ИНСТИТУТ ФИЗИКИ им. Л. В. КИРЕНСКОГО СО РАН
КРАСНОЯРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
КРАСНОЯРСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А. Н. Втюрин, А. Г. Агеев, А. С. Крылов

ЭВМ
В ФИЗИЧЕСКОМ ЭКСПЕРИМЕНТЕ

Учебное пособие

Новосибирск 2003

УДК 004
ББК 32.97
В 87

Втюрин А. Н., Агеев А. Г., Крылов А. С. / ЭВМ в физическом эксперименте. – Новосибирск, Издательство СО РАН, 2003. – 150 с.

Простота и доступность современных ЭВМ, с одной стороны, и усложнение физического эксперимента, с другой, привели к тому, что в современной экспериментальной физике компьютеры находят все более широкое применение. В связи с этим современная экспериментальная физика требует знания не только численных методов и языков программирования, но и архитектуры, элементов устройства управляющих ЭВМ, принципов их организации, существующих методов интеграции ЭВМ с физическими приборами. Предлагаемая книга знакомит читателя с основами архитектуры ЭВМ, применяемых для управления экспериментальными установками, устройством и принципами работы наиболее распространенного интерфейсного оборудования, алгоритмами управления экспериментом и оперативной обработки экспериментальных данных. Текст сопровождается контрольными вопросами, посвященными принципам организации прикладного программного обеспечения и реализации основных алгоритмов оперативной обработки результатов эксперимента, а также примерами практической реализации изложенного материала для управления отдельными интерфейсными модулями и работы на управляемых ЭВМ установках.

Книга адресована студентам и аспирантам физических и инженерно-физических факультетов университетов.

Рецензенты:

Доктор физико-математических наук, профессор И. Н. Флеров,
кандидат физико-математических наук Н. В. Волков,
кандидат физико-математических наук А. Н. Ботвич

Утверждено к печати
Ученым советом Института физики СО РАН

ISBN 5-7692-0630-6

ПРИНЦИПЫ И СРЕДСТВА АВТОМАТИЗАЦИИ ФИЗИЧЕСКОГО ЭКСПЕРИМЕНТА

ПРЕДПОСЫЛКИ ПРИМЕНЕНИЯ КОМПЬЮТЕРОВ В ЭКСПЕРИМЕНТАЛЬНОЙ ФИЗИКЕ

Усложнение экспериментальной техники

Проведение экспериментальных научных исследований можно представить в форме диалога экспериментатора с природой. Природа отвечает на поставленные человеком вопросы, предполагающие поиск зависимостей между явлениями, результатами целенаправленных наблюдений и измерений, и теоретическими понятиями. Каждый исторический этап в развитии науки требует и приводит к созданию новых технических средств ведения этого диалога. Современный этап развития экспериментальной физики характеризуется чертами, которые зачастую делают невозможным проведение измерений без использования комплекса средств, образующих понятие автоматизации физического эксперимента.

Так, характерной чертой современного эксперимента является огромное количество получаемой в его ходе информации, накопление, хранение и обработка которой возможны только с использованием вычислительной техники. Высокие скорости поступления информации и необходимость ее обработки не после завершения эксперимента, а непосредственно в ходе измерений, требуют прямой связи ЭВМ с измерительной аппаратурой, что, в свою очередь, существенно расширяет возможности экспериментальной установки и повышает качество результата.

Еще одна характерная черта современного эксперимента – необходимость контроля и управления большим числом служебных параметров в ходе эксперимента, что тоже наибо-

лее эффективно может быть осуществлено с помощью ЭВМ. Использование управляющей ЭВМ позволяет оперативно в ходе эксперимента изменять настройку аппаратуры, руководствуясь результатами предварительной обработки получаемых экспериментальных данных. Во многих случаях это существенно снижает затраты времени на проведение эксперимента и повышает точность измерений.

Совершенствование ЭВМ

В последние десятилетия произошел стремительный прогресс вычислительной техники. Скорости вычислений и объемы обрабатываемой информации выросли за последние 30–40 лет в сотни тысяч раз; в десятки тысяч раз снизились энергопотребление компьютеров, их размеры и цена. Это обусловило появление и массовое распространение персональных ЭВМ, предоставляющих пользователю те же, а часто и большие, ресурсы, которыми обладали в 60-х годах большие ЭВМ, а в 70-х – миникомпьютеры.

В 60-х годах расходы по приобретению и эксплуатации больших ЭВМ были сопоставимы со стоимостью наиболее крупных экспериментальных установок атомной и ядерной физики и обслуживались десятками, а то и сотнями высококвалифицированных специалистов. Разумеется, именно этими областями и ограничивалось, главным образом, их применение в физике.

Развитие элементной базы, появление транзисторов, а затем – интегральных схем, провело к появлению микроЭВМ, обладающих, при сравнимой производительности, существенно меньшими габаритами, стоимостью, более высокой надежностью и существенно меньшими требованиями к обслуживанию. Это сделало возможным их массовое применение в качестве не только вычислительных, но и управляющих ЭВМ, в том числе – в системах физического эксперимента средней сложности. К этому времени относится появление измерительно-вычислительных комплексов – специализированных систем, предназначенных, в том числе, для автомати-

зации физического эксперимента, начало массовой разработки устройств сопряжения компьютеров и измерительных устройств, создания соответствующего прикладного программного обеспечения.

Появление персональных компьютеров, как нового класса ЭВМ, послужило толчком к революционному перевороту во всех областях человеческой деятельности, в том числе – и в методах экспериментальной физики. Компьютер стал повседневным инструментом физика, столь же привычным, как ранее – калькулятор, а до него – логарифмическая линейка. Массовое распространение прикладного математического обеспечения, в том числе – специально ориентированного для обработки данных измерений породило естественное желание организовать непосредственный ввод этих данных в компьютер, а осуществление этой потребности приносит принципиально новые возможности, связанные с оперативной обработкой данных и управлением многопараметрическими экспериментами.

Новые возможности, предоставляемые автоматизацией

Применение простых и надежных вычислительных средств в системах автоматизации эксперимента, доступных для использования в лабораториях средних размеров, привело к массовой автоматизации рутинных процедур измерений и развитию соответствующего программного и аппаратного обеспечения. Одновременно перед экспериментаторами открылись новые, недоступные ранее возможности. В первую очередь это связано с высокой скоростью производимых операций и возможностью оперативной обработки больших объемов информации. Высокие характеристики и возможности современных ЭВМ и программного обеспечения, видимо, еще не до конца оценены специалистами-экспериментаторами. Временной диапазон процессов, воссоздаваемых, и, следовательно, в принципе контролируемых ЭВМ, простирается от 10^{-12} до 10^{8} с, что, видимо, неосуществимо в других техно-

логиях. Очевидным примером является использование ЭВМ в физике элементарных частиц, где современные исследования немыслимы без многопараметрических систем контроля параметров эксперимента и глубокой статистической обработки результатов; в нелинейной оптике и квантовой электронике только системы автоматизированного контроля позволяют проводить экспериментальные исследования процессов взаимодействия сверхкоротких лазерных импульсов; стали рутинными методики матричной регистрации изображений и многоканальной спектроскопии, требующие параллельной обработки колоссальных массивов информации.

ОБЛАСТИ ПРИМЕНЕНИЯ АВТОМАТИЗИРОВАННЫХ СИСТЕМ В ЭКСПЕРИМЕНТАЛЬНОЙ ФИЗИКЕ

Разработка и создание сложных и ресурсоемких экспериментальных измерительных физических комплексов «с нуля» является скорее исключением, чем повседневным занятием большинства физиков-экспериментаторов. Как правило, автоматизация эксперимента средней сложности происходит поэтапно и производится на основе некоторой уже действующей установки. При этом первым толчком является потребность в ее модернизации – при введении в эксперимент дополнительного контролируемого параметра или новой, более совершенной системы управления. В качестве примера можно привести замену асинхронного двигателя развертки в оптических спектрометрах устаревших моделей на шаговый – это значительно повышает точность позиционирования по спектру и практически исключает механические люфты, но требует введения цифровой системы управления разверткой, то есть той или иной степени компьютеризации управления этим параметром. Таким образом, первым шагом в автоматизации экспериментальной установки становится ***управление отдельными операциями***.

Очевидно, что самой основной операцией любого эксперимента является непосредственное проведение измерений,

поэтому, даже начав с автоматизации некоторой другой операции, экспериментатор довольно быстро приходит к мысли о передаче в управляющий компьютер и собственно результатов измерений – разумеется, если его мощность позволяет это сделать. При этом одновременно, в той или иной степени, реализуются ***сбор и обработка данных в ходе эксперимента*** – в зависимости от возможностей используемого компьютера и программного обеспечения это может быть простая визуализация получаемых результатов, предварительная фильтрация данных и пр.

Наличие как собственно измеряемых данных, так и возможностей автоматизированного контроля параметров эксперимента позволяет решить задачу автоматизации еще более полно – осуществить полную ***автоматизацию управления экспериментальной установкой***, когда управляющая ЭВМ контролирует (полностью или частично) условия эксперимента, ведет сбор измеряемых величин, анализирует их «на ходу» и в зависимости от результатов анализа корректирует условия в соответствии с поставленной экспериментатором задачей.

Как в процессе выполнения эксперимента, так и после его окончания, с полученными данными производятся определенные математические операции – в целях оптимизации процедуры их получения и для того, чтобы извлечь из них наиболее полную физическую информацию. В обоих случаях это предполагает проведение некоторой ***обработки экспериментальных данных.***

Кроме упомянутого выше, автоматизированные системы применяются также при создании баз экспериментальных данных, моделировании эксперимента, проектировании экспериментальных установок теоретической интерпретации экспериментальных данных, разработке и отладке программного обеспечения управления экспериментом и в других областях экспериментальной физики, однако этих вопросов мы будем касаться меньше.

БЛОК-СХЕМЫ СВЯЗИ ЭВМ С ЭКСПЕРИМЕНТАЛЬНЫМИ УСТАНОВКАМИ

Блок-схемы связи ЭВМ с экспериментальными установками или их отдельными узлами показаны на рис. 1.

Первая схема предполагает, что управляющая ЭВМ является неразрывной частью установки. Для создания подобного комплекса необходимо осуществить разработку специализированного компьютера, предназначенного для управления данным экспериментом, либо вести параллельную разработку и самой установки, и управляющей ЭВМ.

Это (особенно в случае управления достаточно сложным устройством) предполагает участие высококвалифицированных специалистов-разработчиков и программистов, и далеко не всегда возможно в условиях исследовательской лаборатории. Кроме того, такая схема автоматизации является весьма жесткой; сколько-нибудь заметное изменение режима измерений, проводимых на данной установке, потребует существенной переделки как аппаратного, так и программного обеспечения системы автоматизации. В связи с этим подобные схемы используются, как правило, при автоматизации серийно выпускаемых приборов, предназначенных для рутинных измерений (например, оборудование заводских лабораторий) либо для управления простейшими операциями (процессами) в ручном или автоматическом режиме с помощью небольших микропроцессорных систем (системы контроля и управления перемещением, регуляторы температуры).

Существенно более простым является использование в системе автоматизации готового стандартного компьютера. Разумеется, для того чтобы подключить его к установке, потребуется изготовить или приобрести некоторое интерфейсное устройство, позволяющее вводить данные, получаемые в

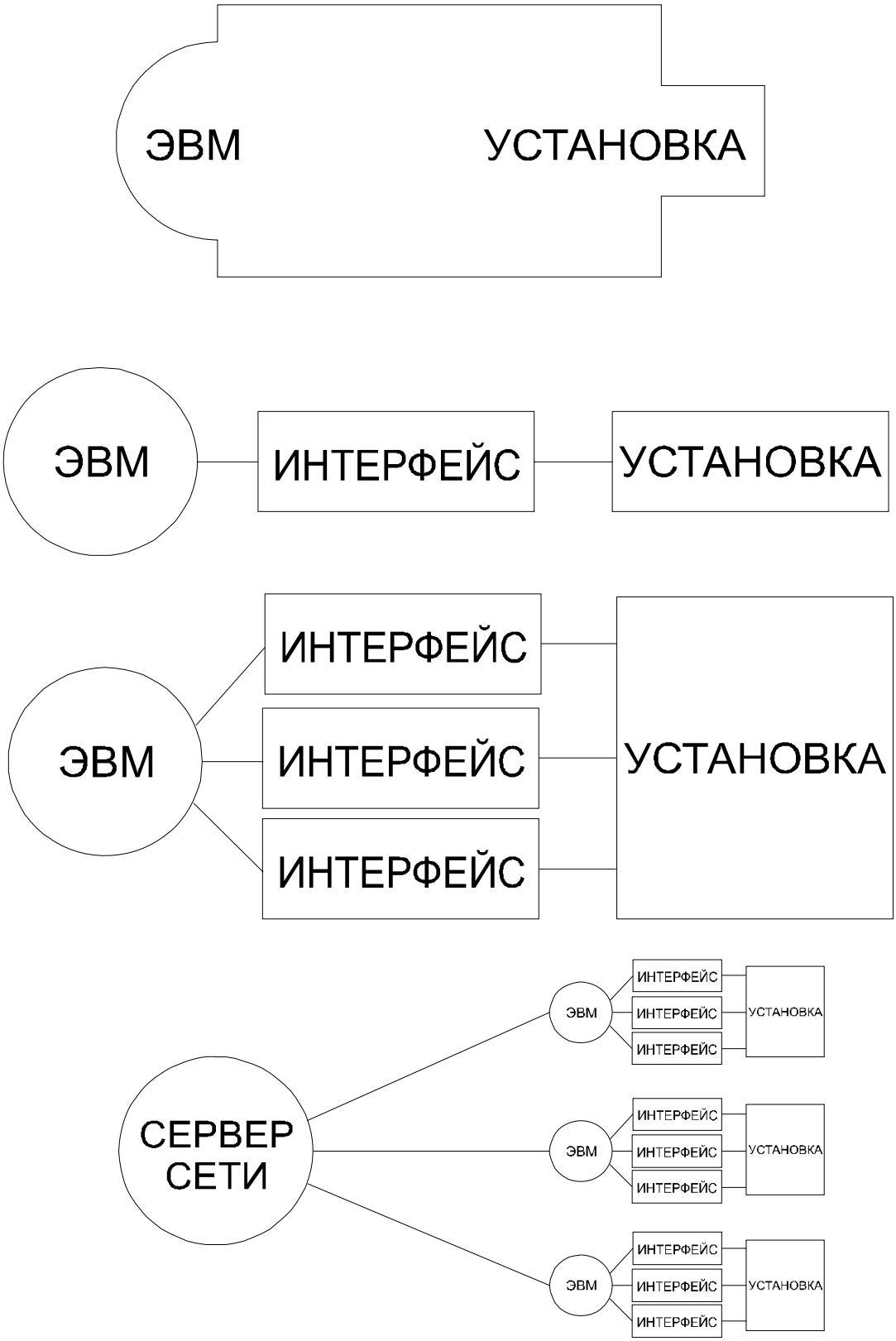


Рис. 1. Блок-схемы связи ЭВМ с экспериментальными установками

результате измерений, в компьютер и преобразовать управляющие сигналы компьютера к тому виду, который требуется управляющим узлам установки. Для его подключения к компьютеру можно воспользоваться либо портами, предназначенными для подключения внешних устройств компьютера (например, параллельные и последовательные порты IBM-совместимых компьютеров), либо внутренними шинами компьютера (ISA, PCI и др. шины IBM PC). Достоинством этого метода подключения является возможность использования стандартного программного обеспечения и развитых средств его разработки, однако здесь по-прежнему отсутствует аппаратная гибкость. Современные экспериментальные установки обеспечивают выдачу широкого набора измерительных сигналов самых разнообразных форматов, и требуют не менее разнообразных управляющих сигналов. Все это приводит к чрезвычайной сложности таких интерфейсных устройств, а внесение любых изменений в установку – требует коренной переработки и аппаратной и программной частей этого интерфейса.

При наличии относительно небольшого числа разнородных информационных и управляющих каналов имеет смысл использовать раздельное интерфейсное устройство для каждого из них. Если при их разработке используется некий стандартный подход, то модернизация установки приводит просто к добавлению еще одного или нескольких интерфейсных модулей и сравнительно небольшой коррекции управляющей программы. Однако при этом возникает ряд проблем, связанных с ростом нагрузки на управляющую ЭВМ при обслуживании большого количества разнородных внешних устройств. В частности, необходимо иметь большое количество каналов ввода-вывода данных в компьютер. В то же время в наиболее распространенных на сегодня компьютерах IBM PC количество таких каналов, как правило, весьма ограничено. В принципе имеется возможность объединения нескольких каналов ввода-вывода в рамках одного интерфейсного устройства. При таком объединении обычно исходят из общего функционального на-

значения объединяемых каналов связи (например, канал измерения температуры и ее регулировки) либо сходства осуществляемого ими преобразования (многоканальные аналого-цифровые либо цифро-аналоговые преобразователи). При этом предполагается возможность некоторой предварительной обработки поступающих данных самим объединенным интерфейсным устройством – по меньшей мере, оно должно уметь коммутировать преобразуемые сигналы.

Следующим шагом в этом направлении является использование модульных интерфейсов (КАМАК, ВЕКТОР, PXI, VXI, ряд разработок фирм, выпускающих сложное измерительное оборудование для научных исследований). Основной идеей модульного интерфейса является использование одного устройства – контроллера, связанного непосредственно с управляющей ЭВМ, которое осуществляет коммутацию всех сигналов между остальными интерфейсными модулями и компьютером. При этом от ЭВМ требуется только один канал для подключения контроллера, и вся установка может быть представлена как одно, хотя и достаточно сложное, внешнее устройство. Для того чтобы все остальные интерфейсные модули можно было подключить к контроллеру, требуется высокая степень стандартизации – механической, электрической, программной. Разумеется, в некоторых случаях это усложняет конструкцию отдельных модулей (и, как следствие, увеличивает первоначальные затраты при создании установки), но существенно упрощает разработку управляющего программного обеспечения и последующую модернизацию установки. Даже замена управляющей ЭВМ требует, в худшем случае, только замены контроллера – все остальные интерфейсные модули остаются теми же.

С целью снижения нагрузки на управляющую ЭВМ можно использовать также стандартные сетевые решения. При этом в случае автоматизации сравнительно небольшой компактно расположенной установки сеть используется просто для передачи части нагрузки на сервер (например, для хранения и обработки больших объемов данных), тогда как при об-

служивании сложных распределенных измерительных систем могут создаваться управляющие многомашинные комплексы, в которых в сеть объединяются ЭВМ, контролирующие работу отдельных узлов установки. Организация подобных сетей достаточно подробно описана в многочисленных литературных источниках и выходит за рамки данной книги.

Приступая к созданию автоматизированной установки, при выборе схемы автоматизации необходимо решить следующие вопросы:

1. Четко сформулировать задачи автоматизации.

2. Построить алгоритм работы автоматизированной установки.

3. Проработать систему измерений, определить набор необходимых интерфейсных модулей.

4. Предусмотреть перспективы развития установки, вопросы надежности работы.

5. Учесть финансовые возможности.

Требования, определяющие параметры используемого компьютера:

1. Объем накапливаемых и обрабатываемых данных.

2. Размер программ управления и обработки.

3. Скорость приема/передачи данных.

4. Скорость обработки данных.

5. Необходимое периферийное оборудование.

Из средств вычислительной техники, используемых в эксперименте, наиболее популярны:

1. Микропроцессорные наборы – как упоминалось выше, используются для управления отдельными простыми операциями.

2. Компьютеры типа PDP-11 (СМ, Электроника, ДВК) – 16-разрядные ЭВМ, выпускавшиеся в 80-х годах и специально разработанные для управления сложными процессами. В настоящее время не производятся, однако имеется большое количество действующего оборудования, использующего эти компьютеры в качестве управляющих. Хотя эти компьютеры и обладают сравнительно низкими вычислительными ресурсами, их архитектура позволяет достаточно просто подключать десятки внешних устройств.

3. Персональные компьютеры типа IBM PC – наиболее распространены, выпускается большое число интерфейсных устройств для подключения к ним через стандартные порты, внутренние шины и с помощью модульных интерфейсов, имеется развитое программное обеспечение. В то же время их архитектура не рассчитана на большое число каналов ввода-вывода информации.

АППАРАТНОЕ ОБЕСПЕЧЕНИЕ АВТОМАТИЗАЦИИ ЭКСПЕРИМЕНТА

Архитектура ЭВМ

Основная функция любой ЭВМ – обработка информации, представленной в цифровом виде. Для этого необходимо иметь возможности:

1. Представить любые данные в заданном цифровом виде (как правило, это двоичные числа).

2. Записывать и считывать эти данные по мере необходимости.

3. Обрабатывать эти данные – т. е. иметь возможность описания тех операций, которые с ними возможно и необходимо совершать.

ПРЕДСТАВЛЕНИЕ ДАННЫХ В ЭВМ

Целые положительные числа – представляются в *двоичной системе исчисления*. Для упрощения записи и восприятия пользователем часто в качестве промежуточных используются восьмеричная или шестнадцатеричная системы.

Целые числа со знаком – представляются в виде *двоичного дополнительного кода*. При этом старший двоичный разряд определяет знак числа (0 – плюс, 1 – минус). Для положительных чисел далее приводится его двоичное значение (например $011_2 = +3_{10}$). Для отрицательного числа n в этих разрядах приводится двоичное значение $n_{max} - |n| + 1$, где n_{max} – наибольшее целое число, представимое в пределах заданного количества двоичных разрядов. Например, если пользоваться 3-разрядными числами (старший разряд – знаковый, т. е. $n_{max} = 011_2 = +3_{10}$), то:

Десятичные числа	–3	–2	–1	0	1	2	3
Дополнительный код (3 разряда)	101	110	111	000	001	010	011

При такой кодировке, несмотря на то, что смысл старшего разряда отличается от остальных, тем не менее сохраняется возможность выполнения арифметических операций обычным путем (с учетом того, что разряды, вышедшие за пределы длины данного представления, теряются). Например, для 3-разрядного представления:

$$-3_{10} + 3_{10} = 0_{10},\ 101_{2д} + 011_{2д} = (1)000_{2д},$$

$$-2_{10} + 3_{10} = 1_{10},\ 110_{2д} + 011_{2д} = (1)001_{2д},$$

здесь в скобках показаны потерянные разряды.

В цифровых измерительных устройствах, широко применяемых в качестве интерфейсов, часто используется *двоично-десятичный код*. Как и в предыдущем случае, здесь старший разряд является знаковым. Остальные разряды делятся на группы по четыре; каждая группа используется для кодировки отдельной цифры десятичного числа:

Десятичное число	+	4	9	7	8
двоично-десятичное представление	0	0100	1001	0111	1000

Исторически первая стандартизованная *кодировка знаковой информации* – код ASCII. Он содержит 128 символов, каждому из которых присвоен свой код, 7-разрядное двоичное число. Первоначально код ASCII включал только латинский алфавит. Затем на его основе был создан код ДКОИ-7; в нем строчные латинские буквы были заменены на заглавные буквы кириллицы. В последствии код ASCII был расширен, введена кодировка ASCII-8, содержащая 256 символов. Дополнительные 128 кодов отведены первоначально для пред-

ставления псевдографики и специальных символов, но очень скоро их стали использовать для представления символов других языков, в том числе кириллицы (код ДКОИ-8). Современные операционные системы ЭВМ могут использовать несколько отличающиеся кодировки на базе ASCII-8, например, приложения ранних версий Windows – ANSI. В этой кодировке первая 128-знаковая страница одинакова и стандартизована, а вторая может заменяться, в зависимости от используемого языка (команды country, codepage операционных систем DOS, Windows 3.11, Windows-95). В последние годы фирмой MicroSoft в качестве нового стандарта вводится многостраничная кодировка символов (Unicode), в которой первая 256-знаковая страница совпадает с ANSI, а остальные содержат коды региональных символов, однако в настоящее время здесь возникают проблемы совместимости – приложения, ориентированные на использование только этой кодировки символов, вызывают появление ошибок при взаимодействии с другими, ориентированными на использование двухстраничной кодировки.

ОРГАНИЗАЦИЯ ПАМЯТИ

Информация хранится в различных устройствах компьютера (оперативной памяти, регистрах процессора, гибких и жестких дисках и т. д.) в виде двоичных чисел.

Основные типы памяти:

1. Оперативная память.

   1.1. Память для чтения и записи с произвольным доступом (RAM).

   1.2. Память только для чтения (ROM).

   1.3. Память с однократной записью (WORM).

2. Внешняя память.

2.1. Память на жестких дисках.

2.2. Память на гибких дисках.

2.3. Память на компакт-дисках.

2.4. Память на магнитооптических дисках.

2.5. Память на лентах.

Элементарная ячейка памяти, место для записи одного двоичного разряда (нуля или единицы) – бит. Биты объединяются в группы по восемь разрядов – байты. 1024 (2^{10}) байта – килобайт, 2^{20} – мегабайт, 2^{30} – гигабайт. В большинстве компьютеров (в том числе в PDP-11 и IBM PC) оперативная память организована в виде одномерного массива байтов – ячеек памяти. В них содержатся как данные, так и коды команд, выполняемых процессором. Каждой ячейке присвоен (двоичный) номер – *адрес*. Методы указания этих номеров, допускаемые данным компьютером, – *методы адресации*. Кроме простого способа – прямого указания номера требуемой ячейки (абсолютной адресации) – используются указание сдвига номера относительно некоторого заранее заданного (относительная адресация), а также другие методы вычисления требуемого адреса. Полный набор реализованных методов адресации определяется конструкцией ЭВМ и является одной из основных характеристик архитектуры компьютера. Общее количество возможных адресов оперативной (RAM) памяти – *адресное пространство* (не обязательно, что все оно используется в конкретном компьютере).

Первоначально байт являлся основной информационной единицей компьютеров, с которой процессор работал как с единым числом или командой. В настоящее время большинство процессоров (включая процессоры PDP-11 и IBM PC) имеют возможность одновременной работы с несколькими байтами памяти, как с единым числом. Группа байтов, которую процессор воспринимает как одно число (одну команду) – ма-

шинное *слово*. Зачастую предполагается, что длина слова равна двум байтам (16 битам), хотя современные IBM-совместимые компьютеры имеют возможность работы с 32- и 64-разрядными числами.

Кроме вышеупомянутых, используются также следующие единицы памяти: *параграфы* (участки по 16 байт, выровненные по началу памяти), *банки* (участки по 64 Кбайт, выровненные по началу памяти), *сегменты* (участки по 64 Кбайт, но начинающиеся с произвольного адреса), *страницы* (участки памяти, которыми компьютер может манипулировать как единым целым: например, перемещать из оперативной памяти на диск или наоборот, отображать на экране дисплея и т. п.).

КОМАНДЫ ПРОЦЕССОРА

Устройством, которое выполняет операции с числами, является процессор. Набор операций, которые может выполнять данный процессор, или его система команд, – одна из основных компонент архитектуры ЭВМ. Каждой команде ставится в соответствие двоичное число – ее код. Исполняемая программа – это упорядоченный набор кодов команд, который должен выполнить процессор. Обычно программы хранятся в устройствах внешней памяти (на дисках, лентах) и по мере необходимости, для их выполнения, перемещаются в оперативную память.

Процессор содержит также несколько специализированных ячеек памяти (*регистров процессора*), предназначенных для хранения служебной информации. Среди них всегда имеется один регистр – *программный счетчик* (PC, program counter) – в котором хранится адрес команды, выполняемой процессором в настоящий момент. По окончании выполнения текущей команды содержимое программного счетчика увеличивается, и процессор переходит к выполнению следующей команды, код которой находится в следующем слове оперативной памяти (разумеется, если содержимое программного счетчика не было изменено в результате выполнения этой команды).

Кроме программного изменения содержимого программного счетчика, существует еще одна возможность управления последовательностью выполнения операций – *прерывание*. Первоначально прерывания использовались для получения возможности приостановления работы процессора по команде внешнего устройства; в настоящее время они применяются и в других случаях, например, для обработки ошибок, возникающих при работе программ. Работа прерывания заключается в следующем. При поступлении в процессор *запроса прерывания* (interruption request, IRQ), от внешнего устройства или сгенерированного программно, выполнение основной программы прекращается. Процессор обращается к ячейкам памяти, поставленным в соответствие данному запросу (их адреса – *вектор прерывания*, эти адреса устанавливаются изготовителем процессора и являются одной из основных его характеристик), где хранится адрес размещения в оперативной памяти *программы обработки прерывания* и, в некоторых случаях, минимальный набор необходимых для ее выполнения данных. Адрес программы обработки прерывания переносится в программный счетчик, и начинается ее выполнение. На время выполнения программы обработки прерывания адрес команды основной программы, выполнение которой было прервано, размещается в векторе прерывания. По окончании выполнения программы выполнения прерывания восстанавливается исходное значение программного счетчика и вектора прерывания, и выполнение основной программы продолжается с того места, где она была прервана. Конструкция компьютеров предусматривает несколько прерываний (как минимум – по одному на каждое внешнее устройство, способное прерывать работу процессора) и, соответственно, несколько адресов прерываний. Для их размещения в оперативной памяти предусматривается *область векторов прерываний*, расположение которой определяется, в конечном счете, разработчиками компьютера. Установка в компьютер устройства, способного прерывать работу процессора, сопровождается (по крайней мере, должна сопровождаться) разме-

щением в оперативной памяти программы обработки соответствующего ему прерывания и размещением начального адреса этой программы в соответствующем векторе прерывания.

В принципе возможно написание программ процессора (как обычных, так и программ обработки прерываний), используя исключительно коды команд процессора (программирование в кодах). Однако чисто технически это неудобно, поэтому принято каждой команде ставить в соответствие некий буквенный код (например, MOVE – перенос числа в указанную ячейку памяти). Набор таких текстовых команд, для которых существует взаимно-однозначное соответствие с командами данного процессора, образует язык программирования *Ассемблер*. Очевидно, что этот язык индивидуален для каждого данного типа процессоров и позволяет выполнять любые операции, на которые данный процессор способен. Остальные *языки высокого уровня* (Fortran, Basic, Pascal, C и др.), относительно независимые от типа процессора, таким взаимно-однозначным соответствием не обладают; одна команда такого языка преобразуется перед выполнением в набор команд процессора (иногда довольно большой), причем такое преобразование может быть неполным (некоторые команды процессора могут быть невыполнимы в рамках данного языка) и неоднозначным (одна и та же команда языка может быть преобразована в различные наборы команд процессора). Существуют специализированные программы, осуществляющие преобразование команд языка программирования в коды процессора. Они делятся на интерпретаторы (осуществляют выполнение полученного кода сразу после выполнения перевода очередной команды) и трансляторы (производят перевод в коды сразу всей программы; этот код запоминается во внешней памяти и переносится в оперативную память для исполнения по мере необходимости).

