Разработка собственных управляющих конструкций в среде ДССП для троичной машины

А.А. Бурцев

ФГУ ФНЦ НИИСИ РАН, Москва, Россия, E-mail: burtsev@niisi.msk.ru

Аннотация: Системы программирования со словарной организацией (ФОРТ и ДССП) предоставляют пользователю-программисту уникальную возможность путём наращивания словаря сотворить по сути свой собственный язык программирования. При этом в качестве новых в словарь можно добавлять и такие слова, которые предназначены для управления ходом исполнения программы. В статье подробно рассматриваются приёмы разработки в среде ДССП-ТВМ (версии ДССП для троичной машины) особых процедур для таких слов (команд), которые обеспечивают функционирование добавляемых в язык ДССП своих собственных управляющих конструкций.

Ключевые слова: структурированное программирование, словарная организация, ФОРТ, ДССП, троичный компьютер, сшитый код, управляющие конструкции.

Введение

Системы программирования со словарной организацией, такие как ФОРТ[1] и ДССП[2,3], предоставляют пользователю (программисту) уникальную возможность путём наращивания словаря сотворить по сути свой собственный язык программирования. При конструировании лексики нового языка достаточно лишь соблюдать простые синтаксические правила базового языка. В языках ФОРТ и ДССП ключевой лексической единицей является слово. За небольшим исключением словом считается любая последовательность символов, отличных от пробела (при этом знаки табуляции и перехода на новую строку считаются эквивалентными пробелу), и путём последовательного сочленения (через пробел) имеющихся слов можно строить любые предложения языка. ДССП и ФОРТ являются интерпретируемыми системами, а всякое действие в них вызывается словом.

Программирование, как процесс созидания программного изделия, в этих системах заключается в расширении словаря, т.е. в наполнении первоначально предоставленного словаря новыми словами. Причём, наряду с обычными словами, обозначающими просто названия новых операций, в создаваемый словарь можно также добавлять и особые слова, которые далее будут сами использоваться при построении программы для объявлений в ней новых сущностей. Таких, как константы, переменные, строки, массивы, структуры, классы для определения новых типов данных вместе с процедурами-исполнителями их методов, сопрограммы, исключительные ситуации и т.п. Подобные особые слова, служащие для определения новых слов и управляющие формированием компонуемой программы, в ДССП и ФОРТе принято называть компилирующими словами или словами-компиляторами.

В обычных языках программирования для управления ходом исполнения программы, как правило, предлагается жёстко установленный ассортимент управляющих конструкций (операторов), для употребления которых в языке заранее закрепляется определённый набор ключевых слов и устанавливаются синтаксические правила их применения. И программист вынужден строить свою программу, ориентируясь лишь на базовые управляющие конструкции, которые были предусмотрены при создании используемого языка и его транслятора (компилятора и/или интерпретатора).

Как правило, в современных языках программирования предусматривается условный оператор **if then else** для организации ветвлений, операторы **while do** и **repeat until** – для организации циклов. Обычно также предусматривается какой-то вариант цикла со счётчиком с перебором всех возможных значений из целочисленного отрезка [a,b]: **for** i**:=** a **to** b **do**. В некоторых языках предлагаются ещё и разновидности операторов выбора для организации множественного ветвления: **case of** или **switch**.

Но каким бы совершенным ни был ассортимент управляющих конструкций, предусмотренный в языке на момент его создания, рано или поздно он может оказаться устаревшим. И история развития языков программирования уже не раз это подтверждала.

Так, например, в конце 60-х годов XX века после публикаций работ Дейкстры [4, с.7-97] о важности соблюдения структурированного подхода при разработке программ не только в новые (Паскаль, Си), но и в уже существующие

на тот момент языки программирования (Фортран, Бейсик) стали добавлять особые структурированные конструкции для организации ветвлений и циклов в программе [5]. Чуть позже в языки программирования стали внедрять и особые управляющие конструкции для структурированной обработки так называемых исключительных ситуаций [6].

Впоследствии в тех или иных языках программирования стали предлагаться и другие полезные управляющие конструкции. Например, команда троичного ветвления по знаку [7, с.29]: **<BR** ProcN,Proc0,ProcP**>;** оператор многовариантного выбора с множественными условиями [8 с. 56-59]:

```
if
  усл1 => действие1
  усл2 => действие2 . . .
  услN => действиеN
fi
```

оператор цикла с множественными условиями и телами [8, с.59-60]:

```
do
   усл1 => тело1
   усл2 => тело2  . . .
   услN => телоN
od
```

конструкция цикла для перебора всех значений множества: **for** x **in S do** [4, с.144]; а также так называемые итераторы – итерационные методы для перебора всех элементов обрабатываемой структуры данных (массива, линейного списка, дерева, коллекции) для совершения над каждым из них заданного действия [9, с. 458-460].

А сколько разнообразных управляющих конструкций [10-13] было предложено ввести в языки программирования, чтобы создавать программы, предусматривающие запуск и взаимодействие параллельных процессов!

Так что можно смело утверждать, что потребность встраивать новые управляющие конструкции в существующий язык программирования у программистов, его использующих, будет оставаться всегда. Вопрос в том, как же эту потребность можно удовлетворить?

Понятно, что при использовании обычного языка программирования программист не может сам добавлять в язык новую нужную ему управляющую конструкцию. Ведь для этого надо вносить изменения в компилятор (или интерпретатор) языка, исходный текст которого ему, как правило, недоступен.

А вот системы программирования со словарной организацией (ФОРТ и ДССП) такую возможность ему как раз таки предоставляют. Но чтобы реализовать такую возможность, программисту надо освоить технику разработки в этих системах особых процедур, осуществляющих доступ (через стек возвратов) к тем позициям кода, откуда эта особая процедура вызывалась. И научиться добавлять в словарь новые служебные (ключевые) слова, вызов которых приводит к исполнению этих процедур.

Далее в статье рассматриваются приёмы разработки подобных особых процедур в среде ДССП для троичной [виртуальной] машины (ТВМ) – ДССП-ТВМ [14,15,16]. В результате освоения таких приёмов программист – пользователь ДССП-ТВМ – получает возможность добавлять в язык ДССП новые управляющие конструкции, в том числе и свои собственные, т.е. разработанные им собственноручно на языке самой системы.

1. Базовые управляющие конструкции ДССП-ТВМ

В ДССП-ТВМ вызов слова эквивалентен вызову процедуры (или команды), предусмотренной для его исполнения. Для организации ветвлений и циклов в ДССП помимо обычного (безусловного) вызова слова предусмотрены команды для условного и повторяемого вызова слова, а также разнообразные команды выбора слова из нескольких в зависимости от условия.

В ДССП (и в ФОРТе) обработка данных осуществляется через арифметический стек. Поэтому в качестве условия в таких командах принято оценивать значение (**T**), сформированное в вершине стека: нулевое оно (=0), положительное (>0) или отрицательное (<0), а затем удалять его из стека непосредственно перед вызовом выбранного слова на исполнение.

