ВВЕДЕНИЕ.

Параллельные вычисления – современная многогранная область вычислительных наук, бурно развивающаяся и являющаяся наиболее актуальной в ближайшие десятилетия. Актуальность данной области складывается из множества факторов, и в первую очередь, исходя из потребности в больших вычислительных ресурсах для решения прикладных задач моделирования процессов в физике, биофизике, химии и др. К тому же, традиционные последовательные архитектуры вычислителей и схем вычислений в преддверии технологического предела. В то же время технологический прорыв в области создания средств межпроцессорных и межкомпьютерных коммуникаций позволяет реализовать одно из ключевых звеньев параллелизма – эффективное управление в распределении вычислений по различным компонентам интегрированной вычислительной установки. Заметим, что развитие квантовых вычислителей и вычислений находится в стадии исследований, и вряд ли стоит ожидать появления промышленных образцов таких вычислителей в ближайшие 20-30 лет. То есть, ближайшие десятилетия пройдут под знаменем развития и распространения параллельных архитектур, средств описания и реализации параллельных вычислений.

История параллельных вычислений начиналась с некоторым, не очень значительным опозданием от начала возникновения концепций последовательных вычислений. Вероятно, к первой и наиболее целостной модели параллельных вычислений можно отнести концепцию клеточных автоматов Дж. Фон Неймана ([]). В то же время инженерно-конструкторские работы над такими проектами вычислителей, как ILLIAC-III (позже, ILLIAC-IV), CRAY-I, STAR-100 и STARAN были ориентированы на использование параллельных схем реализации программ (перечисленные проекты начиналась в 50-х годах).

Начало теоретическим исследованиям параллельных вычислений положила известная работа Карпа и Миллера [1], предопределившая множество вариаций на тему формальных и автоматных схем описания параллельных вычислений [2]. Предложенные варианты представления параллельных вычислений выявили множество свойства и проблем, присущих только параллелизму и не характерных для последовательных вычислений (устойчивость и однозначность, конечной задержки и пр. – см. работы Деннинга [3]).

Одновременно, были определены фундаментальные подходы к параллельной алгоритмизации: глобальный и “близорукий”, потоковой (data-flow) и операторной схемам реализации вычислений. Кроме того, обозначились принципиальные подходы к проблеме программирования параллельных вычислений: автоматическое распараллеливание программ и императивное/директивное программирование параллельных вычислений, мелкозернистое и крупнозернистое распараллеливание программ.

Современные исследования и работы сконцентрированы, в основном, на разработке многоядерных процессоров и многопроцессорных систем (Intel, AMD, IBM), процессоров с интеграцией специализированных подпроцессоров (DSP-процессоры, нейронные процессоры и т.п., производимые HP, TI и другими компаниями). Комплексные решения, в основном, заключаются в построении кластеров, интегрирующих как многопроцессорные, так и однопроцессорные компьютеры.

Резюмируя данный экскурс надо отметить, что на фоне бурного роста технических решений математическое обеспечение в области организации и реализации параллельных вычислений остается крайне проблемной, во многом, открытой областью. Проблемы и принципы решений в этой области – основной предмет обсуждения в данной работе. В

частности, следующие: каковы основания параллельных вычислений и какова предпочтительная актуальность автоматического распараллеливания программ и средств параллельного программирования; какими свойствами должны обладать средства параллельного программирования.

Предпочтения методу при создании параллельных программ диктуются принципиальными свойствами и различиями при алгоритмизации параллельных и последовательных вычислений. Основой любого распараллеливания является наличие независимых групп данных, которые могут быть подвергнуты одновременной обработке, а также - наличие в алгоритме функционально и информационно независимых процедурных единиц, которые также могут быть выполнены одновременно. Стремление осуществить декомпозицию алгоритма с целью получения независимых групп данных часто связано с дроблением или дублированием структур данных, что противоречит основным принципам последовательного программирования, где эффективность программ достигается в том числе через минимизацию используемых в программах данных (эффективность по памяти) и минимизации процедурных единиц (дробление однородной совокупности данных приведет к возникновению дополнительному программированию по управлению фрагментами, что с точки зрения последовательных алгоритмов выглядит нелепо). В то же время, построение алгоритма по последовательной схеме обработки данных так или иначе приводит к последовательному построению вызовов функциональных процедур обработки этих данных. Одновременная множественная обработка единообразных данных требует либо специальной организации в агрегировании данными внутри самих процедур обработки (для реентерабельных процедур необходима особая организация всего пула локальных переменных, внимательная работа с переменными, созданными на статической основе), либо же копированием и/или распределением исполняемых процедур по разным вычислителям с исключением возможности использования общих полей памяти. В то же время, декомпозиция задачи при “параллельном мышлении” должна осуществляться с учетом возможности одновременного исполнения тех или иных процедур.

Таким образом, концепции и критерии последовательной и параллельной алгоритмизации и программирования противоречат друг другу. И как следствие, незначительность результатов в области автоматического распараллеливания последовательных программ (от 10 до 20 процентов ускорения). В целом, это направление носит коммерческо-технологический характер и реализует стремление адаптировать уже разработанное программное обеспечение к новым архитектурам. К тому же, существует проблема перестройки мышления разработчиков программного обеспечения: параллельные алгоритмы требуют особого осмысления задач в части их декомпозиции, алгоритмизации и программирования. Резюмируя, автоматическое распараллеливание программ – направление переходного этапа от последовательных схем вычислений к параллельным.

Наиболее корректным и, безусловно, перспективным является создание средств параллельного программирования и инструментальных сред разработки программ. Данные средства могут быть узко ориентированными на конкретную параллельную архитектуру, либо носить относительно универсальный характер (эффективно охватить все возможные интерпретации параллельных схем вычислений практически невозможно).

Какие же задачи должен решать язык параллельного программирования, и какие задачи решаются при реализации собственно параллельных вычислений?

Потенциал языка программирования должен обеспечить возможность выразить принцип параллельной схемы реализации программы, обеспечить адекватную структуризацию данных, формирование такой структуры программы, которая отвечает заданным принципам распределения процедурных фрагментов для распределенного исполнения.

Реализация задач распределения командных/процедурных фрагментов программы по исполнительным устройствам вычислительной установки, а также задач распределения данных и интеграции всего событийного ряда обеспечивают исполнение вычислений. Решение этих вопросов напрямую зависит от того, какова идейная основа обработки данных. В целом, это двуединая диалектическая проблема: структурирование данных в известном смысле предопределяет процедурную основу и в то же время процедурное начало задает направленность к адекватной структуризации данных.

Важнейшей составляющей практически для всех параллельных архитектур является возможности миграции данных и фрагментов программ. Миграция данных может присутствовать явно или неявно и крайне важна в первую очередь для архитектур с явно или неявно разделяемой памятью. В то же время структурное представления алгоритма и соответствующей программы во многом определяет возможность по распределению программы по исполняющим ее устройствам. Требование к возможности миграции фрагментов программ приводит к идее о представлении программы и ее фрагментов в качестве структуры данных. Возможность представления программ в качестве структур данных обеспечивает еще одно важное свойство – возможность доступа к компонентам программы как к данным, а следовательно – возможность изменения программ в процессе вычисления, в том числе – самоизменения.

Императивное параллельное программирование (запуск параллельных процессов по прямому требованию инструкцией программы) также требует особых подходов к вопросам структуры программы. В первую очередь это связано с тем, какой метод распараллеливания будет выбран. Вопрос крайне субъективный и зависит от множества факторов. Здесь учитывается в первую очередь схема и скорость распараллеливания на заданной вычислительной установке. Мелкозернистое распараллеливание (уровень распараллеливания не выше инструкций/команд) требует более частого включения блока распараллеливания, чем при крупнозернистом распараллеливании (уровень процедур и выше). На сегодняшний день в большинстве случаев используется распараллеливание на уровне модулей, или внутрикомандное распараллеливание – декомпозиция аппаратными средствами одной команды на субкоманды и их распределение по специализированным подпроцессорам (векторно-конвейерные вычислители типа Cray и другие, или же аппаратное разделение машинных команд по специализированным процессорам (DSP процессоры, конвейерные процессоры). Однако, мелкозернистое распараллеливание свойственно системам с автоматическим распараллеливанием на уровне компиляторов языков высокого уровня или на уровне языков ассемблеров или автокодов.

Автоматическое распределение имеет две стилистики решения: статическую, по итогам компиляции исходного кода, и динамическую, с помощью средств выявления потенциально параллельно исполнимых фрагментов исполняемого кода и распределения этих фрагментов по процессирующим элементам. Динамическое распараллеливание позволяет более эффективно выявлять параллельно исполнимые компоненты, однако является крайне затратным по времени исполнения: как правило, ведется постоянный мониторинг линейных (или линеаризируемых) участков программ с целью выявления возможных одновременно выполнимых инструкций программы.

Еще один комплекс проблем связан с категорией события в параллельных вычислениях. Это – способы интеграции событийного поля в распределенных архитектурах, организация процесса реагирования на события.

Многообразие принципов, свойств и проблем параллельных вычислений дополняется вторичным слоем задач, каковыми являются средства разработки и отладки параллельных программ. Отметим основные.

ПРИНЦИПИАЛЬНЫЕ ПОЛОЖЕНИЯ.

CAPER (Caper) – язык параллельного программирования, название которого образовано аббревиатурой от dynamic Compilable, Asynchronous, Parallel Events Routines. Первые версии языка CAPER создавались в 1994 - 1997 гг. для IBM PC.

Язык CAPER создавался с целью разрешения следующих концептуальных положений программирования:

1. Структурированность программ с возможностью динамического самоизменения и самоорганизации.
2. Возможность динамической компиляции и исполнения программы, динамической компоновки исполнимого кода.
3. Параллельное исполнение элементов структуры программы с возможностью поддержания всех возможных схем инициирования параллельных вычислений.
4. Управление процессом параллельных вычислений на основе событий (программирование событиями).

Теоретические основы языка были заложены в [1-3], развиты в [4-6], а конкретные решения по алгоритмизации и программированию апробированы в [7, 8]. Вот некоторые тезисы концепции.

Программа вместе с архитектурами и ресурсами операционной системы и вычислительной установки единым образом должны быть нацелены на достижение функциональных целей вычисления. Данный тезис, по сути, является определением фундаментальной проблемы, т.к. здесь необходимо ответить на следующие вопросы:

- каким должен быть алгоритм решения задачи;
- как и какой язык программирования должен быть выбран для представления алгоритма;
- как должны быть задействованы ресурсы операционной системы и вычислительной установки в процессе реализации вычисления.

Поиск ответов на эти вопросы должен быть комплексным, отражающим все архитектурные особенности вычислителя, операционной системы, а также принципов, заложенных в алгоритм решения задачи и в выбираемый язык программирования. Иными словами, все решения должны быть согласованы в пределах единой идейной основы.

Существующие технологии программирования требуют интегрированного представления программы, всех ее ветвей, альтернатив, а, следовательно, в ходе ее исполнения загружены практически все ее структурные элементы, в том числе и те, которые могут не выполняться или выполняться крайне редко. В иных языках эта проблема решается за счет создания внутренних оверлейных структур без возможностей прямо управлять компонентами программы. Так или иначе, проблема динамической подгрузки нужных и удаления, соответственно, ненужных модулей программ, стилистика решения этой проблемы напрямую зависят от возможностей операционной системы, в среде которой выполняется программа.

А потому актуален тезис: программа должна быть адаптирована (а следовательно, видоизменена и скомпонована) к текущим потребностям вычислений.

Часто возникают ситуации, когда альтернатив дальнейшего хода вычислений достаточно много и естественным решением была бы генерация нужного фрагмента программы - исходного текста и машинных кодов. Т.е. проблема динамической компиляции и исполнения порождаемых фрагментов программы представляется также достаточно актуальной.

Другими словами, в качестве императива был принят следующий: все компоненты программы должны быть доступны целям реорганизации (заметим, что в [] и [] высказывалось более широкое толкование ресурсов и адаптации - к ресурсам относились все компоненты, определяющие понятие вычислительной установки, в том числе операционная среда, язык программирования, интерпретирующая машина и пр., а следовательно, требования адаптируемости относились и к операционной и языковой средам, процессу исполнения программ, не говоря уже о самих программах).

Реорганизация программ может осуществляться удалением собственных компонент, их изменением или же привнесением новых компонент извне. Внешние компоненты могут быть фрагментами исходного текста или объектными модулями.

Следующей целью при определении основания CAPER было создание такой структурной и процедурной организации программ, которая позволила бы эффективно охватить, по возможности, наиболее широкий спектр схем параллельных вычислений, не зависящих от возможностей операционных и архитектурных сред и систем. За основу была взята классификация Флинна [] в процедурной интерпретации: SPSD, SPMD, MPSD, MPMD (M – Multiple, S – Single, P – procedure, D – Data). Результатом в Caper стала емкая конструкция для представления и инициации запуска параллельных процедур всех представленных классов.

Для поддержания решений по реализации параллельных вычислений были определены принципы построения языка на основе комплекса виртуальных машин, реализующие различные схемы параллельного синхронного и асинхронного выполнения программ, разработаны механизмы параллельного выполнения программ, в том числе механизм т.н. покомандного параллелизма [], квантования времени, а также распределения параллельно выполняемых компонентов программ на множестве компьютеров.

Еще одно принципиальное свойство, реализованное в языке, относится к проблеме программирования реакций на асинхронные события, происходящие как "внутри" программы, так и "снаружи". Неэффективность в этом смысле большинства существующих практических языков достаточно очевидна - как правило, программисту приходится организовывать довольно рутинные процедуры отслеживания событий, которые загромождают программу, часто, весьма искусственны и субъективны, либо же опираться на возможности операционной системы, что непосредственно влияет на стиль программирования и делает программу архитектурно и системно связанной (не переносимой). Если же учесть интерактивный характер подавляющего количества программ, то оправдано желание иметь средство программирования, которое позволяло бы эффективно описывать множество событий и связанных с ними процедур, исполняемых при их возникновении.

Столь же эффективно в CAPER решается проблема обработки ошибок, происходящих в процессе вычисления: допускается анализ и исправление команд, целых фрагментов программы, приведших к ошибке.

Базирование Caper на системе виртуальных машин обеспечивает возможность переносимости программ, их транспортирование и распределение по различным вычислительным средам.

СТРУКТУРА ПРОГРАММ.

Одной из ключевых проблем при разработке языка являлся вопрос выбора теоретического носителя программы. Здесь учитывались такие содержательные требования как мобильность (транспортируемость) программ и программных модулей в среде распределенных вычислений, возможность модифицируемости программ. С целью разрешения данных свойств была выбрана категория блока – логической идентифицируемой единицей, объединяющей данные: как наборы команд, так и собственно наборы данных.

Атомарной единицей всякой программы на языке CAPER является команда. Программой называется упорядоченная совокупность команд и подпрограмм, где каждая подпрограмма - также совокупность команд и подпрограмм. Данное определение не вполне точно и будет уточнено далее по мере описания основных конструкций языка. В CAPER принят термин "блок" для обозначения как набора команд, так и наборов различных данных.

Итак, каждый блок может иметь входящие подблоки. Блоки бывают нескольких типов: COMMAND, IMAGE, DATA, ARRAY, TEXT. Первый из них - блок исполняемых команд, остальные типы предназначены для представления данных различного типа и будут описаны ниже.

Исполнение программ языка CAPER начинается с явно указываемого начального

списка команд. Такой список является стартовым и безусловно образует корневой блок с именем MAIN. Имя MAIN назначается компилятором языка. Любая последовательность команд может быть стартовой. Прагматика этого свойства будет рассмотрена ниже.

Мы намеренно дали сначала содержательное описание структуры программы, ибо здесь возможны различные математические модели таких объектов (в [1,2] основной конструкцией является комплект - аналог мультимножества). В данной реализации в качестве "несущей" конструкции для программ и данных взяты массивы.

В этой связи уточним: программой является массив, элементами которого являются исполнимые команды или подмассивы, которые в свою очередь сами являются массивами, содержащими команды или подмассивы, и т.д. (разворотом индукции). Всякий массив команд обладает логическим именем и набором атрибутов. Именно поэтому по аналогии с блочным конструированием вся совокупность названа блоком.

ЗНАКОВАЯ СИСТЕМА И ИДЕНТИФИКАТОРЫ

Знаковой системой CAPER является система знаков в стандарте ASCII. Соответственно, исходный текст программы CAPER должен быть подготовлен любым редактором, обеспечивающим такую кодировку.

Все идентификаторы (буквенно-цифровые последовательности, начинающиеся с буквы, причем к буквам по традиции относится и знак подчеркивания '_') ограничиваются 15-ью знаками. Это техническое ограничение, которое может быть изменено. Оно вовсе не означает, что вы не можете использовать идентификаторы большей длины - разрешается сколь угодно длинное написание, однако синтаксический анализатор компилятора CAPER проигнорирует все, что превышает 15 знаков. Тут необходим контроль со стороны программиста, ибо внешне разные идентификаторы, совпадающие в первых 15-ти знаках, вызовут, в частности, ошибку дублирования имен при определении переменной или блока.

В новой версии CAPER, как и раньше, нет чувствительности к регистру букв.

ФИЗИЧЕСКАЯ И ЛОГИЧЕСКАЯ СТРОКА ИСХОДНОГО ТЕКСТА.

В исходном тексте программы различаются логические и физические строки.

Физическая строка - строка текстового редактора, ограниченная знаком конца

строки - шестнадцатеричный код 0x0A (десятеричное 10); логическая строка - строка команды, которая может располагаться на нескольких физических строках посредством знака: => - знак продолжения логической строки, последний на физической строке, принимаемый к анализу, т.е. все знаки, расположенные после "=>" на физической строке, игнорируются.

В свою очередь, физическая строка может быть разбита на несколько логических посредством знака ";" (точка с запятой) - разделитель логических строк на одной физической.

Обычны средства комментирования:

```
*       - указатель логической строки комментария.
/*...*/  - скобки комментария - действует для физических строк:
          от строки, начинающейся с '/*' - в начале комментария,
          до строки, начинающейся с '*/' - в конце комментария.
//      - комментарий в строке после записи команды.
```

Кроме того возможно косвенное комментирование: после сочетания знаков ; * (точка с запятой и звездочка), или же после знака продолжения строки "=>".

Знак '*' в начале логической строки - комментарий

```
* комментарий
 var1 := var2 * var3 +  =>    комментарий
        var4 * var5

 var1 +=  var2  ; *   еще один комментарий
```

Весь спектр комментирования будет продемонстрирован в приводимых далее примерах.

МЕТКИ.

У автора языка нет известных предубеждений по использованию переходов по меткам - всякое средство программирования должно быть сбалансировано относительно целей программы и способностей самого программиста.

Довольно часто использование переходов по метке избавляет от создания надуманных циклов. Итак, команды программы могут быть помечены метками (перехода). Метки бывают двух типов – внутриблочные (локальные) и межблочные (глобальные). Метки определяются традиционным способом - замыканием идентификатора двоеточием (локальные) или двойным двоеточием (глобальные):

```
<имя метки>:   - определитель локальной метки.
```

<имя метки>:: - определитель глобальной метки.

КОМАНДЫ И ВЫРАЖЕНИЯ (общие положения).

Командой называется список арифметических или логических выражений, или оператор (компонента оператора) управления. Система выражений подобна системам выражений многих языков программирования, в частности, C. Список выражений определяется как перечисленные через запятую выражения:

<выражение 1> [, <выражение 2>] ... [, <выражение N>]

и является командой. Команда может быть пустой, если в логической строке отсутствует список выражений или оператор.

Синтаксис и семантика арифметических и логических выражений языка CAPER строятся традиционным для большинства практических языков способом.

В наборе операций - комплекс арифметических, логических операций.

```
public   var1, var2 :=  0
private  arr1 := array( 'I', 0 ,  2,3,4,5),  =>
         arr2 :=  array('B', null, 2000)
local    i:=1, j:=2, k:=4, p:=2

var1 := (var2:= arr1[i, j, k-i, k-j] + arr2[n] ) * 10.2'D , =>
        i := ( var1 + var2) /       =>
        (j := (var1 * var2)), n := var1 + var2  *=  n + k
```

В последних двух строках приведена одна команда как последовательность выражений.

Конечно же, чаще применимы простейшие:

```
var1 := 0             ;* ноль, по умолчанию  int
var2 := 10.2e3          ;* плавающая константа - по умолчанию double float
var3 := 10'F          ;* число, представляемое  в виде float
var4 := 1.252          ;* double float
var5 := 3'B           ;* число 3 в байте
var6 := 'A'           ;* ASCII-код знака
var7 := "Привет"       ;* указатель на литеральную строку
```

ФУНКЦИИ В CAPER (общие положения).

Как уже говорилось, в языке CAPER основным логическим и исполнимым по вызову структурным звеном программы является блок команд. Понятие функции присутствует как в смысле ресурса языковой среды (вызов виртуальной машины CAPER), так и определяемого программистов блока команд специального типа.

Функции CAPER, как правило, не требуют фиксированного числа параметров, а их выполнение часто зависит от внутренних типов входных параметров (см. ниже) и их количества. Некорректный вызов функции, связанный с ошибочным видом или количеством параметров вызовет внутреннее программное прерывание, которое однозначно идентифицирует вид и место ошибки. Такие ошибки могут быть исправлены по ходу вычисления.

Однако, блоки могут быть прототипированы. В этом случае контроль за количеством параметров и их типом в вызовах блоков ведет компилятор, который выдает предупреждения о несоответствиях.

Параметры функций передаются по значению.

