Considerations for writing and using Intel assembly code in Delphi projects

Использование ассемблера в Дельфи

Оглавление

Рецензия

Как это не парадоксально, но человек, который поддерживает один из крупнейших Интернет архивов готовых компонент, пишет рецензию на удивительную книгу, призывающую использовать Ассемблер…

В 1989-1993 году я много программировал на Turbo Assembler в сочетании с Turbo Pascal и Turbo Pascal for Windows и очень полюбил этот язык, позволяющий в минимальном размере компилированного кода размещать модули, которые позволяли делать многое, и самое главное, гораздо быстрее, чем стандартные решения, предлагаемые стандартными библиотеками. Классическое замечание, к которому могут присоединиться тысячи программистов (IMHO), начинавших программировать для IBM PC в те годы: мы все учились на кодах замечательной библиотеки Turbo Professional (позже – Object Professional) компании Turbo Power Software, в которой значительную часть составлял именно Ассемблер. Благодаря этой книге в наше время, когда исходные коды Turbo Professional в силу лет куда-то затерялись, теперь Вы можете приобщиться к его необычайному очарованию и использовать всю его мощь в Ваших проектах.

А зачем это нужно? Во многих случаях использование встроенного Ассемблера (BASM) даст Вам значительный выигрыш по скорости, особенно при работе со строковыми функциями, сравнении, работе с нестандартными устройствами ввода-вывода и во многом другом. Конечно, современные компьютеры и операционные системы позволяют не сильно заботиться о скорости работы программы, но при обработке большого объема данных, построении отчетов, нестандартной обработке данных и работе в реальном времени скорость обработки потоков информации, а, следовательно, принятия решений становится очень важным фактором.

Данная книга не является очередным пересказом “Руководства пользователя…”, и не только раскрывает как использовать BASM, но и лишний раз заставит Вас обратить внимание на правильность и корректность Вашего кода. А это – немаловажно, поверьте…

На мой взгляд, имя Гуйдо Гайбелса незаслуженно не очень известно широкой публике. Видимо, сказывается тенденция использования «всего готового», чему я сам способствую. Но, искусство программирование – это не только имя, известное широкой публике Дельфийского сообщества, но и тихо сказанное в ночи: «А как круто он это сделал!». А вот таких фраз Гуйдо наверняка «услышал» и «услышит» немало…

Анатолий Подгорецкий в очередной раз порадовал нас не только качественным переводом (а он и не мог быть иным, благо Анатолий очень хорошо знаком с темой книги), но и актуальной темой. Я не знаю, может, это ностальгия, но «в наше время и деревья были выше, и небо - синее», и девушки моложе», но эта книга – именно то, что позволяет надеяться, что программирование не сводится к простому расположению компонентов на форме, за что очень любят критиковать Дельфистов приверженцы Си.

Максим Пересада (mperesada@torry.net)
Torry’s Delphi Pages
http://www.torry.net

Review

It should be looked very interesting when the man which supports one from the largest Internet ready-to-use Delphi components archives, should write review for wonderful book, which is calling to use Assembler

In 19989-1993 I used Turbo Assembler and Turbo Pascal and Turbo Pascal for Windows too many and since these times I like Assembler, language, which allows put small size modules which allows to make too much and, important, must faster than standard programming libraries makes. Classic note, which can be supported by many-many programmers, which had started to make programs for IBM PC in these years: all of us took first programming lessons at excellent Turbo Professional library (later – Object Professional) by Turbo Power Software, which main part was consist from Assembler code. Due this book, in time, when source codes of Turbo Professional missed in time you can now feel all charm of Delphi built-in Assembler (BASM) and use all its power in your projects.

For what, you said? In many cases use of BASM will bring you best speed, especially in case if you are working with string functions, comparison, use non-standard input-output devices and so on. Of course, modern computers and operation systems allows you to do not take in mind speed of your applications, but, when you should process huge amount of data, operative reports, non-standard data processing and real-time data processing the speed of processing and, as consequence is the very important factor, isn't it?

This book isn't just another tales about "Developers Guide", but open your mind how to use BASM, but will another time point your attention to your code, how it is clean and correct. It is very important thing, trust me…

My opinion, Guido Gybels should be more known. Maybe, this is the result of wide using ready-to-use components tendency, maybe, this is the result of my promotion ☺

But, Art of Programming isn't just a name, but the words, which fellow programmer will say into the night: "How cool is the thing he did!". I think that such phrases Guido already "heard" and will "hear" such words in the future…

Anatoly Podgoretsky pleased us with not just high quality translation another time (but translation can not be the other, because Anatoly knows this area of programming too much), but with nice theme choice. Maybe, this is just memories, who knows, but in our times "the trees were higher and the sky was more blue", but this book is the reason to think that programming itself still "just place components at the form", thing for what many C programmers still critics Delphi programmers.

Maxim Peresada (mperesada@torry.net)
Torry's Delphi Pages
http://www.torry.net

От переводчика

Данная книга составлена на основе статей Гуйдо Гайбелса (*Guido Gybels*), опубликованных на сайте http://www.optimalcode.com/Guido/

Развитие книги продолжается, появляются новые главы, и по мере их появления они будут включаться в эту книгу.

Я благодарю Гуйдо Гайбелса за разрешение на перевод книги на русский язык и за право на публикацию ее в Интернете на основе исключительно бесплатного использования, без каких либо видов получения прибыли или оплаты по ее использованию.

Также благодарю Максима Пересаду за оказанную помощь и всю команду Торри за размещение данной книги и других книг на их замечательном сайте http://www.torry.net

Также благодарю Александр Челмодеева за помощь по корректировке книги и исправлению стилистики, без которой в ней бы было много ошибок, квалификация Александра просто великолепна.

Анатолий Подгорецкий

Введение

Вы любите Дельфи? Я да! Мое имя Гуйдо Гайбелс, и многие годы использую Borland Delphi и искренне верю, что это лучший инструмент на рынке RAD инструментов. Дельфи не только обладает мощью, но также надежностью и высокой скоростью. Язык программирования Object Pascal изумительный, а компилятор просто фантастический. Поскольку Я люблю Дельфи, Я также хочу внести маленький вклад для сообщества Дельфи, путем публикация ряда статей и фрагментов кода на этом сайте. Большое спасибо Бобу Ли (Bob Lee) за размещение на его сайте. Я также буду очень признателен за любые замечания и комментарии по поводу содержимого этих страниц. Вы также можете найти меня в Borland Delphi Newsgroups, другой отличный источник информации по Дельфи.

Вводная глава: Нужно ли это вам?

Многие программисты сегодня ассоциируют ассемблер как сложный, низкоуровневый язык программирования. Они считают его быстрым, но большинство их них думает, что его сложно или невозможно изучить. В действительности, положение не настолько сложно. Вполне возможно научиться писать хороший код, не будучи гением. С другой стороны, не думайте, что несколько уроков позволят вам производить более быстрый код, чем получится с помощью Паскаля. Причина в том, что когда вы пишите на Паскале, то вы на самом деле пользуетесь услугами высокоэффективного программиста – компилятором Дельфи. В целом код производимый им очень эффективный и быстрый. После некоторых уроков по специальным методам кодирования вы сможете сделать более эффективный код, чем Паскаль и Дельфи. Робертом Ли был создан и обслуживается наиболее исчерпывающий сайт по этой теме. Каждый серьезный программист должен познакомиться с информацией на этом сайте.

