Троичная виртуальная машина и троичная ДССП

к.ф.м.н. А.А. Бурцев, к.ф.м.н. С.А. Сидоров
НИИСИ РАН, НИЛ троичной информатики МГУ
burtsev@niisi.msk.ru, sidorov@niisi.msk.ru

Аннотация

В НИЛ троичной информатики ВМК МГУ (в период с 2010 по 2013 г.) созданы троичная виртуальная машина ТВМ и кросс-система ДССП-ТВМ разработки программ для неё на языке ДССП-Т – троичном варианте языка ДССП.

В докладе представляются архитектура троичного процессора ТВМ и его система команд, основные черты языка ДССП-Т и возможности среды разработки ДССП-ТВМ, а также поясняются некоторые проблемные аспекты реализации имитатора ТВМ, кросс-компилятора языка ДССП-Т и диалогового интерпретатора ДССП/ТВМ для специфичной троичной машины.

Введение

В настоящее время, когда традиционная (кремниевая) микроэлектроника, построенная на двоичной системе счисления, подходит к пределу своих технологических возможностей, возрождается интерес к исследованию принципиально иных путей построения вычислительных средств, основанных на других системах счисления. Как одна из наиболее перспективных, многих исследователей давно привлекает троичная симметричная система счисления (с цифрами -1,0,+1), которую когда-то Д.Кнут назвал “самой изящной” [1], и которая была полвека назад успешно воплощена в троичных цифровых машинах (ЦМ) “Сетунь” и “Сетунь-70” [2], разработанных в НИЛ ЭВМ МГУ под руководством Брусенцова Н.П.

Использование симметричного троичного кода и трёхзначной логики даёт ряд неоспоримых преимуществ (перед двоичными системами), которые были подтверждены практической эксплуатацией этих машин. Среди множества достоинств рассматриваемой троичной симметричной системы счисления можно выделить следующие:

- тем же количеством разрядов можно кодировать больший диапазон чисел;
- единообразное представление отрицательных и положительных чисел;
- непосредственное воплощение логики с тремя значениями:

1) +1 – “да”, 2) -1 – “нет”, 3) 0 – “не знаю” (“может быть”?) ;

- упрощённая реализация ряда арифметических операций (сдвига, сравнения чисел, смены знака); трёхзначность функции “знак числа”;
- оптимальное округление чисел простым отсечением младших разрядов и взаимокомпенсируемость погрешностей округления в процессе вычислений.

С осознанием этих преимуществ приходит понимание, что с троичными вычислениями и троичными компьютерами связано будущее вычислительной техники. И горизонты этого будущего открываются сегодня всё большему кругу научных работников и специалистов и не только сферы информационных технологий. Так, например, в Интернете появились статьи на тему: “Будущее квантовых компьютеров — в троичных вычислениях”[3]. Встречаются также публикации о преимуществах троичных машин, в которых в качестве одного из доводов в пользу троичности приводится упоминание о троичности нейрона человеческого мозга [4].

В настоящее время в НИЛ троичной информатики МГУ (ранее НИЛ ЭВМ), где продолжаются исследования возможных способов реализации троичного процессора на современной элементной базе [5-6], была предпринята попытка создания имитационной модели троичного процессора, а также системы разработки троичных программ для него. В результате проведённой (в 2010-2013 гг.) работы был создан программный комплекс ТВМ (Троичная Виртуальная Машина) [7], имитирующий функционирование современного варианта троичного процессора двухстековой архитектуры с поддержкой структурированного программирования на уровне машинных команд, аналогичной той, что была обеспечена в троичной ЦМ “Сетунь-70”.

В той же НИЛ ЭВМ МГУ ранее (в 80-х годах XX века) была разработана система программирования ДССП [8], которая на протяжении ряда лет применялась для программирования двоичных мини- и микрокомпьютеров самой разнообразной архитектуры и в различных операционных средах. ДССП убедительно показала, как важны для успешного построения компьютерных программ интерактивная среда разработки, двухстековая архитектура, структурированный набор команд управления и поддержка нисходящей разработки по принципу “сверху-вниз” (top-down). Поэтому в качестве основного средства разработки троичных программ для ТВМ было решено создать троичный вариант этой системы программирования, т.е. троичную ДССП с языком ДССП-Т – троичным вариантом языка ДССП.

Работа по созданию системы разработки программ для ТВМ на языке ДССП-Т была выполнена в два этапа. Сначала (на 1-ом этапе) была построена кросс-система ДССП-ТВМ [9], позволяющая создавать программы для ТВМ на языке ДССП-Т с помощью кросс-компилятора. Затем (на 2-ом этапе), используя уже саму ДССП-ТВМ как среду разработки, был создан диалоговый интерпретатор ДССП/ТВМ [10], способный функционировать на троичной машине (ТВМ) в качестве её резидентной системы программирования.

1. Троичная виртуальная машина

Троичная виртуальная машина (ТВМ, Ternary Virtual Machine - TVM) представляет собой программный эмулятор архитектуры троичного компьютера, основанного на троичной симметричной системе счисления и наследующего некоторые принципы построения троичных ЭВМ «Сетунь» и «Сетунь-70», а также нацеленный на прямое исполнение сшитого кода, являющегося внутренним представлением ДССП-программ.

Программный эмулятор работает на обычных двоичных компьютерах, полностью воссоздавая троичную среду на уровне машинных команд, регистрового файла, памяти и адресного пространства. Программист ТВМ имеет дело только с троичными объектами и не должен беспокоиться о представлении информации. В его распоряжении обычные для этого уровня регистры, память, стеки, прерывания и пр.

Для программирования ТВМ разработан простой транслятор с языка ассемблера троичного процессора, поддерживающий набор машинных команд и несколько необходимых директив для размещения данных и кода. Язык ассемблера используется как для непосредственного программирования ТВМ, в частности, ядра разрабатываемой троичной ДССП, так и в качестве промежуточного языка для кросс-компилятора троичной ДССП : кросс-компилятор генерирует текст на языке ассемблера, который затем транслируется в машинный код. Это позволяет радикально упростить кросс-компилятор ДССП и заметно облегчить внесение изменений в архитектуру ТВМ и реализацию ДССП, развязав их на уровне генерации текста.

Для удобства использования и отладки программ в ТВМ встроен интерактивный монитор, обеспечивающий взаимодействие оператора с виртуальной машиной в консольном режиме. Монитор предоставляет простые возможности загрузки программ в память ТВМ, передачи управления программе, просмотра и модификации памяти и регистров виртуальной машины, задания точки останова, протоколирования работы и пр.

Таким образом, ТВМ включает собственно виртуальную машину (эмулятор), интерактивный монитор и транслятор с языка ассемблера, которые в совокупности образуют среду разработки и отладки программ, достаточно удобную для создания ПО троичной машины (и в частности, троичной версии ДССП).

Еще одна цель создания ТВМ состоит в исследовании способов возможной реализации троичной архитектуры в кремнии с предварительным моделированием в двоичных ПЛИС. В этом контексте тщательная проработка представления данных и алгоритмов выполнения арифметических и логических операций приобретает особое значение.

1.1. Основные особенности архитектуры троичной машины

ТВМ построена с учётом определённых принципов, положительно зарекомендовавших себя в троичных машинах «Сетунь» и «Сетунь-70», а также в системе программирования ДССП.

1. Двухстековая архитектура. Для обработки данных используется арифметический стек, а управляющий стек – для сохранения адресов возвратов из подпрограмм.
2. Поддержка структурированного программирования на уровне машинных команд. Для управления ходом исполнения программы предлагаются машинные команды, эквивалентные основным управляющим операторам (конструкциям) структурированного программирования : 1) обычный (безусловный) вызов подпрограммы; 2) условный вызов одной из подпрограмм; 3) циклический вызов подпрограммы, повторяющийся пока соблюдается заданное условие.
3. Эффективная поддержка исполнения сшитого (процедурного) кода, служащего внутренним представлением откомпилированных ДССП-программ. ТВМ должна уметь сама напрямую исполнять такой сшитый код, т.е. быть его интерпретатором; это значит, что большинство примитивов ДССП должно выполняться напрямую командами ТВМ, а остальные – представляться процедурами в машинном коде, пригодными для непосредственного исполнения на ТВМ.

Каждый из этих принципов утвердился вследствие постоянного стремления разработчиков аппаратуры максимально удовлетворить нужды ПО, чтобы сделать процесс

разработки ПО проще и понятнее. Так, отвечая этим насущным потребностям, а также учитывая опыт создания и использования интерпретатора ПОЛИЗ [11] на ЦМ «Сетунь», в архитектуре ЦМ «Сетунь-70» уже сразу была предусмотрена аппаратная поддержка стека данных (действия с которым в интерпретаторе ПОЛИЗ приходилось осуществлять программным способом). А впоследствии система команд ЦМ «Сетунь-70» была дополнена командами структурированного управления, чтобы оказать поддержку структурированному программированию на уровне машинного кода.

Вот и при разработке ТВМ эта “добрая традиция” была продолжена так, чтобы система команд ТВМ была приспособлена для прямого исполнения сшитого кода ДССП-программ.

1.2. Троичная симметричная система счисления

Троичная система счисления, как и двоичная, основана на позиционном принципе кодирования чисел, при котором вес n-ого разряда равен не 2^n, а 3^n. При этом сами разряды (в отличие от битов) не двухзначны (как биты), а трехзначны (их принято называть тритами – trit). Помимо 0 и +1 они допускают третье значение, которым в симметричной системе служит –1, благодаря чему единообразно представимы как положительные, так и отрицательные числа. Таким образом, в троичной симметричной системе счисления знаком числа оказывается цифра старшего из его значащих (ненулевых) разрядов. Проблемы чисел со знаком, не имеющей в двоичном коде совершенного решения, в троичном симметричном коде просто нет, чем и обусловлены его принципиальные преимущества. Это, в частности, такие, как естественное и неизбыточное представление чисел со знаком, упрощённое округление числа простым отбрасыванием его младших разрядов, явное представление третьего (привходящего) логического значения, необходимого для адекватного представления логических отношений.

1.3 Представление троичных данных в ТВМ

Будем придерживаться следующих понятий. Трит – минимальная единица троичной информации, принимает значения –1,0,+1. Трайт – минимальная адресуемая единица информации для обмена с памятью. Трайт состоит из 9 тритов и соответственно принимает 19683 различных значений в диапазоне [–9841, 9841]. Триты в трайте нумеруются справа налево, начиная с нуля и до 8.

В ЦВМ «Сетунь» использовались понятия короткого (9 тритов) и длинного (18 тритов) машинного слова, В ЦВМ «Сетунь-70» размер трайта был равен 6 тритам, а слова – 18 тритов. Такой выбор был обусловлен разнообразными причинами, но пожалуй одной из главных была экономия весьма дорогостоящей в то время памяти. Сегодня всё же представляется более логичным выбор размеров элементов данных кратными степени тройки.

Троичное слово в ТВМ – группа из 3 смежных трайтов (т.е. 27 тритов). Таким троичным словом можно закодировать 7625597484987 различных значений, расположенных в диапазоне: [–3812798742493, 3812798742493]. Выбор 27 разрядов для троичного слова ТВМ обусловлен как раз тем, что 27 – это минимальная степень тройки ($27=3^3$), обеспечивающая возможность перекрыть таким количеством троичных разрядов диапазон значений числа, представимого 32-разрядным двоичным словом: (3^{27} = 7625597484987 > 2^{32}=4294967296), которое фактически принято в качестве стандарта во многих современных двоичных цифровых машинах.

В ТВМ принята адресация little-endian, т.е. адресом слова является адрес младшего трайта. При этом слово, в т.ч. команда, может начинаться с любого адреса трайта.

Стек данных – структура данных с дисциплиной доступа «последним вошел – первым вышел» (LIFO). Элементом данных в стеке является троичное слово. Первым элементом данных в стеке считается его вершина, т.е. то, что было помещено в стек последним и будет извлечено первым. Далее следует второй элемент (подвершина), третий, и т.д.

Адресное пространство – диапазон адресов трайтов памяти. Адресное пространство ограничено диапазоном целых чисел, представимых одним словом: [–MAX, MAX], где MAX = 3812798742493. Адресное пространство единое для команд и данных. Регистры внешних устройств отображаются также в адресное пространство.

В отличие от привычных двоичных машин, нулевой адрес в адресном пространстве троичного процессора расположен посередине. Это обстоятельство явно используется для

размещения программ и данных.

Очевидно, в двоичном компьютере, используемом для имитации работы троичного процессора, для представления минимальной единицы троичной информации – трита – требуется как минимум два бита. Для троичного процессора выбрано представление парой соответствующих битов в двух двоичных словах {P,M}, причем в первом слове P единицы стоят на тех местах, где в представлении троичного числа стоят “+1”, а в слове M – там, где в троичном числе стоят “–1”. Например, если обозначить “–1” как “–”, а “+1” как “+“, то троичное число “00+0–0+–+” в используемом для имитации двоичном компьютере будет выглядеть как {P,M}={001000101,000010010}. Такое представление в большинстве случаев допускает параллельную обработку целых машинных слов и менее громоздко в программировании.

1.4 Регистры троичного процессора

Все регистры троичного процессора имеют размер в одно (27-разрядное) троичное слово. Часть тритов регистра может не использоваться, в этом случае оттуда считываются нули, а при записи эти триты игнорируются.

Программно доступные регистры:

MEMCAP	- (read only) Размер памяти в трайтах
CSP	- Control Stack Pointer - указатель стека возвратов
CSPL	- CSP Low – нижняя граница стека адресов возврата
CSPH	- CSP High – верхняя граница стека адресов возврата
DSP	- Data Stack Pointer - указатель стека данных
DSPL	- DSP Low – нижняя граница стека данных
DSPH	- DSP High – верхняя граница стека данных
EXC	- Exception – вектор прерывания (исключения)
PC	- Program Counter – программный счетчик
STATUS	- маски и флаги прерываний и пр.
R0-R3	- регистры общего назначения
IVBASE	- базовый адрес векторов прерываний

Имеется также несколько программно недоступных регистров, играющих важную роль в организации исполнения команд.

IC	- Instruction Code – код инструкции, которая будет выполняться
ICL	- Instruction Code Lock – флаг блокировки IC

1.5 Структура стеков

Оба стека располагаются в основной памяти троичного компьютера. В регистрах процессора находятся указатели на вершину, а также на верхнюю и нижнюю границы каждого стека: арифметического (`DSP`,`DSPH`,`DSPL`) и управляющего (`CSP`,`CSPH`,`CSPL`) соответственно.

Стек данных растет в сторону увеличения адресов, стек возвратов – в сторону уменьшения. При попытке извлечь данные из пустого стека и при попытке положить данные в заполненный стек возникают исключительные ситуации, вызывающие прерывание работы процессора.

Элементы данных размером в один или два трайта укладываются в стек данных в младшие трайты вершины, извлекаются также из младших трайтов. Для стека возвратов возможен обмен только целиком троичными словами.

1.6 Форматы команд

Все инструкции троичного процессора занимают одно троичное слово. Это позволяет сделать существенно более простую реализацию и радикально сократить количество команд за счет разнообразия операндов.

Всего предусмотрено 3 формата команд троичного процессора, различающихся старшим тритом. При этом если старший трит равен 0, то это команда **CALL** безусловного вызова подпрограммы, а оставшиеся в коде команды 26 тритов представляют адрес подпрограммы.

Хотя при такой кодировке часть адресного пространства недоступна для непосредственного вызова расположенной там подпрограммы, оставшегося диапазона 3^{26} (а это более 2500 Гигатрайт) представляется нам вполне достаточным для хранения программ.

Все остальные команды кодируются в формате, где в старшем трите стоит значение 1. Под код операции отведено 5 тритов (243 команды, сейчас используется менее половины), 3 трита – под номер регистра, и остальные 18 тритов (т.е. два трайта) – под непосредственное значение. Оно используется как слагаемое при вычислении адреса, как непосредственный операнд в командах обработки данных и т.п. Диапазон [–193710244, 193710244] достаточен для большинства случаев применения таких значений.

Команды же, в старшем трите которых стоит значение –1, зарезервированы для будущих применений.

1.7 Основной цикл исполнения команд

Выполнение команд троичного процессора определяется работой простого автомата, описываемого следующим алгоритмом:

```
NEXT_INSTRUCTION:
// Проверка на прерывание (исключительную ситуацию):
if ((ICL==0) && ((EXC<-3) || ((EXC==-3) && (STATUS[IE]==1))) {
   pushcs(STATUS); // сохранить STATUS
   STATUS[IE]=0;   // запретить прерывания
   pushcs(PC);     // сохранить PC
   PC = m(IVBASE + EXC); // извлечь адрес обработчика
   EXC = 0;
}else{  // Прерывания не требуется
   if (ICL == 1) ICL=0; // IC locked, исполняем инструкцию из IC
   else // Как обычно берем очередную инструкцию
       { IC = m(PC); PC = PC+3; }
}
// здесь IC и PC имеют правильное значение
execute; // исполняется команда, код которой принят в регистр IC
```

1.8 Система команд троичного процессора

Далее приведен обзор набора команд (инструкций) троичного процессора ТВМ.

Управляющие команды представлены вызовами подпрограмм, условными и безусловными, а также командой организации цикла и рядом дополнительных команд.

Безусловный вызов подпрограммы выполняет команда **CALL**, в качестве операнда потребляющая прямой адрес подпрограммы. В языке ассемблера это обычная метка.

Возврат из подпрограммы обеспечивает команда **RET** без аргументов. Для возврата из обработчика прерывания (исключения) должна применяться команда **RFE**, т.к. при входе в обработчик в стек засылается также содержимое регистра `STATUS`.

Имеется также возможность выполнить любую команду троичного процессора, задав ее код в стеке данных; это делает инструкция **EXEC.**

Команды ветвления по условию фактически выполняют условное выполнение одной из инструкций, стоящих следом. Как и в языке ДССП, предусмотрены команды ветвления по знаку операнда в вершине стека данных на одну, две и три ветви – **IF***, **BR***, **BRS** соответственно, где ***** означает букву P (plus), N (minus) или 0. Например, запись

```
{X} BRM P1 P2  P3
```

означает, что перед выполнением инструкции **BRM** (ветвиться по минусу) в стеке находится значение X (в фигурных скобках в качестве комментария указывается содержимое стека); инструкция **BRM** извлекает это значение и анализирует его. Если X<0, то выполняется инструкция P1, иначе P2. В любом случае после выполнения P1 или P2 далее будет выполняться инструкция P3.

