Сибирское отделение Российской Академии наук
ИНСТИТУТ ЯДЕРНОЙ ФИЗИКИ им.Г.И. Будкера

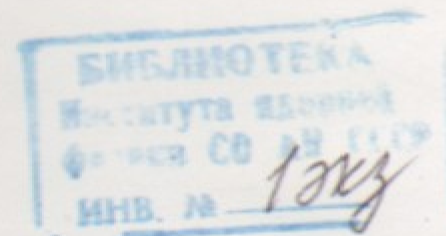
Б.79
1998

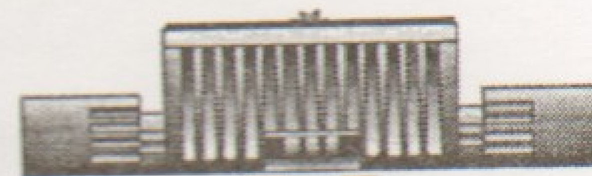
Д.Ю. Болховитянов, Р.Г. Громов,
И.Л. Пивоваров, Ю.И. Эйдельман

ПРОЕКТ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ СИСТЕМЫ УПРАВЛЕНИЯ КОМПЛЕКСОМ ВЭПП-5

ИЯФ 98-53

http://www.inp.nsk.su/publications


БИБЛИОТЕКА
Института ядерной
физики СО АН СССР
ИНВ. № 13кз



НОВОСИБИРСК
1998

Проект программного обеспечения системы управления комплексом ВЭПП-5

Д.Ю.Болховитянов, Р.Г.Громов,
И.Л.Пивоваров, Ю.И.Эйдельман

Институт ядерной физики им Г.И.Будкера
630090 Новосибирск, Россия

Обсуждаются принципы построения ПО системы управления большой электрофизической установкой, которой является комплекс ВЭПП-5, в рамках так называемой "стандартной модели". Рассматриваются протоколы взаимодействия между компонентами ПО трех уровней системы (рабочие станции, интеллектуальные транспьютерные контроллеры и связывающий их сервер). Приводятся данные о текущем состоянии ПО и показывается адекватность принятой архитектуры.

The project of a control system software for a VEPP-5 complex

D.Yu. Bolkhovityanov, R.G. Gromov,
I.L. Pivovarov, Yu.I. Eidelman

Budker Institute of Nuclear Physics
630090 Novosibirsk, Russia

Abstract

The design principles of the VEPP-5 control system software, conforming the so-called "standard model", are being discussed. The protocols for interconnection between the software in three levels of the system (workstations, intellectual transputer controllers and a server, connecting them) are being contemplated. Current state of the software is described and the validity of the selected solutions is being shown.

©*Институт ядерной физики им. Г.И. Будкера СО РАН, Россия*

Содержание

1 Основные концепции **5**
1.1 Требования к системе управления 6
1.2 Архитектура программного обеспечения 6
1.3 Где должны работать программы управления 8
1.4 Единицы информационного обмена 9
1.5 Обмен данными между прикладными программами и сервером 9
1.6 Дополнительные программные модули в контроллере 10
1.7 Исполнитель отдельных NAF'ов 11
1.8 Конфигурирование системы: база данных 11

2 Протоколы, алгоритмы и форматы данных **13**
2.1 Рабочий цикл 13
2.2 Сервер 14
2.2.1 Компоненты сервера 14
2.2.2 Структуры данных, используемые в сервере 15
2.2.3 Алгоритмы работы сервера 16
2.3 Клиентская библиотека 18
2.3.1 Состав клиентской библиотеки 18
2.3.2 Алгоритмы подсистемы связи с сервером 23
2.4 Высокоуровневый интерфейс 31
2.5 Протокол связи клиентов и сервера 32
2.5.1 Формат пакетов 32
2.5.2 Доступ к физическим каналам 34
2.6 ПО в транспьютере 36
2.7 База данных 38
2.7.1 Общие принципы 38
2.7.2 Адресация объектов 38

2.7.3 Форматы файлов, составляющих исходное описание и последовательность компиляции БД 39
2.7.4 Редактирование 40
2.8 Некоторые общие принципы 41
2.8.1 Форматы данных межплатформно-переносимы . . . 41
2.8.2 Никогда не ждать ответа на запрос 42
2.8.3 Буфера растут по мере надобности 43

3 Текущее состояние 46

Глава 1

Основные концепции

Выбранная архитектура аппаратной конфигурации и структуры программного обеспечения (ПО) является естественным продолжением сложившегося в институте подхода к построению систем управления крупными электрофизическими комплексами (см., например, [1]). Несмотря на разнообразие выбираемых типов управляющих компьютеров, контроллеров и стандартов для электроники, он полностью соответствует укоренившейся практически всюду "Стандартной модели" систем управления [2]. Эта модель предполагает 3-уровневую аппаратную реализацию: на нижнем уровне – оконечная электроника в каком-либо стандарте, на следующем – интеллектуальный контроллер (или несколько контроллеров), обслуживающий эту электронику и связывающий ее с верхним уровнем, представленным рабочими станциями (компьютерами).

К моменту начала разработки системы технические и финансовые условия управления комплексом ВЭПП-5 диктовали практически однозначный выбор конкретной реализации каждого уровня: стандарт CAMAC для оконечной электроники, интеллектуальные контроллеры на основе транспьютеров и рабочие станции на основе персональных компьютеров типа Pentium, работающие под ОС семейства UNIX. При разработке архитектуры ПО предпочтение отдавалось таким решениям, которые бы позволили в будущем максимально безболезненным образом внедрить в систему управления аппаратуру другого типа, например, в другом стандарте электроники, другой тип интеллектуальных контроллеров и т.п. И, конечно, выбирались решения, обеспечивающие максимальную возможность использования стандартных программных и аппаратных средств.

1.1 Требования к системе управления

- Управление через крейты CAMAC.
- Количество крейтов – несколько десятков (≤100).
- Количество каналов – несколько тысяч (≤10000).
- Одновременный доступ нескольких программ к одной аппаратуре без конфликтов (прозрачно для программ).
- Простота переконфигурации при изменении состава аппаратуры.
- Режим работы (основной части аппаратуры) – опрос с частотой около 1Гц.
- Максимальная возможность использования стандартных программных и аппаратных средств.

1.2 Архитектура программного обеспечения

Система состоит из трех основных частей (см. рис.1.1): ПО в транспьютере, программы-сервера в PC, подключенной к транспьютеру, и собственно программ управления (в дальнейшем – "*клиентские программы*" или "*клиенты*").

Вся логика управления заложена в клиентских программах, которые работают так, как будто каждая из них является единственным пользователем CAMAC-крейтов.

В транспьютере располагается процесс-диспетчер, обслуживающий канал связи с сервером в PC, процессы-драйверы отдельных устройств, взаимодействующие с диспетчером. Кроме того, в транспьютере могут находиться и другие специализированные процессы.

В сервере имеется полная информация о конфигурации системы (статическая база данных) и состоянии всех устройств (динамическая база данных). Сервер собирает запросы на чтение/запись от клиентских программ и переправляет их в транспьютер, а при получении ответов размещает их в динамической базе данных и раздает клиентам. Клиенты, таким образом, имеют дело только с сервером, а уже он сам разбирается, какой запрос следует направить в какой крейт и предотвращает

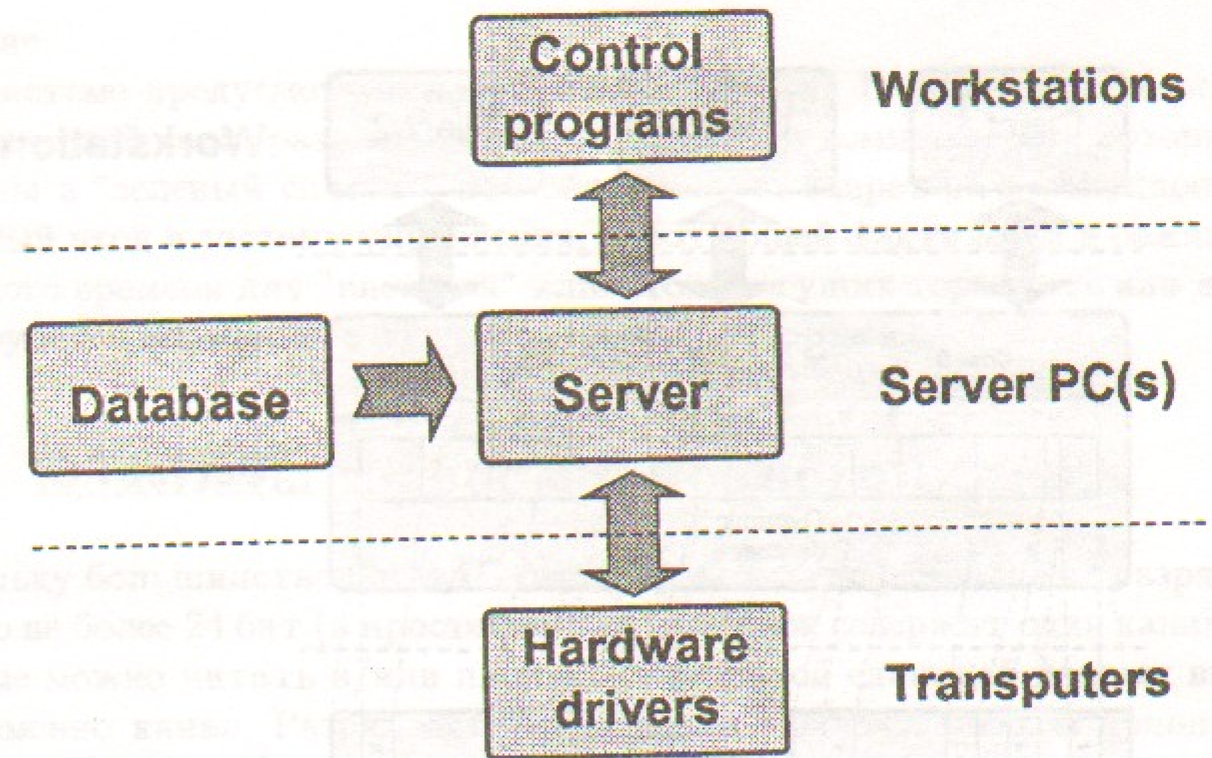


Рис. 1.1: Три уровня ПО.

дублирование запросов. Кроме того, он же обеспечивает доступ к базе данных, в частности, отвечает за трансляцию имен каналов (CNS – channel name service (аналогично DNS)) – возвращает клиентской программе по имени канала его адрес в динамической базе данных и иные свойства (коэффициенты и параметры).

Клиентские программы реализуют логику управления и включают библиотеку связи с сервером (в дальнейшем – "*клиентская библиотека*"). Клиентская библиотека кроме простой пересылки запросов на чтение/запись выполняет некоторые действия "за кулисами", обеспечивая более "прозрачную" связь и повышая производительность системы (подробнее см. раздел 1.5).

Все CAMAC-блоки представляются в виде набора *каналов*, по одному каналу на каждую контролируемую или управляемую физическую величину. Последовательность таких наборов каналов образует *динамическую базу данных*, описывающую текущее состояние установки (см рис.1.2). Для ссылки на канал используется его порядковый номер в динамической БД.