Большинство программистов считают, что собственный ассемблерный код по определению быстрее, чем скомпилированный компилятором Паскаля. Конечно, так не всегда. Плохо написанные процедуры на Ассемблере могут оказаться по качеству хуже и могут вызвать странные ошибки и проблемы в ваших приложениях. Задачей данных статей не является обучить вас основам Ассемблера. Есть много другой информации в других источниках и конечно вы должны быть в хороших отношениях с системой команд процессоров Intel (в статьях фокус сделан на серию Пентиум, особенно PII и PIII) на уровне обычного программирование. Вы можете найти руководства по семейству процессоров на сайте http://developer.intel.com. И также на сайте (http://webster.cs.ucr.edu/) «The Art of Assembly», очень солидное руководство по программированию на Ассемблере, который должны посмотреть, если хотите заняться серьезным программированием на ассемблере.

В тот момент, когда вы придете к выводу, что ассемблер это нужный вам путь, вы не должны хвататься за него, как за соломинку. Проверьте внимательно свою программу на предмет определения слабых мест. Профилировщик может немного в этом помочь, но лучше обратите особое внимание на структуру и алгоритм. Часто, вы сможете получить лучшие результаты, за счет оптимизации алгоритмов, скорее, чем с помощью ассемблера. С другой стороны, в некоторых особых случаях, таких как манипуляции с битами, ассемблер даст лучший и более простой результат.

Если вы решите, что ассемблер действительно нужен, то потратьте немного времени на планирование вашего кода и алгоритмов. Только после того, как вы будете четко представлять, что вы хотите сделать и как, вы можете приступать к реализации вашей идеи. Если вы не будете об этом думать, то получите кашу из кода, закрученные операторы и трудно управляемую программу.

Для фазы реализации существует несколько обычных правил, которым надо следовать.

Первое важное правило: делайте ваши процедуры как можно короче, ассемблер должен использоваться, только для реально необходимых вещей, в которых важна максимальная производительность. В большинстве случаев делайте это, с помощью простых, но специализированных процедур. Если вы видите, что у вас длинные куски ассемблерного кода в вашем проекте, вы вероятнее всего просто фанат ассемблера.

Второе правило: делайте ваши процедуры читабельными за счет комментирования кода. В связи с линейной структурой ассемблерного кода, каждая отдельная инструкция читается легко сама по себе. Комментарии нужны для объяснения реализации алгоритма, чтобы другие могли понять, что же вы здесь делаете. Не добавляйте комментарии ради самих комментариев. Ваши комментарии должны быть полезны и не избыточны. Например, не делайте подобного:

```
inc EDX {увеличить значение EDX}
```

Подобный тип комментариев не имеет смысла, поскольку инструкция говорит сама за себя. Комментарии должны описывать внутреннюю работу алгоритма, не дублируя мнемонику команд.

Третье правило: избегайте использования медленных инструкций. В основном должны использоваться простые инструкции вместо сложных агрегатных инструкций, так как последние реализованы с помощью микрокода. Подробное обсуждение этой темы можно найти в работе Agner Fog на сайте http://www.agner.org/assem/. Попробуйте прочитать это несколько раз, глава за главой, пока полностью не поймете основную идею. Это, однако, не означает, что вы не должны их никогда использовать, но это даст вам понимание написания оптимального кода.

И последнее правило: проверяйте свои ассемблерные подпрограммы особенно тщательно. Для подпрограмм, написанных на ассемблере, компилятор делает намного меньше предупреждений и сообщений об ошибках. Особенно будьте внимательны с логическими конструкциями вашего алгоритма. Неиспользуемые переменные и неверное использование указателей не может быть определено так же просто, как и в коде на Паскале. Так, что готовьтесь к интенсивному тестированию и отладке вашего кода. Это потребует больше времени и усилий по сравнению с отладкой Паскаль-кода. Это одна из причин делать процедуры как можно короче и читабельнее.

После этого вступления, мы готовы приступить к реальному изучению предмета. Эти статьи писались с учетом специфики - это означает, большинство информации относится к BASM (сокращение от built-in assembler – встроенный ассемблер) для всех 32-битных версий Дельфи, начиная с третьей. В основном это верно и для Дельфи 2, но это версия сильно устаревшая. Имеется также некоторые проблемы с утечкой памяти для Дельфи 2, и обычно я рекомендую использовать Дельфи версии 3.02 и выше. Если у вас версия 3,4 или 5, то вся информация в данных статьях применима к ним. Если потребуются специфические требования к версии Дельфи, то это будет особо указано.

Глава 1: Об основах ассемблерных процедур

Когда вы захотите добавить хороший и успешный ассемблерный код внутрь проекта на Дельфи, вам потребуется вызвать BASM процедуру, передать в нее переменные и принять обратно результат. Для этого вы должны знать, как работают соглашения о передаче параметров. На первом шаге мы обсудим основное: подпрограммы на ассемблере, получают некоторые переменные для обработки, возвращают впоследствии результаты. Позже, в других главах, мы обсудим, как вызывать другие функции, методы и процедуры из ассемблерного кода с помощью регистров и стека.

1.1. Где размещать ассемблерный код

Ассемблерные инструкции размещаются внутри блока **asm...end.** Эти блоки могут появляться внутри процедур и функций обычного кода, но я настоятельно не рекомендую поступать таким образом. Гораздо лучше изолировать их в отдельной функции или процедуре. Вставка **asm** блока внутри обычной процедуры создает сложности для компилятора Паскаля и код становится не эффективным с точки зрения производительности. Переменные, которые обычно передаются через регистры, в этом случае будут передаваться через стек или перезагрузку. Также, это заставляет компилятор адаптировать собственный код к вашему вставленному коду, что делает механизм оптимизации менее эффективным. Так, что это становится правилом помещать ассемблерный код в отдельную процедуру или функцию. Кроме того - это вопрос проектирования. Читабельность и управляемость вашего кода становится выше, если он помещен в свой собственный блок.