На месте P1 и P2 могут стоять любые инструкции троичного процессора, за исключением

инструкций ветвления и цикла. Так, если фрагмент программы будет выглядеть так:

```
{X} BRM CALL_S1 CALL_S2  P3
```

то будет выполняться условный вызов подпрограммы S1 или S2.

Команда организации цикла устроена аналогично:

```
COND {CV} DW  BODY
```

Здесь значение CV (нулевое или ненулевое) обозначает результат проверки некоторого условия. Оно вычисляется и посылается в стек каждый раз при исполнении команды `COND` (как правило, это вызов подпрограммы, вычисляющей условие цикла). Команда **`DW`** извлекает это значение из стека и проверяет его. Если оно не равно нулю, то выполняется команда `BODY` (как правило, это вызов подпрограммы, выполняющей тело цикла). После выполнения `BODY` управление попадает на стоящую перед командой `DW` команду `COND`, которая должна оставить после себя в стеке новый результат проверки условия (значение CV). Если же значение CV было нулевым, то управление попадает на команду, стоящую после `BODY`, и тем самым прекращается многократное повторение серии действий (`COND`, `BODY`, `COND`, `BODY`, …).

На основе одной такой команды цикла можно обеспечить построение циклов всех видов: бесконечных, со счетчиком, с проверкой условия до или после тела цикла.

Целочисленные арифметические инструкции обычные: сложение, вычитание, умножение и деление, а также сравнение, максимум и минимум. Особенность их в том, что операнды эти инструкции изымают из стека данных, и засылают в него обратно результат.

Следует отметить, что в симметричной системе счисления используются иные правила вычисления знаков и значений частного и остатка от деления. Они вычисляются по следующему алгоритму (записанному здесь на языке Си):

```
// Деление n/m : n = q*m + r, q – частное, r - остаток
q = n/m; r = n%m; m = abs(m);
if ( 2*abs(r) > m ) {
   if (q < 0) q=q-1; else q=q+1;
   if (r < 0) r=r+m; else r=r-m;
}
```

Как и в любом процессоре, значительное место в наборе команд занимают инструкции пересылки данных. Троичный процессор обеспечивает пересылку данных в/из памяти, стеков, регистров, а также засылку непосредственно заданных в коде команды значений. При обмене с памятью поддерживается чтение и запись одного, двух и трех трайтов, а при пересылках стек-стек и стек-регистр разрешается чтение или запись только троичного слова целиком. Засылка значения в стек и извлечение из стека сопровождается изменением указателя вершины стека, при этом контролируется нахождение указателя в заданных пределах. При выходе за границы возникает исключительная ситуация.

Стековая организация процессора требует набора команд для манипуляции содержимым стека данных. Эти команды были хорошо проработаны в ДССП, здесь они воспроизведены в чуть более общем виде: имеется команда удаления n позиций из стека данных, копирования значения с указанной глубины, обмена вершины и любого элемента стека. Есть варианты этих команд с заданием параметра в вершине стека, т.е. вычисляемого.

К логическим инструкциям традиционно относят сдвиги и поразрядные операции. В троичном процессоре предусмотрены сдвиги (причем специальные арифметические сдвиги в симметричной системе счисления не требуются), поразрядные сложение, умножение, минимум и максимум.

При разработке программ для троичного процессора необходимо было обеспечить хотя бы минимально возможные средства ввода-вывода (по крайней мере, для взаимодействия с консолью терминала, а также доступ к файлам среды). В дальнейшем предполагается разработка интерфейса ТВМ к обычным внешним устройствам (очевидно, двоичным), а пока (т.е. на данный момент) в троичном процессоре реализованы (временно на период разработки) вспомогательные команды, поддерживающие элементарные возможности ввода и вывода: распечатка вершины стека, печать символа и строки, операции для чтения и записи файлов.

1.9 Интерактивный монитор

Интерактивный монитор в составе ТВМ выполняет роль «пульта управления» виртуальной машиной. Аналогичные возможности как правило предоставляются встроенным программным обеспечением компьютеров, в данном же случае функциональность интерактивного монитора реализована в самой эмулирующей программе.

Монитор выполняет команды пользователя, вводимые с клавиатуры, и отображает результаты на экране. Команды монитора обеспечивают основные действия по загрузке, выполнению и отладке программ.

Загрузка программы в ТВМ выполняется из файла, подготовленного компилятором с языка ассемблера, причем загрузить программу можно как при старте ТВМ, так и уже в интерактивном режиме.

Для отладки программ предусмотрены команды просмотра и изменения значения регистров троичного процессора и памяти в различных форматах, а также установка и снятие точек останова, старт и продолжение выполнения программы, пошаговое исполнение.

Протокол отладки можно записать в файл для последующего анализа. Предусмотрена также возможность выполнения последовательности диалоговых команд монитора, приготовленной в файле, что существенно ускоряет процесс отладки.

1.10 Ассемблер троичного процессора

Язык ассемблера предоставляет возможность программирования в символьных обозначениях для троичного процессора ТВМ. Программа на языке ассемблера представляет собой последовательность инструкций в свободном формате. Разделителем инструкций служит пробел или точка с запятой. В случае инструкций переменной длины завершающая точка с запятой обязательна.

Инструкции и операнды отделяются по крайней мере одним пробелом (символом табуляции, концом строки) или знаком “_” (подчеркивание). Таким образом, команды, состоящие из нескольких слов, можно визуально объединять знаком подчеркивания, а в других случаях использовать пробел.

Числа могут быть представлены в троичной, девятеричной, десятичной и шестнадцатеричной системах счисления, в том числе со знаком минус.

В языке ассемблера предусмотрены метки – символические обозначения адресов. Метка – имя, за которым без пробела стоит двоеточие. Метки могут располагаться в произвольном месте программы между инструкциями.

Набор директив транслятору традиционен: размещение кода программы, размещение данных, определение текстовых строк и пр.

В качестве результата трансляции выдается файл с троичным кодом программы и текстовый файл с дизассемблированной программой.

2. Общая характеристика ДССП

Диалоговая система структурированного программирования (ДССП) была создана в начале 80-х годов XX века в проблемной лаборатории ЭВМ на факультете ВМК МГУ под руководством Брусенцова Н.П. ДССП была призвана существенно облегчить разработку ПО для широкого класса малых цифровых машин – мини- и микрокомпьютеров.

При создании ДССП преследовалось сразу несколько целей. Во-первых, ДССП создавалась в качестве программного имитатора (эмулятора) новой двоичной цифровой минимашины, в которой предполагалось воплотить ценные идеи двухстековой архитектуры и структурированных команд управления, унаследованные от модернизированной троичной ЦМ “Сетунь-70”. В роли такого имитатора или образно говоря виртуального процессора по сути и функционирует внутренний интерпретатор ДССП, являющийся сердцевиной (ядром) системы.

Во-вторых, ДССП создавалась как интегрированная среда разработки программ, легко доступная для освоения широкому классу пользователей микрокомпьютерной техники и персональных компьютеров. Поэтому при создании ДССП была заимствована идея словарной организации системы ФОРТ (FORTH) [12] и обеспечен диалоговый характер взаимодействия с пользователем на всех стадиях разработки программы: начиная от её редактирования и вплоть до её окончательной отладки. Такой диалог обеспечивается внешним интерпретатором ДССП, функционирующим в режиме исполнения потока слов-команд.

Программа, составленная на входном языке ДССП, сначала преобразовывается во внутреннее представление, для характеристики которого применяется специальный термин – сшитый (threaded) код. Совокупность программных компонент, выполняющих такое преобразование, принято называть компилирующими средствами или просто компилятором системы. Внутренний и внешний интерпретатор, компилятор и словарь ядра, а также встроенные редактор текстов и отладчик вместе составляют базовый комплект ПО системы. На его основе путём наращивания словаря каждый пользователь может создать свой вариант расширения ДССП, отвечающим нуждам его прикладной области.

Программирование в ДССП заключается в наполнении её словаря новыми словами. Всякая операция (процедура), поддерживаемая системой, обозначается отдельным словом (слово – это любая последовательность печатных символов, отличных от пробела). Совокупность слов, понимаемых системой, и составляет ее язык (словарь). Система расширяется путем определения новых слов, которые сразу же после добавления в словарь могут быть использованы в системе.

Любое действие, совершаемое системой, вызывается словом. Предусмотренные в системе слова исполняют самые разнообразные функции. Они выполняют вычисления или обработку данных в стеке, доступ к памяти или к переменным, служат для организации ветвлений и циклов, обеспечивают построение тел для новых слов, вносимых в словарь, управляют состоянием словаря и даже самим процессом выполнения задания в системе.

ДССП характеризуется двухстековой архитектурой. Стек операндов, над которым действуют все операции по обработке данных, приводит к компактной польской инверсной записи арифметических выражений, а управляющий стек служит для сохранения адресов возвратов из процедур.

Следующие две характерные черты ДССП выгодно отличают её от системы ФОРТ: 1) специфическая процедурная версия структурированных команд управления; 2) и поддержка пошаговой (step-by-step) нисходящей (top-down) разработки программы.

В ДССП каждая из команд ветвлений вместо перехода по телу сшитого кода (как это обычно делается в системе ФОРТ) по сути осуществляет вызов процедуры слова, выбранного по условию, а каждая команда цикла организует многократный вызов процедуры слова, пока такой вызов допускается условием цикла. Таким образом, в ДССП поддержка структурированного программирования осуществляется непосредственно на уровне сшитого кода аналогично тому, как такая поддержка была обеспечена в ЦМ “Сетунь-70” на уровне машинных команд.

Определение нового слова в ДССП и в языке ФОРТ синтаксически выглядят одинаково:

```
: P P1 P2 ... Pn ; [определение процедуры с именем P]
```

Однако ФОРТ допускает подобное определение, только если все слова `P1 P2 ... Pn` уже были определены и стали известны системе. А ДССП допускает, что в этой последовательности слов `P1 P2 ... Pn` могут встречаться в том числе и такие слова, которые еще не были определены и предполагается, что будут определены впоследствии. Именно благодаря такой особенности ДССП и позволяет разрабатывать программу методом нисходящего анализа: сначала определить главное слово, затем слова следующего уровня, из которых оно определяется, и так постепенно спуститься до детального определения самых простых слов.

Первая версия ДССП (ДССП-НЦ [13]) была создана для микрокомпьютеров “Электроника НЦ-03Д”. Уже с этой версии в ДССП нашли отражение характерные черты языка ассемблера ЦМ “Сетунь-70”: его структурированные команды управления для организации ветвления и цикла по условию: **`BRS DW0N`** .

На следующем этапе своего развития ДССП создавалась уже для микрокомпьютеров так называемой унифицированной архитектуры (PDP-11). Причём наряду с основной версией ДССП-80 [14] разрабатывались также и экспериментальные версии, представляющие собой варианты развития ДССП, ориентированные на построение программ управления периферией (ДССП-ПМ [15]) и операционных систем реального времени (ДССП-РВ [16]).

В ДССП-80 были предложены новые команды **`RP DO`** для организации циклов с выходами по условию **`EX- EX0 EX+`** , а также типизированный набор префиксных операций:

`! ’ !+ !- !1 !0 ... !!! CAP?`

для единообразной работы с переменными и массивами различных типов. Была усовершенствована организация словаря (он стал перемещаемым, в нём появились подсловари и секции), позволившая упростить разработку сложных многомодульных программ. Для автоматизации сборки как самой ДССП, так и законченной прикладной программы были предложены специальные инструментальные средства: компоновщик и целевой компилятор

Хотя экспериментальные версии ДССП-ПМ и ДССП-РВ не получили самостоятельного развития, предложенные в них новые программные механизмы были частично реализованы в последующих версиях ДССП [17,18]. Это механизм исключительных ситуаций, сопрограммный механизм, механизм прерываний [19] на уровне сшитого кода, а также разнообразные средства взаимодействия параллельных процессов.

С начала 90-х годов ДССП стала активно использоваться в НИИСИ РАН в качестве инструментальной среды программирования для разработки внутреннего программного оснащения [29] ряда компьютерных модулей, создаваемых на основе разных микропроцессоров (MC68020, R3000, Intel80386).

В дальнейшем над развитием ДССП и распространением её на другие платформы трудились как сотрудники НИЛ ЭВМ МГУ вместе со студентами и аспирантами, так и сотрудники НИИСИ РАН [20]. В результате на протяжении ряда лет ДССП была реализована для компьютеров самой разнообразной архитектуры и в различных операционных средах. К середине 90-х годов ДССП функционировала почти на всех компьютерных платформах (PDP-11, Intel8080, Intel80x86, Motorola 68020, VAX, R3000) и в операционных средах (RT-11, RSX-11, UNIX, CP/M, MSDOS, Windows), получивших распространение в нашей стране.

Отдельные версии ДССП для разных платформ были плохо совместимы друг с другом. Их общий недостаток заключался в том, что ядро каждой из них создавалось на языке ассемблера базовой машины, что затрудняло перенос созданного ПО на другие компьютерные платформы. В 1998 г. в НИИСИ РАН была создана мобильная версия ДССП (ДССП/С), ядро которой было написано на языке Си [17]. При разработке ДССП/С была проведена такая модернизация работы с данными, благодаря которой в ДССП впоследствии были обеспечены возможности построения новых типов [21], аналогичные средствам объектно-ориентированного программирования.

На протяжении почти 20-летнего периода своего развития [8] ДССП постоянно совершенствовалась, приобретая новые возможности, присущие современным языкам и системам программирования. В результате ДССП превратилась в среду программирования [18], позволяющую применять широкий спектр современных методов структурированной разработки

сложных программных систем, сосредоточив в себе средства для структурированного, модульного, объектно-ориентированного и параллельного программирования, а также средства для структурированной обработки исключительных ситуаций.

3. Принципы построения, структура и состав ДССП-ТВМ

Система программирования ДССП-ТВМ предоставляет возможность разработки программ для троичного процессора на языке ДССП-Т и их исполнения на имитационной модели троичного процессора – ТВМ.

При создании ДССП-ТВМ учитывались перечисленные ниже требования как своеобразные принципы построения системы.

1. Кросс-компилятор языка ДССП-Т требовалось реализовать на языке Си, чтобы обеспечить возможность его функционирования в любой современной операционной среде (Windows, Linux).

2. Кросс-компилятор должен формировать внутреннее представление обрабатываемой ДССП-программы на языке ассемблера, чтобы избежать его зависимости от машинной кодировки команд ТВМ.

3. ДССП относится к классу интерпретируемых систем. В них исходная программа сначала переводится во внутреннее представление, которое затем используется внутренним интерпретатором, обеспечивающим непосредственное исполнение программы. При создании ДССП-ТВМ большинство функций внутреннего интерпретатора предполагалось возложить на ТВМ.

4. Для внутреннего представления ДССП-программ всегда использовался так называемый сшитый (threaded) код. В ДССП сшитый код формируется по принципу “1-1”, т.е. одно слово исходного текста переводится в одно слово кода. Требовалось соблюдать такой принцип и в ДССП-ТВМ при формировании (кросс-компилятором) внутреннего представления ДССП-программ.

5. ДССП для организации ветвлений и циклов предлагает специфические команды структурированного управления, для которых тела ветвлений и циклов должны оформляться в форме отдельных процедур. Исполнение большинства таких команд предполагалось обеспечить соответствующими командами ТВМ условного и многократного вызова процедуры, полностью исключая потребность в командах условного и безусловного переходов.

6. Кросс-компилятор должен допускать возможность формирования (на языке ассемблера ТВМ) не только программного кода ДССП-программы, но и соответствующего ей словаря, который мог бы использоваться далее при прогоне полученной ДССП-программы на ТВМ, а также при её отладке.

7. Создаваемая компилятором ассемблерная программа должна позволять подключать к ней другие модули на языке ассемблера ТВМ.

8. Допускалось использование стандартного препроцессора (языка Си) перед обработкой полученной кросс-компилятором программы ассемблером ТВМ.

В ДССП-ТВМ в качестве составляющих её компонент используются следующие программные средства (см. рис.1):

- кросс-компилятор (dsspcomp.exe);
- ДССП-библиотека (kern.dsp,...) – набор файлов на языке ДССП-Т (dsp-файлы), подключаемых в ДССП-программу командой LOAD в ходе её компиляции, которые содержат определения стандартного ассортимента наиболее употребительных слов;
- стандартный препроцессор (cpp.exe) языка Си ;
- ассемблерное ядро ДССП (kern.psm,...) – совокупность ассемблерных файлов (psm-файлов), подключаемых в ассемблерную программу на стадии препроцессорной обработки, которые содержат тела процедур, необходимые для исполнения ряда примитивов ядра ДССП;
- ассемблер (asm.exe) троичной машины;
- имитатор троичной машины (tvm.exe).

Программа (файл prog.dsp), составленная на языке ДССП-Т, проходит несколько стадий обработки в ДССП-ТВМ (см. рис.1). Сначала она обрабатывается кросс-компилятором, который создаёт ассемблерную программу (файл prog.psm), содержащую также препроцессорные директивы. Компилятор формирует листинг сообщений о ходе компиляции (файл prog.lst), а также может создать ещё и ассемблерное тело словаря (файл prog.asl), сформированного программой.

