АКАДЕМИЯ НАУК СССР ВЫЧИСЛИТЕЛЬНЫЙ ЦЕНТР

Ю.И. БРОДСКИЙ, В.Ю. ЛЕБЕДЕВ

ИНСТРУМЕНТАЛЬНАЯ СИСТЕМА ИМИТАЦИИ MISS

ВЫЧИСЛИТЕЛЬНЫЙ ЦЕНТР АН СССР

МОСКВА - 1991

УДК 519.876.5

Ответственный редактор

Доктор физ.-матем. наук Ю.Н.Павловский

Книга посвящена многоязыковой инструментальной системе, предназначенной, в основном, для программирования на IBM-совместимых ПЭВМ сложных имитационных моделей- Подразумевается имитация в классическом понимании, т. е. - численное воспроизведение динамики целенаправленного функционирования совокупностей реальных объектов. Изложены концепция моделирования, поддерживаемая системой, ее собственный язык описания логики и данных моделей, а также все предоставляемые ею инструментальные средства, в том числе графический и текстовый редакторы, СУБД и пользовательский интерфейс. Инструментарий системы в большей своей части вполне универсален и может быть эффективно использован в приложениях, не связанных с имитацией, но состоящих в разработке сложных многокомпонентных программ с диалогом, анимализирующей графикой и собственными базами данных.

Рецензенты: Ю.А.Флеров,

В.В.Федоров

Научное издание

Вычислительный центр
Академии наук СССР, 1991

Св. план 1991, поз. 161

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Настоящая книга содержит описание многоязыковой системы программирования для IBM-совместимых персональных ЭВМ, ориентированной, прежде всего, на имитацию, но пригодной и в качестве универсального инструмента, облегчающего создание сложных (в частности, диалоговых) многокомпонентных программных комплексов с собственными базами данных. Полезность системы в приложениях, не связанных с имитационным моделированием, определяется развитостью поддерживаемой ею технологии разработки хорошо структурированных программ и богатством предлагаемых инструментальных средств общего назначения, включая графические. Ее нацеленность на имитацию проявляется в том, что эта технология формализуется в терминах некой концепции моделирования и обеспечивает возможности гибкого управления модельным временем. Используемое в дальнейшем имя системы - *MISS* - составлено первыми буквами ее англоязычного определения: *Multi-lingual Integrated* System for *Simulation*. Книга может служить руководством пользователю.

Термин *имитация* мы употребляем в классическом понимании, бытовавшем в начале семидесятых годов, когда он прочно закрепился на страницах отечественной специальной литературы. Впоследствии содержание термина интенсивно размывалось, и сегодня многие готовы назвать имитационным экспериментом едва не любую

последовательность вычислений, выполненных на ЭВМ. Именно потому нелишне охарактеризовать деятельность, на которую ориентирована *MISS,* поточнее: это - такое моделирование, при котором важны *ЯВНОЕ ВОСПРОИЗВЕДЕНИЕ ХОДА ВРЕМЕНИ И ОТРАЖЕНИЕ БИХЕВИОРАЛЬНЫХ МОМЕНТОВ, Т.Е. ВОСПРОИЗВОДИТСЯ МОТИВИРОВАННОЕ ПОВЕДЕНИЕ.*

Предлагаемый инструмент - отнюдь не первое средство, разработанное для применений в указанной области. Хорошо известен широкий класс алгоритмических языков имитационного моделирования, имеющих в точности то же назначение: достаточно упомянуть языки *СИМУЛА-67, СИМСКРИПТ* и *GSL.* Уже из названия первого, который является самым "молодым" в тройке, видно сколь не нова идея создания специального математического обеспечения для нужд имитации. Тем не менее, благодаря ряду индивидуальных особенностей, *MISS* не выглядит лишним довеском к существующему арсеналу. Эти особенности есть и в поддерживаемой *MISS* концепции моделирования и, что более важно, - во многих воплощенных в данной системе технологических решениях.

Согласно общепринятой классификации, *MISS* относится к ПРОЦЕССНО-ОРИЕНТИРОВАННЫМ системам и в ее модельной концепции нетрудно усмотреть элементы сходства с концепцией *СИМУЛЫ-67* и с так называемой SDL-технологией проектирования и имитации систем связи, развиваемой за рубежом уже второй десяток лет. Коротко говоря, предмет моделирования видится набором *объектов,* организованных в многоуровневую иерархию, а функционирование *объекта* трактуется как совокупность нескольких *ПАРАЛЛЕЛЬНО ТЕКУЩИХ* во времени процессов. Между любыми *процессами* могут существовать *каналы связи:* посылая по ним *сигналы,* одни *процессы* могут изменять течение других. Эволюция каждого процесса представляется чередованием неких *стадий.* Моменты переключения между *стадиями,* планируются по ходу имитации, множества возможных переключений

для всех *стадий всех.* процессов фиксируются заранее, а выбор конкретного переключения всякий раз определяется набором *сигналов,* полученных *процессом* на момент переключения. Главное отличие концепции *MISS* от известных состоит в более глубокой проработке механизма совместной эволюции *процессов.*

Технологически *MISS* выделяется прежде всего тем, что это - *ИНСТРУМЕНТАЛЬНАЯ* система, а не языковая, как все упомянутые выше. Последнее не означает, конечно, что других инструментальных систем имитации не существует. Такова, в частности, созданная в начале восьмидесятых годов во ВНИИСИ АН СССР система *МИМ* (Монитор Имитационного Моделирования. Однако ее возможности существенно беднее предлагаемых *MISS.* Это можно сказать и про созданную в ИПУ АН СССР систему *SIMOD.* Иные разработки аналогичного сорта и той же степени универсальности нам не известны (о сильно развитых, но жестко ориентированных на задачи массового обслуживания зарубежных системах типа *СЛАМ* и *SIMAN-SINEMA* или отечественной *СИМПЕТРИ* речь не идет). В то же время инструментальный подход к созданию математического обеспечения для имитации представляется перспективнее языкового, так как ведет к более гибким и портабельным системам.

Набор включенных в M*ISS* инструментальных средств в основном определился из опыта конкретного моделирования, накопленного коллективом, к которому принадлежат авторы. С некой степенью условности в этом наборе можно выделить группы средств "низкого" и "высокого" уровней. К низкоуровневым относятся средства, позволяющие виртуализировать память, оперировать списковыми структурами и управлять оверлейными программами. По сути они совершенно универсальны и не только открыты пользователю, но служат также элементной базой самой *MISS.* Средства высокого уровня четче ориентированы на потребности имитации, но и среди

них многие имеют универсальный характер.

Высокоуровневую составляющую M*ISS* можно разбить на блоки *УПРАВЛЕНИЯ ПРОЦЕССАМИ, ПОДДЕРЖКИ ДАННЫХ* и *ГРАФИКИ.* Первый включает развитый аппарат планирования модельного времени, а также службу передачи *сигналов* и определения правил чередования стадий *процессов.* Второй обеспечивает автоматическое генерирование баз данных моделей и удобный доступ к их содержимому как из программ имитации, так и в режиме диалога с терминала; сюда же входит собственный интерфейс ввода-вывода. Наконец, графический блок *MISS* предоставляет редакторы и процедуры для формирования пиктограмм модельных объектов и фоновых картинок, позволяя легко программировать в 16 цветовой EGA-моде высокого разрешения "мультфильмы" эволюции моделируемых процессов, связанных с какими-то перемещениями. Нередко такая мультипликация оказывается удобным и эффективным средством отладки имитирующих программ, в чем мы и видим ее основное назначение.

MISS может использоваться на IBM-совместимых ПЭВМ класса AT/XT или *PS/2* о объемом оперативной памяти не менее 512K, с операционной системой *MS-DOS* версий 3.0 и выше, с *EGA* или *VGA* мониторами. Логика моделей и их данные описываются на собственном непроцедурном языке *MISS.* Программы алгоритмов моделируемых *процессов* должны формироваться в одной из допускаемых *MISS* языковых систем программирования: *МОДУЛА-2/SDS, МОДУЛА-2/JPI, Turbo-C, Turbo-C++.* Инструментальные средства, предназначаемые для этих программ, доступны в них как библиотечные процедуры. Обращение к иным средствам - графическим и текстовому редакторам, редактору базы данных, библиотекарю *KISS,* компилятору языка описания данных, линковщику и загрузчику моделей - осуществляется через диалоговый монитор *MISS.*

2. КОНЦЕПЦИЯ МОДЕЛИРОВАНИЯ

Каждая система имитации, будь она языковой или инструментальной, диктует определенную схему описания моделируемой реальности. Подобные схемы принято называть концепциями моделирования. В данном разделе описана та, которую предлагает *MISS.*

2.1 СТРУКТУРА ОБРАЗУЮЩИХ МОДЕЛИ

Модельная концепция *MISS* объектно-ориентирована и исходит из представления о многокомпонентности предмета имитации. Под многокомпонентностью последнего мы понимаем возможность естественного вычленения в нем некоторого количества относительно обособленных первичных составляющих, функционирование которых во взаимодействии друг с другом и формирует подлежащую имитации динамику. Модельные образы таких составляющих впредь будем называть *объектами.*. Следовательно, основное положение рассматриваемой концепции звучит так: предмет имитации есть совокупность *объектов.* Имея это ввиду, вместо словосочетания "предмет имитации" в дальнейшем естественно использовать термин *комплекс.*

Первичные составляющие реальных комплексов часто компонуются (организационно или конструктивно) в подкомплексы, те, в свою очередь, - в подкомплексы более высокого уровня и т.д. Для модельного отражения иерархий подобного рода в *MISS* введено понятие *группа.* Это - объединение какого-то числа *объектов* и нес-

кольких образований, которые сами являются *группами.* Никаких других понятий, структуризующих множество *объектов,* в M*ISS* нет. Таким образом, *комплекс* видится совокупностью *объектов,* организованных в древовидную иерархию. Верхний уровень этой иерархии в дальнейшем будем называть *головной группой.*

Поскольку модельная жизнь *комплекса* сведена к совместной деятельности *объектов,* для определения в общих чертах принятой схемы ее воспроизведения осталось уточнить подход к представлению жизни одного *объекта.* Она трактуется как несколько параллельно протекающих процессов. Кавдому назначается свой "материальный носитель", именуемый *прибором,* и в этом смысле можно говорить, что *объекты* делятся на *приборы;* однако реальных прототипов *приборов* может и не существовать.

2.2 СТРУКТУРА ОСНОВНОГО ИНФОРМАЦИОННОГО ФОНДА

При построении в MISS модели комплекса стержнем структуризации данных является разделение их по принадлежности *приборам, объектам* и *группам.* Точнее говоря, с каждой из образующих модели можно связать какие-то данные, и в совокупности наборов этих данных, организованной принятым способом компоновки *приборов* в *объекты, в объектов* - в *группы* и т.д., предлагается видеть весь основной информационный фонд модели.

В жизни среди составляющих реальных комплексов часто бывает по нескольку одинаковых и нередко в функционировании разных составляющих выделяются одинаковые процессы. При этом одинаковость, разумеется, означает лишь принадлежность к одному классу, т.е. не требует совпадения во всем. Область допустимых различий - значения неких величин, являющихся атрибутами каждого экземпляра класса. Концепция *MISS* также включает понятия

классов объектов, приборов и *групп,* а величины, чьими значениями исчерпываются различия между экземплярами одного класса, в соответствии с общепринятой терминологией будем называть *фазовыми переменными,* (хотя среди них могут быть и не меняющиеся со временем). Это определение подразумевает, что если кроме *фазовых переменных* есть еще какие-то данные по экземплярам класса, то их значения должны быть одинаковыми для всех экземпляров. Такие данные будем называть *константами* (хотя среди них могут встречаться меняющиеся со временем величины).

Итак. макроструктура данных модели описана: они поделены на *фазовые переменные* и *константы,* причем первые закрепляются за экземплярами *приборов, объектов* и *групп,* а вторые - за их *классами.* Теперь - о микроструктуре данных. Начнем с наборов *констант групп, объектов, приборов* и *фазовых переменных объектов* и *групп.* По признаку постоянства состава на протяжении всего периода имитации в любом из наборов этого сорта могут выделяться статические данные, состав которых заморожен, и динамические, состав которых с течением модельного времени может меняться. Первые должны быть объединены в одну, в дальнейшем именуемую *базовой,* запись. Этим требования к ним и исчерпываются. Что же касается динамических данных, то для их представления в *MISS* предусмотрены средства работы с одиночными записями и со списками записей. Точнее сказать, если в некий набор требуется включить динамические данные, то их надо представить несколькими записями и несколькими списками, каждый из которых будет образовываться записями постоянной структуры. При этом в *базовой записи* выделяются поля, имеющие смысл "ключей" к таким данным.

У внутренних структур совокупностей *фазовых переменных приборов* есть особенность. В таких совокупностях тоже могут встречаться и статические и динамические данные. Статические,

как и у *групп с объектами,* должны быть сведены в *базовую запись,* а что касается динамических *фазовых переленных прибора,* то они делятся на две категории. Первая - это "подвешиваемые к *базовым записям"* списки и динамические записи, аналогичные упомянутым выше. Вторая категория характерна только для *приборов* и связана с поддерживаемым *MISS* механизмом *сигналов* и с*ообщений.* Считается, что *приборы* во время своего функционирования могут вырабатывать *сигналы,* предназначенные конкретным адресатам (другим *приборам,* или самому себе в будущее) и сопровождающиеся *сообщениями.* (возможно, пустыми), а также принимать *сигналы* и *сообщения,* посланные им. Связь адресатов с отправителями обеспечивается постоянными *каналами,* фиксируемыми в описаниях *приборов, объектов* и *групп,* причем эти каналы организуются так, что данный вход данного прибора-получателя всегда связан только с одним конкретным выходом конкретного прибора-отправителя, и есть жесткое соответствие между *каналами* и типами передаваемых по ним *сообщений.*

Обычно *сигналами,* моделируются импульсы, которые возникают в автоматической части системы управления исследуемого комплекса, а *сообщениями.* - сопряженные с ними пересылки данных. Формально же пара *{сигнал, сообщение},* являясь продуктом деятельности прибора-отправителя, должна быть отнесена к его *фазовым переменным,* причем - к динамическим, поскольку эти пары то рождаются *приборами,* то уничтожаются системой (в соответствии с регламентом, который будет рассмотрен в следующем подразделе). *Сигнал* трактуется как булева величина, а *сообщение,* если оно не пустое, может быть только списком из одинаковых по структуре записей. При этом данный *сигнал* либо всегда посылается без *сообщения,* либо может сопровождаться списком из записей одинаковой и жестко связанной с ним структуры.

Теперь поддерживаемая *MISS* структура основных данных описана полностью и осталось лишь указать принятые соглашения о возможностях доступа к ним из программ, реализующих алгоритмы деятельности *приборов.* По парам *{сигнал, сообщение}* все уже сказано - они доступны только в программах соответствующих *приборов-отправителей* и приборов-получателей. Все прочие основные данные образуют общую область: с помощью средств *MISS* значения любых из них можно получить и изменить в программе любого *прибора.* Таким образом, наряду с возможностью явного обмена информацией между *приборами,* реализуемого через механизм *сигналов* и *сообщений,* существует возможность неявного - через общую область данных: имеется ввиду возможность задать значения каких-то *фазовых переменных* программой одного *прибора с* тем, чтобы они были прочитаны программой другого -

2.3 ВНУТРЕННЯЯ СТРУКТУРА ПРОЦЕССОВ

Функционирование *комплекса* видится результатом слияния деятельностей его *объектов,* а деятельность *объекта* представляется несколькими параллельно текущими *процессами.* Расщепление на *процессы* - это декомпозиция, оправданная многофункциональностью реальных объектов. Наряду с ней вводится и декомпозиция каждого *процесса* во времени. Она нужна для формализации правил взаимодействия *процессов* друг с другом.

В эволюции каждого *процесса* предлагается выделять последовательные стадии, впредь именуемые *элементами.* Естественная содержательная трактовка *элемента* - алгоритм поведения в определенных условиях. Помня о взаимна однозначном соответствии процесс-прибор, далее будем говорить, что функционирование *прибора* есть последовательное выполнение им присущих ему *элементов.*

Число допускаемых различных *элементов* прибора произвольно, но считается, что оно всегда конечно. Чередование же во времени *элементов* может быть бесконечным (если бесконечно время имитации) и организуется по правилам, отражающим действие автоматической части системы управления комплекса (см. ниже).

По отношению к модельному времени все *элементы* всех *приборов* делятся на три категории: *сосредоточенные, условно распределенные* и *распределенные. Сосредоточенные элементы* в модельном времени всегда выполняются мгновенно. Выполнение же любого из прочих *элементов* обычно имеет ненулевую продолжительность и может распределяться между несколькими последовательными шагами имитации (отсюда и названия соответствующих категорий).

Условно распределенные и *распределенные элементы* отличаются друг от друга режимами появления во времени своих результатов, под которыми надо понимать новые значения каких-то *фазовых переменных.* Для *условно распределенного элемента* они определяются только в момент его окончания и нет никаких генерируемых этим *элементом* данных, которые появлялись бы на промежуточных шагах и могли бы немедленно использоваться другими *элементами.* Что же касается *распределенного элемента,* то он, как правило, выдает некие результаты на каждом (а не только на завершающем) шаге его выполнения. Содержательно *распределенные элементы* - это отрезки процессов, описываемых эволюционными дифференциальными уравнениями. Типичный пример *условно распределенного элемента* - модель процесса решения вычислительной задачи.

Отметим, что можно было обойтись и без введенной классификации *элементов,* утяжеляющей предлагаемую концепцию моделирования: из описываемой ниже схемы синхронизации *процессов* видно, что хватило бы одних *распределенных элементов.* Однако это утяжеление оправдано соображениями вычислительной эффективности.

2.4 СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

Квазипараллельное и коррелированное течение модельных *процессов* реализуется в *MISS* через *точки синхронизации.* Это - точки оси модельного времени, к которым привязываются все выполняемые вычисления. Последние группируются также в *такты имитации,* естественно ассоциируемые с отрезками временной оси между *точками синхронизации.* Сразу подчеркнем, что смена этих точек не обязательно означает перевод системных часов: возможны такты "нулевой продолжительности". С другой стороны, если на каком-то *такте имитации* часы переводятся, то это происходит до перехода в новую *точку синхронизации,* а не в результате этого перехода.

Значения модельного времени в *точках синхронизации* определяются взаимодействием двух механизмов. Первый - это механизм назначения *элементов* на *очередной такт,* а второй - планирование моментов прерываний (в том числе моментов завершений) выполнения *элементов.* Мы сначала опишем второй механизм и принятую схему его сочетания с первым, а затем конкретизируем первый механизм.