КОМПИЛЯЦИЯ ПРОГРАММ И МАШИНА ЯЗЫКА

Представляемая реализация CAPER для сред имеет как возможность компиляции исходного текста программы и немедленного выполнения, так и компиляции и создания файла объектного кода CAPER. Компиляция программ на CAPER завершается созданием листинга. Однако, если осуществляется динамическая компиляция, т.е. компиляция исходного текста с целью запуска в процессе работы некоей задачи, то создание листинга может быть подавлено и включен режим внутреннего программного прерывания при возникновении ошибок. Это позволяет динамически исправлять исходный текст и корректировать создание исполнимого кода. Как правило, компилятор создает, по возможности, "разумный" код при ошибках, и вы вполне можете пустить на выполнение такой код, однако ошибки редко бывают настолько несущественными, чтобы не повлиять на ход вычисления, как правило, они приводят к внутренним ошибкам в процессе выполнения. Однако и здесь вы можете проанализировать ошибку и внести коррективы в ход вычисления. В этом смысле, идеология и базовая организация CAPER ориентирована на максимальную устойчивость и сохранение жизнеспособности вычислительного процесса.

В основе машины языка лежит т.н. виртуальный трехадресный процессор (трехадресная машина), который обладает системой внутренних команд. Конструкции исходного кода компилируются в комплексы команд именно этого процессора.

Виртуальный процессор обладает внутренней системой прерываний, в том числе

- подсистемой программных прерываний;
- подсистемой прерываний динамической компиляции, позволяющих обрабатывать

ошибки компиляции;
- подсистемой программно-событийных асинхронных прерываний;
- подсистемой прерываний от операционной системы.

Заметим, что наиболее стабильными в смысле интерпретации будут оставаться все подсистемы прерываний, кроме прерываний от операционной системы - компоненты, наиболее подверженной свойствам конкретной ОС. Далее средства взаимодействия с операционной системой будут описаны для сред, базирующихся на Win32.

Отмечу, что в перспективе не исключено появление версии CAPER, решенной на базе системы виртуальных процессоров (отчасти, это есть и в данной версии, однако выражено и исполнено это не достаточно прозрачно, скорее это виртуальные "полупроцессоры" обработки событий и управления параллелизмом).

Параллелизм вычислений рассматривается в двух ипостасях: концептуальной и интерпретируемой, т.е. речь идет о применяемых принципах параллельных вычислений и их конкретной реализации, зависящей от архитектуры вычислительной установки. Что касается концептуального начала, то здесь различаем собственно архитектуры (схемы) параллельных вычислений и способы их инициации/терминирования. Что касается архитектур, то Caper обеспечивается средствами программирования всех архитектур по Флинну (Flynn []): SPSD, SPMD, MPSD, MPMD. Процессы инициации/терминирования поддержаны

- асинхронной схемой, по которой инициация и исполнение нескольких параллельных процессов без приостановки инициирующего, при этом параллельно исполняемые процессы равноправны, самостоятельны, могут порождать новые процессы; в целом такой процесс может быть прерван любым из исполняемых процессов, либо же переведен в режим остановки с последующим восстановлением режима выполнения, либо в режим ожидания синхронизации - естественного завершения всех запущенных параллельно процедур;

- синхронной схемой, которая отличается от асинхронной тем, что инициирующий процесс приостанавливается на время исполнения параллельных процессов, управление в любом случае после завершения параллельных процессов перейдет к инициировавшему;

С точки же зрения интерпретации принципов параллелизма реализована псевдопараллельная схема для однопроцессорной машины.

Псевдопараллельное выполнение процедур заключается в принципе покомандного исполнении каждого блока. Этот принцип был введен уже в первой версии языка, здесь же он получил свое развитие. Данный принцип заключается в том, что фиксируется список параллельно запускаемых блоков, после чего машина языка после выполнения очередной команды или группы команд фиксированного блока переключается на очередную команду или группу команд из следующего блока списка. Такой блок становится текущим, его команда – текущей для виртуального процессора. Т.е. если список выполняемых блоков состоит из

block1, block2, ... , blockn.

то для каждого блока фиксирован индекс Ji - номер элемента массива (команда), который должен быть выполнен. Процедура же такова: в списке фиксируется очередной блок (с номером i), выбирается и реализуется blocki[Ji].

Список параллельно запущенных боков может быть изменен в любой момент времени подстартовкой новых параллельных блоков или принудительным (помимо естественного) завершением отдельных или всех процессов.

Кроме покомандного параллелизма может быть использовано квантование времени, позволяющее регулировать использование процессоров в случаях, когда количество запущенных параллельно процедур превышает количество вычислительных средств - процессоров и/или машин; здесь используется правило предоставления ресурсов на конечное время (квант времени).

Все параллельно запущенные блоки имеют собственное состояние, соответственно, их выполнение может регулироваться установкой определенных состояний. Подробнее об этом ниже.

Кроме перечисленных возможностей Caper обеспечивает возможности прямого регулирования распределением процессов по процессорам/вычислительным машинам.

Виртуальная машина языка имеет три режима функционирования. Режим машины устанавливается с помощью инструкций:

SET MACHINE PRIMARY – виртуальная машина первична по отношению к операционной системе в том смысле, что у виртуальной машины приоритет в регулировании ходом выполнения программы. Данный режим устанавливается по умолчанию.

SET MACHINE SECONDARY - виртуальная машина отдает возможность регулирования ходом выполнения программы операционной системе. Фактически, программа на Caper “засыпает” до возникновения любого сигнала к ней со стороны ОС.

SET MACHINE SOLO – виртуальная машина монополизирует ресурсы вычислителя (в первую очередь – процессора), блокируя, по возможности, любое вмешательство ОС в ее работу.

Данные команды дублируются функцией ВМ

SetMachineReg(<состояние>), где состояние задается числом:

0 – режим Primary;
1 – режим Secondary;
2 – режим Solo.

SetMachineReg возвращает число – индикатор предыдущего режима.

GetMachineReg() возвращает число – индикатор текущего режима.

ИСПОЛНЕНИЕ ПРОГРАММ CAPER (общие принципы).

Выполнение программы языка CAPER начинается с подачи на вход машины языка блока команд с именем MAIN. Обработка блока осуществляется последовательным выбором составляющих его команд и их исполнением. Управление последовательностью выбора команд внутри блока осуществляется с помощью традиционных конструкций типа

if <выражение 0>
. . .
[**elseif** <выражение 1>]
. . .
[**elseif** <выражение 2>]
. . .
[**else**]
. . .
endif

или

switch < выражение 0 >
[**case** [< выражение 1 > | < строковый или числовой литерал >]]
. . .
[**break**]
. . .
[**case** [< выражение N > | < строковый или числовой литерал >]]
. . .
[**break**]
. . .
ends

Все <выражения> в приводимых конструкциях являются арифметическими выражениями (их результатом является число). Ненулевое значение интерпретируется как истина, ноль – ложь.

В Caper представлены операторы итерирования (циклы):

while <выражение>
. . .
endw

и

repeat

. . .
until <выражение>

и команд условного и безусловного перехода

if <выражение> **go** <метка>

go <метка>

Изменить последовательность выполнения команд внутри циклов можно переходом вовне тела цикла, а также командами

continue
и
break

BREAK принуждает покинуть цикл или **switch** и перейти на первую команду, следующую за его телом (после закрывающей скобки конструкции: **endw**, **ends**, **until**).

Принудительный повтор можно осуществить как с помощью **CONTINUE**, так и с помощью команды перехода по метке.

Возможно обусловленное использование данных операторов:

IF <выражение> **BREAK**
IF <выражение> **CONTINUE**

Пример:

```
y := array( 'V', null , a1 )
y[1] := 1, y[2] := 1
i := 0
while ( i += 1 ) < 100
  if i == 50 continue
  y[ i ] := y[ i-1 ] + y[ i-2 ]
endw

repeat
  y[ i ] := y[ i-1 ] + y[ i-2 ]
  if y[ i ] > 10000000 break
  if y[ i ] < 0 go lab1
until ( y[ i ] > 100000000'G  || ( i -= ) > 0 )
. . .
lab1:
. . .
```

Выполнение каждого элемента блока - команды - называется шагом интерпретации-исполнения.

Стартовый блок MAIN не описывается явно, а порождается при компиляции и запуске на выполнение исходного текста программы, или же после загрузки первого объектного модуля CAPER. Впоследствии блок MAIN может быть уничтожен компиляцией нового текста программы или загрузкой в качестве MAIN нового объектного модуля.

Стартовым модулем с именем MAIN может стать любой объектный модуль Caper, также как и любой модуль исходного текста программы, поданного первым на компиляцию и исполнение.

КОМАНДЫ КОМПИЛЯТОРА

Набор команд компилятора - это команды, обычно называемые командами препроцессора. Однако, т.к. в CAPER такой набор довольно узок и они обрабатываются непосредственно компилятором, а не выделенным препроцессором, то и названы они соответственно.

Слабые возможности препроцессирования в CAPER имеют свое обоснование. В первую очередь потому, что CAPER обеспечивает возможность динамической компиляции и загрузки объектных модулей. Как в предыдущих версиях, так и здесь (т.е. выводы, приведенные далее, подкреплены прагматикой программирования в этих версиях) считается, что эти две возможности достаточны для подготовки альтернативных текстов программ, их объектных модулей, и в необходимый момент выбора того варианта, который необходим. Соответственно, макроопределения в CAPER имеют чисто утилитарный характер - как средство обозначения статичных (без переменных и генераций) числовых и строковых констант, фрагментов выражений.

Вот эти команды:

#macro <определяемое> <определитель>

<определяемое> - строка до первого пробела;
<определитель> - строка до конца логической строки.

Примеры:

```
#macro Constant_HEX_TEN    0x0A'B
#macro Const_BIN_TWELVE    0b00001100
#macro Literal_Str         "Literal string"
#macro GetKey              GetKeyb()

  . . .
OutText( Constant_HEX_TEN, Const_BIN_TWELVE, Literal_Str )
  . . .
```

Var := GetKey

Следующие две команды определяют стиль подготовки объектных модулей - фрагментирование программы по критическим секциям (фрагментам, ибо в CAPER - это не всегда процедурная единица или структурная единица). Выше уже описывался механизм работы виртуального процессора и машины CAPER в целом. Так вот, критическим фрагментом здесь называется последовательность команд CAPER, в ходе выполнения которых блокируется механизм переключения параллельных процессов, и, в специальных случаях, блокируется вызов событийных механизмов.

#flow [<шаг>]
. . .
#endflow

<шаг> - число, определяющее количество команд - длину критического фрагмента.
В случае отсутствия <шаг> компилятор сам устанавливает значение по умолчанию - 1.
#flow и #endflow фактически являются логическими скобками выделения критического фрагмента.

Примеры:

```
block BLOCK_NAME( fVar1, fVar2)
 . . .
#flow
 var1 := (var2+var3)/var3 ;* выполнение данных двух команд будет
   var2 := (var1+var3)/var3 ;* блокировать выполнение других параллельных
                    ;* процессов
#endflow
 . . .
endblock

#flow 3
*  Здесь вызов механизмов переключения процессов и обработки событий
*  будет осуществляться после каждых трех команд программы

 block BLOCK_NAME2
  . . .
 endblock

 block BLOCK_NAME3(fVar1, fVar2,fVar3)
  . . .
 endblock

#endflow
```

Отметим, что для принудительного срабатывания отключенных механизмов событий и переключателя процессов внутри критического фрагмента может использоваться команда PASS. Ее подробное описание смотрите ниже.

Команда

#include <путь и имя файла>

трактуется традиционно: указываемый файл будет вставлен в исходный текст программы, откомпилирован и выгружен из оперативной памяти.

#avoid и **#noavoid** - логические скобки, позволяющие обходить фрагмент программы, заключенный между данными скобками, при последовательном выполнении команд.

Пример:

```
var  := 10
var2 := 100

#avoid
  var  := 5
  var2 := 200
#noavoid

var += var2
```

Инструкции

```
var := 5
var2 := 200
```

не будет выполнена при последовательном прохождении по коду:
следом за var2 := 100 будет исполнено var += var2

В результате var будет содержать 110

ОПРЕДЕЛИТЕЛИ БЛОКОВ.

Как уже говорилось, блоки CAPER – основные средства определения процедурных единиц и групп данных, - могут иметь один из следующих типов: блоки команд, блоки данных, блоки образов, блоки массивов, блоки текстов. Фактически, категория блока нацелена на группирование и логическое представление данных, к которым отнесены и команды.

Блок команд - основная логически определяемая совокупность исполняемых команд.

Блок данных - средство логически представленной упорядоченной совокупности

данных - значений выражений.

Блок текста - логически представленная упорядоченная совокупность строк не интерпретированного текста.

Блок образа - средство хранения образов файлов.

Блок массива - средство создания статического массива в теле модуля.

Блок определяется посредством записи следующей формы:

block <имя блока> [[**static**] (<параметр1>, <параметр2>, . . . ,
<параметр N>)]
[**as** <тип блока>]
. . .
endblock

<имя блока> является идентификатором.
<параметр> - формальный параметр блока, получающий конкретное значение в момент вызова.

<тип блока> - **command** - блок команд
data - блок значений выражений.
image - блок образа,
text - блок текста,
array - блок массива.

Все блоки ограничиваются ключевым словом **endblock**.

Если <тип блока> опущен, то считается установленным тип **command**. Список параметров допустим только для блоков типа **command**.

Ключевое слово **static** предписывает компилятору создание мест под фактические параметры внутри модуля и может использоваться только в определении блоков команд. В случае отсутствия **static** параметры создаются динамически в процессе вычислений.

В процессе компиляции по определителю BLOCK имя блока регистрируется в списке имен загруженных блоков. По сути блок является массивом, и, соответственно, каждый элемент адресуем. Элементы блока адресуются традиционным для массивов образом

<имя блока>[i], где i - индекс массива

Кроме того, введена специальная функция выбора элемента блока:
GetElement(<имя блока>, <номер адресуемого элемента>)

Примеры:

block bl1 **as data**
10.25
“string”
‘A’
2006

endblock

Данный блок хранит литеральные значения. Выбор элементов блока может быть осуществлено посредством:

bl1[1] == 10.25
bl1[2] == “string”

и т.д., или же

GetElement(bl1, 1) == 10.25
GetElement(bl1, 2) == “string”

Блоки, определяемые как

func <имя блока>[[**static**] (<параметр1>, <параметр2>, . . . , <параметр N>)]
. . .
endfunc

в процессе выполнения блокируют выполнение программы в псевдопараллельном режиме (если программа находилась в таком режиме) до момента своего завершения.

Блоки, определяемые как

flick <имя блока>[[**static**] (<параметр1>, <параметр2>, . . . , <параметр N>)]
. . .
endflick

не только блокируют псевдопараллельное выполнение программ, но блокируют и обработку событий.

ПЕРЕМЕННЫЕ И МЕСТА: БАЗОВЫЕ ПОНЯТИЯ.

В Caper встроен широкий набор средств организации и управления данными. В основе этих средств понятия переменных и управляемых переменных, названных местами.

В целом, как и ранее, в CAPER не позволяется прямое адресование элементов памяти, что выражено в отсутствии средств создания переменных и компонент, напрямую работающих с оперативной памятью компьютера. В определенной степени - это задел для распределенной организации хранения информации.

Переменные и места в Caper не требуют явной типизации. Определенные таким образом переменные будут иметь тип того значения, которое будет им предписано.

Одновременно, переменные могут быть явно типизированы, причем описание типа переменной не означает автоматическое создание тела значения переменной для составных типов.

К базовым или внутренним типам (ибо тип переменной или места, как правило, не фигурирует явно в конструкциях языка) относятся

'D' - двойное плавающее число (double float) - 8 байт
'F' - плавающее число (float) - 4 байта
'G' - great (unsigned long) "большое" - 4/8 байт
'L' - long - 4/8 байт
'W' - word (unsigned integer) - 4 байта
'I' - integer - 4 байта
'H' - unsigned half-integer - 2 байта
'J' - signed half integer - 2 байта
'B' - байт (byte) - 1 байт
'C' - character - 1 байт
'S' - строка - количество знаков строки
+ 1 байт конца
'V' - переменная - 9 байт
'P' - место - 10 байт
'O' - блок/команда - 8 байт

Особое место в CAPER у переменных-мест. Места снабжены средствами управления: блокировки по чтению-записи, разблокировки. Контроль за использованием мест осуществляет виртуальный процессор: всякая попытка не только обращения к содержимому места, а и прочтения состояния места контролируется, и в определенных случаях приводит к формированию программного прерывания.

Изначально все места имеют статус "FREE". В процессе выполнения программы места могут получать следующие состояния:

WRITE_ONLY - только для записи
READ_ONLY - только для чтения
LOCKED - запрещено для использования.
FREE или **UNLOCKED** (синонимы) - свободно или разблокировано.

При этом изменить состояние места, так же как и всесторонне (без ограничений) использовать место может только блок (но не его подблоки), первым установивший ограничивающее состояние, естественно, до установления им состояния "FREE".

Доступ к статусу (проверка) из других блоков места осуществить с помощью операции '$':

$<имя места>

Фактически, это команда виртуальному процессору выбрать статус. Казалось бы, что нет особой необходимости в определении особой операции, и было бы

достаточным наличие функции среды, возвращающей состояние места. Однако, попытка использования места с установленным состоянием в параметре функции (параметры, как говорилось, передаются по значению) вызовет ошибку контроля виртуального процессора.

В блоках, не владеющих местом и пытающихся использовать их, придется зачастую использовать оператор ожидания необходимого состояния:

```
. . .
Wait $Place1 == FREE
Place1 := 10
```

Подробнее техника использования мест будет описана позже в разделах, посвященных событийным механизмам CAPER и технике компиляции программ.

ТИПИЗИРОВАННЫЕ ПЕРЕМЕННЫЕ.

В язык Caper 4 включены средства типизации переменных.
Тип переменной задается следующими дескрипторами типа:

‘<’[<свойство>] <тип> ‘>’

<свойство> ::= **size** | **memnum** | **array** | **as** | **implant**
<тип> ::= <базовый тип> | <порожденный тип>

Базовые типы определяются ключевыми словами:
<базовый тип> ::= **null** | **double** | **float** | **great** | **long** |
word | **int** | **half** | **shalf** | **byte** |
char | **addr** | **string** | **array** | **var** | **place** |
collect | **block**

<порожденный тип> ::= <имя структуры> | <имя блока прототипа> | <имя internal-блока>

Свойства типа определяют применение оператора к типу:

size - размер типа;
memnum – количество членов структуры (для простых типов не употребляется).
array - массив элементов типа <тип>
implant – используется только для порождаемых типов (структур, коллекций).
Включает весь состав членов <тип> в определяемый.
as - используется только в определении блока оператора **internal**
совместно с

array – блок-массив
data - блок данных
text - блок текста
image – блок-образ файла

Так, в результате выполнения выражения

var := <**size int**> + 10

переменная var получит значение 14, где 4 – размер **int**.
Описатель типизированной переменной (далее – дескриптор переменной):

‘<’ <тип> ‘>’ ‘[’ [числовой литерал], {, <числовой литерал> . . . } ‘]’
<имя переменной>

Примеры представления типизированных переменных:

<**int**> [10] arrOfInt - переменная, предназначенная для хранения указания на массив int, состоящий из 10-ти элементов;
<**block**> blVar - переменная, предназначенная для хранения указания блока;
<**double**> [] arrOfDouble – переменная, предназначенная для хранения указания на массив double (количество элементов не определено);
<**string**> [10, 20, 30] arrOfStrings – переменная, предназначенная для хранения указания на многомерный массив строк.

По переменным - указаниям на массивы с определенными границами могут быть построены реальные массивы (см. оператор **build**).

СТРУКТУРЫ И ПОРОЖДАЕМЫЕ ТИПЫ.

Структуры в Caper 4 определяются посредством следующей определяющей формы:

struct <имя структуры> { <описатель члена структуры>
[, < описатель члена структуры > . . .] }
[{ [<конструктор>] , [<деструктор>] }]

<имя структуры> ::= <идентификатор>
<описатель члена структуры> ::= <дескриптор переменной> [<флаг построения>]
<флаг построения> ::= **build** | **ref**

build используется для указания компилятору включить тело члена структуры в структуру; данный атрибут имеет смысл только для составных данных – структур, коллекций и массивов.
ref – прямое указание компилятору, что определяемый член структуры будет использован как ссылка. Если атрибут опущен, то член структуры интерпретируется как ссылка.
<конструктор> и <деструктор> - имена-указатели блоков, которые будут вызваны в момент создания и удаления структуры соответственно.

Так, в приведенном ниже примере

Пример:

```
struct ListValue { <word> point1, =>
                   <word> point2, =>
                   <block> interp =>
                 }

struct tagStru {                          =>
                <int> iVar,               =>  1
                <implant ListValue>,      =>  2
                <great> [20] myNumbers build,    =>  3
                <ListValue> [10] myFuncs ref,    =>  4
                <ListValue> [10] otherFuncs,     =>  5
                <ListValue> list build           =>  6
               }
```

элемент структуры tagStru с номером 1 (см. нумерацию в комментариях) является переменной простого типа (**build** не имеет смысла). Элемент с номером 2 фактически развертывается в три члена структуры <ListValue>. Здесь бессмысленны оба атрибута (и **build**, и **ref**). Все остальные элементы структуры (3-6) являются составными и могут иметь оба возможных атрибута, что и продемонстрировано.
Так, при генерации тела структуры в нем будут размещены тела массива 3 и структуры 6, в то время как под myFuncs и otherFuncs (элементы 4,5) будут выделены места как для обычных переменных, которые будут хранить указания на массивы структур.
Операция указания элемента структуры определяется символом ‘.’ (точка).
После определения переменной структуры

private <tagStru> stru1, <tagStru> [10] stru2

возможны

```
stru1.iVar := 10
stru1. point1 := stru1.iVar    ;* члены имплантированной структуры
stru1. point2 := stru1. point1 ;*
```

```
stru1.myNumbers[stru1.iVar ] := 20
```

Однако

```
stru1.myFuncs[1] := user_block
```

приведет к ошибке выполнения: массив myFuncs не создан, в то время как элемент структуры предназначен для хранения ссылки на массив.

```
stru1.list.point1 := 30
```

будет выполнен корректно: тело структуры ListValue будет встроено в tagStru и определено при построении.