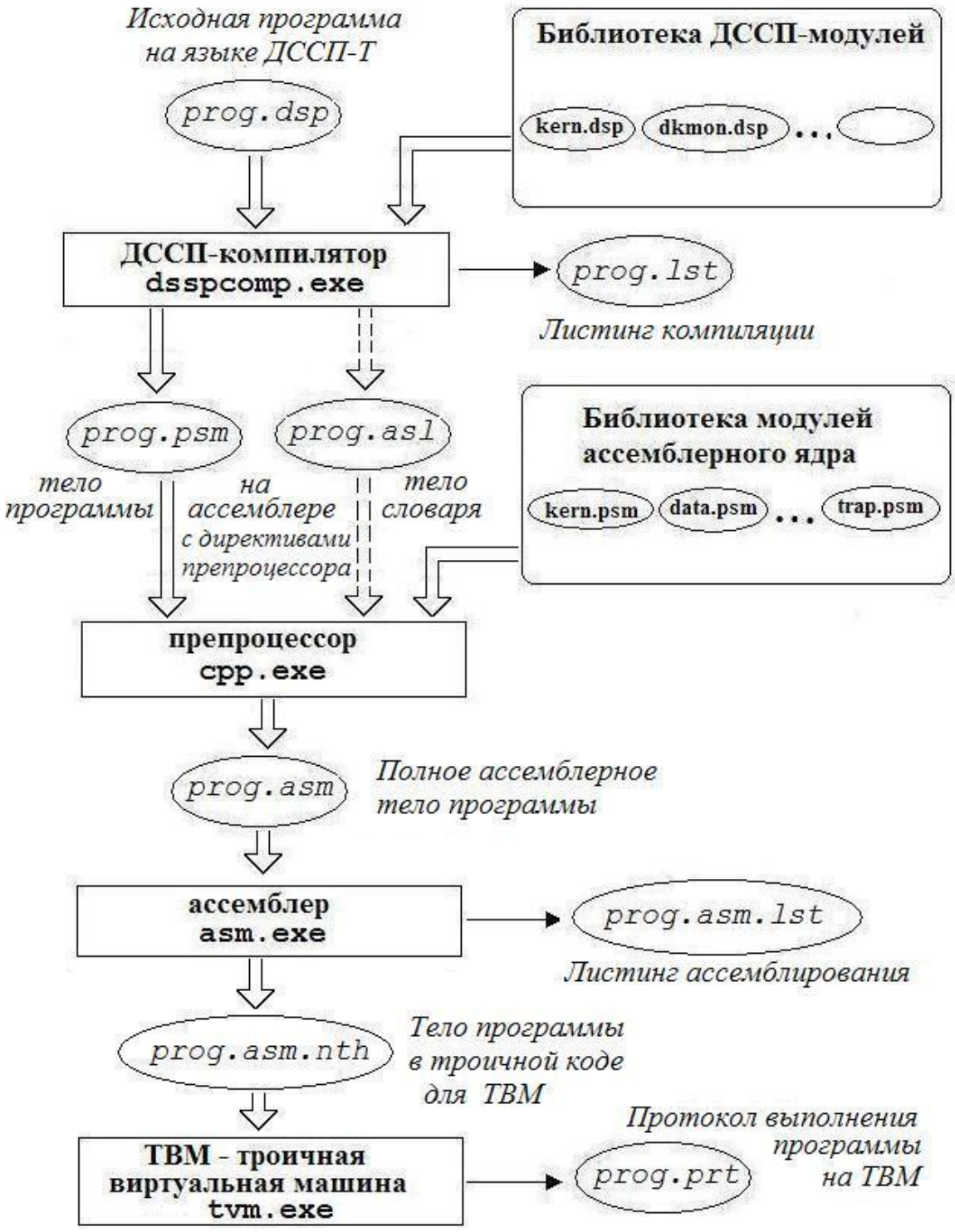

рис.1. Основные компоненты ДССП-ТВМ и схема прогона ДССП-программы.

Далее ассемблерная программа (файл prog.psm) вместе с файлами ассемблерного ядра ДССП обрабатывается препроцессором для получения единой ассемблерной программы (файл

prog.asm), которая затем обрабатывается ассемблером троичного процессора, в результате чего создаётся листинг ассемблирования программы (файл prog.asm.lst) и троичный код (файл prog.asm.nth), готовый для исполнения на троичном процессоре.

Наконец, имитатор троичного процессора (ТВМ) исполняет программу, представленную в троичном коде. Исполнение троичной программы на ТВМ можно отслеживать (на экране) не только в режиме диалога, но и получить протокол её работы (файл prog.prt).

Для автоматизации прохождения всех стадий обработки ДССП-программы (от её компиляции до исполнения на ТВМ), подготовлены соответствующие командные файлы. Так что полный прогон ДССП-программы можно вызвать всего одной командой: dssp-tvm prog .

4. Характеристика возможностей языка ДССП-Т и его библиотеки

Язык ДССП-Т представляет собой ориентированный на троичную машину вариант языка ДССП, в котором удалось сохранить лучшие черты различных версий ДССП [13-22], разработанных для двоичных машин. Рассмотрим основные черты языка ДССП-Т, выделяя те его особенности, которые существенно отличают его от языка ДССП, описанного в [22].

4.1 Основные черты языка ДССП-Т

Синтаксис языка ДССП-Т такой же простой, как и у обычной ДССП. Программа записывается как последовательность слов в свободном формате, разделителями слов являются пробелы (или приравненные к ним символы), а также комментарии. Словарь построен без разделения на подсловари и секции, но в нём можно осуществлять очистку, и обеспечена возможность помечать слова, которые всегда должны оставаться видимыми (неудаляемыми) в словаре.

В языке ДССП-Т допускаются разнообразные варианты комментаторных скобок так, чтобы никакие из символов: **([{ }])** , употребляемых для комментариев, не запрещать программисту использовать в своих словах. Для настройки желаемого вида комментариев предусмотрены служебные слова: **() [] {}** . Допускаются также и комментарии, отделяемые двойными открывающимися скобками: **(([[{{** до конца текущей строки.

ДССП имеет двухстековую архитектуру. Все основные операции осуществляют действия над данными, помещёнными в стек операндов (арифметический стек). Одному элементу стека сопоставлено одно машинное слово из 27-тритов или 3-х трайтов. Стек операндов растёт в сторону увеличения адресов.

Управляющий стек (или стек возвратов) расположен в памяти так, чтобы расти навстречу стеку операндов (в сторону убывания адресов). Элементами стека являются 27-тритные слова, которые, как правило, трактуются как адреса возвратов из процедур. Предполагается, что помимо адресов возвратов в управляющий стек могут помещаться специального вида структуры – так называемые ловушки ситуаций.

Для объявления новых слов, представляющих те или иные сущности в ДССП-программе, используются специальные служебные слова, так называемые слова-компиляторы:

PROGRAM : VALUE VAR VCTR ARR TEXT SITUATION ASM ;ASM

В языке ДССП-Т имя главного слова программы, с которого и предстоит начать исполнение всей программы, принято задавать в первой строке исходной программы после служебного слова **PROGRAM** (пример: **PROGRAM** DKINT).

Слово-процедура в ДССП-программе определяется следующим образом:

: P P1 P2 **...** PN **;**

При своём выполнении такая процедура P будет последовательно вызывать исполнение слов, из которых построено её тело: P1 **...** PN . Причём в ДССП-программе не требуется, чтобы все используемые в таком определении слова должны уже быть определены заранее. В определяемом теле новой процедуры разрешается употреблять и те слова, которые могут быть определены позднее.

В общем виде объявление переменной в ДССП-программе выглядит так:

ИмяТипа **VAR** ИмяПерем {Пример:} TRYTE **VAR** X

Тип переменной зависит от того, какое слово с именем типа употреблялось перед её объявлением. Если тип не был задан, то он принимается по умолчанию (в настоящее время это тип TWORD , задающий троичное слово).

При объявлении вектора (одномерного массива):

```
{N} VCTR ИмяВектора  {Пример:} 10 VCTR Y
```

в стеке задаётся его верхняя граница N (нижняя граница = 0). Так что в данном примере объявляется вектор Y[0:N] с N+1 элементом.

При объявлении многомерного массива вида:

```
{L1,..,Lm,m} ARR ИмяМас {Пример:} 10 20 2 ARR M
```

в стеке задаются его размерность m и m верхних границ по каждому измерению. Так что в примере объявляется массив M[0:10,0:20].

Тип элементов объявляемого массива определяется тем словом с именем типа, которое было употреблено непосредственно перед объявлением.

Первоначально в языке ДССП-Т был предусмотрен лишь минимальный набор префиксных операций над переменными и массивами: операции взятия значения (по имени), адреса **'** и присвоения значения **!** . А в базовом словаре языка для объявления переменных и в качестве элементов массивов предусмотрены лишь слова-типы, представляющие возможность работать с трайтами, двухтрайтовыми значениями и троичными словами:

TRYTE DTRYTE TWORD

При этом предполагалось, что язык будет дополнен средствами объектно-ориентированного программирования (см. п. 4.2), которые позволят расширять набор префиксных операций (методов) для имеющихся типов данных, а также создавать новые типы данных.

В языке ДССП-Т помимо десятеричных и 16-ных чисел можно задавать также троичные и девятеричные числа, характеризующие значения в так называемой троичной симметричной системе счисления. Каждое из таких чисел опознаётся по специальным префиксам:

```
{девятеричные числа:} #8765 0tqwer 0TQWER #1234
{троичные числа:} .++0--. .++0-- 0n++0-- 0N++0--
{16-ные числа:} $13579ace 0x1290abcd 0X345678EF
{10-ные числа:} 56789 +24680 -13579
```

Большие и малые латинские буквы в записи числа не различаются, а для представления отрицательных девятеричных цифр (-1,-2,-3,-4) можно использовать как цифры (8765), так и буквы (qwer или QWER).

Для представления символьных констант используется общепринятая форма их записи в апострофах ‘X’ (как в языках Си и Паскаль).

Хотя числовые константы-литеры автоматически распознаются (по своим специфическим изображениям) как особые слова, в языке ДССП-Т можно объявить слово, представляющее константу, придав ему нужное имя:

```
{N} VALUE ИмяКнст  {Пример:} 10 VALUE Ten
```

При таком объявлении в стеке задаётся то значение константы N (10), которое будет помещаться в арифметический стек при вызове слова-константы (Ten).

В языке ДССП-Т можно определить и слово, задающее константу-строку:

```
TEXT ИмяКС Строка  {Пример:} TEXT Red "Красный"
```

При вызове имени такой константы-строки в арифметический стек будут помещаться её параметры: адрес начала строки и длина (количество символов).

В языке ДССП-Т сохранены привычные обозначения слов для операций целочисленной арифметики, но выполняются они над числами, представленными в троичной симметричной системе счисления, так что результаты некоторых операций (например, сдвигов и делений с остатком), могут оказаться иными, чем в двоичной машине:

```
{деление с остатком:} 47 / 3 { [16,-1] вместо [15,2] }
{сдвиг вправо= /3  :} 47 SHR { [16] вместо [23] }
{сдвиг влево = *3  :} 13 SHL { [39] вместо [26] }
```

В языке ДССП-Т нет слов (**& &0 '+' INV**), выполнявших привычные действия над отдельными битами двоичного машинного слова. Вместо них присутствуют слова,

позволяющие выполнять разнообразные действия над тритами троичного машинного слова:

TMIN TMAX TADD TMUL NEG

Для доступа к одиночным и двойным трайтам, а также к словам троичной памяти в ДССП-Т предусмотрены операции чтения: **@T @TT @W** и записи: **!T !TT !W** . Причём для продвижения по памяти к очередному троичному слову надо добавлять 3 к значению адреса.

В языке ДССП-Т представлен полный ассортимент ДССП-команд условного вызова процедур, предусмотренных для организации ветвлений в ДССП-программе:

IF- IF0 IF+ BR- BR0 BR+ BRS BR ELSE

Чтобы обеспечить возможности вычисления логических условий, эквивалентные средствам построения логических выражений языков высокого уровня, в ДССП-Т имеются операции отношений для сравнения двух чисел: **< <= = <> >= >** , а также операции логических связок: **NOT AND OR** , вырабатывающие значение 1 (ДА) или 0 (НЕТ).

Кроме того, в языке ДССП-Т предусмотрены также слова-операции для сравнения двух чисел и оценки знака числа: **CMP SGN** , предусматривающие троичные результаты: -1 (меньше), 0 (равно), +1 (больше), которые далее могут использоваться для вычисления сложного троичного условия путём применения потритных операций:

TMIN TMAX TADD TMUL NEG

В ходе создания и развития ДССП применялись две различные концепции организации циклов. Первая основывается на том, что условие цикла должно задаваться в самой команде цикла. Вторая предполагает, что выход из цикла может осуществляться с любого места тела цикла по специальным командам, а сама команда цикла должна лишь обеспечивать повторный вызов процедуры её тела.

Первая концепция обозначилась командой **DWON** уже в первой версии ДССП-НЦ [13], которая по сути унаследовала команду цикла **<DW P>** машины “Сетунь-70”. Эта команда многократно вызывала стоящую следом процедуру, пока вершина стека оставалась ненулевой. Затем было предложено (в ДССП-ПМ [15]) усовершенствовать эту команду цикла и выделить вычисление условия в отдельную процедуру, а значение условия удалять из стека после его проверки. Так появилась конструкция:

WHILE условие DO тело

которая впоследствии (в ДССП/С [18]) превратилась в команду:

условие DW тело

Вместе с ней были предложены команда **DO-** для организации цикла со счётчиком, а также команда бесконечного цикла **LOOP**, которые исполняют эти виды циклов эффективней, чем универсальная команда **DW .**

Вторая концепция организации циклов была предложена в версии ДССП-80 [14]. В ней появились команды для повторного вызова процедуры тела цикла заданное или бесконечное число раз: **DO RP** . Были предусмотрены специальные команды для экстренного выхода из цикла по условию или безусловно: **EX- EX0 EX+ EX** . Эти команды могли осуществлять требуемый выход из цикла с любой точки как самой процедуры тела цикла, так и вызванных ею процедур.

Обе концепции организации циклов применялись в ДССП на протяжении достаточно длительного периода и потому стали привычными для многих разработчиков ДССП-программ. Чтобы предоставить каждому из них желаемый способ организации циклов, в языке ДССП-Т обеспечены команды циклов обеих этих концепций:

DW DO- LOOP **RP DO EX EX+ EX- EX0**

В языке ДССП-Т поддерживается полный вариант структурированного механизма обработки исключительных ситуаций как в ДССП/С [18]. Предусмотрены команды для установки трёх типов реакций (**N**otify, **E**scape, **R**etry) на заданную (следом по телу) ситуацию (а); на ситуацию, переданную через стек (б); на произвольную ситуацию **ANY** (в)**:**

(а)	(б)	(в)
ON S! React	**ON_** S! **_NOTIFY** React	**ON ANY** React
EON S! React	**ON_** S! **_ESCAPE** React	**EON ANY** React
RON S! React	**ON_** S! **_RETRY** React	**RON ANY** React

Предварительно требуется объявить ситуацию как переменную (или объект) особого вида. Для этого используется соответствующее предложение-декларация:

```
            SITUATION ИмяСит КонечнаяРеакция
{Пример:} SITUATION S! FReactOnS!
```

При таком объявлении помимо имени ситуации задаётся также имя слова-процедуры конечной реакции, которая будет вызвана (по типу **N**otify) в случае, когда возбуждённая ситуация так и не будет обработана в ДССП-программе (т.е. в случае, когда на возбуждённую ситуацию не встретится ни одной установленной реакции).

Слово из имени ситуации `S!` вызывает возбуждение ситуации. Можно также возбудить ситуацию, переданную через стек, употребив команду **_RAISE** :

```
ON_ S! {S!}   ...   {S!} _RAISE {}
```

или повторно возбудить ту же ситуацию командой **RERAISE .**

Программа, разработанная в ДССП-ТВМ, должна уметь общаться с внешними устройствами, которые ей может предоставить имеющаяся в настоящее время среда исполнения. В качестве минимального набора устройств ввода и вывода, доступных исполняемой ДССП-программе, в настоящее время ей обеспечивается диалоговое общение через консоль терминала и операции доступа к файлам операционной среды для последовательного чтения и записи.

При построении тел некоторых новых слов ДССП-программы может возникнуть потребность запрограммировать такие действия, выразить которые возможно удастся лишь командами базовой машины (т.е. командами ТВМ). Поэтому язык ДССП-Т предоставляет возможность определить новое ДССП-слово, задав процедуру его исполнения на языке ассемблера ТВМ. Такое определение задаётся между служебными словами: **:ASM** и **;ASM .**

4.2 Средства объектно-ориентированного программирования

С помощью таких ООП-средств в языке ДССП-Т предусмотрена [23] возможность определять новые типы данных, задавая при этом и разрешённые операции над ними.

Объявление нового типа как структуры начинается словом **STRUCT:** и завершается словом **;STRUCT** , а располагаемые между ними определения объектов данных (с помощью слов **VAR VCTR ARR**) объявляют в этом случае не самостоятельные переменные и массивы, а поля определяемой структуры:

```
STRUCT: ИмяТипаСтрукт
  VAR ИмяПоля1 . . . VAR ИмяПоляК
;STRUCT
```

```
STRUCT: TPoint
  TRYTE VAR .X  TRYTE VAR .Y
;STRUCT
```

После объявления нового типа данных можно использовать его имя для объявления объектов (переменных и массивов) такого типа:

```
TPoint VAR P        TPoint 100 VCTR V {вектор точек}
```

Имеющиеся в языке ДССП-Т префиксные операции над переменными и массивами можно теперь применять и для доступа как ко всему объекту типа структуры целиком, так и к отдельным его полям:

```
P {'P} 3 ! V {V[3]:=P}
     5 V ! P {P:=V[5]}
   i V j ! V {V[j]:=V[i]}
```

```
P .X {P.X} 55 P ! .Y {P.Y:=55}
P .X 6 V ! .X {V[6].X:=P.X}
8 V .Y P ! .Y {P.Y:=V[8].Y}
```

Ещё одним средством построения новых типов данных в языке ДССП-Т являются классы. Объявление нового типа как класса начинается со слова **CLASS:** и заканчивается словом **;CLASS** , а внутри такого объявления можно задавать не только поля (как и при объявлении структуры), но и методы класса:

```
CLASS: ИмяТипаКласса
     VAR ИмяПоля1   . . .
     VAR ИмяПоляК
  METHOD ИмяМетодаI . . .
  METHOD ИмяМетодаJ
;CLASS
```

```
CLASS: TPoint
    VAR .X  VAR .Y
    METHOD Init  METHOD Show
    METHOD= Show Print
  4 METHOD# >>
;CLASS
```

Словом **METHOD** объявляется новая операция, которую можно будет совершать над объектом этого типа-класса помимо тех методов (префиксных операций **!** и ‘), которые над ним уже предусмотрены изначально (как и над структурами). Слово **METHOD=** позволяет назначить новое имя методу, используя его прежнее имя, а слово **METHOD#** – задав номер метода. Так в примере для метода Show было назначено два новых имени: Print и >> .

