыми.

Цель **apps** – осуществляет трансляцию программ, входящих в проект. Трансляция осуществляется для всех каталогов, входящих в список программ **APPS**. При этом каждой из создаваемых программ доступны заголовочные файлы всех библиотек проекта, что позволяет не указывать их вручную.

Цель **all** – основная, осуществляющая трансляцию всего проекта – библиотек и программ. Цель **all** зависит от целей **libs** и **apps**, то есть выполнение команды

```
# make  all
```

вызовет сначала выполнение цели **libs**, затем выполнение цели **apps**. То есть сначала будет произведена сборка библиотек проекта, а затем программ.

Цель **clean** – очищает проект, т.е. удаляет все созданные объектные файлы, библиотеки и программы.

При создании собственного **Makefile** могут быть добавлены дополнительные цели. Например, цель **install**, позволяющая инсталлировать созданные программы в указанный каталог, цель **flash**, позволяющая вызвать команду программирования созданных бинарных файлов во FLASH-память контроллера и т.п. Называть цели можно произвольным образом, но лучше использовать говорящие имена, такие как **install**, **flash**, **apps**, **libs**.

При написании и изменении Makefile рекомендуем ознакомиться с литературой [13–15].

5. ПРИМЕРЫ ТИПОВЫХ ПРОГРАММНЫХ РЕШЕНИЙ ДЛЯ POSIX-СОВМЕСТИМЫХ ОС

Стандарты POSIX (*Portable Operating System Interface for Unix – переносимый интерфейс операционных систем Unix*) [17] разработаны с целью обеспечить переносимость программ между различными ОС на уровне исходного кода. Стандарты POSIX описывают интерфейсы между операционной системой и прикладной программой. Такие интерфейсы называются также API (*Application Program Interface – интерфейс прикладных программ*). То есть программа, использующая API POSIX, в идеальном случае должна одинаково транслироваться и исполняться на любых POSIX-совместимых ОС.

Список основных задач, решение которых должен обеспечить стандарт POSIX:

- облегчение переноса кода прикладных программ между платформами;
- определение и унификация интерфейсов на этапе проектирования, а не в процессе их реализации;
- определение необходимого минимума интерфейсов прикладных программ.

В данном учебнои пособии все программы для ОС Linux написаны с использованием POSIX-API, поэтому их можно транслировать, запускать и отлаживать на практически любой Unix-платформе, а не только на ВИП-МК.

В учебных целях ниже приводится пример разработки простого протокола обмена данными и реализации программ для его обработки.

5.1. Протокол обмена данными между МПУ

При разработке любой РМПС необходимо определить протоколы обмена данными между МПУ. Предположим, что необходимо разработать протокол обмена, отвечающий нескольким требованиям:

- протокол должен быть пакетно-ориентированный, адресный;
- возможность обмена пакетами с подтверждением или без подтверждения;
- каждый пакет должен содержать уникальный идентификатор;
- протокол должен иметь средства контроля целостности пакета.

Рассмотрим каждое требование к протоколу.

Протокол должен быть пакетно-ориентированный, адресный. Поскольку РМПС имеет в своём составе несколько МПУ, то необходима их адресация для указания приёмника (получателя) и источника (отправителя) каждого пакета данных.

Следует учесть, что различные каналы связи имеют различные системы адресации. Например, в сети Интернет имеется адрес IP, в телефонной сети адресом является уникальный телефонный номер и т.п.

Так как РМПС в общем случае использует для передачи данных различные каналы связи, то нельзя привязать МПУ к системе адресации какого-либо одного канала связи.

Чтобы не привязываться к системе адресации какого-либо одного канала связи, присвоим каждому МПУ уникальный в данной РМПС номер. Данный номер позволяет однозначно идентифицировать отправителя и получателя пакета данных, что является удобным с точки зрения функций обработки пакетов данных. Будем называть этот уникальный номер МПУ «**адресом МПУ**». Адрес МПУ, привязанный к системе адресации конкретного канала связи, будем называть «**канальным адресом МПУ**».

В связи с тем, что у каждого канала связи в общем случае своя система адресации, возникает задача преобразования адреса МПУ в канальный адрес МПУ. Рассмотрим два способа решения этой задачи, зависящих от структуры РМПС.

Способ 1. РМПС с равноправной структурой. В такой РМПС каждое МПУ может передать пакет данных любому другому МПУ. Поэтому, в общем случае, каждое МПУ должно хранить таблицу преобразования адресов, в которой для каждого канала связи указаны адрес МПУ и соответствующий ему канальный адрес МПУ.

Достоинство такого подхода – универсальность. Недостатки – большой объём таблицы преобразования адресов и сложность замены таблиц на всех МПУ, входящих в РМПС, при добавлении, удалении или изменении номера хотя бы одного из МПУ.

Способ 2. РМПС с иерархической структурой. В такой РМПС каждое МПУ может передать пакет данных непосредственно только на уровень ниже (одному из подчинённых МПУ) или на уровень выше (вышестоящему центру). Таким образом, МПУ должно хранить таблицу преобразования адресов только для непосредственно связанных с ним других МПУ – подчинённых и вышестоящего центра. Это позволяет сократить размер таблицы преобразования адресов по сравнению со способом 1.

Передача пакета данных между двумя произвольными МПУ в РМПС с иерархической структурой осуществляется в общем виде путём пересылки через несколько уровней иерархии, что влечёт за собой появление задачи поиска маршрута передачи пакета данных.

Выберем размер адреса МПУ для нашего протокола равным 32 бита, что позволяет обеспечить адресацию до $2^{32} = 4294967296$ МПУ.

Возможность обмена пакетами с подтверждением или без подтверждения. В зависимости от типа передаваемой информации и конкретной задачи получатель может подтверждать или не подтверждать получение пакета данных.

Как правило, не требуют подтверждения, например, широковещательные пакеты и пакеты, содержащие аудио- и видеоинформацию, передаваемую в реальном времени.

Таким образом, в пакете данных необходимо иметь следующую информацию:

- требует пакет данных подтверждения о получении или нет;
- является ли пакет пакетом с данными для получателя или подтверждением для отправителя.

Введём два флага, передаваемых с каждым пакетом данных:

QACK – размер 1 бит, если сброшен в 0, то пакет не требует подтверждения, если установлен в 1, то пакет требует подтверждения о доставке;

ACK – размер 1 бит, если сброшен в 0, то пакет является пакетом с данными, если установлен в 1, то пакет является подтверждением о доставке. Если флаг ACK = 1, то флаг QACK всегда считается равным 0, так как подтверждение не требует подтверждения.

Каждый пакет должен содержать уникальный идентификатор с целью отличия его от других пакетов, передаваемых между двумя МПУ. Уникальный идентификатор служит для диагностики пропадания конкретного пакета данных в канале связи, диагностики повторной доставки пакета данных, а также для идентификации подтверждений доставки данных. Обычно, в качестве уникального идентификатора пакета выступает псевдослучайное число, которое пересылается вместе с пакетом данных. Уникальные идентификаторы пакета и подтверждения о доставке этого пакета данных совпадают.

Протокол должен иметь средства контроля целостности пакета. Каналы связи не являются надёжными и информация при передаче по ним может искажаться вследствие многих причин. Для отбраковки искажённых пакетов необходим один из способов контроля целостности данных в пакете. Используем широко распространённый алгоритм контроля целостности путём вычисления 16-разрядной циклической контрольной суммы пакета **CRC16 [**11**]**. При передаче пакета МПУ-отправитель вместе с данными передаёт и контрольную сумму пакета. При приёме пакета МПУ-получатель вычисляет **CRC16** принятого пакета и сравнивает её с принятой от МПУ-отправителя контрольной суммой. Если вычисленная и принятая контрольные суммы **CRC16** не совпадают, то считается что пакет принят с искажениями, и такой пакет игнорируется.

Построим структуру пакета данных на основании принятых соображений (табл. 3).

Таблица 3

Структура пакета данных протокола передачи данных

DA	**SA**	**F**	**ID**	**LEN**	**DATA**	**CRC16**
4 байта	4 байта	1 байт	1 байт	2 байта	LEN байт	2 байта
Заголовок					Данные	Контроль

Описание полей пакета:

DA – адрес приёмника, т.е. МПУ-получателя пакета.

SA – адрес источника, т.е. МПУ-отправителя пакета.

F – флаги:

ACK – 0-й бит, размер 1 бит, если сброшен в 0, то пакет не требует подтверждения, если установлен в 1, то пакет требует подтверждения о доставке;

QACK – 1-й бит, размер 1 бит, если сброшен в 0, то пакет является пакетом с данными, если установлен в 1, то пакет является подтверждением о доставке. Если флаг ACK=1, то флаг QACK всегда считается равным 0, так как подтверждение не требует подтверждения.

Биты 2–7 – резерв, всегда равны 0.

ID – уникальный идентификатор пакета.

LEN – длина данных (поля **DATA**) в байтах. **LEN** = [0 ... 65535].
Если **LEN** = 0, то поле **DATA** отсутствует.

CRC16 – циклическая контрольная сумма полей **DA, SA, F, ID, LEN, DATA**.

Разработанный протокол обмена данными фактически описывает только транспортный уровень согласно рис. 3. Нижележащие уровни обеспечиваются драйвером конкретного оборудования и протоколами нижнего уровня (если таковые есть).

На вышележащий, прикладной, уровень передаются данные (поле **DATA**), а также дополнительная информация в виде полей **SA, DA, ID**.

5.2. Алгоритмы обработки пакетов данных

Составим общие алгоритмы обработки пакетов данных разработанного протокола обмена данными между МПУ.

5.2.1. Приём пакета данных

Алгоритм приёма пакета данных представлен на рис. 10.

Поясним некоторые обозначения на представленном алгоритме:

CRC – вычисленная циклическая контрольная сумма пакета данных;

CRC16 – принятая вместе с пакетом данных циклическая контрольная сумма;

DA – адрес приёмника, принятый в пакете данных;

Adr – адрес МПУ, принимающего пакет.

Двойной рамкой отмечены блоки алгоритма, осуществляющие обработку принятого пакета данных.

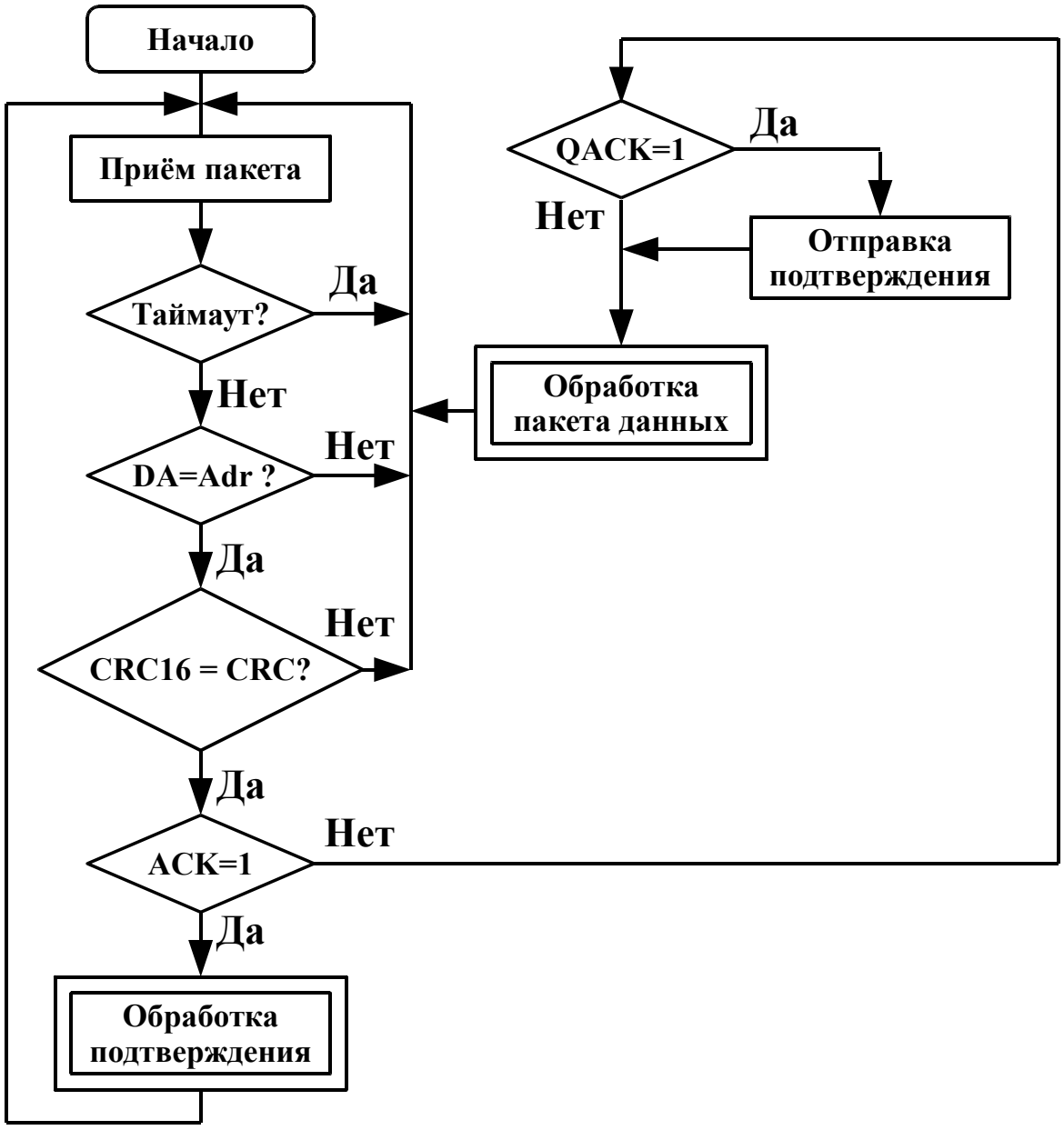

Рис. 10. Алгоритм приёма пакета данных

5.2.2. Передача пакета данных

Алгоритм передачи пакета данных (рис. 11) зависит от режима передачи – с подтверждением или без подтверждения.