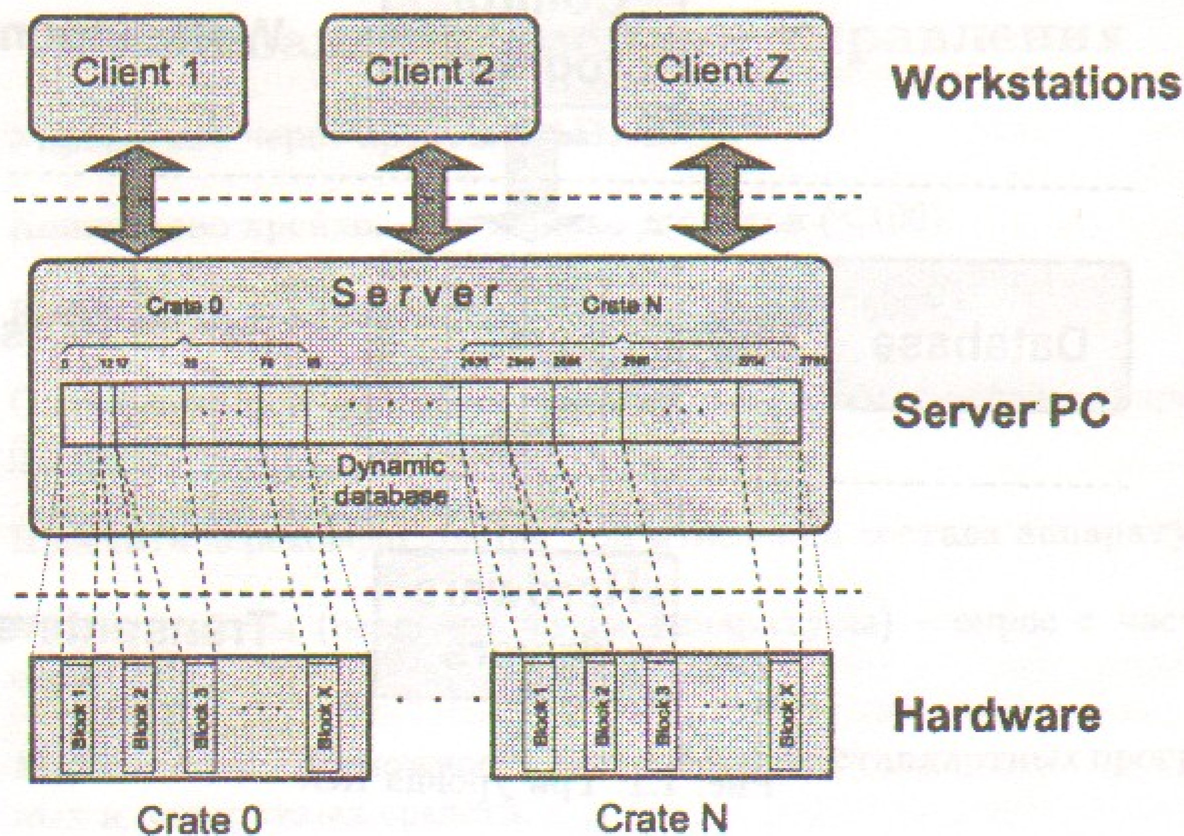


Рис. 1.2: Организация каналов и доступа к ним.

1.3 Где должны работать программы управления

Возможны два варианта. Первый: клиентские программы работают только в машине, непосредственно подключенной к транспьютеру и общаются с сервером через протокол UNIX. Второй: они работают в и других машинах, и общаются с сервером через протокол TCP.

Хотя более разумным в силу ряда причин представляется второй вариант, точно определить это можно лишь при практическом использовании[1]. Поэтому сервер разрешает оба типа соединения[2]. Программная реализация от этого практически не усложняется, поскольку отличается лишь само установление соединения: сервер "слушает" два разных входа вместо одного, а в клиентскую библиотеку добавлен простейший интеллект, определяющий, по какому протоколу установить соединение.

[1] Но для небольшой установки всегда достаточен первый вариант, когда и сервер, и прикладные программы работают в одной ЭВМ.

[2] При работе программ в одной машине устанавливать между ними связь по TCP невыгодно, т.к. из-за накладных расходов она в 1.5-10 раз более медленная, чем по UNIX.

В системе предусмотрен контроль доступа (по IP-адресам) к серверу из внешней сети, разрешающий его лишь тем компьютерам, которые занесены в "зеленый список". Это обеспечивает запрет на несанкционированный вход в систему управления, работающей фактически в режиме реального времени для "внешних" клиентов, могущих тормозить или даже нарушать нормальное функционирование системы.

1.4 Единицы информационного обмена

Поскольку большинство CAMAC-блоков работают с "каналами" разрядностью не более 24 бит (в простейшем случае блок содержит один канал), которые можно читать и/или писать, то основной единицей данных выбран именно канал. Разрядность канала для удобства работы принята 32 бита – размер типа `int` в большинстве ОС UNIX и в транспьютере.

Такой подход позволяет унифицировать работу с блоками разного типа: работу с каналами можно свести к операциям "прочитать канал" и "записать канал". Упрощается создание как клиентских программ, так и интеллектуальной программы-сервера.

В случаях, когда разумной единицей информации является, например, 20 байт, они рассматриваются как последовательный набор из 5 каналов[3].

Поскольку основная работа осуществляется не с одиночными каналами, то они группируются в соответствующие логические "элементы", но это происходит уже в программах управления. Сервер же никак не заботится об этом вопросе и имеет дело лишь с отдельными каналами, ничего не зная об их смысловом содержании. (См. также раздел 1.8.)

1.5 Обмен данными между прикладными программами и сервером

Большинство программ управления каждый цикл читают некоторый набор каналов и, по необходимости, записывают значения в некоторые каналы. Поэтому, в основном их взаимодействие с сервером сводится к отсылке запроса "считай такие-то каналы, и запиши данные в такие-то каналы" и последующему ожиданию ответа. Как вариант, программа

[3] Собственно говоря, любые сложные структуры данных (не использующие указателей) являются лишь байтовым массивом.

может не дожидаться прихода ответа, а работать дальше (например, реагировать на действия оператора). Когда же ответ придет, то она его должным образом интерпретирует и посылает, если нужно, следующий запрос.

Так как большую часть времени программа читает одни и те же каналы (принцип локальности)[4], то постоянно высылать одни и те же запросы на чтение – неразумная трата производительности. Поэтому предусмотрена возможность "подписки", когда клиентская программа один раз говорит "шли мне такие-то данные каждый цикл", и затем сервер посылает результаты измерений каждый раз, без всяких лишних требований. В дальнейшем программа может "подписаться" на другой набор каналов, или "отписаться" вовсе, подписавшись на пустой набор.

1.6 Дополнительные программные модули в контроллере

В большинстве случаев вышеописанная схема, когда в транспьютере находятся лишь простейшие драйверы и обмен данными между сервером и клиентами производится раз в секунду, вполне достаточна. Однако, в некоторых случаях требуется быстрая реакция на внешние события (а Ethernet – не realtime-сеть!), иногда нужна более высокая частота опроса, и, наконец, не всегда удобно ограничиваться интерпретацией данных как набора каналов.

Для наших нестандартных случаев в системе предусмотрена поддержка дополнительных возможностей, в целом практически не усложняющая систему.

Во-первых, кроме драйверов в транспьютер можно загрузить дополнительные программные модули, которые не связаны ни с какими каналами и "молча" занимаются своим делом (хотя они могут одновременно являться и драйверами – это ничему не противоречит). В частности, в качестве такого модуля может выступать "сторож" – мониторирующий процесс, производящий контроль за текущим состоянием установки и выводящий информацию на ЦДР.

[4] Вообще говоря, это является одним из краеугольных моментов организации клиентских программ: они не представляют собой "монстров", охватывающих весь спектр управляющих действий на установке. Наоборот, управление представлено большим количеством как правило достаточно простых программ, реализующих ограниченный тип действий. Поэтому такие клиентские программы работают с ограниченным списком каналов, а не со всей динамической базой данных.

Во-вторых, клиентская программа, желающая работать с "нестандартной информацией", открывает дополнительное соединение, через которое может сообщаться с "интеллектуальными модулями". При этом протокол взаимодействия с конкретным модулем никак не лимитируется, и сервер просто пересылает информацию в обоих направлениях, следя лишь за тем, чтобы такой асинхронный обмен не мешал основной работе.

1.7 Исполнитель отдельных NAF'ов

Один "дополнительный" модуль необходим с самого начала – это исполнитель NAF'ов. Поскольку динамическая загрузка драйверов отсутствует, то возникает проблема: как быть, если появился новый блок, а драйвера к нему еще нет? Ведь драйвер "с ходу" не напишешь, его надо отлаживать. Для сокращения времени отладки до минимума алгоритм работы с новым блоком отрабатывается сначала "руками", NAF за NAF'ом, а потом уже по готовой схеме пишется драйвер.

Поскольку отработка NAF'ов является стандартной и весьма простой задачей, то, она не возложена на отдельный модуль, а непосредственно встроена в систему. Т.е. поддержка этих операций есть непосредственно в сервере и диспетчере, а в клиентской библиотеке имеется функция "выполни NAF".

1.8 Конфигурирование системы: база данных

Поскольку система управления должна быть гибкой, позволять легко менять конфигурацию аппаратуры и добавлять новые драйверы, то вся информация о конфигурации хранится в базе данных.

При запуске сервер и программа в транспьютере считывают базу данных и берут из нее информацию о расположении блоков в крейтах и каналов в блоках. Каждый крейт, блок и канал имеют свое имя. Имена блоков уникальны среди всех крейтов, а каналов – внутри блока.

Программы управления, естественно, также не содержат "зашитых" номеров каналов, а считывают их при запуске из базы данных через сервер.

В базе данных содержатся также коэффициенты преобразования кодов, используемых при работе с CAMAC-блоками, в физические величины – для каждого канала. Например, поскольку все АЦП измеряют

напряжение, а контролируемый параметр может являться током, то в базе данных будет содержаться коэффициент пересчета этого напряжения в ток.

Кроме того, база данных содержит информацию о группировке отдельных каналов в логические элементы, об их взаимосвязи и т.д. Более подробно база данных рассматривается далее.

Глава 2

Протоколы, алгоритмы и форматы данных

Данный раздел содержит конкретные описания протоколов, алгоритмов и форматов данных, используемых в системе UCAM. Некоторая детальность в описании тех или иных моментов организации ПО вызвана тем, что это попытка построения системы управления под общепринятой ОС типа UNIX, и хотелось, чтобы другим разработчикам не приходилось разбираться с теми же ошибками.

2.1 Рабочий цикл

Сервер работает в циклическом режиме, с частотой порядка 1Гц. При этом цикл состоит из следующих тактов: 1) "свободный" такт, в течение которого собираются запросы от клиентских программ; 2) пересылка запросов транспьютеру; 3) такт измерения; 4) прием результатов от транспьютера; 5) рассылка ответов клиентам. Запросы от клиентов, приходящие на тактах 2...5, ставятся в очередь на исполнение на следующем цикле.

Момент пересылки запросов в транспьютер выбирается не сервером, а транспьютером, путем присылки серверу короткого пакета "давай их сюда". Результаты пересылаются также в момент, выбранный транспьютером, в пакете "на, получи".

При общении сервер–транспьютер набор каналов, на которые посланы запросы, и тех, на которые прислан результат, не обязательно совпадают. Если транспьютер не успевает за один цикл обработать некоторые каналы, то он пришлет результаты на них на следующем цикле. Результат от транспьютера содержит просто набор вида `(Номер,Значение)`, и

все эти Значения заносятся в вектор значений каналов (динамическую БД), имеющийся в сервере.

Тем не менее, клиентской программе будет гарантированно послан пакет с результатами. Они просто будут взяты на такте 5 из вектора ("кэша"), имеющегося в сервере. Если трансьютер успел обработать этот запрос, то в векторе будет "свежее" значение, иначе – старое, и при этом клиенту будет послан флажок "значение несвежее".

2.2 Сервер

В настоящий момент для системы используется рабочее название "UCAM" (Unix-based CAMac control), и сервер, соответственно, именуется UCAM-сервер.

2.2.1 Компоненты сервера

Программа-сервер состоит из трех частей: привратник, менеджер, и собственно сервер (см. рис.2.1[1]).