В ДССП для одиночного ветвления предлагаются команды вызова слова по условию:

```
{T} IF- P {} { вызов слова P, если T <0 }
{T} IF0 P {} { вызов слова P, если T =0 }
{T} IF+ P {} { вызов слова P, если T >0 }
```

Для двоичного ветвления предлагаются команды вызова (выбора) одного из двух слов:

```
{ выбор: либо вызов слова P1, либо слова P2 }
{T} BR- P1 P2 {если T<0,то P1, иначе P2}
{T} BR0 P1 P2 {если T=0,то P1, иначе P2}
{T} BR+ P1 P2 {если T>0,то P1, иначе P2}
```

Команда троичного ветвления предлагает выбрать на исполнение одно из трёх слов в зависимости от знака вершины стека (**T**):

```
{T} BRS P- P0 P+ {если T<0,то вызов P-}
{если T=0,то вызов P0; если T>0,то вызов P+}
```

Управляющая конструкция для многовариантного выбора, аналогичная оператору **case** (языка Паскаль), представлена в ДССП с помощью двух команд **BR** и **ELSE** :

```
{V} BR V1 P1 V2 P2 … VX PX ELSE P0
{ вызов слова Pi (если V=Vi), либо P0 }
```

Заметим, что в качестве слов Vi здесь можно задавать не только слова-константы, но и любые другие слова, которые могут поместить в стек одно значение для его последующего

сравнения с заданным значением V. Порядок исполнения этой нетривиальной управляющей конструкции поясняет блок-схема на рис. 1.

{V}**BR** V1 **P1** V2 **P2** … VX **PX ELSE P0**

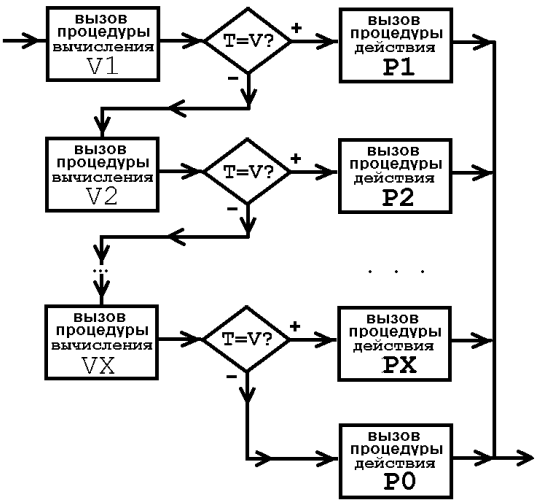

Рис. 1. Блок-схема конструкции **BR … ELSE**

Для организации циклов в ДССП предусмотрены команды **LOOP DO- DW** :

LOOP Body { многократно повторяемый вызов слова **Body** возможно бесконечное число раз }
{**n**} **DO- Body** **≈ for k=n-1 downto 0 do Body** **{** вызов слова **Body n** раз с убывающим значением параметра (в вершине стека) k=n-1,..,1,0 }
Cond {**T**} **DW Body** {} **≈ while Cond do Body** **{**многократный вызов слова **Body** (тела цикла), пока вызываемое слово **Cond** (условие цикла) помещает в вершину стека ненулевое значение**}**

Порядок исполнения цикла со счётчиком поясняет блок-схема команды **DO-** на рис. 2., а цикла с условием — схема команды **DW** на рис. 3.

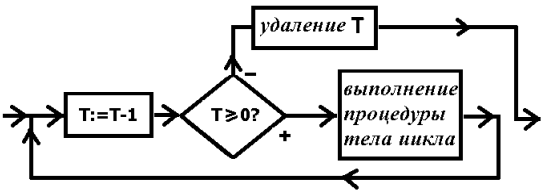

Рис. 2. Блок-схема команды цикла со счётчиком **DO-**

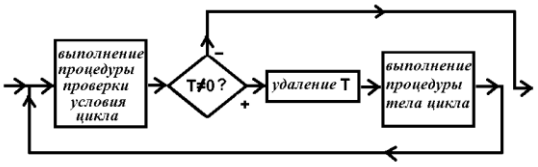

Рис. 3. Блок-схема команды цикла условием **DW**

Представленные здесь управляющие конструкции считаются базовыми, поскольку соответствующие им слова уже занесены в первоначальный словарь ДССП-ТВМ. В прежних версиях ДССП все эти управляющие команды приходилось реализовывать отдельными процедурами на языке ассемблера базовой машины. Но в версии ДССП-ТВМ почти все они теперь реализуются напрямую командами самой троичной машины (ТВМ). В настоящее время лишь команды **BR** и **ELSE** реализованы в виде процедур ассемблерного ядра ДССП.

Отметим важную особенность всех управляющих конструкций ДССП: тела ветвлений и циклов должны обязательно быть оформлены в виде отдельных слов, что стимулирует дробление программы на более мелкие процедуры. Поэтому ДССП характеризуется как система, поддерживающая процедурный вариант структурированного программирования.

Заметим, что впервые такой вариант структурированного программирования был обеспечен системой команд экспериментальной ЦМ «Сетунь-70» [7]. Можно образно сказать, что в «Сетуни-70» он был, таким образом, поддержан на уровне машинного кода. А в ДССП он поддерживается уже на уровне так называемого сшитого кода, который используется для внутреннего представления ДССП-программ.

2. Процедурный сшитый код для тел ДССП-процедур

Для реализации процедур, исполняющих слова-команды управляющих конструкций, необходимо ознакомиться, как в ДССП-ТВМ устроено внутреннее представление откомпилированных программ.

Обычно для такого представления в системах ДССП и ФОРТ используется особый вид кода, состоящий, в основном, из адресных ссылок на вызываемые при его исполнении процедуры. Такой вид кода получил давно устоявшееся название сшитый код (threaded code).

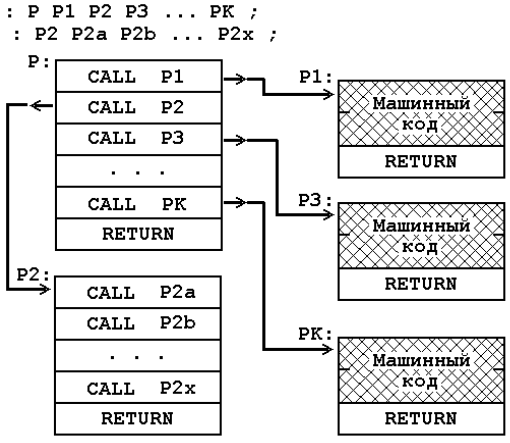

Рис. 4. Строение тел ДССП-процедур

В различных версиях ФОРТа и ДССП применялись разнообразные варианты сшитого кода: прямой; косвенный, знаково-косвенный,

двойной косвенности [17]. В ДССП-ТВМ применяется вариант процедурного сшитого кода [18], который может напрямую исполняться троичной (виртуальной) машиной. Главная особенность такого кода заключается в том, что адреса процедур, содержащиеся в нём, воспринимаются троичной машиной как команды вызова процедур, расположенных по этим адресам в памяти (см. рис. 4).

В ДССП при компоновке кода программы соблюдается так называемый принцип «1 в 1», согласно которому одному слову компилируемого предложения исходного текста должно соответствовать одно машинное слово формируемого машинного кода. Так что в ДССП-ТВМ каждому слову, помещаемому в словарь, назначается одно троичное (27-тритное) слово машинного кода, которое компилятор помещает в позицию формируемого тела компилируемой процедуры.

Если действие, вызываемое этим словом, может быть исполнено одной командой ТВМ, ему назначается слово с кодом этой команды. Иначе для исполнения слова формируется тело процедуры, состоящей уже из последовательности машинных команд, завершающейся командой возврата из процедуры. И в таком случае определяемому слову в словаре приписывается адрес расположения создаваемой для него процедуры, который при исполнении будет расшифровываться машиной как команда вызова процедуры с заданного адреса.

В виде исключения ДССП-компилятор особым образом обрабатывает слова, представляющие числовые и символьные константы (`'X'`), а также строки символов (`"x..x"`) и строки печатаемого текста (`."x..x"`). И при этом принцип “1 в 1” в настоящее время удаётся соблюдать лишь для констант, значение которых укладывается в два трайта (18 тритов). И для слова, изображающего такую константу **V**, формируется 27-тритный код команды засылки её значения в стек (DPushI_**V**), где само значение **V** записано в двух младших трайтах.