Использование **implant** на первой позиции в определении структуры позволяет выравнивать новую определяемую структуру к определяющей:

```
struct tagStru1{ <int> var1, <byte> var2 }
struct tagStru2{ <implant tagStru1>, =>
            <float> flVar }

private <tagStru1> stru1, <tagStru2> stru2
build stru2

stru1 := stru2  ;* корректное присвоение
stru1.var1 := 10
stru1.var2 := 'A'
```

однако:

```
stru2 := stru1  ;* ошибочное присвоение
```

породит сообщение компилятора об ошибке.

К порожденным типам относятся и известные по Caper 3 коллекции. Коллекции являются совокупностями бестиповых переменных.

```
collection <имя коллекции> { <переменная коллекции>
                        [ , <переменная коллекции> . . . ] }
                    [ { [<конструктор>] , [<деструктор>] } ]
```

<имя коллекции> ::= <идентификатор>
<переменная коллекции> ::= <идентификатор>
<конструктор> и <деструктор> - то же, что и для структур.
Операцией указания элемента коллекции является точка. В то же время, элемент коллекции может быть описан строкой, хранящей символическое имя элемента коллекции, или же порядковым номером элемента в коллекции.

Пример:

```
collection tagColl { variable1, =>
                     variable2, =>
                     variable3  =>
                    }

private <tagColl> coll_1, <tagColl> coll_2, <string> str, <int> ii := 0

. . .

coll_1.”variable1” := “ABC”
coll_1.”variable2” := 2
coll_1.”variable3” := coll_2

str       := “variable2”
coll_2.str := coll_1.str + 10
coll_1.”variable3” и coll_2 указывают на одну и ту же коллекцию.

coll_1.1 := “CDE”    ;* указание члена коллекции порядковым номером
coll_1.(ii+1) := 10  ;* указание члена коллекции порядковым номером
```

Если номер указания элемента превышает количество элементов или меньше 1, то это вызовет ошибку в процессе исполнения программы. То же, если будет указано несуществующее имя элемента коллекции.

Коллекции в программных модулях сопровождаются своими описателями. Посредством функций виртуальной машины программист может получить данные описания и использовать их в процессе вычислений. Коллекции позволяют более гибкую организацию программирования. Так, если два вида коллекций имеют элементы с одинаковыми наименованиями (допустим, elem), то

```
collection tagColl_1 { num1, =>
                       elem, =>
                       num2  =>
                      }

collection tagColl_2 { string1, =>
                       elem, =>
                       string2  =>
                      }

private < tagColl_1> coll_1, < tagColl_2> coll_2, <int> test
. . .
if test
  coll := coll_1
else
```

```
  coll := coll_2
endif
coll.”elem” := 10
```

Динамически созданные коллекции могут быть удалены функцией

DelCollection(<указание коллекции>)

или оператором **delete** (см. ниже).

Возвращаясь к операторной интерпретации описателя типа отметим, что **memnum** типа корректен только для порожденных типов и определяет количество элементов структуры:

```
var := <memnum ListValue> * 2   ;* удвоенное количество элементов структуры
                                ;* равно 6-ти.

var := <memnum tagColl_2>       ;* var == 3
```

В Caper 4 особое значение уделено возможностям конструирования данных порожденных типов. Так, расширены возможности задания конструкторов и деструкторов (помимо вышеопределенной формы):

<конструктор> ::= .<идентификатор>
<деструктор> ::= .<идентификатор>

Данная форма может быть использована только в комплексе с инструкциями

#use module <имя модуля>
#use_remove module <имя модуля>
#use block <имя блока>
#use struct <имя переменной-структуры>
#use collect <имя переменной-коллекции>

предворяющими описание структур.
Завершение действия **#use** наступает по

#nouse

или при встрече новой инструкции **#use.**
Заданные конструкторы и деструкторы являются

1) при **#use module** <имя модуля> именами членов коллекции, которая возвращается загруженным и вызванным модулем <имя модуля>;
2) при **#use_remove module** - то же самое, что и **#use**, за исключением того, что

загруженный модуль будет автоматически выгружен после использования;

3) при **#use block** будет использован определенный в программе блок, при этом компилятор будет учитывать тип возвращаемого блоком значения, согласно которому конструктор и/или деструктор будут указаны
   членами структуры (если блок возвращает структуру) или коллекции (если блок возвращает коллекцию);
4) при **#use struct** – членами созданной структуры;
5) при **#use collect** – членами созданной коллекции.

Конструкторы и деструкторы могут быть смешанного типа: конструктор образован согласно **#use**, в то время как деструктор указывать на обычный блок или вовсе отсутствовать, и наоборот.

Примеры:

#use module sbWindow.obc

```
struct Window {                          =>
                  <int>    top,          =>
                  <int>    left,         =>
                  <int>    bottom,       =>
                  <int>    right,        =>
                  <word>   style,        =>
                  <word>   cursorID,     =>
                  <string> class,        =>
                  <string> name          =>
               }  {  .create,  .delete }
```

#nouse

build private <Window**>** myWindow(10, 20, 100, 200)

Здесь по **buld** будет

1) загружен модуль sbWindow.obc под именем sbWindow,
2) вызван блок sbWindow;
3) под результатом вызова будет пониматься коллекция; будет вызван член коллекции create с параметрами (10, 20, 100, 200).

Фактически, будет реализовано:

import sbWindow.obc **as** sbWindow
sbWindow().”create”(10, 20, 100, 200)

В то время как

delete myWindow

приведет к

```
  import sbWindow.obc as sbWindow
  sbWindow().”delete”()
```

```
#use module  sbWindow.obc

 struct Window { . . .  }  {  .create,  .delete }

#nouse
  . . .
build private <Window> myWindow( 10, 20, 100, 200 )
```

приведет к

```
  import sbWindow.obc as sbWindow
  sbWindow().”create”( 10, 20, 100, 200 )
  remove sbWindow
```

Использование блока выглядит следующим образом:

```
struct WinMethods{ <block> create,  =>
                     <block> delete,  =>
                   . . .
                  }

internal <WinMethods> WinLoader

#use block  WinLoader

struct Window { . . . } { .create, .delete }

#avoid
 flick WinLoader
   import sbWindow.obc as _sbWindow
   return _sbWindow ( MODULE_REMOVE, INTERFACE_STRU )
 endflick
#noavoid
```

Отметим еще раз, что WinLoader в данном случае должен быть определен программистом. Всякий раз, когда будет строится экземпляр структуры/коллекции

```
  build private <Window> myWindow( 10, 20, 100, 200 )
```

будет выполняться

```
  WinLoader().create( 10, 20, 100, 200 )
```

Для удаления объекта:

```
  WinLoader().delete()
```

  Далее,

```
private <WinMethodsl> winMeth
build  winMeth
winMeth.create := someBlock1
winMeth.delete := someBlock2

#use  struct  winMeth
struct Window { . . . } { .create, .delete }
#nouse
```

Тогда всякое построение типа

```
build private  <Window> myWin( 10, 20, 200, 300 )
```

будет приводить к вызову компоненты **create** структуры WinMethods:

```
   winMeth.create( 10, 20, 200, 300 )
```

Аналогично и для коллекций:

```
collection WinMethodsColl { create, delete, . . . }

private <WinMethodsColl> winMeth
build  winMeth
winMeth.”create” := someBlock1
winMeth.”delete” := someBlock2

#use  struct  winMeth
struct Window { . . . } { .create, .delete }
#nouse
```

И тогда -

```
  build private  <Window> myWin( 10, 20, 200, 300 )
```

будет приводить к вызову компоненты **create** коллекции WinMethodsColl:

```
  winMeth.”create( 10, 20, 200, 300 )
```

ОПРЕДЕЛЕНИЕ ПЕРЕМЕННЫХ И МЕСТ, ОБЛАСТИ ВИДИМОСТИ

Все переменные CAPER традиционно различаются по

- области видимости;
- времени создания;
- способу инициализации;
- возможности удаления.

По области видимости различаем глобальные, приватные и локальные переменные. Глобальные переменные видны на всех уровнях программы всеми ее компонентами.

Оператор определения и инициализации глобальных переменных:

PUBLIC <список переменных с инициализацией>

<список переменных с инициализацией> - это

<дескриптор переменной>[:= <выражение>] {[, <дескриптор переменной> [:=<выражение>]]}

public var1 := 10’B, var2, var3 := array('C' , 0, 100, 200)

Public-переменные статичны по своему характеру - память под переменные выделяется момент старта виртуальной машины и существует до конца ее работы. В момент определения в общем хранилище фиксируется факт создания переменной с заданным именем. Public-переменная может быть удалена, что реализуется удалением регистрации переменной с данным именем из хранилища.

Public-переменная может быть определена в любом месте любого блока или модуля. Возможность повторного определения допускается, если не установлено противное. Если повторное определение запрещено, в момент такой попытки вырабатывается внутреннее прерывание машины языка, позволяющее прикладной программе обработать данную ситуацию.

Приватные переменные видны только в пределах одного модуля и задаются аналогичными **Public** конструкциями:

PRIVATE <список переменных с инициализацией>
DEFINE <список переменных с инициализацией>

Приватные переменные, определенные в блоке, видны только в данном блоке и его подблоках. Private-переменные статичны по своему характеру: память им выделяется внутри модуля. Переменные данного типа становятся "живыми" в момент загрузки модуля в оперативную память. Соответственно, удаление переменных происходит в момент удаления всего модуля. Удаление отдельной

переменной не допускается.

```
private  pVar1 :=0, pVar2 :=1
 .   .   .
pVar1 += pVar2 *10

block BL1( Par1, Par2 )
  private  p1Var1 :=0, p1Var2 :=1, pVar1 := 2, pVar2
     .   .   .
  pVar1  -= p1Var2              ;* Здесь произошла локализация pVar1
                                ;* (определение, сделанное выше, не
    .   .   .                   ;* действует значение равно 1
    .   .   .                   ;* ( pVar1 := pVar1 - p1Var2 )
  block BL2( Par1, Par2 )
    private  p2Var1 :=0, p2Var2 :=1
    .   .   .
    p2Var2 := (  pVar2 :=100 )
  endblock
  p1Var1 := pVar2 / 10           ;* в pVar2  - значение 100, полученное в BL2
     .   .   .
endblock
pVar2 := 1 - pVar2              ;* здесь используется первое определение
```

Define-переменные формируются в момент выполнения одного из блоков модуля, в котором присутствует оператор DEFINE. DEFINE в модуле позволяет, фактически, создавать их реентерабельными, т.е. одни и те же блоки модуля могут быть вызваны разными параллельными ветвями вычислений. Для каждой ветви создается и используется собственный пул приватных переменных.

Локальные переменные видны только внутри одного блока (но не в подблоках!).

LOCAL <список переменных с инициализацией>

Локальные переменные создаются динамически после входа в блок и выполнения оператора LOCAL. Локальные переменные уничтожаются после выхода из блока.

Пример:

```
/*  Инициализация   arr1   3-х мерным  массивом , двухбайтные элементы которого
    заполняются числом 25
*/

  PUBLIC  arr1 := array(  'H', 25 ,  3, 4, 5 )

  BLOCK   block1(pVar1, pVar2)
      .  .  .
```

```
/*  var2  инициализируется  регионом - частью  массива  arr1   */

   private   var1, var2 := Region( arr1, null,  2, 2, 1, 2, 3, 4)
   local  LVar1 := 0, Lvar2 := 100'H
          .  .  .

   BLOCK block1(pVar1, pVar2)
    private  LVar1 := 0, a2 := "Строковый литерал CAPER"
    .  .  .
    Lvar1 := - a[2] ; * Здесь действует Lvar1, определенный внутри
                * блока
    arr1[ 1, 2, 1 ] :=  a2[ 4 ] ;* код четвертого символа строкового      =>
                          литерала будет присвоен двухбайтовому   =>
                          элементу массива
   ENDBLOCK

   Lvar2[ 1, 2, 2 ] := a2[ 3 ] ; * вызовет сообщение компилятора об ошибке, =>
                         ибо a2 не видна на этом уровне
     .  .  .
 ENDBLOCK
```

PUBLIC-переменные могут быть удалены оператором DELETE (см. далее).
PUBLIC-переменные, предваряемые спецификатором

INIT PUBLIC <список переменных с инициализацией>

запрещают повторную инициализацию переменной. Аналогично INIT действует на PRIVATE-переменные:

init private var:=0

осуществит инициализацию в момент первого выполнения, в последующие проходы присвоение игнорируется.

Язык CAPER позволяет написание реентерабельных программ, т.е. программ (блоков команд), которые могут быть вызваны в разные моменты времени несколькими параллельно исполняющимися блоками. С целью формирования уникального для каждого вызова поля локальных переменных необходимо использовать оператор

LOCAL <список переменных с инициализацией>

а для PRIVATE переменных - оператор

DEFINE <список переменных с инициализацией>

Как и ранее, к локальным переменным и входным параметрам блоков можно

получить доступ из других блоков, т.е. возможны динамические назначения извне значений таких переменных, отслеживание их значений.

Места определяются операторами PLACES, которые, как и PUBLIC, могут находиться в любом месте программы:

PLACES <список переменных с инициализацией>

Управление состоянием места может осуществляться только из блока, установившего состояние, отличное от FREE, или любым блоком в отношении места с состоянием FREE, с помощью функций

LockF(<состояние>, <адрес места> [{ , <адрес места> }])

WRITE_ONLY - только для записи
READ_ONLY - только для чтения
LOCKED - запрещено для использования.

или освобождение места - установление состояния FREE

UnlockF(<адрес места> [{ , <адрес места> }]) - разблокирование

перечисленных в параметрах мест.

WRITE_ONLY, **READ_ONLY**, **LOCKED, FREE** - макросы, которым сопоставлены

числа числовые значения.

Пользовательской программе разрешается устанавливать собственные состояния - числовые значения свыше 10 и менее 256 - с помощью функции LOCKF и ее параметром <состояние места>:

#**macro** MY_PLACE_STATUS 11

LockF(placename, MY_PLACE_STATUS)

ОБЪЯВЛЕНИЕ БЛОКОВ - ПРОТОТИПИРОВАНИЕ.

Имена блоков, определенных в программном модуле, имеют видимость в пределах всей программы. Для локализации поля видимости имен внутри модуля должна быть использована инструкция

internal <описатель блока> {, <описатель блока> }

<описатель блока> ::= [<описатель типа>] [<дескриптор массива>] <имя блока>
[([<дескриптор переменной>] {, < дескриптор переменной> })]

[<описатель типа>] [<дескриптор массива>] задают описание возвращаемого блоком значения.

Пример:

internal <int> bl_1(<**int**> y1, x1), =>
<**null**> bl_2(<ListValue> list, <tagStru> stru),
bl_3, <tagStru> [] bl_4 (<**char**> symbol)

В Caper 4 каждый блок, представленный в internal, интерпретируется как определение структуры и, тем самым, порождается новый тип. Имя блока интерпретируется как тэговое имя структуры. Компилятором Caper в такую структуру стандартно включаются элементы, описываемые как

<**block**> body
<тип> return

где <тип> - тип возвращаемого значения. Так, описанные выше блоки будут представляться структурами:

struct bl_1 { <**int**> return, =>
<**block**> body, =>
<**int**> y1, =>
<**int**> x1 =>
}

struct bl_2 { <**null**> return, =>
<**block**> body, =>
<ListValue> list, =>
<tagStru> stru =>
}

struct bl_3 { <**null**> return, =>
<**block**> body }

struct bl_4 { <tagStru> [] return, =>
<**block**> body, =>
<**char**> symbol }

Важным отличием от обычных структур является инициализация элемента body указанием блока. Элемент body может быть изменен в процессе выполнения программы.

Пример:

internal bl_examp
build private <bl_1> blk

blk.body := bl_examp

Следующая инструкция похожа на **internal**, однако служит несколько иным целям, а именно, созданию классов (типов) блоков. Данная инструкция не описывает реальные блоки:

prototype <описатель блока> {, <описатель блока> . . . }

В <описатель блока> задается имя не реального блока, а класса блоков. На данное имя может опираться реальное определение блока.

Использование прототипа блока осуществляется в операторе **internal**:

prototype <**int**> ADDITIVE (<**int**> x, y)

internal <**as** ADDITIVE> bl_1, <**as** ADDITIVE> bl_2

Тем самым bl_1 и bl_2 должны быть реально определены и, согласно прототипу, должны иметь по два целочисленных параметра и возвращать <int>.

Оператор **prototype** может быть использован для создания общих описаний блоков для множества модулей.

МАССИВЫ, ФАЙЛ-МАССИВЫ И РЕГИОНЫ

CAPER предоставляет определенное разнообразие в организации массивов данных. Массивы CAPER различаются по месту организации: в оперативной памяти или в файле; а так же по способу организации и времени жизни: динамически или статически.

Как правило, мы различаем описание переменной-массива и собственно его, массива, создание. Массивы могут быть описаны в private, local, public и созданы оператором build в необходимой точке программы:

Пример:

```
private <int> [100,200] arrInt,  =>
      <someStructure> [10, 20, 30 ] arrOfStructures, =>
      <someStructure> [ 3, 2, 2 ] refToArrOfStructures

. . .

build arrInt, arrOfStructures
```

В то же время, для переменных массивов без указания размерностей позволяется использовать оператор **build** только в параметризованной форме. Такие переменные служат также для получения указания на уже созданный массив и последующего их использования.

Пример:

```
private <int> [] aInt,  =>
      <someStructure> [] refToArr

build aInt          ;* вызовет ошибку компиляции
build aInt(10, 20)    ;* корректная форма

build refToArr       ;* вызовет ошибку компиляции
build refToArr(100)( “ABC”, 1, ‘A’ ) ;* корректная форма – будет создан
                           ;* одномерный массив структур, “ABC”, 1,
                           ;* ‘A’ – параметры конструктора структуры
```

Мы можем использовать объявленные переменные в следующих качествах:

после построения (см. выше)

```
  refToArr := arrOfStructures
  refToArr[ 1, 3, 7 ].element += 10  ;* element – определенная компонента
                            ;* структуры  someStructure

  aInt[10] := refToArr[ 1, 3, 7 ].element ** 2
```

Обращаем внимание на то, что в первом примере была объявлена переменная refToArrOfStructures, но собственно носитель массива построен не был. Данную переменную можно использовать для хранения указания создаваемых другими способами массивов:

```
  refToArrOfStructures := arrOfStructures
```

Так как размерности массива для refToArrOfStructures определены, то адресование

элементов массива будет формироваться компилятором на основе данных размерностей:

refToArrOfStructures[2, 2, 1] будет указывать на элемент
arrOfStructures[10, 1, 1]

Тем самым мы можем организовывать локализованное адресование фрагментов массивов. Для создания статичных массивов может быть использован оператор **static build** и блоки-массивы.

Динамические массивы могут быть созданы функцией виртуальной машины

array(< тип >, < инициализатор >, < количество элементов 1>, . . . ,
< количество элементов N >)

< тип > - тип элемента массива.
< инициализатор > - значение, которым инициализируется массив, или NULL.

Значение <инициализатора> должно быть согласовано с установленным типом. Так, если зафиксирован числовой тип, то инициализатором должно быть число.
Если определен строковый тип, то инициализатор должен быть строкой. В этом случае длина элемента массива определяется длиной инициализирующей строки, включая завершающий байт с 0x00. Кроме того, инициализатор может быть числом, и тогда это число - длина элемента в байтах.
Если типом элемента является "переменная" или "место", то длина определяется как длина переменной или места. На элементы массива данного типа распространяются все правила работы с переменными и местами.
< количество элементов > - количество элементов в координате массива.

Массивы уничтожаются функцией

DelArray(<указание массива>)

(о создании и удалении массивов с помощью операторов BUILD и DELETE см. ниже).

Примеры:

```
/* Будет создан неинициализированный пятимерный массив целых чисел со знаком
   длинной в слово.
*/
   var1 := array ( 'I', null , 1, 2, 3, 4, 5 )

/* Двухмерный массив беззнаковых полуслов, инициализированных числом 10 */
   var2 := array ( 'H', 10, 2, 3 )

/* Двухмерный массив мест. Каждый элемент массива находится в состоянии
```

```
  "свободен"
*/
  var3 := array ( 'P', 10, 2, 3 )

/* Массив строк, инициализированных  "ABC", длина каждого элемента равна 4 */
  var4 := array ( 'S', "ABC" , 3, 4, 5 )

/* Массив 20-ти байтовых элементов */
  var5 := array ( 'S', 20,   3, 4, 5 )
```

Статические массивы создаются блоками

block <имя блока> **as array**(<тип>, <инициализатор>, < количество 1>, ... ,
<количество N>)
endblock

Элементы массивов адресуются традиционным способом:

<имя переменной>[<элемент 1> , <элемент 2> , . . . , <элемент N>]

< элемент J > - выражение, задающее число. Каждая координата массива адресуется от 1, т.е. первым элементом массива для, например, var3 является
var4[1,1,1], вторым - var4[1,1,2], и т.д., поочередным увеличением координат до значения var4[3,4,5].