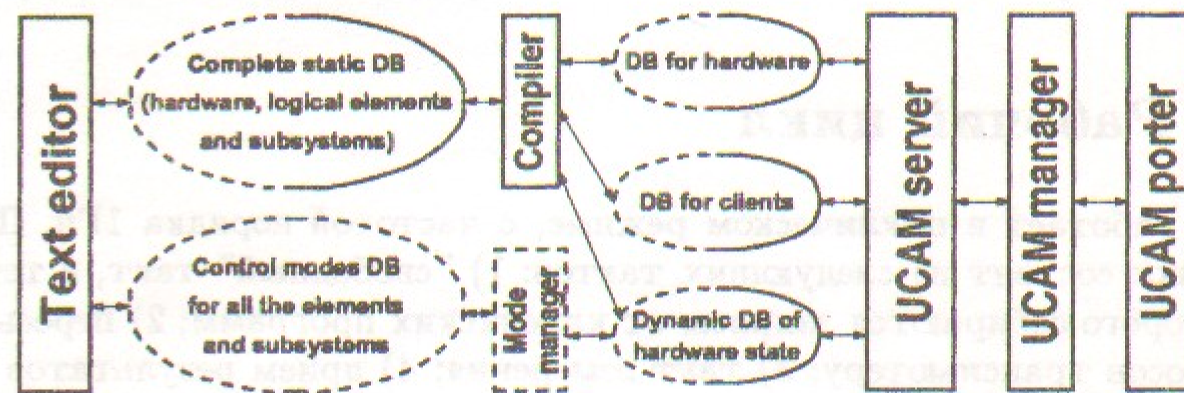


Рис. 2.1: Компоненты сервера.

Привратник принимает запросы на соединение от клиентских программ. Он проверяет, можно ли разрешить данное соединение, и если да, то "передает" его менеджеру или серверу. Если же нет, то клиентской программе отправляется ответ "в доступе отказано" и соединение закрывается.

Менеджер служит для администрирования системы. Это как бы программа-"надзиратель", которая управляет работой привратника и

[1]Уже созданные и работающие компоненты обведены на рисунке сплошным контуром, планируемые и разрабатываемые – пунктирным.

сервера. С точки зрения операционной системы, именно менеджер является демоном, запускающим остальные компоненты системы. В обязанности менеджера входит также чтение файла конфигурации и начальная настройка.

Сервер выполняет основную, "полезную" работу. С одной "стороны" он принимает от клиентских программ запросы на обслуживание и возвращает им результаты, а с другой "стороны" взаимодействует с трансьютером, передавая ему предварительно обработанные запросы и принимая результаты. Кроме того, через сервер производится доступ к базе данных.

2.2.2 Структуры данных, используемые в сервере

Каждый канал обладает определенными свойствами, некоторые из которых постоянны (например, номер крейта), а некоторые – переменны (к примеру, текущее значение канала). Информация о каналах организована в виде набора массивов типа `int`, `short` или `byte`, в зависимости от типа конкретного параметра. Такой подход несколько менее "структурен", чем использование массива структур, но позволяет компилятору использовать режим адресации `SIB` (`scale=4`), сокращая тем самым лишнее умножение на размер структуры.

Массивы проходят по всем блокам и крейтам, так что 0-й элемент является каналом 0 1-го блока в 0-м крейте. Если, например, в первом крейте 3 блока, в последнем из которых 5 каналов (от 0 до 4), то, если, например, 4-й канал 3-го блока 0-го крейта имеет номер 37, то 0-й канал 1-го блока 1-го крейта будет иметь номер 38.

Тем самым мы отказываемся от *фиксированного* `NP` в `#BANK` (как это было на ВЭПП-4), а `NP` формируется каждый раз при компиляции базы данных. В целях отладки имеется утилита, позволяющая выяснить этот номер.

В данный момент имеются следующие массивы (все имена начинаются с "`c_`", чтобы повысить читабельность и избежать совпадения имен):

- `int32 c_value[]` текущее значение канала (т.е. то, которое было в последний раз прочитано или записано);
- `int8 c_request[]` был прислан запрос на чтение или запись этого канала;

- `int32 c_time[]` время последней завершенной операции чтения или записи (фактически, это флаг "свежести": если

  `c_time[NP] == current_cycle`,

  то значение свежее);

- `int8 c_type[]` тип канала: чтение (0) или запись (1);
- `int32 c_crate[]` номер крейта;
- `int32 c_block[]` номер блока в крейте.

Следует отметить, что атрибуты `c_crate[]` и `c_block[]` избыточны, поскольку они могут быть вычислены из номера канала, но эта избыточность значительно уменьшает время маршрутизации.

2.2.3 Алгоритмы работы сервера

Когда приходит запрос на чтение/запись данных, то он вначале дублируется в области ответа для этого соединения, и для каждого канала в запросе выполняется следующее:

1. (а) Запрос на чтение: если уже был запрос на чтение для этого канала, или если это канал записи[2], то следующие шаги пропускаются.

   (б) Запрос на запись: проверяется флаг `c_request` для этого канала. Если он уже взведен, то пришедший ранее запрос отыскивается в очереди, значение в нем заменяется на новое[3] и все следующие шаги пропускаются.

[2]Здесь используется тот факт, что запрос на чтение канала записи *никогда* не пересылается в транспьютер, а ответ на него берется из внутреннего кэша сервера (`c_value[]`), где уже присутствует то значение, которое было записано последним, т.е. текущее. Если же запрос на чтение некоего канала приходит одновременно с запросом на запись в него, то, если за этот цикл транспьютер успеет выполнить запись, "читатель" получит обновленное, только что записанное значение. (Есть одно исключение, когда выполняется запрос на чтение канала записи – это опрос всего содержимого динамической базы данных перед записью состояния устройств (*режим*) для долговременного хранения в соответствующих файлах. Но для передачи такого запроса в транспьютер используется другой механизм.)

[3]Замену предыдущего значения можно, в принципе, пропустить, т.к. два запроса на запись в один канал в любом случае дают коллизию, так что не важно, который из них получит приоритет – последний (как при замене), или первый. (*Вопрос*: а что, если приходит (1) запрос на чтение, а затем (2) запрос на запись? *Ответ*: запрос на чтение *не* устанавливает флаг `c_request`, так что вопроса нет!)

2. Запрос добавляется к очереди для соответствующего крейта[4].
3. Устанавливается флаг `c_request`, дабы зафиксировать тот факт, что на этот канал уже пришел запрос на чтение/запись.

Замечание: если приходит запрос на запись в канал чтения, то он молча игнорируется. Конечно сервер *пошлет* клиенту число об этом канале, но это будет 0 (да, ноль – чтобы указать оператору, что с программой что-то не в порядке)[5].

Когда от транспьютера приходит команда "давай запросы", ему отсылается вся очередь для запрошенного крейта, после чего она очищается. Все запросы, пришедшие после этого момента, добавляются к теперь уже пустой очереди, и будут отработаны на следующем цикле[6].

В конце цикла транспьютер отсылает серверу пакет с результатами, и для каждого канала в этом пакете делается следующее:

1. Значение копируется в `c_value[]`.
2. Флаг `c_request` сбрасывается, чтобы отразить тот факт, что запрос был выполнен и значение свежее.

Поскольку транспьютер может не успеть обслужить в текущем цикле все запросы, то он пришлет результаты лишь для тех, что были выполнены, а для остальных флаг `c_request` в сервере останется взведенным.

Затем все области ответов сканируются и заполняются данными из `c_value`. Если соответствующий `c_request` все еще установлен (сигнализируя, что запрос еще не был обслужен), то значение возвращается с пометкой "старое".

Замечание: поскольку на те запросы, которые он не успел выполнить, транспьютер просто не присылает результатов, то флаг `c_request` останется взведенным, так что следующие запросы на этот же канал не будут

[4]Следует заметить, что в случае запроса на запись новое значение *не* записывается в `c_value[]` – оно попадет туда лишь когда запись действительно произойдет и транспьютер отошлет обратно серверу результат вместе с результатами запросов на чтение.

[5]Весьма вероятно, что следует включить в сервер (среди других отладочных средств) диагностику этой ситуации.

[6]Вообще-то, можно было бы отрабатывать в текущем цикле те из них, которые относятся к измеряемым в данном цикле каналам (т.е. тем, для которых установлен флаг `c_request`) или являются запросами на чтение каналов записи (которые, напомним, читаются из кэша сервера без участия транспьютера). Но это вызовет проблемы в том случае, если вместе с этим же пакетом пришли и запросы, не подпадающие под эти категории, и потому требующие обработки на следующем цикле.

снова и снова пересылаться в транспьютер, вызывая рано или поздно переполнение буферов.

2.3 Клиентская библиотека

2.3.1 Состав клиентской библиотеки

Клиентская библиотека предоставляет программам следующие услуги: установление/разрыв соединения с сервером, работа с логическими каналами, доступ к базе данных, доступ к средствам администрирования, асинхронная связь с дополнительными модулями в контроллере. Ниже эти интерфейсы разобраны более подробно.

Установление соединения с сервером

До выполнения каких-либо иных действий, программа должна соединиться с UCAM-сервером. Перед выходом, естественно, надо закрыть соединение. Это делается при помощи простейших вызовов:

```
int  ucam_connect (const char *host, const char *argv0);
int  ucam_close   (int cd);
```

Параметр host является именем машины, на которой запущен UCAM-сервер. ucam_connect() возвращает дескриптор соединения[7], или -1 если произошла ошибка (в этом случае в errno заносится ее код).

ucam_close() закрывает соединение, так что дескриптор более не связан ни с каким соединением и может быть использован заново. Этот вызов возвращает 0 при успешном выполнении операции, и -1 при ошибке[8].

Любая программа может открыть одновременно более одного соединения, например, к разным серверам. Однако, более предпочтительно использование одного соединения с каждым сервером, и чтение/запись всех данных через него, поскольку такой подход минимизирует накладные расходы[9].

[7]Это *не* файловый дескриптор, а индекс в таблице соединений, которая используется внутри клиентской библиотеки.

[8]Опыт показывает, что результат стандартной функции close() обычно игнорируется. Можно предположить, что это в основном верно и для ucam_close().

[9]Вообще-то, большинству программ, скорее всего, будет достаточно одного-единственного соединения. Но может возникнуть ситуация, когда на "развесистой" установке некоторые логически взаимосвязанные компоненты управляются разными серверами. Вот здесь-то и потребуется использование нескольких соединений одновременно.

Работа с логическими каналами

Поскольку сервер обрабатывает только один запрос за цикл, а клиентской программе может потребоваться читать и писать много каналов одновременно, то все запросы на чтение/запись собираются в программе в один пакет запросов и затем высылаются серверу. Когда сервер реально обработает этот пакет, он пошлет обратно пакет ответов, который, в, свою очередь содержит ответы на все запросы.

Для соответствия этой модели программа должна использовать следующий алгоритм:

1. Инициализировать новый пакет вызвав ucam_begin();
2. Добавить к этому пакету запросы на чтение/запись используя ucam_getXXX() и ucam_setXXX();
3. Отправить пакет, вызвав ucam_run().

Первым параметром всем этим функциям передается "int cd" – дескриптор соединения. Одновременно может использоваться несколько соединений, и вызовы с разными cd могут смешиваться. Сами функции описаны ниже.

Инициализация пакета. Каждый цикл в клиентской программе должен начинаться с вызова ucam_begin():

```
int  ucam_begin (int cd);
```

Эта функция создает новый пустой пакет, к которому могут быть добавлены запросы на чтение и запись, как описано ниже.

Запросы на чтение данных. Есть три функции для запроса чтения данных, имя каждой из них начинается с "ucam_get":

```
int ucam_getvalue (int cd, chanaddr_t addr,
                           int32      value,   tag_t *tag);

int ucam_getvgroup(int cd, chanaddr_t start,   int   count,
                           int32      values[], tag_t tags[]);

int ucam_getvset  (int cd, chanaddr_t addrs[], int   count,
                           int32      values[], tag_t tags[]);
```

ucam_getvalue() предназначена для запроса прочитать одиночный канал, и используется очень редко.

ucam_getvgroup() запрашивает последовательную группу каналов (например, когда десяток однотипных напряжений меряется одним АЦП).

ucam_getvset() запрашивает произвольный набор каналов, и может быть использована для опроса состояния логической "ручки".

Аргументы функций имеют следующий смысл:

cd дескриптор соединения;

addr номер канала для ucam_getvalue();

value/values переменная/массив, куда надо считать канал(ы);

tag/tags переменная/массив, куда следует поместить статус(ы) канала(ов) (значение свежее/старое);

start номер первого канала в группе;

addrs номера каналов в наборе;

count количество каналов в группе или наборе;

В качестве параметров tag и tags может быть указан NULL, в случае, если программе неважно, свежие ли значения или нет.