Часто, ассемблерный код ассоциируется со скоростью. Поэтому циклы вы также должны по возможности организовывать внутри ассемблерного кода. Это не сложно, а иначе вы просто потеряете множество времени за счет постоянного вызова. Вместо того, чтобы делать так (см. примечание 1):

```
function CriticalCode(...): ...; register;
asm
  ...
  {Here comes your assembler code}
  ...
end;

procedure Example;
var
  I: Integer;
begin
  I:=0;
  ...
  while I < NumberOfTimes do begin
    CriticalCode(...);
    Inc(I);
  end;
  ...
end;
```

Вы должны сделать так:

```
function CriticalCode(...): ...; register;
Asm
  ...
  mov ECX,{NumberOfTimes}
@@loop:
  ...
  {Остальной код}
  ...
  dec ECX
  jnz @@loop
```

```
  ...
end;

procedure Example;
begin
  ...
  CriticalCode(...);
  ...
end;
```

Использование цикла в обратном направлении позволяет просто проверять флаг установки нуля после команды **dec.** Если же цикл начинать с нуля, то потребуется больше на одну команду сравнения с конечным значением, каждый раз при проходе цикла.

```
  mov ECX,0
@@loop:
  ...
  inc ECX
  cmp ECX,{NumberOfTimes}
  jne @@loop
```

Другая возможность – это вычесть значение **NumberOfTimes** из **0** и затем увеличивать переменную цикла, пока она не станет равной нулю. Этот метод обычно используется, когда переменная цикла также является индексом в таблице или массиве в памяти, поскольку механизм кэширования работает лучше, чем при доступе в прямом направлении. Это можно сделать так:

```
xor ECX,ECX
  sub ECX,{NumberOfTimes}
@@loop:
  ...
  inc ECX
  jnz @@loop
```

Помните, что в этом случае базовый регистр или адрес, должен указывать на конец массива, вместо его начала.

(1) В данных главах мы специально указываем соглашение о вызове. На самом деле указание **register** избыточно, так как соглашением по умолчанию является передача параметров через регистры, это сделано исключительно для читабельности (или как дополнительный комментарий) и как напоминание читателю, что параметры передаются через регистры. Этот совет поступил от Christen Fihl, http://HSPascal.Fihl.net

1.2. Код входа/выхода и сохранение регистров

Компилятор автоматически генерирует необходимый код входа и выхода из ассемблерных подпрограмм.

Код входа выглядит так:

```
push EBP
mov  EBP,ESP
sub  ESP, {Размер стека для локальных переменных}
```

А код выхода так:

```
mov ESP,EBP
pop EBP
ret {Размер стека резервированный для параметров}
```

Однако, если ваша процедура не имеет никаких локальных переменных или параметров на стеке, то компилятор не делает кода входа/выхода, за исключением инструкции **ret.**

Код входа сначала сохраняет текущее значение регистра **EBP** на стеке, поскольку его требуется восстановить при выходе. Затем, устанавливает значение **EBP**, как базу для

доступа к параметрам и локальным переменным, которые также размещаются на стеке. Более подробно мы обсудим этот механизм позже.

Код выхода сначала освобождает память, распределенную для локальных переменных, путем подстройки указателя стека, а затем восстанавливает регистр **EBP** в его предыдущее состояние и производит возврат в вызвавшую программу. Для всех соглашений, исключая **cdecl**, процедура сама очищает стек, путем соответствующего варианта инструкции **ret**, Для соглашения **cdecl** очисткой стека занимается вызвавшая программа. Снова, все это мы рассмотрим подробнее в дальнейшем.

Внутри вашей функции или процедуры, содержимое регистров **EAX**, **ECX**, **EDX** можно полностью изменять и нет необходимости возвращать их в исходное состояние, кроме того, регистр **EAX** или его часть часто используется для возврата результата. Если вы изменяете, другие регистры общего назначения (**EBX**, **ESI**, **EDI**), то вы обязаны восстановить их первоначальное состояние до выхода из процедуры. То же самое относится и к регистрам **ESP** и **EBP**. Вы также не должны никогда изменять содержимое сегментных регистров (**ds**, **es** и **ss** указывают на один и тот же сегмент; **cs** имеет свое собственное значение; **fs** используется Windows и **gs** резервирован).

Регистр **ESP** указывает на верхушку стека, а **EBP** указывает на текущий фрейм стека и генерируется по умолчанию компилятором как код входа. Поскольку каждая инструкция **pop** и **push** изменяет содержимое регистра **ESP**, то его использование не является хорошей идеей для доступа к стеку. Для этих целей зарезервирован регистр **EBP**. Смотрите Таблицу 1, в которой приведено суммарное описание по использованию регистров в Дельфи.

И в дополнение к регистрам, вы также должны сохранять состояние флага направления. При входе в функцию флаг направления сброшен и если вы его изменяете, то вы должны сбросить его до выхода из функции, сделать это можно с помощью инструкции **cld**.

И наконец, вы также должны очень осторожно относиться к управляющему слову сопроцессора. Поскольку оно позволяет менять режим точности и округления, а также маскировать определенные исключения, то это может драматически изменить результат вычислений в вашей программе. Если у вас возникла нужда в изменении управляющего слова, то постарайтесь восстановить его значение как можно быстрее. Если вы используете типы **Comp** или **Currency**, то не уменьшайте точность!

1.3. Передача информации через регистр

При использовании соглашения по умолчанию, передачу через регистры, Дельфи может передавать до двух методов или до трех параметров через регистры процессора. Это означает, что нет необходимости генерировать фрейм стека для передачи параметров. Не все типы могут быть переданы через регистры, а только те, которые помещаются полностью в регистр. Поэтому, многие сложные типы передаются или через стек, или через память, и вместо самих типов передается указатель через регистр. Это означает, что любой тип может быть передан через регистр, как указатель, за исключением указателей методов, которые всегда передаются как два 32-разрядных указателя, размещенных на стеке.

Текущее поколение процессоров, для которых написана данная статья, обычно называются как Intel Pentium процессоры, имеет регистры шириной в 32 бита. Когда передаваемая информация не полностью использует регистр (для типов слово и байт), то используется, только часть регистра, байты используют младшие восемь бит, например **al** и для слов младшее слово, например **ax.** Указатели всегда 32-битные (по крайней мере, пока не появятся 64-битные процессоры) и занимают весь регистр полностью, например **eax.** В случае переменных типа байт и слово оставшаяся часть регистра не определена и вы не должны делать никаких предположений относительно его содержимого. Например, при

передаче байта в функцию, через регистр **al**, остальные 24 бита регистра **eax** не определены и вы, конечно, не можете рассчитывать на то, что они равны нулю. Вы просто можете использовать инструкцию **and** для очистки оставшихся бит.

```
and EAX,$FF {беззнаковый байт в AL, очистка старших 24 бит}
```

или

```
and EAX,$FFFF {беззнаковое слово в AX, очистка старших 16 бит}
```

Когда вы передаете знаковые параметры (**ShortInt** или **SmallInt**), вы должны расширить их с учетом знака. Для расширения знака для байтового параметра до двойного слова, вы должны использовать две инструкции:

```
cbw  {расширение al до ax}
cwde {расширение ax до EAX}
```

Для демонстрации, напишите следующую тестовую подпрограмму:

```
function Test(Value: ShortInt ): LongInt; register;
asm

end;
```

Разместите кнопку и метку на форме и поместите следующий код в обработчик **OnClick**:

```
var
  I : ShortInt ;
begin
  I := -7;
  Label1.Caption := IntToStr(Test(I));
end;
```

Запустите проект и нажмите кнопку. Тестовая процедура принимает параметр типа **ShortInt** через **al**. И возвращает результат типа **integer** через регистр **EAX**, который возвращает результат неизменным. Вы можете просто считать, что **EAX** имеет неизмененное значение при возврате. Теперь изменим функцию следующим образом и запустим проект снова на выполнение:

```
function Test(Value: ShortInt ): LongInt; register;
asm
  cbw
  cwde
end;
```

Единственным соглашением, согласно которому регистры используются для передачи параметров, это соглашение с ключевым словом **register**, которое также является соглашением по умолчанию. Все другие соглашения используют стек для передачи параметров в функции или процедуры. И конечно, если вы передаете более двух или трех параметров, то стек также используется для передачи оставшихся параметров. В заключение, некоторые параметры всегда передаются через стек - это указатели методов, которые в действительности состоят из двух 32-битных указателей и параметров с плавающей запятой. Обзор можно найти в таблице 2.