Далее необходимо с помощью компилирующих слов **:M:** и **:M=** определить процедуры, которые будут назначены для исполнения новых методов над объектами объявленного класса:

```
TPoint :M: Init {P} 0 C2 ! .X 0 E2 ! .Y { } ;
       : .Show {P} '(' TOB C .X . D ',' TOB .Y . D ')' TOB {};
TPoint :M= Show .Show
```

Имя типа класса следует задать перед употреблением слова **:M:** (или **:M=**). Слово **:M:** создаёт тело новой процедуры, компилируя его как и слово **:** (до **;**), а слово **:M=** принимает имя уже существующей процедуры (см. **.**Show), чтобы назначать её в качестве исполнителя метода.

Внутри объявления нового типа как класса с помощью слова **SUBCLASS** можно задать, какой уже известный тип использовать для наследования:

```
CLASS: T3DPoint SUBCLASS TPoint
    VAR .Z
;CLASS
 T3DPoint :M: Init {P} C Init AS TPoint 0 E2 ! .Z ;
 T3DPoint :M: Show {P} {реализуем по-новому:}
             '[' TOB C .X . D ',' C .Y . D ',' .Z . D ']' TOB ;
```

Такое наследование означает, что у нового объявляемого типа изначально считаются уже заданными все поля и методы, которыми обладает наследуемый тип, после чего можно добавлять к его определению новые поля, задавать новые методы, а также переопределять исполнителей прежних методов (Init, Show).

4.3 ДССП-библиотека

Возможности языка ДССП-Т не ограничиваются только словами базового словаря. Словарь компонуемой программы можно дополнить определениями слов, заготовленных в файлах стандартной библиотеки (ДССП-библиотеки). Для этого достаточно загрузить их определения из требуемого файла командой:

LOAD ИмяФайла

В настоящее время ДССП-библиотека состоит из следующих частей: ДССП-ядра (файл kern.dsp), диалогового командного монитора (файл dkmon.dsp), диалогового интерпретатора (файл dkint.dsp), а также модулей, обеспечивающих сопрограммный механизм (файл context.dsp) и простой монитор (файл spmon.dsp) для организации параллельных процессов на основе такого механизма.

Ядро ДССП складывается как бы из двух частей: первая часть содержит тела процедур ядра, составленные на языке ассемблера ТВМ (в файлах kern.psm и др.), а вторая - определения слов ядра на языке ДССП-Т (в файлах kern.dsp и др.). Ассортимент этих слов весьма разнообразен и достаточно обширен. В настоящее время он охватывает:

- определения ряда констант;
- операции вывода на терминал управляющих символов, строк, чисел ;
- операции ввода с терминала строк;
- операции преобразования строки в число заданного формата;
- операции преобразования числа в строку заданного формата;
- операции сравнения и копирования строк;
- операции проверки символов;
- операции над нумерованными тритами троичного слова;
- определения новых типов данных **ACT** и **TSET** с операциями над ними;
- операции с файлами операционной среды ТВМ;
- а также разнообразные вспомогательные операции.

Диалоговый Командный МОНитор (далее ДКМОН) предназначен для исполнения скомпонованной ДССП-программы в режиме привычного диалога. Такой диалог предполагает, что в ответ на стандартное ДССП-приглашение (в виде символа * с новой строки) можно последовательно ввести в командной строке и исполнить любое слово скомпонованной программы (или несколько слов), просмотреть содержимое стека операндов, стека возвратов, ячеек памяти.

ДКМОН не обеспечивает всех тех функций, которыми обычно должен обладать внешний интерпретатор ДССП. Кроме слов, приготовленных в компонуемой программе, он умеет распознавать только числа (в этом случае в качестве действия он посылает в стек значение принятого числа). Ни комментарии, ни строки вида **"…"**, ни текст для печати вида **."…"** он не воспринимает. На любое непонятное слово он печатает сообщение об ошибке.

ДКМОН принимает слова из командной строки по одному. Встретив известное слово, ДКМОН сначала проверяет, можно ли это отдельное слово исполнить. Если нельзя, сообщает об ошибке, что это неисполнимое слово.

ДКМОНу известны только те слова, которые остались видимыми в словаре после компоновки программы. Из них исполнимыми считаются только те слова, которые могут выполняться в одиночку. В частности, ДКМОН не позволяет исполнить введённую в командной строке последовательность слов, содержащую какие-либо управляющие команды:

IF+ BR- BRS DW DO RP

или префиксные операции **` !** над переменными, либо какие-то другие слова, требующие доступа к последующим словам по телу сшитого кода.

Модуль ДКМОНа можно использовать и просто как ДССП-библиотеку, загрузив в программу содержащиеся в нём определения слов (командой **LOAD**) без обязательного последующего запуска монитора при старте программы.

4.4 Примеры ДССП-программ

4.4.1 Примеры использования цикла DW с условием

```
PROGRAM DEMO_DW
LOAD kern
  ." примеры использования цикла DW  " CR
{{ ------------ Наибольший общий делитель
  : NOD {{  [ x,y ] => [ z ] , z= nod(x,y)
        x<>y? DW x-y|y-x D ;
   : x<>y? {[ x,y ]} C2 C2 <> ;
   : x-y|y-x {[ x,y ]} x>y? BR+ x-y y-x ;
    : x>y?  {[ x,y ]} C2 C2 > ;
    : x-y {[ x,y ]} E2 C2 - E2 {[ x-y,y ]} ;
    : y-x {[ x,y ]} C2 - {[ x,y-x ]} ;
{{ ------------ Число Фиббоначи (n-ое по порядку)
{{              F(0)=1, F(1)=1; F(k+1)= F(k)+F(k-1)
  : Fib {{  [ n ] => [ F(n) ]
        1 1 E3 1- {[ FP,FT,k ]} >0? DW newF D E2D ;
    : >0? C 0 > ;
    : newF {[ FP,FT,k ]} E2 E3 C3 + E2 1- {[ FT,FN,k-1 ]} ;
{{ ------------ Число Фиббоначи (первое по порядку, превыщающее M)
{{              F(0)=1, F(1)=1; F(k+1)= F(k)+F(k-1)
  : Fib> {{  [ M ] => [ Fi ] , Fi=F(i) > M
         1 1 {[ M,FP,FT ]} <=M? DW newFi E3 DD ;
    : <=M? C C4 <= ;
    : newFi {[ FP,FT ]} E2 C2 + {[ FT,FN ]} ;
{{ ------------ Факториал
  : Fctrl {{  [ n ] => [ n! ] , n!= 1*2*...*n
          1 E2 {[ F,k ]} >0? DW F*k D ;
    : F*k {[ F,k ]} E2 C2 * E2 1- {[ F*k, k-1 ]} ;

  : DEMO_DW 77 55 NOD . 7 Fib . { 21 } 20 Fib> . 5 Fctrl . ;
UNDEF CLEAR
```

4.4.2 Примеры использования цикла DO- с параметром-счетчиком

```
PROGRAM DEMO_DO- { DKMON }
[] [*****   примеры использования цикла DO- с параметром-счетчиком.  *****]
[***   Вывод строки заглавных латинских букв ***]
:: : .LatString 26 DO- .LatLetter ;
    : .LatLetter [i] 'Z' C2 - TOB [i] ; [печать i-ой латинской буквы]
[***   Сумма квадратов всех целых чисел от 0 до N ***]
:: : SumKv [N] 0 E2 [S,N] 1+ DO- SumI [S] ;
    : SumI [S,i] E2 C2 C * [i,S,i*i] + E2 [S+i*i,i] ; [ i=N,N-1,..,1,0. ]
[***   Сумма квадратов всех целых чисел заданного диапазона (M..N) ***]
:: : SumKvD [M,N] C2 - 1+ 0 E2 [M,0,N-M+1] DO- SumID [M,S] E2 D [S] ;
    : SumID [M,S,i] E2 C3 C3 + [M+i] C *
           [M,i,S,(M+i)*(M+i)] + E2 [M,S+(M+i)*(M+i),i] ; [ i=N-M,..,1,0. ]
[***   Вычисление факториала ***]
::  : Factor [N] 1 E2 [1,N] DO- FI [N!] ;
     : FI [F,I] E2 C2 1+ * E2 [F*(I+1),I] ; [при I=N-1,..,1,0]
[***   Вычисление полинома по схеме Горнера ***]
    [ P(X) = (..(A9*X+A8)*X+ ..)*X+A0 ]
      9 VALUE n  [задает размер массива коэффициентов ]
    [массив коэффициэнтов A(0..9), определяющих полином : ]
    LONG CNST A 5 -2 0 1 0 0 0 0 0 0 ; [ X^3-2X+5 ]
   : Plnm [X] n A [A(n)] n [X,A(n),n] DO- PI [P(X)] ;
    : PI [X,S,i] E2 C3 * [X,i,S*X] C2 A + [S*X+A(i)] E2 [X,S,i] ;
[***   Сортировка массива методом простого выбора ***]
  9 VALUE N [размер массива]  N VCTR S [сортируемый массив S(0..N) ]
::  : SORT N 1+ [N+1] DO- SORTi ;
     : SORTi [i] MAXi  [i,k] EXCHANGEi D [i] ; [i=N,..,0]
      : MAXi [определяет индекс максим.элемента среди S0..Si ]
             [i] C [i,k] C2 [i] DO- MAXj [i,k] ;
       : MAXj [k,j] C2 S C2 S [S(k),S(j)] < IF+ k:=j [k,j] ;
        : k:=j [k,j] E2 D C [j,j] ;
      : EXCHANGEi [ переставляет i-ый и k-ый элементы массива ]
           [i,k] C S C3 S [i,k,S(k),S(i)] C3 ! S [S(i)=>S(k)]
                [i,k,S(k)] C3 ! S [S(k)=>S(i)]  [i,k] ;
[***   ввод массива S из стека ***]
::  : GETS [S0,S1,..,SN] N 1+ [N+1] DO- GETSi ;
     : GETSi [Si,i] E2 C2 ! S [i] ;
 [***  вывод массива S на терминал ***]
::  : PUT  N 1+ [N+1] DO- PUTi ;
     : PUTi [i] N C2 - S .N "," [i] ;
      : .N C IF- .- .N2 ;   : .N2 2 TON10 ;  : .-  ."-" NEG ;
[***   ввод массива S с терминала ***]
::  : GET  N 1+ [N+1] DO- GETi ;
     : GETi [i] ."=" TIN N C3 - ! S [i] ;
::  : DEMO_DO-." LatString=" .LatString CR
         ." SumKv(10)=" 10 SumKv 4 TON10 CR
         ." Factor(7)=" 7 Factor 6 TON10 CR
         ." Plnm(3)="   3 Plnm   6 TON10 CR
         ." Сортировка массива " CR 55 23 67 12 99 07 66 78 33 47 GETS
         ."S[0:9]=" PUT CR ."SORT_S[0:9]" SORT CR ."S[0:9]=" PUT CR    ;
  UNDEF  CLEAR
$$
```

4.4.3 Примеры программ перестановок в в одномерном массиве

```
PROGRAM Test_PrstVctr { DKMON }
LOAD DKMON
{----- Ввод массива из стека -----}
::  : GetVctr {A0,A1,..,An,'A,k} {{'A=адрес,k=длина (кол-во эл-тов)
  {A0,..,An,'A,n+1} DO- Get_A[i] D { } ; {for i=n,..,1,0 A[i]:=value from stack}
    : Get_A[i] {.., X, 'A, i } E2 E3 {.., 'A, i, X } C3 C3 *3 +
        {.., 'A, i, X, 'A+i*3 } !W { A[i]:=X } {.., 'A, i } ;
{----- Печать массива как десятичных чисел -----}
::  : PrintVctr {'A,k} DO- Print_A[i] D { } ; {{ 'A=адрес,k=длина (кол-во эл-тов)
    : Print_A[i] { 'A[i], i } C2 3+ { 'A[i+1] } E3
        { 'A[i+1], i, 'A[i] } @W 0 TON10 SP  { 'A[i+1], i } ;
{------- Инверсия одномерного массива -------------}
```

```
:: : InvrVctr { A,k } {{ A - адрес массива, k= кол-во элементов , n=k-1
       {{ массив A[0:n]= [ A0,A1,..,An ] => [ An,..,A1,A0 ]
       { A,k } C 1- E2 /2 { A,n, k div2 } DO- A[i]<=>A[n-i] DD { } ;
    : A[i]<=>A[n-i] { for i= k div2,..,1,0 }
       { A,n,i } C3 C3 C3 - *3 + {..,A+(n-i)*3 } C @W {..,A[n-i] }
       { A,n,i,'A[n-i],A[n-i] } 5 CT C4 *3 + {..,A+i*3 } C @W {..,A[i] }
       { A,n,i,'A[n-i],A[n-i],'A[i],A[i] } E3 E2 !W { A[i]:=A[n-i] }
       { A,n,i,'A[n-i],A[i] } E2 !W { A[n-i]:=A[i] }      ;
{--- Циклический сдвиг одномерного массива влево. Простой вариант. ---}
:: : LShiftVctr {A,n,s} {{A-адрес массива, k=кол-во эл-тов, s=число позиций сдвига
       {{ массив A[0:n]= [ A0,A1,..,A(n-1) ] => [ As,..,A(n-1),A0,..,A(s-1) ]
       { A,n,s } DO- LShift1Vctr DD { } ;
    : LShift1Vctr { A,n,i } C3 @W { A0 } C4 {'A[0] } C4 { n } 1- DO- LShift_A[j]
       { A,n,i, A0, 'A[n] } !W { A[n]:= A0 } { A,n,i } ; { for i= s-1,.,1,0 }
    : LShift_A[j] {.., 'A[j],j } { j=n-1,..,1,0 }
       {.., 'A[j],j } C2 3+ C @W {.., 'A[j],j, 'A[j+1], A[j+1] } E2 E4
       {.., 'A[j+1],j, A[j+1], 'A[j] } !W { A[j]:=A[j+1] } {.., 'A[j+1],j } ;
{--- Циклический сдвиг одномерного массива влево. Эффективный вариант. ---}
:: : LShift!Vctr {A,n,s}{{A=адрес массива, k= кол-во эл-тов,s=число позиций сдвига
        { A,n,s } C3 C2 {.., A,s } InvrVctr
        { A,n,s } C3 C2 *3 + C3 C3 - {.., 'A[s],n-s } InvrVctr
        { A,n,s } D { A,n } InvrVctr ;
   100 VCTR A  : 'A 0 ' A ;
:: : Test_PrstVctr CR ." --- Перестановки в одномерном массиве (векторе) ---" CR
       10 11 22 33 44 55 66 77 88 99  'A 10 GetVctr  'A 10 PrintVctr CR
       'A 10 InvrVctr ." A[0:9]=> "  'A 10 PrintVctr  CR
       10 11 22 33 44 55 66 77 88 99 'A 10 GetVctr  'A 10 PrintVctr CR
       'A 10 3 LShiftVctr  'A 10 PrintVctr CR
       'A 10 2 LShift1Vctr 'A 10 PrintVctr CR
       'A 10 1 LShift1Vctr 'A 10 PrintVctr CR
       'A 10 0 LShift1Vctr 'A 10 PrintVctr CR
          10 11 22 33 44 55 66 77 88 99 'A 10 GetVctr  'A 10 PrintVctr CR
       'A 10 4 LShift!Vctr 'A 10 PrintVctr CR    ;
  UNDEF  CLEAR
$$
```

4.4.4 Троичная проверка попадания точки в отрезок

```
PROGRAM Test_TSEG
LOAD KERN
  ." Test_TSEG Троичная проверка попадания точки в отрезок " CR
{{ { a,b, x } TSEG { r } {{ r = +1 (a<x<b) или 0 (x=a|x=b) или -1 (x<a|b<x)
   :: : TSEG { a,b,x } C E4 { x,b, x,a } CMP { x?a } E3
             { x?a, b,x } CMP { x?a,b?x } TMIN { r } ;
  10 VALUE N   { константа, задающая количество точек }
   VAR A VAR B { переменные, задающие границы отрезка [A,B] }
   9  VCTR V   { одномерный массив точек V[0:9] }
  { приём значений точек из стека данных в массив V }
  : GetV {V0,V1,..,V9} N DO- GetV[i] ;
    : GetV[i] {..,Vi,i} E2 C2 ! V {..,i} ; { i=9,8,..,1,0 }
  : .10 {Num} 0 TON10 {Num} ; { печать 10-ного числа }
  { проверка попадания точек массива в отрезок: }
  : .CheckV ."--- для отрезка [" A .10 ',' TOB B .10 ']' TOB CR CheckV ;
  : CheckV N DO- CheckV[i] { i=9,8,..,1,0 } ;
   : CheckV[i] {i} A B ." точка " C3 V {i,A,B,V[i]} C .10
    {i,A,B,V[i]} TSEG {-1/0/1} BRS )A,B( |A,B| (A,B) CR {i} ;
    : )A,B( ." вне отрезка " ;    : |A,B| ." на границе отрезка " ;
    : (A,B) ." внутри отрезка " ;
:: : Test_TSEG ."===== Проверки попадания точек в отрезок: =====" CR
     { задание массива точек V[0:9]=} 5 4 7 3 8 -4 -7 0 6 -6 GetV
     { задание границ отрезка: }  4 ! A 7 ! B  { A:= 4; B:= 7 }
     { проверки попадания точек массива в отрезок:  [4,7] } .CheckV
     { задание границ отрезка: } -6 ! A 6 ! B  { A:=-6; B:=+6 }
     { проверки попадания точек массива в отрезок: [-6,6] } .CheckV  ;
 CLEAR UNDEF
 $$
```