Параметры addr/start/addrs (выполняющие роль NP в #BANK на ВЭПП-4) считываются программой при старте из базы данных (подробнее см. далее раздел "Работа с базой данных").

Запросы на запись данных. Функции ucam_setXXX() очень похожи на набор ucam_getXXX():

```
int ucam_setvalue (int cd, chanaddr_t addr,
                           int32      value,            tag_t *tag,
                           int        *result);

int ucam_setvgroup(int cd, chanaddr_t start,     int    count,
                           int32      values[], tag_t  tags[],
                           int        results[]);

int ucam_setvset  (int cd, chanaddr_t addrs[],   int    count,
                           int32      values[], tag_t  tags[],
                           int        results[]);
```


Эти вызовы имеют дополнительный параметр result/results, который указывает на переменную/массив, в которые попадут новые значения (они могут отличаться от записанного значения в случае если две программы пытаются писать в один канал одновременно). Этот параметр может быть NULL, если программу не заботят возможные конфликты, или он может указывать на ту же переменную/массив, что используется в качестве value/values.

Отправка запроса. Наконец, когда пакет готов, он отправляется на сервер вызовом ucam_run():

```
int  ucam_run (int cd);
```

Эта функция дожидается ответа и складывает полученные данные в переменные, указанные при вызовах ucam_getXXX()/ucam_setXXX().

Подписка на данные. Поскольку большинство клиентских программ разработаны для "следящего контроля", они в основном читают данные, а операции записи сравнительно редки. Набор читаемых каналов также изменяется очень редко. Таким образом, каждый цикл клиент должен посылать одни и те же запросы, тратя таким образом время и пропускную способность. Чтобы избежать такого глупого поведения, клиент может "подписаться" на некоторые каналы.

Для подписки клиенту следует произвести следующие действия:

1. Инициализировать новый пакет, вызвав ucam_begin();
2. Добавить к этому пакету запросы на чтение, воспользовавшись ucam_getXXX()[10];
3. Отправить пакет, вызвав ucam_subscribe().

Функция ucam_subscribe() определена следующим образом:

```
int  ucam_subscribe (int cd);
```

Таким образом, процесс подписки очень схож с обычным чтением данных, с той лишь разницей, что вместо ucam_run() надо вызвать ucam_subscribe().

[10] Естественно, ucam_setXXX() использовать не следует!

Сервер будет слать пакет с запрошенными данными каждый цикл. Для того, чтобы подписаться на другой набор каналов, клиент должен заново послать запрос о подписке, и он автоматически очистит предыдущий набор. Для того, чтобы полностью "отписáться", следует послать пустой запрос.

Присылаемые по подписке данные будут попадать в переменные, указанные в качестве `values/tags` в функциях `ucam_getXXX()`.

Работа с дополнительными программными модулями

Для связи с программными модулями в транспьютере, которые не являются поканальными драйверами, используются так называемые "прямые соединения" (direct connections, direct channels). При этом сервер работает как "труба", просто передавая данные в обоих направлениях. В клиентской библиотеке выполняется лишь преобразование порядка байт в данных из локального в little-endian и обратно (подробнее на эту тему см. раздел 2.8.1).

В настоящий момент используется очень упрощенный вариант – предусмотрена лишь синхронная связь, т.е. на каждый запрос клиентской программе обязательно будет прислан ответ, и наоборот, данные от модуля могут присылаться только непосредственным ответом на запрос.

Открытие прямого соединения. Для открытия прямого соединения используется функция `ucam_opendirc()`:

```
int  ucam_opendirc (const char *host, const char *argv0);
```

Ее семантика идентична `ucam_connect()`. Закрывается прямое соединение при помощи `ucam_close()`.

Обмен данными по прямому соединению. Для пересылки данных применяется функция `ucam_dircmsg()`:

```
int  ucam_dircmsg(int cd,          moduleid_t module,
                  int32 args[],    int        nargs,
                  int32 data[],    int        ndata,
                  int32 results[], int        *nresults);
```

Здесь `module` – идентификатор модуля, полученный из базы данных; `args[]`, `nargs` – параметры, передаваемые модулю и их количество; `data[]`, `ndata` – данные и их количество; `results[]` – массив, куда следует поместить результат от модуля (например, осциллограмму), а в `*nresults` указывается размер `results[]`, после вызова туда помещается количество реально полученных данных. Разделение на `args` и `data` весьма условно, и сделано лишь для того, чтобы в будущем можно было расширить протокол для передачи данных размером не только 4 байта (тогда как параметры всегда являются 4-байтовыми целыми).

Работа с базой данных

Запросы к базе данных могут производиться как через обычные (поканальные) соединения, так и через прямые соединения. Механизм запроса очень похож на тот, что используется при запросах на чтение/запись каналов: сначала инициализируется пакет при помощи `ucam_dbbegin()`; затем он заполняется запросами при помощи `ucam_dbgetcrate()`, `ucam_dbgetblock()`, `ucam_dbgetchan()`, `ucam_dbgetgroup()` и `ucam_dbgetelem()`; и, наконец, все запросы отправляются на сервер вызовом `ucam_dbrun()`.

Каждая из функций `ucam_dbgetXXX()` имеет три параметра: `int cd` – дескриптор соединения, `char *name` – имя компонента базы данных, информация о котором запрашивается, и третий параметр вида `something *result`, где тип `something` зависит от запрашиваемых данных.

Следует заметить, что работа почти любой программы начинается именно с обращения к БД: она запрашивает NP, коэффициенты и еще все, что ей нужно из базы данных (ворота слежения, предельные уставки и т.д.).

2.3.2 Алгоритмы подсистемы связи с сервером

Установление соединения

Сразу после выполнения системного вызова `connect()` сокет переводится в асинхронный режим и дальнейший В/В осуществляется в фоновом режиме.

Библиотечную функцию установления соединения можно было бы сделать *всегда* блокирующейся, поскольку она будет вызываться нечасто, но из соображений полноты все же и ей можно передавать флаг "non-blocking".

Собственно В/В

Все библиотечные функции, осуществляющие В/В по сокетам, поддерживают два режима работы: синхронный (блокирующийся) и асинхронный (неблокирующийся).

В первом режиме возврат управления клиентской программе произойдет лишь после полного завершения операции (механизм блокирования/разблокирования клиентской программы рассмотрен ниже). Например, функция `ucam_run()` отправит пакет серверу, дождется ответа, и лишь затем вернет управление.

Во втором режиме при отсутствии "процессуальных ошибок" (таких, как неинициализированность пакета) функция отправит запрос и затем завершится, вернув результат -1 и `errno=UERUNNING` (аналог `EINPROGRESS`).

Реализация асинхронного режима

В асинхронном режиме после отправки запроса библиотечные функции сразу возвращают управление клиентской программе со статусом "в процессе исполнения".

При приходе ответа возникнет сигнал SIGIO, так что управление будет передано обработчику сигнала из библиотеки (назовем его SIGHANDLER). Он определит, по какому из сокетов (а следовательно, соединений) пришли данные, и считает то, что пришло. В начале любого пакета – заголовок, в котором содержится длина пакета. Если пришел еще не весь пакет (что весьма вероятно), то SIGHANDLER запишет в дескриптор соединения объем уже пришедших данных и отвалит. Когда придет остальное, то опять возникнет SIGIO, вызовется SIGHANDLER и т.д.

Когда пакет будет дочитан до конца, то SIGHANDLER проверит тип пакета и предпримет соответствующие действия. Реально может быть три класса пакетов: данные в ответ на запрос, подписка и исключительная ситуация. Последние два подробно обсуждаются в последующих разделах.

При приходе пакета с данными SIGHANDLER выполнит следующие два действия: 1) сделает пометку, что операция завершена; 2) вызовет указанную клиентской программой функцию. Таким образом, у клиентской программы есть два варианта работы с асинхронными соединениями. При первом варианте она после отправки запроса время от времени проверяет, не пришел ли ответ (при помощи библиотечной функции,

аналога `select()`), т.е. работает по опросу. При втором варианте при переводе соединения в асинхронный режим программа указывает функцию, которую следует вызвать по завершении операции, т.е. работает по прерыванию. В свою очередь, такая функция-hook может вызвать `XtNoticeSignal()` для уведомления Xt/Motif (при использовании оных). Оба варианта могут использоваться одновременно.

Реализация синхронного режима

Синхронный режим, вообще говоря, является фикцией. При его использовании библиотека работает точно так же, как и при асинхронном, но библиотечные функции просто сами дожидаются асинхронного завершения операции посредством простейшего цикла вида

```
while (!operation_completed(cd))
        select(0, NULL, NULL, NULL, &tv);
```

(tv – 1 секунда)

Поскольку для завершения операции должны придти данные, а они вызовут SIGIO, то `select()` будет прерван.

В синхронном режиме указанная программой для асинхронного уведомления функция не вызывается, а "аналог `select()`" всегда возвращает результат "операция завершена", т.к. операция всегда завершится до возврата управления клиентской программе.

Реализация подписки

Приход подписанных данных в силу своей природы всегда является асинхронным событием, поэтому режим работы соединения никак не влияет на подписку. При приходе таких данных будет производиться примерно то же, что и при приходе обычного ответа в асинхронном режиме. Во-первых, будет выставлен флажок "пришли обновленные" (точнее, просто значение некоей переменной будет увеличено на 1). Во-вторых, будет вызвана указанная программой функция (та же, что и при завершении операции, но с другим "кодом причины"). Как и при обычной асинхронной работе, личное дело программы, какой из двух вариантов уведомления использовать.

Замечание: при приходе подписанных данных функция-уведомитель, если она зарегистрирована, будет вызвана вне зависимости от того, находится ли соединение в асинхронном режиме.

Реализация исключительных ситуаций

Термином "исключительная ситуация" обозначается приход пакета, не являющегося ни ответом на некий стандартный запрос, ни подписанными данными.

Таких пакетов может быть три вида: закрытие соединения со стороны сервера, некое сообщение по консольному соединению (тип соединения `CT_CONS`), и пакет по прямому каналу связи (соединение `CT_DIRC`).

В любом случае будет вызвана указанная клиентской программой функция-уведомитель с соответствующим "кодом причины".

Если пакет – закрытие соединения, а программа не указала функции-уведомителя, то дальнейшие вызовы обычных библиотечных функций (кроме `ucam_close()`) будут возвращать результат "соединение закрыто сервером" (`errno=UECLOSED` – аналог `ECONNRESET`). Проверка "операция завершена" будет возвращать "да" – чтобы программа не зависала на ожидании, а пыталась сделать следующий запрос, когда ей и будет возвращен результат "соединение закрыто". Такой подход аналогичен используемому в UNIX по отношению к сокетам – при закрытии другого конца сокет-пары `select()` возвращает "готов для чтения", а `read()` считывает 0 байт.

В случае же прихода данных по консольному соединению или по прямому каналу клиентская программа *обязана* указать функцию-уведомитель. В противном случае данные просто будут потеряны. Если для консольного соединения это означает лишь потерю некоего информационного сообщения (типа "система запущена"), то для прямого канала, с высокой вероятностью, это абсолютно бессмысленно и является явной ошибкой (за исключением каналов, работающих в синхронном режиме – "запрос-ответ").

Замечание: как и в случае подписки, при исключительных ситуациях функция-уведомитель вызывается и для синхронных соединений.

Потенциальные источники ошибок

В принципе, библиотека может придти в непредсказуемое состояние, если переключение между синхронным и асинхронным режимами производится во время выполнения некоей операции. Поэтому возможность переключения режима ограничена моментами, когда не выполняется никакая операция (т.е. соединение находится в состоянии `CS_READY`).

С высокой степенью вероятности может возникнуть следующая ситуация: функция `ucam_subscribe()` сформировала и отослала серверу

новый набор подписываемых каналов, а в этот самый момент сервер присылает очередные значения подписанных каналов – естественно, из старого набора. Поскольку библиотека сохраняет информацию о том, куда складывать подписанное, в своем буфере, то в этот момент содержимое буфера не будет соответствовать присланным данным – т.е., библиотека уже "забудет", куда надо положить старую подписку. Чтобы обойти эту проблему, библиотека помнит поле `Seq` (порядковый номер) пакета заказа, которому соответствуют текущие данные в буфере. Если поле `Seq` присланных данных не соответствует запомненному, то такой пакет просто игнорируется. Учитывая, что `Seq` – `uint32`, т.е. имеет период 136 лет при частоте 1Гц, то вероятность совпадения порядковых номеров от разных подписок практически нулевая.