Тип передачи пакета данных (с подтверждением или без подтверждения) определяется по флагу **QACK** в заголовке пакета.

Если передача пакета производится без подтверждения (QACK = 0), то после передачи пакета в канал связи алгоритм завершается.

Если пакет требует подтверждения о доставке (QACK = 1), то после передачи пакета в канал связи некоторое время (тайм-аут) ожидается приём подтверждения на данный пакет. При получении подтверждения алгоритм завершается. Если подтверждение не получено в течение заданного времени, то фиксируется ошибка передачи, которая обрабатывается обработчиком ошибок.

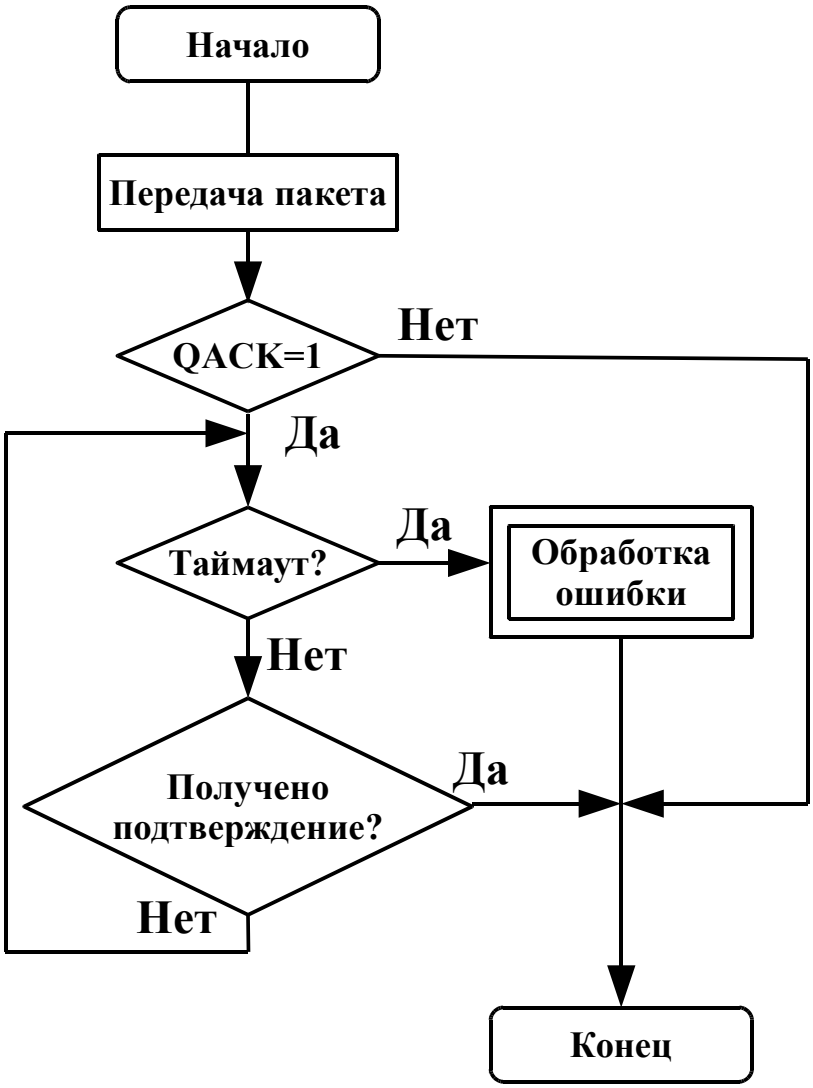

Рис. 11. Алгоритм передачи пакета данных

5.3. Организация ввода-вывода в POSIX-совместимых ОС

В UNIX-подобных POSIX-совместимых ОС исторически применяется концепция «всё есть файл». То есть с точки зрения программ пользователя как обычные файлы на диске, так и устройства ввода-вывода, сокеты, пайпы для межпроцессного обмена представляются в виде файлов [14].

В UNIX-подобных системах существуют различные типы файлов – обычные файлы (т.е. данные на устройстве хранения), файлы устройств, файлы сокетов и пайпов и т.п.

Это позволяет программам пользователя абстрагироваться от конкретной реализации драйверов устройств и обращаться к ним с помощью стандартных вызовов чтения и записи – **open()**, **close()**, **read(), write()**.

Кроме того, имеются два специальных вызова для управления файлами и устройствами – **fcntl()** и **ioctl()**.

Вызов **fcntl()** позволяет выполнять различные аппаратно-независимые операции над свойствами открытого файла, такие как установка, проверка и снятие блокировок, изменения прав доступа и т.п.

Вызов **ioctl()** служит для изменения параметров специальных файлов устройств. Конкретные входные и выходные параметры зависят от типа устройства, над которым производится действие. Таким образом, системный вызов **ioctl()** служит для настройки аппаратно-зависимых параметров устройства, например скорости последовательного порта.

При организации ввода-вывода в UNIX-подобных ОС можно выделить несколько этапов.

Этап 1. Инициализация файла ввода-вывода. Как уже говорилось, в UNIX-подобных ОС любое устройство представлено в виде файла. Также в виде файлов представляются и сетевые соединения – сокеты. Перед использованием файл должен быть открыт. Файлы-устройства открываются при помощи системного вызова **open()**, а файлы-сокеты создаются при помощи вызова **socket()**. Более подробно процесс инициализации файлов ввода-вывода будет рассмотрен ниже.

Этап 2. Проверка возможности операции ввода-вывода. UNIX-подобные ОС имеют специальный вызов **select()**, позволяющий установить состояние ожидания для задачи, пока не произойдёт заданного события в указанных файлах или не истечёт указанный интервал времени. К событиям, на которые реагирует вызов **select()**, относятся: появление данных для чтения в указанном файле, появление возможности записать в указанный файл, возникновение исключительных ситуаций для указанного файла. Такой подход позволяет не тратить процессорное время на те задачи, которые ожидают событий в заданных файлах.

Этап 3. Чтение из файла или запись в файл. Чтение из файлов осуществляется вызовом **read()**, запись в файл осуществляется вызовом **write()**. Это базовые системные вызовы, пригодные для работы с любыми файлами. Но, кроме этих вызовов, существуют и специализированные вызовы работы с сокетами, такие как **recv()** и **send()**.

Этап 4. Закрытие файла. После завершения работы с файлом необходимо освободить ресурсы, выделенные ОС. Завершение работы с файлом осуществляется системным вызовом **close()**.

Указанные четыре этапа работы с файлами практически всегда присутствуют при реализации алгоритмов ввода-вывода. Отметим, что после инициализации файла (этап 1) можно сколько угодно раз записывать и читать этот файл (этапы 2 и 3) и только потом закрыть его.

Ниже мы рассмотрим два примера организации ввода-вывода – для последовательного порта RS232/RS485 и для IP-сокета. Рассмотренные примеры являются схематичными и могут потребовать доработки для каждого конкретного случая.

5.4. Обмен данными по последовательному интерфейсу RS232

Порт RS232 является последовательным дуплексным интерфейсом для обмена данными [18]. На ПК порты RS232 также называются COM-портами. Порты RS232 в Linux представлены как терминальные устройства в каталоге /dev/ttyS0, /dev/ttyS1 и т.д. Работа с любым устройством RS232 состоит из следующих этапов:

– открыть файл-устройство (функцией **open()** или **fopen()**);

– настроить параметры порта RS232 с помощью системного вызова ioctl(). Настраиваются такие параметры, как скорость порта, наличие признака чётности, количество стоп-бит, режим работы порта;

– выполнить операции чтения из порта RS232 или записи в порт RS232;

– закрыть файл-устройство.

Рассмотрим примеры функций на языке С, которые осуществляют все указанные этапы работы с портом RS232.

5.4.1. Инициализация порта RS232 в режиме RAW

Существует два основных режима работы порта RS232:

– режим с преобразованием входного и выходного потоков данных;

– режим без преобразования входного и выходного потоков данных (так называемый режим RAW, т.е. только чтение и запись);

Режим с преобразованием входного и выходного потоков данных имеет множество настроек относительно того, как именно преобразуются данные, и служит главным образом для поддержки выносных терминальных устройств, подключаемых по порту RS232 напрямую или через модем. Примером такого устройства может служить эмулятор терминала HyperTerminal (ОС Windows) или эмуляторы терминалов Minicom, Putty (ОС Linux).

Режим без преобразования входного и выходного потоков данных (режим RAW) используется, когда необходимо передать байты между устройствами без всякой обработки. Заметим, что ничто не мешает использованию режима RAW для работы с выносными терминальными устройствами. В этом случае весь протокол обмена и преобразование данных реализуется на уровне программ пользователя.

На листинге ниже приведён пример функции **init_RS232()**, которая открывает файл-устройство порта RS232 и переключает его в режим RAW:

```
/*
      Открыть порт RS232 и установить режим RAW.
*/
```

```
#include <termios.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int init_RS232(const char* device_name){
  int fd;
  struct termios t;

  if ( ( fd=open ( indev,O_RDONLY ) ) <0 ) {
          // Ошибка
          return(-1);
      }

  if ( ! tcgetattr ( fd, &t ) ) {
      tinit_oldt_in = t;
      cfmakeraw ( &t );
      t.c_cc[VMIN]=1;
      t.c_cc[VTIME]=0;
  } else {
      // Ошибка
      close(fd);
      return(-1);
  }

  if ( tcsetattr ( fd, TCSAFLUSH, &t ) ){
      // Ошибка
      close(fd);
      return(-1);
  }

  return( fd );
}
```

Рассмотрим, как работает функция **init_RS232()**. Входным параметром функции является строка символов device_name, являющаяся полным путём к файлу-устройству порта RS232 (например, «/dev/ttyS0»). В случае успешного выполнения функция возвращает дескриптор файла-устройства, с которым в дальнейшем можно совершать все файловые операции – чтение, запись, установку параметров. Если произошла ошибка, то функция **init_RS232()** вернёт значение **–1**.

Чтобы начать работу с любым файлом, его необходимо открыть функцией **open()**. Если файл открыт без ошибок, то функция **open()** возвращает дескриптор файла (целое неотрицательное число). Если файл не удалось открыть (например, нет такого файла или нет прав доступа к этому файлу), то функция **open()** вернёт значение **–1**.

После того как файл-устройство порта RS232 открыт, его параметры устанавливаются с помощью функций библиотеки **termios**.

Функция **tcgetattr()** получает параметры указанного файла-устройства и сохраняет их в переменной **t**. Затем производится установка режима RAW с помощью специального макроса **cfmakeraw()** и установка новых параметров с помощью функции **tcsetattr()**.

При вызове каждой функции производится контроль ошибок. Если какая-либо из функций завершена с ошибкой, то файл закрывается и функция **init_RS232()** возвращает значение **–1**.

5.4.2. Установка скорости порта RS232

Установка скорости порта RS232 производится с помощью функций библиотеки **termios**. Заметим, что существуют две отдельные функции – установки скорости вывода **cfsetospeed()** и скорости ввода **cfsetispeed()**. Однако в большинстве систем эти скорости должны быть равны, поскольку оборудование не поддерживает различных скоростей для ввода и вывода данных по порту RS232.

Доступны следующие стандартные константы скоростей: B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200, соответствующие скоростям от 50 до 115200 бит/с.

На листинге ниже приведена функция установки заданной скорости для открытого порта RS232.:

```
/*
     Установить скорость порта RS232.
*/
#include <termios.h>
#include <unistd.h>

int speed_RS232(int fd, speed_t speed){
     struct termios settings;

     if( tcgetattr(fd,&settings) ){
          // Ошибка
          return(-1);
     }

     cfsetospeed(&settings, speed);
     cfsetispeed(&settings, speed);

     if( tcsetattr(fd,TCSANOW,&settings) ){
          // Ошибка
          return(-1);
     }
```

```
        return(0);
}
```

Функция **speed_RS232()** принимает два входных параметра – дескриптор открытого файла **fd** и скорость порта – **speed**, значение которой – одна из констант скорости порта **B50** – **B115200**. В случае если скорость порта установлена, то функция **speed_RS232()** возвращает **0**. Если произошла ошибка, то функция **speed_RS232()** возвращает **–1**.

5.4.3. Организация приёма информации по порту RS232 с помощью вызова select()

Приведём пример организации приёма информации по порту RS232 с использованием системного вызова **select()**.

Использование данного системного вызова позволяет рационально использовать ресурсы процессора и организовать цикл обработки файловых событий.

Вызов **select()** позволяет программам отслеживать изменения нескольких файловых дескрипторов, ожидая, когда один или более файловых дескрипторов станут «готовы» для операции ввода-вывода определённого типа (например, ввода). Файловый дескриптор считается готовым, если к нему возможно применить соответствующую операцию ввода-вывода.

Предположим, что имеется открытый файл порта RS232 и программа должна принимать информацию по этому порту. Если информация не пришла в течение определённого времени, то необходимо обработать это событие как тайм-аут.

Вызов **select()** позволяет организовать функцию приёма данных с учётом тайм-аутов, как приведено на листинге ниже:

```
/*
     Пример 3. Организация приёма информации по порту
RS232 с помощью вызова select()
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select_RS232(int fd, int t){
           fd_set rfds;        // Список отслеживае-
мых дескрипторов
           struct timeval tv;  // Таймаут
           int retval;         //

     // Задаём список дескрипторов для чтения
     // В данном случае – один дескриптор fd
```

```
            FD_ZERO(&rfds);
            FD_SET(fd, &rfds);
        // Задаём тайм-аут t секунд
            tv.tv_sec = t;
            tv.tv_usec = 0;
        // Ждём наступления события или таймаута
          retval = select(fd+1, &rfds, NULL, NULL, &tv);

            if (retval == -1) {
          // Ошибка !
          return(-1);
        }
             else if (retval) {
          if( FD_ISSET(fd, &rfds) ){
              // Есть данные!
              // Можно их считать функцией read()
              // и обработать с помощью функции
              // RecieveData()
              RecieveData(fd);
          }
        }
              else {
        // Данные не появились, таймаут истёк.
        // Обработаем данную ситуацию с помощью
        // функции TimeOutData()

        TimeOutData(fd);
      }
      return(0);
    }
```

Рассмотрим, как используется системный вызов **select()** в функции **select_RS232()**.