4.4.5 Новые типы данных очередь (QUEUE) и дек (DEQ) как классы

```
PROGRAM DKMON {QDEMO}
[]
 [* ОЧЕРЕДЬ как Пример построения нового типа данных с помощью КЛАССА *]
LOAD DKMON
   CR ." Новый Тип данных QUEUE (Очередь) " CR :: : QDEMO? 1 ;
 [--- ОЧЕРЕДЬ элементов типа TWORD (3-х трайтовых целых) ---]
 100 VALUE MaxLen [ максимальная длина очереди ]

                [--- Объявление класса ОЧЕРЕДЬ ---]
 ." -------- Объявление класса  QUEUE " CR
:: CLASS:  QUEUE  SUBCLASS TCLASS
     [--- поля данных, составляющих очередь ---]
      VAR .cnt [ кол-во элементов в очереди ]
      VAR .n  [ индекс-указатель начала очереди в массиве ]
      VAR .k  [ индекс-указатель конца очереди в массиве ]
      100 VCTR .MI  [ массив для хранения элементов очереди ]
     [--- методы , представляющие операции над очередью ---]
   ::  METHOD Init    [ инициировать, начать работу очереди ]
   ::  METHOD Done    [ завершить, закончить работу очереди ]
   ::  METHOD Show    [ показать на экране состояние очереди ]
   ::  METHOD Put     [ поместить элемент x из вершины стека в очередь Q ]
                    [ Пример вызова: {x} Put Q  ]
   ::  METHOD Get     [ взять элемент y из очереди Q на вершину стека ]
                    [ Пример вызова:  Get Q {y} ]
   ::  METHOD Empty? [ проверить, пуста ли очередь ]
   ::  METHOD Full?  [ проверить, полна ли очередь ]
   ::  METHOD= ! =>  [  копирование очереди ]
   ;CLASS

 ." -------- Объявление ситуаций " CR
      SITUATION  Empty! .Empty!   : .Empty! ." Empty Queue! " HALT ;
      SITUATION  Full!  .Full!    : .Full!  ." Full Queue! "  HALT ;

               [--- Реализация методов класса ОЧЕРЕДЬ ---]
  ." -------- Реализация методов QUEUE " CR
 QUEUE :M: Init [Q] 0 C2 ! .cnt                [ Q.cnt:=0 ]
                    0 C2 ! .k 1 E2 [Q] ! .n  [ ] ; [ Q.k:=0; Q.n:=1 ]
 QUEUE :M: Done [Q] D ;
 QUEUE :M: Show [Q] ."Queue:"  ShowQueue [] ;
           : ShowQueue [Q] ." (" C .cnt 2 TON10 .")= < "
                  C .n C2 .cnt [Q,n,cnt] DO- ShowItem DD ."> " []  ;
           : ShowItem [Q,t,i] C2 C4 [...,t,Q] .MI [M(t)] 3 TON10 ."  "
                     [Q,t,i] E2 +1mod E2 [Q,(t+1)mod,i] ;
 QUEUE :M: Put [x,Q] C .Full? IF+ Full! [ проверка на искл.ситуацию ]
                     C .k +1mod C C3 ! .k [ Q.k:=(Q.k+1)mod Maxlen ]
               [x,Q,k] C2 .cnt 1+ C3 ! .cnt      [ Q.cnt:=Q.cnt+1 ]
                      E2 [x,k,Q] ! .MI           [ Q.M(k):=x ]
                   [ ] ;
QUEUE :M: Get [Q] C .Empty? IF+ Empty! [ проверка на искл.ситуацию ]
                     C .n C2 .MI E2    [ y:= Q.M(Q.n) ]
            [y,Q] C .cnt 1- C2 ! .cnt [ Q.cnt:=Q.cnt-1 ]
             C .n +1mod E2 [y,n+1,Q] ! .n [Q.n:=(Q.n+1)mod Maxlen]
              [y] ;
        : .Empty? [Q] .cnt 0 <= [1/0] ;     [ Q.cnt<=0 ? ]
 QUEUE :M= Empty? .Empty?
        : .Full? [Q] .cnt MaxLen >= [1/0] ;     [ Q.cnt>=MaxLen ? ]
 QUEUE :M= Full? .Full?

                       [--- Вспомогательные процедуры  ---]
 : +1mod [z] 1+ C MaxLen >= IF+ T0 [z+1 или 0, если z>=MaxLen-1 ] ;
 : -1mod [z] C IF0 +MaxLen 1-      [z-1 или MaxLen-1, если z=0 ]  ;
  : +MaxLen MaxLen + ;
```

```
[-------  Примеры объявленных экземпляров классов QUEUE ----------]
 ." Примеры использования QUEUE " CR
  QUEUE VAR Q    QUEUE VAR Q2
  9 QUEUE VCTR VQ  [ массив очередей VQ(0:9) ]
  3 4 2 QUEUE ARR AQ  [ матрица очередей AQ(0:3,0:4) ]
 :: : QDEMO CR ."Демонстрация операций с очередями " CR
       Init Q
       33 Put Q 44 Put Q 55 Put Q 66 Put Q 77 Put Q ."Q= " Show Q CR
       9 DO- InitVQ(i)
       Q 3 => VQ [ VQ(3):= Q ] -99 3 Put VQ  ."VQ(3)= " 3 Show VQ CR
       -44 4 Put VQ  3 Get VQ . SP 4 Put VQ 3 Get VQ . SP 4 Put VQ CR
       ."VQ(3)= " 3 Show VQ CR ."VQ(4)= " 4 Show VQ CR
      [ действия с одним элементом матрицы очередей: ]
      1 2 Init AQ     [ A(1,2).Init  ]
      4 VQ 1 2 => AQ  [ A(1,2):=VQ(4) ]
      666 1 2 Put AQ  [ AQ(1,2).Put(666) ]
       ."AQ(1,2)= " 1 2 Show AQ CR   ;
     : InitVQ(i) [i] C Init VQ [i] ; [ инициализация i-ой очереди ]
```

```
." Новый Тип данных DEQ на основе QUEUE " CR  :: : DQDEMO? 1 ;
                  [--- Объявление класса ДЕК ---]
 [--- ДЕК элементов типа LONG (4-х байтовых целых) ---]
:: CLASS:  DEQ  SUBCLASS QUEUE [ наследует класс QUEUE ]
     [--- методы , дополняющие операции над очередью ---]
   ::  METHOD Put_ [ поместить элемент из вершины стека в начало очереди ]
   ::  METHOD Get_  [ взять элемент с конца очереди на вершину стека ]
   ::  METHOD Count [ определить количество элементов в очереди ]
   ;CLASS
               [--- Реализация методов класса ДЕК ---]
 DEQ :M= Count .cnt
 DEQ :M: Show [Q] ."Double-End-"  Show AS QUEUE [] ;
 DEQ :M: Put_ [x,Q] C .Full? IF+ Full! [ проверка на искл.ситуацию ]
                    C .n -1mod C C3 ! .n [ Q.n:=(Q.n-1)mod MaxLen ]
               [x,Q,n] C2 .cnt 1+ C3 ! .cnt [ Q.cnt:=Q.cnt+1 ]
                       E2 [x,n,Q] ! .MI     [ Q.M(n):=x ] [ ] ;
DEQ :M: Get_ [Q] C .Empty? IF+ Empty! [ проверка на искл.ситуацию ]
                     C .k C2 .MI E2   [ y:= Q.M(Q.k) ]
           [y,Q] C .cnt 1- C2 ! .cnt  [ Q.cnt:=Q.cnt-1 ]
           C .k -1mod E2 [y,k+1,Q] ! .k  [Q.k:=(Q.k-1)mod MaxLen] [y] ;
 [-------  Примеры объявленных экземпляров класса DEQ ----------]
 ." Примеры использования  QUEUE и DEQ " CR
  DEQ VAR DQ  [ дек , который может использоваться и как очередь ]
 104 VCTR ZA [ произвольная область памяти, которая используется как дек ]
       : Z 0 ' ZA ;  [ её адрес ]
 :: : DQDEMO CR ."Демонстрация операций с очередями и деком " CR
       Init Q
       33 Put Q  44 Put Q  55 Put Q  66 Put Q  77 Put Q ."Q= " Show Q CR
       Init DQ  Q DQ => AS QUEUE [ DQ:= Q as QUEUE ]
       88 Put DQ 22 Put_ DQ 11 Put_ DQ 99 Put DQ ."DQ= " Show DQ CR
       Get DQ 3 TON10 ." <- " Get_ DQ 2 TON10 ." -> "
       Get DQ 3 TON10  ." <- " CR  ."DQ= " Show DQ CR
       DQ => Q [ Q:= DQ as QUEUE ] Get Q 3 TON10 CR ."Q= " Show Q CR
       9 DO- InitVQ(i)
       Q 3 => VQ [ VQ(3):= Q ] -99 3 Put VQ  ."VQ(3)= " 3 Show VQ CR
       -44 4 Put VQ  3 Get VQ . 4 Put VQ SP
       3 Get VQ .  4 Put VQ CR ."VQ(3)= " 3 Show VQ CR
       ."VQ(4)= " 4 Show VQ CR ."VQ(4)= " 4 VQ Show AS DEQ CR ;
CR ." Для запуска демонстрации выполните QDEMO или DQDEMO " CR
UNDEF CLEAR
$$
```

5. Диалоговый интерпретатор ДССП/ТВМ

При разработке интерпретатора ДССП/ТВМ предполагалось обеспечить максимально возможную совместимость его входного языка с версией языка ДССП-Т, которая поддерживается кросс-компилятором ДССП-ТВМ. Такая совместимость означает, что интерпретатор должен уметь воспринимать и обрабатывать любую ДССП-программу, ранее созданную с помощью кросс-компилятора. И обеспечивать при этом точно такой же эффект исполнения данной программы при её запуске в среде интерпретатора, который проявлялся ранее при её прогоне в кросс-системе ДССП-ТВМ, где она могла запускаться на ТВМ автономно или вызываться в диалоговом командном мониторе.

ДССП-программа подаётся на вход интерпретатору как последовательность слов, которую он должен суметь проанализировать за один проход. Основной цикл ДССП-интерпретатора заключается в приёме очередного слова из входного потока и исполнении действия, соответствующего этому слову.

Интерпретатор поддерживает словарь известных ему слов. Каждому такому слову словаря сопоставлена процедура, которую интерпретатору надлежит вызывать, когда он встретит такое слово во входном потоке.

Помимо слов, накопленных в его словаре, интерпретатор умеет также распознавать особые слова-литеры, представляющие изображения чисел, символов, строк текста. Интерпретатор заранее знает, какое следует исполнить действие, встречая слово-литеру. Для числовой константы требуется занести в стек изображаемое ею значение, для символьной – её код, для обычной строки – поместить в стек её адрес и длину, а для строки, изображающей печатный текст, требуется напечатать этот текст.

Все слова, вызывающие исполнение тех или иных функций интерпретатора, можно условно разделить на несколько групп.

5.1 Средства построения тела программы

Основная задача, которую должен уметь выполнять интерпретатор языка ДССП, заключается в том, чтобы переводить анализируемый текст задаваемой во входном потоке ДССП-программы во внутреннее представление (в виде так называемого сшитого кода) с тем, чтобы запускать потом исполнение этого сформированного кода на троичной машине (ТВМ) при поддержке процедур ассемблерного ядра ДССП.

Предполагается, что программа, подаваемая во входном потоке интерпретатору, построена с использованием того же ассортимента компилирующих слов, которые предусмотрены в языке ДССП-Т и которые уже умеет понимать кросс-компилятор системы ДССП-ТВМ. Такие слова-компиляторы предназначены для объявления (определения) новых сущностей в программе. Каждое из них даёт указание интерпретатору создать новое слово в словаре и сформировать соответствующее ему тело во внутреннем представлении:

- слово **:** определяет новую процедуру в ДССП-программе;
- слово **VAR** определяет новую переменную;
- слово **VCTR** определяет одномерный массив;
- слово **ARR** определяет многомерный массив;
- слово **VALUE** определяет числовую константу;
- слово **TEXT** определяет константу-строку;
- слово **SITUATION** определяет исключительную ситуацию.

Обрабатывая эти слова, кросс-компилятор формировал тело компонуемой программы в виде файла ассемблерного кода. А интерпретатор, обрабатывая такие слова, должен формировать тело принимаемой программы во внутреннем представлении в особой области памяти, называемой областью тел.

Комплект процедур, обеспечивающих реализацию действий слов-компиляторов, как раз и позволяет интерпретатору осуществлять свою основную задачу. Этот комплект и составляет, по сути, основное ядро, образно говоря, сердцевину интерпретатора.

Встречая во входном потоке компилирующее слово из перечисленного выше набора,

интерпретатор должен выполнить ряд характерных действий: 1) создать в словаре заголовок для вновь определяемого слова; 2) сформировать для него тело во внутреннем представлении и сохранить адрес этого тела в заголовке слова; 3) распределить память для объекта (переменной или массива) в особой области данных.

5.2 Слова управления входным потоком

В качестве основного способа взаимодействия с пользователем ДССП-интерпретатор предлагает диалоговый режим работы. Это значит, что входной поток слов, предназначенных для исполнения, интерпретатору можно задавать вводом строки с терминала, а получать от него ответы – в виде сообщений на терминал. Первоначально после своего запуска и всякий раз, когда интерпретатор входит в диалоговый режим, он выдаёт приглашение (обычно в виде символа * с новой строки), означающее, что он ожидает от пользователя ввода очередных слов-команд с терминала.

Обычно с терминала удобно задавать короткие последовательности команд, чтобы тут же в ходе диалога наблюдать эффект их исполнения в среде интерпретатора. Но вводить с терминала весь тот огромный поток слов, с помощью которого строится достаточно большая ДССП-программа, не представляется разумным. Такую программу лучше предварительно подготовить в виде отдельного текстового файла, а потом подавать целиком на обработку интерпретатору.

Чтобы воспринимать входной поток с файла, в интерпретаторе предусмотрена операция:

```
LOAD ИмяФайла {Пример:} LOAD MyProg
```

Эта операция запоминает параметры, характеризующие текущее состояние входного потока, и переключает интерпретатор на чтение входного потока слов из заданного файла. Предполагается, что, завершив чтение слов из этого файла, интерпретатор вернётся к обработке прежнего входного потока.

Интерпретатор воспринимает поток слов из файла, пока не встретит в нём специальное завершающее слово (**$$**) или не достигнет конца файла. Если при обработке очередного файла снова встретится слово-операция **LOAD**, то интерпретатор переключится на чтение нового потока из другого файла, завершив который, вернётся к продолжению обработки слов предыдущего файла. Т.е. интерпретатор допускает возможность вложенных вызовов для операции **LOAD**.

С помощью специальных слов: **{} [] ()** можно указать интерпретатору, какой вид комментариев учитывать далее при обработке входного потока. Текущий установленный вид комментариев запоминается интерпретатором при переключении входного потока на другой файл и восстанавливается при возвращении на обработку прежнего потока.

5.3 Слова управления состоянием словаря

Словарь ДССП-программы является одной из важнейших её компонент. Почти все компилирующие операции добавляют в него новые слова. Но в языке ДССП-Т предусмотрены также и особые слова-операции для управления словарём:

- операция **::** помечает следующее определяемое слово так, чтобы оно всегда оставалось в словаре;
- операция **CLEAR** очищает словарь, оставляя в нём лишь заголовки тех слов, которые специально были помечены (операцией **::**) как неудаляемые; заметим, что удаляя из словаря заголовки остальных слов, интерпретатор оставляет всё же их тела;
- операция **UNDEF** позволяет узнать, остались ли на данный момент в словаре неопределённые слова (т.е. слова, которые использовались в описании других слов, но сами описаны не были).

Диалоговый интерпретатор дополняет язык ДССП-Т новыми операциями, которые позволяют запоминать состояние словаря, достигнутое в ходе диалога, так, чтобы впоследствии можно было бы восстановить нужное состояние словаря (запомненное ранее) перед тем, как продолжать дальнейшую работу. Необходимость в таком восстановлении требуемого состояния словаря возникает, например, когда требуется повторно загрузить из файла (командой LOAD) ту программу, которая уже когда-то ранее загружалась в ходе текущего диалогового сеанса работы

с интерпретатором. (Предполагается, что эта программа содержала ошибки, которые удалось обнаружить и исправить).

В языке ФОРТ и в версии ДССП-НЦ, которая имела такое же простое линейное строение словаря, для восстановления состояния словаря использовалась операция **FORGET** с указанием имени обычного слова, которая удаляла из словаря это слово вместе со всеми словами, созданными после него.

Интерпретатор ДССП/ТВМ обеспечивает аналогичную возможность, но разрешает применять операцию **FORGET** не с обычным, а со специальным словом (назовём его маркером), которое создаётся в словаре (операцией **GROW**) именно с целью фиксации в его теле параметров, характеризующих текущее состояние словаря. Восстанавливая значения этих параметров из тела указанного слова, операция **FORGET** сумеет вернуть словарь в то состояние, которое было зафиксировано на момент создания этого слова.

Чтобы повторно загружаемая (командой **LOAD**) ДССП-программа могла корректно обрабатываться интерпретатором, обычно в её начало помещают последовательность команд **FORGET** и **GROW** с одним и тем же словом: **FORGET** `$PROG` **GROW** `$PROG` **.**

Для краткой записи таких действий предлагается команда: **PROGRAM** `$PROG` . Это позволяет интерпретатору корректно обрабатывать любые программы, разработанные ранее в системе ДССП-ТВМ.

5.4 Операция сохранения тела и словаря

Диалоговый интерпретатор обладает ещё одной важнейшей операцией, которая совсем не предусматривалась во входном языке кросс-компилятора. Но такая операция является характерной особенностью диалоговой среды разработки интерпретируемых программ. Она имеется во всех интерпретируемых версиях языков ФОРТ и ДССП и обычно называется **SAVE** :

SAVE `ИмяФайла {Пример:}` **SAVE** `mydssp`

Эта операция позволяет сохранить результаты диалогового сеанса общения с интерпретатором так, чтобы начать с ним следующий диалоговый сеанс в том состоянии, в котором был завершён предыдущий. Для этого в указанном файле сохраняются все компоненты, характеризующие состояние диалоговой среды ДССП-интерпретатора: программный код самого интерпретатора, область его системных переменных, словарь и область сформированных тел ДССП-программы.