Так же адресуются и элементы блоков-массивов.

Если задание < элемент J > не является целым числом, то оно преобразуются в целочисленный тип, причем значение не контролируется на нахождение в диапазоне определения массива. Контроль осуществляется при вычислении абсолютного смещения элемента относительно начала массива. При превышении смещением общего числа элементов массива машиной CAPER выставляется состояние ошибки.

```
  var3[ 0 , 5 ]  ==  var3[ 1 , 2 ] == var3[ 2 , 0 ]
```

Заметим, что элементы массивов именно адресуются. Как правило, внешне это сводится к выбору значения адресуемого элемента, за исключением операции присваивания ":=", по которой осуществляется размещение значения в область элемента массива. Участие элемента массива в вызовах блоков так же сводится к размещению значений элементов массива. Однако в некоторых функциях среды в качестве отдельных параметров передаются именно адреса элементов массивов, а не их значения.

Для быстрого перемещения по массиву в CAPER введены две операции:

<> - смещения от элемента массива (левый операнд) на указываемое количество элементов (правый операнд).

```
  var := arr[5] <> 10
  var := arr[3] <> -3
```

Очевидно, что подобные операции чреваты выходом за границы массива с известными последствиям.

Создание строки (и массива одновременно) осуществляется функцией

String(<длина строки> [, <символ заполнения>])

<длина строки> - число; выделяется память <длина строки> + 1; последний байт заполняется нулем.
<символ заполнения> - байтовая переменная или литерал.

var := string(10, ‘a’)

Здесь десять байтов будут заполнены символом ‘a’, 11-ый бай – 0.

Удаление строки реализуется

DelString(<указание строки>)

Для переменной var, хранящей указание строки (см. выше), это

DelString(var)

В языке CAPER введено понятие файл-массива - инструмента управления файлами фиксированной длины таким же способом, как и массивами.

farray(< указание файла > , < тип > , < инициализатор >,
< количество элементов 1>, . . . , < количество элементов N >)

< указание файла > - строка указания файла (имя файла).

Все остальные параметры имеют то же значение, что и в функции array.

Поведение данной функции зависит от комбинации параметров. Обязательным является первый параметр. Если файл существует, то файл воспринимается как одномерный массив с длиной, равной длине файла и типом элемента 'B' (байт). В случае отсутствия файла farray при единственном параметре порождает состояние ошибки.

Добавление к параметрам указателя типа вызовет такую же интерпретацию с единственным отличием - типом элемента. Наличие инициализатора приведет к перезаписи содержимого файла инициализирующим значением.

Наконец, наличие значений измерений файл-массива интерпретируется следующим образом: если файл существует, то при наличие инициализатора та часть, которая охватывается измерениями массива, будет перезаписана, если же файла нет, то он создается с указанным именем и длиной, определенной размерностям, и инициализируется. Отметим, что файл-массив может охватывать не весь файл, а только его начальную часть.

Примеры:

```
  public  var1, var2, var3, var4

  var1 :=  farray(  "e:\CAPER3\Data\file1.arr" ) ;* файл
  "e:\CAPER3\Data\file1.arr"                  => должен существовать
  var2 :=  farray(  "e:\CAPER3\Data\file1.arr", 2 )

  private i := 9, j

  var1[ i ] := 'O',   var1[  i +1 ] := 'k'
*    var2[ 5 ]   -  2  байта,  содержащие  "Ok"
*    var2[ 5 ] [1]   == 'O,   var2[ 5 ] [ 2 ]  == 'k'
  var3 := array( 'B', 100 )

*    Значение в var3[  1 ]  ==  79  ( десятичное значение ASCII-кода =>
     латинской буквы 'O' )
  var3[ 1 ] := var2[ 5 ]
  var4 := farray(  "e:\CAPER3\Data\file2.arr" , 'H',  null , 640, 480 )

/*
    file2.arr  может  не существовать, тогда он будет создан размером
    2 * 307200 ;
    если файл длиннее, то файл-массивом адресуются первые 307200  полуслов,
    если короче, то при  адресовании более  "далекого"  элемента в файле
    произойдет ошибка в момент доступа к данному элементу.
*/
```

Присвоение элементу файл-массива означает запись в файл в соответствующую позицию. При необходимости управления доступом к файл массиву нужно использовать места:

```
  places  pLVar1, pLVar2
  pLVar1 := farray(  "e:\CAPER3\Data\file2.arr" , 'W',  null , 320, 240 )

  flock( @pLVar1,  READ_ONLY  )
    .  .  .
  pLVar1[ 1,1 ]  := 0    ;*   вызовет установку состояния ошибки.
```

Так же при

```
  flock( @pLVar1,  WRITE_ONLY  )
  intVar  := pLVar1[ 1,1 ]  ;*  есть блокировка по чтению
```

Однако ничто не запрещает создание другого файл-массива

```
  pLVar2  := farray(  "e:\CAPER3\Data\file2.arr" , 'W',  null , 320, 240 )
```

и манипуляцию его элементами без ограничений: pLVar2 - открытое место.

Особое место в системе массивов CAPER занимают массивы переменных и мест. Так, допустимы:

```
var1 := array('H', null, 20,30,40)
. . .
var2 := array( 'P',  null, 10, 20 )
var2[ 2, 3 ] := var1
pLVar2  := farray(  "e:\CAPER3\Data\file2.arr" , 'W',  null , 320, 240 )
var2[ 2, 4 ] := pLVar2
```

или непосредственно

```
var2[ 2, 4 ] := farray(  "e:\CAPER3\Data\file2.arr", 'W', null, 320, 240 )
```

Разрешено:

```
var2[ 2, 4 ] [  10 , 30 ]    - указывает  на элемент  файл-массива.
var2[  3 ,  4 ]  := getAddr(  "BlockName"  )
```

```
/*
   функция getaddr  возвращает внутренний адрес первой команды блока, что
   позволит инициировать вызов блока по данному адресу:
   do  &var2[  3 ,  4 ]( param1, param2 )
   или
   var := var2[ 3 , 4 ]
   do &var( param1, param2 )
*/
```

Массивы мест позволяют оперировать своими элементами так же, как и обычными местами:

```
lockf(  var[ 3 , 4 ] ,  READ_ONLY )
```

что не позволит использовать данный элемент по записи.

К статичным массивам относятся блоки всех типов (о них подробнее в описании определителей блоков).

Над всяким массивом может быть определен набор регионов - фрагментов массива с собственной относительной адресацией элементов.

```
Region( <указание массива>, <коорд. начала1>, <коорд. конца1>
      [{,<коорд. начала2>, <коорд. конца2>}]
    )
```

К функциям обслуживания массивов относится

ArrSize(<массив>), которая возвращает количество байтов в массиве <массив>.

```
  arrVar := array( 'I' , 0 , 100 )
   . . .
  count := ArrSize( arrVar ) ;* count == 400
```

ElemAsStr() - возвращает указатель на элемент любого массива как на элемент строки.

```
  arrVar := array( 'H' , 0 , 100, 20 )
   . . .
//  элементам массива присваиваются коды знаков:
//                  H e                  l l                  o!
   arrVar[20,3] := 0x4865, arrVar[20,4] := 0x6C6C, arrVar[20,5] := 0x6F21
   arrVar[20,6] := 0
   strPtr := ElemAsStr( arrVar[20, 3] )

//  Будет веведенно: Hello!
   outText( 20, 20, strPtr )
```

ДИНАМИЧЕСКОЕ И СТАТИЧЕСКОЕ ПОСТРОЕНИЕ И УДАЛЕНИЕ ПЕРЕМЕННЫХ

Создание структур и массивов осуществляется несколькими способами.
Один из основных – посредством оператора

BUILD <имя переменной> ([<список параметров>])] [: <вызов блока>]
{, <имя переменной> [([<список параметров>])] }

где <вызов блока> - выражение вызова блока. Переменные, перечисляемые в build, должны быть составными и определены.
<вызов блока> описывает вызов с параметрами. С точки зрения прагматики такие конструкции могут быть использованы для динамической инициализации создаваемой структурной переменной. В общем случае - для любых целей. Если же структура была определена с конструктором, то сначала будет запущен конструктор.

Пример:

internal <int> blk(<double> dbVar, <tagStru> pStru), =>
    <tagStru> onCreate(**<**tagStru> pStru), =>
    <null> onDelete(**<**tagStru> pStru)

private <tagStru> stru, <double> [100, 200] arrOfDouble, =>
    <blk> vBlock

. . .

build stru : onCreate, arrOfDouble, vBlock

Тело stru будет динамически создано, после чего будет вызван блок с именем onCreate.

Если stru имеет конструктор, то по

build stru(x, y, z) : onCreate(w)

будет создано тело структуры и вызван конструктор с параметрами x, y, z.

Мы можем совместить описание и создание структурных переменных с помощью

BUILD PRIVATE <список переменных с инициализацией>

build private <tagStru> stru(parm1, parm2, parm3) : onCreate(w), =>
        <double> [100, 200] arrOfDouble, =>
        <blk> vBlock := null

Аналогично для локальных переменных:

BUILD LOCAL <список переменных с инициализацией>

Динамически созданные переменные могут быть удалены с помощью оператора

DELETE <переменная> [: <вызов блока>] { , <переменная> : <вызов блока>] }

Прежде чем тело составной переменной будет удалено из памяти, будет вызван представленный по <вызов блока> блок. Если данная переменная имеет деструктор, то такой блок будет вызван до деструктора.

BUILD_BY <имя переменной>[([<список параметров>])]
    {, <имя переменной> [([<список параметров>])] }

аналогичен **BUILD**, за исключением того, что **build_by** применим только для структур и коллекций с конструкторами; тело структур и коллекций не создается – это возлагается на конструктор.

struct tagStru2 { . . . } { Create, Delete }

build_by private <tagStru2> stru(x, y) : AfterCreate(z)

ОПЕРАЦИИ CAPER

Ниже перечислены основные операции.

Арифметические операции:

" - " - вычитание и унарный минус;
" + " - сложение
" * " - умножение
" / " - деление
" % " - деление по модулю
"**" - возведение в степень

Все операции имеют традиционный смысл, (по "%", как и в C, такое деление дает остаток от деление двух целых чисел).

Операции сравнения и логические операции.

"<" - меньше
">" - больше
">=" - больше или равно
"<=" - меньше или равно
"!=" - не равно
"==" - равно

"&&" - логическое И
"||" - логическое ИЛИ
"!" - логическое НЕ

Двоичные операции

| - ИЛИ
& - И

>> - двоичный сдвиг вправо
<< - двоичный сдвиг влево

Операции присваивания

:= - простое присваивание правого операнда левому;
+= - присваивание левому операнду с предварительным суммированием левого и правого операндов;
-= - присваивание левому операнду с предварительным вычитанием из левого операнда правого;
*= - присваивание левому операнду с предварительным умножением левого и правого операндов;
/= - присваивание левому операнду с предварительным делением левого и правого операндов;
%= - присваивание левому операнду с предварительным делением по модулю левого операнда на правый;
&= - присваивание левому операнду с предварительным применением операции & (И) к левому и правому операндам;
|= - присваивание левому операнду с предварительным применением операции | (И) к левому и правому операндам.

Операции адресования:

& - указание блока или указание глобальной метки по адресу в переменной.
@ - взятие адреса переменной или места.
$ - операция взятия состояния места

=* - копирование составных типов данных (массивов, структур и коллекций).

Пример:

private <**int**> [100] arrArr1, <**int**> [200] arrArr2
arrArr2 =* arrArr1

arrArr1 будет скопирован полностью в первую половину arrArr2

private < tagStru > [200] arrStru1, < tagStru > [50] arrStru2, =>
    < tagStru > stru1, < tagStru > stru2

arrStru1 =* arrStru2
stru1 =* stru2

Корректность копирования составных данных контролируется компилятором.

КОНСТАНТЫ И СТРОКОВЫЕ ЛИТЕРАЛЫ

Представление констант CAPER носит обычный для большинства языков характер.

Тип Варианты

'D' - 123.45'D , -0.12345e3'D , 12345E-2'D
'F' - 123.45 , -0.12345e3 , 12345E-2
'G' - 1234567'G , 0'G , 10'G
'L' - -1234567'L , 0'L , -10'L
'W' - 12345'W , 0'W
'I" - -12345'I , -1'I, 10'I
'H' - 12345'H , 1'H , 10'H
'J' - -123'J , -1'J , 10'J
'B' - 123'B , 200'B , 255'B
'C' - 123'C , 127'C , -127'C

Символ " ' " (одинарная кавычка), следующая за цифровой частью литерала, означает наличие дескриптора типа, в котором должно быть представлено записанное число - одна из букв обозначения типа (см. выше).

Числовые литералы могут быть записаны и без дескриптора типа, и тогда компилятор CAPER представляет число типом "I", если только в записи числа нет признаков, требующих плавающего представления - десятичной точки и/или знака порядка числа с плавающей точкой.

Кроме того предоставляется возможность записывать двоичные и шестнадцатеричные константы:

двоичные -
0b00000100 - число 4
0b00000011 - число 3
0b00000010 - число 2
(все будут переведены в тип 'I')
0b0000000100000011'H - число 300 в 2-х байтах
0b00000011'B - число 3 в одном байте

и шестнадцатеричные -
0x01'B - число 1 в одном байте
0x0A'B - число 10 в одном байте
0x000001F2 - число 498 в четырех байтах (тип 'I').

Традиционно представление и символьных констант:

```
'К'   - символ
"Строковый литерал"
```

```
var1 := 0x01
var2 := 12345  (т.е. 12345'I )
```

ФОРМИРОВАНИЕ И РАБОТА С ТИПАМИ ПЕРЕМЕННЫХ

Для бестиповых (полиморфных) переменных машина CAPER перед выполнением той или иной операции осуществляет выравнивание типов в сторону повышения, за исключением одного случая - присвоения элементу массива значения переменной. В этом случае, если текущий тип переменной выше типа элемента массива, то машина CAPER понижает тип значения переменной до типа элемента массива.

Примеры:

```
var1 := 10, var2 := 20.25
var1 += var2 ;* var1 == 30.25  и имеет тип float.
   . . .
arr1 := array( 'I', 1, 10, 20 )
   . . .
arr1[ 5, 6] := var1 ;* arr1[5,6] == 30 - тип var1 будет понижен до integer
```

Возможно прямое преобразование числового типа переменной с помощью функции

```
ConvertTo( <имя переменной>, <тип> ):

var1 := 12.575 ;* float
ConvertTo( @var1, 'I' ) ;* После чего значением var1 будет целое 12

SizeOf( <тип> ) - возвращает размер типа в байтах.

SizeOf( 'I' ) == 4
SizeOf( 'D' ) == 8
SizeOf( 'H' ) == 2
SizeOf( 'B' ) == 1
```

Для явно типизированных переменных выравнивание типов осуществляет компилятор. Невозможность приведения типов (например, попытка присвоения числовой переменной строки) сопровождается сообщением об ошибке компиляции.

private <**int**> iVar1, iVar2:=0, <**byte**> bVar1 := 1'B

iVar1 := bVar1 ;* присвоение с повышением типа (byte к int)
bVar1 := iVar2 ;* присвоение с понижением типа (int к byte)

СДВОЕННЫЕ И МНОЖЕСТВЕННЫЕ ЗНАЧЕНИЯ ПЕРЕМЕННЫХ

Память, выделяемая под переменные, позволяет определить свойство множественности значений переменной. Так, всякая переменная способна хранить два значения типа 'I', четыре значения типа 'H', восемь значений типа 'B'.

В целях выделения этих значений введены функции:

IntLeft(<выражение>) - возвращает левую составляющую из сдвоенного значения выражения: число типа типа 'I'.

IntRight(<выражение>) - возвращает правую составляющую из сдвоенного значения выражения: число типа типа 'I'.

aHalf(<выражение>, <индекс>) - возвращает значение полуслова с номером <индекс>, который должен быть равен 1, 2, 3 или 4.

aByte(<выражение>, <индекс>) - возвращает значение байта с номером <индекс>, который должен быть в пределах от 1 до 8.

ВЫЗОВЫ БЛОКОВ КОМАНД И ИНИЦИАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Ниже приводятся основные операторы управления запуском (DO-запись).

Форма 1 (запуск перечислением блоков с параметрами):
DO [**SEQ** | **SYNCH** | **ASYNCH**] bl_1,bl_2,...,bl_k

[**WITH** $quant_1$,$quant_2$,...,$quant_k$]
[**WITHIN** med_1,med_2,...,med_k]

Форма 2 (запуск перечислением массивов):

DO [**SEQ** | **SYNCH** | **ASYNCH**] [**ARRAY**] $aname_1$, $aname_2$,..., $aname_K$
[**WITH** $quant_1$,$quant_2$,...,$quant_k$]
[**WITHIN** med_1,med_2,...,med_k]

и обусловленного запуска для обеих форм:

IF <условие> <**DO**-запись>

bl1,bl2,...,blk - список дескрипторов вызова блоков команд, имеющие следующие возможные формы:

<указатель блока >[([<параметр 1>, <параметр 2>, . . . , <параметр N>])]

либо

(<указатель блока 1 > , <указатель блока 2 >, . . . , <указатель блока K >)
[([<параметр 1>, <параметр 2>, . . . , <параметр N>])]

В первом случае указывается указатель блока и, возможно, параметры вызова в скобках. Во втором случае в круглых скобках перечисляются имена боков, после чего в круглых же скобках - параметры. Эта запись предписывает запуск перечисленных блоков с общими параметрами.

<указатель блока> - либо имя блока, либо &<имя переменной> - указание блока через значение в переменной.

В форме 2 $aname_i$ (1<= i <= K) – имя переменной – массива, элементами которого являются структуры порожденного по описаниям в internal блоков или их прототипами.

quant1,quant2,...,quantk - список квантов времени в миллисекундах, выделяемых на исполнение блока - перечень числовых значений.

med1,med2,...,medk - список имен машин многомашинной ассоциации, на которой должны быть исполнены соответствующие блоки.

Опция SEQ требует последовательного исполнения списка блоков команд, т.е. перечисленные в списке блоки запускаются последовательно; выполнение процедуры (блока), инициировавшей данный вызов, приостанавливается до тех пор, пока список не будет исчерпан. Опции квантования и распределения по компьютерам (или процессорам) для вызова списка не действуют.

Опция SYNCH определяет параллельное исполнение блоков с ожиданием в вызывающем

(синхронный параллелизм).

Опция ASYNCH определяет параллельное выполнение как вызываемых блоков, так и вызвавшего (асинхронный параллелизм). В этом случае возможно установить ожидание события завершения в нужном месте с помощью оператора

WAIT <событие>

либо

WAIT IsSynch()

(подробнее см. ниже).

Для асинхронного вызова в определителе опции WITH должно указываться k+1 величин квантов времени, выделенных на каждую запускаемую процедуру; последим числом должна быть величина кванта, выделяемого для вызвавшего процесса.

DO ASYNCH $b_1,...,b_k$ **WITH** $q_1,...,q_{k+1}$

(q_{k+1} - квант времени для работы вызвавшего процесса).

Примеры:

```
        do Block1       ;* Пример 1

        do &var1( )     ;* Пример 2

********************      Пример 3      ********************

        do synch  Block1,   Block2(parm21, parm22),     =>
           Block3( parm31, , parm33 ), block4

********************      Пример 4      ********************
  do asynch ( Block1, Block2, &var1 ) (parm1, parm2),        =>
                Block3( parm31, , parm33 ) ,  =>
                block4

********************      Пример 5      ********************
  do SEQ Block1, Block2(parm1,  null , parm2), Block3
```

или

```
  do Block1, Block2(parm1,  null , parm2), Block3
```

Первые два примера описывают т.н. простой вызов блоков: блок запускается, вызвавший его блок приостанавливает свое выполнения до завершения вызванного. причем во втором примере блок запускается через его указание в переменной.

В третьем примере осуществляется синхронный запуск блоков: первый и четвертый блоки без параметров.

В четвертом примере асинхронно запускаются блоки Block1, Block2, &var1 (блок, адрес которого в переменной var1) с общими параметрами parm1 и parm2, а также блок Block3 с параметрами и block4. Одновременно продолжает выполняться вызвавший их блок.

В последнем примере выполняется последовательный запуск перечисленных блоков.

К уже исполняемым параллельным процессам, запущенным оператором DO, можно присоединить дополнительные процессы с помощью новой команды **DO**, т.е. всякий оператор DO, выполненный в ходе параллельного вычисления, добавляет в список параллельно выполняемых блоков новые.

Кроме того, режим ASYNCH более весом, чем SYNCH. Это выражается в том, что если после инициации режима SYNCH будет выполняться

DO ASYNCH . . .

то текущий режим будет изменен на ASYNCH, в то время как обратное не действует, т.е. вызов

DO SYNCH . . .

не приведет к изменению режима ASYNCH.

Механизм вызовов базируется на стековом управлении: информация о вызвавшем блоке команд записывается в стек, а после возвращения в него удаляется. Возможны прямые способы управления стеком.

Для обычного вызова блока в языке присутствует функция

CallBlock(<указание блока>, <параметр 1> , . . . , <параметр N>)

где <указание блока> - это либо строка - имя блока, или адрес блока;
<параметр 1> - <параметр N> - параметры для вызываемого блока.