В *MISS* каждый *сосредоточенный элемент* трактуется как один алгоритм, а *условно распределенный* или *распределенный* - как пара, включающая *таймер* и основной *алгоритм. Таймеры* служат специальными процедурами планирования времени. При вызове *таймера* должен определяться и сообщаться системе очередной нужный *элементу* момент прерывания. Простейшая (но отнюдь не самая эффективная) схема планирования моментов прерываний стоит на том, чтобы вызывать *таймеры распределенных* и *условно распределенных элементов* в начале каждого *такта* их выполнения. Наряду с ней *MISS* предлагает и другие, более эффективные (см. разд. 5). Так, например, при вызове *таймера* в момент начала выполнения

элемента можно сразу жестко зафиксировать время его окончания, тогда *таймер* больше вызываться не будет. В таких случаях будем говорить о блокировке *таймера.*

Строгое определение реализованного в *MISS* принципа синхронизации *процессов* состоит в следующем. Пусть начинается очередной *такт имитации,* в этот момент известно (почему - сейчас станет ясно), какие *элементы* должны выполнять *приборы,* и здесь:

а) для выполняемых *распределенных* и *условно распределенных элементов* вызываются незаблокированные *таймеры,* и если есть выполняемые *сосредоточенные элементы,* то шаг по времени устанавливается нулевым и вызываются их алгоритмы; если уже не установлен нулевой шаг, то в качестве момента окончания *такта* берется минимальное из времен, назначенных вызванными *таймерами,* и времен, которые заблокированные *таймеры* заказали ранее;

б) системные часы переводятся на время окончания *такта;*

в) вызываются *основные алгоритмы* тех выполняемых *условно распределенных элементов,* чьи *таймеры* запланировали их завершение на момент, совпавший с найденным временем окончания *такта;* если шаг по времени оказался ненулевым, вызываются *основные алгоритмы* всех выполняемых *распределенных элементов;*

г) для каждого прибора, завершившего выполнение *элемента* (либо потому, что он - *сосредоточенный,* либо потому, что заказанное *таймером* время завершения совпало с моментом окончания *такта),* в соответствии с правилами его функционирования (см. ниже) определяется, к какому *элементу* он должен перейти в текущей ситуации; на этом *такт имитации* завершается. Для того чтобы сформулированная схема приобрела четкость рабочего алгоритма, осталось уточнить, что подразумевается в п.б) под правилами функционирования *приборов.*

В *MISS* реализован подход, согласно которому *прибор* формирует свою последовательность *элементов* как конечный автомат, имеющий входами номер предыдущего *элемента* и полученные *сигналы,* а выходом - номер следующего *элемента.* Точнее говоря, когда *прибор* завершил выполнение очередного *элемента,* выбор следующего *элемента* определяется тем. какой именно *элемент* завершился и какие из совокупности всех входных *сигналов прибора* поступили на момент принятия решения. Данное положение формализуется введением *автоматных функций приборов.* Аргументов у каждой *автоматной функции* два: номер завершенного *элемента* и еще одно целое число, которое будем называть "номером старшего реализовавшегося события". Второй аргумент требует разъяснения. Суть в том, что для каждого *прибора* естественно разбить всевозможные комбинации значений булевых величин {есть *сигнал,* нет *сигнала*} на классы и считать, что при выборе очередного *элемента* роль играет не то - какая именно комбинация реализовалась, а то - каким классам она принадлежит. Соответственно, для каждого *элемента* определяется свой набор алгебрологических функций от булевых индикаторов наличия входных *сигналов,* причем - такой набор, что хотя бы одна функция принимает значение "истина" при любой реализации значений аргументов; эти функции, именуемые впредь *событиями,* ранжируются (нумеруются) и вторым аргументом *автоматной функции* служит номер старшего среди принявших значение "истина" *событий.*

Итак, выбор очередного *элемента* осуществляется по правилу:

а) по пришедшим на момент принятия решения *сигналам* "вычисляются" *события* из списка, отвечающего завершенному *элементу;*

б) просмотром в порядке убывания ранга (увеличения номера) определяется номер старшего реализовавшегося *события;*

в) номера завершившегося *элемента* и старшего реализовавшегося *события* подставляются в *автоматную функцию прибора,* и она дает номер следующего *элемента.*

Теперь для полного прояснения способа организации вычислений осталось уточнить режим существования *сигналов* (и связанных с ними *сообщений)* в модельном времени. Появляются, они как продукты деятельности *приборов* при выполнении ими своих *элементов,* а аннулируются автоматически. При завершении очередного *такта имитации* сразу после отработки *основных алгоритмов* к *приборам*-получателям придут все те *сигналы,* которые были посланы *приборами-отправителями* на завершающемся *такте;* эти *сигналы* (и *сообщения)* просуществуют в системе до аналогичного момента следующего *такта имитации* (будут доступны в вызываемых на этом *такте таймерах* и *основных алгоритмах* приборов-получателей).

Проиллюстрируем изложенный алгоритм синхронизации *процессов* диаграммой, показывающей последовательность вызовов *таймеров* и *основных алгоритмов.* Точки между ними подразумевают системные операции, в том числе - пересчет времени, обновление поля *сигналов* и определение переключений по *автоматным функциям.*

Особое внимание надо обратить на расположение точек пересчета модельного времени и моментов обновления поля *сигналов.* Моментами обновления разделены интервалы приема, совпадающие с *так-*

тами имитации с точностью до неких системных операций, включающих вычисления *автоматных функций.* На очередном интервале приема для обработки открыты *сигналы* и *сообщения,* посланные на предыдущем интервале. Вновь же посылаемые *сигналы* и *сообщения* поступают в накопитель и "открываются" адресатам лишь по завершении интервала, причем старое сигнальное поле в этот момент уничтожается. Подчеркнем, что *автоматные функции* всегда работают на только что обновленном сигнальном поле.

2.5 МОДЕЛИРОВАНИЕ КОМПЛЕКСОВ С ПЕРЕМЕННОЙ ЧИСЛЕННОСТЬЮ ОБЪЕКТОВ

По ряду технологических соображений, а также из-за некой несогласованности с принятой схемой сопряжения *процессов* через фиксируемые *каналы* связи в *MISS* не включены средства, позволяющие варьировать состав принципиальных компонент модели при *имитации.* Проще говоря, структуру *групп, объектов* и *приборов* модели предлагается считать замороженной. Это, однако, не означает, что *MISS* не годится для моделирования комплексов переменного состава. Здесь возможен следующий подход: надо оценить сверху непостоянную численность соответствующих *объектов,* завести их в модели в этом количестве и работать с полученным постоянным составом. Его будет достаточно и когда принятая оценка окажется меньше числа "вновь рождающихся *объектов"* - просто потребуется как-то организовать "повторное использование" *объектов* из заведенного постоянного набора. В расчете на этот прием в *MISS* предусмотрено деление *объектов* на *активные* и *пассивные* и включены средства их перевода из одного статуса в другой. При этом подразумевается, что никакая вычислительная работа по имитации функционирования *пассивных объектов* вестись не будет.

3. ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ В MISS

Модельная концепция, изложенная в предыдущем разделе, не так уж далека от традиционных и, например, в части представления структуры компонент модели и использования *сигналов* для сопряжения *процессов* идентична концепции *SDL.* Поэтому, если говорить об особенностях *MISS,* то прежде всего это - особенности воплощения в ней данной концепции. Стержневым принципом технологии *MISS* является ее декомпозиционность - установка на то, что все поддающееся раздельному программированию должно и программироваться раздельно, а завершающая операция создания модели должна иметь характер увязки готовых фрагментов.

3.1. ПРОГРАММИРОВАНИЕ ЛОГИКИ МОДЕЛИ

Фрагменты имитационной модели, реализуемой в *MISS,* распадаются на логические и вычислительные. В этом состоит первое проявление декомпозиционной сути технологии *MISS:* логика модели и вычислительная начинка представленных в ней процессов программируются раздельно, причем первая - на понятном *MISS* специальном непроцедурном языке.

Под логическими фрагментами модели подразумеваются замкнутые описания структур классов ее компонент *(групп, объектов* и *приборов).* Впредь будем называть их *спецификациями. Спецификации* отдельных классов компонент программируются формально

независимо друг от друга. При сборке модели они должны находиться в библиотеке *MISS* в откомпилированном виде и по ним автоматически генерируется иерархическая база данных модели.

Детальный разбор содержания *спецификаций* отложим до следующего раздела а здесь ограничимся кратким пояснением: для *групп* и *объектов* они должны включать параграфы определения состава *{подгрупп,* и *объектов* для *группы, приборов* для *объекта};* в *спецификации класса приборов* обязательны параграфы перечня *элементов* и определения *автоматной функции;* любая *спецификация* может иметь параграфы определений структур *фазовых переменных* и *констант,* связываемых с модельными компонентами соответствующего класса, а также - параграф коммутации *сигналов.*

Спецификация может храниться в библиотеке *MISS* в трех формах, а именно - как:

а) "сырой" текст;
б) кодированный текст;
в) результат компиляции кодированного текста.

Через форму а) проходить не обязательно; формы б) и в) обойти нельзя. Финальной является форма в), причем после компиляции в нее формы б) сама б) автоматически уничтожается. Из любой формы возможны экспорт в текстовой DOS-файл и распечатка; форма а) допускает также просмотр и редактирование с экрана. Обе операции осуществляются через встроенный в *MISS* экранный текстовой редактор, возможности которого не слишком велики, но вполне достаточны для редактирования содержимого обычно небольших по объему *спецификаций.*

(Унификации включаются в библиотеку командами импорта в форду а) или б). По команде сымпортировать *спецификацию* в форму а) *MISS* запросит имя файла-источника, и если введенное имя найдется в доступных DOS-каталогах, перепишет соответствующий файл

в раздел библиотеки, где хранятся сырые тексты, а иначе заведет в этом разделе пустую запись (как бы то ни было, всегда есть возможность довести импортированный текст до нужных кондиций с помощью редактора *MISS)*. Результат импорта будет зарегистрирован под введенным именем. Импорт в форму б) значительно содержательнее. Он используется не только для перевода текстового DOS-файла в кодированный текст спецификации в библиотеке *MISS,* но и как средство преобразования спецификации из формы а) в форму б). В обоих случаях подается одна команда, после чего *MISS* запрашивает имя источника импорта. Если введенное имя найдется в оглавлении раздела сырых текстов, импорт пойдет оттуда; если нет, *MISS* обратится к доступным DOS-каталогам и при наличии в них соответствующего файла спецификация будет взята из него, а иначе импорт просто не состоится.

При импорте в форму б) осуществляется полный синтаксический и частичный семантический контроли содержимого источника. По выявлении ошибки импорт сразу прерывается и выдается фраза, поясняющая суть ошибки, и кусок импортируемого текста, где она обнаружена. Дальнейшее зависит от того, откуда шел импорт: если из DOS-файла, то больше ничего не произойдет; если же - из раздела сырых текстов, *MISS* предложит отредактировать источник и при согласии пользователя вызовет свой редактор, сразу показав на экране страницу текста с ошибочным фрагментом (он будет подсвечен). Таким образом, хотя возможен и прямой импорт из (образованного любым способом) текстового DOS-файла в форму б), учитывая рассмотренные удобства устранения ошибок, форму а) лучше не обходить.

Та часть библиотеки *MISS,* в которой спецификации хранятся в формах б) и в), разбита на разделы: спецификации *групп, объектов* и *приборов* разнесены. При успешном импорте спецификации

в форму б) она регистрируется в соответствующем разделе под именем, с которого начинается (см. разд. 4). Чтобы привести ее в финальную форму в), нужную для сборки модели, надо пропустить форму б) через компилятор MISS.

В заключение подраздела подчеркнем, что выполнение рассмотренных действий над спецификацией не требует соотнесения ее содержания с содержанием каких-либо других спецификаций.

3.2 ПРОГРАММИРОВАНИЕ ВЫЧИСЛИТЕЛЬНЫХ СОСТАВЛЯЮЩИХ МОДЕЛИ И ЕЕ СБОРКА

Для сборки в *MISS* готовой к прогонке модели кроме откомпилированных спецификации, всех классов ее компонент нужны загрузочные модули всех процессов. Точнее говоря, для каждого класса *приборов* надо в одной из разрешенных языковых систем *(МОДУЛА-2/SDS*, *МОДУЛА-2/JPI*, *Turbo-C)* составить, откомпилировать и слинковать отдельную программу (главный модуль *МОДУЛЫ-2* или единицу компиляции, содержащую подпрограмму с именем main, для "C"). Эти программы впредь будем называть модулями приборов. При сборке или запуске модели они должны лежать в доступном DOS-каталоге, имея те же имена, что сами *приборы.,* либо - имена, специально указываемые в соответствующих спецификациях (подразумеваются имена без расширений, а расширение всегда будет "OVY" или "EXE" и его "прилепит" линковщик используемой языковой системы). Возможность дать *модулю* прибора имя, не совпадающее с именем самого *прибора,* позволяет привязывать *приборы* разных классов к одному и тому же модулю, сокращая тем самым имитирующую программу; при этом классы могут действительно быть разными, скажем, иметь разные *автоматные функции.*

Модули приборов служат основными вычислительными узлами модели при ее реализации в *MISS.* В терминах системы они классифицируются как *виртуальные.* Кроме них в комплект имитирующих программ могут входить другие *виртуальные модули* и в том числе один особый модуль, именуемый *сервисным* (см. разд. 8). *Виртуальным модулям* как таковым посвящен подразд. 6.13.

На содержание программ для *MISS* налагается следующее существенное ограничение: используя *МОДУЛУ-2 НЕЛЬЗЯ ОБРАЩАТЬСЯ НИ К ОДНОМУ ИЗ БИБЛИОТЕЧНЫХ МОДУЛЕЙ СИСТЕМ SDS И JPI, ПРЯМО ИЛИ КОСВЕННО ЗАВИСЯЩИМ ОТ БИБЛИОТЕЧНОГО МОДУЛЯ Storage, А ТАКЖЕ - К ВСТРОЕННЫМ ФУНКЦИЯМ NEW И DISPOSE*; *В ПРОГРАММАХ НА ЯЗЫКЕ* "С" *НЕЛЬЗЯ ДОПУСКАТЬ ОБРАЩЕНИЙ К СТАНДАРТНЫМ ФУНКЦИЯМ alloc, malloc.* Этот запрет фактически блокирует прилагаемые к разрешенным компиляторам библиотеки, но взамен дается собственная среда программирования *MISS* (см. разд. 7).

Каждому *элементу прибора* в *модуле прибора* должна отвечать некая процедура без параметра (она может быть и пустой), реализующая основной *алгоритм элемента.* Для каждого *несосредоточенного элемента* требуется также определить процедуру *таймера.* Ей тоже запрещено иметь параметры, и она должна при выполнении в любой ситуации вызывать хотя бы одну из инструментальных процедур планирования времени. Привязка процедур *элемента* к его имени из соответствующей *спецификации* обеспечивается вызовом в теле *модуля прибора* специальной процедуры *MISS.* Применительно к "С" под телом *модуля прибора* подразумевается подпрограмма *main.*

Схема вызовов процедур *элементов* была рассмотрена в подразд. 2.4. Она и сформулированные в двух предыдущих абзацах ограничения являются единственной основой для регламентации их содержимого. Для обращений в них к базе данных модели (в том числе - для посылки и приема *{сигналов, сообщений.}),* для исполь-

зования в них графических средств *MISS* и т.д. имеется широкий спектр инструментальных процедур.

Когда в библиотеке *MISS* лежат в откомпилированной форме все необходимые *спецификации* и построены все модули *приборов,* модель можно собрать. Как следует из подразд. 2.1, "разворачивание" иерархии компонент модели должно начинаться с некой *головной группы.* Сборка модели осуществляется применением соответствующей команды *MISS* к ее *спецификации.* Подключение прочих *спецификаций* и модулей *приборов* выполняется по совпадению имен.

Во время сборки может выявиться ошибка и тогда сборка будет прервана с соответствующей диагностикой. Ясно, что дело не пойдет, если не хватает каких-то нужных *спецификаций* или нет каких-то нужных модулей. В иерархии *групп* может обнаружиться цикл. Наконец, возможны ошибки в сопряжении *спецификаций* по *каналам* связи. Из них самая простая - наличие в *головной спецификации* внешних входных или выходных *каналов:* поскольку у каждого *сигнала* должен быть конкретный получатель и отправитель, а таковых заведомо не найдется для внешних *сигналов головной группы,* то последние просто не имеют права на существование. Более сложные ошибки - когда, например, в *спецификации прибора* показано десять внешних входов, а в *спецификации объекта,* содержащего этот *прибор,* у последнего задействовано только пять входов. Наконец, самая тонкая ошибка - когда при сопоставлении *спецификаций* получается, что прибор-отправитель может посылать по некоему *каналу сигналы* с *сообщениями* одного типа а прибор-получатель ждет по тому же *каналу* что-то иное. Однако *MISS* обнаружит любую из этих ошибок и неправильная сборка исключена.

При успешной сборке модели на твердом диске сгенерируется ее база данных, единицами которой поначалу будут *базовые записи фазовых переменных* и *констант,* "ключи" для посылки и приема

{*сигналов, сообщений*}, некие служебные данные (т.е. формируется статическая часть базы). В секции моделей в библиотеке *MISS* появится. (под именем *головной группы*) запись, нужная для доступа к созданной базе и для сборки программы имитации при запуске модели на счет. Подчеркнем, что программу собирает не линковщик, а загрузчик *MISS*. Поэтому после модификаций модулей заново собирать модель не надо и можно работать с прежней базой данных.

3.3 ПРОГРАММИРОВАНИЕ ОБЩЕЙ ОРГАНИЗАЦИИ ЭКСПЕРИМЕНТА

Кроме основных процедур, воспроизводящих процессы, для проведения численных экспериментов с моделью могут понадобиться некие дополнительные процедуры. Удобно, например, оформлять таковыми инициализацию данных перед началом счета и обработку результатов по его завершении. Нередко хочется иметь процедуру, которая вызывалась бы по асинхронным прерываниям с клавиатуры. Наконец, иногда (особенно - во время отладки) полезно, чтобы какая-то процедура выполнялась в конце каждого *такта* имитации. Реализовать все это позволяет специальный сервисный *модуль,* который можно составить (сформировать как главную программу, откомпилировать и слинковать в одной из разрешенных языковых систем) в дополнение к модулям *приборов.* Он должен иметь имя модели и в нем можно определить все перечисленные процедуры и сообщить о них системе. При сборке модели *MISS* проверит наличие *сервисного модуля* в доступных DOS-каталогах, и если он задан, обеспечит выполнение указанных процедур в надлежащие моменты имитации.

Второе назначение *сервисного модуля* - служить "общим фондом" подпрограмм и данных, только он позволяет обеспечить доступ нескольким модулям *приборов к* одной процедуре. Подробно устройство *сервисного модуля* разбирается в разд. 8.

4. ЯЗЫК СПЕЦИФИКАЦИЙ

В данном разделе подробно разобран специализированный непроцедурный язык, на котором должны составляться спецификации классов компонент модели- Отметим, что хотя аккуратнее употреблять термин "спецификация *класса компонент*", мы ради краткости будем пользоваться словосочетанием "спецификация *компоненты*"- Лингвистические формулы записываются ниже в стандартной нотации, согласно которой служебные слова выделяются жирным шрифтом, разделители - двойными кавычками, а необязательные включения - квадратными или фигурными скобками, причем фигурные скобки указывают на возможность повторения. Разбор языка начнем с двух нетривиальных конструкций, применяемых в *спецификациях* всех типов. Это - *определение структуры данных* и *оператор коммутации.*

4.1 ОПРЕДЕЛЕНИЕ ТИПОВ ЗАПИСЕЙ ФАЗОВЫХ ПЕРЕМЕННЫХ И КОНСТАНТ; КАТАЛОГИЗИРОВАННЫЕ ТИПЫ

Собирая модель, *MISS* автоматически генерирует ее базу данных, руководствуясь при этом содержимым специальных параграфов *спецификаций, групп, объектов* и *приборов.* Речь идет о параграфах определений типов *фазовых переменных* и *констант.* Их синтаксис различается только открывающей ключевой конструкцией, а в остальном сводится к правилам определения структур данных *MISS. Определение структуры данных MISS* похоже на **RECORD** в *МОДУЛЕ-2*

или struct в языке "C". Различия в том, что некоторые типы полей, допускаемые *MISS,* отсутствуют в *МОДУЛЕ-2* и "C", и наоборот. Формально *определение структуры данных* есть набор *определений списков полей,* т.е. - набор списков идентификаторов, завершающихся *указаниями типов* поименованных величин. При этом допускаются *простые типы, структурные типы* и *тип массива.* К *простым* относятся *встроенные* и *каталогизированные типы MISS,* фиксируемые своими идентификаторами. *Структурные типы MISS* - это типы обычных записей, ключей виртуальных записей и ключей виртуальных списков. Их *указания* выглядят как выделенные служебными словами наборы *определений списков полей* (здесь - рекурсия !). Наконец, *тип массива* задается по тем же правилам, что в *МОДУЛЕ-2,* но диапазоны индексов допускаются лишь числовые и тип элемента массива должен быть *простым* или *структурным типом MISS* (а не *МОДУЛЫ-2*).