Существенно, что такой файл обычно формируется в формате, пригодном для последующей его загрузки в память с целью непосредственного исполнения на той же машине или в той же операционной среде, в которой и запускался ранее диалоговый интерпретатор. Поэтому можно сказать, что операцией **SAVE** на самом деле создаётся файл новой версии интерпретатора, в словарь которой каждый пользователь может включить свои новые слова, созданные им в результате проведённого сеанса работы.

Интерпретатор ДССП/ТВМ, выполняя операцию сохранения, создаёт файл в формате троичного кода (nth-файл), который впоследствии может быть загружен для исполнения троичной машиной (ТВМ).

5.5 Операции для создания новых слов-компиляторов

При разработке процедур исполнения компилирующих слов в интерпретаторе были предусмотрены вспомогательные слова-операции, которые могут оказаться полезными для разработчиков ДССП-программ, если им потребуется создавать свои собственные слова-компиляторы.

Так, например, для работы с формируемым телом уже предусмотрен следующий набор вспомогательных операций:

- операция **,** заносит в формируемое тело троичное слово, взятое из стека;
- операция **,T** заносит в формируемое тело трайт, взятый из вершины стека;
- операция **,Str** копирует в формируемое тело строку из трайтов, адрес и длина которой задаются в стеке;

• операция **,WBlk** копирует в формируемое тело блок троичных слов, адрес и длина (количество слов) которого задаются в стеке;

• операция **BA** засылает в стек адрес текущей позиции формируемого тела.

Другой набор вспомогательных операций позволяет прочитать слово из входного потока и выполнить с ним необходимые действия в словаре:

• операция **GETWORD** считывает следующее слово из входного потока;

• операция **_WORD** выдаёт в стек адрес и длину прочитанного слова;

• операция **FINDWORD** производит поиск в словаре введённого ранее слова;

• операция **WORD?** считывает слово и осуществляет его поиск по словарю.

• операция **NewWord** добавляет в словарь заголовок нового слова с заданной строкой его имени и заданной для него командой (адресом тела) в стеке;

• операция **HEAD** считывает слово из входного потока и подготавливает его заголовок в словаре для формирования тела этого слова;

• операция **WORDCODE** считывает слово из входного потока, находит его в словаре и помещает в стек сопоставленную ему команду (адрес тела).

В качестве вспомогательных в словаре интерпретатора предусмотрены даже такие операции, которые осуществляют компиляцию тела как для отдельного слова (**COMPILEWORD**), так и для последовательности слов (**COMPILEBODY**), заканчивающейся **;** . А также ряд операций, помогающих выделить память (**MUSE**) и назначить адрес (**LOCATE**) для объявляемого объекта данных.

5.6 Слова для определения новых типов

Интерпретатор поддерживает также ООП-средства, позволяющие определять в языке ДССП-Т новые типы данных как структуры или классы.

Встречая компилирующее слово **STRUCT:**, он создаёт слово-имя для нового типа данных и переключается в особый режим обработки слов, объявляющих данные, пока не встретит слово **;STRUCT**, завершающее определение нового типа. В этом особом режиме он создаёт для объявляемых объектов данных такие тела, которые позволяют им функционировать не в качестве самостоятельных переменных, а выступать в роли полей объявляемой структуры. Завершая обработку структуры, интерпретатор формирует в области тел дескриптор для нового типа данных.

Создание нового типа данных как класса интерпретатор осуществляет, обрабатывая целый комплект компилирующих слов:

CLASS: SUBCLASS ;CLASS METHOD METHOD# METHOD= :M: :M=

Но в этом случае помимо слова-имени для нового типа и слов-имён полей он создаёт ещё и слова с именами новых методов, объявляемых в определении класса, а также тела процедур, назначаемых в качестве их исполнителей.

Формируя дескриптор для нового типа-класса, интерпретатор сначала заимствует дескриптор наследуемого типа-класса, а затем расширяет его, добавляя позиции для новых методов. Впоследствии интерпретатор заполняет эти позиции ссылками на тела процедур-исполнителей методов, когда обрабатывает слова-компиляторы **:M: :M=** .

5.7 Усовершенствованные возможности командного монитора

Созданный интерпретатор не только поддерживает все функции по прогону построенной ДССП-программы, который умел ранее выполнять диалоговый командный монитор ДКМОН, но и обеспечивает для этого ряд новых возможностей, которыми ДКМОН не обладал.

При разработке интерпретатора были преодолены отмеченные ранее недостатки монитора. Так что интерпретатор умеет корректно распознавать и комментарии, и любые слова-литералы языка ДССП-Т, в том числе: большие числа, строки, печатный текст.

Существенно, что интерпретатор позволяет исполнять в режиме монитора так называемые Р-слова, т.е. особые слова, помеченные Р-флагом, которые требуют для своего исполнения доступа к следующему слову по телу сформированного сшитого кода. Для

исполнения таких Р-слов в режиме монитора интерпретатор создаёт на некоторое время тело невидимой процедуры (без имени), а затем запускает её исполнение.

Большинство Р-слов требуют доступа лишь к одному последующему слову. Поэтому, встретив в командной строке слово, помеченное Р-флагом, интерпретатор помещает в тело такой временно создаваемой процедуры результат компиляции самого этого слова вместе со словом, следующим за ним, а также команду возврата, после чего вызывает исполнение этого тела как процедуры.

Однако, такой алгоритм обработки Р-слов не позволяет интерпретатору, вообще говоря, корректно исполнять (в режиме монитора) слова, требующие доступа более чем к одному слову по телу сшитого кода. Например, таким способом было бы невозможно исполнять последовательности слов, содержащие управляющие команды ветвления и цикла:

```
BR+ BRS BR DW
```

Чтобы обеспечить исполнение всевозможных Р-слов в режиме монитора, интерпретатор предоставляет дополнительно слово-операцию с именем `;:` , которая создаёт на время процедуру без имени из последовательности слов, заканчивающейся, как и обычная процедура, словом `;` , а затем исполняет её. Тем самым открывается возможность задавать в командной строке интерпретатора любые последовательности слов, в том числе содержащие команды ветвления и цикла.

Успешной разработкой интерпретатора ДССП/ТВМ была подтверждена пригодность ДССП-ТВМ для построения полноценного программного оснащения троичной машины, отвечающего задачам автоматизации программирования.

6. Сопрограммный механизм в ДССП-ТВМ

Для создания первичного системного ПО троичной машины потребуется также решать и другую группу задач системного программирования, связанную с управлением одновременной работой нескольких внешних устройств. Совокупность программ, отвечающих за решение этих задач, традиционно считается частью операционной системы. Для разработки программ такого характера, как правило, требуются особые средства программирования, с помощью которых можно легко выразить алгоритмическую сущность параллельной работы нескольких устройств цифровой машины.

В качестве такого особого средства может успешно использоваться сопрограммный механизм [24,п.3.4]. Это было подтверждено ещё Виртом при создании ряда системных программ на языке Модула-2 [25], а также опытом построения на основе такого механизма разнообразных мониторов управления параллельными процессами в ДССП различных версий [16,26]. Поэтому в качестве развития возможностей ДССП-ТВМ (и её языка ДССП-Т) в направлении поддержки разработки операционных систем было предложено [27] дополнить её операциями сопрограммного механизма.

6.1 Операции сопрограммного механизма в ДССП-ТВМ

Средства, обеспечивающие функционирование механизма сопрограмм в ДССП-ТВМ, сосредоточены в отдельном модуле ДССП-библиотеки (файле context.dsp), который может быть подгружен командой: `LOAD` **`context .`**

В этом модуле объявлен новый тип данных **`CONTEXT`** как структура с полями, предусмотренными для сохранения значений регистров, характеризующих сопрограммный контекст. А также определены слова-операции над объектами этого типа.

Для представления каждой сопрограммы сначала требуется объявить в программе объект типа **`CONTEXT`** , например:

```
CONTEXT VAR CA  CONTEXT VAR CB  CONTEXT VAR CMAIN
```

Объявленные так слова-имена объектов (CA,CB,CMAIN) будут теперь засылать в стек адреса самих этих объектов, размещённых в памяти.

Первоначально требуется проинициализировать контекст основной программы как сопрограммы, для чего используется команда:

```
{ CMAIN } MAINCONTEXT { }
```

Необходимо также привести в начальное состояние контекст каждой новой сопрограммы, для чего предусмотрена операция:

```
{ PRA, CA, Addr, DSLen, CSLen } NEWCONTEXT { }
```

где `P  RA` – адрес процедуры, определяющей тело сопрограммы;
`CA` – адрес объекта, представляющего контекст этой сопрограммы;
`Addr` – адрес рабочего пространства, необходимого сопрограмме для своего функционирования (в нём выделяется место под стек операндов и стек возвратов);
`DSLen,CSLen` – длины областей под стеки (операндов и возвратов).

После этого можно осуществлять передачи управления от одной сопрограммы к другой. Для этого предусмотрена основная операция переключения контекста сопрограммы:

```
{ CA,CB } TRANSFER { CZ }
```

где `CA` – указатель контекста текущей сопрограммы;
`CB` – указатель контекста возобновляемой сопрограммы;
`CZ` – указатель контекста приостановленной сопрограммы.

Эта операция сохраняет контекст текущей сопрограммы `CA` и возобновляет сопрограмму `CB`, передавая в стеке указатель контекста прерванной сопрограммы `CZ`, от которой перешло управление к данной. Заметим, что `CZ` может не совпадать с `CB`, например, в случае передачи

управления по такой цепочке: CA -> CB -> CZ -> CA.

Предусмотрены также операции возобновления сопрограммы, в которых не требуется указатель контекста текущей сопрограммы:

```
{ CB } RESUME { }   { CB } RESUME= { CZ }
```

При использовании этих операций требуемый указатель автоматически сохраняется в специально предусмотренной переменной Cntxt, но инициализация сопрограммного механизма должна осуществляться уже другой командой:

```
{ CMAIN } INITCONTEXTS { }
```

Заметим, что реализацию этих дополнительных операций можно обеспечить с помощью описанных выше основных операций:

```
TWORD VAR Cntxt {контекст текущей сопрограммы}
:: : INITCONTEXTS {CMain} C MAINCONTEXT ! Cntxt {};
:: : RESUME= {CB} Cntxt E2 C ! Cntxt {Cntxt:=CB}
                {OldCntxt, CB } TRANSFER { CZ } ;
:: : RESUME {CB} RESUME= {CZ} D { } ;
```

ДССП может служить примером среды исполнения программ, характеризующейся двухстековой архитектурой. Поэтому реализация сопрограммного механизма в ней может быть осуществлена достаточно эффективно.

6.2 Специфика реализации операции контекстного переключения в ДССП-ТВМ

Операция передачи управления сопрограмме (TRANSFER) требует сохранения и последующего восстановления контекста выполняющейся ДССП-программы. Такой контекст в текущей версии ДССП-ТВМ полностью определяется содержимым двух стеков: арифметического и управляющего.

В ТВМ стеки исполняемой программы отображены в память так, что регистры-указатели DSP,DSPL,DSPH определяют расположение вершины, дна и “потолка” арифметического стека, а регистры CSP,CSPL,CSPH – управляющего стека соответственно. Поэтому для реализации операции переключения контекста в ДССП-ТВМ достаточно сохранять имеющиеся значения только этих регистров-указателей стеков (в полях объекта-структуры CA) и восстанавливать их новые значения (из полей объекта CB).

Именно эти действия и выполняет операция TRANSFER, реализованная в специальном модуле (файле context.dsp) на языке ДССП-Т как слово, определённое в ассемблерном коде (т.е. между компилирующими словами: **:ASM** и **;ASM**).

Замечание. Хотя при функционировании ДССП-программы используются ещё и четыре общих регистра R0-R3 ТВМ, при реализации операции переключения контекста их можно не сохранять, ибо они используются лишь как временные на этапе исполнения ДССП-команд, реализованных в машинном коде. Конец замечания.

При реализации операции TRANSFER обнаружился досадный недостаток системы команд ТВМ. Оказалось, что нельзя по отдельности изменять значения регистров-указателей одного и того же стека, поскольку в ТВМ встроен контроль их правильности, подразумевающий соблюдения условия: DSPL≤DSP≤DSPH для арифметического стека и условия: CSPH≤DSP≤DSPL для управляющего стека.

Поэтому было предложено добавить в ТВМ новые команды для совместного группового сохранения в памяти (**SVDS**, **SVCS**) и загрузки из памяти (**LDDS**, **LDCS**) значений триады регистров-указателей, определяющих расположение в памяти арифметического и управляющего стека соответственно.

Наконец, отметим, что такую операцию переключения контекста целесообразно (при дальнейшем развитии ТВМ) реализовать всё же одной машинной командой. Это позволило бы не только ускорить переключения между сопрограммами, но и обеспечить неделимость исполнения этой операции на случай возникновения возможных прерываний.

6.3 Примеры ДССП-программ на основе сопрограммного механизма

Реализованный в ДССП-ТВМ сопрограммный механизм был опробован при создании ряда полезных программ, разработку которых удалось существенно облегчить благодаря применению этого механизма. Кратко охарактеризуем некоторые из них.

1. На основе сопрограммного механизма в ДССП-ТВМ была реализована программа обработки текста, зашифрованного по так называемому методу транслитерации. Согласно этому методу предполагается, что поступающие из файла символы сначала записываются в массив по строкам, а выбираются из него на дальнейшую обработку по столбцам (или “змейкой” по столбцам). Для основной ДССП-программы, занятой непосредственно анализом и обработкой поступающих символов, требовался своеобразный “помощник”, который взял бы на себя всю работу по расшифровке принимаемого из файла текста. Такой “помощник” и был реализован в виде отдельной сопрограммы.

2. Игровая программа угадывания девятеричного числа (своеобразный троичный аналог известной игры “Быки и коровы”) построена в ДССП-ТВМ как совокупность четырёх сопрограмм, “роли” между которыми распределены следующим образом (см. рис.2).

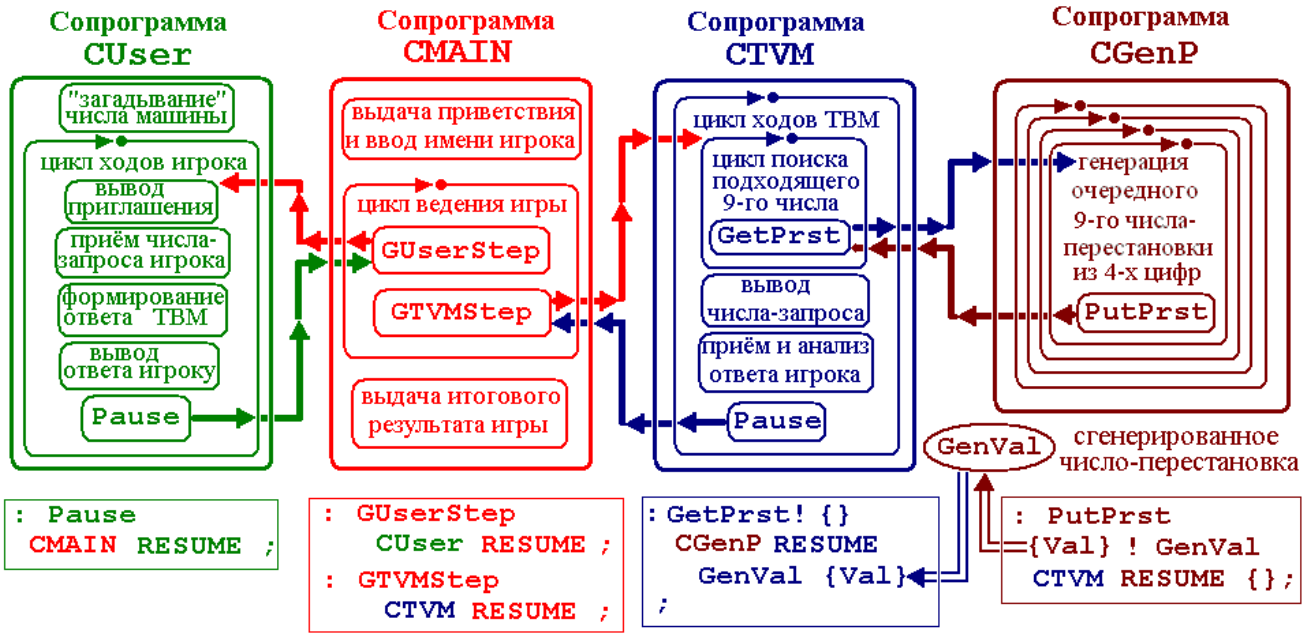

рис 2. Сопрограммная структура игровой программы

Основная сопрограмма CMAIN организует диалог с пользователем и взаимодействие с сопрограммами, выполняющими ходы игрока и машины (ТВМ). Сопрограмма CUser отвечает на ходы-запросы игрока. Сопрограмма CTVM осуществляет основную игровую деятельность машины, т.е. ведёт поиск загаданного числа, формирует ходы-запросы машины, получает ответы игрока на них, корректируя по ним дальнейший поиск решения. Для получения очередного числа-претендента она обращается к вспомогательной сопрограмме CGenP, которой поручено перебирать все числа, являющиеся правильными перестановками 9-чных цифр (012345678) длиной 4. Для генерации всех таких чисел-перестановок сопрограмма CGenP организует 4 вложенных цикла (для перебора по каждой требуемой цифре).

6.4 Простой монитор управления процессами

На основе сопрограммного механизма в ДССП-ТВМ создан модуль-монитор (файл spmon.dsp), позволяющий организовать исполнение параллельных процессов на одном процессоре по круговой очереди. Главная особенность этого монитора заключается в том, что он предоставляет запущенным процессам возможность самим регулировать переключения процессора между ними, не требуя для осуществления таких переключений регулярных прерываний от таймера. Для этого каждый процесс обязан периодически выполнять специальную операцию (см. далее **PAUSE**), чтобы уступить процессор следующему процессу по

круговой очереди (см. рис. 3).