Возврат в вызвавший блок осуществляется по достижении конца блока или же оператором

RETURN [<выражение>] [**TO** <имя блока>]

Если фрагмент с TO опущен, то возврат осуществляется в вызвавший блок. В случае, если текущим (исполняемым) является MAIN, то осуществляется выход из программы. Если указан иной, чем вызвавший блок, то в стеке вызовов осуществляется поиск атрибутов указываемого блока. Если такие атрибуты существуют (возможно, их несколько), то возврат осуществляется по атрибутам последнего вызова. Поясним.

Пусть

{ a_1, a_2, ...,a_{i-1}, b, a_{i+1},..., b,...,b,..., a_n} - текущее состояние стека вызовов. Если в текущем блоке указан возврат в блок, атрибуты которого обозначены как b, то выбирается первый справа.

RETURN [<выражение>] - возврат в вызвавший блок.

Для параллельно запущенных процессов создается массив возвращаемых значений. Возвращаемое значение размещается в элемент массива, соответствующего номеру параллельно исполняемого блока. Т.е. в CAPER создается общее поле результатов различных вычислений.

Получение возвращаемого значение должно осуществляться с помощью функции RetValue([<номер процесса>]).

```
. . .
DO Block1( pVar1, pVar2 )
if retValue() > 0
  . . .
endif
. . .
```

В данном случае применимо и традиционное:

```
. . .
var := Block1( pVar1, pVar2 )


DO SYNCH bL1, bL2( pVar11, pVar12 ), Block1( pVar21, pVar22 )
lVar1 := RetValue( 2 ) + RetValue( 1 )
if ( lVar1 := RetValue( 3 ) ) > 10
  . . .
endif
*---------------------------------------
Block Block1( fVar1, fVar2 )
 . . .
 Return pVar1+pVar2
 . . .
EndBlock
*---------------------------------------
```

Кроме приведенной формы возврата существует оператор обусловленного возврата:

IF <выражение0> **RETURN** [<выражение1>] [**TO** <имя блока>]

Безусловный возврат в операционную среду с прерыванием всех текущих процессов и закрытием всех файлов осуществляется по команде

QUIT

Кроме того, виртуальной машиной будут выгружены из памяти все загруженные модули.

Кроме перечисленного, CAPER предоставляет возможность прямого управления стеком вызова процесса.

EraseStack(<указание блока> [, <стиль>])

<указание блока> - адрес или имя блока;
<стиль> - 0 или 1 (TO_LAST или TO_FIRST).
Функция позволяет сбросить стек вызовов блоков до последнего вызова указываемого блока (TO_LAST) или до первого его вызова TO_FIRST.

$a_1, a_2, ..., B, a_I, ... , B, ...$
F L

При <стиль> TO_LAST стек сбрасывается до B, помеченного L, при TO_FIRST - до B, помеченного F.

BreakStack() - сбрасывает стек полностью.

УПРАВЛЕНИЕ ПАРАЛЛЕЛЬНЫМИ ПРОЦЕССАМИ

Параллельные процессы в CAPER обеспечены разнообразными средствами управления и взаимного воздействия. Помимо тех очевидных средств, которые возможно организовать с помощью общих переменных и которые будут рассмотрены далее в примерах, имеются прямые средства воздействия на ход параллельных процессов.

В первую очередь - это возможности выборочной приостановки и деактивация параллельных процессов.

Временная приостановка процессов осуществляется оператором

STOP <номер процесса 1>, <номер процесса 2>, . . . , <номер процесса N>

или функцией

StopProcess(<номер процесса 1>, <номер процесса 2>, . . . ,
<номер процесса N>)

где <номер процесса i> - число – числовой идентификатор параллельно запущенного процесса.

Все процессы с перечисленными номерами останавливаются до того момента, пока один из параллельно работающих процессов не восстановит их активность командой

ACTIVATE <номер процесса 1>, <номер процесса 2>, . . . , <номер процесса N>

или функцией

ActivateProc([{ <номер процесса > }]), где <номер потока> - число.

Функция активизирует приостановленные ранее параллельные процессы; если параметры отсутствуют, то активизируются все параллельные ветки.

Остановленные процессы могут и не возобновиться. Если завершат работу все остальные, т.е. восстановление некому будет осуществлять, то машиной языка будет удалена вся информация, связанная с этими процессами, параметры, локальные переменные.

Функция

BreakProc(<номер процесса 1>, <номер процесса 2>, . . . ,
<номер процесса N>)

осуществляет выборочное принудительное завершение процессов.

Полное принудительное завершение параллельных процессов осуществляется командой

PARABREAK [**TO** <номер процесса>]

Данный оператор для синхронного и асинхронного режимов ведет себя по разному: **PARABREAK** в асинхронном режиме прерываются все процессы, за исключением вызвавшего. Однако, здесь существует опасность - вызвавший процесс может быть уже завершен. В этом случае возникнет внутреннее программное прерывание по ошибке.

PARABREAK TO <номер процесса> прервет все параллельные процессы, а управление будет передано процессу с указываемым номером; при этом и здесь возможна та же ошибка "неживого" процесса; реакция виртуальной машины CAPER та же - будет вызвана внутренняя ошибка.

Функция ParaBreak( ) прерывает параллельные процессы. Для синхронно вызванных процессов возврат осуществляется в вызвавший блок, для асинхронно вызванных процессов вычисление прерывается и осуществляется выход в систему. Для выхода в асинхронных процессах в нужный блок необходимо использовать команду
PARABREAK [**TO** <номер процесса>]

Пример:

```
 do synch bL1( p1), bL2, bL3( p31, p32, p33 )
 LVar1 := var1 + var2
   . . .
 block bL2
   . . .
*   Здесь параллельный процесс будет прерван =>
    и следующей командой будет первая после DO SYNCH
   parabreak
 endblock
```

В то же время

```
  do asynch bL1( p1), bL2, bL3( p31, p32, p33 )
  while (LVar1 := var1 + var2) < var3
    . . .
  endw
  . . .

  block bL2
    . . .
*   Здесь параллельный процесс будет прерван =>
    и следующей командой будет команда из bL3
    parabreak  to 3
    . . .
  endblock

  block bL3( fp31, fp32, fp33 )
     . . .
  endblock
*--------------------------------
```

В данном примере в bL2 асинхронный параллельный процесс прерывается, машина CAPER переводится в режим последовательного выполнения программы, активным остается bL3.

```
  do asynch bL1( p1 ), bL2, bL3( p31, p32, p33 )
  . . .

* Если выполнено условие, останавливаем два процесса
  if PublVar1 > var2
    StopProcess( 1, 2 )
  endif
```

CAPER позволяет контролировать ход управления параллельными вычислениями с помощью вызова блока команд в момент переключения с одного параллельного процесса на другой. В вызванном блоке можно переопределить схему выполнения параллельных процессов.

SetNext([<имя блока>]) - эквивалент SET NEXT в CAPER II - назначает блок, вызываемый в момент переключения с одной параллельной ветви на другую.

Если параметр отсутствует, то назначение снимается. Кроме того,

FreezeNext() "замораживает" запуск Next-блока, а
DefrNext() "размораживает" Next-блок.

NEXT-блоки должны быть откомпилированы в режиме flow, ибо в противном случае после вызова такого блока и выполнения его первой команды виртуальный процессор попытается переключить параллельную ветвь и вновь вызовет NEXT-блок, и т.д., что приведет к забиванию стека вызовов.

CAPER не контролирует стиль компиляции, а потому формально использование блока с разделителем команд не запрещено и может быть указано. Однако последствия такого использования лежат на программисте.

Определение номера текущего процесса осуществляется с помощью функции

CurrParall()

ParaCount() - возвращает число - количество параллельных процессов; в последовательном режиме возвращается 0.

Принудительное переключение на процесс с определенным номером может быть осуществлен с помощью команды

PASS [**TO** <арифметическое выражение>]

Если просто PASS, то управление передается очередному по порядку параллельному процессу. Если указан номер процесса - число, как значение арифметического выражения, - то машина CAPER переключит на параллельную ветвь с указанным номером.

С помощью функции

SetQuant(<квант>, <номер процесса> [{, <номер процесса i>}])

можно в ходе вычислений изменять кванты времени процессов.
<квант> - число - квант времени, который будет установлен процессу(ам) с номером (номерами) <номер процесса> (<номер процесса i>)
<номер процесса> - число - номер параллельного процесса.

АДРЕСОВАНИЕ И ИСПОЛНЕНИЕ ОТДЕЛЬНЫХ КОМАНД

CAPER позволяет адресовать отдельные команды блоков. С этой возможностью связаны несколько дополнительных нетрадиционных средств управления ходом вычислений и выполнения команд. Эти средства обеспечены, в основном, функциями среды и методами компиляции.

Address() возвращает указание (адрес) команды, в которой присутствует данная функция.

ElementCnt(<указание блока>) - возвращает количество элементов в блоке; если это блок команд, то количество команд блока, если блок данных, то

количество данных блока, если блок типа TEXT, то количество строк.

GetAddr(<имя блока>) - возвращает указание на начало блока.

GetElement(<указание блока>, <номер элемента>) - возвращает адрес элемента блока или сам элемент; <указание блока> - символическое имя блока или указатель блока.

см. в "Динамическая компиляция и загрузка"

СОБЫТИЯ И УПРАВЛЕНИЕ ИМИ

CAPER обладает очень мощными средствами инициации вычислений на основе асинхронных событий.

События в CAPER разделяются на несколько групп:

- программные события, или события, определяемые значением определенных арифметических и логических выражений;
- события виртуального процессора (машины CAPER);
- события операционной системы.

Программные события определяются командами

WHEN <событие> <DO-запись>

<событие> - любое логическое или арифметическое выражение.
<DO-запись> - см. описание оператора DO.

Данной командой предписывается исполнение DO-конструкции всякий раз в случаях, когда возникнет событие-условие. Варианты и возможности обработки событий мы рассмотрим чуть позже.

Атрибуты WHEN добавляются в список в том порядке, в котором выполняются операторы. При повторном прохождении через оператор WHEN не выполняется, т.е. описатель события регистрируется единственный раз.

С целью обслуживания WHEN введены следующие функции среды:

WHENIdent() - функция, возвращающая внутренний числовой идентификатор события

WHENCount() - возвращает текущее количество назначенных WHEN-событий

DeleteWHEN([<идентификатор события>]) - функция удаления события(й); удаляет назначение события, если указан идентификатор, и удаляет все назначения, если идентификатор опущен

FreezeWHEN([<идентификатор события>]) - "замораживает" событие с указываемым идентификатором; или "замораживает" все события, если идентификатор опущен. Аналогичный результат достигается использованием Set When Off.

DefreezeWhen([<идентификатор события>]) - обратная FreezeWHEN функция, "размораживающая" события - одно, если указан числовой идентификатор, и все, если нет.

Удаление событий приводит к снятию назначений в списке событий. При этом, при назначении новых событий они (новые) могут заиметь идентификатор, который был назначен удаленным. При ошибочном программировании все манипуляции с идентификатором события будут относиться к новому. Эта ситуация трактуется как ошибочная в силу того, что CAPER не дает средств регулирования назначения идентификаторов событий, и новое событий может принять необходимый идентификатор только случайно.

Замораживание события означает, что до момента "размораживания" данное событие не анализируется, т.е. попросту обходится машиной языка в списке событий.

Выполнение блока обработки события может осуществиться в любой точке любой параллельной ветви параллельных процессов. В момент инициации обрабатывающего блока событие замораживается. В момент завершения его обработки, если необходимо, событие должно быть "разморожено".

Техника "размораживания" событий дана в следующем примере:

```
. . .
WHEN var1 > var2 DO block1( pVar1, pVar2 )
var3 := WHENIdent()
. . .
block block1(fVar1, fVar2)
 . . .
 return defreezeWHEN( var3 )
endblock
```

Очень аккуратно можно использовать (как правило, это не годится) прямолинейный вариант:

```
block block1(fVar1, fVar2)
  . . .
  defreezeWHEN( var3 )
endblock
```

ибо defreezeWHEN(var3) - команда, после выполнения которой и до выполнения возврата по endblock, событие может снова выполниться, и block1 будет вызван до собственного завершения (последствия - может быть забит стек вызовов, который ограничен памятью компьютера, если только, конечно, событие не будет сброшено неявным образом - выражение в WHEN станет ложным).

Выполнение одного из событий не исключает срабатывание других событий в момент обработки первого. Для исключения помех в момент обработки события, CAPER позволяет создание блоков обработки в качестве критических секций или используя функции замораживания событий.

Кроме функций замораживания CAPER позволяет вовсе отключать механизм обработки WHEN-событий:

SET WHEN OFF

и включать

SET WHEN ON

В начальный момент работы программы, до первого

SET WHEN ON

события могут назначаться, но не будут анализироваться:

Пример:

```
WHEN pVar1 DO block1(pVar1, pVar2)
. . .
WHEN pVar2 != pVar3 DO block2
. . .

Set When On  ;*  Включается механизм обработки событий.
. . .
```

Однако, в случае

```
Set When On  ;*  Включается механизм обработки событий.
WHEN pVar1 DO block1(pVar1, pVar2)

*  Здесь механизм обработки событий CAPER включен, а следовательно, если
*  pVar1 > 0 , то немедленно будет запущен блок block1
. . .
WHEN pVar2 != pVar3 DO block2
. . .
```

Второй принципиальной конструкцией CAPER является оператор асинхронного ожидания события

WAIT <условие> [**BY** <имя блока>]

(вариант WAIT SYNCH в прошлой версии [3] заменен на WAIT IsSynch(); см. ниже).

Данный оператор переводит выполнение программы или ветви параллельных процессов в режим ожидания события до момента выполнения логического условия, причем процесс ожидания может сопровождаться выполнением блока, указываемого после спецификатора BY. Как правило, такой блок должен быть организован как критическая секция.

WAIT IsSynch() предназначен для ожидания завершения параллельно запущенных процессов командой **DO ASYNCH** блоков.

Комбинация операторов WHEN и WAIT позволяет определить очень комфортный стиль программирования:

Пример:

```
private pVar1 := 0, pVar2, pVar3 := 1
. . .
WHEN  pVar1 > pVar3 DO block1
WHEN  pVar2         DO block2
set when on
. . .
Wait 0 By Bl_Wait
. . .
Block Bl_Wait
  . . .
  pVar1 := . . .
  . . .
  pVar2 := . . .
  . . .
  pVar3 := . . .
  . . .
Endblock

Block block1 ;* блок обработки события pVar1 > pVar3 - истинно
  . . .
Endblock

Block block2 ;* блок обработки события pVar2 - истинно (pVar2 > 0)
  . . .
Endblock
```

Либо,

```
private pVar1 := 0, pVar2, pVar3 := 1
. . .
WHEN  pVar1 > pVar3 DO block1
WHEN  pVar2         DO block2
set when on
. . .
```

do aSynch block3(. . .), block4(. . .), block5(. . .)
. . .
Wait 0 **By** Bl_Wait
. . .

Здесь будут выполняться четыре параллельных процесса, один из которых (инициатор) будет переведен в режим асинхронного ожидания, причем, в силу тождественной "лжи" в условии, будет снят только экстраординарными мерами -

PARABREAK TO <номер процесса>

<номер процесса> должен быть меньше 4; либо с помощью

Quit

Третьим типом асинхронных событий являются события операционной системы.
Этот вид событий поддержан в данной версии CAPER набором функций среды, которые обеспечивают определение событий и регулирования их активности. Реализация событий ОС во многом зависит от специфики той или иной операционной системы, а потому описываемые ниже функции будут наиболее подвержены модификациям в интерпретации и исполнении. Здесь же они сориентированы на операционную среду Win 32.

События ОС классифицированы

- на события клавиатуры;
- события "мышки";
- события таймеров;
- служебные события ОС;
- программные события;
- исключительные события ОС.

В данном перечне не исключено появление в будущем событий ввода-вывода в целом, с классификацией этих событий.
Назначение блока-обработчика событий осуществляется функцией

SetOSEvent(<тип событий>, <имя блока обработки> [, <номер процесса>])

<тип событий> - числовой идентификатор типа
1 - события клавиатуры;
2 - события "мышки";
3 - события таймеров;
4 - служебные события ОС;
5 - исключительные события ОС.
0 - все события.

Программные события определяются отдельным набором функций: для назначения обработчиков исключительных ситуаций виртуального процессора -

SetProgErrBlock(<имя блока обработки>, [<имя файла объектного модуля>])

<имя блока обработки> - строковый литерал - имя блока CAPER, вызов которого будет осуществлен немедленно при возникновении события в любой точке параллельного или последовательного выполнения процессов.

Обработчик события будет запущен обычным для CAPER вызовом блока. Процесс, выполняемый в данный момент времени, будет приостановлен до завершения обработки.

Если речь идет о параллельном процессе, то будет приостановлена текущая ветвь, остальные будут продолжать выполняться, если только блок-обработчик не откомпилирован в стиле критического фрагмента (о стилистике компиляции - позже).

Событие будет обрабатываться сразу после назначения. Вызываемый блок должен быть определен с параметрами, количество которых зависит от типа обрабатываемого события и значения которых машина CAPER разместит в момент вызова. Параметры могут быть поименованы как угодно.

block EventsBlock(par1, par2, par3, par4, par5, par6, par7, par8)
. . .
endblock

и которые будут иметь описываемые далее значения:

для всех событий -
1-ый параметр - числовой идентификатор события.
2-ой параметр - внутренний (ОС) описатель события.

для событий клавиатуры:
3-ий параметр - числовой код клавиши (коды клавиш отличаются для нажатий, отжатий и пр.)
4-ый параметр - количество сигналов действий с клавишей ("нажатий").
6-ой параметр - код;
8-ой параметр - собственный идентификатор события;

для событий "мышки":
3-ий параметр - числовой код клавиши мышки (коды клавиш отличаются для нажатий, отжатий и пр.)
4-ый параметр - позиция курсора "мышки" по вертикали;
5-ый параметр - позиция курсора "мышки" по горизонтали;
6-ой параметр - расширенная информация о нажатих клавишах;
7-ой параметр - внутренний идентификатор события;
8-ой параметр - собственный идентификатор события;

для событий таймеров:
3-ий параметр - номер таймера;

К служебным событиям ОС отнесены события Win 32, связанные с окном Windows:
блоку обработки передается информация о закрытии, свертке и развертке окна,
перемещении, изменении размеров и др. -
3-ий параметр - код события окна;
4-ый параметр - lParam CALLBACK функций Win 32;
5-ый параметр - wParam CALLBACK функций Win 32.

По исключительным событиям Win 32 параметры имеют следующие значения:

1-ый параметр - числовой идентификатор события;
2-ой параметр - описатель события - код исключительной ситуации;
3-ий параметр - символическое имя блока, в момент выполнения которого возникла исключительная ситуация;
4-ый параметр - номер команды блока;
5-ый параметр - номер команды виртуального процессора.

Если назначен блок обработки всех событий, то в каждом конкретном случае значения параметров блока будут соответсвовать событиям.

Итак, если по SetOSEvent назначен блок для конкретного типа события, то данный вид событий будет обрабатывать именно данный блок. При этом, отдельные типы событий могут обрабатываться отдельными блоками. Виды событий перечислены в порядке их приоритетов, т.е. наименее приоритетными являются события от клавиатуры, наиболее приоритетными - события по исключительным ситуациям ОС. Все события устанавливаются и классифицируются внутри CAPER-машины. Запуск обрабатывающего блока будет осуществлен по наиболее приоритетному событию в момент внутренней синхронизации виртуального процессора после выполнения очередной команды программы. Остальные, менее приоритетные события не будут сбрасываться, если только это не будет сделано принудительно функцией ClearOSEvent.

ClearOSEvent([<тип событий>])

Если параметр опущен, NULL или 1 - сбрасываются все события, если указан идентификатор конкретного типа, то будет сброшено событие именнно данного типа, если числовой идентификатор не совпадает с перечисленными - вырабатывается программное аварийное событие, функция не осуществит сброса и вернет отрицательное числовое значение.

DelOSEvent([<тип событий>])

удаляет назначение обработчика событий. Как и для ClearOSEvent([<тип событий>]), действие функции распространяется на конкретный тип, либо на все, либо она возвращает отрицательное число по некорректному числу.

FreezeOSEV([<тип событий>])

"замораживает" указываемый тип событий, либо все, если <тип событий> опущен, NULL или 1, либо возвращает отрицательное число, если параметр не соответствует типу.

DefreezeOS([<тип событий>])

размораживает событие(я) указываемого типа.

CreateOSEv(<тип событий> [,<параметр 1>] [,<параметр 2>] . . . [,<параметр N>])

порождает событие указываемого типа и со значениями параметров для вызываемых блоков (см. выше): ни одного или то количество, которое будет необходимым.

В данную версию CAPER введены специальные функции, обеспечивающие "регионализацию" событий "мышки". Они позволяют асинхронным образом обрабатывать события "мыши", которые возникают в конкретных прямоугольных регионах нахождения ее курсора. По-прежнему в наименованиях всех функций присутствует инфикс OS, который подчеркивает, что данная функция в существенной мере зависит от реализации CAPER в среде конкретной операционной системе или среде.