В стандартной нотации *определение структуры данных MISS* выглядит следующим образом:

ОпрСтруктДанных: **::=** *ОпрГруппыПолей* { *ОпрГруппыПолей* }.
ОпрГруппыЛолей **::=** *Имя* { "." *Имя* } ":" *УказаниеТипа* .

УказаниеТипа **::=** *ИмяПростогоТила* ";" |
УказаниеСтруктурногоТипа |
УказаниеТипаМассива |
УказаниеТилаПеречисления.

ИмяПростогоТипа **::=** *ИмяВстроенногоТипа* |

ИляКаталогизированногоТипа.

ИмяВстроенногоТипа **::=** BOOLEAN | CHAR | BYTE | CARDINAL |
BITSET | INTEGER | LONGINT | REAL |
INDEFINITE | *TEXT* | *NAME* | *PICTURE* |
BACKGRND.

ИмяКаталогизированногоТипа **::=** *Имя*

УказаниеСтруктурногоТипа **::=** *ИмяСтруктурногоТипа*
ОпрСтруктДанных END ";".

ИмяСтруктурногоТипа **::=** RECORD | LIST OF | *BLOCKS OF* |
ADDRESS OF.

УказаниеТипаМассива **::=** *ARRAY*
Диапазон { *","* Диапазон }
OF ТипЭлемента.

Диапазон *::* = " [" *ЦелоеЧисло ".."*
ЦелоеЧисло "]".

ТипЭлемента **::=** *ИмяПростогоТипа* ";" |
УказаниеСтруктурногоТипа.

УказанияТипаПеречисления *::* = " ("
Имя { "," *Имя* } ");".

В этих формулах термин Имя означает любую последовательность символов длиной без пробелов не более 20, не содержащую использованных в контексте разделителей.

Понятие *каталогизированного типа MISS* и специфичные для *MISS* встроенные типы *INDEFINITE, TEXT, NAME, PICTURE, BACKGRND, LIST OF, BLOCKS OF, ADDRESS OF* будут разобраны ниже. Прочие встроенные типы определяются следующей таблицей соответствий:

Тип в MISS	Тип в МОДУЛЕ-2/SDS	Тип в МОДУЛЕ-2/JPI	Тип в Языке С	Длина в байтах
BOOLEAN	BOOLEAN	BOOLEAN	Char	1
CHAR	CHAR	CHAR	Char	1
BYTE	BYTE	BYTE	Char	1
перечисл.	перечисл.	перечисл.		1
BITSET	BITSET	BITSET	Unsigned	2
CARDINAL	CARDINAL	CARDINAL	Unsigned.	2
INTEGER	INTEGER	INTEGER	int	2
LONGINT	LONGINT	LONGINT	long	4
REAL	REAL	LONGREAL	double	8

Четыре первых типа таблицы (и только они !) имеют нечетную длину, в связи с этим отметим следующее: *РАЗМЕЩАЯ ДАННЫЕ В БАЗЕ, MISS ВЫРОВНЯЕТ НАЧАЛА МАССИВОВ И ПОЛЕЙ ЗАПИСЕЙ НА ГРАНИЦЫ СЛОВ, А КОМПИЛЯТОР МОДУЛЫ-2/JPI ХРАНИТ ВСЕ ДАННЫЕ ПОДРЯД, И ЭТО НУЖНО*

УЧЕСТЬ, РАБОТАЯ С БАЗОЙ MISS В МОДУЛЕ-2/JPI (см.подразд. 5.2).

Чтобы была возможность использовать в разных *спецификациях* одни и те же структурные типы данных, в *MISS* введен аппарат *каталогизированных типов:* в библиотеке *MISS,* наряду со *спецификациями групп, объектов* и *приборов,* можно заводить в специально выделенной секции *спецификации* структурных типов данных, тем самым каталогизируя их- Заводятся они, как и прочие *спецификации,* операциями импорта и компиляции (см. разд. 3). Синтаксис *спецификации каталогизированного типа* таков:

ОписательКаталогТипа. :: = Имя "=" **RECORD**
ОпрСтруктДанных **END ";".**

Смысл употребленных в правой части терминов тот же, что в предидущей серии определений, причем *Имя* - и есть то имя, под которым данный тип будет зарегистрирован в библиотеке и под которым его можно упоминать в других *спецификациям.* Отметим, что приведенное определение рекурсивно, т.е. допускает использование в *спецификации каталогизированного типа* имен иных типов того же сорта. Циклов в этой рекурсии быть не должно, и *MISS* не допустит их, отказывая компилировать *спецификацию,* если уже не заведены или не откомпилированы *спецификации* всех упоминаемых в ней *каталогизированных типов.*

Примеры *спецификаций каталогизированных типов:*

```
VECTOR = RECORD
           X, Y, Z : REAL;
         END;

MATRIX = RECORD
           Rows : ARRAY [1..3] OF VECTOR;
         END;
```

С каталогизацией непосредственно связан встроенный тип *INDEFINITE.* Он означает, что соответствующие переменные могут

быть записями любого из *каталогизированных типов.* Это - принятый в *MISS* способ поддержки варьируемых структур данных,

Именами *TEXT* и *NAME* помечаются динамические переменные текстовой природы. Первое относится к текстам произвольной длины, хранимым в специальном формате. Основное их назначение - служить развернутыми комментариями к стгецификациям Тип *NAME* тоже введен для справочных текстов, но - для коротких. Работать с текстовыми переменными можно только через диалоговый редактор данных *MISS,* рассматриваемый в разд. 8.

В номенклатуре встроенных типов *MISS* есть два графических. Это - *PICTURE* и *BACKGRND.* Оба определяют ключи неких картинок, формально являющихся динамическими записями особой структуры. Первый тип относится к пиктограммам размера не более 80*70 пикселов. что составляет 1/40 экрана EGA. Второй -тип ключа полномерной картинки- Переменные обоих типов предназначаются для графической анимализации моделей. Использование их в программах имитации в качестве фактических параметров соответствующих инструментальных процедур (см. разд. 5) позволяет создавать "мультфильмы" эволюции воспроизводимых процессов. Инициализация этих переменных, включающая и создание ассоциируемых с ними изображений, обычно осуществляется через графические редакторы *MISS,* но может выполняться и программно.

Все упомянутые выше типы динамических данных реализуются через включенную в *MISS* систему виртуализации памяти (см. разд. 6). Связаны с ней. причем - наиболее тесно, и три еще не разобранных динамических типа. Они соответствуют примитивам этой системы: типы *BLOCKS OF* и *LIST OF* относятся к ключам блокированных и неблокированных списков из записей одной длины, а тип *ADDRESS OF* - к ключам одиночных динамических записей. Структура записей всегда фиксируется текстом, следующим за именем типа.

Для работы с ними и для создания, уничтожения, наращивания и сокращения списков есть соответствующие инструментальные процедуры. Кроме того, это можно делать через диалоговый редактор данных *MISS* (см. разд. 9).

При инициализации все динамические структуры размещаются в *КОПИИ* базы данных, создаваемой *MISS* на время сеанса в виртуальной памяти. По завершении сеанса эту копию можно сохранить и тогда сохранятся все заведенные данные; тем самым будет обеспечена возможность их использования в последующих сеансах. Пример определения *структуры данных MISS:*

```
numb :  CARDINAL;
vect :  VECTOR;
cards : LIST OF
            card :  RECORD
                        name : (knave, queen, king);
                              pict : PICTURE;
          END;
          color : (spades, leaves, diamonds, harts);
          trump : BOOLEAN;
          END;
```

4.2 ОПЕРАТОРЫ КОММУТАЦИИ СИГНАЛОВ

Согласно принятой концепции в собранной модели могут существовать *каналы* связи, по которым *приборы* будут обмениваться *сигналами* и *сообщениями.* Фрагменты этих *каналов* определяются в *спецификациях групп, объектов* и *приборов.* Стыковка фрагментов осуществляется во время сборки модели.

И способ определения фрагментов и процедуру их стыковки легче всего понять, оперируя визуальным образом соединяемых попарно контактных разъемов. Представим себе, что каждая *группа*

(объект) имеет входной и выходной разъемы, у которых столько контактов, сколько разных *сигналов* может приходить извне к ее (его) *приборам* и посылаться вовне ее (его) *приборами,* соответственно (под *приборами группы* понимаются все приборы ее собственных *объектов, объектов* ее *подгрупп, объектов подгрупп* ее *подгрупп* и т.д.). Тогда фрагмент *канала* связи в *группе* - это соединение двух контактов пары разъемов в одном из вариантов:

а) один разъем входной, а другой выходной и оба принадлежат каким-то составляющим *группы* (внутренняя коммутация);

б) оба разъема входные или выходные и один принадлежит самой *группе,* а другой - ее составляющей (внешняя коммутация). Считая, что разъемы для сигнального общения с внешним миром существуют и у *приборов,* фрагмент *канала* связи в *объекте* определим аналогично, причем варианты пары разъемов будут такими:

а) один разъем входной, а другой выходной и оба принадлежат *приборам объекта* (внутренняя коммутация);

б) оба разъема входные или оба выходные и один принадлежит самому *объекту,* а другой - его *прибору* (внешняя коммутация).

Если *прибору* дано принимать или генерировать *сигналы,* в нем тоже надо задать фрагменты *каналов* связи. Они необходимы потому, что *приборам* разрешается посылать *сигналы* себе, и в силу этого их внешние разъемы, упомянутые выше, не могут быть терминальными. Таковыми служат внутренние входные и выходные разъемы, и в *приборе,* участвующем в сигнальных обменах, должны быть фрагменты *каналов* связи, которые соединят эти разъемы с внешними. Точнее говоря, фрагмент *канала* связи в *приборе* есть соединение контактов пары разъемов, и альтернативы таковы:

а) один разъем входной, а другой выходной и оба внутренние (внутренняя коммутация);

б) оба разъема входные или оба выходные и один внутренний, а

другой внешний (внешняя коммутация).
В языке *MISS* фрагменты *каналов* связи в компонентах модели фиксируются *операторами коммутации,* сводимыми в отдельные параграфы *спецификаций.* Общий вид *оператора коммутации* таков:

ОперКоммГруппыИлиОбьекта :: = АдресКонтактаПрихода Сигнала
"=" АдресКонтактаИсхода Сигнала ";".

ОперКоммПрибора **::=** *АдресКонтактаПрихода Сигнала*
"=" *АдресКонтактаИсхода Сигнала*
["+" *ИмяКаталогизированногоТипа*] ";".

Один оператор определяет один фрагмент канала связи, причем четко фиксируется направление движения *сигнала* - от контакта, указанного за равенством, к контакту, указанному перед ним. Что конкретно может быть записано в качестве адресов контактов - зависит от того, принадлежит ли фрагмент *группе, объекту* или *прибору.* При этом указание имени *каталогизированного типа* в *операторе коммутации сигналов* в *приборе* означает, что по соответствующему *каналу сигналы* могут посылаться с *сообщениями,* в виде списков, состоящих из записей указанного типа. Другого способа привязки *сообщений* к *сигналам* нет. Синтаксис адресов контактов в *операторах коммутации* определяется формулами вида:

АдресКонтактаГруппы, :: = ИндексКонтакта |
ИндексКонтакта ИмяПодгруппы ИндексПодгруппы. |
ИндексКонтакта ИмяОбъекта ИндексОбъекта.

АдресКонтактаОбъекта **::=**
ИндексКонтакта [*ИмяПрибора ИндексПрибора*].

АдресКонтактаПрибора **::=** *ИндексКонтакта* [*INT*].

Если адрес контакта сводится к его индексу, то это значит, что контакт принадлежит внешнему разъему описываемой компоненты модели. *ИмяПодгруппы, ИмяОбъекта* и *ИмяПрибора* - суть идентифи-

каторы соответствующих классов компонент. Служебное слово *INT* помечает внутренние разъемы *приборов.* Термины *ИндексКонтакта, ИндексПодгруппы, ИндексОбъекта* и *ИндексПрибора* формально рас-расшифровываются одинаково:

ИндексКонтакта, ИндексПодгруппы, ИндексОбъекта,
ИндексПрибора **::=** "[" *ПростоеЦелоеВыражение* "]".

ПростоеЦелоеВыражение **::=** *ЦелоеБезЗнака* |
[*Целое Знак*] "*i**" *ЦелоеБезЗнака* |
[*Целое Знак*] *ЦелоеБезЗнака* "**i*" |
[*Знак*] "*i**" *ЦелоеБезЗнака Знак ЦелоеБезЗнака* |
[Знак] *ЦелоеБезЗнака* "**i*" Знак *ЦелоеБеэЗнака.*

Во второй формуле все варианты, кроме первого, относятся к случаю, когда записи *оператора коммутации* предшествует

Определитель диапазона **::=** "*i*=" *Целое* ".." *Целое* ":".

Это - конструкция, позволяющая одной записью *оператора коммутации.* определить целую группу фрагментов *каналов* связи, а каких именно - ясно из контекста. Осталось только подчеркнуть, что указываемые в квадратных скобках номера *подгрупп* и *объектов* или *приборов* являются порядковыми во внутренней индексации той *группы* или *объекта,* в чей описатель войдет оператор. Примеры *операторов коммутации:*

i = 1..10 : [2**i*] ОбъектПервогоТипа [0] =
[1] ПодгруппаВторогоТипа [30-*i*];
[0] = [1] ОбъектВторогоТипа [2];
i = 1..4 : [1] *INT* = [2];

4.3 СИНТАКСИС ОПИСАТЕЛЕЙ ГРУПП И ОБЪЕКТОВ

Двух предыдущих подразделов достаточно, чтобы без подробных дополнительных разъяснении определить принятые в *MISS* правила формирования *спецификаций групп* и *объектов.* И те, и другие могут

содержать по четыре параграфа, из которых обязателен только головной, с перечнем составляющих. Формальный синтаксис *спецификации групп* таков:

СпецификацияГруппы **::=** *ИмяГруппы "="* **GROUPE**
ПереченьСоставляющих; { *ПереченъСоставляющих* }

END";"
[*ПараграфФазовыхПеременных*]
[*ПараграфКонстант*]
[*ПараграфКоммутации*].

ПараграфФазовыхПеременных; ***::=*** **PHASETYPE** "=" **RECORD**
ОпрСтруктДанных
END ";".

ПараграфКонстант ***::=*** **CONSTTYPE** "=" **RECORD**
ОпрСтруктДанных
END ";".

ПараграфКоммутации **::= CONNECTIONS** "="
ГруппаКоммутации { *ГруппаКоммутации.* } **END ";".**

Смысл использованного здесь термина *ОпрСтруктДанных* разобран в подразд. 4.1, а несложные формулы определений других новых терминов выглядят следующим образом:

ПереченьСоставляющих **::=** *ПереченьПодгрупп* | *ПереченьОбъектов.*
ПереченьПодгрупп **::= GROUPES** ":" *СписокКомпонент.*
ПереченьОбъектов **::= OBJECTS** ":" *СписокКомпонент.*
СписокКомпонент :: = *ИмяКомпоненты "* (" *ЧислоЭкземпляров* ")"

{"," *ИмяКомпоненты* "(" *ЧислоЭкземпляров* ")" } ";".

ГруппаКоммутации ::= [*ОпределительДиапазона*]
ОператорКоммутации.

В этих формулах *ИмяГруппы* и *ИмяКомпоненты.* суть имена классов компонент соответствующего типа, а формально - идентификаторы длиной не более 20 символов; *ЧислоЭкземпляров* - положительное целое, равное числу составляющих указанного перед скобкой клас-

са; *ОпределительДиапазона, ОператорыКоммутации* - термины из предыдущего раздела.

Имя составляющей в *спецификации группы,* имя соответствующей компоненты модели согласно *спецификации,* имя самой *спецификации* в библиотеке *MISS,* - все это одно и то же. По таким именам осуществляется сборка модели. Требуется, чтобы они были уникальными: нельзя одинаково назвать какой-нибудь *объект* и *группу*. Понятно, что в адресах контактов в *операторах коммутации* могут встречаться только имена из первого параграфа *спецификации,* причем индекс составляющей из адреса контакта должен быть строго меньше (указанного в первом параграфе) числа составляющих с данным именем, поскольку индексация ведется с нуля.

Правила формирования *спецификаций объектов* аналогичны представленным. Поэтому мы приведем их формулы без пояснений:

ОписательОбъекта **::=** *ИмяОбьекта* "=" **OBJECT**
ПереченьПриборов { *ПереченьПриборов* } **END ";"**
[*ПараграфФазовыхПеременных:*] [*ПараграфКонстант*]
[*ПараграфКоммутации*].

ПереченьПриборов **::= DEVICES** ":"
ИмяПрибора "(" *ЧислоЭкземпляров* ")"
{"," *ИмяПрибора* "(" *ЧислоЭкземпляров*")" } ";".

ПараграфФазовыхПеременных **::= PHASETYPE** "=" **RECORD**
ОпрСтруктДанных:
END ";".

ПараграфКонстант **::= CONSTTYPE** "=" **RECORD**
ОпрСтруктДанных:
END ";".

ПараграфКоммутации. **::= CONNECTIONS** "="
ГруппаКоммутации. { *ГруппаКоммутации* }
END ";".

Для иллюстрации этих правил приведем два примера спецификаций:

```
КОМАНДА = GROUPE
   OBJECTS : ТРЕНЕР(1). ФУТБОЛИСТ (16), ВРАТАРЬ(2);
   GROUPES : МАССАЖИСТЫ (1);
END;

CONNECTIONS =
                    [0] МАССАЖИСТЫ[0] = [0] ТРЕНЕР[0];
          i=0..15 : [0]  ФУТБОЛИСТ[i] = [i+2] ТРЕНЕР[0];
          i=0..15 : [1]  ФУТБОЛИСТ[i] = [i];
                                 [ 16] = [1] ТРЕНЕР[0];
          i=0..15 :                  [i] = [0] ФУТБОЛИСТ[i];
END;

PHASETYPE = RECORD
     болельщики : LONGINT;
END;

CONSTTYPE = RECORD;
     ПризЗаПобеду : CARDINAL;
END;

ФУТБОЛИСТ = OBJECT
          DEVICES : Мозг(1), Голова(1), ПараНог(1);
END;

CONNECTIONS =
   i=0..1 : [i] Мозг[0] = [i];
            [0]  Голова[0] = [0] ПараНог[0];
  i=0..1 : [i] ПараНог[0] = [[i] Мозг[0];
                      [0] = [1] ПараНог[0];
END;

PHASETYPE = RECORD
      точка, скорость : ВЕКТОР;
вес, рост : REAL;
номер  : CARDINAL;
травмы : LIST OF
         ЧастьТела : (ЛеваяНога, ПраваяНога);
         тяжесть : (тяжелая, легкая);
         ВремяПолучения : REAL;
       END;
END;
```

4.4 СИНТАКСИС ОПИСАТЕЛЕЙ ПРИБОРОВ

Лингвистическая формула спецификации *прибора* такова:

ОписателъПрибора :: = *ИмяПрибора*
["(" *ИмяМодуляПрибора* ")"] "=" **DEVICE**
СписокЭлементов { *СписокЭлементов* }
УказаниеКорневогоЭлемента
END ";"

SWITCHES "=" *АвтоматнаяФункция* **END ";"**

[*ПараграфФазовыхПеременных*]

[*ПараграфКонстант*]

[ПарграфКоммутации].