Набор характеристик:

1. Длина машинного слова.

2. Система команд процессора.

3. Регистры процессора.

4. Векторы прерываний.

5. Методы адресации памяти.

характеризует архитектуру компьютера.

Компьютеры с одинаковой архитектурой *программно совместимы*, т. е. на них (в принципе) могут исполняться одни и те же программы. Поддержание принципа программной совместимости при разработке новых компьютеров удобно, так как обеспечивает преемственность программного обеспечения, но тормозит развитие новых элементов их архитектуры. Поэтому обычно при совершенствовании вычислительной техники используется принцип совместимости сверху вниз: сохранение всех элементов старой архитектуры и введение новых, расширяющих возможности компьютера. Это позволяет использовать и старое программное обеспечение в полном объеме, и новые возможности. Именно такая совместимость реализована в ряду машин типа PDP-11, а затем – в ряду IBM-совместимых компьютеров.

Особенности архитектуры IBM-совместимых компьютеров

Структура организации IBM-совместимых компьютеров показана на рис. 2.

Архитектура этих компьютеров предусматривает их взаимодействие с внешними устройствами несколькими способами:

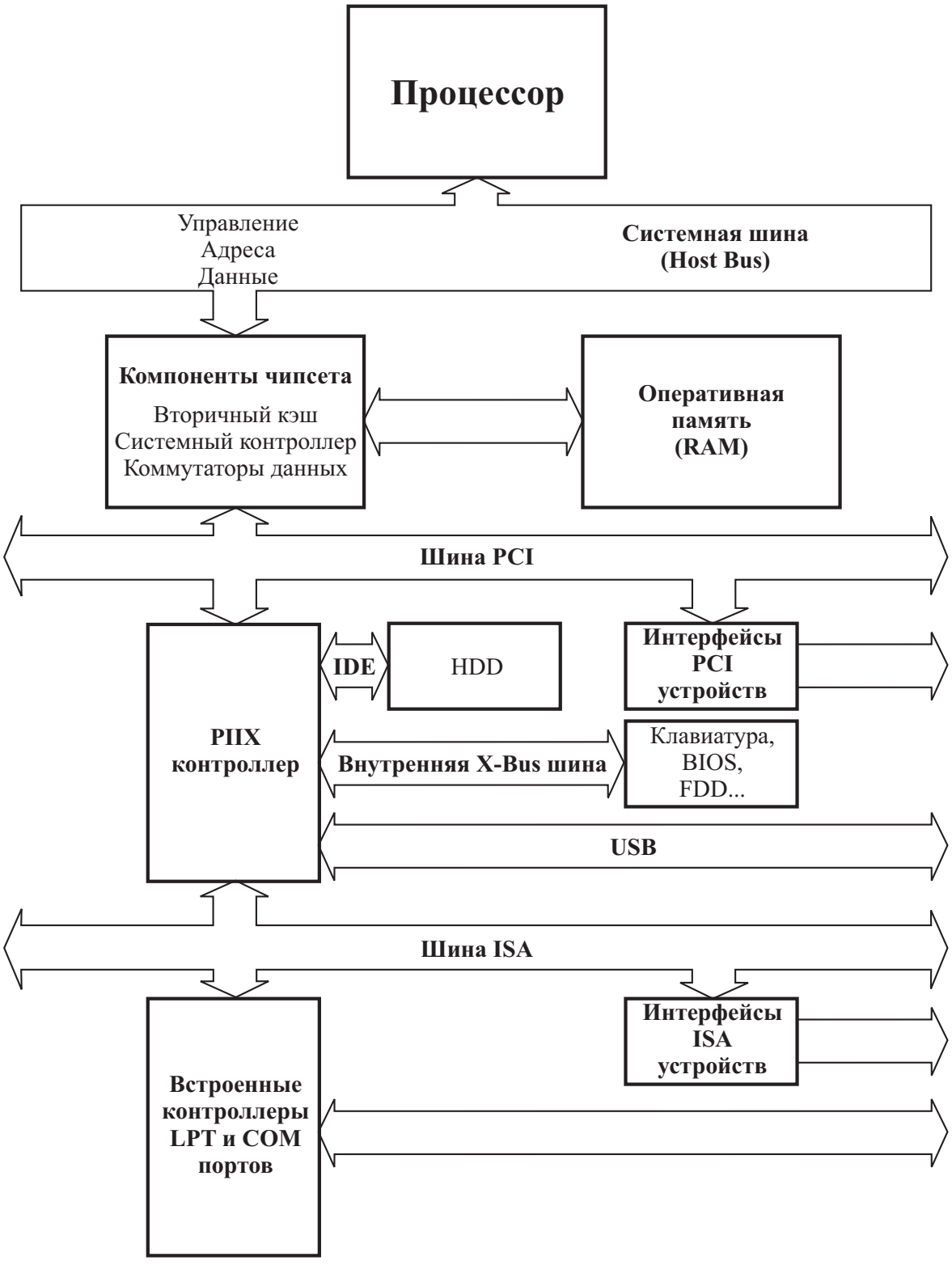


Рис. 2. Структура IBM-совместимых компьютеров.

1. Напрямую взаимодействуя со встроенными регистрами контроллеров интерфейсов.

2. Используя функции базовой системы ввода-вывода (BIOS).

3. Используя встроенные функции операционной системы.

Столь широкий набор вариантов существует благодаря сохранению принятых ранее (возможно, и не оптимальных) решений в последующих разработках.

Любой из этих компьютеров включает:

1. Процессор, совместимый с семейством x86 Intel.

2. Единую систему распределения адресов памяти.

3. Отделенное от памяти унифицированное пространство адресов ввода-вывода, с фиксированным положением и совместимым программным управлением обязательных портов.

4. Минимальный набор системных устройств и интерфейсов ввода-вывода.

5. Единую систему аппаратных прерываний.

6. Унифицированные шины расширения (ISA, PCI и др.), состав которых может варьироваться в зависимости от модели компьютера.

7. Базовую систему ввода-вывода (BIOS), выполняющую начальное тестирование компьютера, загрузку операционной системы и включающую набор функций, обслуживающих системные устройства ввода-вывода.

8. Систему прямого доступа к памяти, позволяющую производить обмен данными между внешними устройствами и оперативной памятью с минимальным участием процессора.

К *обязательным стандартизованным* средствам ввода-вывода относятся:

1. Счетчик.

2. Интерфейс клавиатуры и управления.

3. Канал управления звуком (динамиком).

4. Графический адаптер.

Основы организации этих средств не изменяются от модели к модели (хотя они могут дополняться новыми деталями, что хорошо известно на примере графического адаптера). К *системным* устройствам (то есть обязательно присутствующим в конфигурации компьютера и поддерживаемым BIOS) первоначально относились клавиатура, видеоадаптер с монитором, параллельный (первоначально – один) и последовательные (первоначально – два) порты. Со временем в их число попала дисковая система (сначала – гибкие, затем – жесткие диски, позже – CD-ROM, в некоторых случаях – другие виды накопителей). В некоторых моделях компьютеров (материнских плат) в число системных включают также мышь и цифровой аудиоканал.

ОРГАНИЗАЦИЯ ОПЕРАТИВНОЙ ПАМЯТИ

Первые процессоры семейства x86 – 8086/88 – использовали 16-разрядную адресацию памяти. Шина, используемая для указания адреса памяти, была 20-разрядной, что обеспечивало адресное пространство в 1 Мб, а для вычисления реального адреса на шине (линейного адреса) использовалось два 16-разрядных числа (сегмент и смещение; $Addr = Seg \times 10000_2 + Offset$). В дальнейшем шина адреса была

Рис. 3. Распределение памяти
IBM-совместимого компьютера.

расширена (до 24 бит у 286 и до 32 – начиная с 386 процессоров, что соответствует 4 Гб пространству памяти), и появилась возможность использования 32-разрядной адресации памяти, однако в целях совместимости поддержка изначального метода адресации была сохранена – так называемый реальный режим (*Real mode*) работы процессора. В отличие от него режим работы с использованием 32-разрядных адресов называется защищенным (*Protected mode*).

Распределение памяти показано на рис. 3.

Область 00000_{16}–$9FFFF_{16}$, 640 кБ – *базовая, или стандартная, память*, доступная программам в реальном режиме.

Она, в свою очередь, разделяется следующим образом:

00000_{16}–$003FF_{16}$ – 256 векторов прерываний, по два 16-разрядных слова каждый.

00400_{16}–$004FF_{16}$ – область, используемая BIOS.

00500_{16}–$00xxx_{16}$ – область, используемая операционной системой (верхняя граница не фиксирована и определяется операционной системой).

$00xxx_{16}$–$9FFFF_{16}$ – область, предоставленная пользователю.

Область $A0000_{16}$– $FFFFF_{16}$, 384 кБ – *верхняя память*, зарезервированная для системных нужд. В ней размещаются области буферной памяти системных адаптеров (например, видеопамять), и BIOS с расширениями, предоставленными операционной системой.

Верхняя память распределяется следующим образом.

$A0000_{16}$–$BFFFF_{16}$ – 128 кБ видеопамяти.

$C0000_{16}$–$DFFFF_{16}$ – 128 кБ, зарезервированных для других системных адаптеров, использующих собственные модули BIOS.

$E0000_{16}$–$EFFFF_{16}$ – свободные 64 кБ, иногда используемые системным BIOS.

$F0000_{16}$–$FFFFF_{16}$ – 64 кБ, в которых размещена базовая система ввода-вывода BIOS. Физически эти ячейки памяти находятся в ROM на системной плате.

Как правило, эта область памяти используется имеющимися адаптерами и BIOS не полностью, и операционная система предоставляет пользователю средства использования свободной части верхней памяти. Тем самым увеличивается размер памяти в первом мегабайте, доступном пользователю в реальном режиме работы процессора – но отсюда же зачастую возникают конфликты использования одной и той же области памяти несколькими устройствами или программами.

Область свыше 100000_{16} – дополнительная (расширенная) память, доступная только в защищенном режиме. Ее нижнюю часть, до $10FFEF_{16}$ (HMA), некоторые операционные системы (например, DOS старших версий) используют для размещения своего ядра с целью экономии пространства базовой памяти.

ОБРАБОТКА ПРЕРЫВАНИЙ

Аппаратные прерывания обеспечивают реакцию компьютера на события, происходящие независимо от выполняемой им программы, то есть являются необходимой частью программ управления и контроля за внешними процессами. Процессоры x86 поддерживают до 256 прерываний и программ их обработки и различают четыре вида прерываний:

1. Внутренние прерывания процессора.

2. Немаскируемые внешние прерывания.

3. Маскируемые внешние прерывания.

4. Программно-вызываемые прерывания.

Последние не являются прерываниями в точном смысле этого слова, а представляют собой использование механизма прерываний для вызова подпрограмм – не по их адресу, а по номеру приписанного им прерывания.

При поступлении сигнала прерывания процессор завершает выполнение текущей операции, сохраняет в стеке информацию о своем текущем состоянии, в том числе – адрес следующей операции, отменяет разрешение прерываний и вызывает *программу обработки прерывания* – ее адрес (сегмент-смещение, два 16-разрядных слова) хранится в векторе поступившего прерывания в нижних адресах оперативной памяти. Программа обработки прерывания выполняет указанные в ней действия, и процессор, используя хранящиеся в стеке данные, возобновляет выполнение прерванной программы. По умолчанию программа обработки прерывания не может быть вложенной; однако это ограничение легко можно обойти, возобновив разрешение прерываний в начале программы обработки прерывания. Рекомендуется также максимально сокращать время работы программ обработки прерываний – системное время компьютера также определяется по прерываниям от внутреннего таймера, и длительные процедуры обработки других прерываний при установленном запрете приводят к потере системного времени.

Внутренние прерывания процессора возникают при возникновении особых условий выполнения команд. Под их векторы первоначально были зарезервированы первые 32 из всей таблицы прерываний, однако в настоящее время это выполняется не всегда.

Немаскируемые прерывания обрабатываются процессором независимо от того, установлено или нет разрешение прерываний. Их вызывают системы контроля состояния памяти, управления энергопотреблением, другие внутренние системы ЭВМ.

Маскируемые прерывания – именно тот механизм, который используется для управления большинством внешних устройств. В первых XT компьютерах использовалось восемь маскируемых прерываний с 8-разрядными векторами прерывания. В более поздних AT машинах был установлен дополнительный ведомый контроллер прерываний, который перехватывал прерывание IRQ2 и использовал его для обработки прерываний с IRQ8 по IRQ15. Эта конфигурация сохраняется и до сих пор. Принятые назначения этих прерываний приведены в табл. 1.

Назначения прерываний должны поддерживаться с обеих сторон: адаптер, использующий прерывание, должен быть сконфигурирован на использование установленного номера прерываний (в адаптерах XT использовалась жесткая аппаратная настройка, в современных системах это конфигурирование может устанавливаться джамперами на платах, с по-

Таблица 1. Аппаратные прерывания IBM PC

Номер прерывания	Вектор прерывания	Назначение в IBM XT	Назначение в IBM AT
IRQ0	8_{16}	Таймер	Таймер
IRQ1	9_{16}	Клавиатура	Клавиатура
IRQ2	A_{16}	Резерв	2-й контроллер прерываний
IRQ3	B_{16}	COM1	COM2, COM4
IRQ4	C_{16}	COM2	COM1, COM3
IRQ5	D_{16}	HDD	LPT2, звук, резерв
IRQ6	E_{16}	FDD	FDD
IRQ7	F_{16}	LPT1	LPT1
IRQ8	70_{16}	Не используется	Часы реального времени
IRQ9	71_{16}	Не используется	Резерв
IRQ10	72_{16}	Не используется	Резерв
IRQ11	73_{16}	Не используется	Резерв
IRQ12	74_{16}	Не используется	Резерв
IRQ13	75_{16}	Не используется	Сопроцессор
IRQ14	76_{16}	Не используется	HDD
IRQ15	77_{16}	Не используется	Резерв

мощью специальных программных утилит, либо автоматически); программное обеспечение, управляющее этим адаптером, должно использовать именно это прерывание. Системная шина ISA ориентирована на статическое распределение прерываний и использование одного и того же прерывания несколькими устройствами, подключенными к ней, приводит к ошибкам в работе ЭВМ. Более современная шина PCI имеет четыре независимых линии запросов прерывания, что дает возможность их разделяемого использования; это осуществляется при настройке опций BIOS и системой Plug&Play.

ОРГАНИЗАЦИЯ ВВОДА-ВЫВОДА

Важной особенностью IBM-совместимых компьютеров является раздельная работа с оперативной памятью, процессором и интерфейсами внешних устройств. Как видно из рис. 2, процессор связан с памятью и интерфейсами несколькими шинами. Любая из них имеет выделенные линии передачи сигналов управления, адреса и данных. Шины могут иметь различную разрядность как данных, так и адреса, и обращение к ним осуществляется различными командами процессора. По выставленной команде элементы чипсета идентифицируют нужную шину, по установленному адресу идентифицируется интерфейсная плата, к которой адресована команда, данные пересылаются или считываются из специализированных ячеек памяти – *регистров* – установленных на интерфейсных платах. Таким образом, задачами интерфейсной платы, установленной на шине, являются преобразование внешних данных в формат, определяемый стандартом данной шины, и прием/передача этих данных и команд в соответствии с принятым для данной шины стандартом.

Программы могут обмениваться данными с внешними устройствами, обрабатывая прерывание, используя непосредственное управление шиной либо используя команды обращения к стандартным портам ввода/вывода, если внешнее устройство подключено через такой порт.

Программно-управляемый обмен подразумевает следующую последовательность операций:

1. Чтение регистра состояния устройства для анализа его готовности.

2. Зацикливание предыдущего шага – ожидание готовности.

3. Собственно обмен данными.

Инициатором обмена может быть как основная программа, так и периферийное устройство. Программа ожидает некоторое событие в устройстве (например, установление бита готовности в регистре состояния), периодически считывая содержимое этого регистра. Этот способ называется *обменом по опросу готовности.* При этом время реакции может быть сведено до долей микросекунды (при условии, что программа занимается этим опросом монопольно). Однако при этом процессор во время ожидания оказывается занят бесполезной работой. Другой подход – использование аппаратного прерывания, вырабатываемого устройством при событии, требующем вмешательства управляющей программы. Время реакции на событие в этом случае сильно зависит от режима работы процессора в момент поступления прерывания (так как требуется сохранение информации о работе процессора на момент вызова прерывания), и может составлять от долей микросекунд до десятков и сотен миллисекунд (при работе с виртуальной памятью).

Используется также комплексное решение – опрос готовности устройств не на каждом шагу основной программы, а в моменты времени, определяемые некоторым периодическим процессом – например, прерываниями системного таймера. Периодически опрашиваются все подключенные устройства, и те из них, для которых установлена готовность, обслуживаются. Классическим примером такой работы является утилита фоновой печати PRINT.

Специально для подключения различных адаптеров внешних устройств предназначены *шины расширения.* В компьютерах IBM PC шины расширения начали свое развитие с шины ISA, возникшей в ее первом варианте еще на PC XT. Открытость организации этой шины обеспечила появление широкого спектра плат-адаптеров, позволивших применять эти компьютеры для решения широкого класса задач управления и автоматизации. С появлением AT-286 шина ISA была расширена, что увеличило число подключаемых адаптеров и скорость работы с ними (до 8 Мб/с). Затем, как отклик на потребности в высокопроизводительном обмене данными на серверах возникла еще одна модификация – EISA (33 Мб/с). С появлением процессора 486 возникла потребность в высокопроизводительной работе с графическим адаптером, что привело к появлению специализированной шины VLB (132 Мб/с). Однако узкая специализация и принципиальная привязка к архитектуре процессора 486 привели к быстрому исчезновению этой шины и развитию новой – PCI (также 132 Мб/с). В настоящее время эта шина стала стандартом, используемым не только в IBM PC, но и ряде других семейств ЭВМ и управляющих устройств. Ее современным развитием стал новый специализированный графический порт AGP (532 Мб/с).

Шины ISA, EISA

ISA (Industry Standard Architecture) – первая ставшая стандартом шина расширения IBM PC. В компьютерах XT использовалась ее ранняя версия (ISA-8), с разрядностью данных 8 бит и адреса – 20 бит. В компьютерах AT она была расширена до 16 бит данных и 24 – адреса. Типичный вид интерфейсной платы для этой шины показан на рис. 4. Платы подключаются к шине с помощью двух щелевых разъемов. ISA-8 использует только первый из них (группы контактов A, B), в ISA-16 используются дополнительно группы C, D, содержащие по 18 контактов. Количество пар разъемов колеблется от 2–3 до 8–10 для различных материнских плат. Кроме того, на материнской плате имеются интегрированные уст-

Рис. 4. Вид интерфейсной платы ISA.

ройства (например, системный таймер), также использующие эту шину для обмена данными с процессором.

Адреса регистров управления и передачи данных для стандартных устройств находятся в базовой системе ввода/вывода BIOS. Она хранится в ПЗУ компьютера и при загрузке компьютера копируется в нижнюю часть оперативной памяти, сразу после векторов прерываний. Традиционное распределение этих адресов приведено в табл. 2 (хотя для некоторых моделей компьютеров эти адреса могут отличаться).

Шина поддерживает обмен процессора и интерфейсов внешних устройств одно- и двухбайтными данными. Адресное пространство устройств ввода-вывода составляет 64 кб (758 адресов 8-битных устройств), но практически все существующие интерфейсные платы этого стандарта ограничиваются первым килобайтом этого пространства.

Как уже обсуждалось, шина ISA-8 предоставляет внешним устройствам до 6 линий запросов прерываний, для ISA-16 их число увеличено до 11. Шина не имеет механизма автоконфигурирования, и задачей пользователя является бесконфликтное распределение ее ресурсов между интерфейсами.

Таблица 2. Стандартные адреса регистров
устройств ввода-вывода шины ISA

Адреса	Назначение
00016–01F16	Контроллер прямого доступа к памяти № 1
02016–03F16	Контроллер прерываний № 1
04016–05F16	Программируемый таймер
06016–06F16	Контроллер клавиатуры
07016–07F16	Таймер реального времени
08016–09F16	Контроллер страниц памяти
0C016–0DF16	Контроллер прерываний № 2
0F016–0FF16	Сопроцессор
17016–17716	Жесткий диск HDD2
1F016–1F716	Жесткий диск HDD1
20016–20716	Джойстик
27816–27F16	Параллельный порт LPT2
2C016–2DF16	Видеоадаптер EGA № 2
2F816–2FF16	Последовательный порт COM2
30016–31F16	Резерв
32016–32F16	Жесткий диск для XT
36016–36F16	Резерв
37016–37716	Гибкий диск FDD2
37816–37F16	Параллельный порт LPT1
38016–38F16	Контроллер обмена № 2
3A016–3AF16	Контроллер обмена № 1
3B016–3DF16	Видеоадаптер VGA
3B016–3BF16	Монохромный видеоадаптер
3C016–3CF16	Видеоадаптер EGA № 1
3D016–3DF16	Видеоадаптер CGA
3F016–3F716	Гибкий диск FDD1
3F816–3FF16	Последовательный порт COM1

При этом подразумевается:

1. Каждая интерфейсная плата при операциях чтения должна выдавать информацию только по своим адресам. Адресные пространства интерфейсов не должны пересекаться.

2. В пассивном состоянии плата должна удерживать линию запроса прерывания на низком уровне и переводить на высокий уровень только для активации запроса. Одной линией запроса может пользоваться только одно устройство.

Назначение контактов плат ISA показано в табл. 3.

Таблица 3. Назначение контактов интерфейсных плат ISA

Контакт	Обозначение	Назначение
A1	I/O CH CK	Проверка канала ввода-вывода. Вырабатывается при ошибке, вызывает прерывание.
A2–A9	SD7-SD0	Разряды данных в 8-разрядные регистры интерфейсов либо в младший байт 16-разрядных.
A10	I/O CH RDY	Готовность принять информацию. Снимается интерфейсной платой, если она не успела закончить предыдущую операцию.
A11	AEN	Выставляется процессором при работе в режиме прямого доступа к памяти. При этом все не занятые устройства должны прекратить работу с шиной.
A12-A31	SA19-SA0	Разряды устройств ввода-вывода (практически все устройства используют только SA0–SA9).
B1	GND	Заземление
B2	RESET DRV	Сигнал сброса в начальное состояние. Вырабатывается при включении питания, нажатии кнопки RESET.
B3	+5 В	Питание
B4	IRQ9	Запрос прерывания № 9
B5	–5 В	Питание
B6	DRQ2	Запрос № 2 устройства на прямой доступ к памяти.
B7	–12 В	Питание
B8	0WS	Информирует процессор о необходимости прямого обмена данными. Используется редко.
B9	+12 В	Питание
B10	GND	Заземление
B11	SMEMW	Синхронизация записи данных в память.
B12	SMEMR	Синхронизация чтения данных из памяти.
B13	IOW	Синхронизация записи данных в устройство
B14	IOR	Синхронизация чтения данных из устройства

Контакт	Обозначение	Назначение
B15	DACK3	Разрешение прямого доступа к памяти для устройства № 3.
B16	DRQ3	Запрос № 3 устройства на прямой доступ к памяти.
B17	DACK1	Разрешение прямого доступа к памяти для устройства № 1.
B18	DRQ1	Запрос № 1 устройства на прямой доступ к памяти.
B19	REFRESH	Сигнал обмена данными между процессором и памятью. Процессор занят.
B20	SYSCLK	Сигнал тактового генератора (обычно 8 мГц).
B21–B25	IRQ7–IRQ3	Запросы прерываний.
B26	DACK2	Разрешение прямого доступа к памяти для устройства № 3.
B27	T/C	Окончание работы в режиме прямого доступа к памяти.
B28	BALE	Стробирование разрядов адреса.
B29	+ 5 В	Питание
B30	OSC	Импульсы кварцевого генератора с частотой 14.31818 МГц. Синхронизация работы интерфейсов.
B31	GND	Заземление
C1	SBHE	Тип данных (8- или 16-разрядные).
C2–C8	LA23–LA17	Используются при работе с доп. памятью. При вводе/выводе устанавливаются в нуль.
C9	MEMR	Синхронизация записи данных в память (дополнительные страницы).
C10	MEMW	Синхронизация чтения данных из памяти (дополнительные страницы).
C11–C18	SD8–SD15	Разряды данных в старший байт 16-разрядных регистров интерфейсов.
D1 D2	MEM CS16 I/O CS16	Сообщение, что память использует 16-разрядный обмен. Сообщение устройства, что оно использует 16-разрядный обмен (по умолчанию – 8-разрядный).
D3–D7	IRQ10–IRQ14	Запросы прерываний
D8	DACK0	Разрешение прямого доступа к памяти для устройства № 0.
D9	DRQ0	Запрос № 0 устройства на прямой доступ к памяти.
D10	DACK5	Разрешение прямого доступа к памяти для устройства № 5.
D11	DRQ5	Запрос № 5 устройства на прямой доступ к памяти.
D12	DACK6	Разрешение прямого доступа к памяти для устройства № 6.
D13	DRQ6	Запрос № 6 устройства на прямой доступ к памяти.
D14	DACK7	Разрешение прямого доступа к памяти для устройства № 7.
D15	DRQ7	Запрос № 7 устройства на прямой доступ к памяти.
D16	+5 В	Питание
D17	MASTER	Сообщение устройства о готовности работы в режиме ДМА (после DRQ и DACK).
D18	GND	Заземление

EISA (Extended ISA) – стандартизованное расширение шины ISA. Геометрические размеры плат этих интерфейсов совпадают с ISA, а четыре дополнительные группы контактов расположены в промежутках и выше контактов ISA, так что плату ISA можно вставить в разъем EISA – она просто не будет касаться дополнительных контактных площадок. Дополнительные контакты позволили увеличить разрядность данных до 32 бит, за счет чего увеличилась скорость обмена информацией; рабочее адресное пространство при этом не изменилось. Допускается разделяемое использование запросов прерываний, и предусмотрена возможность программно-настраиваемых интерфейсов, для чего на системной плате дополнительно устанавливается энергонезависимая память конфигурации (NVRAM), где хранится информация о конфигурации каждой из установленных интерфейсных плат. В отличие от режима Plug&Play не предусмотрена динамическая перестройка конфигурации – настройка интерфейсов производится специальной утилитой ECU (EISA configuration utility), по окончании работы которой производится перезагрузка компьютера. В целом, это дорогая архитектура, оправдывающая себя в системах с объемным обменом данными (типа файл-серверов).

В настоящее время выпускается достаточно большой ассортимент интерфейсных плат ISA, ориентированных на задачи передачи данных в ЭВМ и управления. Как правило, на них монтируется несколько однотипных преобразователей сигнала (АЦП, ЦАП, преобразователей кодов и т. п.). Кроме того, выпускаются *карты-прототипы* (*Prototype Card*), представляющие собой печатные платы с крепежной скобой, на которых размещены обязательные интерфейсные цепи (дешифратор адреса, регистр данных и пр.) и оставлено место для размещения макетного варианта интерфейсного устройства, что удобно для отладки и монтажа единичных экземпляров интерфейсов собственной разработки.

Локальная шина VLB (VESA)

Шина была предложена Video Electronics Standards Association для организации обмена между процессором и мониторами высокого разрешения. Были также разработаны контроллеры дисковых накопителей и сетевые карты, выполненные в этом стандарте. Шина представляет собой сигналы процессора 486, выведенные на дополнительный щелевой разъем материнской платы. Позволяет организовать обмен 32-разрядными данными, осуществлять их 32-разрядную адресацию. Максимальная скорость передачи данных 132 Мбайт/с, тактовая частота 33–66 МГц. Разъем VLB имеет 112 контактов, на материнской плате их может быть от 1 до 3. Шина была сильно увязана на архитектуру 486 процессора, и поэтому в настоящее время материнские платы с VLB шиной практически не выпускаются.

Локальная шина PCI

Локальная шина PCI (Peripheral Component Interconnect), появившаяся впервые на 486 машинах, в настоящее время стала мостом между системной шиной процессора и медленной «классической» шиной ISA. Для действующего в настоящее время стандарта PCI при частоте 20–33 МГц теоретическая максимальная скорость обмена по ней составляет 132 (264) Мбайт/с для 32 или 64-разрядных данных. Допускается также использование частоты 66 МГц, если все установленные на шине устройства поддерживают эту частоту.

Слоты (разъемы) для подключения адаптеров к этой шине расположены на материнской плате несколько дальше от задней панели компьютера. Соединительные контакты на интерфейсных платах расположены на них с двух сторон, образуя группы A и B. Существуют две версии шины, отличающиеся напряжениями питания (5 В и 3.3 В); На платах разных версий имеются прорези на месте 12–13 или 50–51 контактов, в соответствующих местах слотов – ключи; это не позволяет установить плату в слот с неверным напряжением питания. Имеются также универсальные слоты и платы, под-

держивающие оба варианта питания. 32-разрядные платы имеют 62 пары контактов, 64-разрядные – 94. Наименования сигналов шины приведены в табл. 4, назначение выводов универсального разъема – в табл. 5, 4-разрядные коды команд – в табл. 6.