1.4. Передача информации через стек

При использовании соглашения по передаче параметров через регистры и передаче более двух указателей метода или трех параметров, остальные передаются через стек. Другие соглашения также используют стек для передачи параметров.

Обычно, для доступа к параметрам на стеке вы должны обращаться к ним через адресацию с помощью регистра **EBP**. Код входа по умолчанию, который генерирует компилятор, устанавливает регистр **EBP** на данный фрейм. Таким образом, использование **EBP** с нужным

смещением позволяет иметь доступ к параметрам на стеке и к локальным переменным. Посмотрим на пример с использованием соглашения о вызове **pascal**.

Данное соглашение помещает параметры слева, направо. Для примера, в следующем объявлении:

```
function Test(First, Second, Third: Integer): Integer; pascal;
```

мы имеем три 32-битных параметра типа **integer**, и каждый параметр помещается на стек следующим образом:

```
First
Second
Third
ESP ->
```

Инструкция вызова добавляет адрес возврата на стек, и стек теперь выглядит следующим образом:

```
First
Second
Third
ESP ->
Return Address
```

Компилятор автоматически генерирует код входа (см. главу 1.2) для сохранения текущего значения регистра **EBP** и затем копирует регистр **ESP** в **EBP** для доступа к фрейму стека:

```
First
Second
Third
Return Address
EBP, ESP->
Previous EBP
```

В данной точке, мы имеем доступ к параметрам на стеке, как смещение относительно регистра **EBP**. Поскольку адрес возврата находится на стеке между текущей верхушкой стека и действительными параметрами, мы можем получить их следующим образом:

```
First    =     EBP + $10   (EBP + 16)
  Second =     EBP + $0C   (EBP + 12)
  Third  =     EBP + $08   (EBP + 8)
```

В действительности, вы будете просто ссылаться на них по их именам, компилятор сам рассчитает необходимые смещения самостоятельно. Так, для выше описанного случая, напишите:

```
mov EAX,First
```

Компилятор превратит их в следующий код:

```
mov EAX,[EBP+0x10]
```

Это избавляет вас от самостоятельного расчета смещений и делает код более читабельным. Так что вы должны использовать имена везде, где только возможно (практически всегда), вместо расчета их вручную смещений.

Данные, переданные через стек, всегда занимают 32 бита, даже если вы передаете байт, оставшиеся биты просто не определены.

1.5. Локальные переменные

Так же как и в обычной процедуре, вы можете использовать локальные переменные для хранения временных значений. Они объявлятся - с помощью директивы **var** и размещаются на стеке. Компилятор генерирует необходимый код пролога, для резервирования

необходимого места на стеке, вместе с параметрами и обеспечивает доступ по имени. Вы помните, что функция объявляется следующим образом:

```
function Test(First, Second, Third: Integer): Integer; pascal;
```

Для временного хранения объявим переменную типа integer, для этого сделаем следующее объявление:

```
var
  MyTemp: Integer;
```

В результате компилятор сгенерирует следующий код и выделит место на стеке. Обратим внимание, что фрейм стека будет выглядеть следующим образом без использования локальных переменных.

```
First
Second
Third
Return Address
EBP, ESP->
Previous EBP
```

Для создания места на стеке для локальной переменной **MyTemp**, компилятор добавляет инструкцию **push,** теперь стек выглядит следующим образом:

```
First
Second
Third
Return Address
EBP ->
Previous EBP
ESP ->
? (MyTemp)
```

Для адресации этих локальных переменных опять же используется смещение относительно регистра **EBP**. Для примера, переменная **MyTemp** доступна через **EBP-4** (напомню: регистр **EBP** инициализируется компилятором в коде входа). И еще раз, нет необходимости рассчитывать это смещение вручную, достаточно использовать имя переменной **MyTemp**:

```
mov EAX,MyTemp
```

будет оттранслировано в следующий код:

```
mov EAX,[EBP-4]
```

Содержимое переменных не инициализируется при входе, и вы не должны делать никаких предположений об их начальных значениях. Поэтому вы должны сами проинициализировать их до использования:

```
mov MyTemp,0
```

Так же, вы должны с осторожностью объявлять и использовать локальные переменные. Они добавляют лишнюю нагрузку по созданию места на стеке и на освобождение при выходе. Еще важно то, что доступ к основной памяти гораздо медленнее, чем доступ к регистрам процессора. Так, что пытайтесь использовать регистры везде, где это возможно, для хранения временных переменных, вместо использования локальных переменных. В обычном коде, при включенной оптимизации, компилятор так же пытается использовать регистры для локальных переменных.

1.6. Возврат информации через регистры процессора

В большинстве случаев (зависит от типа результата), функция возвращает результат через регистры процессора. Напомним, что в отличие от передачи параметров в функцию, где в

большинстве соглашений используют стек, для возврата результата, все соглашения используют регистры для допустимых типов!

Таблица 3 содержит обзор о вариантах возврата результатов. В большинстве случаев, результат возвращается через регистр **EAX** или FP(0). Особый случай, когда в результате возвращается длинная строка или другой тип, возвращаемый через указатель. В случае длинных строк, динамических массивов, больших множеств, вариантов и больших записей, переданных через параметр с директивой **var,** используется 32-битный указатель на результат. Так, где же хранится действительное содержимое результата (например, длинных строк)? Ответ на это в том, что вы должны выделить место в куче, заполнить его данными, и вернуть указатель на эту область памяти через переменную **Result**. Заметим, что, для множеств, записей и массивов, которые могут разместиться в регистре, переменная **Result** возвращает их через регистр. Только для длинных строк, вариантов и множеств, записей и массивов, которые занимают свыше 32 бит, переменная **Result** возвращает указатель на дополнительный указатель, размещенный функцией, аналогично директиве **var** параметра (мы рассмотрим директиву параметра **var** в главе 1.9).

Не беспокойтесь, если что-то сейчас не понятно, позже мы рассмотрим эти типы подробнее.

Теперь поясним это на примере. Функция **PlusMinusLine** возвращает длинную строку, состоящую из последовательности плюсов и минусов, для формирования строки. Например, когда вы напишите так S:=PlusMinusLine(9), то **S** должна получить значение: "-+-+-+-+-".