2.1 Средства доступа к телу сшитого кода через стек возвратов

Процедуры, реализующие действия слов, в ДССП не имеют явных параметров. Параметры для обработки вызванной процедуре передаются через арифметический стек. Входными параметрами для процедуры (слова) могут служить величины, помещённые в стек до её вызова, а выходными параметрами могут считаться величины, оставленные в стеке в результате её исполнения.

Но кроме этих величин процедуре может потребоваться доступ к тем позициям тела сшитого кода, которые располагаются вслед за командой, откуда эта процедура была вызвана. Именно такой доступ как раз и необходим тем особым процедурам, которые создаются для исполнения управляющих команд, и приёмы реализации которых являются предметом нашего дальнейшего обсуждения.

Схему доступа к этим позициям (см. рис. 5) рассмотрим на примере такой команды, процедура исполнения которой принимает **m** параметров X1,X2 .., X**m** из арифметического стека и которой для обработки доступны также **n** слов в теле сшитого кода, стоящих вслед за командой её вызова.

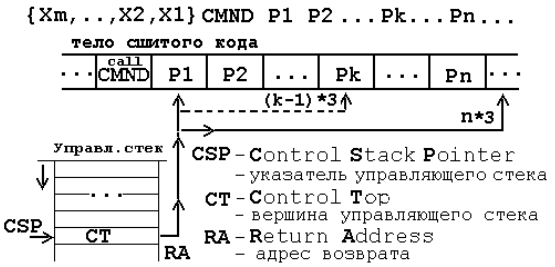

Рис. 5. Схема доступа к телу сшитого кода

Адрес возврата (RA) вызванной процедуры запоминается в вершине (CT) управляющего стека (CSP). Он указывает на позицию первого параметра в теле (P1), расположенного сразу вслед за позицией самой команды вызова процедуры (call CMND). Для взятия k-параметра (Pk) из тела необходимо сдвинуться по телу ещё на (k-1) позицию, для чего прибавить к адресу RA значение (k-1)*3, т.к. адресация ведётся с точностью до трайта, а одна позиция кода занимает троичное слово из 3-х трайтов.

После исполнения процедуры необходимо изменить адрес возврата в вершине управляющего стека так, чтобы вернуться из процедуры на продолжение программы в позицию, расположенную за последним параметром (P**n**) всей управляющей конструкции, охватываемой этой командой. И для этого адрес возврата надо увеличить на значение **n***3.

Для доступа к вершине управляющего стека в ДССП-ТВМ предусмотрены две команды:

@R	взять значение вершины из стека возвратов и заслать его в стек данных
!R	взять значение вершины из стека данных и заслать его в стек возвратов

С помощью этих двух команд можно реализовать все те действия, которые схематично были представлены на рис. 5 и которые могут потребоваться нам для разработки самых разнообразных управляющих команд в дальнейшем.

Чаще всего для взятия очередного параметра из тела вызова с продвижением указателя возврата на следующую позицию тела используется последовательность команд вида:

```
@R C 3+ !R @W
```

Поскольку такой ряд действий приходится исполнять достаточно часто, в ДССП для его исполнения выделено специальное слово **GTP** (которым мы далее и будем заменять этот ряд операций).

2.2 Приёмы передачи параметров процедуре из тела её вызова

Для начала рассмотрим пример реализации процедуры, которой из тела её вызова передаётся всего один параметр. Эта процедура должна печатать таблицу заданной двуместной троичной логической функции (операции):
{StrName,Len} **PrintT3LogFunc TFunc** {}
Для неё в стеке данных задаётся строка названия функции (адрес и длина), а в позиции следом по телу располагается команда или адрес процедуры, предназначенной для вычисления значения логической функции:

В теле определения процедуры

```
: PrintT3LogFunc {StrName,Len}
   .xy {x,y} @R C 3+ !R @W {GTP}
  {x,y,'TFunc} EXEC {TFunc(x,y)}
  ." =" .T9 ." = "
  {StrName,Len} TOS ."(x,y)" CR { };
: .T9  {V} 9 TON3 {V} ;
: .xy .---000+++. ." x=" C .T9 CR
 .-0+-0+-0+. ." y=" C .T9 CR {x,y};
```

с помощью операций **@R C 3+ !R @W** (или **GTP**) выбирается (в стек данных) первый параметр из тела её вызова. По взятому из тела параметру-адресу далее командой **EXEC** вызывается процедура (TFunc), назначенная для вычисления значения троичной логической функции от двух параметров (x,y), заданных в стеке.

После определения такой универсальной процедуры **PrintT3LogFunc** можно воспользоваться ей многократно, чтобы напечатать таблицы троичных операций, исполняемых командами ТВМ:

```
  "TADD"  PrintT3LogFunc TADD
  "TMUL"  PrintT3LogFunc TMUL
  "TMIN"  PrintT3LogFunc TMIN
  "TMAX"  PrintT3LogFunc TMAX
```

А также некоторых троичных логических функций, определённых на их основе:

```
{{ TSUB(x,y) = (x-y) mod 3 = TADD(x,NEG(y))
 : TSUB {x,y} NEG TADD {x-y} ;
{{ Функция следования Брусенцова:
{{ tSLBr(x,y)= x*y( x*y-x+y )
: tSLBr {x,y} C2 C2 TMUL C E4
  {x*y,y,x*y,x} TSUB TADD TMUL
  { x*y(y+x*y-x) } ;
```

```
: Use_PrintT3LogFunc
." Таблицы функций троичной логики:" CR
 CR "TSUB"  PrintT3LogFunc TSUB
 CR "tSLBr" PrintT3LogFunc tSLBr ;
```

В рассмотренном примере процедура (TFunc), взятая из тела в качестве параметра, вызывалась лишь один раз при исполнении процедуры **PrintT3LogFunc**. Приведём теперь пример, где процедуре сортировки одномерного массива (вектора) троичных слов в качестве параметра из тела вызова передаётся процедура (или команда) сравнения двух чисел, которая вызывается многократно, т.е. всякий раз для сравнения двух элементов вектора. С помощью такой процедуры сравнения можно задать порядок, в каком надлежит отсортировать числа массива (по возрастанию, по убыванию или ещё по какому-то критерию). Место её вызова выделено здесь подчёркиванием:

```
{{ {AdrV,Len} SortVctr CmpFunc { }
: SortVctr {AdrV,Len} GTP E3 E2
  {'CmpFunc,A,L} DO- SortVI DD { };
: SortVI {'F,A,i} MaxVI {i=L-1,..,0}
  {'F,A,i,k} SwapVIk D {'F,A,i};
: MaxVI {'F,A,i}
 {опр.индекс макс.элемента среди A[0]..A[i] }
  C {i,k} C2 {i} DO- MaxVj {i,k};
: MaxVj{'F,A,i,k,j} C4 C C4 SHL + @W
  {..,A,A[k]}E2 C3 SHL + @W {..,A[k],A[j]}
  {'F,A,i,k,j,A[k],A[j]} 7 CT EXEC {!!}
  IF+ {if A[k]<A[j]} MaxVk=j {'F,A,i,k,j} ;
: MaxVk=j {..,k,j} E2 D C {..,k=j,j} ;
: SwapVIk {переставляет эл-ты A[k]<=>A[i] }
  {..,A,i,k} C3 C C3 SHL + {'A[k]}
  E2 C4 SHL + {..,A,i,k,'A[k],'A[i]}
  {--,'A[k],'A[i]} C2 @W C2 @W
  {--,'A[k],'A[i],A[k],A[i]} E3 !W {A[i]<=A[k]}
  {--,'A[k],A[i]} E2 !W{A[k]<=A[i]} {..,A,i,k};
```