Таблица 4. Сигналы шины PCI

Сигнал	Назначение
AD[31:0]	Address/Data – 32 линии адреса/данных. Адрес передается в начале транзакции, в последующих тактах передаются данные.
C/BE[3:0]#	Command/Byte Enable – команда/разрешение обращения к байтам. Команда, определяющая тип очередного цикла шины (чтение/запись памяти, ввод/вывод или конфигурационное чтение-запись, подтверждение прерывания и другие), задается четырехбитным кодом в фазе адреса (см. табл. 6).
FRAME#	Кадр. Введением сигнала отмечается начало транзакции (фаза адреса), снятие сигнала указывает на то, что последующий цикл передачи данных является последним в транзакции.
DEVSEL#	Device Select – устройство выбрано (ответ устройства-приемника на адресованный к нему запрос)
IRDY#	Initiator Ready – готовность устройства-передатчика к обмену данными
TRDY#	Target Ready – готовность устройства-приемника к обмену данными
STOP#	Запрос устройства-приемника к передатчику на остановку транзакции
LOCK#	Используется для установки, обслуживания и освобождения захвата ресурса на шине PCI
REQ[3:0]#	Request – запрос от PCI-мастера (передатчика) на захват шины (разрешен для слотов 3:0)
GNT[3:0]#	Grant – предоставление передатчику управления шиной
PAR	Parity - общий бит паритета для линий AD[31:0] и C/BE[3:0]

Сигнал	Назначение
PERR#	ParityError – сигнал об ошибке паритета (от устройства, ее обнаружившего)
RST#	Reset – сброс всех регистров в начальное состояние
IDSEL#	Initialization Device Select – выбор устройства в процессах считывания и записи их конфигурации
SERR	System Error – системная ошибка, активизируется любым устройством PCI и вызывает NMI
REQ64#	Request 64 bit – запрос на 64-битный обмен
ACK64#	Подтверждение 64-битного обмена
INTRA# INTRB# INTRC# INTRD#	Interrupt A, B, C, D – линии запросов прерывания, циклически сдвигаются в слотах и направляются на доступные линии IRQ. Допускается разделяемое использование линий
CLK	Clock – тактовая частота шины, должна лежать в пределах 20–33 МГц, начиная с PCI 2.1 допустима до 66,6 МГц
M66EN	66MHz_Enable – разрешение частоты синхронизации до 66 МГц, если все абоненты ее допускают (введено начиная с PCI 2.1)
SDONE	Snoop Done – сигнал завершенности цикла слежения для текущей транзакции. Необязательный сигнал, используется только устройствами с кэшируемой памятью
SBO#	Snoop Backoff – попадание текущего обращения к памяти абонента шины в модифицированную строку кэша. Необязательный сигнал, используется только устройствами с кэшируемой памятью
TCK	Test Clock – синхронизация тестового интерфейса JTAG
TDI	Test Data Input – входные данные тестового интерфейса JTAG
TOO	Test Data Output – выходные данные тестового интерфейса JTAG
TMS	Test Mode Select – выбор режима для тестового интерфейса JTAG
TRST	Test Logic Reset – сброс тестовой логики

Таблица 5. Назначение контактов разъема шины PCI

Ряд B	№	Ряд A	Ряд B	№	Ряд A
–12 B	1	TRST#	GND/M66EN[1]	49	AD 9
TCK	2	+12 B	GND/Ключ 5 B	50	GND/Ключ 5 B
GND	3	TMS	GND/Ключ 5 B	51	GND/Ключ 5 B
TDO	4	TDI	AD 8	52	C/BEO#
+5 B	5	+5 B	AD 7	53	+3.3 B
+5 B	6	INTRA#	+3.3 B	54	AD 6
INTRB#	7	INTRC#	AD 5	55	AD 4
INTRD#	8	+5 B	AD 3	56	GND
PRSNT 1#	9	Reserved	GND	57	AD 2
Reserved	10	+VI/O	AD 1	58	AD 0
PRSNT 2#	11	Reserved	+VI/O	59	+VI/O
GND/Ключ 3.3 B	12	GND/Ключ 3.3 B	ACK64#	60	REQ64#
GND/Ключ 3.3 B	13	GND/Ключ 3.3 B	+5 B	61	+5 B
Reserved	14	Reserved	+5 B	62	+5 B
GND	15	RST#	Конец 32-битного разъема		
Clock	16	+VI/0	Reserved	63	GND
GND	17	GNT#	GND	64	C/BE7#
REQ#	18	GND	C/BE6#	65	C/BE5#
+V I/O	19	Reserved	C/BE4#	66	+V I/O
AD 31	20	AD 30	GND	67	PAR64
AD 29	21	+3.3 B	AD 63	68	AD 62
GND	22	AD 28	AD 61	69	GND
AD 27	23	AD 26	+VI/O	70	AD 60
AD 25	24	GND	AD 59	71	AD 58
+3.3 B	25	AD 24	AD 57	72	GND
C/BE3#	26	IDSEL#	GND	73	AD 56
AD 23	27	+3.3 B	AD 55	74	AD 54
GND	28	AD 22	AD 53	75	+V I/O
AD 21	29	AD 20	GND	76	AD 52
AD 19	30	GND	AD 51	77	AD 50
+3.3 B	31	AD 18	AD 49	78	GND
AD 17	32	AD 16	+V I/O	79	AD 48
C/BE2#	33	+3.3 B	AD 47	80	AD 46

Ряд B	№	Ряд A	Ряд B	№	Ряд A
GND	34	FRAME#	AD 45	81	GND
IRDY#	35	GND	GND	82	AD 44
+3.3 B	36	TRDY#	AD 43	83	AD 42
DEVSEL#	37	GND	AD 41	84	+VI/O
GND	38	STOP#	GND	85	AD 40
LOCK#	39	+3.3 B	AD 39	86	AD 38
PERR#	40	SDONE#	AD 37	87	GND
+3.3 B	41	SBOFF#	+V I/O	88	AD 36
SERR#	42	GND	AD 35	89	AD 34
+3.3 B	43	PAR	AD 33	90	GND
C/BE1#	44	AD 15	GND	91	AD 32
AD 14	45	+3.3 B	Reserved	92	Reserved
GND	46	AD 13	Reserved	93	GND
AD 12	47	AD 11	GND	94	Reserved
AD 10	48	GND	Конец 64-битного разъема		

[1] Сигнал M66EN определен только начиная с модификации PCI 2.1.

Таблица 6. Коды команд шины PCI

C/BE[3:0]	Тип команды
0000	Interrupt Acknowledge — подтверждение прерывания
0001	Special Cycle — специальный цикл
0010	I/O Read — чтение из устройства на шине
0011	I/O Write — запись в устройство
0100	Резерв
0101	Резерв
0110	Memory Read — чтение памяти
0111	Memory Write — запись в память
1000	Резерв
1001	Резерв
1010	Configuration Read — конфигурационное считывание
1011	Configuration Write — конфигурационная запись
1100	Multiple Memory Read — множественное чтение памяти
1101	Dual Address Cycle — двухадресный цикл

С целью ускорения обмена информацией по шине все процессы обмена данными (транзакции) предполагаются пакетными. На шине выставляется сигнал FRAME#, по 32 линиях AD выставляется адрес приемника, а на четырех линиях C/BE –команда (см. табл. 6). Указанное устройство-приемник команды отзывается сигналом DEVSEL#, передатчик подтверждает готовность данных сигналом IRDY#, приемник подтверждает готовность к приему сигналом TRDY#. При наличии на шине двух последних сигналов начинается передача данных – по тем же линиям AD; этим обеспечивается синхронная работа, если скорости приема и передачи данных у приемника и передатчика не совпадают. Общий объем передаваемых данных заранее не определен; обмен прекращается, когда передатчик снимет сигнал FRAME#. Обмен может быть прекращен также из-за неготовности приемника (нет сигнала DEVSEL#), либо при поступлении с приемника сигнала STOP#.

С целью повышения надежности обмена на шине введены линии контроля четности (число бит в адресах и данных должно быть четным, в противном случае на шине выставляется сигнал PERR#).

Регистры PCI-устройств могут быть 8- или 16-битными. Для их адресации, в принципе, можно использовать все 32 бита линий AD, но процессоры x86 используют только младшие 16 (шина используется и на других типах процессоров). Конфигурирование устройств (выбор адресов регистров, запросов прерываний и т. д.) поддерживается средствами BIOS и ориентировано на технологию Plug&Play. Согласно стандарту, каждое PCI-устройство может иметь до 256 8-разрядных регистра, доступ к которым осуществляется по специальным командам конфигурирования. После перезагрузки системы PCI-устройства не отвечают на обращения к их регистрам данных и доступны в этот момент только для операций конфигурирования. Во время загрузки системы происходит опрос устройств и настройка шины, и только по-

сле этого возможно обращение к устройствам для чтения/записи данных.

Для запросов прерываний используются четыре линии INTR A, B, C, D. Подключение линий к определенному слоту может регулироваться программно и первоначально также устанавливается в процессе конфигурации шины. Прерывания, занятые шиной PCI, становятся недоступными для ISA устройств.

Организация вызова прерываний позволяет установить на одной шине PCI не более четырех устройств; для ее расширения либо для соединения с другими шинами материнской платы (шины процессора, ISA) используются *мосты шины PCI*, которые программируются производителями таким образом, чтобы обеспечить однозначную адресацию подключенных к ним устройств. Не распознанные шиной PCI запросы по умолчанию передаются на шину ISA.

Шина PCI является второй по популярности применения после ISA; для нее также выпускаются как специализированные интерфейсные карты, так и карты-прототипы, ориентированные на самостоятельную разработку периферийных интерфейсов. Разумеется, подобная разработка в данном случае представляет более сложную задачу, что связано как с более сложным протоколом обмена данными, так и более высокими частотами работы.

Шина PCMCIA (PC Card)

Шина стандартизована ассоциацией Personal Computer Memory Card International Association для устройств расширения блокнотных компьютеров. Шина позволяет адресовать до 4080 слотов внешних устройств, разрядность данных 16 бит, частота 33 МГц, ориентирована на программное конфигурирование адаптеров. Большинство выпускаемых адаптеров для этой шины используют технологию Plug&Play и предусматривают возможность «горячего» (без выключения машины) подключения. С этой целью контакты линий питания имеют большую длину, чем сигнальные, а контакты Card De-

тест короче остальных. Тем самым при подключении адаптера сначала на него подается питание, затем подаются управляющие сигналы, и только потом происходит распознавание платы компьютером. Отключение происходит в обратном порядке.

Все устройства PC Card имеют минимальное энергопотребление. В настоящее время делаются попытки ввести эту шину в качестве дополнительной и в настольных PC.

Шина SCSI

SCSI (Small Computer System Interface) – стандартизованный интерфейс системного уровня, предназначенная для подключения внутренних и внешних периферийных устройств, требующих высокопроизводительного обмена данными. В отличие от рассмотренных выше жестких шин расширения, эта шина реализована в виде кабельного шлейфа, который допускает соединение «один в один» (последовательно) до восьми устройств. Одно устройство, *хост-адаптер*, связывает шину с системной шиной компьютера, семь других могут устанавливаться пользователем.

Каждое установленное на шине устройство имеет свой идентификатор номера (SCSI ID), значение которого передается по восьми линиям шины (отсюда и ограничение на число устройств). Шина имеет большое количество вариантов – механических, электрических и логических, отличающихся как разрядностью данных, так и скоростью их передачи:

SCSI-1 – 8 бит, 18 команд, 5 МГц.

SCSI-2 – 8 бит, добавлены специальные команды управления CD-ROM, 5 МГц. Устройства могут выполнять наборы (цепи) до 256 команд и обмениваться данными без участия центрального процессора.

Fast SCSI-2 – частота повышена до 10 МГц.

Wide SCSI-2 – 16- и 32-битные данные.

Ultra SCSI – частота повышена до 20 МГц.

SCSI-3 – разрабатываемый сейчас вариант шины, предусматривающий увеличение числа устройств на шине, под-

держку Plug&Play, увеличение частоты до 100 МГц, возможность использования последовательного оптоволоконного кабеля вместо шлейфа.

Ввиду большого разнообразия вариантов организации шины и возможности как внутренней, так и внешней установки устройств, существует большое разнообразие кабелей и разъемов для их подключения. Из-за жестких требований к адаптерам и, как следствие, их высокой стоимости, для целей управления и сбора информации от нестандартных устройств (экспериментальных установок) эта шина используется крайне редко.

Параллельные порты

В первых моделях компьютеров семейства IBM PC не была заложена возможность для пользователя изменять их конфигурацию; единственным путем подключения дополнительных внешних устройств были *параллельные* и *последовательные порты.*

Основным назначением параллельных портов (стандарт Centronics; обычно обозначаются LPT) является подключение к компьютеру принтеров на расстоянии до 1.8 м. Поэтому распределение контактов разъемов этого порта и соответствующего кабеля, назначение сигналов, стандартные программы работы с этими портами (включая программы обработки соответствующих прерываний) ориентированы именно на это использование. Тем не менее не исключается использование этих портов и для других целей.

Порты предназначены для параллельной передачи восьмибитных групп данных и управляющих сигналов. Назначение контактов показано в табл. 7.

Таблица 7. Сигналы интерфейса Centronics

№ контакта на компьютере	№ контакта на кабеле	I/O	Наименование и назначение сигнала
1	1	O	STROBE, стробирование, сигнал начала передачи данных
2–9	2-9	O	D0–D7, биты данных
10	10	I	ACK, подтверждение принятия данных и готовности к приему следующих
11	11	I	BUSY, отсутствие готовности к приему
12	12	I	PE, конец бумаги
13	13	I	SLCT, готовность к работе
14	14	O	AUTO FD, перевод строки
15	32	I	ERROR ошибка принтера (и неготовность к работе)
16	31	O	INIT, сброс содержимого памяти принтера
17	36	O	SLCT IN, предупреждение перед передачей данных
18–25	остальные	–	GND, заземление

Таблица 8. Назначение битов регистров параллельного порта

Байт	Бит	Назначение
BASE	0 – 7	Данные
BASE+1	3	ERROR, ошибка принтера
	4	SLCT, принтер включен
	5	PE, конец бумаги
	6	ACK, запрос на прерывание
	7	BUSY, запрет передачи данных по неготовности
BASE+2	0	STROBE, стробирующий сигнал
	1	AUTO FD, перевод строки на принтере
	2	INIT, сброс текущего состояния принтера
	3	SLCT IN, предупреждение о передаче данных
	4	разрешение прерывания

Базовые адреса интерфейсов управления параллельными портами находятся в BIOS и при загрузке копируются в оперативную память: в ячейку 408_{16} для LPT1 и $40A_{16}$ для LPT2. Как правило, это 378_{16} для LPT1 и 278_{16} для LPT2. Управление и передача данных осуществляется через три регистра интерфейса, объемом по одному байту каждый, с адресами BASE, BASE+1, BASE+2. Назначение битов этих регистров – в табл. 8.

Процедура передачи данных на порт состоит из следующих шагов:

Передача данных в регистр данных.

Проверка готовности (отсутствие сигнала BUSY). Зацикливается до сообщения о готовности принтера.

По получению готовности – запись сигнала STROBE (начало передачи), затем – его сброс (конец передачи).

Ввиду большого числа операций, необходимых для передачи даже одного байта данных, скорость обмена через параллельный порт относительно невысока и редко превышает 100–150 Кбайт/с.

Последовательные порты

Основная функция последовательных портов (стандарт RS232, обычно обозначаются COM) – подключение стандартных внешних устройств (модема, мыши, некоторых видов принтеров и сканеров), а также связи компьютеров между собой. Основное отличие от параллельного порта – передача данных в виде последовательных серий импульсов (битов); при этом каждый байт «обрамляется» стартовым и стоповым битами. В результате сильно упрощается конструкция соединительного кабеля (в некоторых случаях достаточно трехпроводной линии), увеличивается (до десятков метров) расстояние, на котором возможна связь, но заметно снижается скорость передачи и усложняется управление.

Имеется возможность установки до четырех последовательных портов. На компьютере имеются 25-контактные или

Таблица 9. Назначение контактов последовательного порта

№ контакта DB9P	№ контакта DB25P	I/O	Наименование и назначение сигнала
1	–	–	PG, защитное заземление (экран)
2	3	O	TD, передача данных из компьютера
3	2	I	RD, прием данных компьютером
4	7	O	RTS, запрос на передачу данных
5	8	I	CTS, сброс для передачи, готовность к приему
6	6	I	DSR, готовность приемника
7	5	–	SG, схемное заземление, сигнальный нуль
8	1	I	DCD, детектирование приема сигнала
20	4	O	DTR, готовность передатчика
22	9	I	RI, индикатор вызова

(чаще) 9-контактные разъемы этих портов. Назначение контактов показано в табл. 9.

При передаче данных через последовательный порт каждому байту данных предшествует старт-бит (логический ноль), сигнализирующий о начале передачи, затем – биты данных, в некоторых случаях – бит контроля четности, и завершает передачу стоп-бит. Следующий старт-бит может посылаться через произвольный интервал времени; старт-биты обеспечивают синхронизацию приемника по сигналам передатчика. При этом подразумевается, что скорости приема и передачи битов совпадают. Для контроля скорости передачи используется внутренний счетчик приемника, который генерирует последовательность стробирующих импульсов, запуская ее в момент получения старт-бита. Такой механизм приема накладывает высокие требования на стабильность частоты приема-передачи и форму фронтов передаваемых импульсов, что резко ограничивает возможную частоту обмена.

Адреса 8-разрядных регистров управления последовательными портами (по восемь для каждого порта): для COM1: $3F8_{16}$ – $3FF_{16}$, для COM2: $2F8_{16}$ – $2FF_{16}$, для COM3: $3E8_{16}$ – $3EF_{16}$, для COM4: $2E8_{16}$ – $2EF_{16}$. Интерфейсные устройства па-

раллельных портов осуществляют преобразование параллельного кода в последовательный и наоборот, формирование обрамляющих битов и контроль их правильности при приеме данных, прием и передачу данных, формирование управляющих сигналов и контроль сигналов состояния внешних устройств.

На примере COM1 рассмотрим назначение отдельных битов этих регистров (у остальных портов они аналогичны). Начнем с $3FB_{16}$ – это управляющий регистр, и его содержимое определяет назначение остальных.

$3FB_{16}$ – управляющий регистр

№ бита	Назначение
0, 1	Количество бит в группе передаваемых данных: 00 – 5, 01 – 6, 10 – 7. 11 – 8
2	Количество стоп-битов: 0 – один стоп-бит, 1 – два
3, 4	Метод контроля четности: 00 или 01 – нет контроля, 10 – контроль на нечетность, 11 – контроль на четность
5	Установка контрольного бита: 1 – контрольный бит всегда равен 0 при контроле на четность или 1 при контроле на нечетность
6	1 – передача нуль-сигнала, 0 – нормальная работа
7	Бит, управляющий назначением остальных регистров

$3F8_{16}$ – регистр данных

Если старший бит регистра $3FB_{16}$ установлен в нуль, то этот регистр используется для чтения/записи данных. Если старший бит регистра $3FB_{16}$ установлен в единицу, то содержимое этого регистра – младший байт кода частоты передачи данных (см. далее).

$3F9_{16}$ – регистр управления прерываниями

Если старший бит регистра $3FB_{16}$ установлен в нуль, то в этом регистр используется для управления прерываниями.

№ бита	Назначение
0	1 – разрешение прерывания по окончании приема 0 – запрещение прерывания
1	1 – разрешение прерывания по окончании передачи 0 – запрещение прерывания
2	1 – разрешение прерывания при сбое 0 – запрещение прерывания
3	1 – разрешение прерывания при изменении управляющих сигналов 0 – запрещение прерывания

Если старший бит регистра $3FB_{16}$ установлен в единицу, то содержимое этого регистра – старший байт кода частоты передачи данных:

Код	Частота (бит/с)	Код	Частота (бит/с)
0410_{16}	110	0018_{16}	4800
0300_{16}	150	$000C_{16}$	9600
0180_{16}	300	0006_{16}	19200
$00C0_{16}$	600	0003_{16}	38400
0060_{16}	1200	0002_{16}	57600
0030_{16}	2400	0001_{16}	115200

$3FA_{16}$ – регистр идентификации прерывания

№ бита	Назначение
0	1 – нет прерывания, 0 – есть прерывание
1, 2	Причина (тип) прерывания: 00 –сбой, ошибка, 01 –окончание передачи , 10 – окончание приема, 11 – изменение управляющих сигналов

$3FD_{16}$ – регистр состояния линии (принимающего устройства)

№ бита	Назначение
0	данные получены компьютером
1	переполнение
2	ошибка четности
3	ошибка синхронизации (нет стоп-бита)
4	получен запрос на прерывание передачи (постоянный нуль-сигнал)
5	готовность принять следующий байт
6	передача закончена
7	перерыв в передаче

Использование остальных регистров определяется типом и характеристиками внешнего устройства, с которым осуществляется связь.

Порт (шина) USB

Universal Serial Bus – новый промышленный стандарт расширения архитектуры PC, ориентированный, в первую очередь, на интеграцию с устройствами бытовой техники. Первая спецификация шины появилась в 1996 г. Основные потребительские характеристики этого стандарта: простое решение, обеспечивающее скорость передачи данных до 12 Мб/с; простота кабельной системы подключений; автоматическая связь устройств с драйверами и самоконфигурирование; возможность динамического подключения. Несмотря на сравнительно короткое время, прошедшее с момента появления этих портов, в настоящее время выпускается достаточно большое количество системных плат, в которых они реализованы на уровне чипсета.

Линия USB обеспечивает подключение к компьютеру большого числа одновременно доступных периферийных устройств. Распределение ресурсов между устройствами осуществляется чипсетом материнской платы и реализуется с помощью посылки устройствам специальных сигналов – маркеров. Устройства USB, кроме выполнения своих непосредственных функций, могут выполнять также задачи коммутаторов (хабов), то есть пересылать пришедшую к ним по шине информацию на другие устройства, подключенные далее к той же линии. Каждое USB устройство должно иметь стандартный интерфейс, обеспечивающий поддержку протокола обмена информацией, выполнять стандартные операции конфигурирования и сброса и предоставлять информацию о себе в стандартной форме.

Физическое соединение устройств осуществляется по топологии многоуровневой звезды. Центром звезды любого уровня является устройство-коммутатор, к которому подключаются простые устройства или коммутаторы нижних уров-

ней. Фактически таким же коммутатором является и контроллер линии, установленный на материнской плате; обычно у него имеется две точки подключения (два USB-порта).

Информационные сигналы и напряжение питания устройств (5 В) передаются по четырехпроводному кабелю. На кабелях используются два типа разъемов. Разъем типа A предназначен для подключения к восходящим (лежащим по линии ближе к компьютеру) коммутаторам и обычно устанавливается на кабелях, не отсоединяемых от устройств. Разъем типа B устанавливается на устройствах, от которых соединительный кабель может отсоединяться, и предназначен для подсоединения к нисходящим устройствам. Разъемы отличаются механически, что исключает их неправильное подключение. Конструкция разъемов обеспечивает позднее соединение и раннее отсоединение сигнальных линий по сравнению с линиями питания. Назначение контактов разъемов: 1 – питание (+5 В), 2 и 3 – линии передачи данных (обычно обозначаются D– и D+), 4 – заземление.

Питание устройств может осуществляться как от кабеля, так и от собственных блоков питания. Ряд современных материнских плат могут поддерживать достаточно развитую систему управления энергопотреблением, обслуживая, например, программную приостановку и восстановление питания устройств.

Шина имеет два режима скорости передачи. Полная скорость составляет 12 Мбит/с, для ее обеспечения в качестве кабеля используется экранированная витая пара с длиной сегмента до 5 м. Низкая скорость составляет 1.5 Мбит/с, для нее используется невитой неэкранированный кабель с длиной сегмента до 3 м. Система позволяет использовать одновременно оба режима, переключение скорости для отдельных устройств осуществляется программно; каждому пакету данных предшествует последовательность сигналов синхронизации, что дает возможность приемнику настроиться на частоту передатчика.

Сигналы на линиях D– и D+ управляются независимо, что позволяет организовать достаточно широкий набор состояний линии.

Каждое USB устройство представляет собой набор пронумерованных конечных точек (аналог регистров портов), с которыми вышележащее устройство-коммутатор может обмениваться информацией. Каждое устройство имеет, как минимум, одну точку с номером 0, которая используется для общего управления устройством и опроса его состояния. Эта точка автоматически конфигурируется при подаче питания на устройство. Кроме нее, низкоскоростные устройства имеют две дополнительные точки, полноскоростные – до 16. Прежде, чем осуществлять обмен информацией с ними, контроллер шины осуществляет конфигурирование *канала USB* – то есть согласовывает модель передачи данных для данной точки. Имеются два типа каналов: потоки, которые используются для передачи однонаправленных серий бит от контроллера к точке или обратно, и сообщения, по которым контроллер посылает запрос в точку, получает от нее запрошенную информацию, за которой следует информация о состоянии точки (подтверждение конца передачи).

USB поддерживает четыре базовых типа передачи данных.

1. *Управление* – используется для конфигурирования канала при подключении устройства и для управления устройством в процессе работы. Сигнал представляет собой слово последовательно передаваемых бит длиной 64 байта на полной скорости канала и 8 байт – на низкой скорости.

2. *Сплошная передача* – используется для передачи больших пакетов данных без жестких требований ко времени передачи. Передаваемые пакеты могут состоять из слов по 8, 16, 32 и 64 байта. При перегрузке шины передача прекращается и возобновляется после освобождения. Допускается только на полной скорости канала.

3. *Прерывания* – короткие (64/8 байт) сообщения от устройства к контроллеру. Скорость обслуживания прерывания устанавливается при конфигурировании устройства и должна составлять не менее 255 мс.

4. *Изохронная передача* – похожая на сплошную передача больших пакетов данных, однако, в отличие от нее, с заданным временем доставки. В случае обнаружения ошибки передачи (например, из-за неготовности устройства-приемника) ошибочные данные игнорируются, и передача продолжается. Используется в случаях, когда критичной является скорость передачи, например, при цифровой передаче голоса. Изохронные передачи классифицируются по типу синхронизации передатчика с системой: различают асинхронный, синхронный и адаптивный классы устройств-передатчиков, которые устанавливаются при их первоначальной конфигурации.

Все USB обмены (*транзакции*) состоят из трех пакетов. Первым следует *пакет-маркер*, которым контроллер задает тип передачи, адрес устройства-приемника и номер точки в нем. Получив этот пакет, устройство подготавливается к обмену и передает *пакет данных* (либо сообщение об их отсутствии). После приема устройство-приемник посылает *пакет подтверждения*.

Появление ошибок передачи контролируется аппаратными средствами шины; если после трех попыток передать данные не удалось, контроллер выдает сообщение об ошибке системному программному обеспечению.

Данные пакетов передаются последовательно, начиная с младшего бита. Каждый пакет начинается с поля синхронизации. Далее следует четырехбитное поле – идентификатор типа пакета, за ним – те же четыре бита в инвертированном состоянии.

Таблица 10. Типы пакетов шины USB

Тип пакета	Имя	Код	Содержание и назначение
Маркер	OUT	0001	Адрес устройства и номер точки приемника
Маркер	IN	1001	Адрес устройства и номер точки передатчика
Маркер	SOF	0101	Маркер начала кадра
Маркер	SETUP	1101	Адрес устройства и номер точки управления
Данные	DATA0 DATA1	0011 1011	Пакеты данных с четным и нечетным идентификатором чередуются, для контроля приема
Подтверждение	ACK	0010	Подтверждение приема
Подтверждение	NAK	1010	Ошибка приема
Подтверждение	STALL	1110	Неготовность приемника
Специальный	PRE	1100	Передача на низкой скорости

В пакетах-маркерах OUT, IN, SETUP далее следуют адресные поля: 7-битный адрес устройства и 4-битный адрес точки внутри устройства. В пакете SOF далее идет 11-битное поле номера кадра (блока данных), значение которого последовательно увеличивается в процессе передачи.

Далее следует поле данных, которое может иметь до 1023 целых байт. Завершается передача пакетом подтверждения.

USB поддерживает подключение и отключение устройств в процессе работы; их нумерация является постоянным процессом и отслеживает динамические изменения топологии подключения USB устройств. При подключении нового устройства к коммутатору последний по запросу контроллера сообщает ему о состоянии вновь подключенного устройства. Контроллер, адресуясь через канал управления по нулевому адресу, определяет, является ли вновь подключенное устройство «простым» или коммутатором, назначает ему адрес USB, устанавливает каналы управления и передает уведомление о подключении соответствующему системному программному обеспечению.

Когда устройство отключается, соответствующий коммутатор сообщает об этом контроллеру, который удаляет сведения о нем из всех структур данных. Если отключается коммутатор, то процесс отключения выполняется для всех подключенных к нему устройств.

Исходя из вышесказанного, все устройства USB должны поддерживать следующий набор операций.

Динамическое подключение – означает возможность запуска и остановки устройства в любой момент времени. Устройство до конфигурации и после сброса должно отзываться на нулевой адрес. После конфигурации и назначения уникального адреса устройство должно отзываться только на этот адрес.

Конфигурирование устройства – означает, что устройство должно предоставлять контроллеру по его запросу информацию о имеющихся в нем точках подключения, их свойствах и функциях.

Передача данных – по крайней мере одним из четырех типов. После конфигурирования каждой точке устройства назначается только один тип передачи, даже если она и может поддерживать несколько.

Управление энергопотреблением – любое устройство при начальном подключении не должно потреблять с шины ток свыше 100 мА. При конфигурации устройство сообщает требуемый рабочий ток с шины (не более 500 мА), и если коммутатор не может его обеспечить, устройство не конфигурируется.

Режим приостановки –устройство должно переходить в этот режим при приостановке активности шины. При этом потребляемый им ток не должен превышать 500 мкА.

Удаленное пробуждение – возможность для приостановленного устройства подать контроллеру сигнал запроса на обслуживание. Эта возможность должна быть описана в конфигурации устройства и может быть запрещена при конфигурировании.

Коммутатор – возможность для устройства выполнять функции коммутации сигналов, направленных контроллером на подключенные к нему другие USB устройства.

Как видно из изложенного, USB является развитой системой для подключения большого числа внешних устройств к ЭВМ, обеспечивающей быстрый обмен информацией (хотя и не такой быстрый, как шина PCI) и гибкое конфигурирование сети этих устройств. Однако сложный протокол обмена информацией по последовательному каналу требует наличия достаточно мощного интерфейса непосредственно у измеряющего устройства, что делает их сложными и, как следствие, дорогостоящими. В связи с этим количество аналоговых измерительных устройств, имеющих возможность подключения к ЭВМ по USB каналу, в настоящее время крайне ограниченно.

Устройства сопряжения ЭВМ и экспериментальных установок

С целью обмена данными между контрольно-измерительными приборами и компьютерами было разработано большое количество специализированных интерфейсных устройств. Однако при их использовании в физическом эксперименте возникают следующие проблемы. Во-первых, конфигурация ЭВМ (в частности, IBM PC) предусматривает подключение ограниченного количества интерфейсных устройств, тогда как современные экспериментальные установки требуют работы с достаточно большим числом разнородных каналов связи. Во-вторых, проведение экспериментальных исследований предполагает достаточно частое изменение конфигурации установки. Использование же разнородных интерфейсов приводит к тому, что каждая такая реконструкция измерительной части установки повлечет за собой радикальную переработку аппаратного и программного обеспечения ее автоматизации. В связи с этим широкое распространение получили модульные интерфейсные системы, примером

которых, принятым Академией наук в качестве внутреннего стандарта, является система КАМАК.

СИСТЕМА КАМАК

Историческая справка

В 60-х годах развитие вычислительной техники привело к необходимости стандартизировать не только размеры модулей, но и каналы связи между ними.

Система КАМАК (CAMAC – Computer Automated Measurement And Control) была разработана и предложена совместно европейским комитетом ESONE – European Standards of Nuclear Electronics и американским комитетом US NIM. Она представляет собой модульную систему, предназначенную для связи измерительных устройств с цифровой аппаратурой обработки данных (в большинстве случаев его роль выполняет компьютер). Первоначально концепция системы была принята пятью лабораториями, находившимися во Франции, ФРГ, Италии, Бельгии, Нидерландах, с целью выбора общего стандарта и определения общих потребностей. В 1966 г. началась работа над системой стандартов КАМАК ядерными электронщиками ведущих европейских институтов для оснащения сложных экспериментов, например, на ускорителях атомных частиц. К этому времени в комитет уже входило 26 лабораторий, включая лаборатории Швейцарии, Англии, Австрии и Югославии. С 1974 г. членами комитета являются 29 лабораторий и организаций, среди них: ZERN, Hauell, Saclay, Grenoble, DESI, Frascatti.