Функция **select_RS232()** принимает два входных параметра – дескриптор открытого файла **fd** и длительность тайм-аута в секундах **t**.

Для использования вызова **select()** необходимо создать список файловых дескрипторов для каждого типа события. В нашем примере отслеживается только одно событие – появление данных в файле-устройстве для чтения.

Список файлов для вызова **select()** создаётся с помощью макросов **FD_ZERO()** – создание пустого списка файловых дескрипторов и **FD_SET()** – добавить указанный дескриптор в список файлов.

В нашем случае список дескрипторов файлов состоит только из одного дескриптора – **fd**.

Кроме списка файлов, вызов **select()** принимает также значение тайм-аута – т.е. времени, через которое вызов **select()** завершится, если не на-

ступит ни одного из отслеживаемых событий. Тайм-аут задаётся в переменной **tv**. Для простоты зададим таймаут в секундах.

После того как списки файлов заданы и установлен тайм-аут, вызывается **select()**. Вызов **select()** отслеживает три независимых набора файловых дескрипторов. Для первого набора дескрипторов отслеживается появление символов, доступных для чтения. Для второго набора дескрипторов отслеживается возможность записи. И для третьего набора дескрипторов отслеживается обнаружение исключительных ситуаций. При возврате из вызова наборы изменяются, показывая, какие файловые дескрипторы фактически изменили состояние. Значение любого из трёх наборов файловых дескрипторов может быть равно **NULL**, если слежение за определённым классом событий над файловыми дескрипторами не требуется.

В нашем примере отслеживается только появление символов, доступных для чтения.

Первым параметром вызова **select()** является максимальное значение дескриптора из всех трёх наборов, увеличенное на единицу. В нашем примере – **(fd+1)**. Последним параметром функции является значение тайм-аута **tv**.

После завершения системного вызова **select()** по коду возврата и состоянию списка файловых дескрипторов делается вывод о возникновении одной из трёх ситуаций:

– ошибки системного вызова **select()**. В этом случае функция **select_RS232()** завершается и возвращает **–1**;

– появления данных для чтения в файле. В этом случае вызывается функция чтения и обработки данных **RecieveData()**;

– истечении времени тайм-аута. В этом случае вызывается функция обработки тайм-аута **TimeOutData()**.

Функции **RecieveData()** и **TimeOutData()** зависят от конкретного алгоритма обработки данных и здесь не приводятся.

5.5. Обмен данными по IP-интерфейсам

Обмен данными по IP-интерфейсам происходит двумя способами – с установлением соединения и без установления соединения. Рассмотрим оба способа на примерах.

5.5.1. Обмен данными по протоколу UDP

Данный протокол [19] не требует установления соединения, не гарантирует, что пакеты дойдут до получателя и не гарантирует порядок следования пакетов. Отметим, что в большинстве реализаций протокола UDP для различных платформ пакеты меньшей длины доходят до получателя раньше пакетов большей длины, если они посланы одновременно.

Этот протокол используется в основном для обмена информацией, которая требует постоянного потока данных, но некритичная к потере определённой части передаваемой информации. Например, UDP-протокол используется для трансляции видео и звукового вещания через сеть Internet.

Для отправки и приёма пакетов по протоколу UDP необходимо создать сокет типа **SOCK_DGRAM**, после чего возможно отправлять UDP-пакеты указанным адресатам на указанный порт и принимать UDP-пакеты с указанного порта.

5.5.2. Создание UDP-сокета

Создадим UDP-сокет, по которому пакеты данных можно пересылать произвольному адресу и порту либо принимать пакеты данных с определённого порта. При создании UDP-сокета необходимо знать только один параметр – порт, по которому будут приниматься пакеты данных. На листинге ниже приведён пример создания такого UDP-сокета:

```
/*
     Создание UDP-сокета.
         port – порт приёма данных
*/
#include <string.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int udpCreate(int port){
     // Адрес сокета
     struct sockaddr_in addr;
     // Файловый дескриптор сокета
     int sock;
     // Создаём сокет
     sock = socket(AF_INET, SOCK_DGRAM, 0);
     if(sock < 0){
         // Ошибка
         return -1;
     }
     // Привязываем сокет к адресу и порту
     memset(&addr, 0, sizeof(addr));
     addr.sin_family = AF_INET;
     addr.sin_addr.s_addr = INADDR_ANY;
     addr.sin_port = htons(port);

     if (bind(sock, (struct sockaddr*)&addr,
         sizeof(struct sockaddr_in)) < 0){
         close(sock);
         // Ошибка
         return -1;
     }
```

```
    return(sock);
}
```

Сначала создаётся непривязанный (неинициализированный) сокет системным вызовом **socket()**. Затем он привязывается к адресу addr с помощью системного вызова **bind()**.

Параметр **AF_INET** указывает, что используется интернет-сокет, т.е. сокет будет использовать адресное пространство **IP**.

Параметр адреса **INADDR_ANY** указывает, что данные по указанному сокету буду приниматься независимо от адреса источника.

В случае если при создании сокета возникла ошибка, функция **udpCreate()** возвращает значение **–1**. Если сокет создан и инициализирован, то функция **udpCreate()** возвращает файловый дескриптор сокета, который можно использовать для чтения и записи данных, а также использовать для проверки файловых событий при помощи вызова **select()**.

5.5.3. Отправка UDP-пакета

Отправка UDP-пакета по указанному сокету в общем случае требует указания четырёх параметров: IP-адреса приёмника, порта приёмника, указателя на начало буфера отправляемых данных и длины отправляемых данных.

В нашем примере функция **udpSendTo()** принимает IP-адрес приёмника данных в виде текстовой строки. Это не оптимально с точки зрения быстродействия и может быть улучшено в процессе курсового проектирования.

```
/*
     Отправка пакета данных по UDP-сокету.
         sock – файловый дескриптор сокета
         ip – IP-адрес получателя данных
         port – порт получателя данных
         void – указатель на начало
             буфера отправляемых данных
         len – длина отправляемых данных
*/
#include <string.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
int udpSendTo(int sock,
         const char* ip, int port,
         const void* buf, int len){

     struct sockaddr_in addr;

     // Привязываем сокет к адресу и порту
     memset(&addr, 0, sizeof(addr));
     addr.sin_family = AF_INET;
```

```
    addr.sin_addr.s_addr = inet_addr(ip);
    addr.sin_port = htons(port);

    // Отправляем данные
    return(sendto(sock, buf, len, 0,
        (struct sockaddr *) (&addr), sizeof(addr)));
}
```

В случае успешной отправки данных функция **udpSendTo()** возвращает количество отправленных символов. Если произойдёт ошибка отправки пакета данных, то функция **udpSendTo()** вернёт значение **–1**.

Заметим, что успешная отправка данных по UDP-сокету **не гарантирует**, что получатель данных действительно их получит, а лишь означает, что данные успешно переданы в канал связи.

5.5.4. Приём UDP-пакета

UDP-пакеты могут посылаться различными устройствами. Поэтому при приёме UDP-пакета, помимо самих данных, необходимо выделять информацию об источнике (отправителе) данного пакета, а именно – IP-адрес и порт отправителя пакета данных.

В нашем примере это делает функция **udpRecvFrom()**.

```
/*
    Приём пакета данных по UDP-сокету.
        sock – файловый дескриптор сокета
        ip – IP-адрес источника данных
        port – порт источника данных
        void – указатель на начало
            буфера принимаемых данных
        len –  длина буфера принимаемых данных
*/
#include <string.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int udpRecvFrom(int sock,
    char* ip, int* port,
    void* buf, int len){
    struct sockaddr_in addr;
    socklen_t adr_len;
    int l = recvfrom(sock, buf, len, 0,
        (struct sockaddr*)&addr, &adr_len);
    if(ip){
        sprintf(ip,"%d.%d.%d.%d",
            ((addr.sin_addr.s_addr >> 24) & 0xff),
            ((addr.sin_addr.s_addr >> 16) & 0xff),
            ((addr.sin_addr.s_addr >> 8) & 0xff),
            (addr.sin_addr.s_addr & 0xff));
    }
    if(port){
```

```
            *port = addr.sin_port;
        }
        return(l);
}
```

Функция **udpRecvFrom()** возвращает **–1** в случае ошибки или длину реально принятых данных в случае, когда они имеются. Значения адрес и порта источника данных сохраняются в переменных **ip** и **port**. Для удобства вывода на экран IP-адрес сохраняется в виде текстовой строки, что не оптимально с точки зрения быстродействия и может быть улучшено в процессе курсового проектирования.

5.5.5. Обмен данными по протоколу TCP

В отличие от UDP-протокола, протокол TCP гарантирует доставку пакетов от источника данных до приёмника, а также гарантирует, что пакеты будут доставлены в том же порядке, что и отправлены. С точки зрения надёжности протокол TCP более надёжен, чем UDP.

Протокол TCP подразумевает архитектуру клиент–сервер с установлением соединения [19].

Это означает, что перед обменом данными источник и приёмник данных обмениваются служебными пакетами – устанавливают соединение, которое поддерживается до тех пор, пока источник или приёмник данных не разорвут его.

Обмен служебными пакетами для поддержания соединения и подтверждений доставки данных влечёт за собой снижение быстродействия протокола TCP по сравнению с UDP.

Кроме того, клиент и сервер не являются равноправными с точки зрения установления соединения.

Клиент – это то устройство, по инициативе которого устанавливается соединение с ***сервером***.

Сервер – устройство, которое ждёт запроса на соединение от ***клиента***.

Таким образом, чтобы создать TCP-соединение, необходимо знать, какое из МПУ будет сервером, а какое – клиентом. После создания соединения клиент и сервер обмениваются по нему пакетами данных абсолютно равноправно.

В отличие от UDP-протокола, каждое TCP-соединение представляет собой соединение «точка-точка», т.е., будучи один раз созданным, данное соединение не позволяет передавать данные произвольным приёмникам данных, а обеспечивает связь только между двумя МПУ – клиентом и сервером. Поскольку адреса и порты клиента и сервера жёстко задаются в процессе создания TCP-соединения, то посылка и приём данных посредством TCP-сокета могут осуществляться стандартными системными вызовами **write()** и **read()**.

5.5.6. Создание TCP-клиента

TCP-клиент – это инициатор соединения. Для установления соединения TCP-клиенту необходимо задать IP-адрес и порт сервера, с которым необходимо соединиться. Назовём функцию, создающую TCP-соединение со стороны клиента, **tcpCreateClient()**.

```
/*
     Создание TCP-сокета (клиентская часть)
         ip - адрес сервера
         port - порт сервера
*/
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int tcpCreateClient(const char* ip, int port){
     int sock;
     struct sockaddr_in addr;

     // Создаём сокет
     sock = socket(AF_INET, SOCK_STREAM, 0);
     if(sock < 0){
         return(-1);
     }

     // Привязываем созданный сокет к адресу сервера
     memset(&addr, 0, sizeof(addr));
     addr.sin_family = AF_INET;
     addr.sin_addr.s_addr = inet_addr(ip);
     addr.sin_port = htons(port);

     // Соединяемся с сервером
     if(connect(sock,(struct sockaddr *)&addr,
         sizeof(addr)) < 0){
         return(-1);
     }

     return(sock);
}
```

Функция **tcpCreateClient()** принимает два параметра – ip-адрес в виде строки символов и номер порта. При успешной попытке соединения функция возвращает файловый дескриптор созданного соединения. Если соединение не удалось, то функция вернёт **–1**.

5.5.7. Создание TCP-сервера

Серверная часть TCP-соединения гораздо сложнее клиентской. В общем случае к серверу может быть подключено несколько клиентов, по-

этому процесс создания соединения со стороны TCP-сервера включает два этапа.

Первый этап – создание «слушающего» сокета, привязанного к определённому порту. Задача данного сокета – принимать запросы на соединение от клиентов по определённому порту. При обнаружении запроса на соединение осуществляется переход ко второму этапу.

Второй этап – обработка запроса на соединение от клиента и создание сокета для обмена данными, связанными с определённым клиентом.

Приведём пример создания серверного сокета. Для удобства вынесем каждый из указанных этапов в отдельную функцию.

```
/*
     Создание «слушающего» сокета сервера
         port – порт сервера
*/
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int tcpCreateServerListener(int port){
     int listener;
     struct sockaddr_in addr;
     // Создаём "слушающий" сокет
     listener = socket(AF_INET, SOCK_STREAM, 0);
     if(listener < 0){
         return(-1);
     }
     // Привязываем его к порту
     // Разрешаем принимать соединения
     // с любого IP-адреса
     addr.sin_family = AF_INET;
     addr.sin_port = htons(port);
     addr.sin_addr.s_addr = htonl(INADDR_ANY);
     if(bind(listener, (struct sockaddr *)&addr,
         sizeof(addr)) < 0){
         return(-1);
     }
     return(listener);
}
```

Функция **tcpCreateServerListener()** принимает один параметр – порт, на который сервер должен принимать соединения и создаёт «слушающий» сокет, принимающий запросы на соединение от клиентов. В случае ошибки создания «слушающего» сокета функция **tcpCreateServerListener()** возвращает **–1**. Если сокет успешно создан, то функция **tcpCreateServerListener()** возвращает его файловый дескриптор.

Следующая функция – **tcpCreateServerConnect()** – создаёт сокет для обмена данными с клиентом.

```
/*
     Создание сокета сервера для обмена
     данными с клиентом
          listener – файловый дескриптор
               «слушающего» сокета
*/
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAX_CLIENTS   1

int tcpCreateServerConnect(int listener){
     int sock;
     // Ждём запроса на соединение
     listen(listener, MAX_CLIENTS);

     // Создаём сокет обмена данными с клиентом
     sock = accept(listener, NULL, NULL);
          if(sock < 0){
          return(-1);
     }
     return(sock);
}
```

Функция **tcpCreateServerConnect()** принимает один параметр – файловый дескриптор «слушающего» сокета. Системный вызов **listen()** ожидает запроса на соединение, пришедшего на указанный «слушающий» сокет. Параметр **MAX_CLIENTS** указывает максимально возможное число соединений.