SetOSEventRgn(<Y0>, <X0>, <Y1>, <X1>, <имя блока> [, <фильтр> [, <R-признак> [, <собственный признак>]]])

<Y0>, <X0> - координаты левого верхнего угла прямоугольника региона.
<Y1>, <X1> - координаты правого нижнего угла прямоугольника региона.
<имя блока> - имя блока обработки
<фильтр> - фильтр набора событий, на которые машина CAPER должна отреагировать вызовом блока; он может быть определен посредством булевого сложения (ИЛИ) перечисленных выше макроопределений или прямым заданием полуслова:

```
#macro LBUTTONDOWN    0x0001'H
#macro LBUTTONUP      0x0002'H
#macro LBUTTONDBLCLK  0x0004'H
#macro RBUTTONDOWN    0x0008'H
#macro RBUTTONUP      0x0010'H
#macro RBUTTONDBLCLK  0x0020'H
#macro MBUTTONDOWN    0x0040'H
#macro MBUTTONUP      0x0080'H
#macro MBUTTONDBLCLK  0x0100'H
```

<R-признак> - положительное значение или NULL означают, что номер параллельного процесса будет учтен событийным механизмом и указанный блок будет вызван именно из параллельной ветки, из которой было описано и установлено событие, 0 - обработчик события запускается из любого процесса.

<собственный признак> - значение (в том числе, коллекция, блок, массив), которое будет возвращено восьмым параметром в вызываемый на обработку блока.

Функция SetOSEventRgn возвращает внутренний числовой идентификатор регистрируемого обработчика событий, либо отрицательное число в случае неуспеха регистрации.

Пример определения фильтра:

```
SetOSEventRgn( 20, 30, 400, 500, "Block_Ev", LBUTTONDOWN | RBUTTONDOWN  )
```

или

```
SetOSEventRgn( 20, 30, 400, 500, "Block_Ev", 0x0009'H  )
```

CAPER не проверяет на пересечение определяемых регионов (соответственно, на вхождение региона в регион). Машина CAPER отстартует блок, если курсор находится в регионе и фильтр отвечает событию. При этом, при пересечении регионов активным окажется тот, фильтр которого будет отвечать событию:

```
SetOSEventRgn( 20, 30, 400, 500, "Block_1", LBUTTONDOWN | RBUTTONDOWN  )
SetOSEventRgn( 20, 30, 40, 80, "Block_2", LBUTTONUP | RBUTTONUP  )
```

Здесь нажатие клавиши при координатах курсора, к примеру, (30 , 30), будет вызван Block_1, в то время как отжатие приведет к вызову Block_2.
Если же

```
SetOSEventRgn( 20, 30, 400, 500, "Block_1", LBUTTONDOWN | RBUTTONDOWN  )
SetOSEventRgn( 20, 30, 40, 80, "Block_2", LBUTTONDOWN  )
```

всегда будет запускаться Block_1.

Если определено, к примеру,

```
do asynch block1, block2( p1, p2 ), block3( p3, p4 ), block4
. . .
block block2( p1, p2 )
  . . .
  SetOSEventRgn( 200, 300, 400, 500, "Block_1", LBUTTONDOWN, 1  )
  . . .
endblock

block3( p1, p2 )
  . . .
  SetOSEventRgn( 20, 30, 40, 50, "Block_2", LBUTTONDOWN, 1  )
  . . .
endblock
```

то Block_1 будет запущен из текущего для процесса с номером 2 блока, в то время как Block_2 будет отстартован из процесса (потока) с номером 3 (отметим, что всего параллельных процессов 5).

Если же задано

```
. . .
SetOSEventRgn( 200, 300, 400, 500, "Block_1", LBUTTONDOWN, null, 100  )
. . .
block_1 testBlock( par1, par2, par3, par4, par5, par6, par7, par8 )
 local TF := 0
 TF := ( par8 == 100 ) ;* TF == 1, т.к. par8 == 100
endblock
```

DelOSEventRgn([<идентификатор регистрации>])

функция удаления зарегистрированных событий.

<идентификатор регистрации> - числовой идентификатор, возвращаемый функцией SetOSEventRgn. Если он опущен или NULL, то удаляются все регистрации обработчиков событий в регионах. Если такой идентификатор указан, то описатель региона и обработчика удаляются из списка.

FreezeEventRgn([<идентификатор регистрации>])
и
DefreezeEventRg([<идентификатор регистрации>])

соответственно "замораживают" и "размораживают" обработку событий в регионе.

Аналогично событиям мышки в CAPER присутствуют средства назначения регионов событий, связанных с клавиатурой и местоположением каретки.

SetKeybRgn(<Y0>, <X0>, <Y1>, <X1>, <имя блока>
[, <фильтр> [, <R-признак> [, <собственный признак>]]]
)

<Y0>, <X0> - координаты левого верхнего угла прямоугольника региона.
<Y1>, <X1> - координаты правого нижнего угла прямоугольника региона.
<имя блока> - имя блока обработки
<фильтр> - фильтр набора клавишей, на которые машина CAPER должна отреагировать вызовом блока; он должен быть определен посредством указания элемента массива типа 'H', в котором перечислены коды клавиш.
<R-признак> - признак учета номера параллельного потока, в котором осуществлена установка данного региона. Если признак - ненулевое число или NULL, то номер потока учитывается, и при возникновении события машина CAPER переключит потоки и запустит указываемый блок обработки в данном потоке, если <R-признак> равен 0, то обработчик событий будет запущен в процессе с любым номером.

Функция возвращает внутренний номер регистрации региона.

<собственный признак> - число, которое будет возвращено восьмым параметром в вызываемый на обработку блока.

DelOSKeybRgn([<номер>]) - удаляет обработчик события с указываемым номером, или, если номер отсутствует, то все назначенные обработчики.

FreezeKeybRgn([<номер>]) - замораживает обработчик события с указываемым номером, или, если номер отсутствует, то все назначенные обработчики.

DeFreezeKeybRgn([<номер>]) - размораживает обработчик события с указываемым номером, или, если номер отсутствует, то все назначенные обработчики.

Что касается событий ОС, то CAPER, в отличие от WHEN-событий, не блокирует событие, вызвавшее инициацию блока обработки, а следовательно, это, при необходимости, нужно сделать в вызванном блоке. Сделано это из пока априорных предположений, что в обработке данных событий чаще придется выбирать стиль нейтрализации ненужных повторных вызовов блоков обработки событий - блокировкой, сбросом или отсутствием их, если обрабатывающий блок откомпилирован как критический фрагмент.

FreezeProcKey(<число>) – замораживает обработку событий от клавиатуры, причем если параметр отсутствует, то события клавиатуры замораживаются для всех событий, если задано число, то замораживаются только события, связанные с процессом, указываемым числом.

DelProcKey(<число>) - удаляет обработку событий для процесса <число>; опущенный параметр вызывает удаление обработки всех событий клавиатуры.

DeFreezeProcKey(<число>) – размораживает события согласно логике FreezeProcKey.

Примеры:

Здесь же заметим, что события, связанные с клавиатурой, "мышкой" или таймерами могут фиксироваться и обрабатываться с помощью WHEN и функций выбора координат курсора и клавиш клавиатуры, мышки, функций выбора времени и состояний таймеров, другие:

```
private CodeKey := 0

WHEN ( (CodeKey:=GetKeyb()) == 32 )  || ( CodeKey == 13 )  =>
DO EntryBlock

. . .

WHEN (msX :=MouseX()) >= 100 && msX <= 200 && =>
    (msY :=MouseY()) >= 200 && msY <= 400 && =>
    ( ms_Key := MouseKey()) == ms_LBut || ms_Key == ms_LButP ) =>
DO MouseEvBlock( ms_Key, msY, msX )

. . .
SetTimer( 3, 1000) ;* Устанавливаем таймер с номером 3 на 1000 миллисекунд
```

WHEN TimeIsZero(3) **Do** TimerClosed

Использование регионов событий влечет одну существенную проблему: реакцию на события, области которых пересекаются. С целью обеспечения предопределенной реакции в CAPER введены функции монополизации регионов

SetMouseMono(<Y0>, <X0>, <Y1>, <X1>, <идентификатор события>)
SetKeybMono(<Y0>, <X0>, <Y1>, <X1>, <идентификатор события>)

которые предопределяют обработку события с <идентификатор события> для мышки и клавиатуры соответственно, если оно произошло в заданной <Y0>, <X0>, <Y1>, <X1> области. Естественно, что события должны быть предварительно определены. Функции возвращают числовой идентификатор монопольного региона.

Для отмены монополизации региона должны использоваться функции

DelMouseMono([<идентификатор монополизированного региона>])
DelKeybMono([<идентификатор монополизированного региона>])

для мышки и клавиатуры соответственно. Опущенные параметры влекут отмену всех регионов.

FreezeProcRgn([<номер параллельного процесса>]) - замораживает все события процесса.

DeFreezeProcRgn([<номер параллельного процесса>]) - размораживает все события процесса.

Если параметр опущен, то замораживает все события.

DelProcRgn([<номер параллельного процесса>])

IsAlive(<номер процесса>) - живой ли процесс (1, если процесс активен, 0, если нет (пассивен или завершен))

GetProcessStat(<номер процесса>)- возвращает статус процесса: 1, если процесс активен, 0, если завершен, 3 - пассивен;

GetCurrStat() – возвращает статус текущего процесса;

<числовой идентификатор события> := SetUserEvent(<числовой идентификатор USEREVENT>)

DelUserEvent(<числовой идентификатор события>)

FreezeUserEvent(<числовой идентификатор события>)

DeFreezeUserEv(<числовой идентификатор события>)

InitUserEvent(<числовой идентификатор USEREVENT>) – инициирует пользовательское событие.

<числовой идентификатор USEREVENT> := IsUserEvent()

ClearUserEvent() - обнуляет USEREVENT

ДИНАМИЧЕСКАЯ КОМПИЛЯЦИЯ, ЗАГРУЗКА И ВЫГРУЗКА.

CAPER обладает средствами динамической компиляции файлов исходных текстов программ, средствами загрузки объектных программных модулей и их удаления.

Компиляция исходных текстов позволяет, в частности, динамически формировать тексты программ и исполнять их.

Динамическая загрузка предварительно откомпилированных исходных текстов - объектных модулей CAPER - также позволяет динамически компоновать исполняемый код, естественно, без затрат на компиляцию.

Данные средства CAPER никак не зависят от средств операционной системы.

Что же касается возможностей оперирования общепринятыми библиотечными средствами, то, как и в случае с ОС-ориентированным событийным механизмом, в CAPER встроены функции манипулирования библиотеками ОС, которые могут быть заглушены в тех средах, которые не поддерживают библиотечную организацию объектных и исполнимых модулей.

В данной версии CAPER средства динамической компиляции представлены единственной функцией среды

CompileFile(<стиль>, < указание компилируемого текста >, <имя блока> [, <признак замещения> [, [<имя файла сохранения>] [, [<ключи>]]]])

- в отличие от предыдущей версии, в которой (см. []) был реализован оператор

COMPILE [**BLOCK**] <указание компилируемого текста> [**IN** <имя блока2>]

<указание компилируемого текста> - либо имя файла с возможным указанием пути к нему, либо указание компилируемой строки. Что именно должно быть откомпилировано (как понимать заданную строку: как файл или как строку исходного текста) определяется параметром <стиль>: если стиль равен 0, то компилируется файл, иначе указываемая строка.

<признак замещения> - если присутствует - числовой параметр, значение которого предопределяет замещение имеющегося блока - модуля с указанным именем, либо запрещение этого. Второе - по умолчанию, и если блок с указанным именем существует, то это вызовет программную ошибку.

<имя файла сохранения> - строка - имя файла сохранения.

<ключи> - строка ключей компиляции:

“-C” – откомпилировать;

“-E” – по результату компиляции создать исполнимый файл.

Данной функцией инициируется компиляция файла, и размещение откомпилированного кода в качестве модуля и блока с указанным именем.

CAPER обладает средствами динамической загрузки и удаление программных модулей.

LoadModule(<имя файла> , <имя блока> [, <признак замещения>])

<имя файла> - строка знаков с именем файла объектного модуля,

<имя блока> - строка знаков с именем, с которым модуль будет загружен.

осуществляет загрузку и привязку объектных модулей CAPER. Второй параметр определяет имя блока, которым является загруженный модуль. В принципе, один и тот же модуль может быть загружен многократно при условии, что он не содержит блоков. В противном случае, в момент загрузки машиной CAPER будет выработано аварийное программное прерывание - ошибка Загрузчика, связанное с дублированием имен блоков, и загрузка модуля не будет завершена.

Функцией

DeleteBlock(<имя блока>)

осуществляется удаление любого блока, в том числе и блока, образованного загруженным модулем.

import <указание файла> **as** <описание блока>

загружает модуль с внутренним <описание блока>. Один и тот же модуль может быть загружен многократно под разными именами.
<описание блока> - либо имя блока, либо, если задан прототип, то после ключевого слова **as** может указан прототип блока
‘<’ as <имя блока прототипа> ’>’ <имя блока>.

Пример:

```
prototype <int> [] LOADED_MODULE ( <int> parm1, <word> parm2 )

import d:\Caper_Examples\capPanels.obc as < as LOADED_MODULE > Panel1
import d:\Caper_Examples\capPanels.obc as < as LOADED_MODULE > Panel2
import Graph.obc as Graph
```

Далее вызов Panel1 должен осуществляться согласно описанию в прототипе:

```
build private <int> [] arrOfInt := Panel1( -10, 20 ), =>
        <int> [] arrOfInt2, <null> var

arrOfInt2 := Panel1( -11, 30 )
var := Graph()
```

Оператор

remove <имя блока>

выгружает загруженный ранее модуль.

remove me

средство самовыгружения модуля: выгружается модуль, в теле которого

выполнен данный оператор.

Пример:

Продолжая предыдущий пример

```
remove Panel1, Panel2
remove Graph
if arrOfInt[1] == 1 && arrOfInt2[1]
  delete arrOfInt, arrOfInt2
else
  remove me
endif
```

CAPER позволяет компилировать и исполнять отдельные команды:

CompileCommand (<строка> [, <элемента массива>]) -

функция, компилирующая строку. Если задан элемент массива, то он указывает область памяти, в которую будут размещены результаты компиляции. Возвращаемый тип значения - указатель команды. Если <элемента массива> не указан, то будет выделен собственный участок памяти. В этом случае его необходимо будет освободить командой DeleteCommand(<указатель команды>). Функция возвращает указатель на начало откомпилированных команд <указатель команды> и размер в байтах, занимаемых откомпилированными командами, который может быть получен с помощью IntRight.

DeleteCommand (<указатель команды>) – освобождает память, выделенную под команды динамически откомпилированной строки.

DoCommand (<указатель команды>) – инициирует выполнение откомпилированного с помощью CompileCommand кода.

РЕАЛЬНЫЕ ПРИМЕРЫ

На основе изложенного материала рассмотрим несколько примеров программирования в Caper.

Следующий пример демонстрирует возможности программирования с помощью оператора **WAIT.**

```
internal <null> f_sum
private <int> sum := 0

// в следующем операторе после очередной проверки условия
// вызывается процедура f_sum

wait sum == 30 by f_sum

// ожидание нажатия ESC
wait GetKeyb() == 27
return


// процедура инкремента переменной sum
flick f_sum
  sum += 1
endflick
```

В следующем примере определяются две процедуры, одна из которых перемещает “маленький шарик” вправо (Ping), другая – влево (Pong). Обе процедуры могут работать бесконечно (все преобразования расположены внутри бесконечного цикла без принудительного выхода). Возможность завершения программы обеспечивает процедура WaitExit, внутри которой ожидается нажатия клавиши ESC. Еще одна процедура обеспечивает возможность установки асинхронной паузы.

В программе (блок MAIN) стартуют параллельно 3 процедуры, причем Ping и Pong стартуют на общем поле параметров. Взаимодействие между Ping и Pong осуществляется через общее поле параметров. В любой момент времени для завершения работы программы может быть нажат ESC, на который отреагирует WaitExit.

Процедура Pause используется в Ping и Pong для организации пауз при рисовании “маленького шарика”.

```
#include caper_GDI.ch
#include caper_FRW.ch
#include capGeometry.def
#include capWindow.def

// объявления блоков
internal <int> Ping( <int> y, x ), <byte> Pong( <int> y, x ), =>
        <NULL> WaitExit(),  Pause( <word> tm_pouse )

// синхронный старт Ping, Pong и WaitExit
do synch ( Ping, Pong ) ( 100, 100 ), WaitExit
return
```

```
* Перемещает "маленький шарик" слева направо
block Ping ( y, x )
  while 1
    wait x == 100   ;* ожидает значение координаты по x
    repeat          ;* когда x станет равным 100, начнется выполнение цикла.
      outText( y, x, 'O',, 0x00FF0000 ) ;* вывод “маленкого шара” на экран
      Pause( 1 )                        ;* однамиллисекундная пауза
      outText( y, x, '  ',, 0 )         ;* затирание “шарика” на экране
      x += 1        ;* инкремент x (двигаемся вправо)
    until x >= 500  ;* до тех пор, пока x < 500
  endw
endblock

* Перемещает "маленький шарик" справа налево
block Pong ( y, x )
  while 1
    wait x == 500   ;* waits x-coordinate value
    repeat          ;* when x == 500 cycle will start.
      outText( y, x, 'O',, 0xFF )
      Pause( 1 )
      outText( y, x, '  ',, 0 )
      x -= 1        ;* x increamentation (moving to left)
    until x <= 100  ;* cycle works before x > 100
  endw
endblock

#macro K_ESC    27
// ожидает нажатия ESC
block WaitExit
  outText( 20, 20, "Press ESC to exit",, 0xFF )
  Wait GetKeyb() == K_ESC ;* ожидание ESC
  quit          ;* выйти из программы
endblock

// Данная процедура организует паузу в миллисекундах, количество
// которых указывается параметром
block Pause static ( msec )
local <word> time := GetTicks() ;* динамическое создание локальной переменной,
                                ;* которая хранит текущее время в миллисекундах
    wait GetTicks()- time >= msec ;* ожидание истечения времени, указанного в
                                  ;* параметре
endblock
```

ФУНКЦИИ УПРАВЛЕНИЯ

SetParams(
<имя блока>,
<номер параметра>,
<значение>
)
<имя блока> - строка - имя блока;
<номер параметра> - число - номер параметра;
<значение> - значение любого типа.
Функция позволяет установить значение параметра уже вызванного блока: указывается имя блока, который расположен в стеке вызовов или является текущим (в этом случае вместо имени - NULL), номер параметра и его значение.

SetParamsParall(
<номер потока>,
<имя блока>,
<номер параметра>,
<значение>
)
<номер потока> - число - номер потока;
<имя блока> - строка - имя блока;
<номер параметра> - номер параметра;
<значение> - значение любого типа.
Функция позволяет установить значение параметра вызванного блока в указываемом параллельном потоке: указывается имя блока, который расположен в стеке вызовов или является текущим (в этом случае вместо имени - NULL), номер параметра и его значение.

GetParamsAddr(<имя блока>, <номер параметра>)
возвращает текущее значение параметра с <номер параметра> вызванного блока с <имя блока>; если <имя блока> - NULL, то предполагается текущий блок.

GetPrlParamAddr(<номер потока>, <имя блока>, <номер параметра>)
<номер потока> - число - номер потока;
<имя блока> - строка-имя блока;
<номер параметра> - число-номер параметра.

возвращает текущее значение параметра с <номер параметра> вызванного блока с <имя блока> в параллельном потоке с <номер потока>; если <имя блока> - NULL, то предполагается текущий блок.

ОТЛАДКА ПРОГРАММ, ОТЛАДЧИКИ

Проблема отладки параллельных программ является одной из самых трудноразрешимых в области параллельных вычислений. Ключевой причиной этого является жесткая зависимость проблем сбора и отслеживания информации о параллельных вычислениях от тех моделей параллелизма, которые выбраны в качестве основы. Так, при разработке средств отладки необходимо учитывать (и учитывается, см. []):

- принципы распределения/интеграции событий в параллельной системе;
- принципы распределения/интеграции памяти (оперативной и внешней);
- принципы исполнения параллельных процессов, и в первую очередь, систему распределения процессов, принципы синхронизации.

Виртуальная машина Caper имеет возможность выполнения программы в отладочном режиме. Одновременно, в Caper заложен принцип выборочной (управляемой) отладки: используя команды препроцессирования компилятора

#debug
. . .
#nodebug

мы можем отметить фрагмент исходного кода, который будет откомпилирован для выполнения в режиме отладки. Отсутствие #nodebug приведет к компиляции для режима отладки всего кода, начинающегося непосредственно за #debug до конца модуля (компилятор сам порождает инструкцию #nodebug в конце модуля).

Принцип выполнения программы в режиме отладки заключается в вызове блока, определенного в качестве отладчика, каждый раз после выполнения команды, расположенной в отлаживаемом фрагменте программы. То есть в Caper процедура отладки назначается:

SetDebugger(<указание блока>, <идентификатор процесса>, <директория отладчика>, y0, x0, y1, x1)

<указание блока> - блок, вызываемый в качестве отладчика;
<идентификатор процесса> - числовой идентификатор параллельного процесса, из которого (будет использован стек вызова данного процесса) будет вызываться бок-отладчик. Данный параметр может быть опущен (или NULL), и тогда отладчик будет вызываться в текущем (активном на данный момент) процессе.
<директория отладчика> - директория, в которой находятся файлы отладчика.
y0, x0, y1, x1 – координаты “горячей области” окна программы. Клик мышки в данной области приведет к вызову блока – отладчика.

Процесс отладки может быть управляем:

SetDebugStatus(<режим>)

где <режим> - число, определяющее включение (режим = 1) режима отладки или его отключение (режим = 0).