Декларация функции следующая:

```
function PlusMinusLine(L: Integer): AnsiString; register;
```

Функция принимает один параметр: длину строки символов (L). Поскольку мы используем соглашение по умолчанию, то параметр передается через регистр **EAX**. Функция должна вернуть длинную строку, Что в действительности означает указатель на область памяти, содержащей нашу строку. Вы можете использовать переменную **Result** для обращения к этой области, но поскольку ее поведение аналогично **var**, то в этом случае **@Result** эквивалентно регистру **EDX** (второй параметр отдельной функции передается через регистр **EDX,** при использования соглашения **register**)! Подробности мы рассмотрим в главе 1.9. **EDX** не содержит самого указателя, но указатель на область памяти для этого указателя! Тем не менее, пока еще не распределена память для нашей длинной строки. Будем использовать функцию **NewAnsiString** из модуля system для размещения памяти в куче и установке длины строки. Функция **NewAnsiString** устанавливает длину новой строки, которая передается через регистр **EAX** и возвращает адрес этой строки в том же регистре. Если же мы не вызовем функцию **NewAnsiString** (или другую функцию или процедуру, которая выделит память в куче для нашей длинной строки), то переменная **Result** не будет содержать действительного указателя и мы можем получить ошибку доступа (access violation) если попытаемся использовать его.

```
function PlusMinusLine(L: Integer): AnsiString; register;
asm
  push EDI
  push ESI
  push EBX
  mov ESI,EDX {Указатель памяти на Result}
  mov EBX,EAX {EBX хранит длину параметра}
  call System.@NewAnsiString
  mov EDI,EAX {EDI используется для заполнения строки}
  mov [ESI],EDI
  mov ECX,EBX
  shr ECX,2 {обрабатываем по 4 байта за раз}
  test ECX,ECX
```

```
    jz @@remain
    mov EAX,'+-+-'
  @@loop:
    mov [EDI],EAX
    add EDI,4
    dec ECX
    jnz @@loop
  @@remain: {заполняем оставшие байты, если length/4 не ноль}
    mov ECX,EBX
    and ECX,3
    jz @@ending
    mov EAX,'+-+-'
  @@loop2:
    mov BYTE PTR [EDI],al
    shr EAX,8
    inc EDI
    dec ECX
    jnz @@loop2
  @@ending:
    mov EAX,ESI {для совместимости: возврат указателя через EAX}
    pop EBX
    pop ESI
    pop EDI
  end;
```

Для облегчения понимания, данного примера пришлось пожертвовать некоторой эффективностью. Для ускорения за раз одновременно обрабатывается по 4 байта для заполнения строки. Тем не менее, можно сделать еще быстрее, если использовать указатель в **EDI** на конец строки и использовать отрицательный счетчик в **ECX**, постепенно увеличивая его до нуля и используя его как индекс ([EDI+ECX*4]), что также сделало бы не нужным увеличение регистра **EDI** после каждой итерации. Это прекрасный повод для читателя переписать эту функцию данным образом и сравнить результаты выполнения. Также, может быть, вы захотите уменьшить количество циклов для еще большей эффективности. Например, обрабатывать по 8 байт за каждую итерацию, уменьшив этим количество переходов.

Как видим, возврат информации через регистры не всегда самый простой путь, особенно для структурных переменных типа длинных строк.

1.7. Возврат информации через стек процессора

Даже если на первый взгляд кажется странным возвращать результат через стек, в некоторых случаях это единственный путь для возврата результата. Например, если результат не помещается в регистр или не помещается на стек сопроцессора. Вы должны думать об коротких строках, записях и множествах, которые не помещаются в регистр. Но мы не говорим о таких типах данных, которые возвращаются, как указатель на результат и не говорим о результатах, передаваемые как **var** параметр. В предыдущей главе, мы уже обсуждали эти принципы. Тем не менее, когда размер результата не известен заранее, как для длинных строк до их создания, некоторые типы занимают фиксированное количество байт в памяти, и место для их размещения может быть выделено компилятором еще до вызова функции. Это то, что в действительности применимо для записей, статических массивов и больших множеств, также и для коротких строк.

Допустим, что у нас есть запись **TMyRecord**, объявленная следующим образом:

```
type
  TMyRecord = record
    A: Integer;
    B: Double;
    C: Integer;
```

```
end;
```

Компилятор знает, что эта запись занимает 16 байт в памяти. Поэтому, мы можем объявить функцию, которая возвращает запись как результат, объявление будет выглядеть так:

```
function MyFunction(I: Integer): TMyRecord; register;
```

Как отмечено в главе 1.6, переменная **Result** передается в функцию как дополнительный **var** параметр. Поэтому регистр **EDX** хранит указатель на результат. Но в отличие от примера с AnsiString, память для результата уже выделена компилятором до входа в функцию, если быть точным, то на стеке. Поэтому нам не нужно самостоятельно выделять память в куче. Достаточно заполнить эту память, которую компилятор резервировал для этой цели. Надо быть только острожным и не выйти за пределы отведенной памяти! Если же это произойдет, то будет разрушен стек и как результат - повисание программы или ошибка доступа (access violation).

В связи с тем, что память уже выделена компилятором, и регистр **EDX** содержит указатель на эту память (в действительности это стековая память), мы можем просто использовать регистр **EDX** для заполнения результата:

```
mov [EDX],EAX
```

Мы заполнили первое двойное слово (член записи **A**) содержимым регистра **EAX**. Заметим, что в данном случае мы не заботились о расчете смещения, относительно регистра **EDX,** для доступа к нужному члену записи. Но вы должны все-таки написать так, чтобы позволить компилятору сделать эту работу за вас:

```
mov [Result].A,EAX
```

Это то же самое, что и выше, но компилятор знает, что **Result** - это указатель, хранящийся в регистре **EDX,** и вы можете использовать более ясную точечную нотацию для адресации членов записи. Компилятор сам рассчитает смещение. Строго рекомендуется использовать именно точечную нотацию везде, где только возможно.

1.8. Возврат информации через стек сопроцессора

Функции, которые возвращают результат с плавающей запятой, просто должны возвращать результат в ST(0). Ниже вы найдете пример. Помните, что сопроцессор обрабатывает внутри все числа, как 10-байтные расширенные числа. Указывая формат результата (single, double, extended, comp, currency, и т.д.) мы только указываем, как число будет записано в ST(0) и затем в память. В примере функции `CalcRelatMass` показано это. Передаются два параметра, масса и скорость тела и производится расчет относительной массы, согласно теории относительности. Оба параметра: масса (*m) и скорость (v)*, передаются в функцию как double, результат так же возвращается как double.

```
function CalcRelatMass(m,v: Double): Double; register;
const
  LightVelocity: Integer = 299792500;
asm
  {Расчет относительной массы по следующей ффформуле:
  Result = m / Sqrt(1-v?/c?), где c = скорость света,
  m масса и v скорость движения объекта}

  fild LightVelocity
  fild LightVelocity
  fmulp {расчет c?}
  fld v
  fld v
  fmulp {расчет v?}
  fxch
  fdivp {v?/c?}
```

```
    fld1
    fxch
    fsubp {ST(0)=1-(v?/c?)}
    fsqrt {корень ST(0)}
    fld m
    fxch
    fdivp {деление массы на корень результата}
end;
```

Оба параметра, *m* и *v*, передаются в функцию через стек. Поскольку они типа double, то они занимают по 8 байт стека, который выглядит следующим образом:

```
EBP+0x10  m
EBP+0x08  v
EBP+0x04  адрес возврата
EBP ->    предыдущий EBP
```

Вы никогда не должны забывать, что внутри сопроцессора они обрабатываются как 10-байтные числа с плавающей запятой. Результат остается в ST(0) (верхушка стека математического сопроцессора). Это извлекается кодом, который вызвал функцию.