Используя процедуру такой универсальной сортировки, можно выстроить вектор чисел в нужном порядке такими простыми вызовами:

```
    9 VCTR Z    : 'Z 0 ' Z ;
: <mod10{x,y} E2 mod10 E2 mod10
{xmod10,ymod10}<{(xmod10)<(ymod10)?};
: mod10{z}10 /DivMod E2D {zmod10};
```

```
'Z 10 SortVctr < {по возрастанию}
'Z 10 SortVctr > {по убыванию}
{и в порядке возрастания младшей цифры:}
'Z 10 SortVctr <mod10
```

2.3 Разработка итераторов

Следующим рассмотрим пример передачи процедуры в качестве параметра другой процедуре для того, чтобы вызывать её многократно для выполнения обработки каждого элемента перебираемой структуры данных. Такие универсальные процедуры, предназначенные для обхода всех элементов имеющейся структуры данных с выполнением над каждым из них процедуры (или функции), заданной ей в качестве параметра, с некоторых пор стали назы-

вать **итераторами** (после их применения в широко известной библиотеке Turbo Vision [9]).

Сначала в качестве простейшего примера создания итератора рассмотрим универсальную процедуру обработки элементов произвольно заданного вектора двухтрайтных значений в порядке их обхода от (n-1)-го элемента до 0-го:

```
{..,A,n}ForEachTTVctr Operation{} ...
{{    {..,A,i,Xi} Operation {..,A,i,newXi}
: ForEachTTVctr{..,A,n}
   DO- DoEachTTVItem {..,A} D{..}
   @R 3+ !R{RA:=RA+3} ;
: DoEachTTVItem{..,A,i} C2 C2 C + +
   {..,A,i,'Ai} @TT {..,A,i,Xi}
   @R @R C !R E2 !R @W
   {'Operation} EXEC {..,A,i,newXi}
   {..,A,i,newXi} C3 C3 C + +
   {..,A,i,newXi,'Ai} !TT {..,A,i} ;
```

С помощью этой универсальной процедуры можно умножить каждый i-ый элемент вектора на 10, увеличить на i, распечатать через запятую как десятичное или девятеричное число:

```
 9 DTRYTE VCTR Z    : 'Z 0 ' Z ;
: *10 {X} 10 * {newX=X*10} ;
: +i {i,X} C2 + {i,newX=X+i} ;
: .[i], {i,X} . ',' TOB {i,X} ;
: .9[i], {i,X} C 0 TON9 ',' TOB {i,X};
```

```
 'Z 10 ForEachTTVctr *10
 'Z 10 ForEachTTVctr +i
 'Z 10 ForEachTTVctr .[i], CR
 'Z 10 ForEachTTVctr .9[i], CR
```

В качестве более сложного примера реализации итератора рассмотрим универсальную процедуру обхода всех элементов (узлов) линейного списка, на основе которого построен словарь ДССП:

```
: ForEachWord GTP {адрес ProcNode}
SLOVAR NextNode? DW HndlNode DD{};
: NextNode? @W C ;
: HndlNode {ProcNode,NodeAdr}
   C2 EXEC {ProcNode,NodeAdr} ;
```

{ProcNode – процедура из тела вызова, назначаемая для исполнения над каждым узлом словаря}

Пример применения итератора для распечатки и подсчёта количества тех слов словаря, в имени которых содержится заданная строка:

```
: .WORDS? {pWord,Len}
 0 {Cnt=0} ForEachWord .=Word? CR
 ." count of words= " . DDD { } ;
: .=Word? {pWord,Len,Cnt,PN,NodeAdr}
 5 CT 5 CT C3 _WAdr C4 _WLen
 str_pos? 0 >= IF+ .+Word ;
: .+Word {Cnt,PN,NA} E3 1+ E3
 {Cnt+1,PN,NA} .Word {Cnt+1,PN,NA} ;
: .Word {печать имени слова с пробелом}
{Adr} C _WAdr C2 _WLen TOS SP{Adr};
```

```
{pStr1,L1,pStr2,L2} str_pos? {pos}
определяет позицию pos, где содержится строка (pStr1,L1) в строке (pStr2,L2)
```

Другой вид итераторов **FirstWordThat**, предусмотренный в ДССП, позволяет вести поиск в словаре требуемого слова и в качестве результата получить указатель на найденный в нём узел этого слова.

2.4 Команда вызова процедуры из заданной в теле таблицы

Список параметров P1,..,P**n**, задаваемый в теле вслед за вызовом команды **call_CMND** (см. рис. 5), удобен для представления в этом месте таблицы процедур (или команд), из которых предстоит выбирать на исполнение одну по заданному в вершине стека номеру (k). Продемонстрируем, как можно реализовать команду такого многовариантного выбора для случая, когда длина такой таблицы процедур уже известна (9) заранее:

```
{{... {k} ExecPTable9 P1 P2 ... P9 ...
: ExecPTable9 {k} 5 - #4 TMUL 4+
 {скорректируем k=(k-5)&(.++.)+4 }
  SHL  @R C 27 + !R {RA=RA+9*3}
  {3*k',RA} + @W {Pk} EXEC ;
```

или задаётся (**n**) перед началом таблицы:

```
{...{k} ExecPTable n P1 P2 ... Pn ...}
{{   if k<1 then call_P1 else
{{   if k>n then call_Pn else
{{ { if k in [1,n] then } call_Pk
: ExecPTable {k} @R C @TT {k,RA,n}
   C 1+ SHL C3 + !R {RA=RA+(n+1)*3}
 {k,RA,n} E2 E3 {RA,n,k} crctK
  E2D SHL + @W {Pk} EXEC { };
: crctK {if k>n then k:=n; if k<1 then k:=1;}
  {n,k} C C3 > IF+ k:=n {n,k}
        C 1 < IF+ T1 {n,k'} ;
 : k:=n {n,k} D C {n,k'=n} ;
```

3. Разработка новых команд для организации ветвлений

Для реализации процедур, предназначенных для исполнения команд ветвлений, необходимо извлекать из вызываемого тела адрес процедуры (или команду), выбранной(-ую) на исполнение, а также передвигать адрес возврата по телу сшитого кода, чтобы он стал указывать на позицию, расположенную следом после всей управляющей конструкции, подлежащей исполнению.

Сначала рассмотрим простейшие приёмы реализации команд для организации одиночного, двоичного и троичного ветвлений, а затем - команды, обеспечивающие сложные конструкции многовариантного выбора (ветвления).

3.1 Реализация команд пропуска

Команды условного вызова **IF- IF0 IF+**, используемые для одиночного ветвления, можно трактовать как команды пропуска последующего слова при соблюдении обратного условия. Продемонстрируем приёмы разработки подобных команд на примерах определения процедур для пропуска по заданному условию одной, трёх или серии из **n** команд, схемы которых представлены на рис. 6, 7, 8.