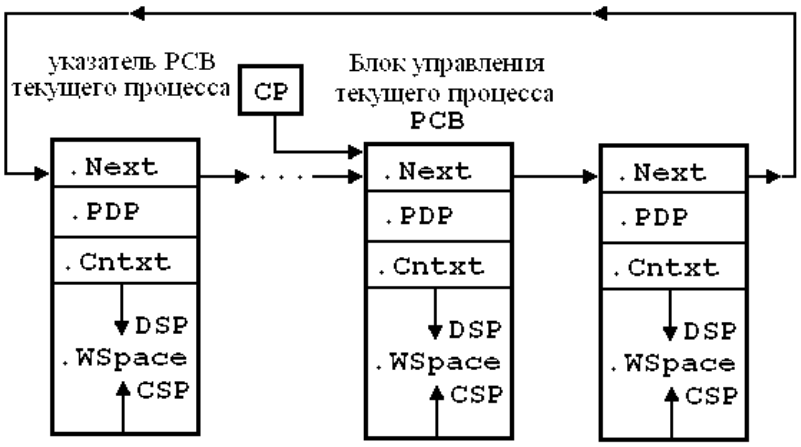

рис 3. Круговая очередь процессов

Процессы в ДССП-программе должны быть объявлены как объекты класса **PROCESS** :

```
PROCESS VAR PR
```

Основные операции монитора над процессами реализованы в нём как методы класса **PROCESS**:

1) создать процесс: **CREATE** PR PB { PB – процедура тела процесса } ;
2) запустить процесс: **START** PR ;
3) остановить процесс: **FINISH** PR .

Над текущим процессом можно также выполнять операции:

4) остановить процесс: **STOP** ;
5) задержать процесс на заданное время: {Dt} **DELAY** ;
6) задержать процесс до заданного времени: {Wt} **AFTER** ;
7) приостановить процесс с возобновлением другого: **PAUSE** .

Перед запуском процессов необходимо проинициализировать работу монитора командой **INITPROCESS**. Далее переключения между запущенными процессами осуществляются путём исполнения ими операций, вызывающих сопрограммную передачу управления:

```
CLASS: PROCESS
   VAR .Next VAR .PDP CONTEXT VAR .Cntxt
;CLASS
: PAUSE NextReady RunReady ;
: NextReady CP .Next {PCB};
: RunReady {PCB} C ! CP {PCB} .Cntxt RESUME { } ;
```

7. Специфика реализации кросс-системы ДССП-ТВМ

ДССП-ТВМ существенно отличается от всех предыдущих версий ДССП ещё и особенностями своей реализации.

Ранее ДССП всегда функционировала как резидентная система программирования, в которой входная программа (сразу вся или по частям) сначала переводилась во внутреннее представление, а исполнение такой закодированной ДССП-программы обеспечивалось специальным ПО, называемым внутренним интерпретатором. В ДССП-ТВМ роль внутреннего интерпретатора фактически возложена на саму ТВМ, а внутреннее представление всей программы подготавливается кросс-компилятором, т.е. отдельной компонентой системы, функционирующей не на самой ТВМ, а на другой машине.

Для внутреннего представления программ в ДССП всегда применялся так называемый сшитый (threaded) код. Хотя в разных версиях ДССП использовались та или иная разновидность сшитого кода (прямой [14], знаково-косвенный [17], двойной косвенности [13,30]), всех их объединяло одно характерное свойство: в сшитом коде в закодированном виде представлялись адреса процедур, которые интерпретатору предстояло вызывать при исполнении программы.

При разработке ДССП-ТВМ предстояло решить проблему, какую же разновидность сшитого кода применять для внутреннего представления, чтобы ТВМ могла напрямую (т.е. без специального интерпретатора) вызывать процедуры по встречающимся в этом коде адресам. Для решения этой проблемы было предложено применить в ТВМ такую кодировку команд, чтобы значение адреса воспринималось ею и как команда вызова процедуры с этого адреса. В результате для внутреннего представления программ в ДССП-ТВМ стал использоваться так называемый процедурный сшитый код (STC согласно классификации, приведённой в [28]), который может непосредственно исполняться троичной машиной.

Кросс-компилятор формирует такой код на языке ассемблера ТВМ. В нём для каждого слова-примитива ДССП-ядра размещается одна машинная команда, которая реализует действие этого примитива (см. таблицу 1). Для тех примитивов, действие которых невозможно исполнить одной машинной командой, заготовлены ассемблерные процедуры, их реализующие, а в сшитый код вставляется адрес (или команда вызова) соответствующей ей процедуры. Совокупность таких ассемблерных процедур образует ассемблерное ядро, которое добавляется к каждой откомпилированной ДССП-программе.

Продемонстрируем реализацию некоторых процедур ассемблерного ядра:

<	**E2D**	**ABS**	**DEEP**

`{[x,y] ==> [x<y?]}`
`LT: CMP;`
`Copy_1`
`BRS SetI_1`
`SetI_0`
`SetI_0`
`RET ;`
	`{[X,Y] ==> [Y]}`
	`E2D:`
	`Exch_2`
	`Drop_1`
	`RET ;`
		`{[X] ==> [|X|]}`
		`ABS:`
		`Copy_1`
		`IFM NEG`
		`RET ;`
			`{[ ] ==> [N]}`
			`DEEP:`
			`DPush_DSP`
			`DPush_DSPL`
			`Sub;`
			`ShI_-1`
			`RET ;`

Объясним теперь, как формируются компилятором тела новых слов, определяемых с помощью компилирующего слова **:** и представляющих собой процедуры в ДССП-программе. Каждой новой процедуре назначается адрес, начиная с которого располагается её тело и который вставляется в тела других процедур в том месте, где она из них вызывается.

Допустим, в программе уже определены слова: `PX PY PZ` , а также известны, но ещё не определены слова: PU PV и назначены (компилятором) такие ассемблерные метки для этих слов:

имена слов:	**PX**	**PY**	**PZ**	**PU**	**PV**
ассемблерные метки:	PN2001	PN2002	PN2003	PN2004	PN2005

Пусть в программе определяется (с помощью **:**) новое слово P как процедура, содержащая в своём теле вызовы процедур с именами этих слов:

```
: P PX PY PU PV ... PZ ;
```

Таблица 1. Таблица соответствия слов ДССП-ядра и ассемблерных команд.

имя слова	ассемблерная команда
T0	SETi_0
T1	SETi_1
T2	SETi_2
T3	SETi_3
T4	SETi_4
T-1	SETi_-1
T-2	SETi_-2
T-3	SETi_-3
T-4	SETi_-4
C	COPY_1
C2	COPY_2
C3	COPY_3
C4	COPY_4
CT	COPYT
E2	EXCH_2
E3	EXCH_3
E4	EXCH_4
ET	EXCHT
D	DROP_1
DD	DROP_2
DDD	DROP_3
DDDD	DROP_4
E2D	call_E2D
DS	call_DS
DEEP	call_DEEP
+	ADD
1+	ADDi_1
2+	ADDi_2
3+	ADDi_3
4+	ADDi_4
-	SUB
1-	SUBi_1
2-	SUBi_2
3-	SUBi_3
4-	SUBi_4
NEG	NEG
ABS	call_ABS
SGN	SIGN
MAX	MAX
MIN	MIN
*****	MUL
***2**	MULi_2
***3**	SHi_1
/	DIV
/2	call_DIV2
/3	SHi_-1

имя слова	ассемблерная команда
SHL	SHi_1
SHR	SHi_-1
SHT	SHT
TADD	TADD
TMUL	TMUL
TMIN	TMIN
TMAX	TMAX
CMP	CMP
<	call_LT
<=	call_LE
=	call_EQ
<>	call_NE
>=	call_GE
>	call_GT
SEG	call_SEG
NOT	call_NOT
AND	TMUL
OR	TMAX
@T	Fetch1
@TT	Fetch2
@W	Fetch3
!T	Store1
!TT	Store2
!W	Store3
!	call_VARStore
'	call_VARAdr
.	TOP
.3	TOP3
.9	TOP9
.10	TOP10
.16	TOP16
LOB	PRINTCHAR
LOS	PRINTSTRING
TIB	GETCHAR
TOB	PUTCHAR
FOPENR_	call_FOPENRp
FOPENW_	call_FOPENWp
FREADB_	call_FREADBp
FREADW_	call_FREADWp
FREADL_	call_FREADLp
FWRITEB_	call_FWRITEBp
FWRITEW_	call_FWRITEWp
FWRITEL_	call_FWRITELp
FCLOSE_	call_FCLOSEp

имя слова	ассемблерная команда
NOP	NOP
;	RET
EXEC	EXEC
HALT	HALT
''	call_LIT
GTP	call_GetCmndRA
@R	CPopDPush
!R	DPopCPush
IF0	IF0
IF-	IFM
IF+	IFP
BR0	BR0
BR-	BRM
BR+	BRP
BRS	BRS
BR	call_BR
ELSE	call_ELSE
DW	DW
DO-	call_DOdec
LOOP	call_LOOP
DO	call_DO
RP	call_RP
EX	call_EX
EX0	call_EX0
EX-	call_EXMi
EX+	call_EXPl
ON	call_NotifyON
EON	call_EscapeON
RON	call_RetryON
ON_	call_ONp
_ESCAPE	call_pEscape
_RETRY	call_pRetry
_NOTIFY	call_pNotify
_RAISE	call_pRAISE
RAISE_	call_RAISEp
RERAISE	call_RERAISE
ANY	call_ANY
SIT?	call_RaisedSIT
IndexOut!	call_IndexOut

Тогда компилятор добавит новое слово P в словарь (назначив ему следующую уникальную метку PN2006) и сформирует для него (на ассемблере) такое тело в сшитом коде:

: P	PN2006: {{ "---- : P "
PX	call_PN2001 {{" PX "
PY	call_PN2002 {{" PY "
PU	call_PN2004 {{" PU "
PV	call_PN2005 {{" PV "
...	...
PZ	call_PN2003 {{" PZ "
;	RET {{" ; "

Для слов, которые использовались в программе, но не были определены, компилятор поместит назначенные им метки в итоговый ассемблерный код перед телом особой специально приготовленной процедуры:

```
PN2004: {{ метки неопределённых слов: PU PV
PN2005: {{ тело особой спец-процедуры, которая:
  DPushI_99999999 TOP {{ печатает значение 99999999
  Drop_1 HALT RET; {{ и останавливает программу
```

В настоящее время такая процедура (будучи вызванной) обеспечивает останов программы с предварительной печатью числа 99999999 в качестве предупреждающего сообщения. В дальнейшем предполагается, что эта процедура будет вызывать отладчик диалогового интерпретатора ДССП.

Более сложное строение имеют тела слов, представляющих другие сущности ДССП-программы: переменные, массивы, исключительные ситуации, числовые и текстовые константы-строки. Каждому из этих слов тоже сопоставляется уникальный адрес его тела. Но по этому адресу предполагается не только вызывать процедуру, с которой начинается тело слова, но и получать доступ к тем или иным дополнительным характеристикам слова, сформированным в его теле (например, адресу памяти переменной, границам массива и т.д.).

Другой ряд проблем возник при реализации кросс-компилятором языка ДССП-Т средств объектно-ориентированного программирования.

1. Проблема №1 Компилятор функционирует в цикле обработки потока известных ему слов-компиляторов, для каждого из которых у него заранее заготовлена своя Си-процедура. Как же “научить ” компилятор понимать новые компилирующие слова, представляющие имена вновь определённых типов (например, QUEUE, DEQ) ?

2. Проблема №2 Компилятор последовательно (за один проход!) генерирует ассемблерный код, отправляя его в выходной файл. Как обеспечить копирование дескриптора наследуемого класса?

3. Проблема №3 Дескриптор требуемого типа как и номер метода теперь невозможно напрямую взять из тела (как ранее это делалось в интерпретаторе ДССП), т.к. сформированные тела в ассемблерном коде помещаются в выходной результирующий файл. Как же узнать номер метода при определении процедуры его исполнения?

Для разрешения этих проблем в кросс-компиляторе пришлось сделать весьма существенные модификации.

1. Для слов, представляющих имена новых типов, был предусмотрен дополнительный словарь, в каждом узле которого содержатся:

1) строка **IName** слова имени типа;

2) индекс **ICode** для доступа в таблицу к телу его дескриптора;

3) строка **ILabel** метки для ассемблерного тела дескриптора.

При этом был изменён алгоритм обработки компилирующих слов: теперь, если встреченное компилирующее слово не найдено в основном словаре, то компилятор пытается найти это слово в дополнительном словаре типов. Если найдено слово типа, то его номер фиксируется в спецпеременной **nTYPE** так, чтобы использовать этот номер типа для последующих объявлений данных (переменных и массивов).

Когда в программе объявляется новый тип данных (см. например, объявление класса

QUEUE) компилятор создаёт новое слово (QUEUE) и в своём словаре типов, и в ДССП-словаре компилируемой программы, формируя для него такое ассемблерное тело в выходном файле, чтобы при его вызове спецпеременная **DTYPE** получала значение адреса дескриптора этого типа:

```
call_LIT  ; tword TN1008  {{ '' QUEUE'Ptr
DPushI_42 ; Store         {{ 'DTYPE !W
RET                       {{ ;
```

2. Компилятор создаёт особую таблицу дескрипторов типов как массив (**TypeBody**) значений, которые должны представлять тела дескрипторов объявленных типов сначала в компиляторе, а затем и в ассемблерном коде формируемой программы. Индекс-ссылка (**ICode**) из каждого узла слова имени типа указывает на расположение тела его дескриптора в этом массиве.

При наследовании типа его дескриптор копируется внутри этого массива. Затем в нём изменяются поля счётчика методов и размера объекта, а в его таблицу методов заносятся номера слов-исполнителей методов.

После завершения компиляции ДССП-программы итоговое содержимое массива отправляется в файл ассемблерного кода с учётом того, что элементы массива (тип long int) имеют разный смысл (см. таблицу 2). Заметим, что тело дескриптора нельзя выдавать в ассемблерный файл сразу же после его создания, поскольку при дальнейшей компиляции ДССП-программы в его таблице методов ещё будут корректироваться ссылки на тела процедур, назначаемых в качестве исполнителей этих методов. В массиве такими ссылками служат номера слов, а в ассемблерный код помещаются соответствующие их словам мнемокоды команд или метки, которые превращаются далее (ассемблером) в адреса тел (или команды вызова) процедур-исполнителей методов.

Таблица 2. Строение тела дескриптора типа

<table>
<tr><td></td><td>в массиве:</td><td>на ассемблере:</td><td></td></tr>
<tr><td>-3</td><td>номер типа-предка</td><td>.tword TN1007</td><td>адрес дескриптора-предка</td></tr>
<tr><td>-2</td><td>счётчик методов</td><td>.tword 10</td><td>числовое значение</td></tr>
<tr><td>-1</td><td>размер объекта</td><td>.tword 315</td><td>числовое значение</td></tr>
<tr><td>0</td><td rowspan="6">Таблица методов:
содержит номера слов процедур, назначенных исполнителями методов</td><td>TN1008: NOP</td><td rowspan="6">Таблица методов:
содержит ассемблерные коды команд исполнителей методов или команды (адреса) вызовов их процедур</td></tr>
<tr><td>1</td><td>call_StructCopy</td></tr>
<tr><td>2</td><td>NOP</td></tr>
<tr><td>3</td><td>call_PN2232</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>10</td><td>call_PN2373</td></tr>
</table>

3. Номера слов, назначаемых в качестве исполнителей методов, требуется вносить в таблицу методов дескриптора типа по мере определения тел их процедур:

```
QUEUE :M: Get  ... ;
```

Для этого необходимо по имени метода узнавать его номер. В интерпретаторе для этого достаточно “заглянуть” в память на второе слово тела метода. А в кросс-компиляторе для решения этой проблемы (№3) потребовалось добавить дополнительное поле в узел слова ДССП-словаря, чтобы хранить в нём номер метода вместе с флагом-признаком метода. Это позволило получить возможность корректировать поля таблицы методов в дескрипторе типа, применяя такой алгоритм:

1) после выполнения слова с именем типа (QUEUE) спецпеременная **nTYPE** получает номер слова-типа nTYPE:= nQUEUE ;

2) по этому номеру из узла словаря типов получаем ссылку **ICode** на его дескриптор в массиве **TypeBody** ;

3) по имени метода (Get) получаем (из поля его узла ДССП-словаря) номер метода **nMtdGet**;

4) изменяем элемент таблицы методов в массиве тел дескрипторов **TypeBody[ICode+nMtdGet]**, записывая в него номер слова, назначенного в качестве исполнителя метода QUEUE::Get.

8. Обеспечение нисходящей разработки программы в интерпретаторе ДССП/ТВМ

Одной из ключевых особенностей ДССП является возможность при определении нового слова употреблять слова, которые еще не были определены в ДССП-программе. Это позволяет разрабатывать ДССП-программу методом нисходящего проектирования (“сверху-вниз”).

В предыдущих версиях ДССП, реализованных для двоичных машин, данная возможность обеспечивалась путём применения для внутреннего представления ДССП-программы сшитого кода двойной косвенности (в ДССП-НЦ [30]) или за счёт усложнения словаря [31] путём хранения в нём для каждого встреченного (но не определённого ещё) слова специальных таблиц, содержащих адреса позиций тел, в которых было использовано (вызвано) это слово.

Но такие варианты решения проблемы ссылок на неопределённые слова не могли быть применены для создаваемого интерпретатора ДССП/ТВМ. Поскольку он должен уметь работать с тем же строением словаря и с тем внутренним представлением ДССП-программ, которые поддерживаются кросс-компилятором и ассемблерным ядром системы разработки ДССП-ТВМ.

В ходе разработки интерпретатора ДССП/ТВМ удалось найти иное решение данной проблемы, не потребовавшее каких-либо модификаций ни словаря, ни внутреннего представления ДССП-программы. Рассмотрим предлагаемый способ разрешения неопределённых ссылок и объясним, почему он сделан именно таким.