Вы можете изменять точность и режим округления вычислений путем изменения контрольного слова процессора. Хотя вы точно знаете, что делаете, но это не поощряется, поскольку смена контрольного слова влияет на все вычисления для всего вашего приложения. Проблему обостряет то, что некоторые DLL также изменяют контрольное слово. Это иногда может привести к непредсказуемым результатам или различным результатам в зависимости от Операционной Системы, на которой запускается программа или в зависимости от того, какие версии DLL реально используются. Как заметил Robert Lee в одном из сообщений в группе новостей, вы должны особенно избегать этого, путем загрузки контрольного слова из глобальной переменной **`Default8087CW`** (объявлена в модуле System) до выполнения важных процедур.

Так же очень важно полное понимание природы чисел с плавающей запятой при использовании их внутри вашего кода. Я написал отдельную статью, в которой обсуждается основы. Статья доступна на моих страницах по Дельфи на сайте http://www.optimalcode.com/Guido/fpv.html.

1.9. Передача параметров по зничению и ссылке

Имеется огромная разница между передачей параметров по значению и по ссылке (через директиву **`var`**). Например, следующее объявление функции:

```
function MyFunction(I: Integer): Integer; register;
```

Значение параметра **`I`** будет передано через регистр **`EAX`** (см. таблицу 2 для обзора, как параметры разного типа передаются в функцию/процедуру). Например, когда I=254, **`EAX`** подобен $000000FE. Но следующее объявление:

```
function MyFunction(var I: Integer): Integer; register;
```

передаст не значение I (254 в нашем примере), а указатель на местонахождение, где переменная I записана в памяти (например, $0066F8BC) и этот указатель будет помещен в регистр **`EAX`**! При передаче параметра с помощью ключевого слова **`var`**, вы всегда передаете 32-битный указатель на переменную (который естественно помещается в регистр соглашения **register**).

Посмотрим на простой пример: допустим, мы желаем, чтобы наша функция вернула сумму целочисленного числа и 12 (Конечно, это очень бессмысленный пример, он нужен просто для демонстрации), передадим параметр по значению (функция вернет результат в регистре **`EAX`**):

```
function MyFunction(I: Integer): Integer; register;
```

```
asm
  add EAX,12
end;
```

В случае же передачи по ссылке, мы должны написать так:

```
function MyFunction(var I: Integer): Integer; register;
asm
  mov EAX,[EAX] {Загрузить значение параметра I через указатель}
  add EAX,12
end;
```

При использовании директивы **const** правила те же, как для переменных, передаваемых по значению. Например, для объявления:

```
function MyFunction(const I: Integer): Integer; register;
```

Регистр **EAX** будет содержать значение I, а не указатель.

Как мы обсуждали в главе 1.6, длинные строки, динамические массивы, варианты, большие множества и записи возвращаются с помощью дополнительного **var** параметра. Позже, в других главах, мы обсудим эти типы более детально.

Глава 2: Замечания о синтаксисе

В этой главе, мы рассмотрим синтаксические требования, которые вам требуется знать. Если вы использовали отдельный ассемблер, то вы заметили, что встроенный ассемблер в Дельфи 1-5 поддерживает только относительно небольшой набор возможностей языка. Эта ситуация была улучшена с выходом Дельфи 6, теперь компилятор распознает набор MMX, SIMD и SSE инструкций (так же и Enhanced 3D для AMD CPU, но данная статья сфокусирована только на Intel, и мы не будем обсуждать это в дальнейшем). С другой стороны, это также дает возможность использовать некоторые OP конструкции внутри ассемблерного кода.

2.1. Инструкции и команды

Ваш код на ассемблере состоит из нескольких выражений. Каждая инструкция состоит как минимум из одной команды. В большинстве случаев, вам потребуется использовать от одного до нескольких операндов. Операнды разделяются символом запятой. Также в инструкции могут использоваться префиксы (например, *rep* или *lock*). Наконец, инструкция может включать метку (смотрите ниже рассуждения о метках).

Примеры допустимых инструкций:

```
    cdq {только команда}
    bswap EAX {команда и один операнд}
    mov EAX,[ESI] {команда и два операнда}
    imul EAX,ECX,16 {команда и триа операнда}
    rep movsd {префикс и коаднда }
@@Start: rep stosd {локальная метка, префикс и команда }
```

Разрешено помещать несколько инструкций в одной строке, разделяя их точкой с запятой, Но я настоятельно не рекомендую так делать. Это сильно снижает читабельность вашей программы, и не добавляет при этом никакой эффективности, повышения скорости или каких-либо других преимуществ. При использовании по одной инструкции в строке не требуется ставить точку с запятой в конце строки (как это требуется для обычного Паскаль кода).

Комментарии могут быть добавлены в конце строки, но не могут размещаться внутри инструкции.

2.2. Набор команд

Встроенный ассемблер Дельфи 2-5 поддерживает только подмножество команд процессора Intel 80486 (документация по Дельфи 3 вообще утверждает, что только 80386, но дополнительные инструкции процессора 80486, например **bswap, xadd, cmpxchg, fstsw ax,** и другие в действительности распознаются и обрабатываются корректно). Тем не менее, специфические команды Pentium, например **cpuid** или условные перемещения из Pentium Pro, PII и PIII, не распознаются встроенным ассемблером в этих версиях. В Дельфи 6, поддержан полный набор команд от Pentium I до IV. Включая специальные расширения MMX, SSE и другие. Это действительно серьезное улучшение, поскольку в более ранних версиях приходилось их кодировать вручную с помощью инструкций **db** (см. ниже). Это было довольно неприятно, так как эти инструкции особо интересны для специальных случаев.

Если вы желали использовать эти инструкции в Д2-Д5, то должны были вставлять их вручную с помощью серии инструкций **db**. Ясно, что вы не только должны были быть очень осторожны при вставке их в код, избегая ошибок, но и также особо комментировать эти строки. Со следующей ссылки вы можете загрузить .pas, который содержит исходный текст класса **TCPUID,** в котором интенсивно используется ассемблер, и в котором инструкция **cpuid** закодирована с помощью инструкций **db**. Нажмите здесь для загрузки cpuinfo.pas с сайта автора или с текущего каталога в формате cpuinfo.zip.

Вы должны проштудировать исходный код **cpuinfo.pas**, обратив особое внимание на функцию **GetCPUIDResult,** которая написана полностью на basm. Программа вызывает **cpuid** для различных уровней ID, которые поддержаны и заполняет запись типа **TCPUIDResult** полученной информацией. Данный тип записи используется в методах класса **TCPUID**. Заметим, что все поля записи **TCPUIDResult** адресуются через их имена, вместо расчета смещения. Компилятор сам рассчитывает смещение, так что если структура записи будет изменена, то код будет продолжать работать корректно.