Если соединение с клиентом успешно установлено, то функция **tcpCreateServerConnect()** возвращает файловый дескриптор сокета для обмена данными. Если произошла ошибка, то функция **tcpCreateServerConnect()** возвращает **–1**.

5.5.8. Отправка TCP-пакета

Данные по TCP-сокету можно отправлять как системным вызовом **write()**, так и системным вызовом **send()**.

Пример использования вызова **write()** для отправки данных по TCP-соединению:

```
write(sock, buf, len);
```

где **sock** – файловый дескриптор сокета для обмена данными; **buf** – указатель на начало буфера данных; **len** – длина отправляемых данных.

Пример использования вызова **send()** для отправки данных по TCP-соединению:

```
send(sock, buf, len, flags);
```

где **sock** – файловый дескриптор сокета для обмена данными; **buf** – указатель на начало буфера данных; **len** – длина отправляемых данных; **flags** – дополнительные флаги.

Если **flags=0**, то вызовы **send()** и **write()** идентичны по своим функциям.

В случае возникновения ошибки вызовы **send()** и **write()** возвращают значение **–1**. Если соединение завершено с противоположной стороны, то вызовы **send()** и **write()** вернут значение **0**. Если данные отправлены, то вызовы **send()** и **write()** вернут количество отправленных байт.

Для детального ознакомления с системными вызовами **write()** и **send()** рекомендуем обратиться к [20].

5.5.9. Приём TCP-пакета

Данные по TCP-сокету можно отправлять как системным вызовом **read()**, так и системным вызовом **recv()**.

Пример использования вызова **read()** для приёма данных по TCP-соединению:

```
read(sock, buf, len);
```

где **sock** – файловый дескриптор сокета для обмена данными; **buf** – указатель на начало буфера данных; **len** – длина буфера данных.

Пример использования вызова **recv()** для отправки данных по TCP-соединению:

```
recv(sock, buf, len, flags);
```

где **sock** – файловый дескриптор сокета для обмена данными; **buf** – указатель на начало буфера данных; **len** – длина буфера данных; **flags** – дополнительные флаги.

Если **flags** = 0, то вызовы **read()** и **recv()** идентичны по своим функциям.

В случае возникновения ошибки, вызовы **read() и recv()** возвращают значение **–1**. Если соединение завершено с противоположной стороны, то вызовы **read() и recv()** вернут значение **0**. Если данные приняты, то вызовы **read() и recv()** вернут количество принятых байтов. Для детального ознакомления с системными вызовами **read() и recv()** рекомендуем обратиться к [20].

6. ПРИМЕР ОРГАНИЗАЦИИ ПО РМПС

Рассмотрим пример организации ПО для МПУ, входящих в состав РМПС, предназначенной для обмена текстовыми сообщениями.

В качестве транспортного протокола используем протокол, разработанный в разд. 5.1. Структура РМПС представлена на рис. 12.

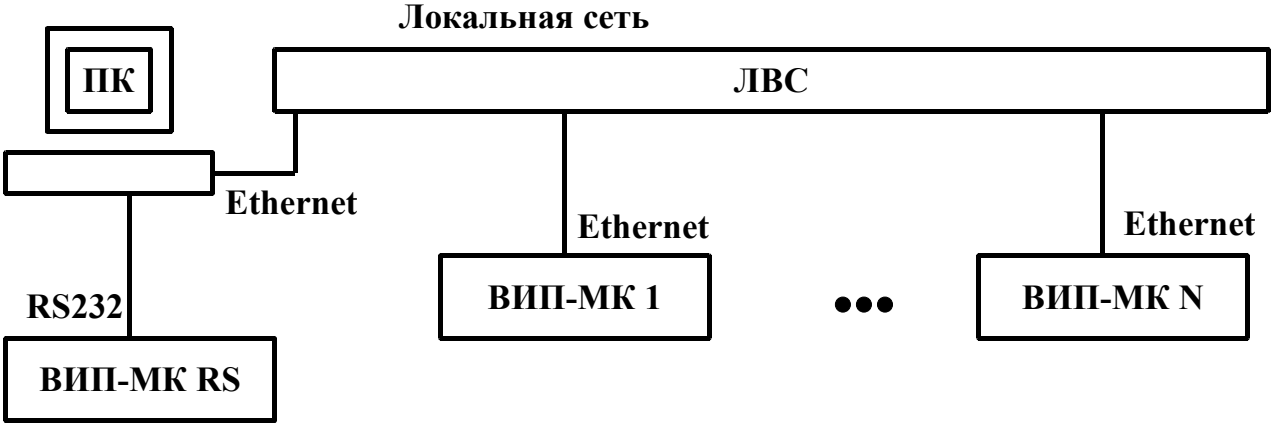

Рис. 12. Структура РМПС (пример)

В РМПС входят следующие МПУ: один ПК, один терминал ВИП-МК, подключённый по интерфейсу RS232 к ПК, и один или несколько ВИП-МК, соединённых по сети Ethernet между собой и с ПК.

Необходимо организовать обмен текстовыми сообщениями между произвольными МПУ, входящими в систему.

Для простоты предположим, что текстовые сообщения создаются операторами с клавиатуры.

Выберем протокол UDP для передачи сообщений по сети Ethernet, поскольку в протоколе, разработанном в разд. 5.1, существует система подтверждений о доставке пакетов данных.

Для написания программы выберем язык C++, а для её сборки – описанный ранее универсальный **Makefile**.

6.1. Общая структура ПО РМПС (пример)

Для реализации ПО с указанной функциональностью необходимы следующие программные модули:

- модуль управления портом RS232;
- модуль управления каналом Ethernet;
- модуль обработки протокола;
- модуль ввода терминала;
- модуль вывода на экран;
- модуль маршрутизации;
- модуль обработки событий;
- модуль начальной инициализации системы.

Рассмотрим функции каждого модуля отдельно и взаимодействие модулей в целом. Структура ПО для МПУ, входящего в состав разрабатываемой РМПС, представлена на рис. 13.

Базовым модулем всей системы является **модуль обработки событий**. Поскольку терминал, UDP-сокет и порт RS232 представлены в UNIX-системе как файлы, то события приёма данных с клавиатуры или из канала связи, а также событие-тайм-аут в данном случае генерируются системным вызовом **select()**.

Модуль обработки событий обрабатывает два типа событий: появление данных для чтения в одном из контролируемых файлов и истечение заданного тайм-аута. Событие-тайм-аут необходимо для периодического совершения каких-либо действий, например вывода часов на экран или контроля за подтверждением о доставке пакета.

Модуль обработки событий решает следующие задачи:

- приём события от файловых дескрипторов или события тайм-аута;
- определение типа события;
- передача события соответствующей процедуре обработки.

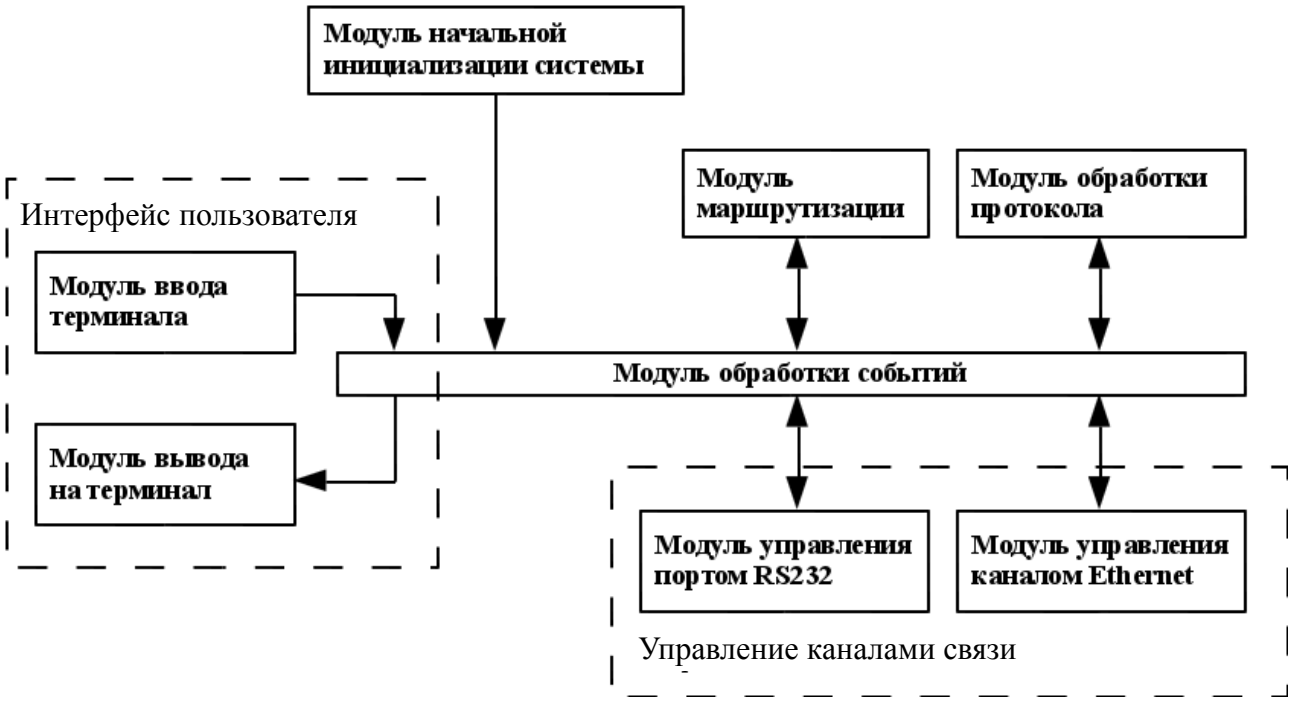

Рис. 13. Структура ПО МПУ

Данные, приходящие от модулей управления каналами связи (модуль управления портом RS232 и модуль управления каналом Ethernet), перенаправляются в модуль обработки протокола.

Модуль обработки протокола определяет, что пришёл пакет данных из канала связи и, если пакет прошёл контроль целостности, генерирует событие «приём пакета данных из канала связи».

Таблица маршрутизации предназначена для перенаправления пакетов данных из одного канала связи в другой. Например, (см. рис. 12) пакеты от ВИП-МК RS не могут передаваться непосредственно на ВИП-МК 1 – ВИП-МК N, поэтому ПК должен выполнять функции маршрутизатора между каналами связи RS232 и Ethernet.

Таблица имеет количество строк, равное количеству МПУ в системе. Каждая строка содержит:

- номер МПУ в системе;
- тип канала связи, по которому осуществляется доступ к указанному МПУ;
- адрес в системе адресации конкретного канала связи. Например, для RS232 нет адреса (поле пустое), для Ethernet – ip-адрес и порт.

Принятый пакет данных обрабатывается следующим образом:

- если адрес МПУ в системе совпадает с адресом назначения в принятом пакете данных, то содержимое пакета обрабатывается на данном МПУ (в зависимости от типа пакета, выводится на экран текстовое сообщение или подтверждение о доставке сообщения);
- если адрес МПУ в системе не совпадает с адресом назначения в принятом пакете данных, то производится поиск в таблице маршрутизации адреса назначения, принятого в пакете данных;
- если адрес назначения найден в таблице маршрутизации, то пакет пересылается по тому каналу связи, который указан в таблице, иначе пакет игнорируется.

Имеется два режима ввода данных – режим редактирования сообщения и режим выбора адреса назначения.

В режиме редактирования сообщения, данные, приходящие от терминала (обычно с клавиатуры), обрабатываются следующим образом. Определяется тип нажатой клавиши и производится одно из действий:

- символьные клавиши добавляются в буфер ввода и выводятся на экран;
- клавиша BackSpace удаляет последний введённый символ;
- клавиша Enter – перейти в режим выбора адреса назначения.

В режиме выбора адреса назначения обрабатываются следующие клавиши:

- ПРОБЕЛ – перейти к следующему адресу назначения из таблицы маршрутизации;
- Enter – отправить сообщение;
- ESC – возврат в режим ввода сообщения без его отправки.

Модуль начальной инициализации системы предназначен для вызова функций инициализации всех модулей, входящих в систему. После выполнения всех функций инициализации, а именно загрузки таблицы мар-

шрутизации, номера устройства в системе, создания файловых дескрипторов терминального ввода-вывода, UDP-сокета и порта RS232, модуль инициализации запускает бесконечный цикл обработки событий.

6.2. Модуль обработки событий

Описание класса-обработчика событий представлено на листинге ниже. Основой класса является функция **eventLoop()**, запускающая цикл обработки событий на основе системного вызова **select()**.

Перед тем как запускать цикл обработки событий, необходимо выполнить следующие действия:

- открыть все необходимые файлы (получить файловые дескрипторы);
- зарегистрировать функции-обработчики для каждого события.