В течение 1967–1970 гг. рабочие группы комитета разработали подробные спецификации и выпустили основные стандарты:

1. Стандарт EUR 4100e (1969 г.), в котором рассматриваются конструктивы, источники питания, цифровые сигналы и магистраль крейта.

2. Стандарт EUR 4600e (1970 г.) – предварительный вариант; охватывает магистраль ветви и контроллер типа А.

3. Стандарт EUR 5100e (1970 г.) – предварительный вариант; распространяется на аналоговые сигналы.

В 1972 г. документ EUR 4100e был пересмотрен и дополнен новыми требованиями и рекомендациями, вытекающими из опыта применения системы.

Через три–четыре года после публикации стандарта десятки фирм в разных странах выпускали модули КАМАК более трехсот типов как для экспериментов, так и для контроля и управления технологическими процессами на производствах. В Советском Союзе ядерные электронщики сразу же оценили новую систему, и уже в 1970 году ведущие лаборатории: Объединенный институт ядерных исследований, Ленинградский и Новосибирский институты ядерной физики – начали разработку модулей в стандарте КАМАК для своих нужд. В СССР также выпускалась модульная аппаратура «Вектор» с логическим протоколом КАМАК, но в модулях собственной конструкции в «миллиметровых» размерах. Несоответствие размеров модулей международным стандартам МЭК-297 и EUR-6100 привело к тому, что «Вектор» не выдержал конкуренции с системой КАМАК и более ста типов разработанных модулей были практически не востребованы. Наличие двух типов модульных систем КАМАК и «Вектор» затянуло стандартизацию системы КАМАК в СССР (соответствующий ГОСТ 26.201-80 был принят лишь в 1980 году в чрезвычайно искаженном виде) и воспрепятствовала промышленному выпуску аппаратуры КАМАК, которую производили лишь институты ядерных исследований, а закупали, в основном, в Польше и Венгрии.

Создатели системы КАМАК в конце 60-х годов начали применять только что появившиеся интегральные микросхемы в своих разработках, однако у них не хватило смелости предположить, что в 1972 году в электронике начнется революция – появится микропроцессор. Неудобства магистрали

КАМАК заставили искать решения, позволяющие эффективно использовать качественно новую интегральную схему. Введение микропроцессоров в модули превращало их в микрокомпьютеры, а крейты – в многопроцессорные системы, которые нуждаются в емкой памяти с большим количеством адресов. 16 адресов в модуле КАМАК оказались совершенно недостаточными, поэтому ведущие электронные фирмы Motorola и Intel к середине 70-х годов создали модульные системы третьего поколения Versabus Module Europe bus (VME bus) и Multibus, магистрали которых содержали 16, а затем и 20 адресных линий, что обеспечивало примерно 1 млн. адресов. В дальнейшем появляются и другие модульные системы (PXI, VXI, Multibus-II, и GPIB), которые обладают большей пропускной способностью шины данных и более высоким быстродействием.

Электронные модульные системы долговременны. Если модули достаточно широко распространились и их количество превзошло некоторый критический уровень, то даже морально устаревшую аппаратуру оказывается выгодным эксплуатировать. Изменить устоявшиеся стандарты модулей практически невозможно; они будут использоваться в рамках своих возможностей. Такая ситуация сложилась и с системой КАМАК: несмотря на то, что ей уже более 30 лет, она все еще широко используется как с персональными ЭВМ, так и с микропроцессорами, встроенными непосредственно в контроллер. Большой парк накопившихся разнообразных модулей позволяет в течение нескольких дней, а то и часов, скомпоновать систему с новыми характеристиками.

Основные структуры системы

Основой системы КАМАК является *крейт* с 25 ячейками (*станциями*), в которые по направляющим вставляются модули, включая контроллер. Модуль может занимать одну или несколько ячеек. Обмен данными между модулями происходит по внутренним шинам крейта – горизонтальной магистрали и организуется контроллером.

В электронной системе модулем является печатная плата с узкой передней панелью и плоским многоконтактным разъемом на противоположной стороне платы. Модули вставляют в каркас с направляющими, в которых скользит плата. Задняя стенка каркаса выполнена также в виде платы с ответными частями разъемов, которые соединены печатными или навесными проводниками, образующими электрические линии магистрали для передачи кодированной информации. По специально назначенным проводникам в модули подается электрическое питание.

Все размеры модулей и каркасов строго стандартизованы: габариты и толщина печатных плат модулей, ширина канавок в направляющих и расстояние между ними, расположение контактов и так далее. Определены длительности и амплитуды электрических сигналов, а также напряжения питания модулей. Стандартизованы не только размеры, но и логический протокол – правила передачи информации по линиям магистрали.

Существует несколько конфигураций системы, обусловленных выбранным способом управления крейтом и организацией его связи с управляющей ЭВМ.

1. АВТОНОМНАЯ СИСТЕМА КАМАК. Возможны два основных варианта ее реализации. Первый – на основе программируемого управления крейтом с использованием простого ориентированного на пользователя языка для организации часто проводимых операций, таких, как например периодический опрос содержимого счетчиков импульсов и передача результатов счета в память. Второй вариант автономной системы базируется на управлении крейтом программируемым компьютером, который встраивают в контроллер крейта (микропроцессорный контроллер) или в качестве вспомогательного контроллера размещают в одной из ячеек крейта и связывают со стандартным контроллером крейта дополнительной магистралью.

   Непосредственное управление от компьютера в этом слу-

чае не предусмотрено, однако допускается обмен с внешней ЭВМ – передача ей сжатых данных измерений и прием системной информации для управления.

2. ПОДСИСТЕМА КАМАК. В системах с компьютером, в которых используется один или несколько крейтов, может оказаться целесообразным выход контроллера крейта непосредственно связывать с каналом ввода-вывода данных машин. Преимущества такого подхода – низкие затраты на сопряжение. Из-за условий ограниченности длины линий связи крейты в этом случае необходимо размещать непосредственно вблизи машины, а для каждого крейта должен быть выделен отдельный программно управляемый канал ввода-вывода данных компьютера.

3. СИСТЕМА С ВЕРТИКАЛЬНОЙ МАГИСТРАЛЬЮ. Обмен информацией между несколькими крейтами (максимально – до семи) и компьютером может осуществляться через так называемую вертикальную магистраль с параллельной передачей данных. Подобная структура из-за больших затрат на организацию кабельной магистрали с параллельными линиями оказывается целесообразной для средних и больших пространственно ограниченных систем. Скорость передачи данных в магистрали может превышать 10^7 бит/с. При определенных условиях к компьютеру может быть присоединено несколько вертикальных магистралей.

4. ПРОСТРАНСТВЕННО-РАСПРЕДЕЛЕННАЯ СИСТЕМА. Для систем, элементы которых удалены друг от друга на значительные расстояния, используется канал с последовательной передачей данных между компьютером и крейтами. Канал представляет собой однонаправленную замкнутую цепь (кольцевую магистраль), в которую последовательно друг за другом включают до 62 крейтов. Двоичные данные передаются поразрядно или пословно

(побайтно) со скоростью, обусловленной характеристиками каналов связи. Так, например, при использовании телефонных линий связи скорость передачи оказывается существенно меньше в сравнении с параллельной передачей данных в вертикальной магистрали.

Виды модулей КАМАК

1. Счетчики.
2. Регистры ввода-вывода.
3. Таймеры.
4. Приводы.
5. Интерфейсы.
6. Блоки цифровой обработки сигналов:
   - 6.1. Логические преобразователи.
   - 6.2. Преобразователи кодов.
   - 6.3. Память.
   - 6.4. Специализированные процессоры.
7. Блоки аналоговой обработки сигналов:
   - 7.1. Преобразователи напряжений.
   - 7.2. Аналого-цифровые преобразователи.
   - 7.3. Цифро-аналоговые преобразователи.
   - 7.4. Мультиплексоры.
   - 7.5. Ключи.
   - 7.6. Прочие.
8. Контрольное оборудование:
   - 8.1. Индикатор магистрали.
   - 8.2. Ручной контроллер.

Организация горизонтальной магистрали (шины КАМАК)

Горизонтальная магистраль состоит из 86 линий, каждая из которых соединена с контактом приемной части разъема крейта. Назначение линий:

Команды

1. Пять линий типа N – номер модуля, которому предназначена команда.

2. Четыре линии типа A – адрес внутри модуля (в случаях, если модуль объединяет в себе несколько устройств).

3. Пять линий типа F – номер функции, которую следует выполнить.

Состояние

4. X – готовность.

5. Q – ответ модуля на запрос.

6. L – запрос модуля.

7. B – шина занята.

Управление

8. Z – пуск.

9. C – сброс содержимого всех регистров.

10. I – остановка операций.

11. S1, S2 – стробирующие сигналы.

Передача

12. 24 линии чтения информации R.

13. 24 линии записи информации W.

Все линии можно разделить на магистральные, соединяющие между собой все соответствующие контакты разъемов станций, и индивидуальные, которые соединяют контакты разъемов нормальной станции с контактами разъемов управляющей станции. К индивидуальным относятся 24 линии адреса или **N**–линии, по которым сигнал от управляющей станции инициирует работу модуля, и линии запроса **L**, по которым сигнал от модуля выставляет требование на обработку данных, например, по завершению какой-либо операции в модуле.

Остальные линии являются магистральными. Из них 14 линий питания и 17 управляющих проходят через контакты всех 25 станций, 24 линии записи, 24 линии чтения и две дополнительные соединяют контакты разъемов только нормальной станции. По управляющим линиям команды и синхронизирующие импульсы поступают на нормальную станцию, от станции выставляется сигнал о состоянии модуля, о ходе и завершении операции.

По линиям записи данные поступают на разъемы нормальной станции, по линиям чтения данные передаются к месту назначения.

Линии питания обеспечивают различные диапазоны напряжений. В качестве обязательного набора в стандарте приняты напряжения питания ±5 В при максимальном токе 25 А на крейт и 2 А на станцию, и ±24 В при максимальном токе 6 А на крейт и 1 А на станцию. В качестве дополнительного набора приняты напряжения ±12 В, ±200 В и переменное напряжение 117 В. Рассеиваемая мощность не должна превышать 200 Вт в крейте и 8 Вт в станции.

Линии N, A, F

АДРЕС ЯЧЕЙКИ **N**. Каждая ячейка крейта (ячейка модуля) адресуется контроллером по отдельной соответствующей линии выборки **N** (ячейки на лицевой панели крейта нумеруются слева направо от **N** = 1 до **N** = 23). При использовании дополнительного регистра в контроллере может быть организовано параллельное адресное обращение к нескольким ячейкам (модулям) крейта.

СУБАДРЕСА **A**. Один модуль может иметь несколько источников/приемников информации. Это могут быть несколько независимых устройств, собранных в виде единого модуля, либо единое устройство, разделенное на аппаратные или логические блоки. Обращение к одному из них задается кодом субадреса. Четыре субадресные линии для передачи двоичного кода (линии **A8**, **A4**, **A2**, **A1**) позволяют выбирать по адресу до 16 различных узлов и частей внутри самого модуля. Адрес узла можно задавать любым от **A(0)** до **A(15)**.

ФУНКЦИИ (ОПЕРАЦИИ) **F**. Каждый элемент модуля может выполнять до 32 различных операций. Для задания одной из функций от **F(0)** до **F(31)** используют пять функциональных линий **F16**, **F8**, **F4**, **F2** и **F1**. Значения этих функций и выборки указаны в табл. 11.

В выбранном по адресу **N** модуле коды субадреса и функции дешифруются. Допускается также разделение отдельных разрядов кода выборки функций на группы с последующим частичным дешифрированием для выделения дополнительных признаков. Так, например, командами **F16** = 0 и **F16** = 1 разделяют функции чтения и записи соответственно.

При инициализации модуль полностью дешифрует субадрес и команду и подает в магистраль сигнал **X**. При определенных командах модуль может выработать сигнал **Q**. Эти сигналы принимаются контроллером крейта по стробу **S1**.

Таблица 11. Команды системы КАМАК

Номер команды	Назначение	F16	F8	F4	F2	F1	Использование линий данных
F(0)	Чтение содержимого регистра группы 1	0	0	0	0	0	Используются линии R
F(1)	Чтение содержимого регистра группы 2	0	0	0	0	1	
F(2)	Чтение и сброс регистра группы 1	0	0	0	1	0	
F(3)	Чтение дополнения – содер-жимому регистра группы 1	0	0	0	1	1	
F(4)	Нестандартная	0	0	1	0	0	
F(5)	Резервная	0	0	1	0	1	
F(6)	Нестандартная	0	0	1	1	0	
F(7)	Резервная	0	0	1	1	1	
F(8)	Контроль требований	0	1	0	0	0	Не используются
F(9)	Сброс регистра группы 1	0	1	0	0	1	
F(10)	Гашение сигнала-требования	0	1	0	1	0	
F(11)	Сброс регистра группы 2	0	1	0	1	1	
F(12)	Нестандартная	0	1	1	0	0	
F(13)	Резервная	0	1	1	0	1	
F(14)	Нестандартная	0	1	1	1	0	
F(15)	Резервная	0	1	1	1	1	
F(16)	Перепись содержимого регистра группы 1	1	0	0	0	0	Используются линии W
F(17)	Перепись содержимого регистра группы 2	1	0	0	0	1	
F(18)	Селективная установка регистра группы 1	1	0	0	1	0	
F(19)	Селективная установка регистра группы 2	1	0	0	1	1	
F(20)	Нестандартная	1	0	1	0	0	
F(21)	Селективный сброс регистра группы 1	1	0	1	0	1	
F(22)	Нестандартная	1	0	1	1	0	
F(23)	Селективный сброс регистра группы 2	1	0	1	1	1	
F(24)	Блокирование	1	1	0	0	0	Не используются
F(25)	Исполнение	1	1	0	0	1	
F(26)	Деблокирование	1	1	0	1	0	
F(27)	Проверка состояния	1	1	0	1	1	
F(28)	Нестандартная	1	1	1	0	0	
F(29)	Резервная	1	1	1	0	1	
F(30)	Нестандартная	1	1	1	1	0	
F(31)	Резервная	1	1	1	1	1	

Для операций чтения определены четыре команды: **F(0)**, **F(1)**, **F(2)**, **F(3)**. По этим командам содержимое регистров, к которым произошло обращение, выставляется на линии чтения R, и по стробу **S1** переписывается в регистр-приемник. Сброс регистра командой **F(2)** происходит по стробу **S2**. Команды **F(4)**, **F(6)** – нестандартные и при разработке модуля могут использоваться их по своему усмотрению. Команды **F(5)**, **F(7)** зарезервированы для дальнейших расширений. Цикл в команде модуля может быть больше цикла КАМАК, в этом случае модуль после окончания операции выработает и выставит на шину **L**-запрос.

Для операций записи определены шесть команд **F(16)**, **F(17)**, **F(18)**, **F(19)**, **F(21)**, **F(23)**. По этим командам содержимое регистра-источника (либо преобразованный код регистра-источника) выставляется на линии **W** и по стробу **S1** переписывается в регистр модуля. Команды **F(20)**, **F(22)** нестандартные, т. е. разработчики модулей могут использовать их по своему усмотрению.

Команды **F(9)**, **F(11)** сбрасывают содержимое модуля.

Содержимое регистров второй группы **A(12)** – регистр состояния, **A(13)** – регистр маски, **A(12)** – регистр запроса можно прочитать или заменить командами чтения или записи. При наличии большого количества источников в модуле рекомендуется пользоваться этими командами. В этом случае каждый источник привязан к конкретному разряду регистров состояния **A(12)**, **A(13)**, **A(14)** и наличие запроса от конкретного источника обнаруживается значением соответствующего разряда.

Каждый модуль может генерировать сигнал **L**-запрос на обработку. Линии, по которым передается этот сигнал, являются индивидуальными, как и **N**-линии. Адресуемый модуль не должен выставлять **L**-сигнал до конца текущей операции. Неадресуемый модуль может устанавливать **L**-сигнал в любое время. Когда модуль, который генерирует **L** = 1, принимает команду, заставляющую его устранить этот вызов, он должен запретить **L**-сигнал или сбросить **L**-запрос.

Команды **F(8)** – **F(15)** шины **R** и **W** не используют. С помощью команды **F(8)** может проверить наличие запроса от конкретного источника, адресуясь к соответствующему разряду регистра запроса **A(14)**. Субадрес команды **F(8)** можно интерпретировать как номер разряда регистра **A(14)**. Например, команда **F(8)A(23)** проверяет наличие запроса от источника, который соответствует разряду 23 запроса. Команда вырабатывает ответный сигнал **Q** = 0, если разряд в состоянии 0 и **Q** = 1, если разряд в состоянии 1. Команда запрос не сбрасывает.

В качестве примера можно привести следующую часто возникающую ситуацию. Например в работе используется модуль «таймер», который должен отмерять длительные (по сравнению с собственными временами обработки команд КАМАК) промежутки времени. Необходимо приостановить работу программы до окончания заданного промежутка времени. В момент окончания промежутка времени модуль «таймер» выставляет на шины сигнал **Q**. В этом случае программируется цикл, состоящий из запроса **F(8)** к интересующему нас модулю. Условием выхода из цикла является получение сигнала **Q**.

Команда **F(10)** сбрасывает запрос от источника, указанного в субадресе команды. При наличии регистра запроса **A(12)** эквивалентна сбросу соответствующего разряда регистра.

Команды **F(24)** – **F(31)** шины **R** и **W** не используют. Команда **F(24)** запрещает какую-либо функцию модуля или маскирует **L**-сигнал. Элемент модуля, функции которого запрещаются, задается субадресом команды, при наличии регистра маски **A(13)**. Действие команды начинается по **S1** или **S2**.

Команда **F(25)** инициирует исполнение какой-либо функции, ее начало или окончание. Команда используется, если команды **F(24)** и **F(26)** непригодны. Элемент, который инициализируется командой, задается субадресом команды. Субадрес может интерпретироваться как задание конкретного действия из множества возможных действий. Действие может начинаться по **S1** или **S2**. Например в модуле «счетчик $СЧ_6$10» эта команда используется для увеличения на единицу количества отсчетов.

Команда **F(26)** разрешает какую-либо функцию элемента или снимает маску **L**-сигнала. При наличии регистра маски выполнение команды эквивалентно установке соответствующего разряда регистра **A(13)**. Это команда, обратная к команде **F(24)**. Действие начинается по **S1** или **S2**.

Команда **F(27)** вырабатывает на **Q**-линии ответ, соответствующей состоянию выбранной части модуля по субадресу команды. Характеристика, которая выбирается субадресом, может статусной, что при наличии регистра состояния **A(12)** эквивалентно проверке соответствующего разряда **A(12)**.

Команды **F(28)**, **F(30)** не стандартизованы, **F(29)**, **F(31)** – зарезервированы для дальнейших расширений.

Данные записи W и считывания R

Информация, передаваемая по линиям чтения **R** или записи **W**, представляет собой коды численных значений данных, сообщения о состоянии модулей и сигналов или сигналы, которые в модулях используются для управления.

ЛИНИИ ЗАПИСИ W1–W24. По этим линиям контроллер передает модулю 24-разрядный параллельный двоичный код. Необходимыми условиями передачи являются завершение установление передаваемых сигналов до посылки контроллером строб-сигнала **S1** и сохранение сигналов данных неизменными во время действия строб-сигнала **S1**. Использование для этих целей сигнала **S2** допустимо лишь в специальных случаях.

ЛИНИИ ЧТЕНИЯ R1–R24. По этим линиям модуль передает данные контроллеру. Сигналы этих данных должны устанавливаться до появления строб-сигнала **S1** и сохранятся неизменными до завершения операции. Операции по приему данных контроллер выполняет во время существования строб-сигнала **S1**.

Ответ – сигнал занятости **B** = 1 о состоянии данных в линиях **W** и **R** модуль должен формировать только при командных операциях и операциях по передаче данных. Допускается обмен двоичными словами, содержащими меньше 24 разрядов,

однако для контроллера рекомендуется использовать всю разрядность.

Сигналы состояния B, X, Q

Сигнал занятости **B** = 1 указывает на выполнение в данное время модулем операции (адресной или безадресной).

Сигнал **X** = 1 о восприятии команды выдает модуль, оповещая тем самым о возможности самостоятельного выполнения этой команды или о ее выполнении совместно со связанным внешним прибором. Сигнал **X** должен быть установлен до появления строб-сигнала **S1**. Во время действия сигнала **S1** выполняется анализ значения **X**, которое должно сохраняться неизменным до завершения строб-сигнала **S2**. Нулевое значение сигнала **X** = 0 указывает на наличие серьезной ошибки, например, на отсутствие самого модуля в ячейке, питания модуля, требуемого внешнего прибора и др. Если в интерфейсной части системы используется процессор, то с получением сигнала **X** = 0 он показывает выполнение текущей программы и начинает реализацию программы диагностики ошибок.

Сигнал подтверждения **Q** используется в ряде случаев. Так, выбранный по адресу модуль во время выполнения им операции посылкой **Q** может сигнализировать о своем соответствующем состоянии. Значение сигнала в линии **Q** контроллер оценивает во время действия строб-сигнала **S1**. При выполнении операций чтения и записи адресованный модуль должен установить нулевое или единичное значение **Q** до появления строб-сигнала **S1** и сохранять его неизменным по крайней мере до завершения действия строб-сигнала **S2**.

Запрос на прерывание L

Сигналом **L** модуль посылает заявку о необходимости прервать текущую программу и начать выполнение программы обслуживания этого модуля. Сигнал передается в контроллер по линии выборки **L**, которые нумеруются в соответствии с номерами станций: L1 – L23.

В модуле может быть несколько источников, требующих обслуживания с прерыванием. Для идентификации таких источников можно присваивать им субадреса (при небольшом числе источников) или использовать специальный регистр заявок на обслуживание.

Общие сигналы управления Z, C, I

Сигналы подготовки **Z** и гашения **C** формируются при выполнении безадресных операций при передаче данных и должны воздействовать на все устройства, связанные общими линиями этих сигналов, до появления строб-сигнала **S2**. Эти сигналы требуют одновременного с ними действия сигнала **B** = 1. Сигнал **Z** имеет абсолютный приоритет по отношению ко всем прочим сигналам, и при установке **Z** = 1 все регистры сбрасываются в свои начальные состояния. Сигналом **C** = 1 сбрасываются в нулевое состояние только выбранные пользователем регистры и отдельные триггеры.

Сигнал блокировки **I** может быть сформирован в произвольный момент времени, блокируя в модуле реализацию всех предусмотренных функций.

Организация установки

Управление работой крейта и, как правило, организация связи крейта с компьютером возлагается на контроллер крейта, который должен занимать управляющую и одну нормальную станцию: они соответствуют двум правым станциям (24, 25). В этом случае он выполняет роль основного контроллера крейта. Возможна ситуация, когда контроллер будет установлен в любую другую станцию крейта. В этом случае он будет выполнять функции вспомогательного контроллера. Такое соединение применяется, если необходимо соединить несколько крейтов. Основной контроллер связан с компьютером, а остальные крейты связаны между собой с помощью вспомогательных контроллеров. Оставшиеся станции занимают исполнительные модули. Они связываются с контроллером крейта через магистраль.

Контроллер крейта играет роль управляющего модуля и, в большинстве случаев, – роль пассивного приемника команд КАМАК, которые ему передает компьютер. После получения команды контроллер дешифрует адрес модуля и генерирует сигнал на индивидуальных линиях **N**.

Для подключения контроллера крейта к компьютеру служат специализированные интерфейсные платы, устанавливаемые внутри компьютера и подключаемые к его шине. Применяя различные интерфейсные платы, можно подключать разные компьютеры к одной и той же установке, при этом необходимо лишь минимально модифицировать программное обеспечение, управляющее процессом обмена данными между процессором и интерфейсной платой.

Данные в контроллер крейта могут передаваться параллельно или последовательно, в зависимости от вида интерфейсной платы.

Управление крейтом КАМАК от IBM PC. Последовательный интерфейс

Интерфейсная плата устанавливается внутри системного блока компьютера в один из разъемов шины EISA (ISA), и работает как последовательный (COM) порт. В качестве базового адреса выступает один из базовых адресов COM порта (какой конкретно – определяется установками перемычек-джамперов на плате). Контроллер имеет четыре внутренних 16-разрядных регистра: CSR, DMR, DHR и регистр управления магистралью КАМАК (рассмотрение ведется на примере контроллера К-16). Все регистры контроллера – 16-разрядные, поэтому работа с ними осуществляется, как правило, с помощью 16-разрядных команд.

Базовый адрес крейта – это адрес управляющего *регистра команд и состояний* (*Command and Status Register, CSR*). Назначение его разрядов:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Q	X*		I*	S	X	Z	C	D	DE	I	F16	F8	F4	F2	F1

Разряды 0–4 используются для указания двоичного номера функции, которую необходимо выполнить.

Пятый разряд I вызывает появление сигнала запрета операций на линии I шины контроллера.

Шестой разряд DE разрешает контроллеру осуществлять прерывание работы процессора и требовать обработки прерывания.

Седьмой разряд D – требование такого прерывания, выставляется контроллером.

Восьмой разряд C устанавливается компьютером и вызывает появление сигнала C на шине КАМАК, что приводит к сбросу всех команд и значений на линиях чтения/записи.

Девятый разряд Z устанавливается компьютером и вызывает появление сигнала Z на шине КАМАК; приводит к воспроизведению начальной установки состояния крейта.

Разряд X – разрешение работы с LAM запросами.

Разряд S используется для установки режима стробирования операций КАМАК. В большинстве крейтов его программная установка приводит к запрету выполнения операций.

Разряд I* используется для контроля за состоянием линии I на магистрали крейта. Программно не изменяется, может только считываться.

Разряд 13 не используется.

Разряд X* – предназначен только для считывания; его значение определяется контроллером и информирует о появлении LAM запроса от одной из станций (если работа с этими запросами была разрешена разрядом X). Если при этом была разрешена работа с прерываниями, то контроллер одновременно формирует запрос на прерывание.

Разряд Q выставляется контроллером и информирует об окончании выполнения команды и готовности принять следующую.

Следующий управляющий регистр – *регистр запросов и маски* (*Demand and Mask Register, DMR*). Этот регистр управляет разрешениями прерываний от отдельных станций, и служит для передачи запросов на прерывания. По умолчанию

подобные запросы могут выставлять восемь станций (5–12), хотя программно эти назначения могут меняться (функция F(8)). Младший байт регистра служит для выставления разрешений на прерывание для каждой из этих станций (байт маски); старший – для индикации поступивших запросов (байт запросов). При возникновении незамаскированного запроса контроллер выставляет запрос на прерывание в CSR, а по состоянию регистра DMR можно определить, какая из станций послужила источником запроса.

Третий служебный регистр – *регистр старшего байта* (*Data High bite Register, DHR*) – предназначен для приема/передачи старшего третьего байта данных. Появление этого байта обусловлено тем, что шина КАМАК 24-разрядная, тогда как регистры данных станций – 16-разрядные. Не вошедший старший байт данных с шины КАМАК передается в DHR.

Четвертый регистр предназначен для определения адреса станции в крейте и приема/передачи данных. В него сначала направляется адрес станции в виде:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				C4	C2	C1	N16	N8	N4	N2	N1	A8	A4	A2	A1

(разряды C1–C4 используются при работе с несколькими крейтами одновременно), и затем он же используется для приема/передачи слова данных.

Таким образом, работа с КАМАК представляет собой процессы записи или чтения данных из соответствующих регистров.

Чтение данных из КАМАК:

1. Если операция вызвана прерыванием – установить, какая станция запросила прерывание (по содержимому DMR).

2. Установить одну из функций чтения в младших битах CSR.

3. В четвертом регистре указать, к какой станции относится команда.

4. Прочитать поступившие данные.

Запись данных в КАМАК:

1. Проверить готовность КАМАК к принятию данных (бит Q в CSR должен быть равен нулю, т. е. двоичное число, содержащееся в этом регистре – положительно).

2. Установить одну из функций записи в младших битах CSR.

3. В четвертом регистре указать, к какой станции относится команда.

4. Передать данные в четвертый регистр (при необходимости и в DHR).

Управление крейтом КАМАК от IBM PC.
Параллельный интерфейс

В качестве примера интерфейса параллельного доступа рассмотрим плату «GEOSOFT».

Контроллер крейта также подключается к шине ISA или EISA IBM PC. Он имеет двенадцать восьмиразрядных регистров интерфейса ввода-вывода.

Линия адреса ввода/вывода соответствует настройке программных средств на диапазон адресов 240_{16} – $24F_{16}$. Назначение регистров приведено в табл. 12.

Разряды AX1–AX4 используются для указания адреса крейта в системах с несколькими последовательно соединенными крейтами, регистр $24C_{16}$ – для контроля состояния интерфейсной платы, $24F_{16}$ – для организации работы режима прямого доступа к памяти. В остальном работа этого контроллера тоже осуществляется аналогично. Программный модуль, реализующий управление этим контролером крейта КАМАК, приведен в Приложении 1.

Таблица 12. Назначение регистров параллельного интерфейса КАМАК

Регистр	Адрес	Разряды								Назначение
		7	6	5	4	3	2	1	0	
0	240_{16}	W24	W24	W22	W21	W20	W19	W18	W17	Запись
1	241_{16}	W16	W15	W14	W13	W12	W11	W10	W9	Запись
2	242_{16}	W8	W7	W6	W5	W4	W3	W2	W1	Запись
3	243_{16}					A8	A4	A2	A1	Субадрес станции
4	244_{16}				F16	F8	F4	F2	F1	Функция
5	245_{16}				N16	N8	N4	N2	N1	Адрес станции
6	246_{16}		AX4	AX3	AX2	AX1	I	C	Z	Адрес крейта и состояние
7	247_{16}									не используется
8	248_{16}	LAM	L16	L8	L4	L2	L1	X	Q	LAM запросы и состояние
9	249_{16}	R24	R23	R22	R21	R20	R19	R18	R17	Чтение
10	$24A_{16}$	R16	R15	R14	R13	R12	R11	R10	R9	Чтение
11	$24B_{16}$	R8	R7	R6	R5	R4	R3	R2	R1	Чтение
12	$24C_{16}$						AO	AL	AC	Состояние контроллера
15	$24F_{16}$									DMA

СИСТЕМА PXI

Модульная система интерфейсов PXI (PCI extension for instrumentation, ранее – CompactPCI) была разработана компанией National Instruments с целью объединить знакомое пользователю и потенциальным разработчикам программное и аппаратное обеспечение с разработанными в последние годы технологиями. В качестве механического стандарта использован стандарт интерфейса Евромеханика (EuroCard), разработанного для применений в промышленных условиях, – это обеспечивает механическую надежность системы и устойчивость электрических соединений. Обмен информацией между интерфейсами осуществляется по шине, аналогичной шине PCI с расширенной адресацией. Это дает возможность установки до семи станций в крейте и допускает дальнейшее расширение с использованием стандартных PCI–PCI мостов. PXI является развитием разработанной ранее системы CompactPCI и полностью с ней совместима.