Блок-отладчик должны иметь определенные параметры, значения которых ему будут переданы виртуальной машиной:

Параметр 1: число – вариант вызова; 2 – перерисовка окна отладчика; иное значение – генерация окна отладчика.
Параметр 2: указание блока, из которого вызывается отладчик;
Параметр 3: номер команды псевдоассемблера в данном блоке;
Параметр 4: символическое имя блока;
Параметр 5: адрес массива параметров текущего блока;
Параметр 6: адрес пула локальных параметров;
Параметр 7: числовой идентификатор блока;
Параметр 8: адрес стека вызовов текущего процесса;
Параметр 9: значение текущей позиции в стеке вызовов;
Параметр 10: указание строки исходного текста модуля, в котором расположен данный блок;
Параметр 11: смещение от начала исходного кода модуля (количество символов) к текущей исполняемой команде;
Параметр 12: длина записи команды;
Параметр 13: номер текущего параллельного процесса;
Параметр 14: имя файла – модуля исходного кода;
Параметр 15: номер предыдущей откомпилированной для отладки команды (в порядковом расположении, а не контексте выполнения);
Параметр 16: порядковый номер данной команды исходного текста в нумерации компилятора;
Параметр 17: код ошибки выполнения (если таковая случилась);
Параметр 18: уточненный код ошибки;
Параметр 19: код команды псевдоассемблера;
Параметр 20: тип первого операнда команды псевдоассемблера;
Параметр 21: базовая составляющая адреса первого операнда команды псевдоассемблера;
Параметр 22: индексная составляющая адреса первого операнда команды псевдоассемблера;
Параметр 23: тип второго операнда команды псевдоассемблера;
Параметр 24: базовая составляющая адреса второго операнда команды псевдоассемблера;
Параметр 25: индексная составляющая адреса второго операнда команды псевдоассемблера;
Параметр 26: указание переменной результата трехадресной команды псевдоассемблера.

Таким образом, отладка в Caper может быть организована следующим образом:

Пример:

Пусть задан фрагмент кода программы

```
#macro   DEBUGGER_ACTIVE   1
#macro   DEBUGGER_PASSIVE  0

#debug

* мы определили два варианта отладчиков, выбор которых регулируется параметром

  InitiateDebugger( 1 ) ;* включаем первый вариант отладчика

  private <int> sum1 := 0'F, <int> ind := 0

  while ( ind += 1 ) < 10
    sum += i
  endw

    InitiateDebugger( 2 ) ;* включаем второй вариант отладчика

  Sum /= Something()
  . . .
  InitiateDebugger( 3 ) ;* выключаем режим отладки (см. текст процедуры)
  . . .
#nodebug
  . . .
  return sum



/* Initiate debugger */
flick InitiateDebugger static ( variant )

    switch variant
  case 1
    SetDebugger( Debugger ) ;* остальные параметры функции могут быть опущены,
                    ;* в этом случае вызов будет осуществляться из
                    ;* текущего процесса, директорией отладчика является
                    ;* текущая директория, зона реакции на интерактивный
                    ;* клик тоже отсутствует.

    break

    case 2
* здесь определены директория отладчика и координаты зоны реакции на mouse click
    SetDebugger( Tracer, null, "c:\myDebugFolder\", 10, 20, 40, 60 )
    break
```

```
  case 3
   SetDebugStatus( DEBUGGER_PASSIVE ) ;* дезактивируем отладчик
   return
  ends

  SetDebugStatus( DEBUGGER_ACTIVE ) ;* прямо активируем отладчик
endflick


/* block of debugging: debugger */
block Debugger static ( style, cmndArray, curCmnd, bl_name, inputParams,     =>
                locals, blDescr, stack, stackCounter, source,        =>
                offset, lenght, paralllProcNum, srcFileName, srcTxt,  =>
                prevCmnd, txtStrNumb, errCode1, errCode2, opc, type1, =>
                base1, ind1, type2, base1, ind1, resVar )


  . . .
endblock

/* the second block of debugging */
block Tracer static ( style, cmndArray, curCmnd, bl_name, inputParams,       =>
                locals, blDescr, stack, stackCounter, source,        =>
                offset, lenght, paralllProcNum, srcFileName, srcTxt,  =>
                prevCmnd, txtStrNumb, errCode1, errCode2, opc, type1, =>
                base1, ind1, type2, base1, ind1, resVar )


  . . .
endblock


//
func Something
  . . .
endfunc
```

Итак, процедурой InitiateDebugger назначаются отладчики (в последнем случае вызова режим отладки будет выключен). После каждой команды, начиная с первого оператора **private,** будет вызываться процедура Debugger, которая будет получать
полную информацию о состоянии программы и виртуальной машины. Анализ и реакция по результатам анализа текущей команды и состояния программы в целом могут быть выражены в совершенно различных видах: мы можем отобразить результаты анализа в некотором виде, можем принять управляющие действия по функционированию программы, можем вести журнал трассировки и прочая, прочая, прочая.

После завершения цикла **while** будет включен отладчик Tracer, который может анализировать работу программы уже в другом (в отличии от Debugger) стиле. Следующие за #nodebug команды уже не будут выполняться в режиме отладки.

Следующие две функции виртуальной машины обеспечивают получение текущего статуса режима отладки

GetDebugStatus()

и получение текущих значений переменных программы в массивах:

GetDebuggedParm(<тип данных>, <адрес переменной1> , <адрес переменной2>)

GetDebuggedParm возвращает массив имен переменных в <адрес переменной 1> и их значений в
<адрес переменной 2>. Тип переменных – параметры, локальные переменные, private-переменные
и public-переменные указывается первым параметром.

```
#macro  DEBUG_PARAMETERS   0
#macro  DEBUG_LOCAL          1
#macro  DEBUG_PRIVATE       2
#macro  DEBUG_PUBLIC        3
```

private <**var**> [] varNames, <**var**> [] varValues

GetDebuggedParm(DEBUG_LOCAL, @varNames , @varValues)

varNames будет содержать массив строк - имен локальных переменных, varValues – массив их значений. Если локальные переменные не были определены, то обе переменные будут не определены (равны **NULL**).

Заметим, что нынешняя реализация Caper сопровождается набором написанных на нем утилитарных модулей, которые обеспечивают типовые решения по тем или иным функциям, а именно, созданы модули, обеспечивающие работу: с экраном, графические операции, операции с файлами, а также средства анализа и отображения ошибок и типовую отладку. Прагматика использования модуля отладки предполагает два стиля: загрузку модуля с определенного момента и сохранение его в памяти до завершения программы, и загрузку модуля с отладчиком только в нужных точках программы, анализ ее фрагмента и выгрузку после завершения отлаживаемого фрагмента.

Как уже отмечалось, отладка параллельных вычислений требует множества решений нетривиальных проблем, и в первую очередь, сопряжена с выбором той или иной модели. Так, согласно представленной схеме отладка ведется во вторичном режиме
(отладчик вызывается из активной прикладной программы), в то время как отладчик может доминировать, т.е. регулировать процесс выполнения программы. Одновременно, отладчик последовательно вызывается из текущей исполняемой процедуры. Однако, возможна схема,

когда отладчик является еще одним параллельным процессом, “подсматривающим” за другими.

Обсуждение возможностей реализации схем отладки является самостоятельным и очень объемным предметом и здесь опускается. Заметим только, что они реализуемы в Caper на основе тех средств, которые были представлены выше.

ОБРАБОТКА ОШИБОК ВЫПОЛНЕНИЯ

Для анализа ошибок выполнения программ возможно назначить блок обработки ошибок.

SetErrorBL(<указание блока обработки> [, <указание модуля> [, <режим>]])

<указание блока обработки> - указание блока, который будет анализировать ошибки.
<указание модуля> - указание модуля (возможно, вместе с путем к файлу), в котором содержится блок обработки.
<режим> - указывает вариант исполнения блока обработки:
#macro LOAD_IMMEDIATELY 1
#macro LOAD_WHEN_EROR 2

если режим равен LOAD_IMMEDIATELY, то модуль загружается в момент исполнения функции (собственно функцией SetErrorBl);
если LOAD_WHEN_EROR, то модуль загружается в момент возникновения ошибки.
<указание модуля> может быть опущено. В этом случае блок обработки ошибок должен находится в пределах видимости текущего (либо internal-блок внутри текущего модуля, либо public-модулем).

Отмена блока обработки ошибок осуществляется функцией

DelErrorBL()

СЛУЖЕБНЫЕ ФУНКЦИИ

Symbol(<параметр>) - преобразует параметр в тип 'C'

<параметр> не может быть указанием блока, команды, строки, т.е. не быть числом или кодом знака - элемента строки.

GetByAddr(<элемент массива>, <количество>)
<элемент массива> - указание элемента массива, начиная с которого выбранное количество байтов будет преобразовано в число.
<количество> - определяет количество байтов, которые будут переведены в целое число. Если

ShiftAddr(<указание элемента массива - строки>,
<тип>
<признак выровненности>,
<смещение>
)

<указание элемента массива> - указание элемента массива;
<тип> - тип данных, определяющий смещение;
<признак выровненности> - признак, определяющий стиль смещения;
<смещение> - размер смещения.

Функция работает следующим образом:
указатель на элемент массива изменяется на величину, равную <смещение> в длине того типа, который определен параметром <тип>.
При этом указатель выравнивается на длину типа, если <признак выровненности> - положительноен число. При иной установке указатель указывает на байт и смещение определяется в байтах (<смещение> * <длина типа в байтах >).

GetByType(<строковый массив>, <тип>) - возвращает число указанного типа прочтением из строкового массива последовательности байтов.

LastError() - возвращает код последней ошибки функций Windows.

СТРОКОВЫЕ ФУНКЦИИ

NToFS (<число>[, <база> [, <стиль заполнения>,
<длина> [, <символ заполнения>]]])

<число> - результат любого выражения, дающего число.
<стиль заполнения> - 0, положительное число, отрицательное число или NULL - если больше нуля, то число располагается слева, а заполнение осуществляется справа; если отрицательное, то заполнение слева, а расположение нумерала - справа; NULL эквивалентен 0.
<символ заполнения> - если задан, то заполнения осуществляются данным символом,

иначе - пробелом.

StrCpy(<строка 1>, <строка 2>) – копирует <строку 2> в <строку 1>.

AT(<строка> , <искомая подстрока>) - осуществляет поиск <искомая подстрока> в <строка>. Возвращает позицию, с которой <искомая подстрока> начинается в <строка>, если такое вхождение имеет место, иначе возвращаемое число отрицательно.

PADL(<исходная строка>, <результирующая строка>, <>, [, <знак-заполнитель>])

PADR(<исходная строка>, <результирующая строка>, <>, [, <знак-заполнитель>])

StrLen(<строка>) - длина строки

Symbol(<число>) - преобразует в тип байт

ElemAsStr(<выражение>) - преобразует значение <выражения> в строку. <Выражение> должно быть элементом массива, строкой или элементом блока типа DATA или IMAGE.

IsAlnum(<код символа>) - определяет, является ли <код символа> кодом буквы или цифры. Возвращает TRUE, если так, и FALSE, если иначе.

IsDigit(<код символа>) - определяет, является ли <код символа> кодом цифры. Возвращает TRUE, если так, и FALSE, если иначе.

IsAlpha(<код символа>) - определяет, является ли <код символа> кодом буквы. Возвращает TRUE, если так, и FALSE, если иначе.

StrnCpy(<строка 1>, <строка 2>, <количество>) копирует <количество> знаков строки 2 в строку 1. Если количество знаков меньше или равно длине <строка 2>, то в <строка 1> байт завершения строки не устанавливается, и наоборот, если имеет место обратное, строка 1 заполняется 0x00 в байтах, превышающих длину строки 2.

AtOneOf(<строка>, <символ> [{, <символ> }]) осуществляет поиск одного из перечисленных символов в указанной строке до байта конца строки 0x00. Возвращает номер позиции и код символа в интегрированной 'I'

pos := IntLeft(_capAtOneOf(...)) == _capAtOneOf(...) - <позиция символа>
или 0;
code := IntRight(pos) - код символа.

AtOneOfx0A(<строка>, <символ> [{, <символ> }]) осуществляет поиск одного из перечисленных символов в указанной строке до символа перевода каретки 0x0A.

Возвращает номер позиции и код символа в интегрированной 'I'

pos := IntLeft(_capAtOneOf(...)) == _capAtOneOf(...) - <позиция символа>
или 0;
code := IntRight(pos) - код символа.
Если ни один из перечисленных символов не найден, то в позиция будет равна 0, а вместо <код символа> будет стоять позиция 0x0A.

StoN(<строка>, <тип>) - возвращает результат преобразования содержимого строки в число заданного типа.

ЭКРАННЫЕ ФУНКЦИИ

Функции CAPER, обеспечивающие возможности отображения информации, сориентированы на Win32 GUI, хотя в целом я пытался сохранить стилистику CAPER предыдущих версий, которая, в свою очередь, опиралась на принципы, заложенные в BGI: инициализацию работы с экраном, базовые операции вывода на экран текста, пиксела, линий, прямоугольников, окружностей, взятия пиксела, региона, завершения работы с экраном. В среде MS Windows реализация более или менее внятной концепции работы с экраном - довольно серьезная проблема, если учесть "размазанность" графического интерфейса Win 32. И в определенной степени пришлось следовать этой "размазанности". Существенно, что в CAPER исключен многооконный стиль работы, т.е. активно всегда одно окно Windows. Это имеет свое принципиальное и техническое обоснование. Во-первых, преследуется цель создания языка, не зависящего от возможностей операционной системы, обладающего внятной концепцией работы с экраном (экранами), и в то же время, цель разработки данной версии - разработка в первую очередь эффективных базовых средств управления параллельными вычислениями. Т.е. данный вопрос оставлен на будущее.

Здесь же представлены функции привязки к графическим средствам Windows в однооконном режиме.

Создание окна Windows

```
VMCreateWindow(
        <строка имени класса окна>,
        <строка имени окна>,
        <тип окна>,
        <Y0>,
        <X0>,
        <Y1>,
        <X1>
         )
```

<Y0>, <X0>, <Y1>, <X1> - координаты окна.

Здесь и далее координаты определяются согласно программистским традициям с левого верхнего угла. Вторая пара координат указывает правый нижний угол.

Turn_Display(On/Off) -включает/выключает окно.
VMDestroyWin() - уничтожает окно.
ScrollWindow(
<DY>,
<DX>,
<Y10>, <X10>, <Y11>, <X11>,
<Y20>, <X20>, <Y21>, <X21>
)
<DY> - величина прокрутки по вертикали,
<DX> - величина прокрутки по горизонтали,
<Y10>, <X10>, <Y11>, <X11> - координаты прокручиваемого прямоугольника.
<Y20>, <X20>, <Y21>, <X21> - координаты clip-прямоугольника

прокручивает область окна, ограниченную координатами <Y10>, <X10>, <Y11>, <X11>, на <DY> и <DX>.
Биты, расположенные вовне прямоугольника втягиваются вовнутрь и отображаются.
Биты, выталкиваемые из прямоугольника вовне, не отображаются.

isClosed() - возвращает 1, если окно закрыто, и 0, если открыто.
GetDevCaps(<параметр>) - функция - прямой аналог функции GetDeviceCaps из Win32.

Виртуальной машиной Caper поддерживаются прочие функции управления окном согласно логике Windows, однако здесь они не представлены – концепция графики в Caper поддержана специальной виртуальной машиной.

Фонты и управление ими.

CreateFont - прямой аналог функции CreateFont в Win 32

CreateFont(<логическая высота фонта>,
<средняя логическая ширина знака>,
<угол испускания>,
<угол орентации базы>,
<ширина>,
<плотность фонта>,
<флаг "italic">,

```
        <флаг "underline">,
        <флаг "strikeout">,
        <идентификатор множества символов>,
        <точность вывода>,
        <точность урезания>,
        <качество вывода>,
        <шаг и семейство>,
        <имя гарнитуры шрифта>
      )
```

Подробное описание функции можно найти в любой документации по Win32 API.

```
fnt2 := CreateFont( 14,  8,  0,  0,  500 ,  1,  1,  0, =>
                RUSSIAN_CHARSET, OUT_DEFAULT_PRECIS, =>
                CLIP_DEFAULT_PRECIS, =>
                DEFAULT_QUALITY, =>
                DEFAULT_PITCH, =>
                "ARIAL NEW" =>
              )

FontDescr(
        <логическая высота фонта>,
        <средняя логическая ширина знака>,
        <угол испускания>,
        <угол орентации базы>,
        <ширин>,
        <плотность фонта>,
        <флаг "italic">,
        <флаг "underline">,
        <флаг "strikeout">,
        <идентификатор множества символов>,
        <точность вывода>,
        <точность урезания>,
        <качество вывода>,
        <шаг и семейство>,
        <имя гарнитуры шрифта>
      )
```

SetFontBy() - установка фонта с предварительно заполнеными атрибутами функцией FontDescr.

SelectFont(<идентификатор фонта>) - переключение фонта.
Простейший вывод на экран осуществляется функцией
  OutText(<коорд. Y>, <коорд. X>, <строка> [, <длина>, <цвет>, <фонт>])
<коорд. Y>, <коорд. X>

GetStringSize(<строка или символ>, <длина> [, <фонт>])
<строка или символ> - именно строка или символ,
<длина> - число - количество символов строки, размер в пикселах которых должен быть измерен; если NULL, то берется вся строка.
<фонт> - указывается идентификатор фонта.
Высота и ширина строки возвращается в пикселах. Раздельно выбираются с помощью IntLeft(), возвращающей ширину, и IntRight(), возвращающей высоту.

```
   x := GetStringSize( "Hello !", NULL, fnt )
   y := IntRight( x )
```
или
```
   y := IntRight( x := GetStringSize( " World ", NULL, fnt ) )
```

```
OutFormatString(
            <Y0>,
            <X0>,
            <строка>,
            <длина>,
            <цвет>,
            <фонт>
            [ { , <строка>,
                <длина>,
                <цвет>,
                <фонт>
              }
            ]
           )
```

выводит в координаты Y0, X0 набор строк, каждая из которых представлена собственным цветом и фонтом, как одну строку на экран. Высота такой строки равна максимальной высоте всех заданных фонтов.
Возвращает ширину в пикселах выводимой строки.

```
DrawFormatStr(
            <Y0>,
            <X0>,
            <Y1>,
            <X1>,
            <строка>,
            <длина>,
            <цвет>,
            <фонт>,
            <опции>
            [ { , <строка>,
```

```
            <длина>,
            <цвет>,
            <фонт>,
            <опции>
          }
        ]
    )
```

выводит в прямоугольник с координатами Y0, X0, Y1, X1 набор строк, каждая из которых представлена собственным цветом и фонтом, как одну строку на экран. <опции> определяют размещения фрагмента в собственной части области, начало которой определяется с конца предыдущей: к началу предыдущего прибавляется его длина + 1 пиксел.
Возвращает ширину в пикселах выведенной строки в целом.
Подробное описание опций вы найдете в любом описании функции DrawTextEx Win32 API. Здесь перечислены опции, макросы которых представлены в файле CAPER_GDI.ch :

```
DT_TOP
DT_LEFT
DT_CENTER
DT_RIGHT
DT_VCENTER
DT_BOTTOM
DT_WORDBREAK
DT_SINGLELINE
DT_EXPANDTABS
DT_TABSTOP
DT_NOCLIP
DT_EXTERNALLEADING
DT_CALCRECT
DT_NOPREFIX
DT_INTERNAL
DT_EDITCONTROL
DT_PATH_ELLIPSIS
DT_END_ELLIPSIS
DT_MODIFYSTRING
DT_RTLREADING
DT_WORD_ELLIPSIS
```

```
DrawFormatStrA(
        <Y0>,
```

```
    <X0>,
    <Y1>,
    <X1>,
    <строка>,
    <массив>
  )
```

функция подобна DrawFormatStr, за исключением того, что структурированный вывод указанной строки осуществляется посредством описателей в массиве <массив>. <Массив> должен быть организован как набор пятиэлементных совокупностей типа 'I' (к примеру, AR := array('I', 0, 20, 5)).

AR должен быть предварительно заполнен компонентами
<команда>, <строка>, <длина>, <цвет>, <фонт>

<команда> - число 1 или 99. Последнее указывает конец массива и инициирует
завершение функции.
<позиция> - число - позиция в <строка>, с которой выводится фрагмент
длиной <длина> с заданными атрибутами.
<длина> - длина выводимого фрагмента строки.
<цвет> - цвет вывода
<фонт> - идентификатор фонта.

Итак, функцияя выводит в прямоугольник с координатами Y0, X0, Y1, X1 структурированную массивом строку как одно целое. Каждый фрагмент строки представлен собственным цветом и фонтом.
Возвращает ширину в пикселах выведенной строки в целом.

SetTextColor(<цвет>) - установливает цвет текста и возвращает старый.

GetTextCol() - возвращает текущий цвет текста

CharWidth(<строка>) - возвращает ширину первого знака строки.

Примеры:

```
CharWidth( "ABC" ) вернет ширину в пикселах сивола 'A';
string := "ABC"
CharWidth( string[2] ) вернет ширину в пикселах сивола 'B';
```

"Штучки" Windows и управление ими

CreatePen(<стиль>, <ширина>, <цвет>) - устанавливает одну из "штучек" Windows - Pen - стиль, ширину и цвет.

CreateCaret(
<числовой идентификатор битмап>,
<высота>,
<ширина>
)
создает каретку.
<числовой идентификатор битмап> - либо идентификатор, полученный посредством CreateBitmap, либо NULL;
<высота> - число, определяющее высоту каретки,
<ширина> - число, определяющее ширину каретки.
Возвращает положительное число, если удачно, и 0, если нет.