Имеются два типа функций-обработчиков событий: **fdEvent** и **timeOutEvent**. Функция типа **fdEvent** – обработчик события прихода данных получает на вход параметр типа **int** – файловый дескриптор, с помощью которого можно прочитать пришедшие данные. Функция типа **timeOutEvent** не имеет параметров и вызывается, если в течение времени, определённого в поле **timeOut**, не пришло данных ни на один из зарегистрированных файловых дескрипторов.

```
/**
    @file eventhandler.h
    Цикл обработки событий
*/

#ifndef eventhandler_h_E
#define eventhandler_h_E

#include <time.h>
#include <memory>
#include <functional>
#include <vector>

//

/**
 * @brief Тип события
 *
 */
enum evType{
     evFile = 0, // событие - появление данных
     evTimeOut   // событие тайм-аут
};

// Тип - функция для обработки файлового дескриптора
typedef std::function<void(int)> fdEvent;

// Тип - функция для обработки таймаута
```

```
typedef std::function<void()> timeOutEvent;

// Тип-запись, хранящая обработчик события
typedef struct fdEventElem{
     evType       evtype;
     int fd;      // файловый дескриптор
     // Обработчики событий, в зависимости от их типа
     fdEvent  fdHandler; // Появление данных в файле
     timeOutEvent toHandler; // Тайм-аут
}fdEventElem;

// Список обработчиков событий
typedef std::vector<fdEventElem> fdEventList;

// Класс-обработчик сообщний
class eventHandler{
public:
     // Конструктор класса
     eventHandler();

     // Добавить обработчик файлового дескриптора
     void addFdHandler(int fd, fdEvent handler);

     // Добавить обработчик таймаута
     void addToHandler(timeOutEvent handler);

     // Запуск цикла обработки событий
     void eventLoop();

     // Завершить цикл обработки событий
     void exitLoop();

     // Получить ссылку на объект-синглетон
     static eventHandler& instance();
private:
     // список обработчиков событий
     fdEventList  eventList;

     // Таймаут для select()
     time_t       timeOut;

     // значение наибольшего дескриптора файлов
     int nfds;

     // Флаг завершения цикла обработки событий
     bool termLoop;
};

#endif /* eventhandler_h_E */
```

Обработчиков прихода данных из файлового дескриптора и тайм-аута может быть несколько. Пример регистрации обработчика прихода данных из файлового дескриптора и обработчика тайм-аута:

```
// Функция чтения и обработки данных
void fdHandler(int fd){
     // Тут можно прочитать данные,
     // например так
     char buf[0x100];
     read(fd,buf,sizeof(buf));
}

// Функция обработки тайм-аута
void timeoutHandler(){
     // Таймаут.
     //Нет данных в течении заданного времени
}

….....................
// Регистрация обработчика прихода данных
// fd – дескриптор файла, с которого производится чтение
eventHandler::instance().addFdHandler(fd,fdHandler);
….................
// Регистрация обработчика тайм-аута
eventHan-
dler::instance().addToHandler(timeoutHandler);
….................
// Зарегестрированы все обработчики
// Запуск цикла обработки событий
eventHandler::instance().eventLoop();
```

Отметим, что нельзя дважды регистрировать обработчик прихода данных из файлового дескриптора с одним и тем же дескриптором файла. Это приведёт к ошибкам при выполнении программы.

Исходный код методов класса-обработчика событий приведён на следующем листинге:

```
/**
       @file eventhandler.cxx
      Цикл обработки событий
*/

#include <unistd.h>
#include <sys/select.h>

#include <memory>
#include "eventhandler.h"

eventHandler::eventHandler(){
     // По умолчанию таймаут – 1 секунда
     timeOut = 1;
     nfds = -1;
     termLoop = false;
}
```

```
void eventHandler::addFdHandler(int fd, fdEvent han-
dler){
     fdEventElem ev;
     ev.evtype = evFile;
     ev.fdHandler = handler;
     ev.fd = fd;
     eventList.push_back(ev);
     if(fd>nfds){
         nfds = fd;
     }
}

void eventHandler::addToHandler(timeOutEvent han-
dler){
     fdEventElem ev;
     ev.evtype = evTimeOut;
     ev.toHandler = handler;
     eventList.push_back(ev);
}

void eventHandler::exitLoop(){
     termLoop = true;
}

void eventHandler::eventLoop(){

     while(!termLoop){
         fd_set rfds;
         FD_ZERO(&rfds);
         // Добавляем все дескрипторы в список
         for(auto &i:eventList){
             if(i.evtype == evFile){
             FD_SET(i.fd, &rfds);
         }
     }

     struct  timeval  tv;
     tv.tv_sec = timeOut;
     tv.tv_usec = 0;

     int retval = select(nfds+1, &rfds, NULL, NULL,
&tv);

     if(retval<0){
         // Ошибка
         termLoop = true;
     }
     else if(retval > 0){
```

```
            // Появились данные. обрабатываем их
            for(auto &i:eventList){
                if(i.evtype != evFile) {
                    continue;
                }

                if(FD_ISSET(i.fd, &rfds)){
                    if(i.fdHandler){
                        i.fdHandler(i.fd);
                    }
                }
            }
        }
        else{
            // таймаут
            for(auto &i:eventList){
                if(i.evtype == evTimeOut){
                    if(i.toHandler){
                        i.toHandler();
                    }
                }
            }
        }
    }
}

  static std::unique_ptr<eventHandler>
                    eventHandlerInstance;

  eventHandler& eventHandler::instance(){

      if(!eventHandlerInstance){
          eventHandlerInstance.reset( new eventHandler );
      }
      return(*eventHandlerInstance);
}
```

Исходный код модуля может быть дополнен необходимыми функциями в процессе выполнения курсового проекта.

6.3. Модуль обработки протокола

Модуль обработки протокола состоит из двух классов – **recProto** (обработка принятых данных) и **sendProto** (подготовка данных к отправке).

Класс **recProto** представляет собой конечный автомат, имеющий несколько состояний, описанных в типе **protoStay**.

Для каждого соединения необходимо иметь экземпляр класса **recProto**.

Класс **recProto** имеет три метода для связи с внешними объектами – это методы **recData()**, **reset()** и функция, задаваемая методом **onRecievePacket()**.

Метод **recData()** вызывается при приёме произвольного количества байтов из канала связи.

Метод **reset()** – сброс состояния протокола, например по таймауту или после обработки очередного принятого пакета.

Функция, задаваемая методом **onRecievePacket()** предназначена для обработки корректно принятых пакетов и является внешней по отношению к классу **recProto**.

Описание класса **recProto** приведено на листинге ниже:

```
/**
    @file protocol.h
  Модуль обработки протокола (приём пакетов)
*/
#ifndef protocol_h_E
#define protocol_h_E

#include <sys/types.h>
#include <functional>
#include <vector>

// Признак пакета-подтверждения
const u_int8_t pfAck  = 1;
// Признак запроса, требующего подтверждения
const u_int8_t pfQack = 2;

// Массив байт
typedef std::vector<u_int8_t> rawData;

// Описатель пакета (см. пп.0)
typedef struct protPacket{
     u_int32_t    da; // Адрес назначения
     u_int32_t    sa; // Адрес источника
     u_int8_t f;   // Флаги пакета
     u_int8_t id; // Идентификатор пакета
     u_int16_t    len;// Длина поля data
     rawData  data; // Данные
     u_int16_t    crc;   // Принятая из канала
                  //контрольная сумма
}protPacket;

// Тип – функция для обработки принятого пакета
typedef std::function<void(protPacket&)> recPacketE-
vent;
```

```
// Состояния приёмика
enum protoStay{
     psWDA = 0, // Приём поля DA
     psWSA,   // Приём поля SA
     psWF,    // Приём поля F
     psWID,   // Приём поля ID
     psWLEN,  // Приём поля LEN
     psWDATA,// Приём поля DATA
     psWCRC   // Приём поля CRC
};

// Протокол (приёмник)
class recProto{
public:
     // конструктор
     recProto();

     // сброс состояния обработчика протокола
     void reset();

     // установка обработчика принятого пакета
     void onRecievePacket(recPacketEvent handler);

     // Функция обработки данных,
     // принятых из канала связи
     // data – указатель на буфер с принятыми данными
     // len – размер данных в байтах
     void recData(const void* data, int len);

private:
     // обработчик принятого пакета
     recPacketEvent  recPacketHandler;

     // принимаемый пакет
     protPacket   recPacket;

     // Счётчик принятых байт
     int      recCounter;

     // Состояние приёмика
     protoStay    stay;

     // Вычисляемая контрольная сумма
     u_int16_t    crc;

     // Обработчик очередного принятого байта
     void recDataByte(const u_int8_t b);
};
```

Для использования класса-приёмника протокола необходимо зарегистрировать обработчик принятого пакета – функцию типа **recPacketEvent**, как показано в примере:

```
// Функция-обработчик принятого пакета
void packetHandler(protPacket& packet){
     // Принят пакет данных packet

}
….....................
// Создадим экземпляр класса recProto
recProto rec;

// Регистрация обработчика принятого пакета
rec.onRecievePacket(packetHandler)
….....................
```

Данные, пришедшие из канала связи, необходимо передавать в класс-приёмник с помощью метода **recData()**. Удобнее всего вызывать метод **recData()** из функции-обработчика прихода данных из файлового дескриптора, как было показано ранее.

Метод **reset()** служит для сброса состояния протокола, например по тайм-ауту.

Исходный код методов класса-приёмника **recProto** приведён на листинге ниже:

```
/**
    @file protocol.cxx
  Модуль обработки протокола (приём пакетов)
*/

#include "protocol.h"
#include <sys/types.h>

// Приёмник пакетов

recProto::recProto(){
     recPacketHandler = nullptr;
     reset();
}

void recProto::reset(){
     recCounter = 0;
     stay = psWDA;
     recPacket.da = 0;
     crc = 0x5A5A;
}

void recProto::onRecievePacket(recPacketEvent han-
dler){
     recPacketHandler = handler;
}

void recProto::recData(const void* data, int len){
```

```
    u_int8_t* b = (u_int8_t*)data;

    for(auto i=0; i<len; i++){
        recDataByte(*(b++));
    }
}

void recProto::recDataByte(const u_int8_t b){

    // Вычисление контрольной суммы происходит
    // для всего пакета, кроме
    // принимаемой контрольной суммы
    if(stay != psWCRC){
        if(crc & 0x8000){
            crc = (crc + ((u_int16_t)b) + 1) &
0x7FFF;
        }
        else{
            crc = crc + ((u_int16_t)b);
        }
    }

    switch (stay){
        case psWDA:{
            recPacket.da = (recPacket.da >> 8) |
                        (((u_int32_t)b) << 24);
            recCounter++;
            if(recCounter >= sizeof(u_int32_t)){
                stay = psWSA;
                recCounter = 0;
                recPacket.sa = 0;
            }
            break;
        }
        case psWSA:{
            recPacket.sa = (recPacket.sa >> 8) |
                        (((u_int32_t)b) << 24);
            recCounter++;
            if(recCounter >= sizeof(u_int32_t)){
                stay = psWF;
                recCounter = 0;
            }
            break;
        }
        case psWF:{
            recPacket.f = b;
            recCounter++;
            if(recCounter >= sizeof(u_int8_t)){
                stay = psWID;
                recCounter = 0;
```

```
            }
            break;
        }
        case psWID:{
            recPacket.id = b;
            recCounter++;
            if(recCounter >= sizeof(u_int8_t)){
                stay = psWLEN;
                recPacket.len = 0;
                recCounter = 0;
            }
            break;
        }
        case psWLEN:{
            recPacket.len = (recPacket.len >> 8) |
                        (((u_int16_t)b) << 8);
            recCounter++;
            if(recCounter >= sizeof(u_int16_t)){
                recCounter = 0;

                if(recPacket.len == 0){
                    stay = psWCRC;
                    recPacket.crc = 0;
                }
                else{
                    stay = psWDATA;
                    recPacket.data.clear();
                }
            }
            break;
        }
        case psWDATA:{
            recPacket.data.push_back(b);
            recCounter++;
            if(recCounter >= recPacket.len){
                stay = psWCRC;
                recCounter = 0;
                recPacket.crc = 0;
            }
            break;
        }
        case psWCRC:{
            recPacket.crc = (recPacket.crc >> 8) |
                        (((u_int16_t)b) << 8);
            recCounter++;
            if(recCounter >= sizeof(u_int16_t)){
                // Проверка контрольной
                // суммы пакета
```

```
                    if( recPacket.crc == crc){
                        // Обработка принятого пакета
                        if(recPacketHandler){
                        recPacketHandler(recPacket);
                        }
                    }
                    // Сброс состояния пртокола
                    reset();
                }
                break;
            }
        }
}
```

Основой приёмника является метод **recDataByte()**, принимающий очередной байт и обрабатывающий его, в зависимости от состояния приёмника, хранящегося в поле **stay**.

По мере приёма данных вычисляется значение контрольной суммы. После приёма контрольной суммы из канала связи производится сравнение принятой и вычисленной контрольных сумм. Если они равны, то пакет считается корректно принятым и вызывается обработчик принятого пакета.

Класс **sendProto** осуществляет подготовку данных к отправке по каналу связи. Подготовка пакета заключается в преобразовании структуры типа **protPacket** в последовательность байтов в формате пакета данных протокола, описанного в разд. 5.1. Тип данных **rawData**, возвращаемый методами **setQuery()** и **setAck()**, представляет собой готовую последовательность байтов, предназначенную для отправки в канал связи. Описание класса **sendProto** приведено на листинге ниже:

```
// Подготовка пакета для передачи
class sendProto{
public:
     // Конструктор
     // sa – номер отправителя в системе
     sendProto (u_int32_t sa);

     // Сериализация пакета
     rawData rawPacket(protPacket& packet);

     // Подготовить пакет для передачи по адресу da
     rawData setQuery(u_int32_t da, rawData data);

     // Подготовить подтверждение
     // для передачи по адресу da
     // с идентификатором id
     rawData setAck(u_int32_t da, u_int8_t id);
  private:
     u_int32_t m_sa; // номер отправителя в системе
```

```
        u_int8_t m_id;   // идентификатор пакета
    };
```

При создании класса **sendProto** номер отправителя в системе запоминается в поле **m_sa** и в дальнейшем не указывается.

Метод **rawPacket()** осуществляет преобразование пакета, описанного в типе данных **protPacket**, в последовательность байтов для передачи по каналу связи. Методы **setQuery()** и **setAck()** используются соответственно для подготовки пакета с данными и пакета-подтверждения.