Использование стандартной шины PCI дает возможность широкого использования существующего программного обеспечения – начиная от средств операционных систем до драйверов устройств и программ обработки данных и графического представления результатов. PXI легко интегрируется в операционные системы Windows 95, Windows 98, Windows NT4, Windows 2000, Windows XP. Для нее разработана архитектура VISA (Virtual Instrument Software Architecture), позволяющая объединять крейты PXI с модульными системами VXI и GPIB.

На рис. 5 схематично показан вид крейта PXI. Крейт представляет собой шасси, в задней части которого расположены разъемы для подключения станций – контроллера и интерфейсов внешних устройств. Контроллер устанавливается в слот № 1; справа от него располагаются слоты для подклю-

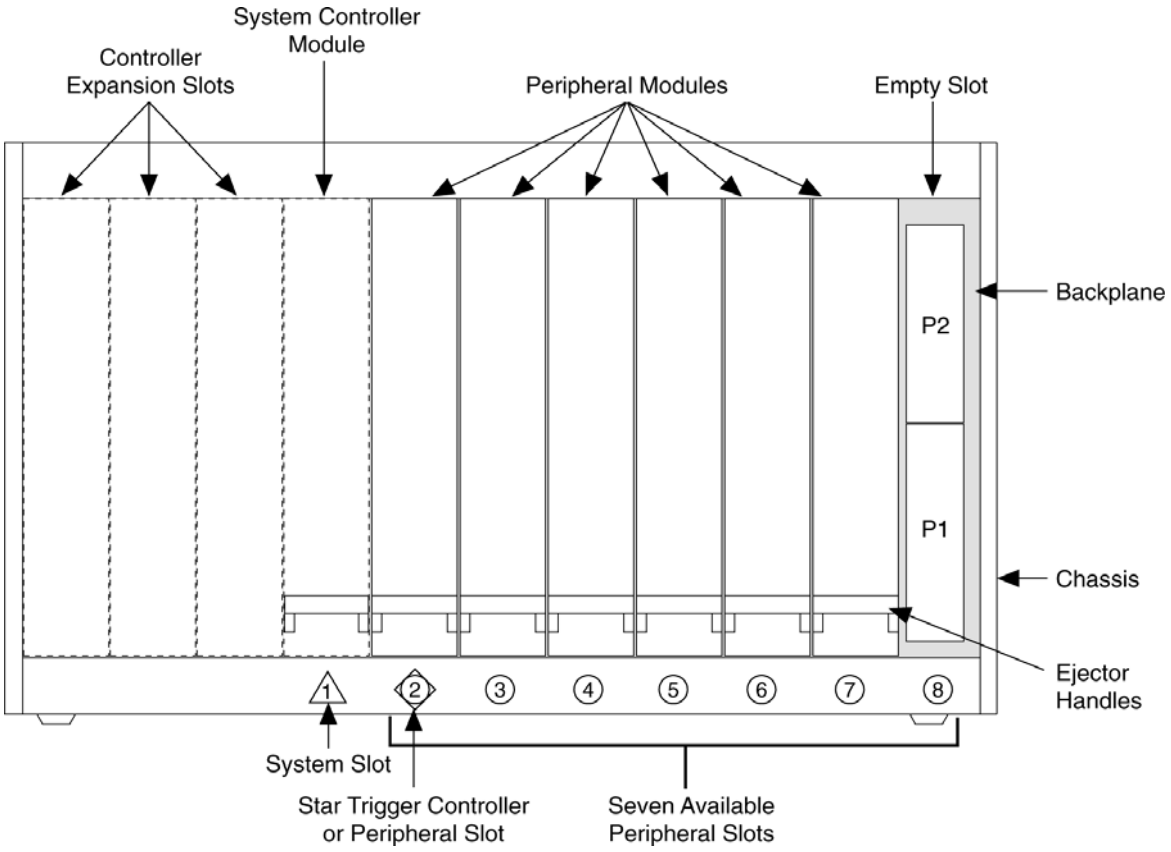


Рис. 5. Контроллер PXI – 33 МГц.

чения интерфейсных модулей (до семи для крейтов с тактовой частотой 33 МГц, и до 3 – для 66 МГц), слева могут располагаться дополнительные слоты для установки дополнительных плат-расширений контроллера.

Механический стандарт

Стандарт PXI допускает два типоразмера интерфейсных модулей (показаны на рис. 6) – формат 3U, размером 100×160 мм с двумя разъемами для подключения к шине крейта (J1 и J2), и формат 6U, размером 233.35×160 мм, с четырьмя разъемами. Разъем J1 полностью аналогичен стандартному слоту PCI-32, разъем J2 содержит контакты, расширяющие шину данных PCI до 64 бит, и выполняющие дополнительные функции PXI, назначение контактов двух других разъемов не стандартизовано.

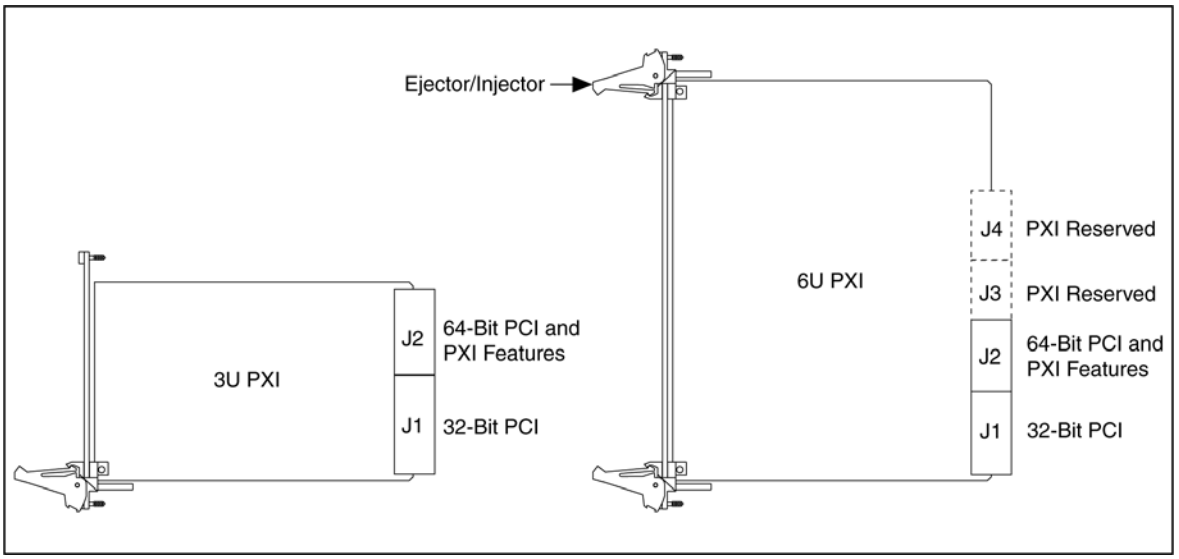


Рис. 6. Типоразмеры плат PXI.

Электрический стандарт

Основные электрические характеристики стандарта PXI:

Тактовые частоты 33/66 МГц.

Возможность подключения до семи (33 МГц) или четырех (66 МГц) интерфейсных модулей в одном крейте.

Передача 32- и 64-битных слов данных.

Пиковая скорость обмена от 132 Мб/с (32 бита, 33 МГц) до 528 Мб/с (64 бита, 66 МГц).

Возможность подключения дополнительных крейтов через PCI-PCI мост.

Линии питания 3.3 В.

Поддержка режима конфигурации Plug&Play.

В дополнение к обычным линиям PCI шины, стандарт PXI включает дополнительные линии, образующие локальную шину PXI. Каждый из интерфейсных модулей соединен 13 линиями этой шины со своим правым и левым соседом в крейте; кроме этого, ближайший к контроллеру крейта слот (№ 2) звездообразно соединяется с остальными шестью (рис. 7). 13 линий локальной шины могут использоваться как для передачи цифровых сигналов в стандарте TTL, так и для аналоговых сигналов напряжением до 42 В, что определяется конструкцией отдельных модулей и должно учитываться при разработке соответствующего программного обеспечения.

Контроллер PXI имеет встроенный таймер с тактовой частотой 10 МГц, который используется для синхронизации процессов в интерфейсных модулях.

Наличие звездообразного соединения модулей (триггерлинии, аналог линий **L** системы КАМАК), как и в КАМАКе,

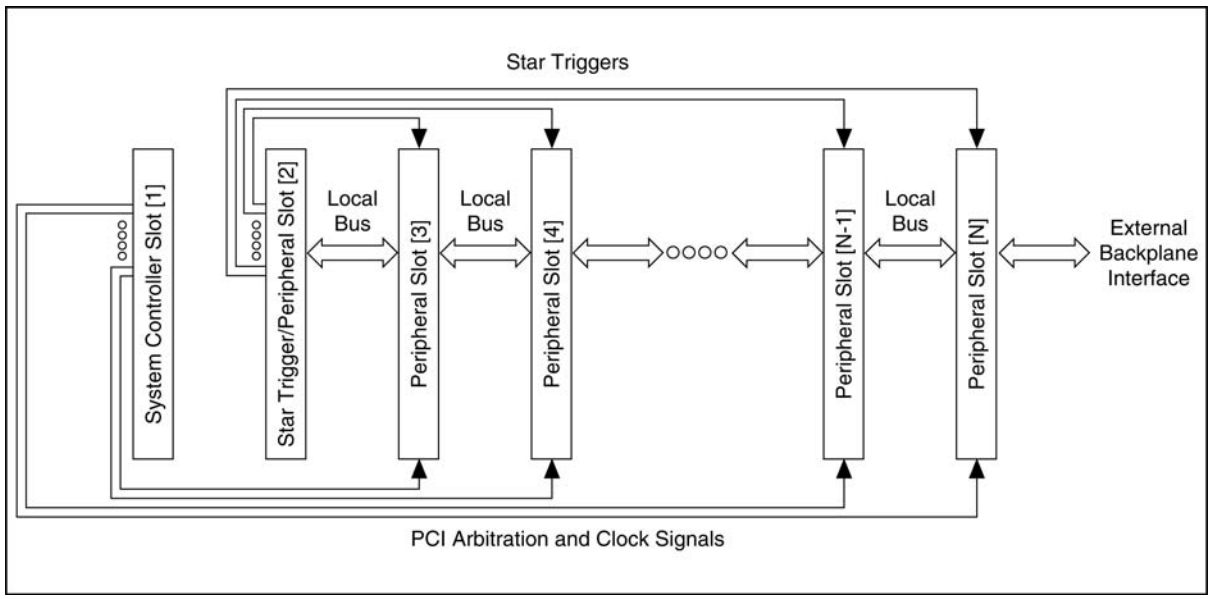


Рис. 7. Локальная шина PXI.

позволяет передавать информацию непосредственно от интерфейсного модуля к контроллеру. Эти же линии могут использоваться для синхронизации процессов в интерфейсных модулях. При наличии специализированного модуля – контроллера триггеров возможна передача сигналов по триггерлиниям от одного модуля к другому, минуя контроллер, и, таким образом, организация достаточно сложных последовательностей операций, выполняемых интерфейсными модулями без участия контроллера крейта и управляющей ЭВМ.

Как компанией-разработчиком, так и другими фирмами (этому способствует то, что PXI является открытой архитектурой) выпускается широкий ассортимент крейтов, контроллеров и интерфейсных плат этого стандарта, назначение которых аналогично интерфейсам КАМАК; в то же время более высокие требования к электрическим и механическим параметрам компонент приводят к заметно более высоким ценам этих устройств.

СИСТЕМА VXI

Модульная система VXI (VME Extensions for Instrumentation), аналогично PXI, основана на использовании шины, первоначально разработанной для обмена данными между компонентами компьютера. В данном случае использована шина VMEbus, разработанная совместно компаниями Motorola, Mostek и Signetics. Шина и весь стандарт разрабатывались для применений в промышленных управляющих компьютерах, поэтому, подобно PXI, с самого начала были ориентированы на работу в жестких эксплуатационных условиях. В качестве механического стандарта использован тот же формат Евромеханика (EuroCard). Первая версия этой шины, известная также как VERSabus, была использована Motorola Corporation в 1979 году при создании промышленных многопроцессорных систем на базе микропроцессора Motorola 68000. Разработка оказалась весьма удачной, главным образом, за счет того, что имела большой запас ресурсов: не была привязана к одному типу процессоров, допускала существенное

расширение разрядности данных, повышение рабочих частот; кроме того, благодаря открытой архитектуре, ее использовало большое число разработчиков промышленной электроники. В результате доработок и совершенствования архитектуры шины наиболее распространенный сейчас стандарт VME64 поддерживает обмен 64-разрядными данными с со скоростью 80 Мб/с (160 Мб/с для принятого недавно VME64x, свыше 500 Мб/с для разрабатываемого VME320), семь уровней внутренних прерываний, до 21 процессора, архитектуру Plug&Play.

В дополнение к этому в системах VXI (аналогично PXI) добавлено 10 триггерных линий для синхронизации процессов, выполняемых отдельными модулями, введена 12-разрядная внутренняя шина обмена данными между модулями. В целом система во многом аналогична PXI, но предполагает использование более высокоинтеллектуальных модулей, как правило, включающих собственный управляющий процессор. IBM-совместимые контроллеры VXI, по сути, сами являются достаточно мощными компьютерами – они включают процессор семейства Pentium, жесткий диск, оперативную память, допускают прямое подключение монитора и клавиатуры и вполне могут выполнять функции управляющей ЭВМ. Все это, несомненно, расширяет функциональные возможности систем этого стандарта, но значительно увеличивает их стоимость.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ АВТОМАТИЗАЦИИ ЭКСПЕРИМЕНТА

ТРЕБОВАНИЯ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ УПРАВЛЕНИЯ И ОБРАБОТКИ ДАННЫХ ЭКСПЕРИМЕНТА

Программное обеспечение, предназначенное для управления экспериментальными установками и оперативной обработки данных, должно отвечать специфическим требованиям, которые определяются особенностями его разработки и использования.

Как правило, при разработке автоматизированных экспериментальных установок используются компьютеры ограниченной мощности (относительно медленные процессоры, малые объемы оперативной и долговременной памяти). Это приводит к необходимости максимальной экономии этих ресурсов, то есть использования алгоритмов, способных производить по крайней мере предварительную обработку данных непосредственно в ходе эксперимента, при минимальных требованиях к памяти.

Как правило, программное обеспечение разрабатывается самим пользователем – экспериментатором и затем определенное (достаточно долгое) время используется им без изменений. Но рано или поздно наступает момент, когда экспериментальная установка требует более или менее серьезной модификации – а вместе с ней и ее программное обеспечение. Для облегчения решения этой задачи (иногда – для того, чтобы ее вообще возможно было решить, не переписывая все программное обеспечение заново) необходимо иметь подробно документированные, прокомментированные исходные тексты программ.

Как правило, экспериментальная установка эксплуатируется несколькими операторами (по крайней мере, таковы требования техники безопасности). Это означает, что пользо-

ваться разработанными программами будет не только их автор, но и другие лица. Отсюда возникает необходимость создания максимально дружественного, интуитивного интерфейса программы. В любом случае должна быть обеспечена гарантия того, что полученные в результате эксперимента (долговременного или уникального) данные не будут уничтожены из-за случайного ошибочного действия оператора.

Операционные системы автоматизированных установок

WINDOWS 95 – WINDOWS 98

Операционные системы Windows 95/98 являются логическим продолжением развития линии Windows 3.11. Они обладают несколько большей устойчивостью, вытесняющей многозадачностью для win32 приложений, потоками, файлами, отображаемыми в память, поддерживают длинные имена файлов. В то же время, хотя они и являются 32-разрядными операционными системами, их архитектура во многом схожа с архитектурой Windows 3.11; более того, их многие важные компоненты по-прежнему остались 16-разрядными. Каждый 32-разрядный процесс исполняется в отдельном адресном пространстве, однако 16-разрядные Windows-приложения исполняются на одной системной виртуальной машине. Как следствие, некорректное 16-разрядное приложение (например, написанное с использованием драйверов, ориентированных на DOS) может привести к краху всей системы. Для более полной совместимости с DOS и ускорения работы здесь упрощен доступ к портам ввода-вывода. Порты доступны для непосредственной записи и чтения программами пользовательского уровня привилегий. Если же вы планируете одновременное использование вашего нестандартного оборудования несколькими процессами, вам придется написать vxd драйвер, который будет управлять доступом к устройству. Vxd драйверы разрабатываются с использованием DDK на языке Assembler.

Распространенность этих систем и простота доступа к нестандартному оборудованию делает привлекательным этот выбор операционных систем как платформы для автоматизации, однако при этом следует учесть их низкую надежность. Кроме этого, неравномерность и неуправляемость разделения квантов процессорного времени между процессами ведет к непредсказуемости времени отклика такой системы на внешние события. Таким образом, Windows 95/98 могут быть применены как OS для управляющей программы только тогда, когда не требуется высокая надежность и быстрая (быстрее десятков миллисекунд), стабильная реакция системы на внешние события.

WINDOWS NT – WINDOWS 2000 – WINDOWS XP

Windows NT задумывалась как операционная система для серверов и рабочих станций. Главное требование к такой системе: надежность. Архитектуру Windows NT и более поздних, созданных на ее основе операционных систем Windows 2000 и Windows XP можно охарактеризовать эпитетами: модульная, объектная, масштабируемая, расширяемая. В архитектуре системы можно выделить две части. Одна часть работает в режиме ядра, обеспечивает уровень абстракции от оборудования, управляет разделяемыми ресурсами, представленными как объекты. Это NT executive. Другая часть исполняется в режиме пользователя и представляет собой серверы подсистем. Каждая подсистема исполняется в своем адресном пространстве. Подсистема связывается с уровнем NT executive посредством механизма LPC (Local Procedure Call). Программа пользователя для связи с подсистемами использует вызовы функций соответствующего API. Эти функции скрывают LPC вызовы от программы пользователя. Части системы максимально изолированы и их взаимодействие четко определено. Доступ к оборудованию возможен только на уровне ядра. Таким образом, если вам необходим доступ к оборудованию, вам придется написать драйвер уровня ядра (kernel mode driver). Драйверы в NT разрабатываются на языке C (C++) с использованием

заголовочных файлов, поставляемых в составе DDK. Хотя Windows NT не является системой реального времени, она предоставляет средства управления приоритетами процессов и, по сравнению с Windows 95/98, лучше подходит в качестве платформы для управляющей программы и может быть рекомендована для задач, требующих высокой надежности, но не требующих быстрого и стабильного времени реакции системы. Так как драйверы NT пишутся на языке высокого уровня, можно относительно легко перенести большую часть логики управления оборудованием в драйвер, оставив на пользовательском уровне только интерфейс пользователя. Для многих задач полученного в результате повышения скорости и стабильности реакции системы может оказаться достаточно.

LINUX

В основе ОС LINUX лежит концепция процесса – единицы управления и единицы потребления ресурсов. Процесс представляет собой программу в состоянии выполнения. Процессы могут выполняться в двух режимах: системном и пользовательском. В системном режиме осуществляется управление процессами, доступ к оборудованию, управление памятью. Каждый процесс работает в отдельном адресном пространстве. Связь с ядром осуществляется посредством системных вызовов. Программа пользователя практически никогда не использует системные вызовы непосредственно. Вместо этого она вызывает функции из системной библиотеки libc. Процесс, осуществляющий системный вызов, становится системным на время вызова; каждый процесс может находиться и в пользовательской, и в системной фазе. Когда процесс находится в системной фазе, он не может быть вытеснен другим процессом. Таким образом, Linux, как и Windows, не является системой реального времени, то есть не может обеспечить короткое стабильное время реакции системы на внешнее событие. Однако существуют модификации Linux, являющиеся системами реального времени, например, RTLinux. RTLinux содержит маленькое ядро-планировщик, которое запускает

традиционное ядро как задачу с наименьшим приоритетом. Задачи реального времени представляют собой загружаемые модули ядра. Связь задачи реального времени и остальных процессов Linux осуществляется через буферы fifo. Процесс реального времени имеет полный доступ ко всему оборудованию. Такой подход позволяет получить систему реального времени и старый добрый Linux в одном флаконе. Скромные требования к ресурсам позволят обойтись более медленным процессором и маленьким жестким диском или даже его flash аналогом. Минимальная версия с поддержкой сети требует 8 Мб оперативной памяти и 1.4–2 Мб на жестком диске, дискете или flash. Такие характеристики делают RTLinux привлекательной операционной системой для встроенных применений. Выгодным представляется так же распределенное решение, состоящее из относительно маломощного компьютера под управлением RTLinux, осуществляющего все управление в режиме реального времени, и системы на основе Windows, посылающей по сети высокоуровневые команды для RTLinux системы.

Особенности метода объектно-ориентированного программирования

Метод объектно-ориентированного программирования широко применяется в практике создания программного обеспечения. Большинство реальных задач слишком сложны, чтобы реализовать их в виде линейной программы. Если в задаче есть подзадачи, выполняющиеся более одного раза, или просто некие выделенные этапы, то их целесообразно выделить в подпрограммы. Применение подпрограмм позволяет структурировать задачу, увидеть ее в более общем плане, не вдаваясь в детали реализации. Процедура (подпрограмма) принимает данные и делает с ними необходимую работу. Такой способ дробления большой задачи называется процедурно-ориентированным программированием.

Не менее часто возникает ситуация, когда группа данных представляет собой логическое целое. Например, четыре числа задают координаты прямоугольника. Если программа оперирует прямоугольниками, то такие четверки чисел в большинстве случаев обрабатываются группами. Предположим, у нас имеется процедура, выполняющая некоторую операцию с прямоугольником, в нее передаются четыре числа, задающие координаты его вершин. Для компилятора эти числа никак не связаны между собой. Возникает возможность по ошибке передать в процедуру неверные данные, которые в принципе не могут быть такими координатами. Решением этой проблемы является применение структур: групп данных, представляющих собой логическое целое. Структуры позволяют объединить логически связанные данные вместе.

Итак, у нас есть структуры и процедуры, работающие с ними, принимающие их в качестве исходных параметров. Логическое объединение данных и методов их обработки в единое целое – и есть основа объекта. Разбиение задачи на взаимодействующие объекты – есть объектно-ориентированное программирование. Идея объекта заключается в объединении данных и методов их обработки; однако это – не просто сумма. Часть данных и методов скрыта от других объектов и доступна только изнутри объекта. Таким образом, объект предоставляет внешнему миру небольшой набор методов, некий интерфейс работы с ним. Все внешние программные модули, использующие объект, знают только этот интерфейс и не знают деталей его внутренней реализации. Такое свойство объекта называется *инкапсуляцией*.

Представим, например, объект, описывающий время. Объект time умеет выдавать дату, время, день недели, год. При этом для всей программы, использующей объекты time, совершенно безразлично, как конкретно time хранит время: в секундах от первого января 1970 г. либо год, месяц, день недели и время в виде отдельных чисел. Можно сколько угодно менять внутреннюю реализацию time без переделки (требуется только перекомпиляция) частей программы, его использующих.

Предположим, мы разрабатываем говорящие часы. Нам необходим объект, обладающий теми же свойствами, что и time, но вдобавок способный проговаривать время. Разрабатывая новый объект talking_time, мы хотим повторить функциональность time. Чтобы не повторять детали реализации и интерфейса, воспользуемся *наследованием*. Унаследуем talking_time от time и добавим метод talk и связанные с ним дополнительные поля данных. Нам необходимо написать только реализацию нового метода, остальные методы будут взяты из time. Кроме добавления новых методов, в объекте-наследнике можно переопределить реализацию какого-либо метода объекта предка. Помимо использования реализации и интерфейса time в talking_time мы посредством наследования явно выразили общность понятий. Теперь произведем из talking_time объект talking_time_ru и talking_time_eng, изменив реализацию метода talk так, чтобы время произносилось на соответствующем языке. Предположим, какая-то часть нашей программы имеет переменную time_ptr типа указатель на talking_time, тогда мы можем присвоить переменной time_ptr как адрес экземпляра объекта типа talking_time_eng, так и адрес экземпляра объекта типа talking_time_ru, так как все эти объекты произведены из talking_time.

Что произойдет, если обратиться по адресу, лежащему в переменной time_ptr и вызвать метод talk? Возможны два варианта. Так как time_ptr имеет тип указатель на talking_time, то исполнится метод talk от talking_time. Так как на самом деле в переменной time_ptr лежит адрес экземпляра объекта типа, например, talking_time_ru, то исполнится метод talk от объекта talking_time_ru. Во втором случае time_ptr ведет себя *полиморфно*. Чтобы реализовать такое поведение, необходимо объявить метод talk виртуальным. Виртуальные методы позволяют разрабатывать абстрактные объекты и на их основе организовывать взаимодействие частей программы между собой. Абстрактный класс содержит объявления каких-либо виртуальных методов без их реализации. Нельзя создать экземпляр абстрактного класса, но можно создать переменную типа указатель на абст-

рактный объект и присвоить ей адрес экземпляра какого либо объекта, произведенного от абстрактного объекта.

Язык объектно-ориентированного программирования C++

Описанные особенности объектно-ориентированного программирования не связаны с конкретной реализацией объектной модели в том или ином языке программирования, а присущи объектно-ориентированному подходу как таковому. Рассмотрим, как этот подход реализуется в языке C++.

Программируя на C++, вы встретите как минимум два типа файлов: файлы заголовков (обычно с расширением hpp или h) и файлы реализаций (обычно с расширением cpp или c). В заголовочных файлах пишутся определения, а в файлах реализации пишется реализация. В принципе, все можно написать и в одном файле, но это очень нерационально. Совокупность заголовочных файлов и файлов реализации совместно с опциями их компиляции и сборки образует проект. Компиляция файла реализации состоит из двух этапов: исполнения директив препроцессора и собственно компиляции. Директивы препроцессора начинаются со знака #. Самая часто используемая директива – это директива include.

```
#include ”my_header.hpp”
```

Действие директивы эквивалентно вставке содержания файла my_header.hpp в месте расположения директивы. Часто используется директива define.

```
#define ERROR_CODE 10
```

Действие директивы состоит в замене строки ERROR_CODE на строку 10. Директивы ifdef, ifndef, endif позволяют осуществлять условную компиляцию:

```
#ifdef DEBUG
// дополнительные проверки
. . .
// конец дополнительных проверок
```

```
#endif
```

Или предотвратить препроцессор от многократного просмотра заголовочного файла:

```
#ifndef MY_HEADER_HPP
#define MY_HEADER_HPP
// начало определений
. . . .
// конец определений
#endif
```

В C++ объекты называются классами. В определении класса используется ключевое слово class. Ключевые слова public, protected, private служат для управления инкапсуляцией. Функции и поля класса, объявленные в разделе public, доступны для других классов; все, что объявлено в private, доступно только из функций данного класса и недоступно другим классам; все, что объявлено в protected, недоступно другим классам, но доступно для классов, произведенных от данного класса.

```
class my_class: public my_class_parent
{
public:
  my_class();
  my_class(const my_class &mc);
  ~my_class();
  my_class &operator=( const my_class &mc);
protected:
. . .
        private:
. . .
        };
```

Наследование указывается сразу после имени класса. В примере класс my_class унаследован от класса my_class_parent с атрибутом public. Это означает, что все доступные функции

класса my_class_parent имеют ту же доступность, как если бы они были объявлены в разделе public класса my_class. Как уже отмечалось, класс – это совокупность данных и методов их обработки. При создании экземпляра класса необходимо произвести начальную инициализацию данных. Для этого служит группа функций класса, называемых конструкторами. Конструктор имеет то же имя, что и класс. Конструктор без параметров называется *конструктор по умолчанию*. Конструктор, принимающий константную ссылку на другой объект того же типа, называется *конструктором копии*. При необходимости можно определить и другие конструкторы. Конструктор по умолчанию и конструктор копии должны быть всегда. Если конструктор не определен, компилятор пытается создать его сам. Настоятельно рекомендуется всегда в явном виде определять конструкторы, так как конструкторы, самостоятельно созданные компилятором, часто работают не так, как требуется разработчику. При разрушении экземпляра класса вызывается функция, имеющая название в виде имени класса с префиксом ~. Это – *деструктор*. Все конструкторы имеют одинаковое имя, но разные параметры. Наличие нескольких функций, отличающихся параметрами, называется *перегрузкой функций*. Перегрузка функций позволяет более прозрачно организовать программу; функции, делающие в принципе одно и то же, но с разными типами данных, удобно и называть одинаково. Работая с другими языками программирования, вы, возможно, привыкли считать значок, например +, частью синтаксиса языка, обозначающей операцию сложения. На самом деле это вызов функции, принимающей два аргумента – слагаемых и возвращающей значение суммы. В языке C++ такая функция доступна в явном виде. Это – функция operator+. Операторы, как и другие функции, можно перегружать. Определим класс матриц matrix. Определим функцию:

```
matrix operator+(const matrix &m1, const matrix &m2);
```

и напишем:

```
matrix a;
matrix b;
matrix c=a+b;
```

Теперь мы можем складывать экземпляры класса matrix, просто написав +. Используя перегрузку оператора присваивания, мы можем в явном виде указать, как производить присваивание одного экземпляра объекта другому. По умолчанию компилятор сам создает оператор присваивания из конструктора копии, но рекомендуется определить его в явном виде. Для объекта my_class оператор присваивания выглядит следующим образом:

```
my_class &operator=(const my_class &mc);
```

Обычно последней строкой реализации оператора присваивания будет:

```
return *this;
```

Оператор this служит для обозначения указателя на данный экземпляр класса. Оператор const обозначает, что мы не будем вызывать неконстантные функции у mc , то есть не сможем менять значения полей mc. По умолчанию функции класса неконстантные. Чтобы объявить константную функцию класса, необходимо добавить const после объявления функции:

```
class my_class: public my_class_parent
{
 public:
 . . .
 . . .
  bool is_ready() const;
 . . .
 . . .
 protected:
```

```
 private:
};
```

Функция is_ready не может менять значения полей объекта. Если в функции класса не планируется изменять поля класса, то следует обязательно объявлять ее с атрибутом const. Это позволит избежать ошибочных изменений данных.