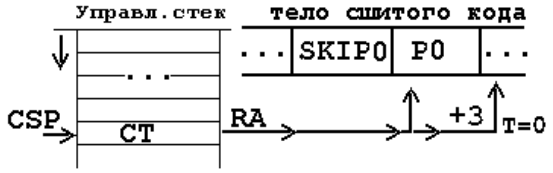

Рис. 6. Схема команды пропуска **SKIP0**

```
...{T} SKIP3- P1 P2 P3 ...
```

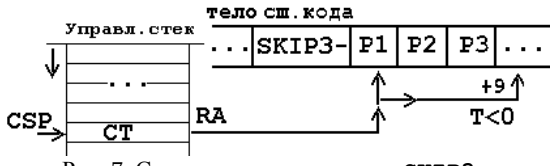

Рис. 7. Схема команды пропуска **SKIP3-**

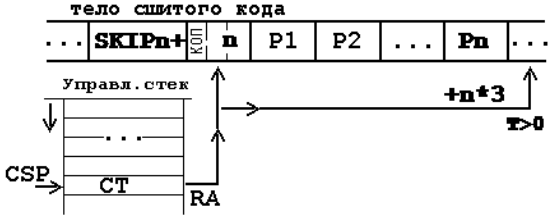

Рис. 8. Схема команды пропуска **SKIPn+**

```
{{проскок следующей команды (P0),если
{{вершина стека нулевая: ... SKIP0 P0 ...
: SKIP0 {x} @R E2 {RA,x} IF0 3+ !R
{ if x=0 then RA:=RA+3 } { } ;
```

```
{{проскок следующих 3-х команд (P1,P2,P3),
{{если вершина <0 ... SKIP3- P1 P2 P3 ...
: SKIP3- {x} @R E2 {RA,x} IF- 9+
  !R{if x<0 then RA:=RA+9 }; : 9+ 9 + ;
```

```
{{проскок следующих n команд P1,P2,P3 .. Pn,
{{если вершина стека положительна:
{{ ... SKIPn+ n P1 P2 P3 ... Pn ...
: SKIPn+ {x} @R C @TT
  {@W #444444444 TMUL }
  {x,RA,n} SHL E2 E3 {RA,3*n,x}
  {RA,3*n,x} BR+ + D 3+ !R { } ;
{if x>0 then RA:=RA+(n+1)*3 else RA:=RA+3;}
```

Заметим, что для извлечения из тела числа **n** в процедуре **SKIPn+** применялась операция **@TT**, чтобы сразу взять из памяти двутрайтное значение (хотя в комментарии далее показан и другой возможный вариант выделения числа **n** из 27-тритного кода команды DPushi_**n**, сформированного компилятором для слова **n**).

3.2 Реализация команд двоичного выбора (ветвления)

При реализации команд двоичного ветвления требуется выбрать на исполнение одно из двух слов, расположенных следом в теле:

```
{{Команда 2-го ветвления ... BR0N P1 P2 ...
{{ if T<>0 Then P1 else P2
: BR0N {T} @R C E3 {RA,RA=^P1,T}
   IF0 3+{RA,^Pi}E2 6 + !R{RA:=RA+6}
  {^Pi} @W EXEC { } ;
```

```
{{ Команда 2-го ветвления ... BRLE P1 P2 ...
{{ if T<=0 Then P1 else P2
: BRLE {T} @R C E3 {RA,RA=^P1,T}
  IF+ 3+{RA,^Pi}E2 6 + !R{RA:=RA+6}
   {^Pi} @W EXEC { } ;
```

3.3 Реализация команд троичного выбора (ветвления)

При реализации команд троичного ветвления предстоит в зависимости от условия выбрать на исполнение одно слово из расположенной следом по телу тройки слов и обеспечить дальнейшее продолжение программы с позиции, стоящей за этой тройкой.

```
...{x,a,b} BRSEG P1 P2 P3 ...
```

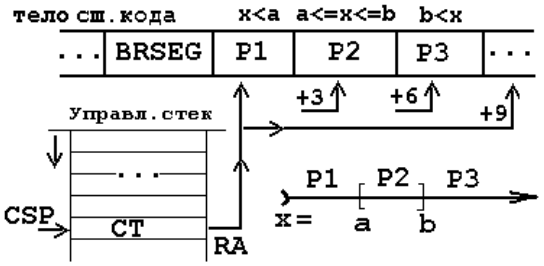

Рис. 9. Схема команды ветвления **BRSEG**

Приёмы реализации команд троичного ветвления рассмотрим на примере команды **BRSEG** (см. рис. 9), которая вызывает одну из трёх предусмотренных процедур (или слов-команд) в зависимости от расположения оцениваемого значения **x** на числовой оси слева, внутри или справа от заданного сегмента [**a,b**]:

```
{{ Команда троичного ветвления BRSEG:
{{ ... {x,a,b} BRSEG P1 P2 P3 ...
{call P1,if x<a ; call P2,if x in[a,b]; call P3,if b<x}
: BRSEG {x,a,b} inSeg? {T=-1/0/+1}
   @R C E3 {RA,RA=^P1,T}
   1+ SHL + {RA,^Pi=^P1+3*(T+1)}
   E2 {^Pi,RA} 9 + !R {RA:=RA+9}
   {^Pi} @W EXEC { } ;
: inSeg? {x,a,b} C3 E2 CMP E3
   {x cmp b, a, x} E2 CMP
   {x cmp b, x cmp a } tCarry {-1/0/+1}
   {= -1,if x<a ; 0,if x in[a,b]; +1,if b<x } ;
: tCarry{x,y} C2 C2 TADD E3
   {x+y,y,x}TMUL TMUL NEG {-x*y*(x+y)};
```

3.4 Реализация команд многовариантного выбора (ветвления)

В многовариантных управляющих конструкциях ветвления предлагается выбрать действие из предусмотренного заранее списка процедур (или команд). В ДССП подобный список располагается следом за командой **BR** (см. рис. 1), где перед каждой процедурой (Pi) помещается ещё и слово, выдающее значение (Vi), при совпадении с которым оцениваемого значения (V) и должна выбираться данная ветка продолжения программы.

Раннее в одной из версий ДССП предусматривалась ещё одна команда многовариантного выбора **BRINT**, которая позволяла выбрать одну из процедур в зависимости от того, в какой из нескольких полуинтервалов, заданных числами V1,..,Vk, попадает оцениваемое значение V. Продемонстрируем, как можно реализовать в ДССП-ТВМ такую новую команду для многовариантного ветвления.

Если заменить в блок-схеме исполнения команды **BR** на рис. 1 все условия вида **T=V?** на **T>=V?**, то получившаяся блок-схема станет задавать порядок исполнения команды **BRINT**:

```
{x} BRINT V1 P1 V2 P2 ... Vk Pk _ELSE P0
  if x<=V1 then P1 else
  if x<=V2 then P2 else ...
  if x<=Vk then Pk else P0
```

Фрагмент тела сшитого кода, который сформирует компилятор для управляющей конструкции **BRINT ... _ELSE**, схематично представлен на рис. 10. На нём стрелками изображено, как пройти по этому телу и найти правильный вариант процедуры для выбора, а также найти позицию продолжения программы после её исполнения.

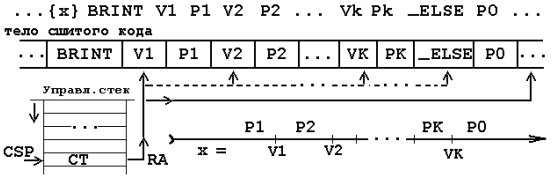

Рис. 10. Схема команды выбора **BRINT .. _ELSE**

Руководствуясь этой пояснительной схемой, можно предложить такую реализацию процедур исполнения команд **BRINT** и **_ELSE**:

```
: BRINT {поиск такого Pi, что Vi>=V }
  {V} BRI_Vi? DW BRI_nextViPi
  {V} D @R C !R {RA} 3+ @W {Pi}
   {поиск конца _ELSE для BRINT}
    BRI_ELSE?_++ DW NOP
  @R 6 + !R {RA=>на поз.после _ELSE P0}
  {Pi} EXEC {исполняем Pi} ;
: BRI_Vi?{V} @R @R C !R E2 !R @W
  {V,'Vi}EXEC{V,Vi} C2 < {V,Vi<V?};
```

```
: BRI_nextViPi {V} {RA':=RA'+6}
   @R @R 6 + !R !R {V} ;
: BRI_ELSE?_++ {[RA,RA!DW]} @R @R
 {[]RA!DW,RA}6 + C !R{RA:=RA+6}
 {[RA]RA!DW,RA}E2 !R{[RA,RA!DW]RA}
 {[RA,RA!DW]RA} @W '' _ELSE <>
 {[RA,RA!DW] Mem[RA]<>'_ELSE?} ;
```

```
: _ELSE{V}{вызвали вместо очередного Vi}
  @R D{удалим адрвозврата из вызова EXEC}
  @R D{удалим адрвозврата из BRI_Vi?}
  @R {адрес возврата из BRINT --> _ELSE}
  3+ !R {передвинем его дальше по телу}
 {V}D {};{и сразу же вернёмся по нему на
 продолжение исполнения тела после _ELSE}
```