Прежде всего отметим, что спецификация *прибора* может открываться не одним, а двумя именами. Второе, указываемое в круглых скобках, не обязательно. Это - имя (без расширения) *модуля прибора,* привязываемого к классу *приборов,* который задает спецификация (см- подразд. 3.2). Если оно опущено, будет считаться, что имена *прибора* и модуля совпадают.

Необязательные параграфы спецификации *прибора,* выделенные квадратными скобками, определяются так же, как для спецификаций групп и *объектов.* Поэтому необходимо разобрать форматы только двух первых параграфов; их назначение понятно из контекста и из изложенной в разд. *2* концепции моделирования-

В головном параграфе спецификации прибора одним или несколькими предложениями перечисляются все выполняемые *прибором элементы* и фиксируется корневой *элемент, с* которого по умолчанию будет начинаться модельная жизнь каждого экземпляра *прибора.* Имена *элементов* должны быть уникальными в пределах спецификации: нельзя одинаково назвать два разнотипных *элемента* одного *прибора,* но использование одного имени в спецификациях разных

приборов не возбраняется. Формально синтаксис первого параграфа определяется так:

СписокЭлементов **::** = **FASTELS ":"** *СписокИмен* |
SLOWELS ":" *СписокИмен* |
CONVELS ":" *СлисокИмен.*

СписокИмен **::=** *ИмяЭлемента* { "." *ИмяЭлемента* } ";".

УкаэанисКорневогоЭлемента :: = **ROOTELM** ":" *ИмяЭлемента* ";" -

Здесь *ИмяЭлемента* - идентификатор длиной не более 20 символов, а ключевые слова **FASTELS, SLOWELS и CONVELS** помечают *сосредоточенные, распределенные* и *условно распределенные элементы.*

Второй параграф определяет *автоматную функцию прибора.* В нем для каждого *элемента* должны быть перечислены все разрешенные переходы, т.е. названы все *элементы,* на которые *прибор* может переключаться по завершении данного, и указаны *события,* в зависимости от которых реализуется то или иное переключение (см. подразд. 2.4). Формат параграфа таков:

АвтоматнаяФункция **::=** *ПереключенияЭлемента*
{ *ПереключенияЭлемента* }.

ПереключенияЭлемента :: = *ИмяЭлемента* ":" *СписокПереходов*.

СписокПереходов **::=** { *ИмяЭлемента* "," *ЗаписьСобытия* "."}
ИмяЭлемента ";".

Понятно, что перед двоеточием во второй формуле стоит имя того *элемента,* переходы с которого определяет следующая за двоеточием конструкция. В последней перечисляются возможные переключения. Они упорядочиваются по старшинству *событий* (см. подразд. 2.4), причем завершается перечень именем *элемента,* при котором не стоит никакого *события.* Это значит, что если не реализуется ни одно из вошедших в *СписокПереходов событий,* то произойдет

переключение на *элемент с* указанным именем.

Чтобы полностью описать синтаксис параграфа *автоматной функции,* осталось уточнить формат, в котором представляется *ЗаписьСобытия.* Это - дизъюнктивная нормальная форма от булевых переменных, фиксирующих приход *сигналов* на внутренний входной разъем *прибора.* Принятие ею значения **TRUE** есть признак реализации *события.* В качестве имен булевых индикаторов *сигналов* используются номера соответствующих контактов внутреннего входного разъема. Знаком дизъюнкции служит "+", конъюнкции - "*" а отрицания - "^". Итак,

ЗаписьСобытия **::=** *ФрагментСобытия* { "+" *ФрагментСобытия* }.

ФрагментСобытия. **::=** *КонъюктивныйФрагмент* | *Терм.*

КонъюктивныйФрагмент **::=** "(" *Терм* { "*" *Терм* } ")".

Терм **::=** ["^"] *ЦелоеБезЗнака* [".." *ЦелоеБезЗнака*].

Пример записи события:

1 + ^2 + 4..6 + (7..9 * 11 * ^13).

Смысл конструкций вида *n1..n2,* допускаемых четвертой формулой, зависит от контекста: если такая конструкция стоит в конъюнктивном фрагменте, то ее следует понимать как конъюнкцию всех *сигналов* с номерами от *n1* до *n2* включительно, иначе - как их дизъюнкцию. Символ отрицания перед такой конструкцией не меняет ее конъюнктивного или дизъюнктивного смысла и должен трактоваться как эквивалент отрицания каждого из соответствующих *сигналов.* Так, используя знак равенства как символ эквивалентности, можно написать, что

+ ^*2..4* = +^*2* +^3 +^4 и * ^2..4 = *^2 *^3 *^4.

Для иллюстрации рассмотренных правил формирования *спецификаций приборов* предлагается следующий простой пример:

```
МОЗГ = DEVICE

    FASTELS : СменаПоведения;

    SLOWELS : Ожидание, Управление;

    CONVELS : ВыборПоведения;

   ROOTELM : Ожидание;
END;

CONNECTIONS =

   i=0..1 : [i] INT = [i];

            [2] INT = [2] INT + ИНФОРМАЦИЯ;

                [0] = [0] INT + ПРИКАЗ;

                 [1] = [1] INT;
END;

SWITCHES =
          Ожидание : СменаПоведения, 0 + 1,
                  ВыборПоведения;

СменаПоведения : Управление;

ВыборПоведения : Управление;
Управление  :  СменаПоведения, 1,
                Ожидание;
END;

PHASETYPE = RECORD

   Состояние : (ясный, усталый);
END;
```

5. ОСНОВНЫЕ ИНСТРУМЕНТАЛЬНЫЕ ПРОЦЕДУРЫ

Основные (обслуживающие первоочередные потребности имитации) процедуры *MISS* распадаются на три группы. Первая нужна для организации модулей приборов, вторая - для работы с базой данных моделей, третья - для анимализации имитационных экспериментов. В таком порядке мы их и рассмотрим- Кроме того, в четвертом подразделе описаны возможности останова имитации без выхода из системы. Записи заголовков процедур даются в нотации *МОДУДЫ-2/JPI*. При вызове в программы на "С" все процедуры берутся из системной библиотеки, а при работе в *МОДУЛЕ-2* они импортируются из библиотечных модулей *Monitor* (процедуры подразд., кроме 5.3) и *Screen* (подразд. 5.3).

5.1 ПРОЦЕДУРЫ ОРГАНИЗАЦИИ МОДУЛЕЙ ПРИБОРОВ

Чтобы диспетчер *MISS* мог организовать прогонку модели, должно быть установлено соответствие между именами элементов из спецификаций приборов и их процедурами из модулей приборов. Это соответствие фиксируется вызовами процедуры

LinkElement(ИмяЭлемента **: ARRAY OF CHAR;**
ОсновнойАлгоритм, Таймер **: PROC**)

в телах модулей приборов (для программ на языке "С" - в функциях с именем *main),* причем *НИКАКИХ ДРУГИХ ИНСТРУМЕНТАЛЬНЫХ ПРО-*

ЦЕДУР, ОПИСАННЫХ В ЭТОМ РАЗДЕЛЕ, ВЫЗЫВАТЬ В ТЕЛАХ ЭТИХ ИЛИ КАКИХ-ЛИБО ЛИНКУЕМЫХ С НИМИ МОДУЛЕЙ НЕЛЬЗЯ.

Обращение к *LinkElement* привязывает имя *элемента* согласно *спецификации прибора* к соответствующим процедурам *основного алгоритма* и *таймера,* определенным в модуле *прибора.* Последние ставятся вторым и третьим параметрами вызова; первым параметром должна быть запись имени *элемента* в виде строковой константы. Если *элемент сосредоточенный,в* качестве фиктивного *таймера* указывается специальная "пустая" процедура

Dummy().

Два механизма диспетчеризации процессов, описанные в подразд. 2,4, поддерживаются двумя наборами инструментальных процедур. В первом их две; это - процедуры, обеспечивающие прием и посылку *сигналов* и *сообщений.* Они могут использоваться и в *основних алгоритмах* и в *таймерах элементов.*

Посылка *сигнала* осуществляется вызовом процедуры

PutOutSig(НомерВыхода **: CARDINAL;**
КлючСообщения : *LISTCODE*).

При этом в качестве первого параметра дается номер посылаемого выходного *сигнала,* а точнее, в терминах подразд. 4.2. - номер того контакта внутреннего выходного разъема *прибора,* через который посылается *сигнал.* Второй параметр имеет специфический тип ключа виртуального списка, определенный в инструментальном модуле *VirtvalMemory* (см. разд. 6). и нужен для передачи сообщений. Если *сигнал* пойдет без *сообщения,* на место этого параметра ставится константа *EmptySignal,* определенная в модуле *Monitor.* В противном случае, когда с *сигналом* надо передать непустой список однотипных записей (а только таким и может быть *сообщение),* в качестве второго входа процедуры *PutOutSig* ста-

вится ключ этого списка. Сам список должен быть сформирован процедурами, описываемыми в разд. 6, причем все начнется как раз с заведения указанного ключа. Заметим, что посылка *сообщения* корректна только тогда, когда ее возможность предусмотрена в *спецификации, прибора* и когда образующие *сообщение* записи имеют указанную в *спецификации,* структуру (см. подразд 4-2). Для приема *сигналов* и *сообщений* имеется процедура

GetInSig(НомерВхода **: CARDINAL, VAR** КлючПриема : *LISTCODE*).

При вызове ее первым параметром должен быть номер принимаемого входного *сигнала,* т.е. - в терминах подразд. 4.2 - номер того контакта внутреннего входного разъема *прибора,* на котором ожидается *сигнал.* Вторым параметром ставится идентификатор переменной, которая в результате вызова может получить либо значение константы *NonSignal* из модуля *Monitor,* означающее, что на данном *такте имитации сигнал* не пришел (т.е. на предыдущем *такте* не послан), либо - иное значение, что укажет на наличие *сигнала*. Если он может еще и сопровождаться *сообщением, GetInSig* вернет вторым параметром ключ. по которому *сообщение,* коль скоро оно действительно послано, удастся прочесть процедурами из модуля *VirtualMemory* (см. разд. 6). В отсутствие *сообщения* этот ключ укажет на пустой список.

Второй набор инструментальных процедур диспетчеризации *процессов* связан с поддерживаемым *MISS* механизмом планирования модельного времени. Это время трактуется абстрактно - как безразмерная скалярная величина типа **LONGREAL.** Его привязка к какому-то физическому или календарному времени остается на усмотрение пользователя. Система же привязывает его только к "моменту рождения модели": начальное значение модельного времени при первом запуске модели на счет будет нулевым.

Для считывания показаний системных часов в имитирующие программы заведена функция

GetTime() **: LONGREAL.**

Будучи вызванной в каком-нибудь *таймере,* она вернет системное время начала текущего *такта имитации,* а при обращении из процедуры *основного алгоритма* - время окончания этого *такта.* Продолжительность *такта* выясняется через функцию

TimeStep() **: LONGREAL.**

К ней, в отличие от предыдущей, имеет смысл обращаться лишь в *основных процедурах элементов,* так как только там она возвращает истинное значение искомой продолжительности.

Сам ход модельного времени целиком определяется *таймерами.* в результате планирования в них моментов прерываний (см. подразд. 2.4). На то есть две пары инструментальных процедур. В каждой паре одна процедура планирует просто прерывание - момент следующего вызова *таймера,* а вторая - момент завершения *элемента.* Первая пара процедур такова:

BreakAtTime(ВремяПрерывания : **LONGREAL**).
StopAtTime(ВремяЗавершения : **LONGREAL**).

Обе они блокируют *таймер* и заказывают будущую точку *синхронизации* на момент времени, указанный параметром обращения. Первая фиксирует "ВремяПрерывания" как момент следующего вызова *таймера,* а вторая указывает "ВремяЗавершения" в качестве момента окончания *элемента* (вообще исключая последующие вызовы *таймера* в данной реализации *элемента).*
Процедуры второй пары

BreakNotLater(ПредельноеВремяПрерывания **: LONGREAL**),

StopNotLater(ПредельноеВремяЗавершения **: LONGREAL)**

не блокируют *таймеров* и не задают моментов прерываний однозначно. Обращение к первой есть требование вызвать *таймер* в следующей *точке синхронизации* откуда бы она не определилась, и если никакой другой *таймер* не закажет прерывания на момент меньший, чем "ПредельноеВремяПрерывания", то установить прерывание на это время. Результатом применения второй процедуры тоже может оказаться вызов *таймера* в следующей *точке синхронизации,* и это произойдет, если какой-то другой *таймер* установит прерывание на момент, опережающий "ПредельноеВремяЗавершения"; в противном случае результатом будет установка очередной *точки синхронизации.* на "ПредельноеВремяЗавершения". которое при этом станет моментом окончания *элемента.*

При вызове любой из четырех описанных процедур в качестве значения параметра можно взять величину, которая меньше текущего модельного времени. Результат будет в точности тем же, как если бы значение параметра было равно этому времени.

Кроме представленных есть еще четыре процедуры управления *процессами.* Все они служат для организации вызовов *таймеров,* хотя ни одна из них не планирует моментов прерываний. Чтобы была возможность эффективно увязывать времена завершения *элементов* с *сигналами,* заведены процедуры

BreakAfterSig(НомерВхода **: CARDINAL**),
BreakAfterEvent ().

Обе блокируют *таймер.* После первой следующий вызов *таймера* произойдет на том *такте имитации,* на котором *прибор* получит *сигнал* указанного номера. Вторая обеспечит вызов *таймера* на *такте,* где *прибору* придет хоть один *сигнал. В* связи с направленностью этих

процедур нелишне предостеречь: те *сигналы,* по которым будет вызван *таймер,* исчезнут в конце *такта* вызова и поэтому надо отличать их от *сигналов,* которые попадут в *автоматную функцию,* если завершить *элемент* на этом *такте* выполнением в *таймере* оператора *StopAtTime(GetTime()).* Наконец, есть процедура

Continue(),

обращение к которой означает запрос на следующий вызов *таймера* на следующем *такте,* и блокирующая *таймер* "навечно" процедура

NonStop().

В заключение подчеркнем, что все восемь представленных процедур планирования вызовов *таймеров* и завершений *элементов* предназначены для использования в *таймерах несосредоточенных элементов.* Их исполнение в процедурах *основных алгоритмов* чревато путаницей и не рекомендуется. Лишь для *распределенных элементов* возможно корректное управление *таймерами* из *основных алгоритмов.* И последнее: обращение при одном исполнении *таймера* к нескольким процедурам этой группы эквивалентно обращению к одной из них, а именно - к той, которая будет вызвана последней.

5.2 ПРОЦЕДУРЫ РАБОТЫ С БАЗОЙ ДАННЫХ

Здесь будут описаны процедуры доступа к содержимому базы данных модели из имитирующих программ- В этой базе хранятся, во-первых, *фазовые переменные* и *константы* всевозможных компонент модели, а во-вторых, - специальные данные вроде текущих *статусов* (см. подразд. 2-5) *групп, объектов* и *приборов.* Подразумевается расширенное толкование *фазовых переменных,* когда к ним относятся и *сигналы* с *сообщениями.* Однако, в связи с тем, что процедуры приема и посылки *сигналов* и *сообщений,* уже были рассмотрены в

предыдущем подразделе, ниже речь пойдет только о тех *фазовых переменных,* которые ассоциируются со введенным в подразд. 2.2 понятием *базовых записей.*

MISS обеспечивает доступ к любым *фазовым переменным* и *константам* из любой имитирующей программы. Принципиальная схема общения с их *базовыми записями* такова:

[копирование в программу-->] обработка [--> запись копии в базу].

Квадратные скобки говорят о том, что ни первая, ни последняя операции не обязательны и это понятно. Ясно также, что именно они составят предмет дальнейшего обсуждения. По поводу обработки можно лишь отметить, что она фактически является основным содержанием вычислительной части модулей *приборов* и согласно технологии *MISS* должна программироваться стандартными средствами используемых языков.

Проще всего организовать такой обмен с базой, когда в процедуре какого-то модуля *прибора* нужно связаться с данными, относящимися к экземпляру *прибора,* обслуживаемому текущим вызовом процедуры, или к его классу, а также - к содержащим *прибор* экземплярам *объекта* и *группы,* либо к их классам. В подобных случаях уместно говорить о работе со *своими* для модуля *прибором, объектом* и *группой.* На то есть четверка процедур:

CopyMyPhase (ЧьюФазуСчитать : *WHO;*
VAR КудаСчитатьФазу : **ARRAY OF BYTE**),

CopyMyConst (ЧьиКонстантыСчитать : *WHO;*
VAR КудаСчитатьКонстанты : **ARRAY OF BYTE**),

SaveMyPhase(ЧьюФазуЗаписать : *WHO;*
VAR ОткудаЗаписатьФазу **: ARRAY OF BYTE),**

SaveMyConst (ЧьиКонстантыЗаписать : *WHO;*
VAR ОткудаЗаписатьКонстанты **: ARRAY OF BYTE).**

Первая пара занимается считыванием из базы, вторая - записью в базу. Какие данные будут считываться или записываться - ясно из имен параметров. Имена первых параметров, кроме того, подсказывают, как задается принадлежность данных при вызовах этих процедур. В каждом случае первый параметр может быть одним из трех идентификаторов перечисления

WHO = (group, object, device),

отмечающих *группы, объекты* и *приборы* соответственно. Само перечисление определено в модуле *Monitor*.

Использование в модуле *прибора* любой процедуры рассматриваемой четверки требует заведения в нем буфера для копирования глобальной или локальной переменной, аналогичной копируемой *базовой записи* по структуре. Этот буфер и ставится вторым параметром вызова процедуры. Требование соответствия типа буфера структуре *базовой записи MISS* означает следующее:

а) для программы в *МОДУЛЕ-2/SDS* первое должно получаться из второго заменой каждого специального типа данных *MISS* на отвечающий ему тип данных из модулей *Monitor* и *VirtualMemory;*

б) для программы в *Turbo-C* аналогия та же, но с учетом отличия синтаксиса определений записей в "C" от *MISS;*

в) для программы в *МОДУЛЕ-2/JPI* кроме замен одних специальных типов другими может потребоваться ввести в буфере избыточные байтовые поля: в связи с выполняемым *MISS* и не выполняемым компилятором *МОДУЛЫ-2/JPI* выравниванием (см. подразд. 4.1) поля типа RECORD, а также строки массивов и группы смежных байтовых полей должны здесь иметь четные длины.