"Правила игры" для клиентской программы

Поскольку функции-уведомители вызываются из обработчика сигнала SIGIO, то они должны быть максимально короткими – в идеале просто выставлять некий флажок (или вызывать `XtNoticeSignal()`).

При нарушении этого условия может возникнуть ситуация, когда программа непрерывно занимается только обработкой SIGIO и ничем более, т.к. новые данные будут приходить быстрее, чем обрабатываться. Подробнее см. следующий раздел.

По некоему конкретному соединению пакеты могут приходить очень часто, поэтому при приходе пакета с данными, не являющегося ответом на запрос или подписанными данными (их библиотека сама скопирует куда надо), программа должна скопировать пришедшие данные сразу же – в функции-уведомителе. В противном случае, поскольку на каждое соединение имеется лишь один буфер приема, при приходе следующего пакета старые данные будут утеряны.

Замечание: в связи с этим для каждого соединения создается еще один буфер – для данных варьирующегося размера, приходящих в ответ на запрос. Примерами являются ответ на команду (EXECCMD) в соединениях типа `CT_CONS` и ответы из базы данных в соединениях `CT_DATA`. Т.е. ответы, содержащие данные, которые нельзя сразу куда-то скопировать, так как клиентская программа получает к ним доступ непосредственно в буфере (при помощи библиотечных функций).

Кроме всего прочего, программа не имеет права высылать следующий запрос непосредственно из уведомителя. Во-первых, соединение находится при этом в не вполне определенном состоянии (хотя это, в принципе,

преодолимо – надо только очень корректно выбирать момент для вызова уведомителя). Во-вторых, поскольку SIGIO приходит в произвольный момент, то и состояние самой программы может быть неподходящим – например, одна половина массива уже заполнена новыми данными, а другая – нет (опять же, можно и программу заставить использовать семафоры, или даже блокировать SIGIO на критических секциях, но вряд ли это оправданно). И в-третьих, SIGIO может придти в тот момент, когда программа формирует пакет для отсылки – на него попросту наложится тот, что пошлет уведомитель (возможное решение то же, что и во втором случае).

Установка и использование обработчика SIGIO и обоснование корректности этого

Обработчик SIGIO устанавливается один раз при открытии первого соединения.

В каждый конкретный момент в статической переменной типа `fd_set` содержится маска используемых файловых дескрипторов. В начале обработчика вызывается select() с нулевым таймаутом для выяснения, по каким из них пришли данные. Затем в цикле для каждого из готовых дескрипторов определяется, какому соединению он принадлежит (т.е. выясняется соответствие `fd` ⇔ `cd`), и производятся действия, описанные в разделе "Реализация асинхронного режима".

Поскольку возможна (и даже очень вероятна) ситуация, когда по одному соединению придет почти сразу два пакета, а SIGIO возникнет лишь один раз, то в конце обработчик еще раз вызывает `select()`, чтобы проверить, не пришли ли данные еще по какому-нибудь соединению, и если да, то вызывает `raise(SIGIO)`. Следует вызывать именно `raise()`, а не переходить на начало, т.к. в противном случае может возникнуть ситуация, когда управление *все время* находится внутри обработчика.

Наиболее любопытна реализация SIGIO в Solaris: сигнал возникает не тогда, когда приходят новые данные (т.е. объем несчитанных данных меняется с *unread* на *unread + delta*), а когда в *пустой* буфер приходят новые данные (т.е. объем данных в буфере меняется с 0 на *delta*). Таким образом, без проверки `select()`'ом из обработчика соединение просто "подвиснет" – даже если будут постоянно приходить новые данные, SIGIO больше ни разу не возникнет. Хотя при корректной реализации библиотеки такая ситуация никогда не должна возникнуть, ее надо иметь в виду на случай отладки.

В некоторых ОС сигнал вызывается лишь один раз, а затем сбрасывается в состояние `SIG_DFL`. В таких системах в самом конце обработчика будет заново вызываться `sigaction()`, чтобы восстановить требуемое поведение. В большинстве же ОС есть возможность указать `sigaction()`, что сбрасывать сигнал в начальное состояние не надо, но в разных ОС это делается по разному. Для достижения требуемого эффекта фрагмент кода, устанавливающий обработчик, содержит условные конструкции вида `#if defined(OS_xxx)`.

Замечание 1. Отправка данных всегда производится сразу

Поскольку в отличие от сигнала "получены новые данные" (SIGIO) сигнал "все данные наконец отправлены, можно слать новые" отсутствует[11], то *писать* данные в сокет нужно все сразу. Т.е. запись данных должна производиться функцией вида:

```
size_t writeall(int fd, const void *buf, size_t count)
{
  int     r;
  size_t  rest = count;

    while (rest) {
          r = write(fd, buf, rest);
          if (r < 0) {
             if (errno == EINTR) continue; else return -1;
          }
          rest            -= r;
          ((const char *) buf) += r;
    }
    return count;
}
```

Таким образом, запись данных является, вообще говоря, блокирующейся. К сожалению, этого нельзя избежать, т.к. единственный способ – пользоваться неблокирующимися сокетами (с опцией `O_NONBLOCK`) и время от времени при помощи `select()` проверять, нельзя ли отослать очередную порцию данных. Поскольку такой подход потребовал бы явной

[11] Вообще-то, в некоторых системах (например, FreeBSD) SIGIO присылается и после отправки всего буфера, но, поскольку такое поведение не является общепринятым стандартом, то на него нельзя полагаться.

кооперации со стороны клиентской программы, тем самым существенно усложняя ее, он неприемлем.

Замечание 2. Сокеты не должны быть неблокирующимися

Поскольку записываются данные все сразу, а считываются – по мере прихода, по прерыванию, то сокеты не должны быть неблокирующимися (т.е. с флагом `O_NONBLOCK`).

Замечание 3. При асинхронных ошибках соединения закрываются

Вообще говоря, соединения стоит помечать как закрытые практически при всех "непроцессуальных" ошибках, поскольку, например, невозможность выполнить `realloc()` ставит под вопрос дальнейшее использование соединения.

Если же ошибка происходит асинхронно (а в этом случае практически все ошибки "непроцессуальные"), то после нее соединение обычно находится просто в совершенно неопределенном состоянии, и поэтому в таких случаях сразу выполняется `MarkAsClosed(cp, errno)`.

Замечание 4. Почему надо использовать select(), а не sleep() или pause()

Казалось бы, при ожидании завершения операции вместо `select(...)` можно воспользоваться `sleep(1)` или даже `pause()`:

```
while (!operation_completed(cd)) pause();
```

Оказывается, нет!

Во-первых, вместо `sleep(1)` *нельзя* использовать `pause()`, т.к. сигнал может придти в тот момент, когда проверка уже завершилась (с результатом 0), а `pause()` еще не вызвана. При этом, если больше никаких сигналов не возникает, `pause()` зависнет навечно. `sleep()` же через секунду завершится, и проверка будет выполнена заново. Тем, что программа в этой ситуации зависнет на секунду, можно пренебречь, так как вероятность прихода сигнала на этом участке кода ничтожно мала.

Во-вторых, как выяснилось при написании данного фрагмента, `sleep()` использовать нельзя, поскольку он использует `alarm()`, и вследствие не вполне корректной реализации `sleep()` в libc иногда возникает ситуация, когда SIGALRM появляется при уже отсутствующем обработчике, так что программа аварийно завершается с диагностикой типа "`zsh: alarm <имя_программы>`". Для того, чтобы избежать таких неприятностей, надо вместо `sleep()` использовать `select()` с отсутствующими наборами дескрипторов (т.е. `NULL`) и таймаутом в 1 секунду.

2.4 Высокоуровневый интерфейс

Поскольку большинство прикладных программ будут иметь графический интерфейс, то имеется высокоуровневое расширение клиентской библиотеки, скрывающее детали реализации и облегчающее создание унифицированного интерфейса. Это расширение предоставляет следующие услуги:

- Очень простое создание и манипулирование окнами со всем положенным оформлением (меню, линейки инструментов, строки состояния, всплывающие подсказки (tooltips)).
- Высокоуровневый интерфейс к соединениям с сервером.
- Манипулирование высокоуровневыми структурами, такими, как элементы и группировки.
- Набор функций и widget'ов для визуализации данных – каналов, элементов, осциллограмм.

В клиентской библиотеке заложена возможность использования в одной программе нескольких соединений с сервером одновременно, в том числе и с несколькими разными серверами. Высокоуровневое расширение позволяет использовать эту возможность – каждое окно программы может использовать собственное соединение, а может и разделять его с другими окнами. В планах предусмотрен даже довольно экзотический вариант, когда одно окно работает с несколькими серверами (например, отображается логический элемент, считываемый с одного сервера, и осциллограмма с другого)[12].

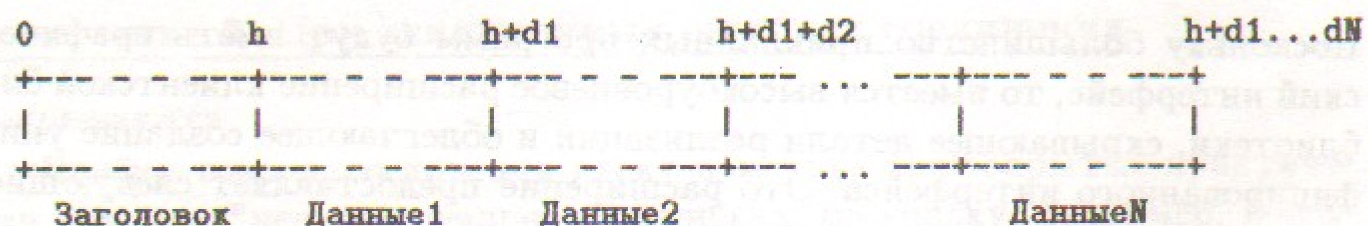
[12] Собственно, этот случай не столь уж и экзотичен: например, одно окно может работать с логическими элементами обслуживаемыми разными серверами – ситуация "развесистой" установки, для которой не хватает одного сервера.

2.5 Протокол связи клиентов и сервера

2.5.1 Формат пакетов

Пакеты посылаются как сервером, так и клиентами. Пакеты от клиента к серверу мы будем называть "*пакетами запросов*", а от сервера к клиенту – "*пакетами ответов*"[13].

Любой UCAM-пакет состоит из нескольких частей: заголовок и ноль или более разделов данных. Структура пакета показана на рис.2.2.



h – размер заголовка, d_n – размер n-го раздела данных (он содержится в самих данных, см. далее).

Рис. 2.2: Общая структура UCAM-пакета

Заголовок содержит: (а) идентификационную информацию, (б) информацию о типе пакета и данных. Структура заголовка показана на рис.2.3.

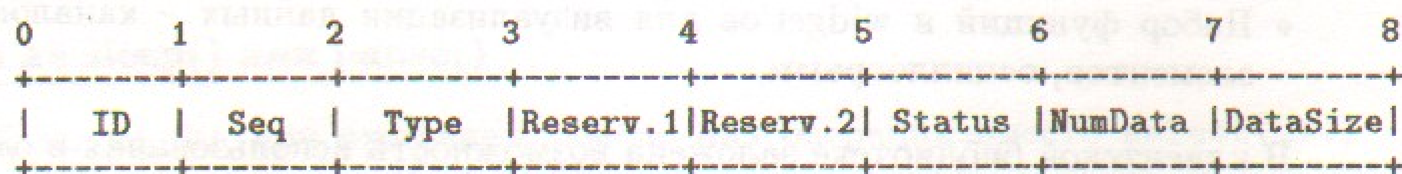


Рис. 2.3: Структура заголовка UCAM-пакета

Все поля – типа `unsigned int` (`uint32`). Они имеют следующий смысл:

ID идентификатор соединения, фактически пароль. Это число генерируется сервером при установлении соединения по формуле

$$ID = fd + RCC * Factor,$$

[13] Строго говоря, термин "ответ" не всегда применим, поскольку не все пакеты от сервера являются ответами на пакеты от клиентов. Но введение терминов типа "прямой пакет" и "обратный пакет" лишь напустило бы лишнего туману.