Разбивая задачу на функции, не следует забывать, что есть определенные расходы процессорного времени на осуществление вызова функции. Если функция небольшая, то расходы на вызов сопоставимы со временем выполнения функции. Как же быть? Смириться с потерей производительности или пожертвовать стройностью организации программы, поставив под удар ее надежность? Идеальное решение – сохранить и первое и второе. Язык C++ поддерживает так называемые inline функции. Встраиваемые функции отличаются от обычных тем, что компилятор вместо вызова функции вставляет ее код. Это позволяет избежать накладных расходов на вызов функции, сохранив стройность организации программы. Встраиваемыми могут быть как обыкновенные функции, так и функции класса. Реализация встраиваемых функций пишется в заголовочном файле. Встраиваемость задается явно атрибутом inline или неявно, посредством написания реализации непосредственно внутри объявления класса:

```
class my_class: public my_class_parent
{
public:
 . . .
 . . .
 bool is_equal();
 . . .
 . . .
protected:
private:
 int _a;
 int _b;
```

```
};

inline bool my_class::is_equal()
{
  return a==b;
};
```

или неявно

```
class my_class: public my_class_parent
{
public:
 . . .
 . . .
 bool is_equal()
{
  return a==b;
};
 . . .
 . . .
protected:
private:
 int _a;
 int _b;
};
```

Часто необходимо производить одни и те же действия, но над разными типами данных. Например, необходимо написать функцию сортировки массива целых чисел, а потом функцию сортировки массива чисел с плавающей точкой, а потом функцию сортировки массива объектов типа my_class. При решении этой задачи вы обнаружите, что пишите одно и то же, заменяя кое-где int на double, а потом double на my_class. Язык C++ позволяет написать такую функцию один раз сразу для всех типов данных. Для этого нужно применить функцию-шаблон. Шаблон описывается в заголовочном файле и объявляется с помощью ключевого слова template. В заголовке шаблона указываются формальные названия типов данных. Описывая реализацию шаблона, вы используете формальные

типы данных. Когда компилятор встречает вызов функции-шаблона, он заменяет формальные типы данных на фактические типы данных и компилирует необходимый код:

```
template <class T> T
        find_min(const T &arg1, const T &arg2)
{
  if (arg1 < arg2)
    {return arg1;}
  return arg2;
}
```

Шаблон find_min умеет находить минимум из двух аргументов любого типа данных, для которого определен operator<. C++ позволяет писать не только шаблоны функций, но и шаблоны классов:

```
template <class T> class rectangle
{
 public:
  ...
  T square()
  {
    return _width*_height;
  }
  ...
 private:
  T _width;
  T _height;
};
```

Чтобы использовать шаблон класса, необходимо указать фактические имена типов при объявлении типа переменной:

```
rectangle<int>      int_rect;
rectangle<double>   double_rect;
rectangle<my_class> my_class_rect;
```

Шаблоны позволяют написать реализацию алгоритма в чистом виде, безотносительно к конкретному типу данных,

причем обобщение алгоритма не вызывает потери производительности исполнения. Все это делает их идеальным средством написания библиотек. В настоящее время современный компилятор C++ поставляется с стандартной библиотекой шаблонов. Стандартная библиотека шаблонов включает в себя библиотеку алгоритмов, библиотеку функций и библиотеку контейнеров. Библиотека содержит быстрые и удобные решения задач, наиболее часто стоящих перед программистом. Для доступа к данным стандартная библиотека шаблонов использует *итераторы*. По сути итератор – это обобщенный указатель. Итератор представляет собой класс, у которого перегружен operator*. В случае константного итератора он возвращает константную ссылку на класс, а в случае не константного итератора – не константную ссылку на класс. Аналогично указателю, итератор имеет перегруженные операторы operator++, operator— и operator==. Итератор случайного доступа имеет к тому же перегруженный operator+. Итак, * возвращает данные, ++ и — передвигают итератор на следующий (предыдущий) элемент последовательности данных. Все элементы стандартной библиотеки шаблонов принадлежат пространству имен std. Чтобы компилятор мог находить элементы библиотеки, необходимо, помимо включения директивой include файлов библиотеки, либо указать на использование пространства имен std директивой using namespace, либо каждый раз явно указывать пространство имен. Объявим переменную a типа вектор целых чисел:

```
#include <vector>
using namespace std;
. . .
vector<int> a;
```

или

```
#include <vector>
. . .
std::vector<int> a;
```

Итераторы объявлены внутри объявления соответствующего контейнера, поэтому, чтобы объявить переменную типа итератор, необходимо указать класс, в котором этот итератор объявлен. Объявим переменную it и it_const:

```
vector<int>::iterator it; // неконст. итератор вектора
                          // целых чисел
. . .
vector<int>::const_iterator it_const; // конст. итератор
                                      // вектора целых
```

Библиотека содержит ряд очень удобных контейнерных классов. Разные контейнеры по-разному хранят данные, обладают своими особенностями, но идеология работы с ними общая. Классы, хранимые в контейнерах, должны удовлетворять ряду условий. Должны быть определены конструктор по умолчанию, конструктор копии, оператор присваивания, оператор сравнения и, в некоторых случаях, оператор меньше и оператор больше. Контейнеры, хранящие данные в несортированном порядке, для добавления данных в начало списка используют метод push_front, а для добавления в конец списка – метод push_back. Метод erase позволяет удалить элемент по указанному итератору. Метод size возвращает количество элементов, хранящихся в контейнере. Метод empty позволяет узнать, пуст ли контейнер. Метод clear стирает данные и делает контейнер пустым. Метод begin возвращает итератор на первый элемент последовательности данных. Метод end возвращает итератор, обозначающий конец последовательности. В случае не сортирующего контейнера метод insert позволяет вставить новый элемент в определенную позицию последовательности данных, а в случае сортирующего контейнера – добавить данные. Перечисленные методы являются основными и составляют скелет работы с контейнерами. Теперь рассмотрим особенности и область применения различных контейнеров.

Контейнер вектор (vector) аналогичен массиву. Данные в векторе лежат в одной области памяти, подряд одно за другим. Вставка в середину последовательности или удаление из середины приводит к копированию элементов последовательности. Время вставки возрастает с приближением к началу последовательности. Если при добавлении элементов превышена емкость вектора, происходит выделение новой области памяти, копирование туда старых данных, добавление нового элемента. При изменении вектор может временно потребить примерно в два раза больше памяти, чем ему действительно нужно. При выделении памяти итераторы вектора становятся некорректными. Вектор имеет перегруженный operator[ ] и итераторы случайного доступа. Это позволяет получить быстрый доступ к любому элементу последовательности. Метод reserv позволяет заранее выделять память для нужного количества элементов, избегая лишних выделений памяти. При удалении элемента память, занимаемая вектором, не освобождается. Вектор применяется тогда, когда предполагаются нечастые вставки и удаления, и необходима высокая скорость доступа к любому элементу последовательности. Для использования вектора необходимо включить заголовочный файл vector.

Список (list) представляет собой совокупность структур. Структура содержит значение и указатели на предыдущую и последующую структуру. Следовательно, доступ к данным осуществляется последовательно. Для добавления или удаления элемента в середине, начале или конце последовательности требуется одинаково небольшое время. При удалении элемента делается некорректным только итератор, указывающий на этот элемент. Списки применяются в тех случаях, когда часто изменяется длина последовательности данных, происходят частые удаления и вставки в середину и начало последовательности. Для использования списка необходимо включить заголовочный файл list.

Множество (set) представляет собой контейнер на основе бинарного дерева. Множество содержит данные в сортиро-

ванном виде и предоставляет быстрый поиск элемента с помощью метода find. По работе с памятью множество аналогично списку. Множество не может содержать одинаковых элементов. Однако разновидность множества мультимножество может содержать одинаковые элементы. Применяйте множество, когда необходим быстрый поиск элемента. Для использования множества необходимо включить заголовочный файл set или multiset.

Карта (map) представляет собой множество пар ключ – значение. Карта позволяет поставить значение в соответствие ключу и обеспечивает быстрый поиск по ключу. Карта имеет перегруженный operator[ ]. Это позволяет использовать ключ словно индекс массива. Карта не может содержать пары с одинаковым ключом, но мульти-карта может. Применяйте карту, когда необходимо отношение ключ – значение, сортировка и быстрый поиск по ключу. Для использования карт необходимо включить заголовочный файл map или multimap.

Стек (stack) работает по принципу вошел последним – вышел первым. Стек может быть реализован как на базе вектора, так и на базе списка. Метод top служит для доступа к вершине стека. Метод pop выталкивает верхний элемент. Метод push добавляет элемент в вершину стека. Для использования стека необходимо включить заголовочный файл stack.

Очередь (queue) работает по принципу первый вошел – первый вышел. Метод push добавляет элемент в конец очереди. Остальное – как в стеке. Для использования очереди необходимо включить заголовочный файл queue.

Приоритетная очередь отличается от очереди тем, что элементы внутри этого контейнера расположены не в порядке их поступления в очередь, а в порядке сортировки.

Для решения стандартных задач обработки данных, хранящихся в контейнерных классах, применяются стандартные алгоритмы, список которых приведен в табл. 13.

Таблица 13. Стандартные алгоритмы обработки данных

Алгоритмы инициализации последовательностей	
Fill	Заполняет последовательность начальными значениями
fill_n	Заполняет начальными значениями последовательность *n* позиций
copy	Копирует последовательность в другую последовательность
copy_backward	Копирует последовательность в другую последовательность
generate	Инициализирует последовательность с использованием генератора
generate_n	Инициализирует *n* позиций с использованием генератора
swap_ranges	Меняет местами значения двух последовательностей
Алгоритмы поиска	
Find	Находит элемент, равный заданному
find_if	Находит элемент, удовлетворяющий условию
Adjacent_find	Находит последовательно повторяющиеся элементы
Search	Находит подпоследовательность в последовательности
max_element	Находит максимальное значение
min_element	Находит минимальное значение
Mismatch	Находит первое отличие в двух последовательностях
Преобразования	
Reverse	Изменяет порядок элементов на обратный
Replace	Заменяет одно значение другим
replace_if	Заменяет одно значение другим при выполнении условия
Rotate	Циклически переставляет элементы последовательности
Partition	Разбивает элементы на две группы
stable_partition	Разбивает элементы на две группы с сохранением порядка
next_permutation	Осуществляет перестановку
prev_permutation	Осуществляет перестановку
Inplace_merge	Соединяет две смежных последовательности в одну
Random_shuffle	Случайным образом перемешивает элементы последовательности
Алгоритмы удаления	
Remove	Удаляет элемент, удовлетворяющий условию
Unique	Удаляет повторяющиеся элементы

Скалярные алгоритмы

Count	Подсчитывает количество элементов, равных заданному
count_if	Подсчитывает количество элементов, удовлетворяющих условию
Accumulate	Находит сумму
inner_product	Внутренне произведение двух последовательностей
Equal	Проверяет две последовательности на равенство
Lexicographical_ compare	Сравнивает две последовательности

Алгоритмы, генерирующие последовательности

Transform	Преобразует каждый элемент
partial_sum	Создает последовательность частичных сумм
Adjacent_ difference	Создает последовательность разностей последовательных элементов

Вспомогательные алгоритмы

for_each	Применяет функцию над каждым элементом последовательности

Для использования алгоритмов необходимо включить заголовочные файлы algorithm и numeric.

Многие стандартные алгоритмы принимают в качестве аргумента функцию. Создание таких функций облегчается при использовании шаблонов классов-функций. Класс-функция представляет собой класс, в котором перегружен operator( ). Все классы-функции библиотеки произведены из двух классов: класса unary_function и сласса binary_function.

Для доступа к данным стандартные алгоритмы используют итераторы. Поэтому, если доступ к данным осуществляется с помощью класса, имеющего все признаки итератора, можно использовать стандартные алгоритмы. Например, если данные хранятся в массиве, то указатель на элемент массива обладает всеми признаками итератора и его можно передавать в качестве аргумента в шаблоны стандартных алгоритмов.

Программирование в многозадачной среде

Программа, исполняющаяся в однозадачной операционной системе, безраздельно владеет всеми ресурсами компьютера. Если такая программа нарисовала что-нибудь на экране, то изображение не будет меняться до тех пор, пока программа сама его не изменит. Другое дело – в графической многозадачной операционной системе. Пользователь может переключиться на другое приложение, может изменить размер окна, может полностью или частично перекрыть окно одного приложения окном другого. Возникает и проблема с вводом информации пользователем. В однозадачной системе программа опрашивает клавиатуру, проверяя, нажата ли клавиша. Если нажата, то нужно производить необходимое действие. В многозадачной системе приложение не сможет так просто определить, предназначено ли это нажатие ему или другому приложению. Приложение как бы должно знать о других запущенных приложениях и корректно делить с ними ресурсы компьютера. Писать такие приложения было бы сложно и неэффективно. Задачу разделения ресурсов берет на себя операционная система. Философия многозадачной операционной системы такова: не программа спрашивает у системы, нажал ли пользователь клавишу, пора ли перерисовать содержимое окна, а система говорит приложению о том или ином событии в системе.

Рассмотрим подробнее, как реализован этот принцип в самой распространенной на сегодня многозадачной операционной системе Windows. Приложение состоит из цикла обработки сообщений и обработчиков сообщений. Цикл обработки сообщений занимается тем, что выбирает сообщения из системной очереди сообщений и передает его обработчикам сообщений, пока не получит сообщение WM_QUIT. Приложение имеет хотя бы одно окно. Окно имеет функцию окна. Сообщение попадает в функцию окна, и приложение либо обрабатывает сообщение, либо передает его функции окна по умолчанию. Таким образом, Windows, посылая сообщение,

вызывает соответствующий обработчик. То есть не только приложение вызывает функции операционной системы, но и операционная система вызывает функции приложения. Сообщений очень много; перечислим только самые важные. Сообщения от клавиатуры WM_KEYDOWN, WM_KEYUP, WM_CHAR. Сообщения от мыши WM_MOUSEMOVE, WM_LBUTTONDOWN, WM_LBUTTONUP. Сообщения от кнопок и меню WM_COMMAND. WM_PAINT – пора перерисовать содержимое окна. WM_CREATE – завершено создание окна. WM_CLOSE – окно закрывается. WM_SIZE – окно изменило размер. Окон в операционной системе Windows гораздо больше, чем может показаться на первый взгляд. Все интерфейсные элементы: кнопки, полосы прокрутки, поля ввода, панели инструментов, индикаторы прогресса, и т. д., – это окна, отличающиеся стилем и классом (не путать с классом в C++) окна. Изначально программы для Windows писались на C и других процедурно-ориентированных языках. Однако вскоре были разработаны разнообразные объектные оболочки. Использование таких библиотек заметно облегчает программирование под Windows и избавляет от многих рутинных моментов.

ОСОБЕННОСТИ BORLAND C BUILDER

Borland C Builder – современное средство быстрой разработки приложений для Windows. C Builder, как и Delphi, использует библиотеку компонентов VCL. Компоненты обладают некоторыми особенностями по сравнению с классами C++. Данные класса скрыты. Доступ к данным осуществляется посредством функции, обычно имеющей в названии get, а изменение осуществляет функция, обычно имеющая в названии set. В компоненте функции get и set объединены в единое целое – свойство. Обращение к свойству выглядит как обращение к public полю класса, но на самом деле при считывании генерируется вызов функции get, а при записи вызов функции set. Следующая особенность компонент – наличие обработчиков событий (Events). Обработчики событий компонент представ-

ляют собой по сути свойства типа указатель на функцию определенного типа. Кроме того, компоненты, будучи произведены от Tcomponent, обладают поддержкой визуального программирования. Программирование с использованием компонент выглядит очень просто. Запустите C Builder, по умолчанию создастся новый проект, содержащий одну форму. Постройте проект и запустите приложение. Вы не писали цикл обработки сообщений, не создавали окон; все это C Builder сделал за вас. Программирование с помощью компонент состоит в том, чтобы таскать компоненты мышкой на форму, в инспекторе объектов задавать их свойства и назначать обработчики событий. Всю работу по созданию компоненты, ее инициализации и ее удалению сделает за вас C Builder. Впрочем, ничто не мешает создать экземпляр компоненты так же, как и экземпляр любого другого класса C++. Единственное ограничение – компоненты нельзя создавать в стеке.

```
TButton bt;              // в стеке нельзя

TButton *bt=new TButton(); // в куче можно
. . .
delete bt;
```

Чтобы поменять свойство, надо пометить компоненту мышкой и в инспекторе объектов на закладке свойств отредактировать необходимое свойство. Чтобы создать обработчик события, надо в инспекторе объектов на закладке событий щелкнуть два раза на событии, и C Builder создаст заготовку функции. Осталось написать в обработчиках событий необходимый код, и приложение готово. Такой подход очень эффективен при создании относительно небольших приложений из готовых компонент. Однако у него есть и слабые стороны. Фактически назначение функции обработчику события компоненты заменяет наследование. Если поместить на форму компоненту Label, то Label никак не реагирует на щелчок мыши на нем. Предположим, нам необходим Label, реагирующий на щелчок мыши изменением цвета. Мы можем в

инспекторе объектов назначить обработчик события OnClick. В теле обработчика напишем код, изменяющий цвет Label. Теперь представим, что класс TLabel имеет виртуальную функцию OnClick. Определим класс TmyLabel, унаследовав его от TLabel и переопределив функцию OnClick. Экземпляр TmyLabel и экземпляр компоненты TLabel с назначенным обработчиком On Click ведут себя одинаково. Однако между ними есть существенная разница. Обработчик события On-Click компоненты является функцией формы, на которую мы положили Label, следовательно, из этого обработчика доступны все другие компоненты, принадлежащие форме. Это делает их незащищенными от изменения в этом обработчике. Другая особенность состоит в том, что реализация функциональности, по смыслу принадлежащей Label, вынесена в форму. Вариант с настоящим наследованием лишен всех выше отмеченных недостатков. Если необходимо быстро создать небольшую программу и имеющиеся компоненты не требуют «навески» многих обработчиков событий, можно применять быстрые средства разработки типа C Builder. Если предполагается создание большого и сложного приложения, более эффективным является классический подход: он позволяет избежать многих ошибок за счет правильной организации программы.

РАБОТА С ГРАФИКОЙ В WINDOWS

Для обеспечения аппаратной независимости программного обеспечения необходимо, чтобы программа работала с аппаратурой не напрямую, а используя средства операционной системы. В Windows все графические устройства, будь то принтер, плоттер или дисплей, представлены в виде некоего контекста устройства (Device Context). Когда программе необходимо что-нибудь нарисовать, она получает контекст устройства и осуществляет рисование или вывод текста в этот контекст; после завершения рисования программа возвращает контекст устройства операционной системе. Рисование осуществляется линией, зависящей от карандаша (Pen), вы-

бранного в контекст. Закрашивание зависит от кисти (Brush), выбранной в контекст. Вывод текста зависит от атрибутов вывода текста и выбранного шрифта (Font). Перед возвращением контекста операционной системе необходимо восстановить состояние контекста. Библиотека VCL, применяемая в C Builder, предоставляет оболочку на контекст устройства – TCanvas. Canvas имеет свойство Pen, свойство Brush и свойство Font, которые и определяют, как рисуются линии, как закрашиваются области и как выводится текст. Кроме того, TCanvas предоставляет набор функций для рисования линий, рисования графических примитивов и вывода текста. В TCanvas представлены только основные функции контекста устройства, однако, свойство Handle представляет собой описатель контекста устройства; зная описатель контекста устройства, вы можете вызывать функции Windows API.

Алгоритмы оперативной обработки данных

ФИЛЬТРАЦИЯ СЛУЧАЙНЫХ ШУМОВ В ХОДЕ ЭКСПЕРИМЕНТА

Известно, что нормальный (гауссов) шум можно достаточно успешно отфильтровать путем простого усреднения сигнала. При этом среднеквадратичная ошибка результата:

$$\langle \delta \rangle \propto \frac{1}{\sqrt{N}}, \qquad (1)$$

где N – число измерений, по которым выполняется усреднение.

Отсюда видно, что если одно из измерений оказалось случайным выбросом, который на порядок превышает истинное значение сигнала (и соответственно увеличивает ошибку измерений), то для устранения его влияния необходимо увеличить количество измерений на два порядка. Очевидно, что это сильно увеличивает время проведения эксперимента и не всегда осуществимо. Этим определяется необходимость

предварительной фильтрации результатов измерений. Особенностью всех применяемых здесь методов является то, что фильтрация выполняется в ходе эксперимента, когда полный набор данных отсутствует и имеется только крайне ограниченное число уже полученных измерений.

Метод «ворот»

В этом методе предполагается, что изменения измеряемого сигнала от точки к точке достаточно невелики и имеются априорные (заданные оператором) данные о возможной величине этих изменений (ширина ворот).

Насколько начальных точек рассматриваются как корректные и по ним выполняется аппроксимация сигнала к точке очередного измерения. На рис. 6 показан простейший случай – линейная аппроксимация, когда достаточно двух начальных точек (1 и 2 на рисунке). Если экспериментально измеренное значение лежит на расстоянии, меньшем установленного допустимого отклонения, то она считается истинной (точка 3 на рисунке), и дальнейшая аппроксимация выполняется с учетом полученного значения (в данном случае – по точкам 2 и 3). Если новое измеренное значение выпадает за пределы допустимого отклонения от экстраполированного значения (точка 4 на рисунке), то ее значение рассматривается как случайный выброс, и оно отбрасывается. В идеальном случае, если значение измеряемой функции в этой точке необходимо, следует повторить измерения. Если это невозможно в силу характера проводимых измерений, значение 4 заменяется на экстраполированное значение (точка 4’).

Подобная процедура замены опасна тем, что экстраполированная линия может теперь заметно отличаться от действительной, и высока вероятность того, что последующие правдоподобные точки также будут выпадать за пределы экстраполяции. В результате все дальнейшие экспериментальные значения будут утеряны, независимо от того, являются они выбросами или нет. Чтобы избежать этого, в алгоритмах таких измерений обычно предусматривается условие, запрещающее

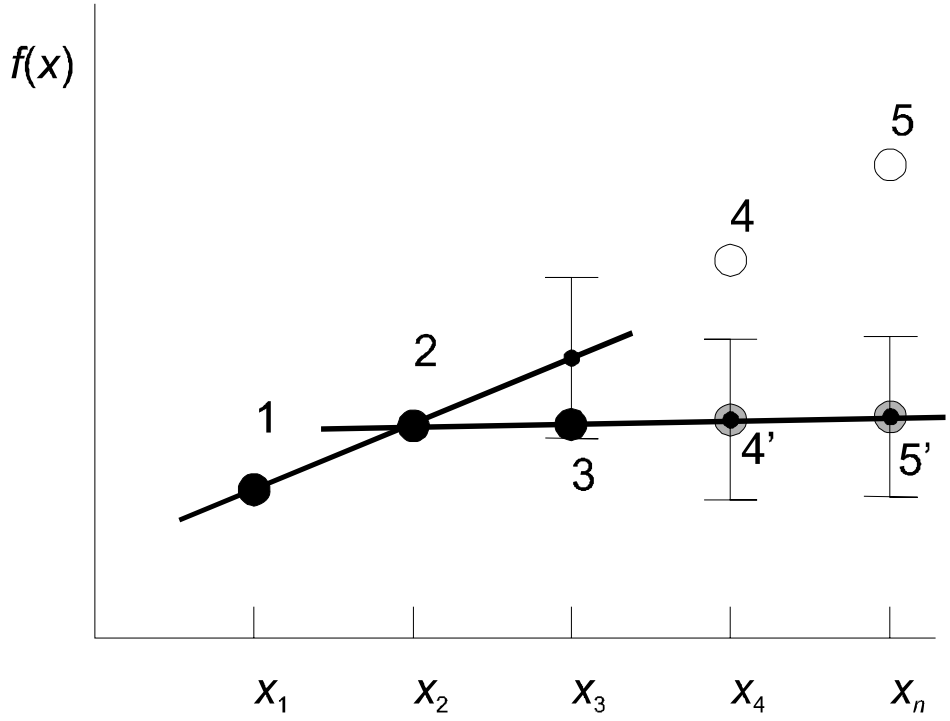


Рис. 6. Фильтрация методом «ворот».

производить повторные замены по экстраполяции; причиной их возникновения обычно является установление слишком узкого доверительного интервала «ворот».

К недостаткам этого метода, таким образом, следует отнести: необходимость произвольного вмешательства оператора при установлении ширины «ворот»; опасность утери данных из-за неудачно установленных «ворот»; замену реально измеренных значений на экстраполированные в «подозрительных» точках.

Влияние этих недостатков в какой-то мере можно ослабить, используя более сильные, нелинейные методы экстраполяции. Однако они, как правило, требуют использования большего количества точек для экстраполяции. Это повышает вероятность попадания случайного выброса среди начальных опорных точек, что приведет к заведомо неверному результату экстраполяции.

Метод выборки

В какой-то мере свободен от этих недостатков метод выборки. Для пояснения его работы введем понятие *робастного среднего*.

Пусть у нас имеется набор из N случайных величин (для простоты будем считать N нечетным). Упорядочим их по величине (по возрастанию или убыванию) и пронумеруем. Значение величины с номером $(N + 1)/2$ (лежащее в середине этой упорядоченной последовательности) называется робастным средним. Существует теорема, доказывающая, что для случайно (гауссово) распределенных величин робастное среднее стремится к «обычному» среднему при $N \to \infty$. В то же время очевидно, что если среди значений выборки окажется случайный выброс, он не повлияет на величину робастного среднего, так как окажется на одном из краев упорядоченной последовательности значений. На этом факте и основан рассматриваемый метод.

Пусть мы имеем первые три (в общем случае – любое нечетное число) измерений экспериментальной последовательности $f(x)$. Упорядочим их по величине и найдем среднюю (по номеру) точку. В показанном на рис. 7 примере это точка 3 из первых трех. Считаем ее средним значением измеряемой величины $f(x)$ на интервале x_1–x_3; на рисунке эта усредненная последовательность обозначена $f'(x)$. Далее берем точки x_2–x_4 и повторяем процедуру: средним значением опять оказывается точка 3. Повторяем это процедуру до конца измерений. Как видно из рисунка, получившаяся в результате фильтрации последовательность $f'(x)$ выглядит более гладкой и не содержит минимальных и максимальных значений последовательности $f(x)$, которые могли быть случайными выбросами.

Достоинствами этого метода являются: простота (не требуется никаких математических действий, только операции сравнения и обмена данными между массивами); отсутствие вычисленных значений измеряемых величин – все значения в

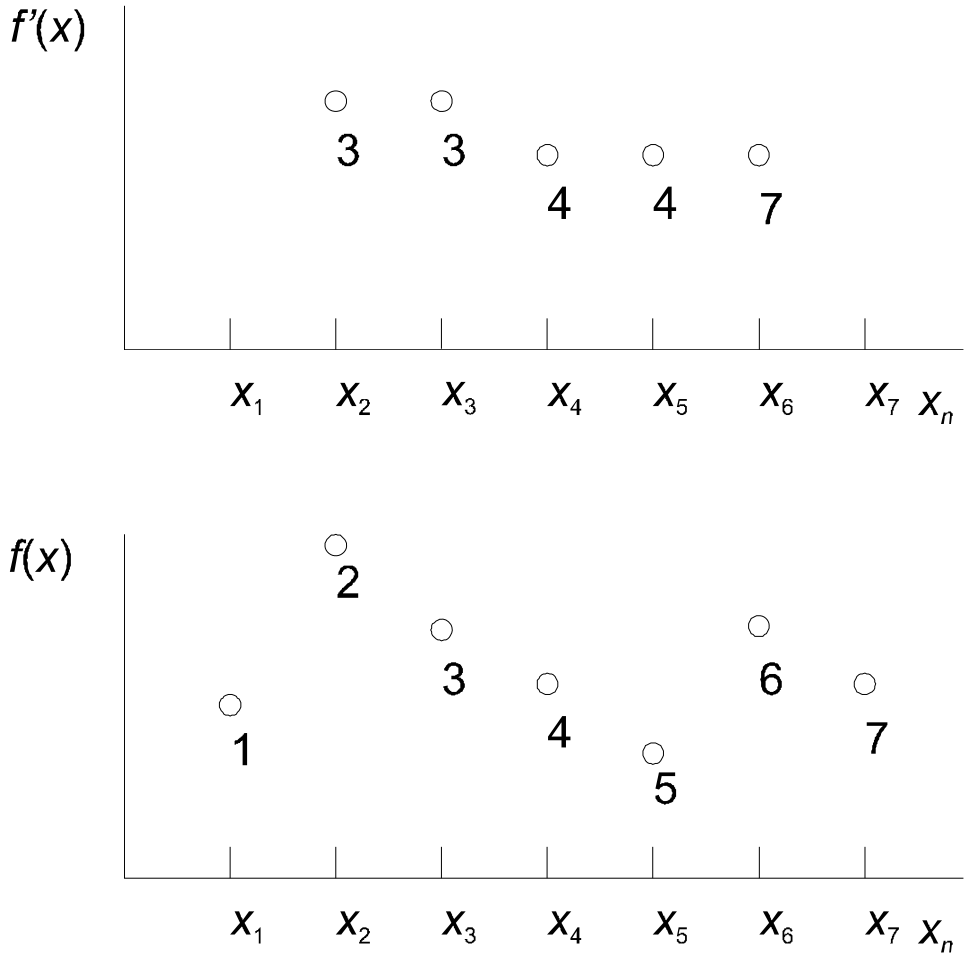


Рис. 7. Фильтрация методом выборки.
Снизу – экспериментально измеренные значения,
сверху – результат фильтрации.

отфильтрованном массиве получены экспериментально; отсутствие необходимости ввода априорных данных об измеряемых величинах (за исключением числа точек, по которым следует производить фильтрацию).

Легко убедиться, что фильтрация с выборкой по трем точкам, как в данном примере, не позволяет удалить два случайных выброса, если они следуют друг за другом; один из них проникает в отфильтрованную последовательность. Чтобы отфильтровать n последовательных выбросов, необходима фильтрация по $2n + 1$ точкам. При увеличении числа точек качество фильтрации улучшается (на выходе образуется более гладкая последовательность), но увеличивается количест-

во отброшенных значений. В любом случае, фильтрация по $2n + 1$ точкам приводит к потере n точек в начале и в конце массива, что, несомненно, является недостатком метода.

Отметим, что данный алгоритм является эффективным средством диагностики случайных выбросов и поэтому часто применяется в качестве предварительного фильтра в сочетании с другими. Так, при многократных измерениях в повторяющихся условиях, когда предполагается установить истинное значение зашумленного сигнала, его можно использовать для предварительного отделения заведомых выбросов, после чего использовать обычное усреднение.