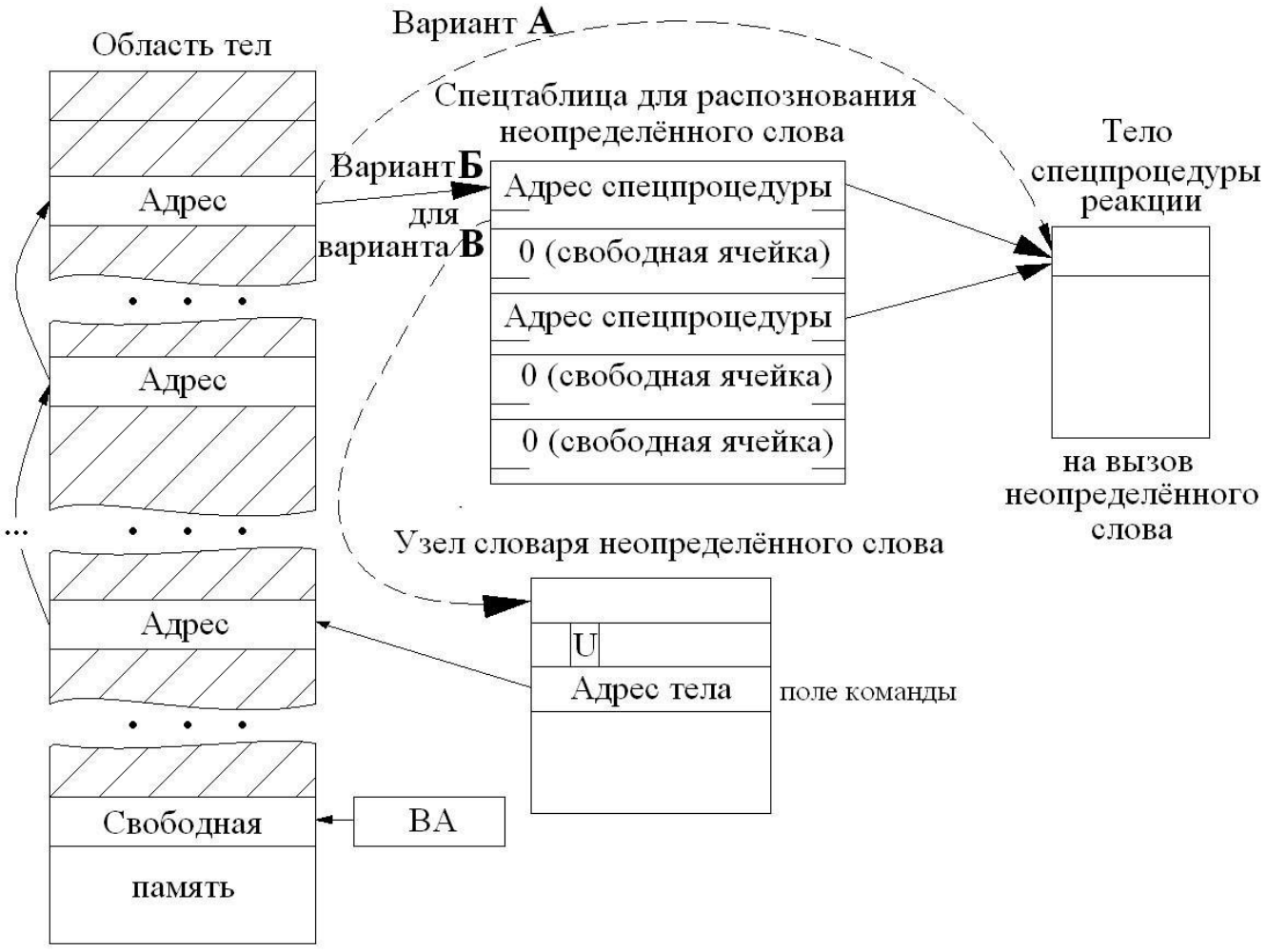

рис. 4. Схема решения проблемы неопределённых слов.

Для каждого встреченного неопределённого слова ДССП-интерпретатор должен уметь, по крайней мере, запоминать адреса всех позиций в телах, где такое слово было вызвано. А встретив определение этого слова, проставить во всех запомненных позициях ссылку на сформированное тело этого слова.

Для сохранения адресов таких позиций можно связать все эти позиции в цепной список,

записав в каждой из них ссылку на предыдущую позицию, где встречался вызов этого неопределённого слова. А начальную ссылку такого списка, указывающую на позицию последнего вызова этого слова, можно разместить в заголовке неопределённого слова (в поле команды).

Существенный недостаток такого приёма сохранения списка позиций проявляется при попытке запускать в среде интерпретатора ДССП-программу, содержащую вызовы неопределённых слов. Такие вызовы в прежних реализациях ДССП-интерпретаторов могли бы вызывать непредсказуемые последствия, поэтому разработчики ДССП вынуждены были отказываться от такого привлекательного приёма сохранения списка позиций.

Но в создаваемом интерпретаторе ДССП/ТВМ описанный приём сохранения списка позиций можно выгодно использовать, если поставить в конце созданного списка (т.е. в той позиции тела, где неопределённое слово встретилось первый раз) адрес специальной процедуры, назначенной для общей реакции на вызов неопределённого слова (см. рис. 4, вариант А).

Воспользуемся тем, что ТВМ воспринимает 26-тритный адрес как команду вызова процедуры с этого адреса. Значит, при вызове такой команды с одной из позиций, используемой в построенном цепном списке, произойдет череда вызовов процедур по адресам из этого списка и, в конечном итоге, будет вызвана назначенная спецпроцедура реакции.

Предполагается, что в такой процедуре реакции интерпретатор может предложить разработчику ДССП-программы выполнить действия, имитирующие эффект исполнения не определённой пока процедуры, и затем вернуться к продолжению прерванной программы. (В дальнейшем такая спецпроцедура реакции, возможно, станет компонентой специализированного отладчика в составе интерпретатора).

Поэтому спецпроцедура реакции должна, во-первых, уметь распознать, какое неопределённое слово было вызвано, т.е. получить адрес его словарного заголовка, и во-вторых, обеспечить продолжение прерванной программы с той точки, где было вызвано неопределённое слово.

По адресу возврата из спецпроцедуры реакции можно узнать адрес позиции, откуда она была вызвана. В этой позиции должен завершаться цепной список, построенный для вызванного неопределённого слова. Далее потребуется осуществить обход словаря и для каждого неопределённого слова (помеченного флагом U) проверить, завершается ли его цепной список именно в этой позиции. Так можно узнать, какое неопределённое слово было вызвано.

Но к сожалению, в ДССП не во всех случаях по адресу возврата можно правильно определить позицию, откуда была вызвана процедура (у нас это спецпроцедура реакции). Например, её нельзя однозначно определить, если процедура вызывалась управляющими командами: **`BR+ BRS BR DW EXEC .`**

Чтобы обеспечить возможность правильного определения позиции конца цепного списка по адресу возврата из спецпроцедуры, потребуем, чтобы он заканчивался на позиции, где поставлена команда явного вызова спецпроцедуры реакции на неопределённое слово. Такая дополнительная позиция требуется для каждого неопределённого слова на время работы с его цепным списком. Для таких позиций будем выделять память динамически, используя для этой цели свободные (нулевые) ячейки специально организованной таблицы, размещённой в памяти так, чтобы не мешать формированию тела и словаря ДССП-программы (см. рис. 4 вариант Б).

При определении тела неопределённого слова, когда во все позиции его цепного списка заносится адрес формируемого тела, сам цепной список расформировывается, а отведённая ему ячейка спецтаблицы освобождается (она помечается нулевым значением) и может быть распределена снова.

Предложенный вариант (Б) позволяет не только однозначно распознать, какое неопределённое слово было вызвано, но и вернуться из спецпроцедуры реакции к продолжению прерванной программы. Заметим, что для обеспечения правильного возврата (именно в ту точку, где было вызвано неопределённое слово), процедура реакции должна будет очистить стек возвратов от “лишних” адресов, накопившихся в нём в результате промежуточных вызовов процедур по адресам из цепного списка.

В данном варианте для распознавания неопределённого слова требуется осуществлять

обход словаря и цепных списков. Можно значительно ускорить процесс поиска неопределённого слова, если в спецтаблице рядом с позицией конца цепного списка сохранять ссылку на заголовок неопределённого слова, для которого этот список построен (см. рис. 4 с вариантом В). Но тогда потребуется усложнить процедуру передвижения словаря, т.к. ей придётся ещё и корректировать такие ссылки, сохраняемые в спецтаблице.

Если даже в качестве таких ссылок на узлы словаря использовать не абсолютные, а относительные адреса, то их всё равно пришлось бы корректировать: если не при передвижении словаря, так при его очистке, выполняемой операцией CLEAR .

Усложнение операций передвижения и очистки словаря может замедлять регулярную работу интерпретатора по построению ДССП-программы. А длительный поиск неопределённого слова может проявиться лишь в виде паузы при реакции интерпретатора на вызов неопределённого слова. Поскольку такая реакция, как правило, сама предусматривает приостановку исполнения ДССП-программы с выходом в режим диалога, то данная пауза вряд ли будет заметна. Поэтому в настоящее время для решения проблемы неопределённых слов в интерпретаторе ДССП/ТВМ используется вариант Б, изображённый на рисунке 4 как основной.

Заключение

В НИЛ троичной информатики ВМК МГУ (в период с 2010 по 2013 г.) созданы троичная виртуальная машина ТВМ и кросс-система разработки программ для неё ДССП-ТВМ. В среде ДССП-ТВМ разработан диалоговый интерпретатор ДССП/ТВМ, способный функционировать на троичной машине в качестве её резидентной системы программирования.

В настоящее время создаваемые на языке ДССП-Т программы можно прогонять на ТВМ как автономно, так и в режиме привычного для ДССП диалога, если скомпоновать и запускать их вместе с диалоговым командным монитором (ДКМОН), а также разрабатывать непосредственно в среде диалогового интерпретатора ДССП/ТВМ.

Хотя ТВМ и ДССП-ТВМ функционируют пока в среде ОС Windows, их можно относительно легко перенести и в другую операционную среду (например, Linux), т.к. кросс-компилятор, ассемблер и имитатор ТВМ написаны на языке Си (в системе lcc-win32).

Опыт создания интерпретатора ДССП/ТВМ в кросс-системе разработки ДССП-ТВМ позволяет утверждать, что эта кросс-система (как и сам созданный интерпретатор) могут служить основой для построения полноценного первичного программного оснащения аппаратно реализованной троичной машины той же архитектуры.

Реализация имитатора и ассемблера ТВМ выполнена Сидоровым Сергеем Александровичем. Реализация кросс-компилятора, ассемблерного ядра и модулей библиотеки ДССП-ТВМ выполнена Бурцевым Алексеем Анатольевичем. Реализация диалогового интерпретатора ДССП/ТВМ выполнена Бурцевым Михаилом Алексеевичем.

Более подробную информацию о ТВМ и ДССП-ТВМ можно найти в Интернете на сайте лаборатории троичной информатики ВМК МГУ по адресу: http://ternarycomp.cs.msu.ru/

Авторы выражают благодарность сотрудникам НИЛ троичной информатики ВМК МГУ Рамилю Альваресу Хосе и Владимировой Юлии Сергеевне за ценные советы и замечания, выраженные ими в ходе плодотворных дискуссий на семинарах НИЛ троичной информатики, посвящённых обсуждению архитектуры ТВМ и языка ДССП-Т.

Литература

1. Кнут Д. Искусство программирования на ЭВМ. т.2 Получисленные алгоритмы. п. 4.1. с. 216-219.

2. Брусенцов Н.П., Рамиль Альварес Х. Троичные ЭВМ “Сетунь” и “Сетунь 70”. // Первая Международная конференция "Развитие вычислительной техники в России и странах бывшего СССР: история и перспективы" SORUCOM-2006 (3-7 июля 2006 г., г. Петрозаводск, Россия): Труды конференции в 2-х ч. Петрозаводск: Изд-во ПетрГУ, 2006. ч.1, с. 45-51.

3. Будущее квантовых компьютеров — в троичных вычислениях. URL: http://www.infuture.ru/news.php?news_id=475 (дата обращения: 02.11.2014).

4. Малашевич Д.Б. Недвоичные системы в вычислительной технике. URL: http://www.computer-museum.ru/books/archiv/sokcon27.pdf (дата обращения: 02.11.2014)

5. Маслов С.П. Об одной возможности реализации троичных цифровых устройств // Программные системы и инструменты. Тематический сборник №12, М.: Изд-во факультета ВМК МГУ, 2011. С.222-227.

6. Маслов С.П. Троичная схемотехника // Программные системы и инструменты. Тематический сборник №13, М.: Изд-во факультета ВМК МГУ, 2012. С.152-158.

7. Сидоров С.А., Владимирова Ю.С. Троичная виртуальная машина // Программные системы и инструменты. Тематический сборник №12, М.: Изд-во ф-та ВМК МГУ, 2011. С.46-55.

8. Бурцев А.А., Сидоров С.А. История создания и развития ДССП: от "Сетуни-70" до троичной виртуальной машины. // Вторая Международная конференция "Развитие вычислительной техники и её программного обеспечения в России и странах бывшего СССР" SORUCOM-2011 (12-16 сентября 2011 г., г. Великий Новгород, Россия): Труды конференции. В.Новгород: Изд-во НовГУ, 2011. с. 83-88.

9. Бурцев А.А., Рамиль Альварес Х. Кросс-система разработки программ на языке ДССП для троичной виртуальной машины // Программные системы и инструменты. Тематический сборник №12, М.: Изд-во факультета ВМК МГУ, 2011. С.183-193.

10. Бурцев А.А., Бурцев М.А. ДССП для троичной виртуальной машины. // Труды НИИСИ РАН, т.2, №1. М.: Изд-во НИИСИ РАН, 2012. с.73-82.

11. Рамиль Альварес Х., Владимирова Ю.С. Программное обеспечение малой ЭВМ «Сетунь». // Третья Международная конференция "Развитие вычислительной техники и её программного обеспечения в России и странах бывшего СССР: история и перспективы" SORUCOM-2014 (13-17 октября 2014 г., г. Казань, Россия): Труды конференции. Казань: Изд-во КНИТУ-КАИ, 2014. с. 315-318.

12. Баранов С.Н., Ноздрунов Н.Р. Язык Форт и его реализации. – Л.: Машиностроение, 1988.

13. Брусенцов Н.П., Златкус Г.В, Руднев И.А. ДССП - диалоговая система структурированного программирования. // Программное оснащение микрокомпьютеров. М.: Изд-во МГУ, 1982, с.11-40.

14. Брусенцов Н.П., Захаров В.Б., Руднев И.А., Сидоров С.А. Диалоговая система структурированного программирования ДССП-80. // Диалоговые микрокомпьютерные системы. М.: Изд-во МГУ, 1986, с.3-21.

15. Бурцев А.А. Периферийный монитор – развитие архитектуры ввода/вывода ДССП. // Диалоговые микрокомпьютерные системы. М.: Изд-во МГУ, 1986, с.42-51.

16. Борисов А.В. Диалоговая система структурированного программирования в реальном времени – ДССП-РВ. // Диалоговые микрокомпьютерные системы. М.: Изд-во МГУ, 1986, с.51-62.

17. Бурцев А.А., Frантов Д.В., Шумаков М.Н. Разработка интерпретатора сшитого кода на языке Си. // Вопросы кибернетики. Сб. статей под ред. В.Б.Бетелина. М., 1999. с.64-76.

18. Бурцев А.А. ДССП – среда структурированной разработки программ как сложных систем. // Вторая Международная конференция “Системный анализ и информационные технологии” САИТ-2007 (10-14 сентября 2007 г., Обнинск, Россия): Труды конференции. Изд-во ЛКИ, 2007. т. 2. с.190-194.

19. Бурцев А.А. Механизм прерываний для интерпретатора сшитого кода. // сб.статей

“Информационная безопасность. Микропроцессоры. Отладка сложных систем.” (под ред. академика РАН В.Б.Бетелина.) М., НИИСИ, 2005, с. 177-191

20. Сидоров С.А., Шумаков М.Н. ДССП как открытая система. // Дискретные модели. Анализ, синтез и оптимизация. Спб.: СпбГУ, 1998. с.191-201.

21. Бурцев А.А., Рамиль Альварес Х. Средства объектно-ориентированного программирования в ДССП. // Программные системы и инструменты. Тематический сборник №4, М.: Изд-во факультета ВМК МГУ, 2003. с.166-175.

22. Брусенцов Н.П., Захаров В.Б., Руднев С.А., Сидоров С.А., Чанышев Н.А. Развиваемый адаптивный язык РАЯ диалоговой системы программирования ДССП. М.: Изд-во Моск. Ун-та, 1987 г. – 80 с.

23. Бурцев А.А., Рамиль Альварес Х. Реализация средств объектно-ориентированного программирования в кросс-компиляторе языка ДССП-Т. // Программные системы и инструменты. Тематический сборник №13, М.: Изд-во факультета ВМК МГУ, 2012. с.28-37.

24. Дал У., Дейкстра Э., Хоор К. Структур(ирован)ное программирование. М.: Изд-во Мир, 1975.

25. Вирт Н. Программирование на языке Модула-2. М.: Изд-во Мир, 1987.

26. Бурцев А.А., Шумаков М.Н. Сопрограммный механизм в ДССП как основа для построения мониторов параллельных процессов. // Вопросы кибернетики. Сб. статей под ред. В.Б.Бетелина. М., 1999. с.45-63.

27. Бурцев А.А., Рамиль Альварес Х. Сопрограммный механизм в системе структурированного программирования для троичной машины. // Программные системы и инструменты. Тематический сборник №14, М.: Изд-во факультета ВМК МГУ, 2013. с.193-208.

28. Ritter T., Walker G. Varieties of threaded code for language implementation. // BYTE, 1980, v.5, No.9, p.206.

29. Лякина Е.Н., Пшеничный К.А., Сидоров С.А., Шумаков М.Н. Система внутреннего программного оснащения DPROM. // Вопросы кибернетики. Сб. статей под ред. В.Б.Бетелина. М., 1999. с.77-86.

30. Захаров В.Б., Златкус Г.В., Руднев И.А., Сидоров С.А. Реализация диалоговой системы структурированного программирования на микрокомпьютере “Электроника НЦ-03Д”. // Архитектура и программное оснащение цифровых систем. М.: Изд-во МГУ, 1984, с.10-17.

31. Сидоров С.А. Программирование сверху вниз и организация словаря ДССП. // Вопросы кибернетики. Сб. статей под ред. В.Б.Бетелина. М., 1999. с.32-44.