Копировать можно не всю *базовую запись,* а лишь некую ее головную часть. Тогда в определении буфера должны быть представлены в правильном порядке не все, а только несколько первых

полей *базовой* записи.
Таблица соответствий специальных типов данных *MISS* типам *МОДУДЫ-2* и "С" такова:

Тип в языке MISS	Тип в С и МОДУЛЕ-2	Экспортирующий модуль МОДУЛЫ-2	Место определения в С
ADDRESS OF	DATACODE	VirtualMemory	miss.h
LIST OF	LISTCODE	VirtualMemory	miss.h
BLOCKS OF	ARRCODE	Arrays	miss.h
PICTURE	PICTURE	Screen	miss.h
BACKGRND	BACKGRND	Screen	miss.h
TEXT	ARRCODE	Arrays	miss.h

Длина поля любого из перечисленных типов - 4 байта. Типу *NAME* языка *MISS* в языках *МОДУЛА-2* и "С" отвечают

```
TYPE                              tupedef
NAME = RECORD                         struct {
         Text   : DATACODE;                 DATACODE  Text;
         Length : CARDINAL;                 unsigned Lengtn;
       END;                                                  } NAME;
```

Работа с переменными типов *TEXT* и *NAME* разрешена только через текстовой редактор *MISS,* а графические переменные типов *PICTURE* и *BACKGRND* могут использоваться имитирующими программами исключительно в роли параметров процедур визуализации (см. следующий подраздел). Таким образом, из всех специальных переменных *MISS* для программной обработки по существу открыты лишь переменные первых трех типов таблицы. Доступ к стоящим за ними структурам данных осуществляется через процедуры поддержки виртуальной памяти (см. следующий раздел) и требует введения дополнительных буферов, соответствующих определениям этих структур в спецификациях.
Пример соответствия определений типов *фазовых переменных в*

спецификации и в программе на *МОДУЛЕ-2/JPI*

```
PHASETYPE = RECORD                    Phase = RECORD
  Список : LIST OF                        List : LISTCODE;
             Время : REAL;
             Очередь : CARDINAL;
           END;

  Запись : RECORD                     Record : RECORD
             bool : BOOLEAN;              bool, dummy : BOOLEAN;
             X, Y : REAL;                 X, Y : LONGREAL;
           END;                           END;
  Перечень : BLOCKS OF                Set : ARRCODE;
             Номер : CARDINAL;
           END;
END;

END;                                  END;
```

В этом примере в записи Record введено дополнительное поле dummy а в его отсутствие буфер был бы некорректен по причине выравнивания полей в *MISS.*

Завершим разбор первой четверки процедур обменов с базой данных строгим определением результатов их работы. Процедура *CopyMyPhase,* будучи вызванной в неком модуле *прибора,* полностью или частично скопирует в указанный вторым параметром буфер *базовую запись фазовых переменных:*

а) представляемого модулем прибора,
если первый параметр = *device;*
б) содержащего этот *прибор объекта,*
если первый параметр = object;
в) *группы, в* которую входит содержащий этот *прибор объект,*
если первый параметр = *group.*

Поскольку модуль относится не к единичному *прибору,* а к их классу, то, четкости ради, еще раз отметим, что имеется ввиду тот конкретный *прибор,* который обсчитывается в момент рассматриваемого вызова- Процедура *SaveMyPhase* отличается от *CopyMyPhase* только направлением копирования - она пишет не из базы в буфер,

а из буфера в базу. Ей аналогична процедура *SaveMyConst,* копирующая *константы,* и**,** соответственно, - не экземпляров *прибора, объекта* и группы, а их классов. Такова же аналогия между процедурами *CopyMyConst* и *CopyMyPhase.* Подчеркнем, что вне модулей *приборов* описанные процедуры вызывать нельзя.

Доступ к произвольным (не *своим)* наборам *фазовых переленных* и *констант* требует точного указания их принадлежности. Для базовой записи *констант* это - код класса компонент, к которому она приписана, а для *фазовых переменных* - код класса компоненты-хозяина и ее номер в *СКВОЗНОЙ* индексации модели. Формально коды классов являются величинами специального типа *TYPEPATH,* экспортируемого модулем *Monitor.* Они "вычисляются" по именам. под которыми классы описаны в *спецификации,* причем дополнительно требуется называть тип компонент класса. Искомые коды возвращает процедура

AbsTypePath(ТипКомпонентКласса : *WHO;* ИмяКомпонентКласса :
ARRAY OF CHAR; VAR КодКласса : *TYPEPATH*) -

Смысл параметров ясен из контекста-

Для копирования произвольных *фазовых переменных* и *констант* из базы данных в программы и обратно есть процедуры

CopyAbsPhase(КодКлассаКомпоненты : *TYPEPATH;*
СквозноИндексКомпоненты **: CARDINAL;**
VAR КудаСчитатьФазу **: ARRAY OF BYTE),**

GopyAbsConst (КодКлассаКомпонент : *TYPEPATH;*
VAR КудаСчитатьКонстанты : ARRAY OF BYTE).

SaveAbsPhase(КодКлассаКомпоненты : *TYPEPATH;*
СквозноИндексКомпоненты **: CARDINAL;**
VAR ОткудаЗаписатьФазу **: ARRAY OF BYTE).**

SaveAbsConst (КодКлассаКомпонент : *TYPEPATH;*
VAR ОткудаЗаписатьКонстанты **: ARRAY OF BYTE).**

Они аналогичны процедурам доступа к данным своих *приборов, объектов* и *групп.* Соответствие очевидно из сопоставления имен самих процедур и их последних параметров, имеющих здесь в точности тот же смысл, что прежде. Смысл прочих параметров ясен из предыдущего абзаца. Вызывать эти процедуры можно и в модулях *приборов* и в *сервисном модуле.*

При моделировании часто возникает потребность циклической обработки данных по всем компонентам одного класса. Именно на такие случаи прежде всего рассчитаны процедуры *CopyAbsPhase* и *SaveAbsFhase;* отсюда и способ указания в них компоненты - через ее номер в, вообще говоря, скрытой от пользователя *СКВОЗНОЙ* индексации модели- Диапазон значений номеров компонент любого класса начинается с нуля. Правую границу соответствующего диапазона, если она не ясна априори, всегда можно получить обращением к функции

NumOfExemps (КодКлассаКомпонент : *TYFEPATH*) **: CARDINAL.**

Она возвратит число компонент в том классе, на который укажет единственный параметра ее вызова -

Циклический доступ, конечно, не удовлетворяет всех мыслимых потребностей общения имитирующих программ с "чужими" *фазовыми переменными.* Поэтому заведены три процедуры, позволяющие выяснить сквозной номер компоненты модели по ее "локальному прикреплению". Вызывать их можно только в модулях *приборов.*

Чтобы при вызове процедуры некого модуля *прибора* выяснить сквозной индекс какой-то компоненты из состава *группы,* которой принадлежит обслуживаемый вызовом *прибор* (точнее - непосредственно принадлежит *объект,* к которому относится этот *прибор),* надо обратиться к функции

OneOfMyGroupAbsExemp(КодКлассаНужнойКомпоненты : *TYPEPATH;*
ЛокальныйНомерВГруппе **: CARDINAL) : CARDINAL.**

При этом заранее должен быть определен код класса той компоненты модели, чей индекс нужно узнать. Этот код послужит первым входом функции. Вторым будет номер компоненты во внутренней индексации *группы.* Возвращаемое значение - искомый сквозной индекс. Аналогичным образом индекс "соседнего *прибора"* своего *объекта* выясняется через функцию

OneOfMyObjExemp(КодКлассаНужногоПрибора : *TYPEPATH;*
ЛокальныйНомерВОбъекте : **CARDINAL**) : **CARDINAL**.

Наконец, собственные адресные атрибуты *прибора* или адресные атрибуты содержащих его *объекта* и *группы* можно узнать с помощью процедуры

MyPath(ТипНужнойКомпоненты : *WHO;*
VAR ИскомыйКодКласса : *TYPEPATH;)* и функции

MyAbsExemp(ТипНужнойКомпоненты : *WHO*) : CARDINAL.

Входные параметры показывают, чьи атрибуты надо взять - *прибора,* при обсчете которого был вызов *MyPath* или *MyAbsExemp,* включающего этот *прибор объекта* или содержащей этот *объект группы.* Выходы - код класса и номер в сквозной индексации.

Из всевозможных специальных данных по компонентам модели, хранящихся в базе, для программной обработки открыты только *статусы* (см. подразд. 2.5). Процедура

KillMyself(ТипНужнойКомпоненты : *WHO*),

будучи вызванной из какого-то модуля *прибора* (а только в модулях *приборов* ее и можно вызывать), немедленно переведет в пассивный статус сам *прибор,* содержащий его *объект* или включающую этот *объект группу,* в зависимости от того, какой идентификатор из тройки *WHO* будет стоять фактическим параметром вызова. При этом пассивизация *объекта* означает, что пассивными станут все

его *приборы,* а пассивизация *группы* означает пассивизацию всех ее *объектов* со всеми их *приборами* и всех ее *подгрупп,* со всеми их *обьектами* всеми их *приборами* и так далее...

Чтобы изменить статус произвольной (не своей) компоненты, надо обратиться к процедуре

AbsChangeStatus(КодКлассаКомпоненты : *TYPEPATH;*

СквознойИндексКомпоненты **: CARDINAL;**
НовыйСтатус : *STATUS*).

Значение третьего параметра вызова берется из перечисления
STATUS = (*active, passive*),

определенного в модуле *Monitor.* Подстановка *active* означает требование активизации компоненты, *passive* - пассивизации. Изменения статуса *объекта* или *группы* и здесь будут "развернуты вниз" как для *KillMyself,* причем вновь активизированные *приборы* начнут функционировать со следующего *такта* и - с *элементов,* выделенных в их описателях как корневые. Перевод компонент в пассивный статус выполняется немедленно.

Выяснить статус произвольной компоненты позволяет функция

AbsStatus(КодКлассаКомпоненты : *TYPEPATH*
СквознойИндексКомпоненты **: CARDINAL**) **:** *STATUS.*

Все предыдущие процедуры предназначены для программной обработки базы данных модели. Возможна и диалоговая работа с ней во время имитации. Ее обеспечивает, в частности, процедура

DataShow(),

она передает управление диалоговому редактору данных *MISS.* С помощью последнего можно менять вручную любые, в том числе специальные, данные из базы по любым компонентам модели (см. разд. 9). По выходе из редактора имитация будет продолжена с оператора, который стоит за *DataShow,* причем автоматически восстано-

вится экран- Аналогичный доступ к данным возможен и через процедуру *WindowsProc* (см. разд. 8).

В заключение подраздела нелишне особо подчеркнуть, что ни одну из описанных в нем процедур нельзя вызывать ни в самих телах имитирующих модулей, ни в вызываемых там процедурах.

5.3 ПРОЦЕДУРЫ АНИМАЛИЗАЦИИ

Подсистема графики *MISS* задумывалась прежде всего в качестве средства анимализации программ, имитирующих пространственные перемещения реальных объектов. Рабочая графическая мода *MISS* - 640*350 пикселов, 16 цветов. Для показа "мультфильма" хода имитации можно использовать произвольную (и допускающую варьирование в процессе счета) часть EGA-дисплея с параметрами, удовлетворяющими неравенству

(ШиринаВПикселах / 8) * ВысотаВПикселах <= 18000,

причем отступы (в пикселах) используемой части экрана от его левой и правой границ должны быть кратны 8. Это неравенство позволяет брать под мультфильм до 9/14 экранной площади, а остаток можно использовать для буквенно-цифрового ввода/вывода.

В соответствии со своим назначением графическая подсистема *MISS* обеспечивает возможности:

а) заводить *пиктограммы,* которые будут представлять модельные *обьекты* на дисплее;

б) заводить *фоновые картинки,* фиксируя для них *сектора отображения* - выделяемые под мультипликацию части экрана дисплея;

в) выбирать экраны, т.е. комбинировать *фоновые картинки* и *сектора обзора* - прямоугольники в плоскости модельных перемещений, отождествляемые с *секторами отображения;*

г) во время счета готовить и показывать на дисплее *кадры* мультфильма; *кадр* - некий фон с нарисованными по нему пиктограммами *объектов,* размещенными в соответствии с их модельными координатами и текущим масштабом *экрана* (соотношением параметров текущих *секторов обзора и отображения).* Первые два пункта реализуются с помощью специальных данных *MISS* типа *PICTURE* и *BACKGRND* и графического редактора *MISS,* описываемого в разд. 9; *фоновые картинки* можно рисовать и представленными в разд. 7 инструментальными процедурами. Здесь же речь пойдет, в основном, о процедурах выбора *экранов,* формирования *кадров* (из готовых картинок) и их вывода.

Типовая схема организации мультфильма в *MISS* такова

{ установка *экрана*
{ { "раскладывание" *пиктограмм* по *фоновой картинке* }
вывод сформированного *кадра* по запросу в конце *такта* } }.

Самые внешние фигурные скобки означают цикл по группам *тактов имитации* и охватывают действия, выполняемые на каждом шаге этого цикла. Видно, что шаги отделяются друг от друга операцией выбора (точнее было бы сказать - смены) *экрана.* Пара следующих скобок относится к циклу по (не обязательно последовательным) *тактам имитации.* Завершающей операцией шага этого цикла является вывод на экран сформированного *кадра.* Цикл формирования *кадра* показан в самых внутренних скобках, и в рассматриваемой схеме только его предлагается выполнять в модулях *приборов;* прочее удобно делать в *сервисном модуле* (см. разд. 8).

Для установки *экранов* есть несколько процедур. Их состав определен тем, что *MISS* ведет список, состоящий из пар вида *{сектор* обзора, ключ *фоновой картинки}.* Этот список заводится автоматически при каждом запуске счета в *MISS* и в него сразу включается *экран* с *сектором обзора* 0-639 по горизонтали и 0-224

по вертикали и с ключом "пустого" фона; параметры сектора *отображения* берутся такие же как у *сектора обзора.* Во время счета всегда определен текущий *экран, с* которым и работают процедуры формирования *кадров.*

Описание средств установки экранов начнем с процедуры

SetBgr(КлючСчитыванияФона : *BACKGRND* ;
СменаВсегоЭкрана : **BOOLEAN**).

Она фиксирует в текущем *экране* тот фон, что указан первым параметром ее вызова, и в зависимости от значения второго параметра либо целиком заменяет картинку на дисплее на установленный фон, либо "перекрашивает" лишь связанный с ним *сектор отображения.* Фон может быть цветным или монохромным; во втором случае он воспроизводится в текущих цветах рисования, задаваемых процедурой *SetPaintColora* (см .разд. 7).

Сектор обзора текущего *экрана* устанавливается процедурой

SetScale(ЛеваяГраницаОбзора, ПраваяГраницаОбзора,
НижняяГраницаОбзора, ВерхняяГраницаОбзора : **REAL**),

смысл параметров которой ясен из контекста.

Работа со списком *экранов* как с таковым выполняется четырьмя процедурами- Он наращивается и сокращается с помощью

AddScreen() и
DeleteScreen(СменаВсегоЭкрана : **BOOLEAN**) : **BOOLEAN**

соответственно- *AddScreen* вставляет в список за текущим *экраном* новый, с теми же параметрами, и после этого текущим становится он. Результаты выполнения *DeleteScreen* зависят от того, стоит ли текущий *экран* первым в списке. Если стоит, то возвращается **FALSE** и ничего не происходит. В противном случае текущий экран (но не связанная с ним *фоновая картинка* !) будет уничтожен, а

значением функции будет **TRUE.** При этом

а) новым текущим станет либо следующий экран, если он есть, либо - первый, если уничтоженный *экран* был в списке последним;

б) на дисплей будет выведена *фоновая картинка* нового экрана - целиком, если на вход *DeleteScreen* подать **TRUE,** и только в границах нового *сектора отображения,* если подать **FALSE.**

Переходы по списку *экранов* выполняются двумя процедурами:

NextScreen(СменаВсегоЭкрана **: BOOLEAN**) и
PrevScreen(СменаВсегоЭкрана **: BOOLEAN).**

Первая означает шаг "к хвосту" а вторая - "к голове" списка. Если при вызове *NextScreen* текущим является экран, замыкающий список, то результатом будет переход на головной *экран.* Аналогично. если выполнить *PrevScreen,* когда текущий *экран* - первый в списке, новым текущим *экраном* станет замыкающий. Для обеих процедур смысл параметра тот же, что для *DeleteScreen.*

Программный выбор *экранов* возможен только с помощью шести описанных процедур- Есть и диалоговые возможности. Они реализуются через экспортируемую модулем *Monitor* процедуру

WindowsProc().

К *WindowsProc* можно привязать некую определяемую в *сервисном модуле* процедуру коррекции - процедуру без параметров для автоматической подгонки непредсказуемых результатов диалогового назначения текущего *сектора обзора* в *WindowsProc* к заранее очерчиваемому множеству вариантов: привязанная процедура будет вызываться после каждого изменения *сектора обзора* во время диалога. Привязка осуществляется вызовом определенной в модуле *Monitor* процедуры

DefineCorrection(ПроцедураКоррекции **: PROC**).

Для программной работы с *экранами* могут оказаться полезными справочные процедуры

GetScale(**VAR** ЛеваяГраницаОбзора, ПраваяГраницаОбзора,
НижняяГраницаОбзора, ВерхняяГраницаОбзора **: REAL**), и

GetScreen(**VAR** КлючТекущегоФона : *BACKGRND ;*

VAR ЛеваяГраницаОтображения, ПраваяГраницаОтображения.
ВерхняяГраницаОтображения, НижняяГраницаОтображения :
CARDINAL),

возвращающие атрибуты текущего *экрана.*

Следующие пять процедур ориентированы на программную инициализацию графических элементов "кинопроизводства". Для записи в базу данных модели построенных программно *фоновых картинок* (с тем, чтобы позже использовать их в *экранах)* заведена процедура

SaveBgr(**VAR** КлючЗаписиФона : *BACKGRND* ;
СохранитьОдноцветным : **BOOLEAN** ;
ЛеваяГраница, ПраваяГраница, ВерхняяГраница, НижняяГраница :
CARDINAL).

Как бы ни было получено изображение на дисплее к моменту вызова *SaveBgr:*

а) оно запоминается в виртуальной памяти в качестве *фоновой картинки,* в цвете или нет - в зависимости от второго параметра вызова: если взять **TRUE,** то сохранится лишь "контур" фона - позиции пикселов, имеющих цвет, отличный от текущего фонового (см. разд. 7);

б) с ним связывается *сектор отображения,* заданный четырьмя последними параметрами вызова; это - соответствующие границы. выраженные в горизонтальных и вертикальных номерах крайних пикселов. причем отступы от левого и правого краев экрана дисплея должны быть кратны восьми;

г) первым параметром возвращается ключ к сохраненному фону.