АППРОКСИМАЦИЯ ЭКСПЕРИМЕНТАЛЬНЫХ ДАННЫХ С ПОМОЩЬЮ АНАЛИТИЧЕСКИХ ФУНКЦИЙ

Полученные в ходе эксперимента данные, как правило, представляют собой набор точек. В то же время для удобства их отображения, а также для последующей интерпретации желательно представить их в виде непрерывной зависимости, лучше всего – аналитической. Для этого разработаны и используются многочисленные алгоритмы. Здесь описаны наиболее простые из них, реализация которых не требует значительных затрат машинного времени или больших ресурсов используемого компьютера.

В простейшем случае задача аппроксимации формулируется следующим образом. В результате эксперимента в точках x_1–x_N найдены (приближенные) значения y_1–y_N неизвестной функции $y = f(x)$. Пользователем или программистом задан некий набор функций $\{F(x, \mathbf{p})\}$, где $\mathbf{p} = (p_1, \ldots p_\alpha)$ – набор параметров этих функций. Задача состоит в том, чтобы выбрать из этого набора функцию $F(x, \mathbf{p})$, удовлетворяющую заданному критерию близости к $f(x)$. В зависимости от выбора этого класса функций и критерия близости можно построить самые разнообразные алгоритмы такого поиска, позволяющие решать различные практические задачи.

Одним из простейших критериев близости является совпадение $f(x)$ и $F(x, \mathbf{p})$ во всех экспериментальных точках; та-

кая задача называется *задачей интерполяции*, а точки, через которые должна проходить искомая функция – *узлами интерполяции*. Такая задача может быть решена в том случае, если имеется достаточно большой набор варьируемых параметров **p**, а ошибки измерений настолько малы, что их можно не принимать во внимание (либо сама задача носит промежуточный характер – например, выполняется для удобства визуализации полученных результатов измерений).

В противном случае за критерий близости принимается менее жесткое условие, например минимизация выражения:

$$S(\mathbf{p}) = \sum_{1}^{N} (y_i - F(x_i, \mathbf{p}))^2, \qquad (2)$$

соответствующие алгоритмы являются разновидностями *метода наименьших квадратов*.

Наконец, в некоторых случаях в качестве критерия близости выбирается условие минимакса (*минимаксный критерий*), то есть минимизация функции:

$$Z(\mathbf{p}) = \max | y_i - F(x_i, \mathbf{p}) |. \qquad (3)$$

Интерполяция с помощью полиномов

Задача интерполяции с помощью полиномов формулируется следующим образом. На некотором отрезке $[a, b]$ заданы N точек $y_i(x_i)$. Необходимо найти полином:

$$P_n(x) = \sum_{\alpha=0}^{n} a_\alpha x^\alpha, x \in [a, b]; \qquad (4)$$

для которого

$$P_n(x_i) = y_i, i = 1,...N. \qquad (5)$$

Коэффициенты полинома a_α являются искомыми параметрами; их можно найти, решая систему уравнений (5).

В общем случае, если $n + 1 < N$, эта система не имеет решений. Если $n + 1 = N$, то определителем этой системы является определитель Вандермонда

$$D = \prod_{i>j}^{N} (x_i - x_j), \tag{6}$$

и система имеет единственное решение, если он не равен нулю, что, достаточно просто обеспечить (это означает, что для каждой экспериментальной точки имеется единственное значение экспериментально измеряемой величины y).

Отсюда следует, что по заданному набору N экспериментальных точек можно построить интерполяционный полином степени $N - 1$, причем он является единственным.

Интерполяционная формула Лагранжа

Предположим, что удалось построить систему полиномов $\varphi_i(x)$, каждый из которых равен единице в точке x_i и равен нулю во всех точках измерений. Тогда искомый полином степени $N - 1$ имеет вид:

$$P_{N-1}(x) = \sum_{i=1}^{N} y_i \varphi_i(x). \tag{7}$$

Этот набор полиномов $\varphi_i(x)$ называется *фундаментальной системой полиномов.* Поскольку этот полином единственный и обращается в нуль во всех точках измерений, кроме x_i, его можно представить в виде:

$$\varphi_i(x) = C_i \prod_{k \neq i}^{N} (x - x_k), \tag{8}$$

где C_i – некоторая постоянная. Найдя ее из условия $\varphi_i(x_i) = 1$ и подставив в (8), получим:

$$\varphi_i(x) = \prod_{k \neq i}^{N} (x - x_k) \Big/ \prod_{k \neq i}^{N} (x_i - x_k). \tag{9}$$

Подставляя это в (7), получим *интерполяционный полином Лагранжа*:

$$P_{N-1}(x) = \sum_{i=1}^{N} y_i \prod_{k \neq i}^{N} (x - x_k) \Big/ \prod_{k \neq i}^{N} (x_i - x_k). \qquad (10)$$

Чтобы представить это выражение в более традиционном и компактном виде, введем вспомогательную функцию:

$$\Pi(x) = \prod_{k=1}^{N} (x - x_k), \qquad (11)$$

производная которой равна

$$\Pi'(x) = \sum_{m=1}^{N} \prod_{k \neq m}^{N} (x - x_k). \qquad (12)$$

Тогда:

$$P_{N-1}(x) = \sum_{i=1}^{N} y_i \frac{\Pi(x)}{(x - x_i)\Pi'(x_i)} = \Pi(x) \sum_{i=1}^{N} \frac{y_i}{(x - x_i)\Pi'(x_i)}. \qquad (13)$$

Для равноотстоящих точек $x = x_1 + (t - 1)h$:

$$P_{N-1}(x) = \prod_{m=1}^{N} (t - m) \sum_{k=1}^{N} \frac{(-1)^{N-k} y_k}{(t - k)(k - 1)!(N - k)!}. \qquad (14)$$

Аппроксимация с помощью этого алгоритма является достаточно эффективной, когда аппроксимируемые функции достаточно гладкие, а число точек невелико (и, соответственно, невелика степень аппроксимирующего полинома). Большинство распространенных библиотек имеют готовые программные модули, реализующие этот алгоритм. Рост количества точек и степени полинома приводит к быстрому усложнению выражений (10), (13), (14) и зачастую неоправданному возрастанию требований к ресурсам ЭВМ. Еще одним слабым местом этого алгоритма является то, что для его реализации необходимо знание всех точек аппроксимируемой

функции, то есть его сложно реализовать в ходе измерений. Последний недостаток преодолен в следующем алгоритме.

Интерполяционная формула Ньютона

Введем понятие *разделенных разностей*. Определим разделенную разность первого порядка как:

$$\Delta(x_i, x_{i-1}) = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}. \quad (15)$$

Разность второго порядка:

$$\Delta(x_i, x_{i-1}, x_{i-2}) = \frac{\Delta(x_i, x_{i-1}) - \Delta(x_{i-1}, x_{i-2})}{x_i - x_{i-2}}, \quad (16)$$

и в общем случае разность *n*-того порядка:

$$\Delta(x_i, \ldots x_{i-n}) = \frac{\Delta(x_i, \ldots x_{i-n+1}) - \Delta(x_{i-1}, \ldots x_{i-n})}{x_i - x_{i-n}}. \quad (17)$$

Отметим, что эти разности являются разностным аналогом производных *n*-того порядка.

Рассмотрим разность первого порядка:

$$\Delta(x, x_1) = \frac{f(x) - y_1}{x - x_1}. \quad (18)$$

Отсюда:

$$f(x) = y_1 + (x - x_1)\Delta(x, x_1). \quad (19)$$

Выразив разность первого порядка из (16) и подставив в (19), получим:

$$f(x) = y_1 + (x - x_1)\Delta(x_1, x_2) + (x - x_1)(x - x_2)\Delta(x, x_1, x_2), \quad (20)$$

и, повторяя эту процедуру для разностей более высоких порядков,

$$f(x) = P_{N-1}(x) + R_{N-1}(x), \quad (21)$$

где полином

$$P_{N-1}(x) = y_1 + (x - x_1)\Delta(x_1, x_2) + \\ + (x - x_1)(x - x_2)\Delta(x_1, x_2, x_3) + ... \\ + (x - x_1)...(x - x_{N-1})\Delta(x_1, x_2, ...x_{N-1}) \qquad (22)$$

является интерполяционным, так как

$$P_{N-1}(x_i) = y_i, \qquad (23)$$

а остаточный член

$$R_{N-1}(x) = (x - x_1)...(x - x_N)\Delta(x, x_1, ...x_N) \qquad (24)$$

определяет точность интерполяции. Выражение (22) является разностным аналогом формулы Тейлора разложения функции в ряд.

Как уже отмечалось выше, интерполяционный полином для заданного набора точек является единственным, поэтому путем перегруппировки членов полинома Ньютона можно получить полином Лагранжа, и наоборот.

Преимущество данного представления заключается в том, что *n*-тое слагаемое полинома Ньютона зависит только от *n* первых экспериментальных точек, и последующие точки приводят только к появлению новых слагаемых, без изменения первоначальных. Таким образом, построение этого полинома может идти непрерывно в процессе измерений.

В то же время этот метод не свободен от другого недостатка метода Лагранжа: как и в предыдущем случае, сложность вычислений, и, соответственно, требования к ресурсам компьютера быстро возрастают с увеличением числа экспериментальных точек. Чтобы избежать этого, возникает идея разбить весь исследуемый интервал на отдельные участки и выполнять интерполяцию на каждом из них раздельно. Но тогда интерполирующая функция будет иметь особенности на границах участков интерполяции; здесь возникнут точки перегиба, т. е. производные этой функции здесь будут иметь разрывы. Этого недостатка лишен следующий метод.

Интерполяция с помощью кубических сплайнов

Разобьем область экспериментальных точек $[a, b]$ на участки $\delta_i = [x_{i-1}, x_i]$. Интерполирующая функция $F(x)$ должна проходить через все экспериментальные точки: $F(x_i) = y_i$; она сама, ее первая и вторая производные должны быть непрерывны на отрезке $[a, b]$. Естественно искать эту функцию, интерполируя экспериментальные точки на каждом из отрезков $\delta_i = [x_{i-1}, x_i]$ полиномом. Тогда низшей степенью такого полинома может быть только третья – иначе его производные превратятся в нули. Введем $C_i = F''(x_i)$. Тогда для кубического сплайна:

$$\frac{F''(x) - C_{i-1}}{x - x_{i-1}} = \frac{C_i - C_{i-1}}{x_i - x_{i-1}}, \tag{25}$$

откуда

$$\frac{F''(x) - C_{i-1}}{x - x_{i-1}} = \frac{C_i - C_{i-1}}{x_i - x_{i-1}}. \tag{26}$$

Выражая отсюда F'' и дважды интегрируя полученное выражение, получим:

$$\begin{aligned} F(x) = {} & C_i \frac{(x_i - x)^3}{6(x_i - x_{i-1})} + C_i \frac{(x - x_{i-1})^3}{6(x_i - x_{i-1})} + \\ & + \frac{(x_i - x)}{(x_i - x_{i-1})}\left(y_{i-1} - \frac{C_{i-1}(x_i - x_{i-1})^2}{6} \right) + \\ & + \frac{(x - x_{i-1})}{(x_i - x_{i-1})}\left(y_i - \frac{C_i (x_i - x_{i-1})^2}{6} \right). \end{aligned} \tag{27}$$

Легко убедиться, что данная функция отвечает поставленным условиям.

Коэффициенты сплайна C_i можно найти, исходя из условий непрерывной дифференцируемости (27) во всех точках x_i:

$$F'(x_i - 0) = F'(x_i + 0), i = 2, N - 1. \tag{28}$$

Выражение (28) после подстановки туда (27) дает систему из $N-2$ линейных уравнений для N неизвестных коэффициентов сплайна C. Недостающие два уравнения необходимо получить, поставив граничные условия на интерполирующую функцию или ее производные на границах участка $[a, b]$.

Аппроксимация экспериментальных данных методом наименьших квадратов

Как правило, экспериментально измеряемые величины y_i содержат ошибки, а их зависимость от параметра эксперимента x_i не является полиномиальной. Поэтому требование прохождения интерполирующей кривой через каждую экспериментальную точку оказывается чрезмерно жестким и не несет физического смысла. В то же время часто, исходя из физических соображений, удается найти предполагаемое аналитическое выражение для зависимости $y_i = F(x_i, \mathbf{p})$, содержащее некоторое ограниченное число неизвестных параметров p_α. В этом случае целесообразно и физически значимо аппроксимировать эту зависимость, используя такое выражение, и определять неизвестные параметры физической модели *методом наименьших квадратов* (2), минимизируя функционал $S(\mathbf{p}) = \sum_1^N (y_i - F(x_i, \mathbf{p}))^2$.

Простейшей ситуацией при этом является случай *линейной модели*, когда

$$F(x_i, \mathbf{p}) = \sum_\alpha p_\alpha F(x_i). \qquad (29)$$

В этом случае поиск параметров p_α сводится к решению системы линейных уравнений

$$\frac{\partial S(\mathbf{p})}{\partial p_\alpha} = 0, \qquad (30)$$

и проблемы возникают только при плохой обусловленности этой системы.

Но, как правило, физические модели исследуемых процессов не сводятся к простым выражениям вида (29), уравнения (30) оказываются нелинейными, и возникает задача минимизации этого функционала в пространстве нескольких переменных p_α.

Общих методов решения таких задач в настоящее время не существует; они сильно отличаются как по своей природе, так и по математическим особенностям. В зависимости от характера доступной информации о *целевой функции* (2) или (3) различают методы нулевого порядка (имеется только информация о значениях функции в конечном числе точек), первого порядка (известны также значения первых производных целевой функции в этих точках) и второго порядка (используются также сведения о вторых производных).

Большинство этих методов сводится к вводу некоторого начального приближения $p_\alpha^{\,0}$ для значений искомых параметров, затем по определенному алгоритму находится направление **d** поиска минимума $S(\mathbf{p})$ в многомерном пространстве параметров p_α. В этом направлении осуществляется одномерная минимизация целевой функции. Найденные в результате значения $p_\alpha^{\,1}$ принимаются в качестве новых начальных значений параметров, и процедура повторяется до достижения заранее поставленных условий остановки. В качестве условий остановки обычно выбираются достижение достаточно малого значения целевой функции, ее производных, выполнение достаточно большого количества итераций, либо некоторая комбинация этих условий.

Методы нулевого порядка

Вообще говоря, методы нулевого порядка, к которым относятся методы прямого поиска, обладают самой медленной скоростью сходимости. Однако, когда число варьируемых параметров является большим, а получение значений производных целевой функции невозможно либо требует значительных вычислительных мощностей, эти методы могут дать приемлемые результаты.

Достоинством этих методов является также то, что их можно применять не только в методе наименьших квадратов, но и в задачах минимакса, когда целевая функция (3) заведомо недифференцируема.

Метод покоординатного спуска

Метод покоординатного спуска является одним из самых простых методов нулевого порядка. Если $\mathbf{e}_\alpha$ – единичные векторы пространства параметров p_α, то на первом шаге в качестве направления поиска минимума принимается $\mathbf{e}_1$ (достигнутое при этом значение параметров обозначим $\mathbf{p}^1$), затем $\mathbf{e}_2$ ($\mathbf{p}^2$), и так далее, до последнего единичного вектора $\mathbf{e}_\omega$ ($\mathbf{p}^\omega$). Затем процедура повторяется. Алгоритм имеет свойство «застревать», если в пространстве параметров имеется направление, в котором целевая функция изменяется особенно сильно (линии равных значений целевой функции имеют вид сильно вытянутого оврага). Чтобы избежать этого, Хуком и Дживсом было предложено дополнить процедуру. После итерации с некоторым заданным номером ω они предложили выполнять поиск минимума в направлении, соединяющем точки $\mathbf{p}^0$ и $\mathbf{p}^\omega$. После этого покоординатный поиск продолжается. Такая модификация существенно повышает эффективность метода.

Существуют также модификации метода координатного спуска, отличающиеся тем, что система единичных векторов $\mathbf{e}_\alpha$ после каждого цикла тем или иным образом модифицируется таким образом, чтобы избежать «застревания» алгоритма и ускорить сходимость метода.

Симплексный метод

Этот метод чаще всего используется для минимизации сложных либо недифференцируемых целевых функций с большим числом параметров. По определению, *симплексом* называется многогранник в ω-мерном пространстве, имеющий ω+1 вершину с координатами, равными столбцам матрицы:

$$B_{\omega,\omega+1} = \begin{pmatrix} 0 & b_1 & b_2 & \dots & b_2 \\ 0 & b_2 & b_1 & \dots & b_2 \\ 0 & b_2 & b_2 & \dots & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & b_2 & b_2 & \dots & b_1 \end{pmatrix},$$

$$b_1 = \frac{a}{\omega\sqrt{2}}(\sqrt{\omega+1}+\omega-1), \quad , \qquad (31)$$

$$b_2 = \frac{a}{\omega\sqrt{2}}(\sqrt{\omega+1}-1),$$

a – расстояние между вершинами (сторона) многогранника.

Координаты вершин такого многогранника выбираются в качестве начальных значений параметров $p_\alpha{}^{o}$.

В каждой из вершин находится значение целевой функции, и точка, в которой значение этой функции максимально, отбрасывается. После этого, используя тот или иной алгоритм, строится новый симплекс.

В стандартном методе симплекса новый многогранник является отражением исходного в плоскости грани, противоположной отбрасываемой точке. В его различных модификациях используются более сложные алгоритмы, допускающие изменение размеров симплекса (увеличение после удачных итераций и сжатие после неудачных), либо его деформацию.

Имеется большое количество библиотечных подпрограмм, реализующих различные варианты этого метода.

Методы случайного поиска

Эффективность методов поиска минимума целевой функции сильно зависит от того, насколько удачно выбрано начальное приближение $p_\alpha{}^{0}$. В некоторых случаях начальные значения параметров можно выбрать из физических соображений, однако иногда такая возможность отсутствует. Неудачный выбор начальных параметров поиска может привести к тому, что алгоритм окажется в окрестности локального

минимума целевой функции, где большинство из них «застревает». Метод случайного поиска в некоторой степени позволяет обойти эту трудность.

Для большинства вариантов этого метода последовательность наборов параметров определяется соотношением:

$$\mathbf{p}^{i+1} = \mathbf{p}^{i} + \rho_i \mathbf{r}, \tag{32}$$

где ρ_i – величина шага, $\mathbf{r}$ – случайный вектор. Поиск заключается в генерации заданного числа этих случайных векторов направления поиска, определении соответствующих значений целевой функции, и выбору минимального из них. Соответствующая точка принимается далее за исходную для начала следующей итерации. В простейших вариантах метода величина шага при этом остается неизменной, в более сложных – варьируется, аналогично длине стороны многогранника в предыдущем методе. В алгоритмах *случайного поиска с адаптацией* при последующих генерациях случайных векторов увеличивается вероятность генерации векторов в направлении последнего удачного поиска.

Методы первого порядка

Градиентный метод (метод Коши)

По определению, градиентом функции является вектор ее частных производных по координатам:

$$\mathrm{grad}S(\mathbf{p}) = \left(\frac{\partial S}{\partial p_1}, ..., \frac{\partial S}{\partial p_\omega} \right). \tag{33}$$

Из математического анализа известно, что вектор градиента задает направление наискорейшего возрастания функции. Тогда вектор –grad(S) определяет направление ее наискорейшего убывания. Градиентный метод заключается в поиске очередного набора параметров целевой функции, исходя из соотношения:

$$\mathbf{p}^{i+1} = \mathbf{p}^{i} - \rho_i \mathrm{grad}S(\mathbf{p}^{i}), \tag{34}$$

где величина шага ρ_i на каждой итерации находится одним из методов одномерной минимизации. Различные версии градиентного метода отличаются, главным образом, выбором алгоритма одномерной минимизации.

Градиентный метод является одним из самых медленных среди методов первого порядка и имеет свойство «застревать» в «оврагах» целевой функции (областях со слабой зависимостью целевой функции в одном из направлений пространства параметров), особенно если точность одномерного поиска недостаточно высока. Следующий метод разработан для коррекции этого недостатка.

Овражный метод

В этом методе, в случае застревания градиентного поиска в некоторой точке $\mathbf{p}^1$, производится повторный поиск из другой, близкой к исходной, начальной точки. После того, как метод снова застрянет в некоторой точке $\mathbf{p}^2$, осуществляется одномерный поиск минимума вдоль прямой, проходящей через точки $\mathbf{p}^1$ и $\mathbf{p}^2$. Далее процедура повторяется.

Метод сопряженных градиентов
(метод Флетчера – Ривза)

В градиентном методе при выборе очередного направления одномерного поиска не учитывается информация о целевой функции, полученная на предыдущих итерациях. Данный метод осуществляет такой учет. Направление поиска минимума здесь задается соотношениями:

$$\begin{aligned}
&\mathbf{p}^{i+1} = \mathbf{p}^i - \rho_i(\mathbf{g}^i + \lambda_i \mathbf{g}^{i-1}),\\
&\mathbf{g}^i = \mathrm{grad}S(\mathbf{p}^i),\\
&\lambda_i = (\mathbf{g}^i \mathbf{g}^{i-1} - (\mathbf{g}^i)^2)/(\mathbf{g}^{i-1})^2.
\end{aligned} \qquad (35)$$

Используются и другие методы задания параметра сопряжения λ.

Метод сопряженных градиентов сходится быстрее, чем обычный градиентный метод; он особенно эффективен на за-

вершающих этапах минимизации, когда зависимость целевой функции от варьируемых параметров становится слабой.

Методы второго порядка
(квизиньютоновские методы)

Метод Ньютона – Рафсона

Разложим целевую функцию в ряд вблизи начальной точки поиска и ограничимся членами второго порядка малости:

$$S(\mathbf{p}) \approx S(\mathbf{p}^0) + (\mathbf{p} - \mathbf{p}^0)\mathbf{g}^0 + (\mathbf{p} - \mathbf{p}^0)\mathbf{H}(\mathbf{p}^0)(\mathbf{p} - \mathbf{p}^0). \qquad (36)$$

Здесь **H** – матрица Гессе, образованная вторыми производными целевой функции по ее параметрам (иногда матрицей Гессе называют матрицу, обратную матрице вторых производных).

Условием минимума целевой функции является равенство нулю ее градиента, то есть:

$$0 = \text{grad}S(\mathbf{p}^{\min}) \approx \mathbf{g}^0 + \mathbf{H}(\mathbf{p}^0)(\mathbf{p}^{\min} - \mathbf{p}^0), \qquad (37)$$

откуда

$$\mathbf{p}^{\min} \approx \mathbf{p}^0 - [\mathbf{H}(\mathbf{p}^0)]^{-1}\mathbf{g}^0. \qquad (38)$$

Отсюда следует рекуррентное соотношение поиска минимума методом Ньютона:

$$\mathbf{p}^{i+1} = \mathbf{p}^i - [\mathbf{H}(\mathbf{p}^i)]^{-1}\mathbf{g}^i. \qquad (39)$$

Метод Ньютона сходится только вблизи точек минимума целевой функции и при положительно определенных матрицах Гессе. Кроме того, вычисление новой матрицы Гессе на каждой итерации требует значительных вычислительных затрат. В связи с этим в практике обычно применяется *обобщенный метод Ньютона*. Матрица Гессе используется для определения направления поиска:

$$\mathbf{d}^i = -[\mathbf{H}(\mathbf{p}^i)]^{-1}\mathbf{g}^i, \qquad (40)$$

которое затем используется для одномерной минимизации, аналогично градиентным методам:

$$\mathbf{p}^{i+1} = \mathbf{p}^{i} + \rho_i \mathbf{d}^{i}. \tag{41}$$

Различные варианты обобщенного метода Ньютона отличаются используемыми алгоритмами одномерной минимизации.

Метод Ньютона является одним из наиболее эффективных алгоритмов многомерной минимизации. Для целевых функций, являющихся квадратичными формами параметров, он сходится за один шаг. Для сложных целевых функций, с различной зависимостью от входящих в них параметров, скорость его сходимости может оказаться в 100–1000 раз выше, чем, например, у градиентного метода или его аналогов. Однако недостатком метода является необходимость расчета и обращения громоздкой матрицы Гессе на каждой итерации.

Метод Ньютона – Гаусса

С учетом выражения (2) для целевой функции ее градиент и матрицу Гессе можно представить в виде:

$$g_\alpha = -2\sum_{i=1}^{N}(y_i - F(x_i, \mathbf{p}))\frac{\partial F(x_i, \mathbf{p})}{\partial p_\alpha}, \tag{42}$$

$$H_{\alpha\beta} = 2\sum_{i=1}^{N}\frac{\partial F(x_i, \mathbf{p})}{\partial p_\alpha}\frac{\partial F(x_i, \mathbf{p})}{\partial p_\beta} - 2\sum_{i=1}^{N}(y_i - F(x_i, \mathbf{p}))\frac{\partial^2 F(x_i, \mathbf{p})}{\partial p_\alpha \partial p_\beta}. \tag{43}$$

Введем матрицу:

$$P_{i\alpha} = \frac{\partial F(x_i, \mathbf{p})}{\partial p_\alpha} \tag{44}$$

размера $N \times \omega$. Тогда (43) можно представить в виде:

$$\mathbf{H} = \mathbf{H}^{(1)} - \mathbf{H}^{(2)},$$

$$\mathbf{H}^{(1)} = 2\mathbf{P}^T\mathbf{P},$$

$$H^{(2)}_{\alpha\beta} = 2\sum_{i=1}^{N}(y_i - F(x_i, \mathbf{p}))\frac{\partial^2 F(x_i, \mathbf{p})}{\partial p_\alpha \partial p_\beta}. \qquad (45)$$

Если мы находимся недалеко от минимума, а вторые производные F'' не слишком велики, то элементы матрицы $\mathbf{H}^{(2)}$ малы и $\mathbf{H} \approx \mathbf{H}^{(1)}$. С учетом этого, подставляя (43) – (35) в (39), получим рекуррентное соотношение для поиска минимума целевой функции методом Ньютона – Гаусса:

$$\mathbf{p}^{i+1} = \mathbf{p}^i + [\mathbf{P}^{iT}\mathbf{P}^i]^{-1}\mathbf{P}^{iT}(\mathbf{Y} - \mathbf{F}(x_i, \mathbf{p}^i)), \qquad (46)$$

где $\mathbf{Y}$ и $\mathbf{F}$ – векторы, составленные из экспериментально измеренных величин и рассчитанных значений модельной функции F в этих точках при заданных параметрах. По объему вычислений этот метод значительно лучше, чем метод Ньютона, так как не требует вычисления матрицы Гессе, а по скорости сходимости мало ему уступает, особенно вблизи минимума. Для ускорения сходимости этот метод можно модифицировать аналогично методу Ньютона, введя переменную длину итерации:

$$\mathbf{p}^{i+1} = \mathbf{p}^i + \rho_i[\mathbf{P}^{iT}\mathbf{P}^i]^{-1}\mathbf{P}^{iT}(\mathbf{Y} - \mathbf{F}(x_i, \mathbf{p}^i)), \qquad (47)$$

величина которой определяется методом одномерной минимизации.

Метод Дэвидона – Флетчера – Пауэлла

Другим методом использования сведений о форме целевой функции при сокращении вычислительных затрат на расчеты и обращение матрицы Гессе является ее оценка одним из методов конечных разностей на основе сведений, полученных при расчете целевой функции и ее первых производных в нескольких точках. Метод Дэвидона – Флетчера – Пауэлла явля-

ется одной из реализаций этой идеи. Для оценки матрицы, обратной матрице Гессе, здесь используется выражение:

$$(H)^{-1}{}^{i+1}_{\alpha\beta} = (H)^{-1}{}^{i}_{\alpha\beta} - \frac{(p^{i+1}_{\alpha} - p^{i}_{\alpha})d^{i}_{\beta}}{\sum_{\gamma}(g^{i+1}_{\gamma} - g^{i}_{\gamma})d^{i}_{\gamma}} -$$

$$- \frac{\sum_{\gamma}(H)^{-1}{}^{i}_{\alpha\gamma}(g^{i+1}_{\gamma} - g^{i}_{\gamma})\sum_{\gamma}(H)^{-1}{}^{i}_{\beta\gamma}(g^{i+1}_{\gamma} - g^{i}_{\gamma})}{\sum_{\gamma\delta}(g^{i+1}_{\gamma} - g^{i}_{\gamma})(H)^{-1}{}^{i}_{\gamma\delta}(g^{i+1}_{\delta} - g^{i}_{\delta})}, \quad (48)$$

и далее используется итерационная процедура, аналогичная (40) – (41). На первом шаге в качестве оценки матрицы, обратной матрице Гессе, используется единичная матрица. Таким образом, на первом шаге этот метод совпадает с градиентным (34), а затем, по мере приближения в точке минимума из-за уменьшения длины одномерных итераций ρ, разностная оценка (48) становится все более точной и метод сводится к обобщенному методу Ньютона. Существуют разновидности этого метода, отличающиеся, видом разностной схемы, которая используется для оценки матрицы Гессе, что приводит к несколько иному виду выражения (48); все эти алгоритмы признаны надежным методом решения задач оптимизации, хотя и требуют достаточно больших вычислительных ресурсов на каждом шаге итерации.

КОНТРОЛЬНЫЕ ЗАДАНИЯ

Разработка класса spectrpoint для хранения точек спектра с применением перегрузки операторов ввода-вывода в поток, хранение экземпляров spectrpoint в стандартном векторе. Цель задания – освоение приемов объектного программирования и стандартной библиотеки шаблонов.

Разработка класса TAxis – базового класса, включающего в себя общую функциональность присущую осям *X* и *Y*. Цель задания – ознакомиться с применением виртуальных функций.

Разработка окна диалога Axis setup. Цель задания – освоение приемов построения диалоговых окон.

Разработка классов TX_Axis и TY_Axis, унаследовав их от TAxis. Цель задания – применение наследования, работа с Canvas и Device context, изучение основных приемов работы с графикой (вывод текста в различных режимах, вывод графических примитивов).

Разработка класса Tgrath на основе TX_Axis и TY_Axis, написание обработчиков сообщений Windows. Цель задания – построение полноценного, автоматически масштабируемого, настраиваемого объекта – компонента, показывающего график функции одной переменной.

Разработка класса, реализующего обработку данных. Цель задания – практическая реализация алгоритма фильтрации случайных ошибок.

Создание Windows-приложения обработки данных с применением ранее разработанных классов. Цель задания – освоение приемов построения Windows-приложений, работа со стандартными диалогами File Open и File Save, разработка главного меню приложения.

ПРАКТИЧЕСКИЕ ЗАДАНИЯ

Работа № 1.
Программирование интерфейсной платы
и контроллера крейта КАМАК

ОБОРУДОВАНИЕ: Крейт КАМАК с интерфейсной платой подключенной к компьютеру, индикатор магистрали.