Заметим, что такую управляющую конструкцию **BRINT .. _ELSE** можно считать расширенным вариантом команды **BRSEG**, так как

```
X a b BRSEG P1 P2 P3
```

почти что эквивалентно

```
x BRINT a P1 b P2 ELSE P3
```

4. Создание новых команд для организации циклов

Команды повторения **LOOP DO- DW**, предусмотренные в ДССП-ТВМ для организации циклов, в настоящее время исполняются напрямую троичной [виртуальной] машиной. Но первоначально они были реализованы в ДССП-ТВМ сначала на языке самой системы, а затем (для эффективности) переписаны на языке ассемблера ТВМ и помещены в ассемблерное ядро, которое подключается при компоновке ДССП-программы.

Продемонстрируем здесь их возможные реализации в виде процедур на языке ДССП.

4.1 Команда бесконечного цикла и команда выхода из него

Команду **LOOP** реализовать проще всего: надо взять из тела вызова процедуру **PBody** (или команду), представляющую тело цикла, и перед её вызовом скорректировать адрес возврата (RA) так, чтобы после исполнения тела цикла осуществить возврат на позицию, откуда будет снова взята на исполнение эта же команда **LOOP** (см. рис. 11):

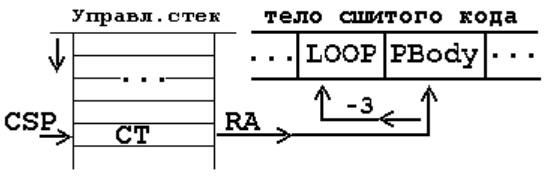

Рис. 11. Схема команды цикла **LOOP**

```
{{ команда организации бесконечного цикла
{{ ... LOOP_ PBody ...
: LOOP_ @R C 3- !R {RA:=RA-3}
  {^PBody} @W {PBody} EXEC { } ;
```

При применении команды **LOOP** обычно предполагается, что предусматриваются также и некоторые команды, с помощью которых можно будет осуществить экстренный выход из организуемого ею бесконечного цикла. Рассмотрим пример реализации одной из них:

```
{{ Команда EXIT_LOOP, чтобы прекратить
{{ цикл, исполняемый командой LOOP_
: EXIT_LOOP @R { адрес возврата (AB) }
  D {удалили AB из процедуры EXIT_LOOP}
@R D{удалим AB из EXEC в теле проц. LOOP}
 @R {AB из процедуры, реализующей LOOP }
6 + {изменим, чтобы продолжить исполнение}
 !R {программы после команды LOOP} { } ;
```

Заметим, что такой командой можно осуществить выход из цикла **LOOP**, только если употребить её в теле самой процедуры цикла (**PBody**). А в общем случае для прекращения нескольких вложенных (друг в друга) процедур или циклов следует применять предусмотренный в ДССП-ТВМ механизм обработки исключительных ситуаций [14, п. 4.1].

4.2 Команда цикла со счётчиком

При исполнении команды цикла **DO-** (см. рис. 2) всякий раз перед вызовом процедуры (команды) тела цикла необходимо уменьшать на 1 значение счётчика–параметра цикла, которое сохраняется в вершине стека, и проверять, что оно остаётся неотрицательным. При продолжении цикла адрес возврата требуется вернуть в позицию самой команды цикла **DO-**, а для завершения цикла передвинуть его вперёд на позицию слова, стоящего за всей конструкцией этого цикла (см. рис. 12):

```
{{команда организации цикла со счётчиком
{{ ... {n} DO- PBody ...   {i} PBody {i}
: DO- {n} 1- @R C 3- !R {RA:=RA-3 }
 {n-1,RA=^PBody} @W C2 {n-1,'PBody,n-1}
     BR- DOmEnd EXEC {n-1} ;
 : DOmEnd {n-1,'PBody} @R DDD { }
   @R 6 + !R { RA:=RA+6 } ;
```

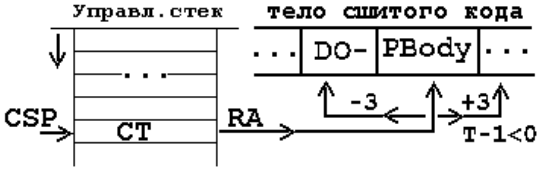

Рис. 12. Схема команды **DO-** цикла со счётчиком

4.3 Команда цикла с условием

Для реализации команды цикла с условием (см. рис. 3) необходимо передвигать адрес возврата не на одну, а уже на две позиции назад (см. рис. 13), чтобы после исполнения тела цикла снова вызывать процедуру, вычисляющую условие цикла:

```
{{ Цикл с условием: ... Cond DW Body ...
{{ Cond вычисляет условие: Да (<>0) | Нет (0)
: DW{U=0/1} @R E2 {RA,U} IF0 DWEnd
 {RA} C @W {'Body} E2 6 -
 {'Body,RA-6} !R {RA:=RA-6}
 {'Body} EXEC { } ;
: DWEnd{RA} @R D 3 + !R {RA:=RA+3};
{ сразу с выходом из команды DW !! }
```

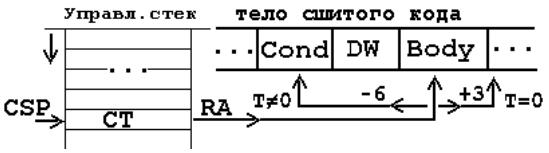

Рис. 13. Схема команды **DW** цикла с условием

4.4 Цикл с троичным условием

В ДССП для троичной машины появилась новая команда цикла с троичным условием. Порядок исполнения такой команды **DW-+** изображён в виде блок-схемы на рис. 14.

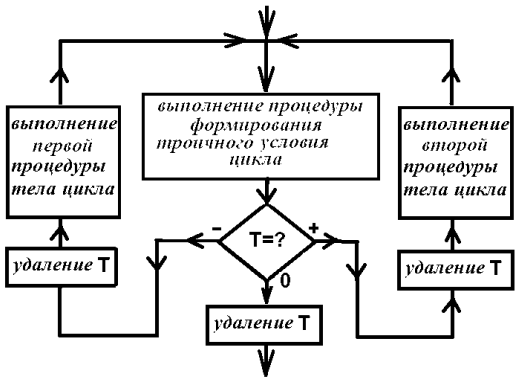

Рис. 14. Блок-схема исполнения команды цикла с троичным условием

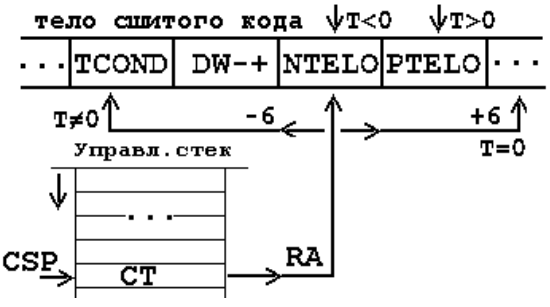

Рис. 15. Схема команды цикла с троичным условием

А схема корректировки адреса возврата в процедуре реализации этой команды — на рис. 15:

```
{{ Команда цикла с троичным условием:
{{ ... TCOND DW-+ NTELO PTELO ...
{{TCOND вычисляет троичное условие: -1,0,+1
{{если вершина<0,то выполняется тело NTELO
{{если вершина>0,то выполняется тело PTELO
{{если вершина =0, то цикл прекращается
: DW-+ {U} @R E2 {RA,U} SGN
  {-|0|+} BRS DWTN DWTEnd DWTP
  {NTELO|PTELO,RA-6} !R {RA:=RA-6}
  {NTELO|PTELO} EXEC { } ;
: DWTEnd{RA} @R D 6 + !R{RA:=RA+6};
: DWTN{RA} C @W E2 6 -{NTELO,RA-6};
: DWTP{RA}C 3+ @W E2 6 -{PTELO,RA-6};
```