где

fd дескриптор сокета, полученный от `accept()`;

RCC (redundant check code) число, которому присваивается случайное значение при старте сервера, и затем

$$RCC = (RCC + 1) \% Factor$$

, т.е. это циклическое значение с большим периодом, что гарантирует, что ошибки сети и клиентов не вызовут проблем у сервера;

Factor фактор сдвига: $Factor = 2^{31}/OPEN_MAX$, где `OPEN_MAX` – максимальное количество соединений, возможных с сервером одновременно (т.е., максимальное значение `fd`)[14].

Поле ID требуется для следующего: например, клиент номер N прислал запрос, который не может быть непосредственно обработан получившей его частью сервера, и передается другой части с пометкой "сообщи-ка мне вот это, для клиента N". Затем из-за каких либо проблем этот клиент "отваливается". Далее, поступает запрос на соединение от другого клиента. Ему присваивается наименьший свободный номер сокета, и это может оказаться именно тот, который принадлежал "покойному", т.е. N. Затем поступает ответ от другой части сервера: "ты тут спрашивал вот это для N". По логике, этот ответ следует отправить в сокет N. Но он-то уже принадлежит другому, который ничего такого не ждет! При несовпадении же присланного с ответом от собрата ID с текущим ID для сокета N сервер определит, что ответ уже никому не нужен. Кроме того, поле ID может использоваться для повышения степени защищенности системы.

Seq порядковый номер пакета. Присваивается клиентской библиотекой. Используется, в основном, для отладки. Кроме того, требуется для асинхронной связи (например, если в одной программе используется несколько соединений, то они все могут разделять один счетчик).

Type тип пакета. Типы обозначаются константами семейства `UCAMT_xxx`.

Reserv.1, Reserv.2 зарезервированы для будущего использования.

[14] Константа `OPEN_MAX` обычно определяется директивой `#define` в файле `limits.h`.

Status результат операции, используется только в ответных пакетах (в пакетах запросов должно быть 0). Конкретное содержимое поля Status зависит от типа пакета. Обычно является комбинацией флагов, определяемых константами семейства `UCAMS_xxx`.

NumData количество разделов данных.

DataSize суммарный размер всех разделов данных. Может быть использовано для дополнительных проверок:

`DataSize = РазмерПакета − РазмерЗаголовка`

2.5.2 Доступ к физическим каналам

Каждый пакет чтения/записи данных содержит один или более раздел данных. Он, в свою очередь, состоит из (а) управляющей информации (заголовка) и (б) собственно данных.

Формат заголовка показан на рис.2.4.

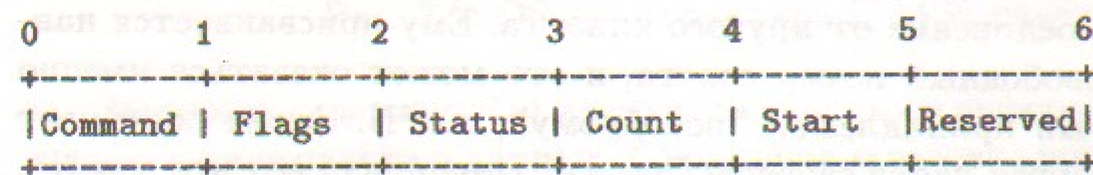


Рис. 2.4: Формат заголовка раздела данных UCAM-пакета

Все поля – типа `unsigned int` (`uint32`). Они имеют следующий смысл:

Command код команды (см. ниже).

Flags флаги (модификаторы) команды. Используется только в пакетах запросов (и оставляется неизменным в пакетах ответов).

Status результат операции. Используется только в пакетах ответов (должно быть 0 в пакетах запросов).

Count количество каналов в запросе.

Start номер первого канала в группе (для команд `{get|set}vgroup`, иначе должно быть 0).

Reserved зарезервировано для будущего использования.

Содержимое данных зависит от кода команды, и в некоторых случаях данные могут вообще отсутствовать. Существуют 4 команды:

GETVGROUP считать последовательную группу каналов (`ucam_setvgroup()`)

GETVSET считать произвольный набор каналов (`ucam_setvset()`)

SETVGROUP записать в последовательную группу каналов (`ucam_getvgroup()`)

SETVSET записать в произвольный набор каналов (`ucam_getvset()`)

Вызовы `ucam_getvalue()` и `ucam_setvalue()` реализованы как вызовы `ucam_{get|set}vgroup()` с одним каналом в группе.

Форматы данных в пакетах запросов и пакетах ответов очень похожи и более подробно рассматриваются ниже.

<u>Данные в пакетах запросов</u>

Форматы данных для пакетов запросов представлены на рис.2.5 и 2.6.

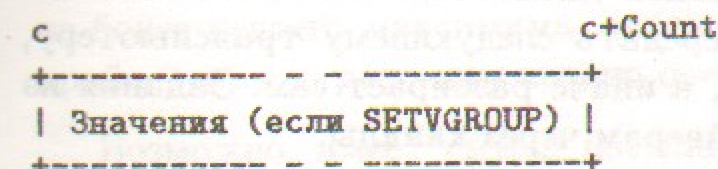


Рис. 2.5: Формат данных в пакете запросов для команд GETVGROUP и SETVGROUP

Следует отметить, что при команде SETVSET *всe* значения расположены после *всех* номеров, вместо набора пар `<Номер,Значение>`. Это делает форматы для GETVSET и SETVSET схожими и упрощает код.

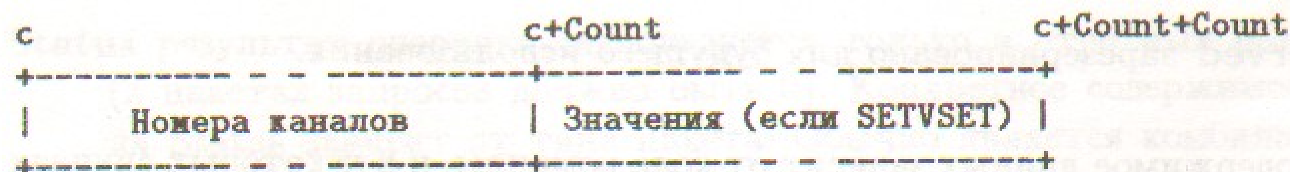


Рис. 2.6: Формат данных в пакете запросов для команд GETVSET и SETVSET

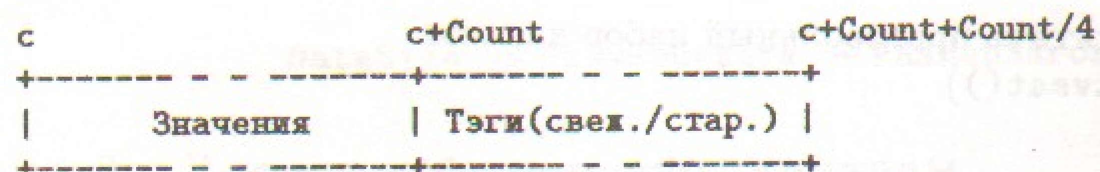


Рис. 2.7: Формат данных в пакете ответов

Данные в пакетах ответов

Формат пакета ответов одинаков для всех команд и представлен на рис.2.7.

Следует напомнить, что значения каналов присылаются в ответ как на команду чтения, так и на команду записи.

2.6 ПО в транспьютере

Все общение с сервером ведет основной диспетчер. Он принимает пакет, проверяет, не нужно ли его передать следующему транспьютеру, и если да, то переправляет дальше, а иначе разбирает сам. Задания из пришедшего пакета он раздает драйверам через каналы.

Драйверы блоков центрального крейта непосредственно связаны с основным диспетчером.

Драйверы блоков периферийных крейтов управляются диспетчерами периферийных крейтов (ДПК). На каждый DS-24 выделяется один ДПК. Он выглядит для основного диспетчера как обычный драйвер, и ему переправляются пакеты для всех подчиненных ему драйверов. Он, в свою очередь, разбирает эти пакеты и через каналы раздает их подчиненным.

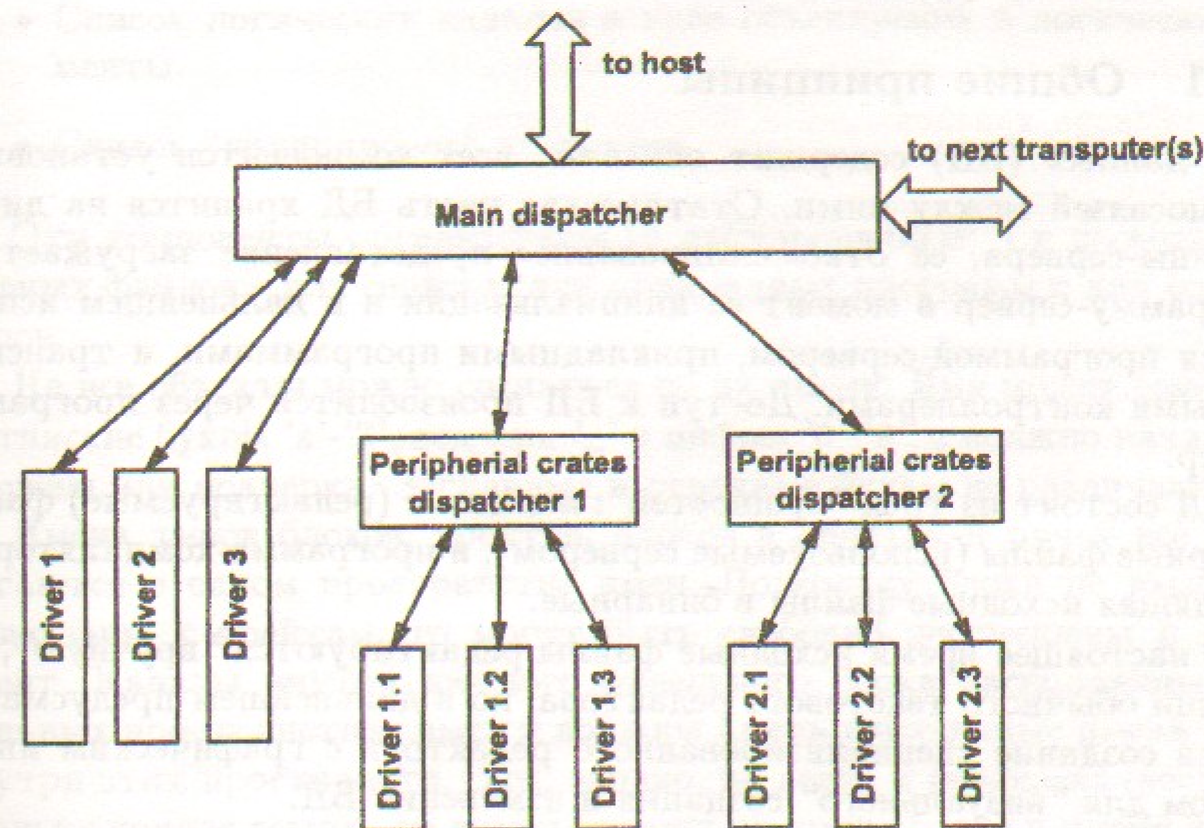


Рис. 2.8: Структура ПО в транспьютере

ДПК играют две роли: во-первых, являются собственно диспетчерами, а во-вторых, исполняют NAF'ы в периферийных крейтах[15].

Когда подчиненному драйверу требуется выполнить какой-либо NAF, то он посылает запрос об этом к ДПК. Тот является единственным процессом, имеющим доступ к DS-24. Тем самым предотвращаются конфликты и обеспечивается гарантированный доступ каждого подчиненного драйвера к DS (правда, через не вполне определенное время, но не более некоего максимума, определяемого "болтливостью" остальных драйверов, т.к. рано или поздно очередь дойдет до каждого).

Возможно, имеет смысл посылать к ДПК запрос на выполнение не одиночного NAF'а, а некоей серии, с возможностью указывать между элементами серии (или блока при последовательном NAF'е) задержки.