Процедуры

ImportBgr(**VAR** ИмяФайлаНаДиске **: ARRAY OF CHAR ;**

VAR КлючСымпортированногоФона : *BACKGRND*) и

ImportPict(**VAR** ИмяФайлаНаДиске **: ARRAY OF CHAR ;**

VAR КлючСымпортированнойПиктограммы : *PICTURE*)

импортируют из DOS-каталога в виртуальную память *MISS* фоны и *пиктограммы,* хранящиеся в файлах с указываемыми первыми параметрами именами и возвращают вторыми параметрами коды соответствующих графических объектов. Файлы-источники могут формироваться только графическим редактором *MISS.*

Процедуры

DestroyBgr(КодУничтожаемогоФона : *BACKGRND*) и
DestroyPict(КодУничтожаемойПиктограммы : *PICTURE*)

уничтожают в виртуальной памяти *MISS фоновые* картинки и *пиктограммы.*

В завершение подраздела рассмотрим процедуры формирования и вывода *кадров.* Вывод *пиктограммы* на *фоновую картинку* текущего *экрана* выполняет процедура

DrawObJect(ГоризонтальнаяКоординатаОбъекта.
ВертикальнаяКоординатаОбъекта **: REAL;**
КлючПиктограммы : *PICTURE* ;
ВывестиМонохромно **: BOOLEAN).**

Смысл первых трех параметров очевиден- Наличие четвертого параметра говорит о том, что даже цветную *пиктограмму* можно вывести монохромно, т.е. - в текущих цветах рисования, устанавливаемых процедурой *SetPaintColors* (см. разд. 7). Относительно результатов вывода нужно сделать следующие уточнения:

а) полностью или частично попадет в *кадр* лишь та *пиктограмма,* поместив центр которой в точку фона, определяемую по указанным координатам, мы получим непустое пересечение с *сектором отображения:* центры *пиктограмм* фиксируются при их формировании в графическом редакторе и во время вывода помещаются в точки секторов *отображения,* отождествляемые с теми точками *секторов обзора,* координаты которых подаются на вход *DrawObJect;*

0) как "провзаимодействуют" *пиктограммы* с *фоновой картинкой* и друг с другом при наложении (будут ли перекрываться, просвечиваться и т.д.), зависит от текущей моды рисования, устанавливаемой специальной процедурой (см. подразд. 7.4).

Показ на дисплее сформированного вызовами *DrawObJect кадра* мультипликации осуществляется вызовом процедуры

ShowStill().

5.4 ВОЗМОЖНОСТИ ПРЕРЫВАНИЯ ИМИТАЦИИ

При прогонке модели может возникнуть желание прервать счет, чтобы проанализировать текущие значения каких-то данных, имеющихся в ее базе. *MISS* предлагает на этот случай два способа:

а) диалоговый - по сигналу с клавиатуры,

б) программный - вызовом одной из предлагаемых ниже процедур. Еще раз подчеркнем, что в обоих случаях подразумевается не акция, аналогичная по последствиям выключению машины, но аккуратно оформленный выход из имитации в диалоговый монитор с сохранением имеющихся на момент останова данных. Диалоговый выход всегда возможен нажатием "Ctrl+c", что приводит к немедленному завершению счета и передаче управления на корневое меню работы с готовыми моделями; продолжить счет при этом, вообще говоря, нельзя. Возможность диалоговых прерываний счета с последующими

продолжениями реализуется через сервисный *модуль,* подробно рассматриваемый в разд. 8.

Программные остановы обеспечиваются процедурой

Exit()

из модуля *Monitor,* вспомогательной процедурой

Error (СообщениеОбОшибке **: ARRAY OF CHAR**)

из модуля *Errors* и стандартной процедурой *МОДУДЫ-2*

HALT().

Обращение к *Exit* означает требование прекратить вычисления по окончании текущего *такта* имитации. Как только он закончится. управление перейдет либо к определяемой в *сервисном модуле* процедуре завершения, если она там есть (см. разд. 8), а потом - к диалоговому монитору *MISS,* либо - сразу к диалоговому монитору; в данном случае открыта возможность продолжить счет с точки, где он был прерван-

Две другие процедуры срабатывают моментально, и если это произойдет "в середине" такта, продолжить счет так, как будто он не прерывался, не удастся. По *HALT* управление сразу передается в основное меню диалогового монитора, а по *Error* его сначала получит программа, которая:

а) будет подавать звуковой сигнал останова до первой реакции пользователя;

б) по первой посылке с клавиатуры выключит звук и будет ждать второй посылки;

в) по второй посылке выдаст на экран строку, которую получила в качестве параметра вызова, и дополнительно сообщит, где произошел вызов;

г) по "Enter" или "Esc" передаст управление основному меню диалогового монитора.

6. СИСТЕМА ВИРТУАЛИЗАЦИИ ПАМЯТИ

Ниже описаны средства виртуализации памяти, послужившие элементной базой *MISS* и открытые для пользователя. Прежде всего надо обратить внимание на процедуры работы со *списками,* которые, в частности, нужны для посылки и приема *сообщений.* Кроме того, нередко оказываются полезными специальные средства работы с *виртуальными модулями.* Они помогают в ситуации, когда есть необходимость использования некоторых подпрограмм и переменных в нескольких модулях *приборов.* Знание остального материала раздела, вообще говоря, необязательно, но и небесполезно.

6.1 ОСНОВНЫЕ ПОНЯТИЯ

Центральные понятия системы виртуальной памяти таковы:

запись - непрерывный участок памяти длиной не более 32768 байтов;

список - цепочка из одинаковых записей длиной не более 32764 байтов каждая; доступ к этим записям осуществляется либо выборочно по номерам, либо поочередно с начала или с конца *списка;* можно и включить новую запись в *список* на любое место и уничтожить любую существующую запись; записи *списка* нумеруются с нуля;

массив - блокированная цепочка из одинаковых записей длиной не более 16382 байтов каждая; доступ к этим записям

осуществляется либо выборочно по номерам, либо поочередно с начала или с конца *массива;* можно добавить новую запись в конец *массива* и уничтожить запись, стоящую в нем последней; записи *массива* объединяются в блоки, одинаковые по числу составляющих, каковых может быть от 2 до 32764 на один блок; максимальная суммарная длина блока равна 32764 байтам; применение *массивов* целесообразно при работе с короткими записями, поскольку уменьшает накладные расходы памяти по сравнению со списками по 4 байта на запись; записи *массива* нумеруются с нуля;
И *списки* и *массивы* могут быть пустыми, т.е. не содержать ни одной записи.

Длинная запись - непрерывный участок памяти любой длины в пределах доступной системе памяти;

виртуальный модуль - главная программа (модуль в смысле *МОДУЛЫ-2), подпрограммы* и данные которой, отмеченные с помощью специальных процедур *ExportProc* и *ExportData* могут быть доступны любой другой программе с помощью процедур *ImportPгос* и *Import Data;*

классы - независимые разделы виртуальной памяти с номерами от 0 до 8, в которых во время работы программ размещаются записи, *массивы, списки* и *виртуальные модули;* память *классов с* номерами от 2 до 7 может быть специальным образом сохранена на диске в файлах с расширениями соответственно от ".2" до ".7";
в *классе 7* хранятся *виртуальные модули MISS* и пользователю запрещается работать с ним; память *классов* 0, 1 и 8 - временная, не сохраняемая при кон-

сервации системы, память *класса* О сохраняется на весь сеанс работы с *MISS,* память *классов* 1 и 8 сохраняется на время прогонки модели; память класса 8 - это обычная динамическая память типа "Heap", размер которой определяется объемом свободной оперативной памяти; предельный размер каждого *класса* с номерами от 0 до 7 - 8 МБ;

Библиотеки, поддерживаемые *MISS,* содержат по два файла с именами вида <имя библиотеки> .2 и <имя библиотеки> .5; при вызове библиотеки в работу эти файлы копируются во второй и пятый *классы* виртуальной памяти соответственно. Когда работа завершена и выдана команда на сохранение библиотеки, *MISS* запишет в ее файлы новое содержимое *классов* 2 и 5. Второй *класс* отдается *спецификациям,* а в пятый заносятся собранные модели, готовые к запуску. Если по ходу имитации требуется заводить в сохраняемой памяти собственные динамические структуры данных, то лучше также помещать их в пятый *класс.* Для пользовательских не сохраняемых динамических данных рекомендуется первый *класс.* При использовании *класса 8,* надо помнить, что его чрезмерное разрастание может снизить эффективность работы виртуальной памяти. Не запрещено и использование *классов* 3, 4 и 6, но в случае обращения к ним, забота об их сохранении и восстановлении ложится на плечи пользователя.

коды - ключи для доступа к *записям, длинным записям, массивам, спискам и модулям;* они делятся на *коды записей* (скрытый тип *DATACODE), длинных записей* (скрытый тип *PARACODE), массивов* (скрытый тип *ARRCODE),* списков (скрытый тип *LISTCODE)* и *виртуальных модулей* (скрытый тип *MODCODE);*

Коды класса 8 есть просто адреса памяти, *коды* остальных *классов*

есть аналоги адресов памяти с той разницей, что в отличие от обычных адресов они сохраняемы: заново вызвав систему после консервации, по прежним *кодам* можно обращаться к тем же данным, что и до консервации. Имеется возможность перехода от *кодов записей* к адресам. Кроме того, есть средства получить для *записи массива, списка* или глобальной переменной *виртуального модуля* ее *код записи* и затем работать с ней, как с самостоятельной записью. По *коду виртуального модуля* находятся *коды записей* его глобальных переменных и адреса его глобальных процедур, сообщаемые системе вызовами в его теле (если он написан на "C", то под телом подразумевается функция с именем *main)* специальных процедур *ExportProc* и *Export Data.*

6.2 ПРИНЦИПЫ РАБОТЫ

Основная оперативная память машины распределяется системой следующим образом: в "нижнюю часть" памяти загружается резидент *MISS* (.EXE -файл), "верхняя часть" отводится под стек, оставшаяся память между резидентом и стеком делится на страницыпо 32К. Неполная страница отводится под "Кучу" (Heap), в дальнейшем Heap может быть увеличен на целое число страниц. Во время работы система постоянно поддерживает разбиение всех выделеных страниц оперативной памяти на три непересекающихся подмножества - на виртуальные, загруженные и фиксированные страницы. Любая фиксированная страница находится в памяти ниже любой загруженной, а любая загруженная - ниже любой виртуальной. Подмножес-

Распределение памяти

тво виртуальных страниц всегда непусто, подмножества загруженных и фиксированных страниц могут быть пустыми. Если на машине имеется или эмулируется LIM-память, ее буфер размером 64К также разбивается на две страницы, одна из которых относится к загруженным, а другая к виртуальным.

Виртуальные страницы предназначены для организации обменов при работе с *записями, списками,* и *массивами.* Чем больше виртуальных страниц, тем эффективнее работает система. На фиксированных страницах размещаются фиксированные (т.е., помещаемые в память надолго) *длинные записи* и (или) *виртуальные модули.* Загруженные страницы содержат один временно загружаемый *виртуальный модуль* или одну временно загруженную *длинную запись;* впрочем. из-за большого объема страницы одновременно могут оказаться загруженными и другие *модули* и *длинные записи.*

Классы памяти тоже имеют страничную (по 32К) организацию и хранятся в LIM-памяти, в дополнительной оперативной памяти (если таковые имеются) или на жестком диске. Когда при работе с *записями, списками* и *массивами* необходимого объекта не оказывается в оперативной памяти, на свое место хранения в соответствующем классе выгружается та виртуальная страница, что была загружена раньше других, а вместо нее загружается страница с нужным объектом.

При загрузке *длинной записи* или *виртуального модуля* используются страницы начиная со следующей после последней фиксированной и до предпоследней виртуальной включительно. Если их не хватит, загрузка завершится аварийно. В противном случае содержимое затребованных загруженных и (или) виртуальных страниц возвращается на места хранения в соответствующих *классах* и освобожденные тем самым страницы отдаются под загружаемый объект. Они становятся фиксированными или загруженными, в зависи-

мости от того, каким способом выполняется загрузка. При этом предыдущая картина распределения загруженных страниц запоминается и после будущей выгрузки объекта автоматически восстановится. Имеется специальная команда, по которой выгружается тот объект, что был загружен последним.

6.3 СОСТАВ МОДУЛЕЙ СИСТЕМЫ

Процедуры и функции, работающие с *записями* и *списками,* находятся в библиотечном модуле

VirtualMemory.

С динамической памятью "кучи" работают процедуры модуля

Storage.

Для работы с *массивами* заведен библиотечный модуль

Arrays.

Длинными записями ведает библиотечный модуль

LongDataManager.

Операции с *виртуальными модулями* осуществляются через модуль

Ovy Manager.

В дальнейшем принадлежность описываемых процедур одному из этих модулей указывается только в тех редких случаях, когда она не ясна из приведенной классификации -

6.4 СПРАВОЧНЫЕ ПРОЦЕДУРЫ

Описание инструментальных средств системы виртуальной памяти начнем с процедур, выдающих справки по заведенным *записям, спискам, массивам* и *виртуальным модулям.* Справку по составу

списка можно получить через функцию

LstNumOfRecs (КодСписка : *LISTCODE*) : **CARDINAL,**

которая возвращает число *записей списка.* Размер записи возвращает функция

LstRecLength(КодСписка : *LISTCODE*) **: CARDINAL.**

Аналогичные процедуры имеются и для *массивов:* процедура

ArrNumOfRecs(КодМассива : *ARRCODE;*
VAR ТекущееЧислоЗаписейМассива **: LONGINT).**

возвращает число записей массива, а функция

ArrRecLength(КодМассива : *ARRCODE*) **: CARDINAL**

возвращает длину записи.

Относительно любого объекта виртуальной памяти можно узнать также в каком *классе* он размещен. *Класс* сообщает функция

MemClass(КодДинамическогоОбъекта **: ADDRESS) : CARDINAL.**

В качестве значения единственного входа этой функции можно использовать *код записи, длинной записи, списка, массива* или *модуля,* место хранения которых хочется узнать; возвращаемое ею число - искомый номер *класса* памяти.

6.5 ПРОЦЕДУРЫ РАБОТЫ С ОДИНОЧНЫМИ ЗАПИСЯМИ

Этот подраздел начнем с процедур, используемых при обработке уже заведенных одиночных динамических записей. Их три и первой естественно представить наиболее часто употребляемую. Таковой

является процедура
DataToAdr (КодЗаписи : *DATACODE;* **VAR** АдресЗаписи **: ADDRESS**),

которая позволяет перейти от заданного первым параметром *кода* записи к ее адресу, возвращаемому через второй параметр. Присвоение этого адреса указателю соответствующего типа позволит работать с содержимым *записи* без предварительного копирования - так, как обычно работают с *записями* через их указатели. Надо только подчеркнуть, что правильность обращения к полям *записи* по полученному через *DataToAdr* указателю гарантируется лишь до тех пор, пока нет вызовов каких-либо других процедур системы. После любого такого вызова связь *записи с* указателем, вообще говоря, теряется (поскольку виртуальная страница, содержащая запись, может быть выгружена при этом из оперативной памяти). При использовании средств мультизадачности (см. подразд. 7.7) надо иметь в виду, что процедура работы с виртуальной памятью, вызванная в параллельном процессе, может нарушить связь указателя с адресом *записи.* Однако гарантируется, что после вызова процедуры *DataToAdr,* вызвавший ее процесс не будет прерван в течение 1/18 с., не считая времени на обмены с диском (если таковые произойдут), что достаточно для выполнения 1000 операторов языка высокого уровня. Поэтому при работе нескольких параллельных процессов, следует использовать результат процедуры *DataToAdr* сразу после его получения. Вполне "безопасная", но менее эффективная работа с *записями* осуществляется по схеме

[копирование записи в переменную программы —>]
обработка переменной-копии [— > переброска копии в запись],

где» как обычно, квадратными скобками выделены необязательные операции. Эту схему поддерживает пара процедур обмена данными

между записями и переменными программ. Для копирования записи в локальную или глобальную переменную программы надо обратиться к процедуре

ReadData(КодКопируемойЗаписи : *DATACODE;*
VAR КудаКопируетсяЗапись **: ARRAY OF BYTE).**

Обратный обмен осуществляется процедурой

WriteData(КодЗамещаемойЗаписи : *DATACODE;*
ОткудаБеретсяНовоеСодержание : **ARRAY OF BYTE**).

В обоих случаях в качестве фактического значения второго параметра можно использовать либо идентификатор переменной программы, либо выражение-ссылку на такую переменную. Допускается подстановка переменной, которая "короче" записи и соответствует некой ее головной части (т.е. можно оперировать начальными отрезками записей). Подставлять переменные, которые "длиннее" записей, нельзя, причем нарушения этого запрета системой не детектируются и могут приводить впоследствии к плохо диагностируемым аварийным ситуациям.

Чтобы завести новую запись, надо вызвать процедуру

Allocate (**VAR** ВозвращаемыйКодНовойЗаписи : *DATACODE;*
ДлинаЗаписи,
КлассПамятиДляРазмещенияЗаписи : **CARDINAL**).

Смысл ее параметров ясен из контекста.

Для уничтожения записей служит процедура

Deallocate(**VAR** КодУничтожаемойЗаписи : *DATACODE;*
ДлинаУничтожаемойЗаписи : **CARDINAL**).

ликвидировав запись, она присваивает переменной, в которой хра-

нился соответствующий *код,* значение *NonData,* отличное от *кода* любого виртуального объекта. При обращении к этой процедуре надо быть особенно внимательным, чтобы не ошибиться в указании длины уничтожаемой *записи,* т.е. не указать длину, отличную от объявленной ранее при заведении записи процедурой *Allocate:* такая ошибка чревата неконтролируемым системой замусориванием памяти. Наконец, есть процедура копирования одной *записи* в другую:

CopyData(КодКопируемойЗаписи,
КодЗаписиКудаИдетКопирование : *DATACODE;*
КоличествоКопируемыхБайтов **: CARDINAL).**

Откуда и куда произойдет копирование - определяется первым и вторым параметрами, а что именно будет скопировано, устанавливает третий параметр. Его значение не должно превышать длин задействованных *записей,* причем следить за этим обязан пользователь. Если оно меньше какой-то из длин (а они могут быть разными), то это значит, что имеется ввиду копирование или перезапись начального отрезка соответствующей *записи.*

6.6 СОЗДАНИЕ. УНИЧТОЖЕНИЕ. ОЧИСТКА И КОПИРОВАНИЕ СПИСКОВ

Списки составляют, пожалуй, самый нужный для имитации класс динамических данных. Это ясно хотя бы из того, что они являются единственным полноценным заменителем массивов варьируемой длины, запрещенных в большинстве алгоритмических языков и в частности - в языковых системах, с которыми работает *MISS.* Чтобы завести *список,* нужно обратиться к процедуре

CreateList(**VAR** ВозвращаёмыйКодСписка : *LISTCODE;*
ДлинаОднойЗаписиСписка,
КлассПамятиДляРазмашенияСписка : **CARDINAL**).

Она создает пустой (не содержащий ни одной *записи) список,* длина *записей* которого и "место расположения" задаются параметрами вызова- Возвращаемое значение - *код* вновь заведенного *списка.* Уничтожаются *списки,* процедурой

DeleteList(**VAR** КодУничтожаемогоСписка : *LISTCODE*).

По выходе из нее та переменная, что сыграла роль фактического параметра, получает значение *NonList,* отличное от *кода* любого *списка.* Когда нужно "обнуление" *списка* - ликвидация всех входящих в него *записей,* но не его уничтожение - следует обратиться к процедуре

ClearList(КодОбнуляемогоСписка : *LISTCODE*).

Она. естественно, не меняет значения своего параметра. Важно подчеркнуть, что обе представленных процедуры освобождают лишь ту память, которая была занята непосредственно *записями списка;*
если в этих *записях* есть поля-ключи с "подвешенными" к ним динамическими структурами, то об их уничтожении пользователь должен позаботиться сам. К выполняемому *MISS* автоматическому уничтожению *сообщений* данное замечание не относится: там гарантирована полная чистка памяти в соответствии с определениями структур *сообщений,* в *спецификациях*

Последняя возможная операция над *списком* в целом - копирование. Оно выполняется процедурой

CopyList(КодКопируемогоСписка : *LISTCODE;*
КлассПамятиКудаКопируетсяСписок **: CARDINAL;**
VAR ВозвращаемыйКодСпискаКопии : *LISTCODE*).