Исходный код методов класса-приёмника **recProto** приведён на листинге ниже:

```
// Подготовка пакетов к отправке
#include "protocol.h"
#include <sys/types.h>
sendProto::sendProto(u_int32_t sa){
     m_sa = sa;
     m_id = 0;
}

rawData sendProto::rawPacket(protPacket& packet){
     rawData pkt;
     pkt.clear();

     //da
     pkt.push_back(packet.da & 0xFF);
     pkt.push_back((packet.da>>8) & 0xFF);
     pkt.push_back((packet.da>>16) & 0xFF);
     pkt.push_back((packet.da>>24) & 0xFF);

     //sa
     pkt.push_back(packet.sa & 0xFF);
     pkt.push_back((packet.sa>>8) & 0xFF);
     pkt.push_back((packet.sa>>16) & 0xFF);
     pkt.push_back((packet.sa>>24) & 0xFF);

     // f
     pkt.push_back(packet.f);

     // id
     pkt.push_back(packet.id);

     // len
     pkt.push_back(packet.data.size() & 0xFF);
     pkt.push_back((packet.data.size()>>8) & 0xFF);
     // data
     if(packet.data.size()){
         pkt.insert(pkt.end(),
             packet.data.begin(),packet.data.end());
     }
```

```
    // crc
    u_int16_t crc = 0x5A5A;
    for(u_int8_t b:pkt){
        if(crc & 0x8000){
            crc = (crc + ((u_int16_t)b) + 1) & 0x7FFF;
        }
        else{
            crc = crc + ((u_int16_t)b);
        }
    }
    pkt.push_back(crc & 0xFF);
    pkt.push_back((crc>>8) & 0xFF);

    return(pkt);
}

rawData sendProto::setQuery(u_int32_t da, rawData data){
    protPacket packet;
    packet.da   = da;
    packet.sa   = m_sa;
    packet.f    = pfQack;
    packet.id   = m_id++;
    packet.data = data;
    return( rawPacket(packet) );
}

rawData sendProto::setAck(u_int32_t da, u_int8_t id){
    protPacket packet;
    packet.da   = da;
    packet.sa   = m_sa;
    packet.f    = pfAck;
    packet.id   = id;
    packet.data.clear();
    return( rawPacket(packet) );
}
```

Пример подготовки данных для отправки по каналу связи приведён на листинге ниже:

```
….....................
// Создадим экземпляр класса sendProto
//  1912 – адрес устройства-отправителя
sendProto p(1912);

// Подготовим данные для отправки (десять символов 'A')
rawData mySendData;
mySendData.assign(10,'A');

// Подготовка пакета-запроса, требующего подтверждения
// о доствке. Адрес в системе устройства-назначения – 5.
// Адрес в системе устройства-отправителя – 1912.
```

```
// chanQuery – подготовленный пакет для отправки по
// каналу связи.
rawData chanQuery = setQuery(5, mySendData);
….....................

// Подготовка пакета-подтверждения о доствке. Адрес в
системе устройства-назначения – 5.
// Адрес в системе устройства-отправителя – 1912.
// id – идентификатор пакета,
// на который создаётся подтверждение
rawData chanAck = setAck(5,  id);
….....................
```

Так как адрес отправителя в системе не меняется, то экземпляр класса **sendProto** создаётся один раз при инициализации программы.

6.4. Модуль управления каналами связи

Модуль канала связи создаёт единый программный интерфейс между функциями ввода-вывода конкретного канала связи и остальным ПО.

Все классы-каналы связи представляют собой потомка класса **baseChannel**. Класс имеет три метода, предназначенные для организации канала связи: метод **sendData()**, метод **recvData()** и метод **getFd()**.

Помимо класса **baseChannel**, имеется важная структура-описатель адреса в системе канала связи. Как отмечалось ранее, каналы связи имеют в общем случае систему адресации, зависящую от типа конкретного канала. Поэтому была определена структура **chanAdr**, которая хранит адрес устройства в системе адресации канала связи. Например, канал RS232 не имеет адресации, каналы Интернета задают адреса в виде IP-адреса и порта и т.п. В ходе выполнения курсового проекта система адресации может быть расширена, если это необходимо.

Описание класса **baseChannel** приведено на листинге ниже:

```
/**
    @file channel.h
     Класс-предок классов-каналов связи
*/
#ifndef channel_h_E
#define channel_h_E

#include <string>
#include <protocol.h>

// Тип канала связи. При добавлении новых типов
// каналов связи, необходимо добавлять их имена в
// данный тип-перечисление
enum chanTyeps{
     // Канал не задан
```

```
        chtNone = -1,
        // Порт RS232
        chtRS232 = 0,
        // Internet-канал, протокол UDP
        chtUDP
};

// Адрес в системе канала связи.
// Тип адреса определяется полем type.
typedef struct chanAdr{

        // Тип канала связи
        chanTyeps    type;

        // IP-адрес для Internet-канала
        std::string  ip;

        // порт для Internet-канала
        int      port;

}chanAdr;

// Класс-предок классов-каналов связи
class baseChannel{
public:
        // Конструктор
        baseChannel();

        // Деструктор
        virtual ~baseChannel();

        // Получить файловый дескриптор канала связи
        int getFd(){
            return(fd);
        }

        // Отправить данные в канал
        virtual int  sendData(const void* data, int len,
                              chanAdr* adr=NULL);

        // Принять данные из канала
        virtual int  recvData(void* data, int len,
                              chanAdr* adr=NULL);

protected:
        int fd;
};

#endif /* channel_h_E */
```

Класс-предок классов-каналов связи **baseChannel** сам по себе не отправляет и не принимает данные, а лишь обеспечивает интерфейс для классов-потомков. Единственное поле класса **baseChannel** – дескриптор

файла, через который идёт обмен данными с каналом связи. Получить этот дескриптор можно с помощью метода **getFd()**. Конструктор **baseChannel** класса обеспечивает установку поля **fd** в значение **–1**, что означает «дескриптор не инициализирован». Деструктор класса **baseChannel** проверяет и закрывает дескриптор файла **fd**, если он инициализирован. Таким образом, в потомках класса **baseChannel** дескриптор файла **fd** закрывается автоматически при уничтожении класса. Описание методов класса **baseChannel** приведено на листинге ниже:

```
/**
    @file channel.cxx
  Класс-предок классов-каналов связи
*/
#include <unistd.h>
#include "channel.h"

baseChannel::baseChannel(){
     fd = -1;
}

baseChannel::~baseChannel(){
     if(fd>=0){
         close(fd);
     }
}

int baseChannel::sendData(const void* data, int len,
                          chanAdr* adr){
     return(-1);
}

int baseChannel::recvData(void* data, int len,
                          chanAdr* adr){
     return(-1);
}
```

При корректном написании программы никогда не создаётся экземпляров класса **baseChannel**. Всегда используются только потомки данного класса, работающие с конкретным каналом связи.

Методы **sendData()** и **recvData()** класса **baseChannel** возвращают значение –1, что означает ошибку работы с каналом связи. Это нужно для того, чтобы ситуация, в которой ошибочно будет создан экземпляр класса **baseChannel**, а не одного из его потомков, не привела к краху системы.

Приведём примеры двух классов для работы с конкретными каналами связи, рассмотренными в разд. 5.4 и 5.5: канал связи по порту RS232 и канал связи по сети Интернет, протокол UDP.

6.4.1. Класс-канал связи по порту RS232

Канал связи по порту **RS232** требует в общем случае задания многих параметров – скорости порта, количества бит в символе, количества стопбит, наличие признака чётности и т.п.

Для упрощения задачи предположим, что порт RS232 всегда работает в режиме 8N1, т.е. символ имеет размер 8 бит (1 байт), один стоповый бит и не имеет признака чётности. Поскольку мы передаём произвольные бинарные данные, то используется описанный ранее в пп. 5.4.1 режим RAW.

Класс-канал связи по порту **RS232** называется **channelRS232** и является потомком класса **baseChannel**. Описание класса приведено на листинге ниже:

```
/**
    @file chan_rs232.h
     Класс-канал связи по порту RS232
*/
#ifndef chan_rs232_h_E
#define chan_rs232_h_E

#include "channel.h"

class channelRS232: public baseChannel{
public:
     channelRS232(const char* device, const char* speed);

     // Отправить данные в канал
     virtual int  sendData(const void* data, int len,
                      chanAdr*     adr=NULL);

     // Принять данные из канала
     virtual int  recvData(void* data, int len,
                      chanAdr* adr=NULL);

     // Получить путь к устройству
     // в виде текстовой строки
     const char*  getDevice();

     // Получить скорость  в виде текстовой строки
     const char*  getSpeed();
private:
     // Путь к устройству (в виде текстовой строки)
     std::string  m_device;

     // Скорость устройства (в виде текстовой строки)
     std::string  m_speed;
};

#endif /* chan_rs232_h_E */
```

В классе **channelRS232** переопределены методы **sendData()** и **recvData()** класса-предка **baseChannel**.

Конструктор класса **channelRS232** имеет два параметра: **device** – путь к устройству-порту **RS232** и **speed** – скорость передачи данных по порту **RS232**, бит/с. Оба параметра – текстовые строки. Конструктор принимает эти параметры, запоминает их в полях **m_device** и **m_speed**, затем открывает указанное устройство-порт **RS232** и производит настройку указанного порта **RS232** в режим **RAW** на указанной скорости.

Если произошла ошибка, например неверно указано устройство или неверно указана скорость, то дескриптор файла, возвращаемый функцией **getFd()**, будет иметь значение **–1**. Если устройство-порт открыто и настроено, то функция **getFd()** вернёт дескриптор файла, по которому производится работа с портом.

Методы **sendData()** и **recvData()** имеют защиту от их использования при неинициализированном устройстве. Метод **sendData()** передаёт данные по порту **RS232**. В случае ошибки передачи возвращает **–1**. Если данные успешно переданы в порт, то возвращается количество переданных байтов.

Метод **recvData()** принимает данные из порта **RS232**. В случае ошибки приёма возвращает **–1**. Если данные успешно приняты из порта, то возвращается количество принятых байтов.

Отметим, что метод **recvData()** блокирует выполнение программы, пока не появятся данные в порту **RS232**. Поэтому рекомендуется вызывать его в обработчике появления данных в файле, описанного в модуле обработки событий в пп. 6.2. В этом случае метод считает данные, не блокируя выполнения программы.

Описание методов класса **channelRS232** приведено на листинге ниже:

```
/**
    @file chan_rs232.cxx
     Класс-канал связи по порту RS232
*/
#include "chan_rs232.h"
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>

// Структура-таблица скоростей терминала (RS232)
static struct {
     // значение строки, преведённое\
     // к целому с помощью atoi
     unsigned value;

     //скорость в формате библиотеки termios
     speed_t speed;
```

```
} speed_table [ ] = {
  {0, B0},
  {50, B50},
  {75, B75},
  {110, B110},
  {134, B134},
  {150, B150},
  {200, B200},
  {300, B300},
  {600, B600},
  {1200, B1200},
  {1800, B1800},
  {2400, B2400},
  {4800, B4800},
  {9600, B9600},
  {19200, B19200},
  {38400, B38400},
  {57600, B57600},
  {115200, B115200},
  {230400, B230400}
};

channelRS232::channelRS232(const char* device,
              const char* speed):baseChannel() {
     // Установка скорости
     unsigned us = atoi ( speed );
     speed_t spd = -1;
     for (auto &i:speed_table) {
         if( i.value == us ){
             spd = i.speed;
             break;
         }
     }

     if(spd==-1){
         // задана неверная скорость
         return;
     }

     // Открывем файл
     fd=open( device, O_RDWR);
     if(fd<0){
         // Ошибка при открытии файла
         return;
     }

     struct termios t;

     if ( !tcgetattr ( fd, &t ) ) {
         cfmakeraw ( &t );
```

```
            t.c_cc[VMIN]=1;
            t.c_cc[VTIME]=0;

            cfsetispeed ( &t, spd );
            cfsetospeed ( &t, spd );
        } else {
            // Ошибка.
            // Не удалось получить атрибуты устройства
            close(fd);
            fd = -1;
            return;
        }

        if (tcsetattr ( fd,TCSANOW,&t ) < 0){
            /// Ошибка.
            // Не удалось установить атрибуты устройства
            close(fd);
            fd = -1;
            return;
        }

        m_device = device;
        m_speed = speed;
    }

    int channelRS232::sendData(const void* data,
                           int len, chanAdr* adr){
        if(fd<0){
            return(-1);
        }
        return(write(fd,data,len));
    }

    int channelRS232::recvData(void* data, int len,
                       chanAdr* adr){
        if(fd<0){
            return(-1);
        }
        return(read(fd,data,len));
    }

    const char* channelRS232::getDevice(){
        return(m_device.c_str());
    }

    const char* channelRS232::getSpeed(){
        return(m_speed.c_str());
    }
```

Настройка порта **RS232** производится с помощью библиотеки терминального ввода-вывода **termios**. Таблица **speed_table** служит для преобра-

зования скорости, заданной в виде числа в константу, определяющую скорость из библиотеки **termios**.

Пример использования класса **channelRS232** приведён на листинге ниже:

```
….....................
// Функция приёма данных по порту RS232
void fdRS232Handler(int){
     // Данные в порту RS232 есть, читаем их
     char buf [0x100];
     int len =  chRS232.recvData(buf,sizeof(buf));
     // len – количество прочитанных байт или
     // -1 в случае ошибки
}
….....................
// Создадим экземпляр класса  channelRS232
// Канал связи – порт /dev/ttyS1 на скорости 115200
channelRS232  chRS232(«/dev/ttyS1», «115200»);

// Регистрация обработчика прихода данных (см. пп.0.)
eventHandler::instance().addFdHandler(chRS232.getFd(),
                                      fdRS232Handler);
….....................
// Передача данных
// pkt – подготовленный пакет типа rawData, см. пп.0

chRS232.recvData(pkt.data(), pkt.size());
….....................
// Запуск цикла обработки событий
// С этого момента обеспечивается приём данных
// функцией fdRS232Handler(int)
eventHandler::instance().eventLoop();
….....................
```

В приведённом примере отправка данных по каналу RS232 производится в любой момент времени. Приём данных осуществляется только по событию, генерируемому обработчиком событий (вызов функции **fdRS232Handler()**). Принятые данные обычно передаются классу-обработчику протокола **recProto**, собирающему принятые данные в пакеты и контролирующему их целостность.