[15] Поскольку исполнение NAF в периферийном крейте реализуется путем выполнения последовательности NAF'ов в блоке DS-24, расположенном в центральном крейте, а воизбежание конфликтов с каждым блоком должен работать лишь один процесс – на эту роль наилучшим образом подходит именно ДПК.

2.7 База данных

2.7.1 Общие принципы

База данных (БД) содержит описание всех компонентов установки и взаимосвязей между ними. Статическая часть БД хранится на дисках машины-сервера, ее откомпилированное представление загружается в программу-сервер в момент ее инициализации и в дальнейшем используется программой-сервером, прикладными программами, и транспьютерными контроллерами. Доступ к БД производится через программу-сервер.

БД состоит из трех "сущностей": исходные (редактируемые) файлы, бинарные файлы (используемые сервером), и программа-компилятор, переводящая исходные файлы в бинарные.

В настоящее время исходные файлы редактируются "вручную", при помощи обычного текстового редактора, но в дальнейшем предусматривается создание специализированного редактора с графическим интерфейсом для "визуального" создания и изменения БД.

Исходное описание установки (иногда именуется "файл описания установки") состоит из нескольких текстовых файлов, каждый из которых описывает некий слой: крейты, блоки и т.д. Все файлы следуют стандартному стилю UNIX:

- строки, начинающиеся с '#', считаются комментариями и игнорируются, также как и пустые строки;
- '\' в конце строки означает, что эта строка продолжается на следующей;
- любой набор пробелов и символов табуляции рассматривается как один символ пробела.

2.7.2 Адресация объектов

БД содержит следующие компоненты (в порядке снизу-вверх по иерархии):

- Типы блоков.
- Список имеющихся блоков и физических каналов в них.
- Список имеющихся крейтов и распределение блоков в них.

- Список логических каналов в виде объединения в логические элементы.
- Список "группировок" элементов.

Эти компоненты организованы (с дублированием[16]) в несколько бинарных файлов – для сервера, для прикладных программ и для контроллеров.

На все объекты можно сослаться по их имени. Имя может содержать латинские буквы 'A'-'Z', подчерк '_' и цифры '0'-'9', и должно начинаться с буквы или подчерка. Заглавные и строчные буквы не различаются.

Имена типов блоков, крейтов, блоков в крейтах и ручек все располагаются в одном пространстве имен. Поскольку блоки по именам не привязаны к крейтам, то могут быть свободно перемещены в другой крейт. Каналы внутри каждого отдельного блока расположены в локальных пространствах имен и должны иметь уникальные имена только внутри этих пространств. Это, однако, не ведет к неоднозначности, поскольку полная ссылка на канал состоит из имени блока и имени канала, например "`block1.channelB`".

Как легко заметить, такой подход очень похож на используемый в языках программирования: там все имена типов, переменных и иных объектов располагаются в одном пространстве имен, а поля в структурах – в отдельных. Ссылка на поле внутри структуры выглядит как `ИмяСтруктуры.ИмяПоля`.

2.7.3 Форматы файлов, составляющих исходное описание и последовательность компиляции БД

Используемые файлы:

`types.uds` описания различных типов блоков;

`blocks.uds` список используемых блоков и физ. каналов в них;

`crates.uds` список крейтов и расстановка в них блоков;

`elements.uds` описания используемых логических элементов.

[16] Дублирование информации осуществляется в момент трансляции файлов, следующей автоматически после завершения какой-либо редакции исходных данных. Поскольку последняя осуществляется "вручную" (опаратором), то время трансляции является лишь накладными расходами к процессу редактирования данных и в задачу практически не входит.

Примечание: ".uds" – Ucam Database Source. Вообще говоря, можно было выбрать любое неиспользуемое расширение (хоть ".abc123").

Последовательность компиляции:

1. Считывается файл `types.uds`. Создается пространство имен типов блоков.

2. Считывается `blocks.uds` с использованием метрик блоков, полученных на предыдущем шаге. Создается таблица используемых блоков, пространство имен блоков и локальные пространства имен каналов в блоках.

3. Считывается `crates.uds`, производится проверка расстановки блоков. Создается пространство имен крейтов, которое реально нужно лишь для отладки.

4. Генерируется собственно база данных (не считая логических каналов). (1) На этом этапе блоки упорядочиваются по их расположению в крейтах и присваиваются номера физических каналов. (2) Создается набор данных для драйверов (всевозможные таблицы уставок, пределов и т.д.) и (3) таблицы запуска драйверов (для диспетчеров). (4) Кроме того, изготавливается "карта" установки и информация о маршрутизации.

5. Считывается `elements.uds` и создается таблица логических элементов (непосредственно в том формате, который будет отсылаться клиентам плюс каталог для сервера (индекс для ускорения поиска)).

2.7.4 Редактирование

Поскольку редактирование базы данных – процесс очень ответственный, а ошибку допустить легко, то редактирование не сводится просто к запуску текстового редактора. Для поддержания целостности базы данных вводится специальная процедура (очень похожая на процесс редактирования информации о пользователях в /etc/passwd при помощи программы vipw).

Суть заключается в том, что редактор запускается не непосредственно пользователем, а специальной программой, которой указывается, какая часть БД должна подвергнуться редактированию. Эта программа копирует соответствующий файл во временную директорию и запускает для него текстовый редактор. После завершения редактора запускается

компилятор (причем его вывод также идет во временную директорию), и производится проверка корректности изменений. Если ошибок не обнаружено, то обновленными файлами заменяются старые версии (и исходные, и бинарные), а иначе повторно вызывается редактор. Предусмотрена также возможность одновременного редактирования различных компонентов БД разными операторами.

2.8 Некоторые общие принципы

2.8.1 Форматы данных межплатформно-переносимы

При разработке протоколов передачи данных надо учитывать следующие системные особенности организации данных:

1. У разных процессоров разный порядок байт в слове при представлении целых чисел: x86, Alpha, T805 – little-endian (LSB first); MIPS, SPARC – big endian (MSB first).

2. У разных процессоров разные требования на выравнивание данных в памяти: x86 позволяет доступ к слову по произвольному адресу, а MIPS и SPARC требуют выравнивания по границе слова.

3. Не существует хоть сколько-нибудь стандартного способа представления вещественных чисел, который был бы совместим со всеми процессорами.

В системе UCAM используются следующие решения:

1. Во всех протоколах целые числа передаются в формате little-endian, а в транспортной библиотеке они преобразуются в "родной" для данной машины формат. Поскольку в основном используются машины на базе x86, то обычно преобразование не требуется вовсе (именно по этой причине в качестве базового выбран именно little-endian, а не являющийся сетевым стандартом big-endian).

   Так как и x86 (на котором работает сервер), и T805 являются little-endian, то на ветви сервер↔транспьютер преобразование в протоколе даже не предусматривается.

2. Все целочисленные данные всегда выравниваются по своему размеру: int всегда начинаются со смещения, кратного 4, short – со

смещения, кратного 2 (впрочем, в целях унификации реально вместо short всегда используется int – потери от чуть большего объема данных минимальны, а реализация существенно проще и надежнее).

Кроме того, все структуры всегда дополняются в конце нулевыми байтами до размера, кратного 4 – это позволяет безболезненно пересылать такие структуры "друг за дружкой" в одном пакете.

3. Вещественные числа передаются в виде строк, а в клиентской библиотеке преобразуются в тип double (прозрачно для прикладной программы). Например, число 3.1415927 будет передаваться в виде последовательности байт "3.1415927\0".

   Единственным иным способом (причем ограниченно приемлемым только в нашем случае, когда реально все вещественные числа получаются из больших целых из CAMAC) было бы передавать два целых числа – числитель и знаменатель (в данном случае – 31415927 и 10000000). Но при этом, во-первых, оставалась бы проблема потери точности, и, во-вторых, пришлось бы изобретать алгоритм выбора этих чисел из вещественного.

4. Поскольку вещественные числа передаются строками, а есть еще и реально текстовые данные, да и число элементов в структуре (например, каналов в элементе) не фиксировано, то передаваемые структуры получаются переменного размера. Чтобы минимизировать возникающие неудобства, используются следующие принципы:

   - В начале каждой такой структуры передается ее размер (включая само поле размера).
   - Все поля фиксированного размера располагаются в начале структуры, а строки переменной длины – за ними. Если структура содержит в себе также массив структур переменного размера (как, например, в описании элемента передаются описания логических каналов), то этот массив располагается после строк, причем его смещение выравнивается на 4.

2.8.2 Никогда не ждать ответа на запрос

Во всех частях сервера и в клиентских программах используется общий принцип: если по протоколу нужно послать запрос, на который потом придет ответ, то никогда не следует это делать следующим образом:


```
write(s, &request, sizeof(request));
read(s, &reply, sizeof(reply));
```

Дело в том, что `read()` зависнет до тех пор, пока не придет ответ. А это может произойти весьма нескоро. Более того, может возникнуть блокировка: процесс А послал запрос к Б и ждет ответа, а Б в этот же момент послал запрос к А и тоже ждет.

Чтобы избежать потерь времени и возникновения тупиков, ***никогда не следует ждать ответа***.

Если возникает необходимость послать запрос, требующий ответа, то следует отправить запрос и сделать пометку, что он послан. Когда же придет ответный пакет, то операция `select()` вернет результат "по этому сокету пришли данные". После чего можно выполнить `read()` и снять пометку ожидания.

Таким образом, основной цикл программы будет состоять из вызова `select()` с последующей проверкой, по каким сокетам пришли данные, чтением и соответствующей обработкой этих данных.

Пример: когда к менеджеру приходит запрос "дай список всех клиентов", то ему надо запросить эту информацию у сервера: отправить запрос, получить результат, и передать его клиенту. Вместо того, чтобы послать вопрос и зависнуть на `read()`, менеджер просто отправляет пакет "скажи-ка мне вот это, для такого-то", и возвращается к своей обычной работе (т.е. зависает на `select()` в ожидании любых пакетов). Когда сервер обрабатывает запрос и возвращает ответ, то сокет `server` в менеджере становится "готовым", менеджер считывает пакет, обнаруживает, что это запрошенные клиентом данные, и отсылает их заказчику.

2.8.3 Буфера растут по мере надобности

Как в сервере, так и в клиентской библиотеке используется множество буферов. При этом возникает небольшая проблема: зачастую заранее неизвестно, какого размера и сколько буферов потребуется.

Вопрос количества решается очень просто – буфера делаются не статическими переменными-массивами, а отводятся по мере надобности в динамической памяти функцией `malloc()`. Когда буфер становится не нужен, то отведенная под него память освобождается функцией `free()`.

Проблема размера буферов на первый взгляд кажется более сложной. В самом деле, если при создании буфера просто отводить под него некий фиксированный объем памяти, то придется ввести ограничение на максимальный размер данных, которые могут быть размещены в этом буфере.

Что еще хуже, чаще всего реально будет использоваться лишь очень небольшой объем в начале буфера, а большая часть памяти, отведенной под него, станет тратиться впустую.

Реальне решение состоит в следующем: при создании буфера под него выделяется некоторый минимальный объем памяти, а затем по мере надобности размер увеличивается с помощью функции `realloc()`.

Начальный объем выбирается в зависимости от предназначения буфера. Это может быть, например, 128 байт, хотя можно ограничиться даже 16 (0 нельзя использовать лишь из-за того, что `malloc()` не отводит блоков памяти такого размера)[17].

Поскольку чаще всего за первые несколько циклов использования буфера размер данных для него успеет достигнуть своего максимального значения, то выполнять `realloc()` придется лишь несколько раз в самом начале.

Следует отметить, что размер буфера надо *только увеличивать*, уменьшать же его никогда не нужно[18].