Создав копию указанного первым параметром *списка* в указанном вторым параметром *классе* памяти, эта процедура третьим параметром возвращает *код* вновь заведенного *списка-копии.*

6.7 ПРОЦЕДУРЫ РАБОТЫ С ЗАПИСЯМИ СПИСКОВ: ПРЯМОЙ ДОСТУП

В этом и следующем подразделах в числе прочих представлены процедуры для обмена содержимым между записями *списков* и переменными программ. При вызове каждая из них проверяет совпадение длины переменной с длиной записи *списка,* объявленной при его создании. В отсутствие такого совпадения вызов приведет к аварийному завершению программы с соответствующей диагностикой.

Под прямым доступом, которому посвящен подраздел, понимается обращение к записям *списка* по их явно указываемым порядковым номерам. Ошибаться в задании этих номеров нельзя и если при обращении к какой-то из представленных ниже процедур в качестве номера нужной записи будет указано число, превосходящее длину *списка* на момент обращения за вычетом единицы (записи в *списке* нумеруются с нуля), то произойдет аварийный останов программы.

Для работы с записью *списка* как с отдельной *записью* имеется процедура

```
GetDataCode( КодСписка : LISTCODE;
             НомерНужнойЗаписи : CARDINAL;
             VAR ВозвращаемыйКодЗаписи : DATACODE ).
```

Ее результат - *код* требуемой записи из указанного *списка.* Получив его, с этой записью можно работать как с одиночной, используя процедуры подразд. 6.3.

Для работы с записью *списка* "на месте" есть процедура

```
RecordAddress( КодСписка : LISTCODE;
               НомерНужнойЗаписи : CARDINAL;
               VAR ВозвращаемыйАдресЗаписи : ADDRESS ).
```

Характер связи получаемого процедурой адреса с самой записью подробно обсуждался при описании процедуры *DataToAdr.* Для копирования нужной записи *списка* в переменную программы следует об-

ратиться к процедуре

ReadRecord(КодСписка : *LISTCODE;*
НомерНужнойЗаписи : CARDINAL;
VAR КудаКопироватьЗапись **: ARRAY OF BYTE),**

а обратное копирование осуществляет процедура

WriteRecord(КодСписка : *LISTCODE;*
НомерЗамещаемойЗаписи **: CARDINAL;**
ОткудаБеретсяНовоеСодержание : **ARRAY OF BYTE**).

Смысл первой пары параметров и той и другой очевиден. Третьим параметром в обе можно подставлять или идентификатор переменной, или выражение-ссылку.

Вставки новых записей в *списка* на произвольные места выполняются процедурой

AddRecord(КодРасширяемогоСписка : *LISTCODE;*
ПодКакимНомеромВставитьЗапись : **CARDINAL**;
ОткудаВзятьСодержаниеЗаписи : **ARRAY OF BYTE**).

Первый параметр ясен, третий имеет тот же смысл, что у процедуры *WriteRecord,* а второй - это номер в *списке,* присваиваемый новой записи. Ее можно поставить либо на место уже существующей *записи,* и тогда эта и следующие за ней записи по завершении *AddRecord* получат номера на единицу больше старых, либо - в "хвост" *списка.* Соответственно, значение второго параметра выбирается из диапазона от нуля до текущей длины *списка.* Добавить запись в конец *списка* можно и процедурой

AddRecordToEnd(КодРасширяемогоСписка : *LISTCODE;*
ОткудаВзятьСодержаниеЗаписи : **ARRAY OF BYTE**).

Для удаления записей из *списков* имеется процедура

DeleteRecord(КодСокращаемогоСписка : *LISTCODE;*
НомерУдаляемойЗаписи **: CARDINAL).**

После ее выполнения все записи, стоявшие в *списке* после удаленной, получат номера на единицу меньше старых.

6.8 ПРОЦЕДУРЫ РАБОТЫ С ЗАПИСЯМИ СПИСКОВ: ПОСЛЕДОВАТЕЛЬНЫЙ ДОСТУП

Работа со *списком* процедурами предыдущего подраздела утяжеляется выполняемыми ими операциями поиска записи нужного номера. При циклической обработке *списка* этого можно избежать, обратившись к описанным ниже процедурам последовательного доступа.

Технология последовательного доступа опирается на понятие *текущей записи списка.* Суть в том, что в каждом непустом *списке* система всегда выделяет одну из записей и позволяет обращаться к таким записям, именуемым впредь *текущими,* не указывая их номеров. Во время работы со списком индекс *текущей записи* может меняться и, в частности, этого следует ожидать после обращения к любой процедуре прямого доступа. При чтении, модификации и вставке записи по прямому доступу *текущей* становится прочитанная, модифицированная и вставленная запись соответственно. При уничтожении записи, если *список* при этом не опустеет, *текущей* станет следующая запись, если она найдется, а иначе - последняя запись *списка.* Номер *текущей записи списка* всегда можно получить как значение функции

GetCurrent (КодСписка : *LISTCODE*) **: CARDINAL.**

Чтобы организовать цикл по *списку,* надо прежде всего установить начало цикла, т.е, - сделать первую обрабатываемую в нем запись *текущей.* Для этого есть три процедуры. При запуске цикла с "середины" *списка текущая* запись устанавливается с помощью

процедуры

SetCurrent(КодСписка : *LISTCODE;*
НомерЗаписиНазначаемойТекущей **: CARDINAL).**

Чтобы *текущей* стала головная запись, нужно вызвать функцию

First (КодСписка : *LISTCODE*) **: BOOLEAN,**

возвращающую **TRUE,** если *список* непуст, и **FALSE** в противном случае. Наконец, назначение *текущей записью* последней записи *списка* осуществляется функцией

Last(КодСписка : *LISTCODE*) **: BOOLEAN.**

Ее результат определяется как у *First.*

Переход к очередной записи при циклической обработке *списка* в режиме последовательного доступа тоже возможен тремя способами. Для просмотра *списка* от начала к концу с обработкой *текущей записи* без ее уничтожения используется функция

Next(КодСписка : *LISTCODE*) **: BOOLEAN.**

Она действует так: если запись, являющаяся *текущей* в момент ее вызова, не стоит в *списке* последней, то возвращается значение TRUE и *текущей* становится следующая по порядку запись; иначе возвращается FALSE и ничего не происходит. То же направление перебора при уничтожении *текущей записи* обеспечивает функция

DeleteCurrentRecord(КодСписка : *LISTCODE*) **: BOOLEAN.**

Возвращаемое ею значение определяется как у *Next,* но она, будучи примененной в случае, когда *текущая запись* замыкает *список* и не единственна, после ее уничтожения делает *текущей* последнюю запись *списка.* Переход к предыдущей записи осуществляет функция

Prev(КодСписка : *LISTCODE*) **: BOOLEAN.**

Если *текущей* в момент вызова *Prev* окажется начальная запись, то ничего не происходит и возвращается **FALSE,** если же переход возможен, он происходит и возвращается **TRUE.** Отметим, что *Prev* работает медленнее, чем *Next.*

Теперь - о средствах доступа к содержимому записей, перебираемых описанными выше способами. Чтение записей в цикле по *списку* выполняется процедурой

Read(КодСписка : *LISTCODE;*
VAR КудаСкопироватьТекущуюЗапись **: ARRAY OF BYTE**).

Она скопирует содержимое *текущей* записи указанного первым параметром *списка* в ту переменную программы, чей идентификатор или ссылка на которую будут вторым параметром обращения. Естественно, есть и парная к ней процедура обратного копирования

Write(КодСписка : *LISTCODE;*
ОткудаВзятьНовоеСодержимоеТекущейЗаписи **: ARRAY OF BYTE**).

Доступ к *коду текущей записи списка* обеспечивает процедура

GetData(КодСписка : *LISTCODE;*
VAR ВозвращаемыйКодЗаписи : *DATACODE*).

Она возвращает искомый *код записи,* если *список* не пуст, и *NonData* - в противном случае. Для работы с *текущей записью* "на месте" можно использовать процедуру

CurRecAddreas(КодСписка : *LISTCODE;*
VAR ВозвращаемыйАдресЗаписи **: ADDRESS**).

Связь полученного адреса с записью обсуждалась выше.

В связи с рассмотренными возможностями циклической обработки записей, необходимо следующее предостережение: поскольку обращение к процедуре прямого доступа обычно сопровождается сменой *текущей записи,* то надо следить за тем, чтобы внутри цикла

обработки *списка* по схеме последовательного доступа не было вызовов таких процедур для того же списка.

Хотя последовательный доступ в основном ориентирован на циклические приложения, это не единственное его назначение: например, функция Last в комплекте с *Read, Write, GetData* или *DeleteCurrentRecord,* позволяет поработать с замыкающей записью *списка* без выяснения ее номера. Сказанное особенно относится к четырем еще не описанным процедурам, которые тоже привязаны к *текущей записи,* но обычно используются для одиночных вызовов. Процедура

InsertAfterCurrent (КодСписка : *LISTCODE;*
ОткудаВзятьНовуюЗапись **: ARRAY OF BYTE**)

вставляет за *текущей записью списка,* указанного первым параметром, новую запись и копирует в нее значение переменной, имя которой или ссылка на которую задается вторым параметром. Вставленная запись становится *текущей,* Чтобы сделать то же за вычетом копирования, надо обратиться к процедуре

InsAftCurrent (КодСписка : *LISTCODE*),

и тогда содержимое заведенной записи будет неопределенным. Вставка записи перед *текущей* возможна либо через процедуру

InsertBeforeCurrent (КодСписка : *LISTGODE;*
ОткудаВзятьНовуюЗапись **: ARRAY OF BYTE**),

одновременно копирующую в новую запись содержимое переменной, указанной вторым параметром, либо - процедурой

InsBefCurrent (КодСписка : *LISTCODE*),

которая заводит запись с неопределенным содержимым. Первая запись в пустом *списке* может быть заведена любой из этих четырех процедур.

6.9 СОЗДАНИЕ, УНИЧТОЖЕНИЕ, ОЧИСТКА И КОПИРОВАНИЕ МАССИВОВ

Этот и два следующих подраздела посвящены *массивам* - блокированным *спискам* записей, которые иногда предпочтительнее обычных *списков* (когда именно - сказано в подразд. 6.1). Поскольку *массивы* все же являются *списками,* есть аналогия между обслуживающими их процедурами и процедурами трех предыдущих подразделов. Она избавляет от необходимости давать подробные разъяснения по всем средствам работы с *массивами,* и мы будем делать это только в случаях, когда не сможем на нее сослаться.

Четырем процедурам подразд. 6.6 отвечает такая четверка:

а) процедура

CreateArray (**VAR** КодСозданногоМассива : *ARRCODE;*
ДлинаОднойЗаписиМассива, ЧислоЗаписейВБлоке,
КлассПамятиДляРазмещенияМассива **: CARDINAL**)

для заведения новых *массивов;*

б) процедура

DeleteArray (**VAR** КодУничтожаемогоМассива : *ARRCODE*)

для уничтожения *массивов* (возвращает значение *NonArray};*

в) процедура

ClearArray(**VAR** КодОбнуляемогоМассива : *ARRCODE*)

для обнуления (уничтожения всех записей) *массивов;*

г) процедура

CopyArray(КодКопируемогоМассива : *ARROODE;*
КлассПамятиКудаКопируетсяМассив **: CARDINAL;**
VAR ВозвращаемыйКодМассиваКопии : *ARRCODE*)

для копирования *массивов.*
В пояснении нуждается только второй параметр первой процедуры:

все остальные параметры интерпретируются прямым сопоставлением с соответственными параметрами схожих процедур из подразд. 6.6. Значение указанного параметра, имеющее смысл числа записей в одном блоке, должно лежать в диапазоне [2-.32764], а его произведение на величину первого параметра не должно превосходить 32764.

6.10 ПРОЦЕДУРЫ РАБОТЫ С ЗАПИСЯМИ МАССИВОВ: ПРЯМОЙ ДОСТУП

Четырем первым процедурам подразд. 6.7 соответствуют следующие процедуры прямого доступа к записям *массивов:*

а) процедура

ElemCode(КодМассива : *ARRCODE;*
НомерНужнойЗаписи **: LONGINT;**
VAR ВозвращаемыйКодЗаписи : *DATACODE*),

возвращающая *код* указанной первыми двумя параметрами записи;

б) возвращающая адрес записи для работы "на месте" процедура

ElemAddress(КодМассива : *ARRCODE;*
НомерНужнойЗаписи **: LONGINT;**
VAR ВозвращаемыйАдресЗаписи : **ADDRESS**);

в) процедура

ReadElem(КодМассива : *ARRCODE;*
НомерНужнойЗаписи **: LONGINT;**
VAR КудаКопироватьЗапись **: ARRAY OF BYTE**),

копирующая содержимое указанной первыми двумя параметрами записи в указанную третьим параметром переменную программы;

г) процедура

WriteElem(КодМассива : *ARRCODE;*
НомерЗамещаемойЗаписи : **LONGINT**;
ОткудаБеретсяНовоеСодержимое **: ARRAY OF BYTE**),

копирующая в указанную первыми двумя параметрами запись значение указанной третьим параметром переменной.

Поскольку, в отличие от обычных *списков,* в *массив* нельзя вставить новую запись на произвольное место, а можно - только в "хвост", и нельзя выкинуть из *массива* произвольную запись, а можно - только последнюю, то из тройки последних процедур подразд. 6.7 для *массивов* имеет аналог только одна, предпоследняя. Этим аналогом является процедура

AddElemToEnd(КодПополняемогоМассива : *ARRCODE;*
ОткудаВзятьСодержаниеЗаписи **: ARRAY OF BYTE**),

добавляющая в конец *массива* новую запись, содержимое которой берется из указанной вторым параметром переменной. Добавить новую запись с неопределенным содержимым позволяет процедура

AddToEnd(КодПополняемогоМассива : *ARRCODE*).

6.11 ПРОЦЕДУРЫ РАБОТЫ С ЗАПИСЯМИ МАССИВОВ: ПОСЛЕДОВАТЕЛЬНЫЙ ДОСТУП

Процедуры последовательного доступа к записям *массивов,* как и те, что были описаны в подразд. 6.8, ориентированы на обработку записей в цикле и также опираются на понятие *текущей записи*. Сохраняется предостережение относительно возможности смены *текущей записи* вызовом какой-то из процедур прямого доступа.

Номер *текущей записи* непустого *массива* всегда можно получить как результат аналогичной *GetCurrent* процедуры

GetCurrentElem(КодМассива : *ARRCODE;*
VAR НомерТекущейЗаписи : **LONGINT**).

Полная аналогия есть и по процедурам установки *текущей записи.*

Тройке таких процедур для обычного списка отвечают:

а) процедура

SetCurrentElem(КодМассива : *ARRCODE;*
НомерЗаписиНазначаемойТекущей : **LONGINT**).

результат вызова которой состоит в том, что запись, имеющая указанный вторым параметром номер, становится *текущей* для указанного первым параметром *массива;*

б) функция

FirstElem(КодМассива : *ARRCODE*) **: BOOLEAN,**

которая делает *текущей* начальную запись *массива,* если он не пуст, и при этом возвращает **TRUE, а** иначе просто возвращает **FALSE;**

в) функция

LastElem(КодМассива : *ARROODE*) **: BOOLEAN.**

которая делает *текущей* последнюю запись *массива,* если он не пуст, и при этом возвращает **TRUE,** а иначе возвращает **FALSE.**

Для перехода к следующей и предыдущей записям *массива* есть, во-первых, соответствующая процедурам *Next* и *Prev* пара

NextElem(КодМассива : *ARROODE*) **: BOOLEAN**

PrevElem(КодМассива : *ARRCODE*) **: BOOLEAN,**

а во-вторых, имеется "похожая" на *DeleteCurrentRecord* процедура

DeleteLastElem(КодМассива : *ARRCODE*) **: BOOLEAN.**

которая работает несколько иначе. Она уничтожает запись *массива,* стоящую в нем последней, и при этом предыдущая запись, если она существует, становится *текущей* и тогда возвращается TRUE, а если уничтоженная запись была единственной, возвращается FALSE.

Процедуры доступа к *текущей записи массива* также вполне аналогичны четырем соответствующим процедурам для *списков,* это:

а) процедура чтения

ReadCurrentElem(КодМассива : *ARRCODE;*
VAR ПеременнаяКудаБудетСчитанаЗапись **: ARRAY OF BYTE**);

б) процедура записи

WriteCurrentElem(КодМассива : *ARRCODE;*
ОткудаВзятьНовоеСодержимоеЗаписи **: ARRAY 0F BYTE**);

в) процедура, возвращающая адрес для работы "на месте"

CurrentElemAddress(КодМассива : *ARRCODE;*
VAR ВозвращаемыйАдресЗаписи **: ADDRESS**);

г) процедура

CurrentElemCode (КодМассива : ARRCODE;
VAR ВозвращаемыйКодЗаписи : *DATACODE*),

возвращающая *код текущей записи массива.* В случае, если *массив* пустой, последние две процедуры возвращают значение *NonData.*

Так как вставлять в *массив* записи на произвольные места нельзя, то никаких процедур, вполне аналогичных четверке последних из подразд. 6.8, для массивов нет.

6.12 ПРОЦЕДУРЫ РАБОТЫ О ДЛИННЬМИ ЗАПИСЯМИ

Здесь будут рассмотрены процедуры, позволяющие работать с записями длиной более 32768 байтов. Как составляющие базы данных модели, такие записи скорее всего не понадобятся, тем более. что вызывать процедуры их загрузки и выгрузки можно только из *сервисного модуля.* Однако они используются в качестве "носителей" программ *виртуальных модулей* и поэтому для того, кто соби-

рается строить собственную систему *виртуальных модулей* со сложными связями, знание представленных ниже процедур небесполезно. *Длинные записи* заводятся процедурой

GetParas(**VAR** КодНовойДлинноЗаписи : *PARACODE;*
ДлинаЗаписиВПараграфах,
КлассПамятиДляРазмещения : **CARDINAL**).

Смысл параметров ясен из названий (размеры *длинные записей* даются в параграфах; параграф = 16 байтов). Процедура

FreeParas(**VAR** КодУничтожаемойЗаписи : *PARACODE;*
ДлинаУничтожаемойЗаписиВПараграфах : **CARDINAL**)

уничтожает *длинную запись,* ее *код* примет значение *NonPara.*

Чтобы загрузить *длинную запись* в оперативную память, надо обратиться к одной из функций

Load(КодЗагружаемойЗаписи : *PARACODE;*
ДлинаЗаписиВПараграфах **: CARDINAL**) : **CARDINAL** или

Fix(КодЗагружаемойЗаписи : *PARACODE;*
ДпинаЗаписиВПараграфах : **CARDINAL**) : **CARDINAL**.

В обоих случаях возвращаемое значение - сегмент адреса загруженной *длинной записи,* через который открывается доступ к ее содержимому. Если к моменту обращения в оперативной памяти имелась загруженная ранее функцией *Load* другая *длинная запись,* то она может быть выгружена; *длинные записи,* загруженные функцией *Fix,* не выгружаются: эта функция не только загружает *запись,* но и (фиксирует ее в памяти (см. подразд. 6.2). Счетчик загруженных *длинных записей* увеличивается на единицу.