6.4.2. Класс-канал связи по сети Интернет, протокол UDP

Класс-канал связи **channelUDP** требует задания лишь одного параметра – номера порта приёма данных. Приём и отправка данных по протоколу **UDP** осуществляются системными вызовами **recvfrom()** и **sendto()**. Настройка обмена по протоколу **UDP** состоит из двух этапов, как показано в пп. 5.5.2: создание **UDP**-сокета и привязка созданного со-

кета к порту приёма данных. Эти действия выполняет конструктор класса **channelUDP**. Описание класса **channelUDP** приведено на листинге ниже:

```
/**
    @file chan_udp.h
     Класс-канал связи по сети интерент (протокол
UDP)
*/
#ifndef chan_udp_h_E
#define chan_udp_h_E

#include "channel.h"

class channelUDP: public baseChannel{
public:
     // Конструктор
     channelUDP(int port);

     // Отправить данные в канал
     virtual int  sendData(const void* data, int len,
                     chanAdr* adr=NULL);

     // Принять данные из канала
     virtual int  recvData(void* data, int len,
                     chanAdr* adr=NULL);

     // Получить порт  приёма данных
     int getPort();

private:
     // Номер порта приёма данных
     int portUDP;
};

#endif /* chan_udp_h_E */
```

Как и в предыдущем примере, переопределены методы **sendData()** и **recvData()** класса-предка **baseChannel**.

Конструктор создаёт сокет **UDP** и привязывает его к порту приёма данных (порту прослушивания).

Если произошла ошибка, например порт приёма данных уже используется, то дескриптор файла, возвращаемый функцией **getFd()**, будет иметь значение **–1**. Если устройство-порт открыто и настроено, то функция **getFd()** вернёт дескриптор файла, по которому производится работа с сокетом.

Методы **sendData()** и **recvData()** имеют защиту от их использования при неинициализированном сокете. Метод **sendData()** передаёт данные **UDP**-сокета. В случае ошибки передачи возвращает **–1**. Если данные успешно переданы в порт, то возвращается количество переданных байтов.

Метод **recvData()** принимает данные из порта приёма данных, к которому привязан **UDP**-сокет. В случае ошибки приёма возвращает **–1**. Если данные успешно приняты из порта, то возвращается количество принятых байт.

Отметим, что метод **recvData()** блокирует выполнение программы, пока не появятся данные в порту приёма. Поэтому, рекомендуется вызывать его в обработчике появления данных в файле, описанного в модуле обработки событий в пп.0. В этом случае метод считает данные, не блокируя выполнннения программы.

Описание методов класса **channelUDP** приведено на листинге ниже:

```
/**
    @file chan_udp.cxx
     Класс-канал связи по сети интерент (протокол
UDP)
*/

#include <unistd.h>
#include <string.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#include "chan_udp.h"

channelUDP::channelUDP(int port):baseChannel(){
     // Адрес сокета
     struct sockaddr_in addr;

     portUDP = -1;
     // Создаём сокет

     fd = socket(AF_INET, SOCK_DGRAM, 0);
     if(fd < 0){
         // Ошибка
         return;
     }
     // Привязываем сокет к порту
     memset(&addr, 0, sizeof(addr));
     addr.sin_family = AF_INET;
     addr.sin_addr.s_addr = INADDR_ANY;
     addr.sin_port = htons(port);

     if (bind(fd, (struct sockaddr*)&addr,
              sizeof(struct    sockaddr_in)) < 0){
         close(fd);
         fd = -1;
         // Ошибка
         return;
     }
```

```
    portUDP = port;
}

int channelUDP::sendData(const void* data, int len,
                         chanAdr* adr){

    if((!adr) || (adr->type != chtUDP)){
        return(-1);
    }

    struct sockaddr_in addr;

    // Привязываем сокет к адресу и порту
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr(adr-
>ip.c_str());
    addr.sin_port = htons(adr->port);

    // Отправляем данные
    return(sendto(fd, data, len, 0,
            (struct sockaddr *) (&addr),
sizeof(addr)));
}

int channelUDP::recvData(void* data, int len,
chanAdr* adr){
    struct sockaddr_in addr;
    char ip[0x20];
    socklen_t adr_len=sizeof(ip);

    int l = recvfrom(fd, data, len, 0,
                (struct sockaddr*)&addr, &adr_len);

    if(l<0){
        return(-1);
    }

    if(adr){
        sprintf(ip,"%d.%d.%d.%d",
            (addr.sin_addr.s_addr & 0xff),
            ((addr.sin_addr.s_addr >> 8) & 0xff),
            ((addr.sin_addr.s_addr >> 16) & 0xff),
            ((addr.sin_addr.s_addr >> 24) & 0xff));

        adr->type = chtUDP;
        adr->ip = ip;
        adr->port = addr.sin_port;
    }

    return(l);
}
```

```
int channelUDP::getPort(){
     return(portUDP);
}
```

Использование класса **channelUDP** аналогично предыдущему примеру с классом **channelRS232** и приведено на листинге ниже:

```
….....................
// Функция приёма данных из UDP-сокета
void fdUdpHandler(int){
     // Данные в порту приёма есть, читаем их
     char buf [0x100];
     int len =  chUDP.recvData(buf,sizeof(buf));
     // len – количество прочитанных байт или
     // –1 в случае ошибки
}
….....................
// Создадим экземпляр класса  channelUDP
// Канал связи – порт приёма данных 4000
channelUDP   chUDP(4000);

// Регистрация обработчика прихода данных (см. пп.0.)
eventHandler::instance().addFdHandler(chUDP.getFd(),
                                      fdUdpHandler);
….....................
// Передача данных
// pkt – подготовленный пакет типа rawData, см. пп.0
// adr – адрес получателя, которому передаётся пакет
chanAdr adr;
adr.type = chtUDP;    // Тип адреса – IP
adr.ip = «192.168.3.123» // IP-адрес получателя пакета
adr.port = 5000;      // порт получателя пакета

chUDP.recvData(pkt.data(), pkt.size(), &adr);
….....................
// Запуск цикла обработки событий
// С этого момента обеспечивается приём данных
// функцией fdUdpHandler(int)
eventHandler::instance().eventLoop();
….....................
```

В приведённом примере отправка данных по каналу Интернет производится в любой момент времени. Приём данных осуществляется только по событию, генерируемому обработчиком событий (вызов функции **fdUdpHandler()**). Принятые данные обычно передаются классу-обработчику протокола **recProto**, собирающему принятые данные в пакеты и контролирующему их целостность.

Если необходимо узнать IP-адрес и порт отправителя пакета, то функция приёма пакета **recvData()** должна вызываться с тремя параметрами:

```
….....................
chanAdr radr; // IP-адрес приёмника
int len =  chUDP.recvData(buf,sizeof(buf), &radr);
….....................
```

Третий параметр – **radr** – адрес отправителя в формате адреса канала связи. В данном случае это IP-адрес и порт отправителя.

6.5. Модуль таблицы маршрутизации

Под термином «маршрутизация» в данном случае понимается возможность передавать пакеты данных между устройствами, не связанными непосредственно одним каналом связи Например, (см. рис. 12) пакеты от ВИП-МК RS не могут передаваться непосредственно на ВИП-МК 1 – ВИП-МК N, поэтому ПК должен выполнять функции маршрутизатора между каналами связи **RS232** и **Ethernet**. Описание класса-таблицы маршрутизации routeTable приведено на листинге ниже:

```
/**
  @file routeable.h
      Класс-таблица маршрутизации
 */

#ifndef ROUTETABLE_H
#define ROUTETABLE_H

#include <string>
#include <vector>
#include <channel.h>

// Запись в таблице маршрутизации
typedef  struct routeRecord{
     u_int32_t    sysadr; // Номер устройства в системе
     chanAdr  chanadr; // Адрес в формате канала связи
}routeRecord;

// Тип-таблица маршрутизации
typedef std::vector<routeRecord>   routeTableType;

class routeTable{
     // Тип-список строк
     typedef std::vector<std::string>  stringList;

public:
     // Конструктор
     routeTable(const char* path=NULL);

     // Загрузка таблицы
     bool load(const char* path=NULL);

     // Поиск маршрута по номеру устройства в системе
     bool search(u_int32_t sysadr, chanAdr& adr);
```

```
        // Получить экземпляр объекта
        static routeTable& instance();

    private:
        // Путь к файлу-таблице маршрутизации
        std::string  pathTable;

        // Таблица маршрутизации
        routeTableType   table;

        // Разбиение строки на поля,
        // разделённые пробелами
        stringList splitString(std::string& s);

        // Определение типа канала и адреса
        void newRecord(stringList list);
    };

    #endif // ROUTETABLE_H
```

Таблица маршрутизации представлена в нашем примере классом **routeTable**. Класс-таблица маршрутизации **routeTable** имеет функции загрузки таблицы маршрутизации из файла и функции поиска записи о маршруте в таблице по номеру устройства в системе.

Файл-таблица маршрутизации является текстовым файлом следующего формата:

```
# Файл таблицы маршрутизации
# № в системе     Тип канала   Адрес
  10              UDP          127.0.0.1    4000
  24              UDP          127.0.0.1    4001
  3232                UDP          127.0.0.1    4002
  412             RS232
  1100                RS232
  600             UDP          127.0.0.1    4003
```

Каждая строка файла-таблицы маршрутизации имеет несколько полей, разделённых пробелами.

Строки, начинающиеся с символа # (решётка), считаются комментариями и игнорируются при загрузке файла.

Первое поле каждой строки – номер устройства в системе, для которого данная строка определяет маршрут.

Второе поле – тип канала связи. В нашем примере определены два типа каналов связи: **RS232** и **UDP**.

Третье и последующие поля являются адресом в системе адресации канала связи указанного типа. Например, для типа канала **RS232** адрес отсутствует, для канала **UDP** адрес определяется полями 3 (IP-адрес) и 4 (номер порта).

При загрузке файла-таблицы маршрутизации формируется массив записей типа **routeRecord**.

Описание методов класса **routeTable** приведено на листинге ниже:

```
/**
  @file routeable.cxx
      Работа с таблицей маршрутизации
 */

#include <iostream>
#include <fstream>
#include <memory>

#include "routetable.h"

// Текстовые константы-типы каналов связи
#define  TXT_UDP      "UDP"
#define  TXT_RS232    "RS232"

routeTable::routeTable(const char* path){
     if(path){
         load(path);
     }
}

bool routeTable::load(const char* path){

     table.clear();
     pathTable.clear();

     if(path){
         pathTable = path;
     }

     std::ifstream file(path);

     if (!file) {
         return(false);
     }

     std::string line;

     while (getline(file, line)) {
         newRecord( splitString(line) );
          }

     file.close();

     return(true);
}

routeTable::stringList
routeTable::splitString(std::string& s){
     stringList list;
```

```
    std::string  ss;

    list.clear();
    ss.clear();

    bool pass=true;

    for(auto i:s){
        if( (pass)&&(!isblank(i)) ){
            pass=false;
        }

        if( (!pass)&&(!isblank(i)) ){
            ss.push_back(i);
        }

        if( (!pass)&&(isblank(i)) ){
            pass=true;
            if(ss.length()){
                list.push_back(ss);
            }
            ss.clear();
        }
    }

    if(ss.length()){
        list.push_back(ss);
    }

    return(list);
}

void routeTable::newRecord(routeTable::stringList list){
    // Слишком короткая запись, игнорируем
    if(list.size()<2){
        return;
    }

    // Комментарий
    if(list[0][0] == '#'){
        return;
    }

    routeRecord  r;

    // Номер в системе
    r.sysadr = atoi(list[0].c_str());

    // Канал Ethernet UDP
    if(! (list[1].compare(TXT_UDP)) ){
        if(list.size()<4){
            return;
        }
```

```
            r.chanadr.type = chtUDP;
            r.chanadr.ip = list[2];
            r.chanadr.port = atoi(list[3].c_str());

        table.push_back(r);
        }
        // Канал RS232
        else if(! (list[1].compare(TXT_RS232)) ){
            r.chanadr.type = chtRS232;
            table.push_back(r);
        }
    }

    bool routeTable::search(u_int32_t sysadr, chanAdr& adr){
        for(auto &i:table){
            if(i.sysadr == sysadr){
                adr = i.chanadr;
                return(true);
            }
        }
        return(false);
    }

    static std::unique_ptr<routeTable> routeTableIn-
    stance;
    routeTable& routeTable::instance(){
        if(!routeTableInstance){
            routeTableInstance.reset(new routeTable);
        }
        return(*routeTableInstance);
    }
```

Обычно в программе применяется один экземпляр класса **routeTable**, поэтому введён метод **instance()** для создания класса-синглетона.

Метод **load()** загружает файл-таблицу маршрутизации, расположенный по указанному пути.

С целью поиска записи в таблице маршрутизации для заданного номера устройства в системе используется метод **search()**. Если запись найдена, то тип и адрес канала связи помещаются в переменную **adr**, а функция возвращает значение **true**. Если записи для заданного номера устройства в системе нет в таблице, то метод **search()** вернёт значение **false**.

Пример использования класса **routeTable** приведён на листинге ниже:

```
….....................
// Загружаем файл ./route.tbl с таблицей маршрутизации
if(!routeTable::instance().load(«./route.tbl»)){
    // Ошибка загрузки таблицы
}
```

```
else{
     // Таблица загружена
}
…..................
// Поиск маршрута для устройство
// номером 25 в системе
chanAdr adr;

if(routeTable::instance().search(25, adr)){
     // Запись найдена
     // маршрут помещён в описатель адреса adr
}else{
     // Запись не найдена
}
…..................
```

В приведённом примере класс **routeTable** используется как класс-синглетон. Сначала производится загрузка таблицы из файла, затем производится поиск адреса типа **chanAdr**, указывающего по какому каналу связи и какому канальному адресу следует отправлять пакеты для устройства с номером **25** в системе.

6.6. Модуль начальной инициализации

Модуль начальной инициализации создаёт все необходимые классы для работы программы, передавая им различные параметры. Например, для создания класса-канала связи по порту **RS232 channelRS232** необходимо задать файл-устройство **RS232** и скорость обмена данными; для создания класса-канала связи по каналу Интернет **channelUDP** необходимо задать номер порта приёма данных и т.п.

Жёстко задавать все параметры в программе не следует, так как это приведёт к необходимости трансляции программы каждый раз при изменении параметров функционирования программы. Широко используемым механизмом для передачи параметров в программу является механизм передачи параметров из командной строки.