Заметим, что команда **cpuid** уничтожает содержимое всех нормальных регистров, так что требуется особая осторожность при работе с ними. При этом так же сбрасываются все конвейеры, и ожидается окончание работы всех оставшихся инструкций, поэтому вы не должны использовать это в критических ситуациях. После выполнения инструкции **cpuid**, все нормальные регистры, включая **EAX** и другие, будут изменены.

Полное описание набора команд процессоров можно найти на сайте фирмы Intel http://developer.intel.com. Как я заметил во введении, данная статья посвящена только процессорам фирмы Intel не только, поскольку они самые распространенные, но и потому что они являются стандартом де-факто для набора команд процессора и сопроцессора. Некоторые другие производители имеют в составе своих процессоров дополнительные команды, но поскольку они присутствуют только в их процессорах, вы не должны пытаться их использовать, чтобы ваши приложения могли работать на более широком спектре систем. Другим решением является иметь различные варианты критичных по времени кусков кода, оптимизированные для различных процессоров. В начале вашей программы вы должны проверить тип процессора и установить глобальный флаг, который будет указывать, какую версию использовать.

Аналогичный вопрос: какой минимальный набор инструкций должен быть в вашей программе, что бы она могла работать на 80486 или более ранних процессорах. Конечно, 80486 и более старые процессоры уже устарели и, как минимум, стоит ориентировать вашу программу на Intel Pentium Plain или выше. Тем не менее, если выбрать более новую модель, как базис, например Pentium II и выше, то вы игнорируете многие компьютеры с Pentium Plain и Pentium MMX, которые еще в ходу. Если вы выберите минимум как Pentium II, то вы сможете получить преимущества от дополнительных инструкций, таких как условные перемещения. Так же, в данном случае проще написать кусок кода, который будет поддерживать все платформы. Если вы решили включить поддержку Pentium Plain и MMX CPU, чтобы быть более осведомленным в других вещах, таких как парность команд, различные особенности по предсказанию переходов и т.д. Все это можно изучить в деталях в превосходном руководстве от Agner Fog на сайте http://www.agner.org/assem/, но давайте начнем, ниже несколько основных правил.

2.2.1. Не используйте комплексные команды

В большинстве случаев, комплексные строковые инструкции очень медленны и должны быть заменены оптимизированным циклом с простыми инструкциями. Без префикса **rep**, комплексные инструкции вообще за пределами вопроса. Только при определенных специфических условиях инструкции **rep movsd** и **rep stosd** могут быть быстрее, но только при условии, что оба адреса, приемник и источник, выровнены на границу восьми байт и при этом не должно быть конфликтов в кэше, и в свете того, что в данный момент Дельфи не дает возможности управлять выравниванием, вы не должны их использовать.

2.2.2. Используйте 32-битные алгоритмы, везде, где только возможно

Если только невозможно иначе, то вы не должны использовать инструкции, которые оперируют словами, более правильно использовать те, которые работают с переменными типа двойное слово. Байтовый доступ еще может иногда использоваться, но остерегайтесь использовать операции со словами. Ниже пример использования 32-битного алгоритма для

поиска символа в строке. Идея основана на генерации уникального значения, если и только если символ найден. Поскольку пример обрабатывает строку по четыре байта за раз, то это значительно быстрее, чем обработка по одному байту за раз, несмотря на дополнительное усложнение, поскольку требуется обрабатывать сразу четыре байта.

```
function ScanStrForChar(C: Char; S: String): Integer; register;
asm
   push EBX
   push EDI
   push ESI
   test EDX,EDX
   jz @notfound
   mov ESI,[EDX-4]
   test ESI,ESI
   jz @notfound
   add ESI,EDX
   mov ah,al
   mov di,ax
   shl EDI,16
   or di,ax
   mov EAX,EDX
   lea EDX,[EAX+3]
@L1:
   cmp EAX,ESI
   ja @notfound
   mov EBX,[EAX]
   xor EBX,EDI
   add EAX,4
   lea ECX,[EBX-$01010101]
   not EBX
   and ECX,EBX
   and ECX,$80808080
   jz @L1
   mov EBX,ECX
   shr EBX,16
   test ECX,$8080
   jnz @L2
   mov ECX,EBX
@L2:
   lea EBX,[EAX+2]
   jnz @L3
   mov EAX,EBX
@L3:
   shl cl,1
   sbb EAX,EDX
   inc EAX
@ending:
   pop ESI
   pop EDI
   pop EBX
   ret
@notfound:
   xor EAX,EAX
   jmp @ending
end;
```

В зависимости от длины обрабатываемых строк, функция может быть быстрее стандартной функции **pos** раза в два. Заметим, что должна прочитать от одного до трех символов в конце строки, но в текущей версии компилятор всегда размещает строки по модулю четыре, так что, это не приносит проблем, но нельзя гарантировать, что в следующих версиях это будет так же. Вы можете устранить эту проблему, путем расчета остатка символов в конце строки по модулю четыре и обработать остаток побайтно. Вы потеряете некоторое быстродействие,

но выиграете в надежности. Заметим, что компилятор добавляет одну или несколько инструкций ret после **jmp @ending**. Но они никогда не будут выполнены, поскольку мы включили инструкцию **ret** сами.

Вы должны избегать подобных вещей, поскольку если нужен фрейм стека, то вы должны будете сами написать код выхода (см. главу 1.2, где рассмотрены коды входа и выхода). В вышеприведенном примере нет фрейма стека, так что нет нужды и в коде выхода. Вы можете избежать этой проблемы, путем добавления инструкции **jmp** для условия, когда символ не найден, это просто пропустит установку результата в ноль, если строка пустая или символ не найден. После этого пример будет выглядеть так:

```
    ...
    shl cl,1
    sbb EAX,EDX
    inc EAX
    jmp @ending
@notfound:
    xor EAX,EAX
@ending:
    pop ESI
    pop EDI
    pop EBX
end;
```

Но, в этом случае вы будете вынуждены всегда добавлять дополнительную инструкцию **jmp** в ваш алгоритм, который немного от этого замедлится. Если вы обрабатывает достаточно длинные строки, то это почти незаметно на общем процессе обработке, и добавленный код может быть субъектом для оптимизации в дальнейшем, когда это станет важным.

2.2.3. Избегайте деления

Деление, как правило, более медленное - заменяйте его сдвигами или умножением с соответствующим значением.

2.2.4. Замечания по особым инструкциям

Битовые инструкции (**bt, btc, bts, btr**) должны по возможности заменяться на инструкции **and, or, xor** и **test,** когда приоритетом является скорость.

Избегайте инструкции **wait**. На старых процессорах инструкция **wait** была нужна для синхронизации доступа к памяти и уверенности, что сопроцессор был готов к выполнению операции. На процессорах Pentium это абсолютно лишнее. Единственная причина использования инструкции **wait** это отлов исключения из предыдущей инструкции. Сейчас большинство инструкций сопроцессора, отлавливают исключение без инструкции **wait** (исключая **fnclex** и **fninit**), вы можете опускать инструкцию **wait** в большинстве случаев. Если бит исключения устанавливается, то следующая инструкция с плавающей запятой отлавливает это. Если вы хотите быть уверенным, что любые, необслуженные исключения, были обработаны до окончания процедуры, то вы можете добавить инструкцию **wait** после критического по времени куска, что обработает все необслуженные исключения.