Такая команда цикла с троичным условием **DW-+** была реализована сначала на языке ДССП, затем для повышения эффективности — на ассемблере ТВМ. А после того, как стала очевидной полезность её применения (см. далее п. 5.1 и [16]) на троичной машине, аналогичная команда цикла (с троичным условием) была добавлена уже в систему команд самой троичной машины, а также была реализована в её имитаторе (ТВМ).

4.5 Реализация конструкции цикла с многими условиями и телами

И в качестве последнего примера реализации собственной довольно сложной управляющей конструкции покажем, как в ДССП-ТВМ можно создать аналог конструкции цикла **do**…**od**, предложенной Э. Дейкстрой в [8, с.59-60] и упомянутой нами во введении.

Для обеспечения функционирования такого цикла с множественными условиями и телами предложим добавить в ДССП-словарь две команды: **WHILE** и **ENDW**. Порядок исполнения цикла, образуемого этими двумя командами, изображён на рис. 16.

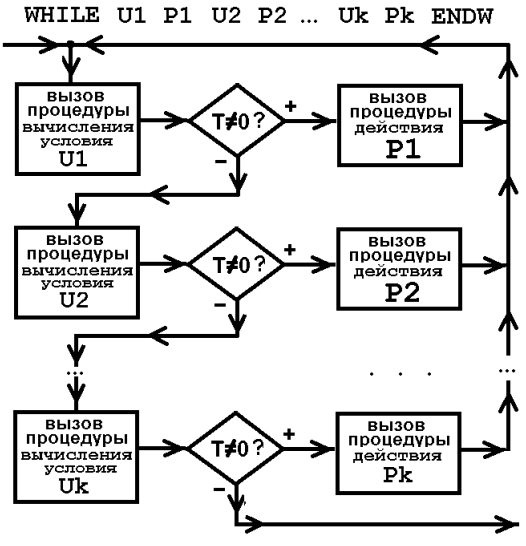

Рис. 16. Блок-схема исполнения конструкции цикла с множественными условиями и телами

Для реализации этих двух команд можно определить такие процедуры на языке ДССП:

```
{{цикл с множественными условиями и телами
{{ WHILE U1 P1 U2 P2 ... Uk Pk ENDW Px
: WHILE {[RA=^U1]} @R C 3- !R !R
  {[RAW=^While,RAi=^U1]}
  {[RAW,RAi=^Ui]} ~Ui? DW RAi+3
  {[RAW,RAi=^Pi]} @R @W
  {[RAW] Pi} EXEC {[RAW] } ;
: ~Ui?{[RAW,RAi=^Ui,RA]}
    @R @R C @W E2 3+ !R E2 !R
    {[RAW,RAi=^Pi,RA] cmnd_Ui }
   { Ui } EXEC {U} NOT {~U} ;
: RAi+3{[RAW,RAi=^Pi],RA} @R @R 3+
     !R !R {[RAW,RAi=^Ui,RA]} ;
: ENDW {[RAW,RAi=^Px,RA,RA']}
   @R D {{выбросили AB вызова call_ENDW
   @R D {{выбросили AB в команду DW
   @R @R D !R {[RAi=^Px]} ;
   {{теперь Адрес Возврата ==> ^Px
```

Действия с адресами возврата, указывающими на отдельные позиции этой конструкции цикла, отображены схематично на рис. 17. Выполняя последовательно процедуры Ui, вычисляющие условия, надо определить, какая первая из них даёт в качестве ответа значение истины (T≠0), и вызвать на исполнение в качестве тела цикла процедуру (Pi), соответствующую этому условию. Перед вызовом этой процедуры надлежит установить адрес возврата на позицию, где расположена команда **WHILE** этого цикла, чтобы после вызова процедуры тела цикла Pi продолжить исполнение цикла. Если же на очередном шаге цикла ни одна из процедур вычисления условий Ui не выдаст значения истины, то последует вызов команды **ENDW**, которая должна будет прекратить исполнение цикла и обеспечить продолжение программы с позиции, стоящей после команды **ENDW**.

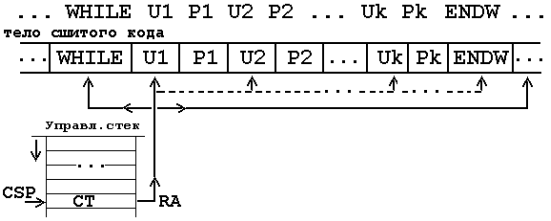

Рис. 17. Схема конструкции цикла **WHILE** … **ENDW**

5. Примеры применения новых управляющих конструкций

Приведём примеры применения некоторых новых управляющих конструкций, объяснению приёмов реализации которых (на языке ДССП) была посвящена основная часть изложенного ранее материала.

5.1 Применение команды DW-+

Сначала сравним два варианта ДССП-процедур вычисления наибольшего общего делителя двух натуральных чисел НОД(X,Y) алгоритмом Евклида: 1) с применением команды цикла **DW** с двоичным условием и команды BR- условного выбора; 2) с применением команды цикла **DW-+** с троичным условием:

```
: NOD1 {x,y} x-y? DW y-x|x-y D ;
: y-x|x-y{x,y} x-y? BR- y-x x-y ;
 : x-y?{x,y} C2 C2 - ;
 : y-x{x,y} C2 - {x,y-x};
 : x-y{x,y} E2 C2 - E2 {x-y,y};
```

```
: NOD2 x?y DW-+ y-x x-y  D ;
 : x?y{x,y} C2 C2 CMP ;
 : y-x{x,y} C2 - {x,y-x} ;
 : x-y{x,y} E2 C2 - E2 {x-y,y};
```

Обе этих процедуры вычисления НОД запускались на программном имитаторе троичной машины (ТВМ) с замером количества исполненных команд. В результате проведённых испытаний было подтверждено, что алгоритм процедуры **NOD2**, применяющий цикл с троичным условием, не только выглядит проще, но ещё и работает быстрее (более, чем в полтора раза), так как исполняется за меньшее число команд ТВМ (см. таблицу 1).

Таблица 1. Сравнение ДССП-процедур NOD1 и NOD2 по числу исполненных команд

аргументы		результат	кол-во исп.команд		
X	**Y**	**Z=NOD(X,Y)**	**NOD1**	**NOD2**	**КВ**
1010	15100	10	958	542	1.77
19683	45	9	8814	5294	1.66
1001	29	1	970	578	1.68
2000	155	5	492	300	1.64
9999	199	1	1446	870	1.66
6000	2250	750	96	64	1.48
КВ – коэффициент выигрыша, который процедура NOD2 обеспечивает по отношению к NOD1					

5.2 Применение команды BRINT

Допустим, требуется решить такую задачу. Дана последовательность чисел, каждое из которых (в диапазоне от 0 до 100) задаёт число баллов, полученных студентами группы на экзамене. Пусть признаком конца последовательности будет число -1. Требуется составить программу, которая должна подвести итог экзамену и подсчитать, сколько получено студентами группы оценок «отлично», «хорошо», «удовлетворительно» и «неудовлетворительно», полагая, что критерии выставления оценок таковы: 0-59 баллов — 2 (неуд), 60-74 — 3 (удовл), 75-89 — 4 (хорошо) и 90-100 — 5 (отлично).