ЦЕЛЬ: Изучение основных команд системы КАМАК, организации связи между компьютером и контроллером крейта КАМАК.

КРАТКОЕ ОПИСАНИЕ РАБОТЫ: В ходе выполнения работы необходимо изучить назначение команд КАМАК. Написать программные модули для передачи команд и данных в компьютер. Проверить прохождение команд по показаниям индикатора магистрали.

Перед началом работы убедитесь, что станции 24 и 25 заняты контроллером крейта КАМАК, подключенным через интерфейсную плату к компьютеру, а модуль индикатор магистрали установлен в крейте в любой свободной станции.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

1. Ознакомьтесь со способами передачи управляющих команд и данных из компьютера в контроллер.

2. По техническому описанию контроллера и интерфейсной платы определите способ передачи команд, данных и адреса интерфейсной платы, с которыми вам предстоит работать.

3. Составьте алгоритм передачи команд и данных в контроллер крейта.

4. Напишите программные модули, реализующие ваш алгоритм (образцы программных модулей для параллельного и последовательного интерфейсов приведены в Приложениях 1 и 2 соответственно).

После отладки программы необходимо проверить ее работоспособность. Правильное выполнение можно проконтролировать с помощью модуля «Индикатор магистрали». Записывая различные команды по адресу станции, в которой расположен этот или какой-либо другой модуль, по индикации на передней панели индикатора можно прочитать полученную станцией команду. Сравните полученную команду с посланной. В случае необходимости исправьте программу.

Проверьте правильность передачи данных, при этом контролируйте передачу на всех 24 линиях шины.

Работа № 2.
Изучение основных команд и принципов управления в системе КАМАК

ОБОРУДОВАНИЕ: Крейт КАМАК; модули: счетчик $СЧ_6$10, индикатор магистрали.

ЦЕЛЬ: Изучение основных команд системы КАМАК, определение набора команд используемых модулем.

КРАТКОЕ ОПИСАНИЕ РАБОТЫ: Подготовить модуль к работе. Определить набор адресов и команд, используемых модулем. Установить на табло счетчика заданное число. Добиться его увеличения и уменьшения на единицу командами управления модулем, повторить данную процедуру несколько раз. Считать программным способом число из регистра данных модуля.

Перед началом работы убедитесь, что модули, используемые в работе, установлены в крейте в произвольных станциях с номерами от 1 до 23, а станции 24 и 25 заняты контроллером крейта КАМАК, подключенным через интерфейсную плату к компьютеру.

Каждый модуль получает через шины от контроллера управляющие команды и данные. Если команда модулем опо-

знана и правильно дешифрована, он выставляет на шине сигнал **X**. Таким образом, посылая модулю все возможные команды (**F(0) – F(31)**) по всем возможным субадресам (**A(0) – A(15)**) и отслеживая выставляемый модулем сигнал **X**, можно определить весь набор команд, используемых модулем.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

1. Составьте программу, которая бы выполняла указанную процедуру, используя модуль процедур для управления крейтом, который был написан вами в ходе лабораторной работы № 1.

2. С помощью индикатора магистрали проследите за передаваемыми командами и откликом модуля.

3. Проанализируйте полученный набор команд, определите назначение каждой команды используя приведенные данные из раздела описания команд КАМАК. Уточните полученные вами выводы в руководстве по эксплуатации модуля.

4. Установите на табло счетчика заданное число. Добейтесь его увеличения и уменьшения на единицу командами управления модулем. Программным способом считайте число из регистра данных модуля.

5. Определите максимальную скорость счета счетчика для использованного в работе оборудования, сравните ее с паспортным значением.

6. Определите факторы, влияющие на скорость работы. Проанализируйте код программы и постарайтесь его изменить для достижения максимально возможной скорости.

Ниже приведен пример программы, которая выполняет описанные выше действия и использует модуль команд для

контроллера КАМАК GEOSOFT. Пример модуля приведен в приложении 1. В данном примере модуль находится в станции 23. Поскольку передача каких-либо данных в модуль не нужна и совершенно не влияет на выполнение программы, то переменная data в теле программы не изменяется и имеет значение, равное 0.

```
program WhatCommand;

var f     : byte;          {Переменная для номера команды}
    a     : byte;          {Перемена для }
    data  : word;          {Данные, в этом примере не
                            важно какие }

begin
    for f=0 to 7 do        {Цикл по всем функциям}
       for a=0 to 15 do    {Цикл по всем субадресам}
          begin
             CamO(23, f, a, data);  {Посылаем
                                     команду модулю}
             if CamX then writeln(f,' ',a);
                            {Если опознана - печатаем f и a}
          end;
    for f=8 to 15 do
       for a=0 to 15 do
          begin
             CamDrv(23, f, a);
             if CamX then writeln(f,' ',a);
          end;
    for f=16 to 23 do      {}
       for a=0 to 15 do
          begin
             CamI(23, f, a, data);
             if CamX then writeln(f,' ',a);
          end;
    for f=24 to 31 do
       for a=0 to 15 do
          begin
             CamDrv(23, f, a);
             if CamX then writeln(f,' ',a);
```

```
        end;
end.
```

Блок циклов в программе повторяется, т. к. для передачи команд, по-разному использующих шины данных, используются различные программные процедуры: CamO записи, CamI для блока команд, использующих шины чтения, CamDrv для блока команд, не использующих шины данных. В принципе возможно написание единой управляющей процедуры для всего набора команд КАМАК, но, скорее всего, такая процедура будет неэффективна с точки зрения времени исполнения.

Данная процедура полезна для определения работоспособности модуля или для определения неизвестного набора команд для модулей, у которых по какой-либо причине отсутствует подробное описание.

Работа № 3.
Управление устройствами в системе КАМАК

ОБОРУДОВАНИЕ: Крейт КАМАК, сопряженный с компьютером; модули КАМАК: счетчик $СЧ_6$10, синхронизатор-таймер СТ, индикатор магистрали; генератор сигналов, частотомер.

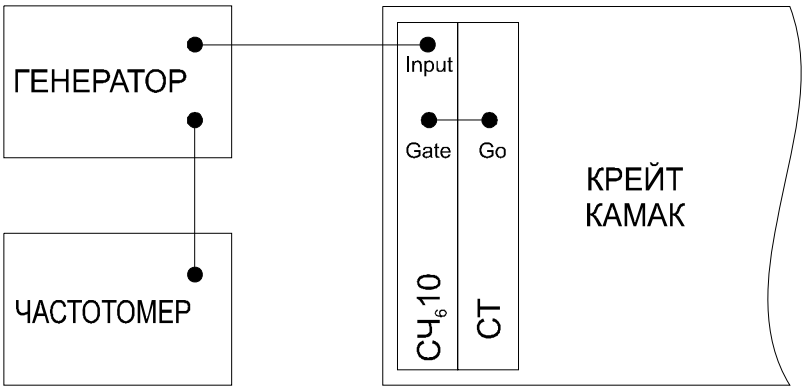
ЦЕЛЬ: Провести сопряжение стандартной аппаратуры с персональным компьютером, провести измерения.

КРАТКОЕ ОПИСАНИЕ РАБОТЫ: Изучить стандартный интерфейс используемых в работе приборов, команды и режимы дистанционного и автоматического управления. Собрать измерительную установку, состоящую из крейта КАМАК, модулей КАМАК, измерительной аппаратуры. Составить алгоритм управления и проведения измерений и написать программу. С помощью составленной программы провести необходимые измерения. Результаты представить в электронном виде, допускающем последующую математическую обработку. Провести статистическую обработку результатов, оценить погрешности, предложить методы для уменьшения погрешности при проведении повторных измерений.

Перед началом работы убедитесь, что станции 24 и 25 заняты контроллером крейта КАМАК подключенным через интерфейсную плату к компьютеру, используемые в работе модули установлены в крейте в свободных станциях.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

1. Соберите схему, изображенную на рисунке. Подключите генератор сигналов к разъему «Вход» модуля СЧ$_6$10 и параллельно к частотомеру. На вход «Gate» подключите разъем «Go» модуля СТ.



2. Составьте алгоритм измерений для моделирования частотомера.

3. Напишите программу, с помощью которой возможно было бы отмерять промежутки времени используя модуль СТ.

4. Откалибруйте модуль. Определите квант времени, т. е. сколько секунд приходится на одну запись в регистр данных модуля.

5. Подайте на вход счетчика СЧ$_6$10 сигнал с генератора сигналов, и через заданный промежуток времени прекратите счет. Рассчитайте частоту полученного сигнала.

6. Сравните полученные данные с показаниями частотомера. Повторите измерения, изменяя частоту подаваемого с генератора сигнала.

7. Результаты измерения представьте в электронном виде, допускающем последующую математическую обработку.

8. Проведите серию измерений со статистической обработкой результатов, оцените погрешности проведенных измерений. Предложите методы для уменьшения погрешности. При необходимости модифицируйте алгоритм измерений и используемое программное обеспечение. Повторите измерения по улучшенной схеме.

Работа № 4.
Сбор данных от внешних приборов

ОБОРУДОВАНИЕ: Крейт КАМАК сопряженный с компьютером; модули КАМАК: входной регистр, индикатор магистрали; микровольтметр, термопара.

ЦЕЛЬ: Провести сопряжение стандартной аппаратуры с персональным компьютером, провести измерения.

КРАТКОЕ ОПИСАНИЕ РАБОТЫ: Изучить стандартный интерфейс используемых в работе приборов, команды и режимы дистанционного и автоматического управления. Собрать измерительную установку, состоящую из крейта КАМАК, модулей КАМАК. К микровольтметру подключить градуированную термопару. Микровольтметр подключить к системе КАМАК. Составить управляющую программу. С помощью составленной программы провести необходимые измерения. Получить график зависимости температуры от времени при быстром нагревании и охлаждении образца малой массы. Провести статистическую обработку результатов, оценить погрешности, предложить методы для уменьшения погрешности при проведении повторных измерений.

Перед началом работы убедитесь, что станции 24 и 25 заняты контроллером крейта КАМАК, подключенным через

интерфейсную плату к компьютеру, используемые в работе модули установлены в крейте в свободных станциях.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

1. Ознакомьтесь с техническим описанием модулей и приборов, использующихся в работе.

2. Соберите необходимую установку. Термопара подключается к микровольтметру. Сигналы с разъема управления подаются на вход модуля «Входной регистр».

3. При необходимость проградуируйте термопару.

4. Составьте алгоритм получения данных микровольтметра. Для этого проанализируйте данные, приходящие на вход модуля «Входной регистр». Определите необходимые преобразования данных, составьте управляющую программу.

5. Получите данные с микровольтметра. Преобразуйте их в температуру. Напишите программу, позволяющую получать температуру на термопаре в зависимости от времени.

6. Получите последовательные значения температур тестового объекта в процессе его нагревания и охлаждения. Результат представьте в виде графика зависимости температуры от времени. Объясните вид полученной кривой.

7. Проведите серию измерений и сделайте статистическую обработку результатов, оцените погрешности проведенных измерений.

8. Предложите методы для уменьшения погрешности. При необходимости модифицируйте алгоритм измерений и используемое программное обеспечение.

9. Повторите измерения по улучшенной схеме. Сравните полученные результаты.

Работа № 5.
Управление шаговым двигателем

ОБОРУДОВАНИЕ: Крейт КАМАК, сопряженный с компьютером; модули КАМАК: модуль управления шаговым двигателем МУШД-2, индикатор магистрали; шаговый двигатель подключенный к системе позволяющей контролировать количество шагов.

ЦЕЛЬ: Научиться управлять шаговым двигателем, плавно регулировать скорость вращения.

КРАТКОЕ ОПИСАНИЕ РАБОТЫ: Изучить стандартный интерфейс используемых в работе приборов, команды и режимы дистанционного и автоматического управления. Собрать измерительную установку, состоящую из крейта КАМАК, модулей КАМАК и шагового двигателя. Составить алгоритм управления и плавного изменения скорости и написать программу. Выяснить соотношение между шагами и оборотами. С помощью составленной программы провести необходимые измерения. Провести статистическую обработку результатов, оценить погрешности, предложить методы для уменьшения погрешности при проведении повторных измерений.

Перед началом работы убедитесь, что станции 24 и 25 заняты контроллером крейта КАМАК, подключенным через интерфейсную плату к компьютеру, используемые в работе модули установлены в крейте в свободных станциях.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

1. Ознакомьтесь с техническим описанием модулей и приборов, использующихся в работе.

2. Составьте алгоритм управления шаговым двигателем. Напишите программу, которая позволяет совершать перемещения шаговым двигателем с постоянной скоростью.

3. Выясните соотношение между количеством шагов и перемещением.

4. Составьте алгоритм для плавного изменения скорости вращения шагового двигателя и напишите управляющую программу. Входные данные программы: начальное и конечное положение. Шаговый двигатель должен начинать движение на медленной скорости, затем плавно увеличить скорость вращения, основное перемещение должно осуществляться на максимальной скорости и затем плавная остановка. При этом должно осуществляться позиционирование с заданной точностью.

5. Проведите анализ полученных результатов, оцените погрешности позиционирования.

6. Предложите методы для уменьшения погрешности. При необходимости модифицируйте алгоритм управления и используемое программное обеспечение.

7. Повторите измерения по улучшенной схеме.

Работа № 6.
Создание макета измерительной установки для оптического эксперимента

ОБОРУДОВАНИЕ: Крейт КАМАК, сопряженный с PC; модули: модуль управления шаговым двигателем, АЦП, индикатор магистрали; спектрометр ДФС-24; ФЭУ с блоками питания и усилителем сигнала; шаговый двигатель, подключенный к системе развертки спектрометра, блок управления и питания шагового двигателя.

ЦЕЛЬ: Смоделировать процесс автоматического измерения оптического сигнала.

КРАТКОЕ ОПИСАНИЕ РАБОТЫ: В ходе выполнения работы необходимо изучить схему спектральной установки и особенности проведения спектральных измерений. Предло-

жить алгоритм измерений, составить программу, реализующую этот алгоритм. Провести регистрацию оптического спектра. Результаты представить в электронном виде, допускающем последующую математическую обработку. Провести статистическую обработку результатов, оценить погрешности, предложить методы для уменьшения погрешности измерений.

Перед началом работы убедитесь, что станции 24 и 25 заняты контроллером крейта КАМАК, подключенным через интерфейсную плату к компьютеру, а используемые в работе модули установлены в крейте в свободных станциях.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ:

1. Ознакомьтесь с техническими описаниями всех приборов и модулей, используемых в данной работе.

2. Составьте алгоритм проведения измерений спектра сигнала, лежащего в оптическом диапазоне. При составлении алгоритма следует учесть, что сигнал записывается в дискретном режиме. Это означает, что значение интенсивности оптического сигнала измеряется в отдельных заранее выбранных точках, которые, как правило, располагаются на равных спектральных промежутках друг от друга (под равными спектральными промежутками в данном случае понимаются точки, отстоящие друг от друга на равные промежутки длин волн или частот). Поэтому процесс измерений можно разделить на два блока. Первый – это выбор заданной длины волны или частоты. Второй – это измерения интенсивности оптического сигнала в выбранной точке. Затем эти два блока повторяются.

3. Напишите программу, реализующую алгоритм. Обратите особое внимание на возникающие в процессе измерений ошибки, которые могут привести к увеличению шумов при сборе данных. Используйте программные модули, подготовленные в предыдущих лабораторных работах.

4. Получите и запишите на диск измеренные данные. Они должны быть представлены в виде пар чисел: длина волны (частота), интенсивность оптического сигнала.

5. После проведения измерений необходимо построить полученный спектр. Для этого можно воспользоваться любой программой, позволяющей графически представлять данные (например: SigmaPlot, MicroCalc Origin, Mathematica, Mathlab и т. д.), либо воспользоваться программными модулями, разработанными самостоятельно.

6. Проанализируйте полученные результаты.

7. Предложите методы повышения эффективности алгоритма измерений.

8. Предложите алгоритм измерения, с помощью которого можно было бы уменьшить шумы в записи сигнала.

9. Реализуйте предложенные алгоритмы. Проведите повторные измерения.

10. Сравните полученные результаты.

Приложение 1

Пример библиотеки для работы с системой КАМАК с контроллером, использующем параллельный интерфейс.

```
unit CAMAC;

interface

procedure CamO(const n, F, a:byte; const d:Word);
procedure CamO24(const n, F, a:byte; const d:longint);
procedure CamI(const n, F, a:byte; var d:Word);
procedure CamI24(const n, F, a:byte; var d:longint);
procedure CamDrv(const n, F, a:byte);
procedure CamCl(const i:byte);
procedure CamL(var l:byte);
Function CamQ: boolean;
Function CamX: boolean;

implementation
{$R-,S-}
procedure CamO(const n, F, a:byte; const d:Word);
begin
     portw[$0240]:=$000000;
     portw[$0241]:=(d and $00FF00) shr 8;
     portw[$0242]:=d and $0000FF;
     portw[$0243]:=a;
     portw[$0244]:=F;
     portw[$0245]:=n;
     portw[$0247]:=1;
end;

procedure CamO24(const n, F, a:byte; const d:longint);
begin
     portw[$0240]:=(d and $FF0000) shr 16;
     portw[$0241]:=(d and $00FF00) shr 8;
     portw[$0242]:=d and $0000FF;
     portw[$0243]:=a;
     portw[$0244]:=F;
     portw[$0245]:=n;
```

```
      portw[$0247]:=1;
end;
procedure CamI(const n, F, a:byte; var d:Word);
var c : integer;
begin
      portw[$0243]:=a;
      portw[$0244]:=F;
      portw[$0245]:=n;
      portw[$0247]:=1;
         d:=((portw[$024A] and
         $0000FF)*$100)+(portw[$024B] and $0000FF);
end;

procedure CamI24(const n, F, a :byte; var d:longint);
var c,e : longint;
begin
      portw[$0243]:=a;
      portw[$0244]:=F;
      portw[$0245]:=n;
      portw[$0247]:=1;
         d:=((portw[$0249] and
         $0000FF)*$10000)+((portw[$024A] and
         $0000FF)*$100)+( portw[$024B] and $0000FF);
end;

procedure CamDrv(const n, F, a:byte);
var c,e:longint;
begin
      portw[$0243]:=a;
      portw[$0244]:=F;
      portw[$0245]:=n;
      portw[$0247]:=1;
end;

procedure CamCl(const i:byte);
begin
      portw[$0246]:=i;
end;

procedure CamL(var l:byte);
begin
```

```
    I:=(portw[$0248] and $00007C) shr 2;
end;

Function CamQ: boolean;
begin
    CamQ:=(portw[$0248] and $000001) = 1;
end;

Function CamX: boolean;
begin
    CamX:=((portw[$0248] and $000002) shr 1) = 1;
end;

end.
```

Приложение 2

Пример библиотеки для работы с системой КАМАК с контроллером, использующем последовательный интерфейс.

```
unit camac;
interface

procedure GTW;
function F0(K,N,A,F: Byte): integer;
                                { Функция типа «Ввод” }

function CTRLQ(K: byte): boolean;

procedure F16(K,N,A,F: Byte; W: Word);
                                { Процедура типа «Вывод” }
                                { K - номер канала }
                                { N - номер модуля }
                                { A - субадрес }
                                { F - функция }

procedure Zero;

implementation
uses CRT;

const                           {Адреса регистров драйвера.}
    RGAS=$310;
    RGAM=$314;
    RGDS=$312;
    RGDM=$316;
    RGZ=$318;
var
    AdrS, AdrM: byte;

procedure GTW;          {Проверка конца передачи по посл. каналу.}
var
    GOT, N :integer;
begin
    N:=0;
```

```
    repeat
      GOT:=port[RGZ];
      N:=N+1;
    until (GOT and 16=16) {Выход если готов }
    or (N>100);                    {или отсутствует связь}
    if N>100 then WRITELN          ('Нет связи с контроллером.')
end;

function F0(K,N,A,F: Byte): integer;
var
    STB,MLB: integer;

begin
    AdrS:=(8 shl K)+((N and 24) shr 3)+4;
    AdrM:=((N and 7) shl 5)+(A shl 1);
    port[RGAS]:=(8 shl K)+8;
    port[RGAM]:=0;
    port[RGDS]:=0;
    port[RGDM]:=F;
    GTW;

    port[RGAS]:=AdrS;
    port[RGAM]:=AdrM;
    GTW;

    STB:=port[RGDS];
    MLB:=port[RGDM];
    F0:=(STB shl 8)+MLB
end;

function CTRLQ(K: byte): boolean;        { Проверка состояния Q. }
begin
    CTRLQ:=false;
    port[RGAS]:=(8 shl K)+4;
    port[RGAM]:=0;
    GTW;
    if (port[RGDS] and $80) = $80 then CTRLQ:= true
end;

procedure F16(K,N,A,F: Byte; W: Word);
begin
```

```
      port[RGAS]:=(8 shl K)+8;
      port[RGAM]:=0;
      port[RGDS]:=0;
      port[RGDM]:=F;
      AdrS:=(8 shl K)+((N and 24) shr 3)+8;
      AdrM:=((N and 7) shl 5)+(A shl 1);
      port[RGAS]:=AdrS;
      port[RGAM]:=AdrM;
      port[RGDS]:=Hi(W);
      port[RGDM]:=Lo(W)
end;

procedure Zero;
var
      K:  byte;
begin
      for K:=1 to 4 do
      begin
         port[RGAS]:=(8 shl K)+8;
         port[RGAM]:=0;
         port[RGDS]:=1;
         port[RGDM]:=0
      end;
      delay(100)
end;

end.
```

Приложение 3

Заголовочный файл и пример реализации некоторых функций библиотеки «gen_func.dll» для работы с КАМАК под управлением контроллера с параллельным интерфейсом через драйвер «**genport»,** для операционных систем Windows NT/2000/XP.

С полными текстами приведенных библиотек можно ознакомиться по адресу http://www.kirensky.ru/master/camac/.

«main.h»

```
void __declspec(dllexport) __stdcall Camac_Port_Init();
// Функция, инициализирующая работу с системой КАМАК,
// открытие «файла драйвера устройства» на чтение и на запись.

bool __declspec(dllexport) __stdcall Camac_Port_Close();
// Функция, закрывающая «файл драйвера устройства» от чтения и записи.
// В случае успешного выполнения этих операций возвращает True,
// в противном случае Fasle.

void __declspec(dllexport) __stdcall CamO(const unsigned char n, const unsigned
char f, const unsigned char a, long d);
// Функция, отправляющая модулю команду и данные.
// В качестве аргументов передается адрес ячейки модуля (n), субадрес (a),
//функция (f) и данные (d). Используются шины записи.

long __declspec(dllexport) __stdcall CamI(const unsigned char n, const unsigned
char f, const unsigned char a, long d);
// Функция, отправляющая модулю команду и получающая от него данные.
// В качестве аргументов передается адрес // ячейки модуля (n), субадрес (a),
функция (f) и данные (d). Используются шины чтения.

void __declspec(dllexport) __stdcall CamDrv (const unsigned char n, const un-
signed char f, const unsigned char a);
// Функция, отправляющая модулю управляющую команду без использования
// шин данных. В качестве аргументов передается адрес ячейки модуля (n),
// субадрес (a) и функция (f).
```

```
void __declspec(dllexport) __stdcall Waiting(const unsigned char n);
// Функция приостановки работы программы до получения
// сигнала Q от модуля расположенного в станции с номером n

void __declspec(dllexport) __stdcall WaitingExt(const unsigned char n, const un-
signed char a);
// Функция приостановки работы программы до получения сигнала Q
// от модуля расположенного в станции с номером n при проверке субадреса a

void __declspec(dllexport) __stdcall SetC();
// Функция, вызывающая передачу сигнала гашения.

void __declspec(dllexport) __stdcall SetZ();
// Функция, вызывающая передачу сигнала подготовки.

void __declspec(dllexport) __stdcall SetI();
// Функция, вызывающая передачу сигнала блокировки.

unsigned int __declspec(dllexport) __stdcall CamL();
// Функция, вызывающая проверку сигнала требования на обслуживание.
// Возвращает адрес модуля, выставившего требование. Наличие сигнала L
// означает необходимость прервать текущую программу и начать выполнение
// обслуживания модуля.

bool __declspec(dllexport) __stdcall CamQ();
// Проверка сигнала подтверждения Q.

bool __declspec(dllexport) __stdcall CamX();
// Проверка сигнала X. Наличие сигнала обозначает, что
// модуль принял адресованную ему команду и готов к ее исполнению.

long __declspec(dllexport) __stdcall CamI(const unsigned char n, const unsigned
char f, const unsigned char a);
// Проверка сигнала блокировки I.
```

«main.cpp»

```
HANDLE hndFile_Write;
// Переменная устройства для записи.
HANDLE hndFile_Read;
// Переменная устройства для чтения.
```

```
void __stdcall Camac_Port_Init()
{
    hndFile_Read = CreateFile(
            "\\\\.\\GpdDev",
            GENERIC_READ,
            FILE_SHARE_READ,
            NULL,
            OPEN_EXISTING,
            0,
            NULL);

    if (hndFile_Read == INVALID_HANDLE_VALUE)
    {
       throw(/*CAMAC_Except::err=
            */"Unable to open the device.");
       exit(1);
    }
    hndFile_Write = CreateFile(
            "\\\\.\\GpdDev",
            GENERIC_WRITE,
            FILE_SHARE_WRITE,
            NULL,
            OPEN_EXISTING,
            0,
            NULL);
    if (hndFile_Write == INVALID_HANDLE_VALUE)    {
       throw(/*CAMAC_Except::err=
            */"Unable to open the device.");
       exit(1);
    }
}

bool __stdcall Write_To_Port(GENPORT_WRITE_INPUT NewInformation)
{
    BOOL            IoctlResult;
    LONG            IoctlCode;
    ULONG           DataLength;
    ULONG           ReturnedLength;

    IoctlCode = IOCTL_GPD_WRITE_PORT_UCHAR;
```

```
    DataLength = offsetof(GENPORT_WRITE_INPUT, CharData)
    +sizeof(NewInformation.CharData);
    IoctlResult = DeviceIoControl(
            hndFile_Write,
            IoctlCode,
            &NewInformation,
            DataLength,
            NULL,
            0,
            &ReturnedLength,
            NULL);
    if (IoctlResult)
    {
       return true;
    }
    else
    {
       return false;
    }
}

void __stdcall CamO(const unsigned char n, const unsigned char f, const unsigned
char a, long d)
{
    GENPORT_WRITE_INPUT InputBuffer;

    InputBuffer.PortNumber = 0x0;
    InputBuffer.CharData = (d & 0xFF0000) >> 16;
    Write_To_Port(InputBuffer);

    InputBuffer.PortNumber = 0x1;
    InputBuffer.CharData = (d & 0x00FF00) >> 8;
    Write_To_Port(InputBuffer);

    InputBuffer.PortNumber = 0x2;
    InputBuffer.CharData = d & 0x0000FF;
    Write_To_Port(InputBuffer);

    InputBuffer.PortNumber = 0x3;
    InputBuffer.CharData = a;
    Write_To_Port(InputBuffer);
```

```
    InputBuffer.PortNumber = 0x4;
    InputBuffer.CharData = f;
    Write_To_Port(InputBuffer);

    InputBuffer.PortNumber = 0x5;
    InputBuffer.CharData = n;
    Write_To_Port(InputBuffer);

    InputBuffer.PortNumber = 0x7;
    InputBuffer.CharData = 0x1;
    Write_To_Port(InputBuffer);
}
void __stdcall SetZ()
{
    GENPORT_WRITE_INPUT InputBuffer;

    InputBuffer.PortNumber = 0x6;
    InputBuffer.CharData = 0x1;
    Write_To_Port(InputBuffer);
}

bool __stdcall CamQ()
{
    ULONG   PortNumber;
    UCHAR   Data;

    PortNumber = 0x8;
    Data = Read_From_Port(PortNumber);
    return (Data & 0x000001) == 1;
}
void __stdcall Waiting(const unsigned char n)
{
    do
    {
       CamDrv(n, 8, 0);
       Sleep(1);
    } while (!CamQ());
}
```

Литература

1. Ступин Ю. В. Методы автоматизации физических экспериментов с помощью ЭВМ. М., Энергоатомиздат, 1983.
2. Певчев Ю. Ф., Финогенов К. Г. Автоматизация физического эксперимента. М., Энергоатомиздат, 1986.
3. Задков В. Н., Пономарев Ю. В. Компьютер в эксперименте. Архитектура и программные средства систем автоматизации. М., Наука, 1988.
4. Курочкин С. С. Системы КАМАК–ВЕКТОР. М., Радио и связь., 1981.
5. Компьютеры в оптических исследованиях. /ред. Б. Фриден/ М., Мир, 1983.
6. Отнес Р., Эноксон Л. Прикладной анализ временных рядов. М., Мир, 1982.
7. Новиков Ю. В., Калашников О. А., Гуляев С. Э. Разработка устройств сопряжения. М., ЭКОМ, 1997.
8. Кулаков В. Программирование на аппаратном уровне. Специальный справочник. СПб., Питер, 2003.
9. Сван Т. Освоение Borland C++5. Киев, Диалектика, 1996.
10. Носач В. В. Решение задач аппроксимации с помощью персональных компьютеров. М., МИКАП, 1994.
11. Смит Дж. Сопряжение компьютеров с внешними устройствами. Уроки реализации. М., Мир, 2000.
12. CAMAC — A Modular Instrumentation System for Data Handling. Revised Description and Specification. Report. EUR 4100e, CEC, Luxembourg 1972; deutsche Ьbersetzung; EUR 4100d; revised form: IEEE Standard 583-1975, IEC Recommendation 516; Bloc Transfers in CAMAC Systems, Supplement EUR 4100e, CEC, Luxembourg 1977; IEEE Standard 683-1976.
13. CAMAC — Organization of Multi-Crate System. Specification of the Branch Highway and CAMAC Crate Controller Typ A, Report EUR 4600e, CEC, Luxembourg 1972; revised form: IEEE Standard 596-1976.
14. CAMAC — A Modular Instrumentation System for Data Handling, Specification of Amplitude Analogue Signals, Report EUR 5100e, CEC, Luxembourg 1972; CAMAC Bulletin No. 8, 28 (1973).
15. Государственный стандарт Союза ССР. Единая система стандартов приборостроения. Система КАМАК. Крейт и сменные блоки. Требования к конструкции и интерфейсу. ГОСТ 26.201–80.
16. Науман Г., Майлинг В., Щербина А. Стандартные интерфейсы для измерительной техники. М. Мир, 1982.
17. http://sine.ni.com/apps/we/nioc.vp?cid=1525&lang=USpxi_tech.htm
18. http://www.vmebus-systems.com/
19. http://www.ee.ualberta.ca/archive/vmefaq.html#intro
20. http://www.rtsoft.ru/ect/
21. http://www.picmg.com/specifications.stm

Оглавление