ShowCaret() - отображает каретку на экран.

SetCaretPos(<вертикальная позиция>, <горизонтальная позиция>) - устанавливает каретку в позицию с заданными координатами.

DestrCaret() - разрушает каретку.
HideCaret() - выключает каретку.
GetCaretPos() - возвращает позицию каретки в составном виде двух целых типа 'I'. Должны отбираться в стиле

```
x := IntRight( y := GetCaretPos() )
```

SetCaretBlink(<время>) - устанавливает период мерцания каретки в миллисекундах.

GetCaretBlink() - возвращает текущий период мерцания каретки.
GetCurrentPEN() - возвращает числовой идентификатор текущего Pen.
GetCurrentBRUSH() - возвращает числовой идентификатор текущего Brush.
GetCurrentFont() - возвращает числовой идентификатор текущего фонта.

GetCurrentBM() - возвращает числовой идентификатор текущего битмапа.
GetPenAttr(<идентификатор Pen>, <номер атрибута>) возвращает значение атрибута, указанного числом <номер атрибута>: 1 - стиль, 2 - ширина, 3 - цвет.
SetPenCol(<идентификатор Pen>, <цвет>) - не реализована (только для NT).
SelectPen(<идентификатор Pen>) - переключение Pen.
SelectBrush(<идентификатор Brush>) - переключение Brush.
SerCursor(<идентификатор курсора>) - переключение курсора.
CreateCurs(
<horizontal position of hot spot>,
<vertical position of hot spot>,
cursor width,

cursor height,
pointer to AND bitmask array,
pointer to XOR bitmask array
) - создание курсора

DestrCurs(<идентификатор курсора>) - удаляет курсор.

SetCursPos(
< вертикальная позиция >,
< горизонтальная позиция >
) - устанавливает позицию курсора

GetCursPos(<идентификатор курсора>) - возвращает позицию курсора; выделить их можно фукцией IntLeft и IntRight:

```
Y_pos := GetCursPos( IdCurs )          - вертикальная позиция
Y1_pos := IntLeft( GetCursPos( IdCurs ) )
Y_pos == Y1_pos
X_pos := IntRight( Y_pos )  - горизонтальная позиция
```

т.е. оптимальным является следующий стиль:

```
X_pos := IntRight( Y_pos := GetCursPos( IdCurs ) )
```

GetCursY() - возвращает позицию курсора по Y

```
Y_pos := GetCursY()
```

GetCursX() - возвращает позицию курсора по X

```
X_pos := GetCursX()
```

ShowCursor(<cursor visibility flag>) - показать/погасить курсор;
<cursor visibility flag> - параметр фукнкции ShowCursor(TRUE | FALSE) Windows, устанавливаемый в CAPER положительным значением параметра функции (TRUE) или нулевым значением (FALSE).

ClipCursor(<Y0>, <X0>, <Y1>, <X1>) - ограничивает перемещение и расположение курсора в пределах заданной параметрами <Y0>, <X0>, <Y1>, <X1> прямоугольной области.

LoadCursor(<name string|cursor resource identifier>) - функция загрузки курсора как ресурса системы (Windows) - реализует функцию LoadCursor: параметры

```
IDC_ARROW
IDC_IBEAM
IDC_WAIT
IDC_CROSS
IDC_UPARROW
```

IDC_SIZE
IDC_ICON
IDC_SIZENWSE
IDC_SIZENESW
IDC_SIZEWE
IDC_SIZENS
IDC_SIZEALL
IDC_NO
IDC_APPSTARTING
IDC_HELP

определены в CAPER_GDI.ch

SetBruCol(<цвет>) - устанавливает цвет Brush

DeleteObj(<идентификатор ресурса>) – удаляет ресурс Windows - Brush, Pen, Font и пр.

CreateBrush(<стиль>, <ширина>, <цвет>) - устанавливает одну из "штучек"
Windows - Brush - стиль, ширину и цвет.

Стиль определен макросами в файле CAPER_GDI.ch и соответствует описанию стиля

BS_SOLID
BS_NULL
BS_HOLLOW
BS_HATCHED
BS_PATTERN
BS_INDEXED
BS_DIBPATTERN
BS_DIBPATTERNPT
BS_PATTERN8X8
BS_DIBPATTERN8X8
BS_MONOPATTERN

цвет

DIB_RGB_COLORS 0
DIB_PAL_COLORS 1

Функции рисования.

RGB(<красный>, <зеленый>, <голубой>) - преобразует
в четырехбайтовую величину цвета Windows: 0x00RRGGBB.

LineFromTo(<y0>, <x0>, <y1>, <x1> [, <цвет>]) – прорисовка линии заданного цвета.
MoveTo(<y0>, <x0>) – устанавливает текущую позицию начала рисования.
LineTo(<y0>, <x0>) – прорисовка линии от текущей стартовой позиции до указанных координат в текущем цвете.
SetPixel(<y0>, <x0>, <col>) – установка указываемого пиксела в цвет.
GetBkCol() - возвращает цвет фона.
GetPixel() - возвращает цвет пиксела.
Rectangle_Fin(<y0>, <x0>, <y1>, <x1> [, <цвет>]) - рисует прямоугольник по
с заливкой.
Rectangle(<y0>, <x0>, <y1>, <x1> [, <цвет>]) - рисует прямоугольник по
координатам указанного цвета без заливки.
SetColor(<цвет>) - установка текущего цвета прорисовки.
Ellips(<Y0>, <X0>, <Y1>, <X1> [, <цвет>]) - рисует эллипс в прямоугольнике,
ограниченном координатами <Y0>, <X0>, <Y1>, <X1>
FillRectangle(<Y0>, <X0>, <Y1>, <X1>, <идентификатор Brush>) - заливка
прямоугольника.
SetBkColor(<цвет>) - устанавливает цвет фона.
InvertRect(<Y0>, <X0>, <Y1>, <X1>) - инвертирует прямоугольную область
применением NOT к каждому биту цвета пиксела.

Загрузка и отображение Bitmap.

LoadNShow(<Y0>, <X0>, <Y1>, <X1>, <YY0>, <XX0>, <имя файла> [, <буфер>])
<Y0>, <X0>, <Y1>, <X1> - координаты прямоугольника, в который выводится
изображение,
<YY0>, <XX0> - координаты левого верхнего угла изображения, начиная с которого
выводится изображение (номер начальной сканируемой строки и номер пиксельной
колонки, с которых выводится прямоугольный фрагмент),
<имя файла> - имя файла изображения.
<буфер> - указание элемента массива, в который заполняется все (!)
изображение из файла, и из которого будет выведен отмеченный фрагмент;
если буфер не указан, то буфер будет сформирован внутри функции и освобожден
после вывода на экран.

GetIMGInfo(<указание элемента массива>,
<идентификационный номер параметра изображения>)

<указание элемента массива> - элемент массива, с которого начинается предварительно заполненное изображение.
<идентификационный номер параметра изображения> - номер параметра заголовка
изображения:
1 - biSize - размеры структуры
2 - biWidth - ширина битмапа в пикселах
3 - biHeight - высота битмапа в пикселах
4 - biPlanes - кол-во плоскостей = 1

5 - biBitCount - кол-во бит на описание цвета
6 - biCompression - BI_RGB 0L - без компрессии
7 - biSizeImage - при BI_RGB == 0
8 - biXPelsPerMeter - кол-во пикселов на метр - горизонт.
9 - biYPelsPerMeter - кол-во пикселов на метр - вертик.
10 - biClrUsed
11 - biClrImportant
21 - bmiColors[0].rgbBlue
22 - bmiColors[0].rgbGreen
23 - bmiColors[0].rgbRed
25 - 65536 * bmiColors[0].rgbBlue + 256 * bmiColors[0].rgbGreen + bmiColors[0].rgbRed
100 - указатель начала собственно изображения (битмапа).

ScaleBitmap(
<указатель элемента массива>,
<YD0>, <XD0>, <YD1>, <XD1>,
<YS0>, <XS0>, <YS1>, <XS1>
[, <растровая операция>]
)
отображает смасштабированный битмап из области битмапа, указанной <указатель элемента массива>, ограниченной <YS0>, <XS0>, <YS1>, <XS1> в область, ограниченную координатами <YD0>, <XD0>, <YD1>, <XD1> с применением битовой операции <растровая операция> к обеим областям.

GetBmRegion(
<указатель элемента массива>,
<YD0>, <XD0>, <YD1>, <XD1>
)
размещает битмап экрана в массив, начиная с указываемого элемента массива.

ShowBitmap(<y0>, <x0>,
<y1>, <x1>,
<yy1>, <xx1>,
< array pointer>
)
<y0>, <x0> - координаты левого верхнего угла.
<y1>, <x1> - координаты правого нижнего угла.
<yy1>, <xx1> - координаты левого верхнего угла отображаемого изображения.

StretchBITMAP (<массив битмапа>, <y0>, <x0>, <y1>, <x1>,
<yy0>, <xx0>, <yy1>, <xx1> [, <операция>])

АРИФМЕТИЧЕСКИЕ ФУНКЦИИ

Abs(<число>) - возвращает абсолютное значение числа.
Min(<число1>, <число2>) - возвращает минимальное из двух чисел.
Max(<число1>, <число2>) - возвращает максимальное из двух чисел.
Sqrt(<число>) - возвращает значение квадратного корня.
Round(<число>) - возвращает округленное значение <числа>.
Log (<число>) – логарифм числа
Log10 (<число>) – десятичный логарифм числа
Pow (<основание>, <степень>) – степень числа (<основание>)

// LDiv (<числитель>, <знаменатель>) – возвращает составное значение из двух целых:
IntLeft(. . .) позволит вернуть из результата частное, IntRight(. . .) – остаток.

Div (<числитель>, <знаменатель>) – возвращает составное значение из двух целых:
IntLeft(. . .) позволит вернуть из результата частное, IntRight(. . .) – остаток.

FMod (<числитель>, <знаменатель>) – возвращает остаток от деления плавающих числителя и знаменателя.

Exp (<порядок>) – возвращает экспоненциальное значение.
Rand( ) – возвращает псевдо-случайное число.
SRand (<начальное число>) – устанавливает начальное число для генерации псевдо-случайных чисел.
Sin (<число>) – синус числа.
Sinh (<число>) – гиперболический синус числа.
Cos (<число>) – косинус числа.
Cosh (<число>) – гиперболический косинус числа.
Tan (<число>) – тангенс числа.
Tanh (<число>) – гиперболический тангенс числа.
ACos (<число>) – арккосинус числа.
ASin (<число>) – арксинус числа.
ATan (<число>) – арктангенс числа.
Ceil (<число>) – округление числа сверху.
Floor (<число>) – округление числа снизу.

РАБОТА С БИБЛИОТЕКАМИ ОС

LoadLIB(<имя библиотеки>, <зарезервировано>, <флаги>) – загружает

библиотеку <имя библиотеки>, <зарезервировано> - должно быть NULL,
<флаги> - см. LoadLibraryEx версий Windows.
CallLibFunc(<имя функции>, <параметры функции>) – не работает.

ФАЙЛОВЫЕ ФУНКЦИИ

CAPER предоставляет весь стандартный набор манипуляции файлами.

FOpen(<имя файла> [,<флаг> [<режим>]]) -
<имя файла> - строка с именем открываемого файла,
<флаг> - флаги варианта открытия файла, определенные макросами в файле CAPER_FRW.ch:

O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT
O_TRUNC
O_EXCL
O_TEXT
O_BINARY
O_NOINHERIT
O_TEMPORARY
O_SHORT_LIVED
O_SEQUENTIAL
O_RANDOM

<режим> - устанавливается только при O_CREAT и определяет момент закрытия нового файла. Режимы определены макросами:

S_IREAD
S_IWRITE

Функция возвращает число - заголовок файла или отрицательный код ошибки.

FCreate(<имя файла>, [<режим>])
<имя файла> - имя создаваемого файла,
<режим> - S_IREAD
S_IWRITE
возвращает число - заголовок фала или отрицательный код ошибки.

FClose(<заголовок файла>) возвращает 0, если закрытие файла произошло нормально, или отрицательное число - код ошибки.

FCommite(<заголовок фала>) - принуждает немедленно записать изменения файла, возвращает отрицательный код ошибки или 0.

FEOF(<заголовок фала>) возвращает 1, если указатель стоит в конце файла, 0 - если нет, и отрицательное число - код ошибки.

FSeek(<заголовок фала>, <смещение> [, <начальная позиция>])
<заголовок фала> - как и ранее, число - заголовок.
<смещение> - смещение от начальной позиции: отрицательное или положительное.
<начальная позиция> - SEEK_SET - начало файла;
SEEK_CUR - текущее положение указателя позиции;
SEEK_END - конец файла;
если последний параметр опущен, то принимается начало файла.
Макросы расположены в CAPER_FRW.ch

FRead(<заголовок файла>, <указатель элемента массива>, <количество>)
<указатель элемента массива> указывает место в массиве, с которого начинается побайтное заполнение из файла.
<количество> - количество байтов, должных быть считанными из файла. Если возникла ошибка чтения, то возвращается отрицательный код ошибки, иначе - количество считанных байтов.

FWrite(<заголовок файла>, <указатель элемента массива>, <количество>)
<указатель элемента массива> указывает место в массиве, с которого начинается запись в файл.
<количество> - количество байтов, должных быть записанными в файл.
Если возникла ошибка записи, то возвращается отрицательный код ошибки, иначе - количество записанных байтов.

LoadFile(<элемент массива>, <имя файла>
[,<с позиции> [, <количество> [, <тип>]]]
)

<указатель элемента массива> - указание элемента массива arr[i, ...] - начиная с которого массив будет заполняться данными из считываемого файла.

FileLenght(<имя файла>) - длина указываемого именем файла.

ФУНКЦИИ ПЕРИФЕРИИ

SetMouseKey(<код клавиши>) - функция возвращает имеющийся код мышки и устанавливает новый <код клавиши>.

SetKeyb(<код клавиши>) - возвращает код клавиши и устанавливает новый <код клавиши>.

GetKeyb() - возвращает код клавиши и устанавливает новый <код клавиши>.

// GetKeybDetail() –
// SetKeybDetail()

MouseKey() - возвращает код кнопки клавиши
MouseY() - возвращает координату Y курсора мышки
MouseX() - возвращает координату X курсора мышки

ФУНЦИИ ТАЙМЕРА

SetTimer(<идентификатор таймера>, <время в миллисекундах>)
<идентификатор таймера> - число - номер таймера.
<время в миллисекундах> - число - время в миллисекундах, на которое устанавливается таймер.
KillTimer(<идентификатор таймера>) – удаляет таймер.

TimeIsZero(<идентификатор таймера>) - если таймер обнулен, то возвращается 1, иначе 0 (истина или ложь).

GetTimer(<идентификатор таймера>) - возвращает установленное начальное значение таймера.

TimeSec() - возвращает значение системного таймера.

УСТАНОВКИ РЕЖИМОВ УПРАВЛЕНИЯ.

Ход вычислений в CAPER, ресурсы вычислительной установки, а также ресурсы языковой среды CAPER регулируются с помощью команды SET:

SET BLOCK/LABEL/WHEN ON/OFF

BLOCK ON - включить автоматическое выполнение входящего в исполняемый блок подблоков.
BLOCK OFF - выключить.

LABEL ON - включить механизм локализации переходов по меткам внутри одного блока - разрешает переход по глобальной метке только внутри одного блока.
OFF - отключает режим.

WHEN ON - включить событийный механизм (анализ WHEN-назначений).
OFF - выключить.

FindFirstFile - выбор оглавления директории (по Windows)
FindNextFile - выбор следующего файла директории.
FindClose – завершение выбора/перечисления файлов.

CTime() - возвращает значение таймера
DelCollection(<переменная>) - удаляет указываемую динамически созданную коллекцию

IsEqAddr(<переменная 1>, <переменная 2>) -сравнение абсолютных адресов
ParamNumber() - количество параметров
TypeOf(<имя переменной или выражение>)
Addressable(<имя блока/имя коллекции>) - превращает локальный для модуля адрес в глобальный

GetStockObject(идентификатор объекта) - аналог Windows: GetStockObject(LTGRAY_BRUSH)
GetBlockType(<имя блока>) - возвращает тип блока

Функции сравнения строк (аналоги C):
StrCmp
StrnCmp
StrniCmp
StriCmp

ChangeMSRegion(<идентификатор событийного региона>, <y0>, <x0>, <y1>, <x1>)
<идентификатор событийного региона> - возвращается SetOSEventRgn;
<y0>, <x0>, <y1>, <x1> - новые координаты региона.

ChangeKBRegion <идентификатор событийного региона> - возвращается SetKeybRgn;
<y0>, <x0>, <y1>, <x1> - новые координаты региона.

ChDir
ChDrive
GetCWD
GetDCWD
GetDrive -

MkDir - создать директорию
RmDir - удалить директорию
SearchEnv
SetModulesDir - устанавливает текущую директорию, из которой загружаются объектные модули

ФУНКЦИИ КОНВЕРТИРОВАНИЯ ТИПОВ

AddrAsInt
IntAsArr
IntAsRef
IntAsCom
IntAsStr

НЕ ДОКУМЕНТИРОВАНО !!!!!!!!!

GetByte
SetByte
SetElement
ReSizeArr
CharSize
AsStr
StrCat
CaptureMSBy
FreeMouse

GetParallInfo
GetStackInfo

ChangedEvent
StrLenxDxA
StrLenxA
GetDebuggedParm
NTypeOf
FilesNumber

GetAssocIco
DrawIco
DestroyIco
ExtractIco

GetFuncName
SplitPath
GetLogDriveStr
GetDriveType
GetDiskInfo
CaptureKBBy
FreeKeyb
GetCaptMSInfo
GetCaptKBInfo
GetCollInfo
GetBlockInfo
ForceType
SetMousePos
Clock
AtChr
RightChr
GetMsEvInfo
CreateOSBitmap
ShowOSBitmap
DelOSBitmap
UpCase
LoCase
ToSign
ToAlNum
ToAlpha
ToDigit
SkipSymb
String
DelString
SetCallDeb
SetEnvComp
GetSysMetrics
SetSVMEvent
DelSVMEvent
FreezeSVMEvent
DeFreezeSVMEvn
ClearSVMEvent
IsSVMEvent
AsNumber
BytesCpy
ConvBMArray
SetImgInfo
Master
CollVarNumb
CollVar

isCollVar
StoreMsEvents
RestoreMsEvents
StoreKBEvents
RestoreKBEvents
StoreOSBitmap
CreateComputBmp
atVal
tplCmp
FillBytes
FillBytesCol
FillWords
FillWordsCol
FillHWords
FillHWordsCol
FillFloat
FillFloatCol
FillDouble
FillDoubleCol
ChangeMSsys
ChangeKBsys
RestoreMSsys
RestoreKBsys
LineFromToWdt
FillPixInArray
FillInMask
StorePixel
LoadPixel
ChangeOSsys
RestoreOSsys
StretchOSBlt
FillPixBM
StorePixBM
CreateOSBMIndir
StretchBltDEV
typeSize
GetCursor
AddFont
RemoveFont
Exec
System
SetDestroy
SetQuit
getDestroy
getQuit
AtNoZero

ChngPixBM
SumPixInArray
MaxRGBInArray
MinRGBInArray
SortInArray
SumOnMatrix
SetCurrentImage
SetGraphParams
CompOnMatrix
EnumFonts
GetGlyphArr
GetGlyphSize
GetTextMetrics
GetOLTextMetr
MemStatus
GetErrorInfo
MasterBlock
GetTicks
BuildObject

/* Документировать ? */

ФУНКЦИИ ВЫБОРА АЛЬТЕРНАТИВНЫХ ЗНАЧЕНИЙ

Iff(<арифметическое выражение>, < значение по "истина" >,
< значение по "ложь" >) - возвращает
< значение по "истина" >, если значение <арифметического выражения>
положительно, и < значение по "ложь" > в противном случае.

CaseF(<арифметическое выражение>, < значение по "истина" >,
< значение по "ложь" >)

==========

Литература:

1. Вартанов С.Р. Язык программирования CAPER. Препринт 97-5. Национальная Академия Наук Украины, Институт Кибернетики им. Глушкова. Киев (1997).
2. Вартанов С.Р. О языке программирования КАРСВАК. - Деп. в АрмНИИНТИ, рег. номер: N4-Ар88, 1988, 34 с.
3. Вартанов С.Р. Язык и методы программирования в задачах обработки изображений. Автореферат диссертации на соискание ученой степени кандидата физ.-мат. наук.
4. Вартанов С.Р. Основания к концепции мультиязыков. В сб.: Информационные

технологии и управление. Том 4-3, 2005, 8-23.
5. Vartanov S.R. Parallel Programming in Caper. In Mathematical Questions of Cybernetics and Computing Technique. Vol.22, Yerevan, 2001, 100-112.
6. Vartanov S.R. On Parallel Programming Language Caper. Lect. Notes in Computer Sci., HCPN-2001, 501-503.
7. Vartanov S.R. Parallel Programming Methods in Caper Language and its Application in Image Processing. In: SCI2002/ISAS2002, Orlando, USA, 2002, vol. XI
8. Вартанов С.Р., Нуриджанян Ш.Р., Акопджанян В.А., Манукян А.Г.: Параллельные алгоритмы Решета Эратосфена и их программирование. В сб.: Информационные технологии и управление. Том 1, 2004, 13-22.
9. Flynn