Динамическое увеличение размера может производиться по такому алгоритму: при приходе новой порции данных, которые должны быть размещены в буфере, нужно проверять, достаточен ли текущий объем, и если нет, то увеличивать его. Например, можно просто завести следующую функцию, которую вызывать перед помещением в буфер очередной порции:

```
/*
 *  Аргументы:
 *      *bufptr   указатель на буфер
 *      *sizeptr  текущий размер буфера
 *      newsize   требуемый объем
 */
int GrowBuf(void **bufptr, size_t *sizeptr, size_t newsize)
{
 void *newbuf;

  if (*sizeptr >= newsize) return 0;
```

[17] В принципе можно при создании блока нулевого размера обходиться без `malloc()` – просто присвоив указателю `NULL`. Последующий вызов `realloc()` сработает корректно – по стандарту ANSI C `realloc(NULL, size)` эквивалентно `malloc(size)` (в SunOS 4.1 этот стандарт не соблюдается).

[18] Если, к примеру, размер буфера 16K, а реально используется лишь 4K, то операционная система сама при необходимости переместит неиспользуемые три страницы в своп, так что они не будут занимать реальную память.


```
  newbuf = realloc(*bufptr, newsize);
  if (newbuf == NULL)  return -1;
  *bufptr  = newbuf;
  *sizeptr = newsize;

  return 0;
}
```

Несмотря на то, что в `GrowBuf()` предусмотрена проверка на нехватку памяти, реально такой ситуации никогда не возникнет – для этого нужно, чтобы переполнились и реальная память, и своп. В такой ситуации программа в любом случае станет неработоспособна.

Замечание: этот принцип нельзя использовать, если `GrowBuf()` придется вызывать из обработчика сигнала. Дело в том, что `malloc()` и `realloc()` нереентерабельны, а сигнал может возникнуть именно тогда, когда выполняются они сами. В этом случае придется все же вводить ограничение на размер буфера, и отводить память сразу по максимуму – тогда `GrowBuf()` просто сразу вернется (именно так делается с буферами для приема пакетов от сервера в клиентской библиотеке).

Глава 3

Текущее состояние

В настоящий момент первоначальная версия системы используется для управления ВЧ системой накопителя-охладителя и линейным ускорителем комплекса ВЭПП-5.

В качестве операционной системы как в серверах, так и в рабочих станциях используется Linux – свободно распространяемая система семейства UNIX. Кроме того, клиентские программы могут работать под BSDI/386, IRIX, SunOS, Solaris, FreeBSD, Unixware, OSF1 (поддержка любой POSIX-совместимой системы добавляется тривиально).

Проводились тесты производительности системы и требуемых ресурсов (486DX2/66 под Linux 1.3, опрос с частотой 1Гц), давшие следующие результаты:

- Сервер занимает около 800 Кб памяти.
- На каждую программу управления дополнительно требуется от 0.5 до 50 Кб памяти сервера.
- При 20 запущенных программах управления сервером занимается 10–15% производительности процессора.

В настоящее время 486DX2/66 используются в качестве серверов и рабочих станций в "одном лице", и их производительности хватает с запасом. Очевидно, что P5/133/166 вполне справятся с любыми задачами управления, которые могут возникнуть.

На рис.3.1,3.2 приведены два примера уже существующих программ управления.

Рис. 3.1: Пример простейшей программы управления.

В дальнейшем планируется объединить все ЭВМ комплекса ВЭПП-5 в единую локальную сеть (см. рис.3.3), "оторванную" от общеинститутской сети, чтобы полностью исключить влияние последней на работу комплекса.

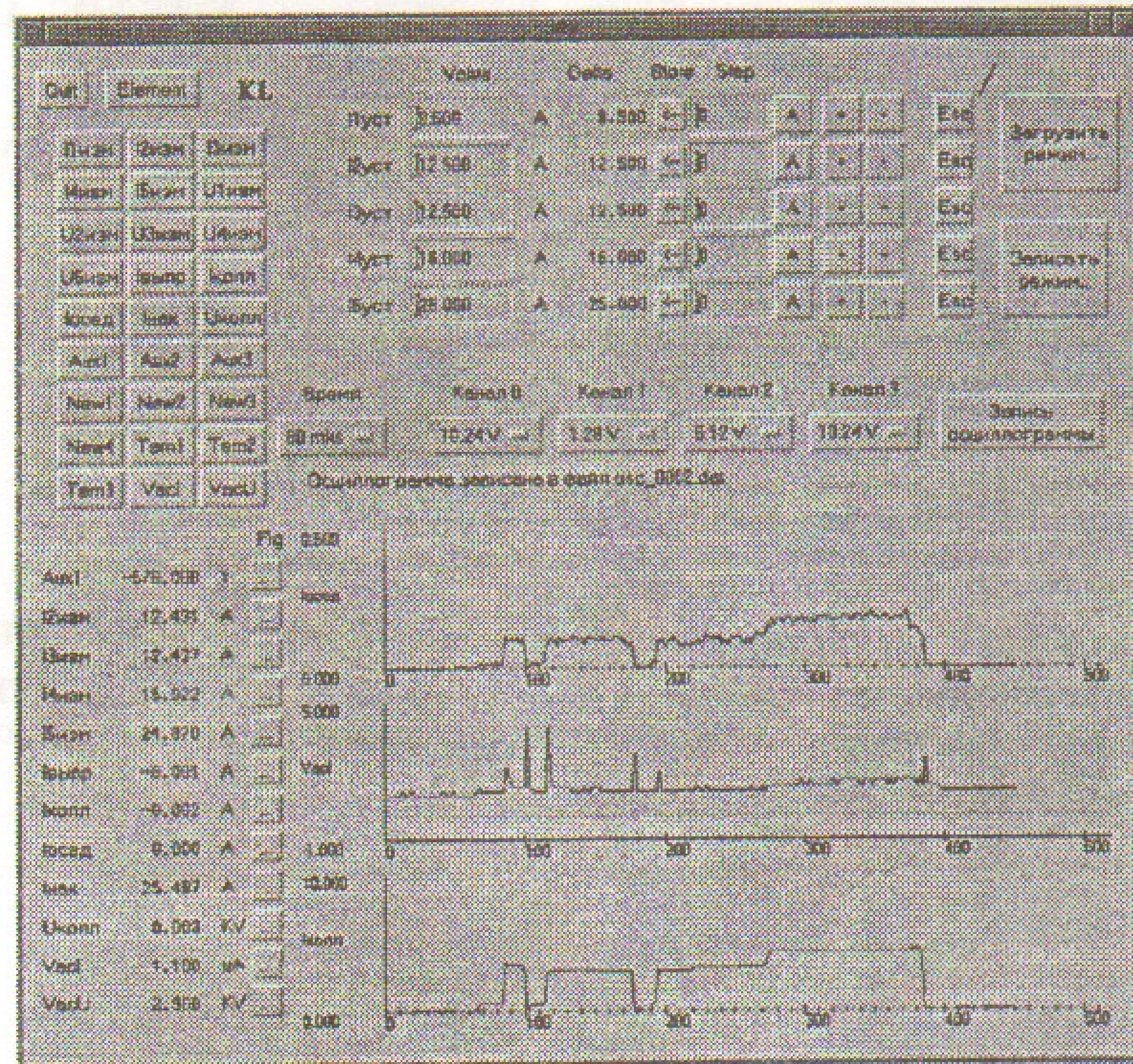


Рис. 3.2: Пример программы управления "логическим элементом".

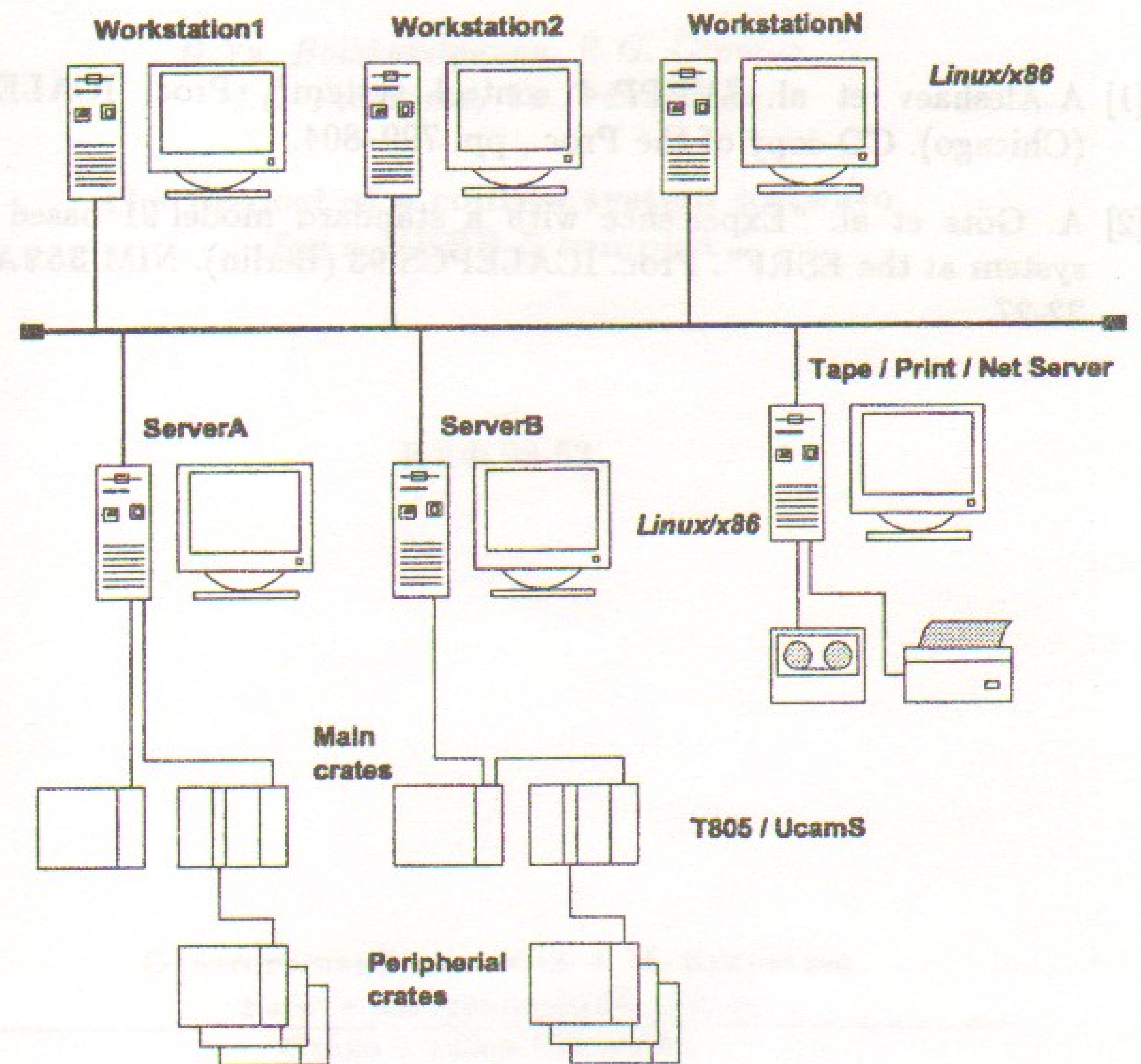


Рис. 3.3: Организация сети системы управления.

Библиография

[1] A.Aleshaev et al. "VEPP-4 control system", Proc. ICALEPCS'95 (Chicago). CD-copy of the Proc., pp. 799–804.

[2] A. Götz et al. "Experience with a standard model'91 based control system at the ESRF". Proc. ICALEPCS'93 (Berlin). NIM **352A** (1994) 22-27.

Д.Ю. Болховитянов, Р.Г. Громов,

И.Л. Пивоваров, Ю.И. Эйдельман

Проект программного обеспечения системы управления комплексом ВЭПП-5

D.Yu. Bolkhovityanov, R.G. Gromov,
I.L. Pivovarov, Yu.I. Eidelman

The project of a control system software for a VEPP-5 complex

ИЯФ 98-53

Ответственный за выпуск А.М. Кудрявцев
Работа поступила 30.06.1998 г.

Сдано в набор 7.07.1998 г.
Подписано в печать 7.07.1998 г.
Формат бумаги 60×90 1/16 Объем 2.0 печ.л., 1.6 уч.-изд.л.
Тираж 90 экз. Бесплатно. Заказ N° 53

Обработано на IBM PC и отпечатано на
ротапринте ИЯФ им. Г.И. Будкера СО РАН
Новосибирск, 630090, пр. академика Лаврентьева, 11.