Загруженная через *Load, длинная запись* остается в памяти до выгрузки представленной ниже процедурой *Unload* или до загрузки следующей *длинной записи* программами *Load* или *Fix;* при загрузке через *Fix длинную запись* можно будет выгрузить только процеду-

рой *Unload.* Функция *Fix* уменьшает общий объем памяти системы. Перед загрузкой *длинной* записи полезно вызвать функцию

ParasAvail() **: CARDINAL.**

Ее результат - количество параграфов оперативной памяти, которое можно использовать для загрузки. Процедура

Unload()

выгружает текущую *длинную запись* и уменьшает на единицу счетчик загруженных *длинных* записей. Если его значение останется ненулевым, то в память будет возвращена и станет доступной по прежнему адресу предпоследняя загруженная *длинная* запись. Текущее значение счетчика возвращает функция

Level() **: CARDINAL.**

6.13 ПРОЦЕДУРЫ РАБОТЫ С ВИРТУАЛЬНЫМИ МОДУЛЯМИ

В виртуальной памяти наряду с данными разрешается хранить и программы, которые могут вызываться как из имитирующих программ так и одна из другой. Эти программы будем называть *виртуальными модулями.* Таковыми, в частности, являются модули приборов и сервисный *модуль. Виртуальные модули* - это принятый в *MISS* способ организации оверлейного режима прохождения программ. Глобальные процедуры, функции и переменные виртуального *модуля* доступны через специальный интерфейс в других модулях.

К *виртуальным модулям* предъявляются следующие требования:

а) *виртуальный модуль* должен быть главной программой;

б) *виртуальный, модуль* должен быть откомпилирован и собран (в случае *МОДУДЫ-2/SDS* - с оверлейной опцией линковщика);

в) слинкованный *виртуальный, модуль* должен иметь имя с рас-

ширением ".OVY" или ".EXE" и при запуске на счет или сборке модели должен находиться в доступном DOS-каталоге;

г) глобальные процедуры и переменные *виртуального модуля,* предназначенные для использования в других модулях, должны быть отмечены в его исполняемой части вызовами специальных процедур *ExportProc* и *ExportData,* описание которых приводится ниже.

При запуске на счет слинкованной в *MISS* модели модули *приборов* и сервисный *модуль* виртуализируются автоматически. Для того же чтобы можно было работать с произвольным *виртуальным модулем,* его сначала нужно самому включить в систему. Это можно сделать двумя способами, первый - через процедуру

```
Import ( VAR ВозвращаемыйКодМодуля : MODCODE;
         ИмяМодуля : ARRAY OF CHAR;
         КлассПривязкиМодуля,
         КлассХраненияПрограммыМодуля : CARDINAL ).
```

При вызове она прежде всего попытается отыскать в доступном DOS-каталоге оверлейный модуль, зарегистрированный там под именем, указанным вторым параметром (его значением может быть либо текстовая константа либо идентификатор массива букв, в начало которого записано нужное имя с нулевым символом после завершающей буквы), и с расширением ".EXE" или ".OVY" (в подставляемое ИмяМодуля расширение входить не должно), Если такого модуля не найдется, функция вернет значение *NonModule* и больше ничего делать не будет. В противном случае она включит найденный модуль в систему (виртуализирует его), поместив информацию о нем и его содержимое в *классы* виртуальной памяти, заданные третьим и четвертым параметрами соответственно. Возвращаемым значением при этом будет сохраняемый *код виртуального модуля* в системе, обеспечивающий доступ к нему из других программ. Отделение программной части модуля от информационной мотивируется тем,

что это - наименее "устойчивая" его часть, часто меняющаяся, например, в ходе отладки; поэтому может возникнуть желание помещать ее во временный *класс* памяти.

Независимо от того, виртуализируется ли модуль автоматически или "вручную", операторы из его тела исполняются при виртуализации и должны включать обращения к процедурам *ExportProc* и *Export Data* (см, ниже) по поводу каждой переменной, процедуры или функции модуля, открываемой для использования извне. Сказанное не относится к открываемым диспетчеру *MISS* процедурам *элементов* из модулей *приборов:* они, как уже было сказано в подразд. 5.1 экспортируются процедурой *LinkEIement.*

При обращении к функции *Import* модуль будет включен в систему, даже если там уже есть одноименные *виртуальные модули.* С ними ничего не произойдет и различать их можно будет по *кодам,* которые возвращает Import и которые всегда уникальны.

Второй способ включить модуль в систему - вызов процедуры

Include (**VAR** ВозвращаемыйКодМодуля : *MODCODE;*
ИмяМодуля **: ARRAY OF CHAR,**
КлассПривязкиМодуля **: CARDINAL**).

Если в указанном *классе* уже есть *виртуальный модуль* с указанным именем, она просто вернет его *код,* а иначе сработает как функция *Import с* третьим параметром, равным четвертому.

По *виртуальным модулям* можно получать справки трех типов. Процедура

*Find***(VAR** ВозвращаемыйКодМодуля : *MODCODE;*
ИмяВиртуальногоМодуля **: ARRAY OF CHAR,**
КлассПривязкиМодуля **: CARDINAL**)

ищет в указанном третьим параметром *классе* памяти *виртуальный модуль с* указанным вторым параметром именем и если находит. возвращает сохраняемый *код* найденного *модуля,* а иначе -

NonModule; процедура

Name(КодМодуля : *MODCODE;*
VAR ИмяМодуля **: ARRAY OF CHAR**)

позволяет узнать имя *виртуального модуля* по его *коду;* функция

ModBodyLength(КодМодуля : *MODCODE*) : **CARDINAL**

дает длину программы *виртуального модуля* в параграфах. Использование ее совместно с *ParasAvail* из предыдущего подраздела позволяет определить возможность обращения к *модулю:* если результат *ParasAvail* меньше результата *ModBodyLength,* то попытка вызвать *модуль* в оперативную память (либо применением к нему описанной ниже процедуры *FixInMemory*, либо вызовом какой-то из его процедур) закончится аварийно.

Изменив после завершения сеанса с системой программу хранимого в ней *виртуального модуля,* можно обновить *виртуальный. модуль* во время следующего сеанса» не заменяя его *кода* в системе. Это делает функция

Renovate (КодОбновляемогоМодуля : *MODCODE;*
КлассХраненияПрограммыОбновленногоМодуля : **CARDINAL**)
: BOOLEAN.

Обновленная программа будет помещена в *класс* памяти, который указывается вторым параметром обращения и может отличаться от *класса,* где хранилась прежняя программа (*класс* привязки модуля не меняется). При вызове *Renovate* в доступном DOS-каталоге должна находиться новая версия оверлейно слинкованного модуля с прежним именем. В случае успешного обновления возвращается **TRUE,** иначе - **FALSE.** При отладке программ во избежание "разбухания" файла, содержащего *виртуальные модули,* рекомендуется включать программы отлаживаемых модулей во временные *классы* памяти и обновлять перед использованием.

Подчеркнем, что прерывания сеансов с системой никак не отражаются на значениях глобальных переменных *виртуального модуля* до тех пор, пока к нему не применена функция *Renovate:* при ее вызове заново выполнятся операторы тела *модуля,* и если среди них есть операторы инициализации каких-то глобальных переменных, то эти переменные по завершении *Renovate* будут иметь значения. заданные указанными операторами; значения прочих переменных станут неопределенными.

Для уничтожения *виртуальных модулей* служит процедура

Exclude (**VAR** КодИсключаемогоМодуля : *MODCODE*).

По выходе из нее параметр вызова получит значение *NonModule.*

Виртуальный модуль можно зафиксировать в оперативной памяти (см. подразд. 6.2) и тогда он будет сохранен в ней до соответствующего обращения к процедуре *Unload* (зафиксированные модули образуют список, ведомый по схеме LIFO - "последний вошел - первый вышел") из предыдущего подраздела. *Сервисный модуль* фиксируется в памяти автоматически при запуске модели на счет; модули *приборов* автоматически не фиксируются, но могут быть зафиксированы пользователем. Фиксацию выполняет процедура

FixInMemory (КодФиксируемогоМодуля : *MODCODE*),

которую можно вызывать только из уже зафиксированного *модуля;*

сначала таковым будет лишь *сервисный модуль.* Смысл фиксации в том. что обеспечиваемое ею постоянное пребывание *виртуального модуля* в оперативной памяти увеличивает эффективность работы с его процедурами и данными.

Возможность доступа извне к глобальным переменным, процедурам и функциям *виртуального модуля* реализуется через специальный интерфейс. Прежде всего, надо сказать, что доступ возможен только к тем объектам *виртуального модуля,* которые открываются

для этого вызовами в его теле процедур

ExportProc(ЭкспортируемаяПроцедура **: PROC;**
НомерЭкспортируемогоОбъектаВМодуле **: CARDINAL**). или

ExportData (АдресЭкспортируемойПеременной **: ADDRESS;**
НомерЭкспортируемогоОбъектаВМодуле **: CARDINAL**).

Если экспортируется глобальная переменная, то первым параметром должен быть ее адрес, вычисленный через стандартную функцию **ADR.** Если же требуется сэкспортировать процедуру или функцию, то надо завести в *виртуальном модуле* глобальную переменную соответствующего типа. присвоить ей в теле *модуля* значение экспортируемой процедуры или функции и подставить первым параметром вызова *ExportProc* эту переменную. Второй параметр процедур экспорта служит для идентификации экспортируемых объектов вне модуля-хозяина. Их достаточно просто пронумеровать и тогда можно задавать в качестве значений этого параметра их номера. Поясним сказанное примером:

```
DEFINITION MODULE ИнтерфейсныйМодуль;
  TYPE
  ИменаОбъектов = ( функция, переменная );
ТипФункции = PROCEDURE( VAR LONGREAL; CARDINAL ) :
BOOLEAN;
   ТипПеременной = RECORD
     ЦелоеБезЗнака : CARDINAL;
 вещественное : LONGREAL;
                    END;

END ИнерфейсныйМодуль.

MODULE ВиртуальныйМодуль;
  FROM SYSTEM IMPORT ADR;
  FROM ИнтерфейсныйМодуль IMPORT ИменаОбъектов, ТипФункции,
  ТипПеременной;
  FROM OvyManager IMPORT ExportProc, ExportData;

  VAR
    ЭкспортируемаяПеременная : ТипПеременной;
```

```
    ПеременнаяДляЭкспортаФункции : ТипФункции;
PROCEDURE ЭкспортируемаяФункция( VAR параметр1 : LONGREAL;
                                 параметр2 : CARDINAL ) : BOOLEAN;

    END ЭкспортируемаяФункция;
                •                    •                    •
BEGIN
ПеременнаяДляЭкспортаФункции := ЭкспортируемаяФункция;
ExportProc (PROC (ПеременнаяДляЭкспортафункции), ORD (функция));
  (*          Оператор          экспорта          функции;          *)
  ExportData (ADR (ЭкспортируемаяПеременная). ORD (переменная));
    (* Оператор экспорта переменной. *)

END ВиртуальныйМодуль.
```

Рассмотрим теперь процедуры, позволяющие воспользоваться вне *виртуального модуля* экспортируемыми из него переменными, процедурами и функциями. Сразу подчеркнем, что для обращения к экспортируемым объектам *сервисного модуля* имеется более простой интерфейс (см. разд. 8).

Для использования какой-то процедуры или функции *виртуального модуля* в другом модуле надо получить ее в качестве значения переменной соответствующего типа. Это делается процедурой

```
ImportProc( КодМодуляХозяина : MODCODE;
            НомерИмпортируемогоОбъектаВМодуле : CARDINAL;
            VAR ПеременнаяТипаПроцедурыИлиФункции : PROC ).
```

Первый параметр ее вызова - *код виртуального модуля-хозяина* нужной процедуры или функции; второй параметр - номер (задаваемый непосредственно либо через элемент специального перечисляемого типа из интерфейсного модуля), под которым эта функция или процедура зарегистрирована в *модуле;* третий параметр - переменная-функция или переменная-процедура вызывающего модуля, отождествляемая с импортируемым объектом: после выполнения

ImportProc имя этой переменной становится "кличкой", по которой можно вызвать данный объект.

Если экспортирующий *виртуальный модуль* не зафиксирован в оперативной памяти процедурой *FixInMemory,* то использование переменной, получившей значение от *ImportProc,* корректно только до следующего вызова *ImportProc*. Если же *виртуальный модуль* фиксирован, то следующий вызов *ImportProc* не портит результата предыдущего и этот результат сохранит смысл вплоть до выгрузки *модуля* процедурой *Unload.*

Рассмотрим пример обращения к импортируемой из *виртуального модуля* функции. В качестве таковой возьмем функцию "ЭкспортируемаяФункция" из предыдущего примера. При этом предположим, что "ВиртуальныйМодуль" из того же примера либо включен в 3-й *класс* системы, либо находится в доступном DOS-каталоге в виде оверлейного файла. Тогда использование импортируемой функции могло бы выглядеть так:

```
FROM OvyManager IMPORT MODCODE, Include, ImportProc;
FROM ИнтерфейсныйМодуль IMPORT ИменаОбъектов, ТипФункции;
VAR
   ИмпортируемаяФункция : ТипФункции;
   КодМодуляХозяина     : MODCODE;
   результат : BOOLEAN;
   параметр  : LONGREAL;

  Include( КодМодуляХозяина, "ВиртуальныйМодуль". 3 );
  ImportProc( КодМодуляХозяина, ORD (функция),
                                PROC (ИмпортируемаяФункция) );
  результат := ИмпортируемаяФункция( параметр, 1 );
```

Для импорта данных имеется процедура

```
ImportData( КодМодуляХозяина : MODCODE;
            НомерИмпортируемогоОбъектаВМодуле : CARDINAL;
            VAR КодИмпортируемойПеременной : DATACODE );
```

Возвращаемое третьим параметром значение - *код записи* идентифицируемой по второму параметру глобальной переменной *виртуального модуля.* Доступ к импортируемой переменной по этому коду осуществляется обычным способом - через процедуры подразд. 6-5. Связь *кода* с переменной сохраняется до применения процедур *Renovate* или *Exclude* к *виртуальному модулю-хозяину* (см, выше замечание к процедуре *Renovate).*

Для иллюстрации импорта переменной вновь воспользуемся модулями из примера экспорта (считаем, что "ВиртуальныйМодуль" либо включен в 3-й *класс* памяти, либо находится в доступном каталоге). Программа с обращением к переменной, которую экспортирует "ВиртуальныйМодуль" могла бы выглядеть так:

```
FROM OvyManager IMPORT MODCODE, Include, ImportData;
FROM Интерфейсныймодуль IMPORT ИменаОбъектов, ТипПеременной;
FROM VirtualMemory IMPORT DATACODE, DataToAdr;
VAR
   УказательИмпортируемойПеременной : POINTER TO ТипПеременной;
   КодМодуляХозяина : MODCODE;
   КодЗаписиИмпортируемойПеременной : DATACODE;

Include ( КодМодуляХозяина, "ВиртуальныйМодуль", 3 );
    ImportData ( КодМодуляХозяина, ORD (переменная),
КодЗаписиИмпортируемойПеременной );
    DataToAdr ( КодЗаписиИмпортируемойПеременной,
                              УказательИмпортируемойПеременной );
    WITH УказательИмпортируемойПеременной^ DO
       ЦелоеБезЗнака := 2;
      вещественное  := 3.1415;
    END;
```

В заключение, сделаем несколько замечаний:

а) для импорта процедур и переменных из *сервисного модуля* имеются более удобные средства (см. разд. 8);

б) процедуры *Unload, Load, Fix* и *FixInMemory* можно вызывать только из фиксированных *виртуальных модулей;*

в) вызывая *Unload,* надо быть уверенным в том, что текущий уровень загрузки (значение функции *Level)* строго больше, чем он был сразу после фиксирования *модуля* в памяти, т. е., - что она не выгрузит вызывающий *модуль;*

г) к остальным процедурам, и в том числе - к *ImportProc,* можно обращаться в любом *модуле.*

6.14 РАБОТА С НЕВИРТУАЛЬНОЙ ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

Как уже упоминалось, 8-й *класс* постоянно находится в оперативной памяти. Все перечисленные выше процедуры применимы к объектам этого *класса,* хотя такие процедуры как *DataToAdr, Fix, Load, FixInMemory, Unload* теряют смысл. Объясняется это тем, что объекты 8-го *класса* находятся в памяти постоянно, а их *коды* - это обычные адреса.

Перейдем к описанию процедур работы с памятью Heap . Выделения и освобождения этой памяти осуществляются процедурами

ALLOC(**VAR** ВозвращаемыйАдресЗаписи **: ADDRESS;**
ДлинаЗаписиВБайтах **: CARDINAL)**

и

DEALLOC(**VAR** ВозвращаемыйАдресЗаписи : **ADDRESS**;
ДлинаЗаписиВБайтах : **CARDINAL**).

Результаты их вызовов эквивалентны результатам вызовов Allocate и *Deallocate* с явным указанием 8-го *класса.*

функция

Available (ДлинаЗаписи **: CARDINAL) : BOOLEAN**

возвращает TRUE, если в Heap можно завести *запись* размером ДлинаЗаписи в байтах, и FALSE в противном случае. функция

PagesAvail() *:* **CARDINAL**

возвращает число страниц по 32К, на которое можно увеличить память под Heap Процедура

IncrHeap(НаСколькоСтраницУвеличить **: CARDINAL)**

увеличивает память Heap на указанное число страниц. Процедура *DeleteHeap*()

освобождает память, занятую Heap. Процедуры

SetAuto() и *StopAuto*()

разрешают и соответственно запрещают автоувеличение Heap на одну страницу в случае, если для выделения очередной *записи* в Heap нет памяти. По умолчанию автоувеличение разрешено.

6.15. СОЗДАНИЕ, КОНСЕРВАЦИЯ И РАСКОНСЕРВАЦИЯ ВИРТУАЛЬНОЙ ПАМЯТИ

Инициализация, консервация и восстановление *классов* памяти с номерами 2, 5 и 7 обеспечиваются MISS автоматически, а *классы* О, 1 и 8 являются временными и не подлежат консервации и восстановлению. Поэтому применять представленные ниже процедуры и функции пользователю разрешено только для *классов* 3, 4 и 6.

Классы виртуальной памяти рождаются сами собой при создании виртуальных объектов процедурами *Allocate, CreateList,* и т.д. с соответствующими значениями указывающих *класс* парамет-

ров. Уничтожается *класс* процедурой

ClearClass (НомерУничтожаемогоКлассаПамяти **: CARDINAL).**

При этом будет освобождена вся занимаемая им память.

Пользователь может законсервировать свой *класс* - перенести его содержимое на диск при гарантированной возможности восстановления в виртуальной памяти по надлежащему запросу. Консервация осуществляется процедурой

CloseClass(ИмяФайлаГдеБудетСохраненКласс **: ARRAY OF CHAR;**
НомерКонсервируемогоКлассаПамяти **: CARDINAL**);

При ее вызове указанный вторым параметром *класс* памяти переписывается в файл, имя которого без расширения - первый параметр вызова; расширением будет ".N", где *N*- номер сохраняемого *класса.* Виртуальная память, занимаемая *классом,* освобождается.

Скопировать *класс* на диск, не уничтожая его. позволяет процедура

SaveClass(ИмяФайлаГдеБудетСохраненКласс : **ARRAY OF CHAR**;
НомерСохраняемогоКлассаПамяти **: CARDINAL).**

Класс, сохраненный ранее на диске процедурой *SaveClass* или *CloseClass,* можно вернуть в виртуальную память функцией

OpenClass(ИмяФайлаГдеХранитсяКласс **: ARRAY OF CHAR;**
НомерВосстанавливаемогоКласса **: CARDINAL) : BOOLEAN;**

Ее значением будет **TRUE** при успешной расконсервации и **FALSE** - в случае неудачи. Если при вызове *OpenClass* в виртуальной памяти уже есть *класс* с тем же номером, он будет уничтожен.