Сначала приведём вариант решения этой задачи с применением конструкции множественного ветвления **BRINT .. _ELSE** :

```
  VAR Neud { кол-во неудов }
  VAR Udov { кол-во удовл. }
  VAR Hor  { кол-во хорошо }
  VAR Otl  { кол-во отлично}
: ExamResults {..,x} 0 ! Otl
   0 ! Hor 0 ! Udov 0 ! Neud
   {..,x} >=0? DW Look D { }
  ." Otl="  Otl . ." Hor="  Hor .
 ." Udov=" Udov . ." Neud=" Neud .
   DDDD { } ;
: >=0?{x} C 0 >= {x,x>=0?};
: Look {x} BRINT
        59 Neud++ 74 Udov++
        89 Hor++ _ELSE Otl++ {} ;
: Otl++ Otl 1+ ! Otl {};
: Hor++ Hor 1+ ! Hor {};
: Udov++ Udov 1+ ! Udov {};
: Neud++ Neud 1+ ! Neud {};
```

```
: Task CR
 ."Задача распределения оценок экзамена:"
  -1 60 75 90 100
  45 61 67 74 75 79 80 85 36 89 90
   0 55 70 87 77 99 68 93 88
   ExamResults  ;
```

5.3 Применение цикла WHILE...ENDW

А теперь для решения той же задачи приведём пример программы, в которой применяется цикл **WHILE** ... **ENDW**:

```
: ExamResults {..,x} 0 ! Otl
   0 ! Hor 0 ! Udov 0 ! Neud
   WHILE >=90? `Otl+  >=75? `Hor+
         >=60? `Udov+ >=0?  `Neud+
   ENDW D
  ." Otl="  Otl . ." Hor="  Hor .
 ." Udov=" Udov . ." Neud=" Neud .
   DDDD { } ;
: >=90?{x} C 90 >= ;
: `Otl+{x} D Otl 1+ ! Otl {};
: >=75?{x} C 75 >= ;
: `Hor+{x} D Hor 1+ ! Hor {};
: >=60?{x} C 60 >= ;
: `Udov+{x} D Udov 1+ ! Udov {};
: >=0?{x} C 0 >= ;
: `Neud+{x} D Neud 1+ ! Neud {};
```

Заключение

В статье рассмотрены основные приёмы разработки в ДССП-ТВМ собственных управляющих команд, который каждый пользователь-программист может создать сам, чтобы добавить в словарь ДССП. Применение этих новых управляющих команд позволяет не только выразить в сокращённом виде исходный текст созидаемой ДССП-программы, но также и уменьшить в ряде случаев количество исполняемых ею команд, что позволяет в итоге повысить её производительность.

Публикация выполнена в рамках государственного задания по проведению фундаментальных научных исследований (ГП 14) по теме (проекту) «Исследование и разработка моделей микропроцессоров, ориентированных на задачу горения для создания отечественной супер-ЭВМ. (0065-2014-0060); АААА-Б18-218042790104-8; 27/04/2018.

Development of own control constructions in DSSP for the ternary computer

A.A.Burtsev

Abstract. Programming systems with the dictionary organization (FORTH and DSSP) give to the user programmer a unique opportunity by building of the dictionary to create in fact an own programming language. As new words it is possible to add to the dictionary also such words which are intended to control the commands flow of program execution. In the article methods of development in the DSSP-TVM (the version of DSSP for the ternary computer) special procedures for such words (commands) which provide functioning of the own control constructions to be added to the DSSP language are considered in detail.

Keywords: structured programming, dictionary organization, FORTH, DSSP, ternary computer, threaded code, control constructions.

Литература

1. Баранов С.Н., Ноздрунов Н.Р. Язык ФОРТ и его реализации. - Л.: Машиностроение, 1988.- 157 с.

2. Бурцев А. А., Сидоров С.А. История создания и развития ДССП: от «Сетуни-70» до троичной виртуальной машины // Труды 2-ой международной конференции «Развитие вычислительной техники и её программного обеспечения в России и странах бывшего СССР» SORUCOM-2011, г. Великий Новгород, Россия, 12–16 сентября 2011г.— В.Новгород: Изд-во НовГУ, 2011.— С. 83–88.

3. 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.

4. Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975. –247 с.

5. Дапкунас С.А., Раннев К.Н. Простые средства обеспечения работы с подпрограммами и структурированного программирования на языке Бейсик. // Анализ и синтез дискретных устройств (межвуз. сб. статей под ред.М.К. Чиркова, С.П. Маслова). – Л.: Изд-во ЛГУ, 1984. — с. 45-49.

6. Бурцев А. А. Об одной возможности расширения языка программирования средствами обработки исключительных ситуаций. // Дискретные модели. Анализ, синтез, и оптимизация (сб. статей под ред.М.К. Чиркова, С.П. Маслова). – Спб.: СпбГУ, 1998. — с. 171-181.

7. Златкус Г.В. Ассемблер малой ЦМ «Сетунь-70» // Вычислительная техника и вопросы кибернетики. Вып №17. (сб. статей под ред. Брусенцова Н.П., Шаумана А.М.). М.: Изд-во МГУ, 1981, с. 26-33.

8. Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978. — 275 с.

9. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс. Учебное пособие. — М.: Нолидж, 1997.

10. Вирт Н. Программирование на языке Модула-2. — М.: Мир, 1987. – с. 179-182.

11. Янг С. Алгоритмические языки реального времени: конструирование и разработка. — М.: Мир, 1985. – с. 146-177.

12. Джехани Н. Язык Ада. — М.: Мир, 1988. – с. 125-180.

13. Борисов А.В. Схемы взаимодействия параллельных процессов. // Программное оснащение микрокомпьютеров (сб. статей под ред. Брусенцова Н.П. и Маслова С.П.). - М.: Изд-во МГУ, 1982. - с. 88-99.

14. А. А. Бурцев, С. А. Сидоров. Троичная виртуальная машина и троичная ДССП. // Программные системы: теория и приложения. — 2015. — Т. 6, №:4 (27). — с. 29–97. [электронный ресурс] URL: http://psta.psiras.ru/read/psta2015_4_29-97.pdf.

15. Бурцев А.А., Сидоров С.А. Программный комплекс ДССП-ТВМ для структурированного программирования троичной [виртуальной] машины. // Современные информационные технологии и ИТ-образование. 2015. — Т. 11, № 2. — с. 371–379.

16. Бурцев А.А. Особенности программирования в ДССП для троичной машины. // Современные информационные технологии и ИТ-образование, 2017. —Т.13, № 4, 2017. с. 188-196.

17. Бурцев А.А., Франтов Д.В., Шумаков М.Н. Разработка интерпретатора сшитого кода на языке Си. // Вопросы кибернетики (сб. статей под ред. В.Б.Бетелина) — М.: Изд-во НИИСИ, 1999, с. 64-77.

18. Бурцев А.А. Процедурный сшитый код для обеспечения нисходящей разработки структурированных программ в ДССП для троичной машины. // Труды НИИСИ РАН, т. 5, № 1.— М.: Изд-во НИИСИ РАН, 2015.— с. 122–129.