Функция **main()** программы на языках **C** и **C++,** с которой начинается выполнение программы, в качестве входных параметров получает параметры командной строки в виде списка строк.

Прототип функции **main()** выглядит так:

```
int main(int argc, const char* argv[]);
```

где **argc** – количество параметров командной строки; **argv** – список строк-параметров командной строки.

Список параметров **argv** всегда имеет как минимум один член – имя исполняемого файла.

С целью разбора командной строки создан класс **cmdLineParse**. При инициализации данного класса ему передаётся таблица, описывающая параметры командной строки и определяющая функции обработки каждого из них.

Класс **cmdLineParse** позволяет передавать параметры в командной строке в любом порядке.

В общем случае имеется три типа параметров командной строки, которые разбираются классом **cmdLineParse**: ключи, одиночные параметры ключей и составные параметры ключей.

Параметры-ключи принято начинать с символа **'-'** (горизонтальная черта) или **'--'** (две горизонтальные черты подряд).

Параметры-ключи, состоящие из одного символа (так называемые «короткие ключи»), обычно начинают с символа **'-'** (горизонтальная черта), а параметры-ключи, состоящие из нескольких символов (так называемые «длинные ключи»), обычно начинают с символа **'--'** (две горизонтальные черты подряд).

Пример командной строки:

```
# ./messenger  --udp 4000 --rs232 /dev/ttyS1,19200 -v
```

В данном примере нулевым параметром командной строки будет имя исполняемого файла **./messenger**. Длинные ключи **--udp** и —**rs232**. Короткий ключ **-v**.

Параметром ключа **--udp** является одиночный параметр ключа – строка **4000**.

Параметром ключа **--rs232** является составной параметр ключа, состоящий из двух строк-параметров **/dev/ttyS1** и **19200**.

Ключ **-v** не имеет параметров.

Описание класса **cmdLineParse** приведено на листинге ниже:

```
/**
    @file cmdlineparse.h
     разбор аргументов командной строки
*/

#ifndef CMDLINEPARSE_H
#define CMDLINEPARSE_H

#include <unistd.h>
#include <string>
#include <vector>
#include <functional>

// Разбор командной строки и установка параметров
class cmdLineParse{
public:
```

```
        // Описатель аргумента командной строки
        typedef struct cmdLineArgDsc{
            // Ключ (NULL - конец списка)
            const char* key;

            // Есть ли аргумент у данного ключа
            bool arg;

            // Функция обработки данного ключа
            std::function<void(const char*,
    cmdLineParse&)>
                                          keyFunc;

            // Подсказка по данному ключу
            const char* help;
        }cmdLineArgDsc;

        // Список строк для аргументов,
        // разделённых запятыми
        typedef std::vector<std::string> stringList;

        // Конструктор
        cmdLineParse(cmdLineArgDsc* argsdsc);

        // Обработать ключи
        bool handle(int argc, const char* argv[]);

        // Разбить строку arg на
        // несколько подстрок, разделённых
        // символом separator и поместить
        // список строк в split
        stringList splitArg(std::string arg, char sepa-
    rator=',');

        // Вывести подсказку
        void help(FILE* f=stdout);

    private:
        // Указатель на список описателей
        // аргументов командной строки
        cmdLineArgDsc* argsDsc;

        // Поиск описателя ключа в списке
        cmdLineArgDsc* searchKey(const char* key);

    };

    #endif // CMDLINEPARSE_H
```

Тип **cmdLineParse::cmdLineArgDsc** является описателем ключа командной строки. В описание входят следующие поля:

key – строка-ключ командной строки;

keyFunc – функция обработки данного ключа командной строки;

help – строка-подсказка по данному ключу.

Массив структур-описателей ключей командной строки передаётся конструктору класса **cmdLineParse**. Признаком конца массива описателей ключей командной строки является описатель с полем **key=NULL**.

Метод **handle()** принимает ключи командной строки в том же формате, что представлены в функции **main()**, и производит разбор командной строки.

Если разбор прошёл успешно, то метод **handle()** возвращает значение **true**. Если обнаружен ошибочный ключ или не указан параметр ключа, то метод завершается и возвращает значение **false**.

Метод **splitArg()** предназначен для разбора составных параметров, т.е. параметров ключей, разделённых заданным символом-разделителем (по умолчанию этот символ – запятая).

Метод **help()** выводит строки-подсказки в заданный файл. По умолчанию вывод подсказки осуществляется в файл **stdout**.

Описание методов класса **cmdLineParse** приведено на листинге ниже:

```
/**
    @file cmdlineparse.cxx
     разбор аргументов командной строки
*/

#include "cmdlineparse.h"
#include "string.h"

cmdLineParse::cmdLineParse(cmdLineArgDsc* argsdsc){
     argsDsc = argsdsc;
}

bool cmdLineParse::handle(int argc, const char*
argv[]){
     if(!argsDsc){
         return(false);
     }

     cmdLineArgDsc* r;
     int counter = 1;
     const char* arg;

     while( counter < argc){
         if(!( r=searchKey(argv[counter]) )){
             // Ошибка – нет такого ключа
             return(false);
         }
         // Проверка – есть ли у ключа аргумент
         arg=NULL;
         if(r->arg){
             counter++;
             if(counter>=argc){
```

```
                    // Ошибка - нет аргумента
                    return(false);
                }
                arg = argv[counter];
            }
            // Вызов функции-обработчика ключа
            if( r->keyFunc ){
                r->keyFunc(arg, *this);
            }
            // Переход к следующему ключу
            counter++;
        }
        //
        return(true);
}

void cmdLineParse::help(FILE* f){
        cmdLineArgDsc* r=argsDsc;

        if(!r){
            return;
        }

        fprintf(f, "Usage:\n");
        while(r->key){
            fprintf(f, "\t%s - %s\n",r->key, r->help);
            r++;
        }

}

cmdLineParse::cmdLineArgDsc*
   cmdLineParse::searchKey(const char* key){
        cmdLineArgDsc* r=argsDsc;

        if(!r){
            return(NULL);
        }

        while(r->key){
            if( !strcmp(key, r->key)){
                return(r);
            }
            r++;
        }

        return(NULL);
}

cmdLineParse::stringList
        cmdLineParse::splitArg(std::string arg,
                               char separator){
```

```
        std::string s;
        stringList split;

        split.clear();
        s.clear();

        for(auto c:arg){
            if( c != separator){
                s.push_back(c);
            }
            else{
                split.push_back(s);
                s.clear();
            }
        }

        if(s.size() != 0){
            split.push_back(s);
        }
        return(split);
    }
```

Пример использования класса cmdLineParse приведён на листинге ниже:

```
    ….....................
    #include <cmdlineparse.h>

    // Описатель ключей командной строки
    cmdLineParse::cmdLineArgDsc cmdLineHandlers[ ] ={
      {
          "--help", false,
          [ ](const char* arg, cmdLineParse& obj){
              obj.help();
              exit(0);
          },
          "Вывод подсказки"
      },
      {
          "--udp", true,
          [ ](const char* arg, cmdLineParse& obj){
              // arg – одиночный параметр ключа --udp
          },
          "Включение канала Ethernet UDP. --udp <port>"
      },
      {
          "--rs232", true,
          [ ](const char* arg, cmdLineParse& obj){
              cmdLineParse::stringList l =
    obj.splitArg(arg);
```

```
            // arg – составной параметр ключа – -rs232
            // l – список строк составного параметра arg
        },
        "Включение канала RS232. --rs232 <de-
vice>,[speed]"
    },
    // Конец списка
    {NULL,false,nullptr,NULL}
};

int main(int argc, const char* argv[]){
        cmdLineParse cmdlineparse(cmdLineHandlers);
        if(!cmdlineparse.handle(argc,argv)){
            // Ошибка – неверный ключ или не определён
            // параметр
        }
}
…....................
```

В приведённом примере массив описателей ключей **cmdLineHandlers** задан в виде константы. Очень удобно использовать лямбда-функции **C++** для описания функций обработки ключей командной строки, как показано в примере.

В приведённом примере описаны следующие ключи:

--help – вывод подсказки. Ключ не имеет параметров;

--udp – включение канала **Ethernet UDP**. Ключ имеет один простой параметр;

--rs232 – включение канала **RS232**. Ключ имеет один составной параметр.

Функции обработки ключа типа **keyFunc** передаётся два параметра: строка-параметр ключа **arg** и ссылка на объект **cmdLineParse**.

Если ключ не имеет параметров, то параметр **arg=NULL**. Если передаётся составной параметр, то его можно разбить на составляющие функцией **obj.splitArg(arg)**. Является параметр простым или составным зависит только от функции обработки ключа.

ЗАКЛЮЧЕНИЕ

Распределенные микропроцессорные системы – это сформировавшаяся сфера микропроцессорной техники, обладающая своей спецификой и ярко выраженным классом решаемых задач и методами их решения. Получение теоретической подготовки, а также практических навыков проектирования и реализации РМПС является актуальной задачей при подготовке специалистов в области микропроцессорных систем.

В части 1 настоящего учебного пособия приведены краткие теоретические основы построения распределённых микропроцессорных систем, изложены принципы организации программного обеспечения и приведены практические примеры построения стендов и программного обеспечения.

Для студентов, магистрантов и аспирантов специальностей 230100 – «Информатика и вычислительная техника», 211000.62 – «Конструирование и технология электронных средств», 05.13.11 – «Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей» особый интерес представляют практические решения типовых задач, приведённых в части 1 настоящего практикума, а именно:

- вопросы трансляции программного обеспечения;
- вопросы организации проектов и написания сборочных скриптов;
- типовые решения по организации событийного интерфейса обработки файловых дескрипторов;
- вопросы организации приёма и передачи данных по различным каналам связи и т.д.

Надеемся, что настоящий практикум окажется полезным для обучающихся по смежным с указанными специальностям.

ЛИТЕРАТУРА

1. Таненбаум Э., Стеен М. Распределенные системы. Принципы и парадигмы. – СПб.: Питер, 2003. – 877 с.

2. Олзоева С.И. Распределенное моделирование в задачах разработки АСУ. – Улан-Удэ: Изд-во ВСГТУ, 2005. – 219 с.

3. Kshemkalyani D., Singhal M. Distributed Computing: Principles, Algorithms, and Systems. – New York: Cambridge University Press, 2008. – 756 с.

4. Бунин С.В., Сонькин М.А., Харламов А.М., Ямпольский В.З. Новые функциональные возможности системы оповещения и резервной документированной связи внутренних войск МВД России // Изв. ТПУ. – 2008. – Т. 312, вып. 2.

5. Гринемаер В.В., Шамин А.А. Мониторинг работоспособности системы оповещения и контроль параметров удаленных объектов // Приборы. – 2012. – №12.

6. Багдасарова Е.П. Применение современных технологий сбора данных с наблюдательной сети // Метеоспектроскопия. – 2005. – № 2. – С. 89–93.

7. Сонькин М.А., Гринемаер В.В., Печерская Е.И. и др. Опыт создания интегрированной системы сбора метеоданных с сети труднодоступных станций на основе спутниковых и радиоканалов // Кибернетика и вуз. – 2003. – Вып. № 30. – С. 87–95.

8. Семёнов Ю.А. Протоколы Интернет. – М.: Горячая линия-Телеком, 2001. – 1100 с.

9. IEEE STANDARTS ASSOCIATION [Электронный ресурс]. – Режим доступа: http://standards.ieee.org/getieee802/portfolio.html свободный. – Загл. с экрана.

10. Голубцов М.С. Микроконтроллеры AVR: от простого к сложному. – М.: СОЛОН-Пресс, 2003. – 288 с.

11. Тревор М. Микроконтроллеры ARM7. Семейство LPC2000 компании Philips. Вводный курс. – М.: Додека XXI век, 2006. – 240 с.

12. Richard Barry. USING THE FREERTOS REAL TIME KERNEL. – 2009. – 163 с.

13. Рэймонд Э. Искусство программирования для Unix. – М.: Изд. дом «Вильямс», 2005.

14. Стивенс Р., Раго С. UNIX. Профессиональное программирование. 2-е изд. – СПб.: Символ-Плюс, 2007. – 1040 с.

15. Managing Projects with GNU Make, 3.Xth Edition / O'Reilly & Associates, Inc., 2004 [Электронный ресурс]. – Режим доступа: http://oreilly.com/catalog/make3/book/ свободный. – Загл. с экрана.

16. Франка П. C++: учебный курс. – СПб.: Питер, 2003. – 521 с.

17. Интерфейс прикладных программ (API) [Язык программирования C]: в 2 т. – М.: НИИСИ РАН, 1999.

18. Гук М. Аппаратные интерфейсы ПК. Энциклопедия. – СПб.: Питер, 2002. – 528 с.

19. Дуглас Э. Камер. Сети TCP/IP. Принципы, протоколы и структура. – 4-е изд. – М.: Изд. дом «Вильямс». – 2003. – Т. 1.

20. Стивенс У.Р., Феннер Б., Рудофф Э.М. UNIX, разработка сетевых приложений. – 3-е изд. – СПб.: Питер, 2007. – 1039 с.

Для заметок

Учебное издание

Сонькин Михаил Аркадьевич,
Шелупанов Александр Александрович,
Шамин Алексей Алексеевич

РАСПРЕДЕЛЁННЫЕ МИКРОПРОЦЕССОРНЫЕ СИСТЕМЫ

УЧЕБНОЕ ПОСОБИЕ
ПО КУРСОВОМУ ПРОЕКТИРОВАНИЮ

ЧАСТЬ 1

Корректор – В.Г. Лихачева
Верстка – В.М. Бочкаревой
Дизайн обложки – Е. Межевая

Издательство «В-Спектр»
Подписано к печати 20.12.2013.
Формат 60×84$^{1}/_{16}$. Печать трафаретная.
Печ. л. 6,5. Тираж 500 экз. Заказ 81.

Тираж отпечатан в издательстве «В-Спектр»
ИНН/КПП 7017129340/701701001, ОГРН 1057002637768
634055, г. Томск, пр. Академический, 13-24, Тел. 49-09-91.
E-mail: bvm@sibmail.com