2.3. Метки

Имеется два типа меток, которые вы можете использовать в BASM: обычные Паскаль метки и локальные метки. Обычные метки требуется объявлять в секции **label**. Вторые начинаются с символа @. Поскольку символ @ не может быть частью идентификатора Object Pascal, то вы можете использовать локальные метки только внутри блока **asm...end**. Иногда, вы видите метки с двумя символами @. Этому есть объяснение. Я использую это для привлечения внимания, но это не требуется (некоторые ассемблеры используют @@ для идентификации особого класса меток, например **@@:** для анонимных меток). Метки автоматически

объявляются написанием идентификатора и добавлением символа двоеточия в конце (**@loop:** или **MyLoopStartsHere:**). Для ссылки на такую метку, просто используйте идентификатор в выражении (естественно без двоеточия). Пример, как использовать локальную метку для организации цикла:

```
    mov ECX,{Counter}
@loop:
    ... {команды цикла}
    dec ECX
jnz @loop
```

Для того, что бы превратить локальную метку в обычную, сделайте так:

```
label
    MyLoop;
asm
    ...
    mov ECX,{Counter}
Myloop:
    ... {команды цикла}
    dec ECX
    jnz MyLoop
    ...
end;
```

Ни один из этих типов не лучше, чем другой. Нет никаких преимуществ ни в скорости, ни в размере кода, поскольку метки - это только указатель для компилятора для расчета смещений и переходов. Различие между нормальными и локальными метками больше в стиле программирования в Паскале. В действительности, даже нормальные метки, по сути, являются локальными, поскольку вы не можете перейти на метку за пределами текущей функции или процедуры. Все бы хорошо, если бы одно “но”. Инструкция **jmp** в ассемблере более применима, чем команда **Goto** в Паскале (Если вы используете команду **Goto** без нужды в Паскале, то это является признаком плохого программирования. Всегда можно спроектировать ваш код без использования **Goto**), но также всегда вы должны искать пути для реализации алгоритма с минимальным использованием инструкций перехода (что, важно с точки зрения производительности).

2.4. Определение данных и констант

(*Данная глава еще находится в стадии разработки...*) Встроенный ассемблер поддерживает три директивы объявления данных, **db-dw-dd** (четыре в Дельфи 6: **dq** "четыре слова"), но в основном вы используете секции объявления **var** и **const** в вашем коде. Директивы определения данных могут использоваться только внутри **asm**…**end** блока, для генерации последовательностей байтов (**db**), слов (**dw**) и двойных слов (**dd**) соответственно (или четырех слов, **dq**, только Дельфи 6). Эти данные записываются в кодовый сегмент, и вы должны их изолировать с помощью инструкции перехода **jmp** от остального кода. Все это немного излишне, но вы можете использовать **db, dw** и **dd** для генерации инструкций, процессора, которые basm не поддерживает, например условные пересылки или MMX инструкции в Дельфи версий 2-5. В главе 2.2 я дал пример использования их для генерации кода инструкции **cpuid**. Директивы определения не могут использоваться для определения типов данных, так как в трансляторах masm или tasm. Для этого вы должны использовать обычные команды Паскаля.

2.5.Инструкции и приведение типов

(Данная глава в стадии разработки...)

Примеры

В данной главе, мы приведем несколько примеров на basm. Это только первая часть моих статей по Дельфи и встроенному ассемблеру, которая опубликована на данном сайте. Расположено это на странице featured articles, и называется Considerations for writing and using Intel assembly code in Delphi projects (Последнее изменение 1 сентября 2001). Мы будем признательны за замечания или советы по этим страницам.

Есть замечания насчет этих примеров? Пожалуйста, посылайте их по адресу в главе Как связаться с Гуйдо Гайбелсом!

Прямой доступ к портам в Windows 95 и 98

```
function PortInByte(PortAddress: Word): Byte;
asm
  mov dx,ax
  in al,dx
end;

procedure PortOutByte(PortAddress: Word; Data: Byte);
asm
  xchg dx,ax
  out dx,al
end;
```

Подсчет количества установленных бит в integer

```
function CountBits(const Value: Integer): Integer;
asm
  mov ECX,EAX
  xor EAX,EAX
  test ECX,ECX
  jz @@ending
@@counting:
  shr ECX,1
  adc EAX,0
  test ECX,ECX
  jnz @@counting
@@ending:
end;
```

Проверка установки отдельного бита (0-31)

```
function IsBit(Value, Pos: Integer): Boolean;
asm
  mov ECX,EAX
  xor EAX,EAX
  and EDX,31
  bt ECX,EDX
  adc EAX,0
end;
```

Установка отдельного бита (0-31) в единицу

```
function SetBit(const Value, Pos: Integer): Integer;
asm
  and    EDX,31
  bts    EAX,EDX
end;
```

Сброс отдельного бита

```
function ClearBit(const Value, Pos: Integer): Integer;
asm
  and    EDX,31
  btr    EAX,EDX
end;
```

Извлечение битовой маски из integer

```
function ExtractBits(const Value, Start, Count: Integer): Integer;
const
  Mask: array[0..31] of Integer =
    ($01,$03,$07,$0F,$1F,$3F,$7F,$FF,
    $01FF,$03FF,$07FF,$0FFF,$1FFF,$3FFF,$7FFF,$FFFF,
    $01FFFF,$03FFFF,$07FFFF,$0FFFFF,
    $1FFFFF,$3FFFFF,$7FFFFF,$FFFFFF,
    $01FFFFFF,$03FFFFFF,$07FFFFFF,$0FFFFFFF,
    $1FFFFFFF,$3FFFFFFF,$7FFFFFFF,$FFFFFFFF);
asm
  xchg ECX,EDX
  test EDX,EDX
  jnz @@isoke
  xor EAX,EAX
  jmp @@ending
@@isoke:
  dec EDX
  and EDX,31
  shr EAX,cl
  and EAX,dword ptr [Mask+EDX*4]
@@ending:
end;
```

Модуль CpuInfo

Здесь приведена часть проекта, модуль CpuInfo. Полностью проект находится на сайте Гуйдо Гайбелса http://www.optimalcode.com/Guido/cpuinfo.html и в виде архива cpuinfo.zip вместе с этой книгой.

```
unit cpuinfo;
  {Author: Guido GYBELS, april 2001, All rights reserved.
   This unit offers some types and a global instance that
   uses the features of the CPUID instruction as it is
   implemented on modern Intel processors.
   By using the properties of the global CPUID object,
   application builders can quickly evaluate the features
   of the CPU their program is running on. This allows to
   optimise routines for specific CPU's.
   REVISION HISTORY:
   - april 2001, First version}
interface

uses Windows, Sysutils;

type
  {The TCPUIDResult record type contains fields that
   stores the values returned by the various levels of
   the CPUID instruction.}
  TCPUIDResult = packed record
    IsValid: ByteBool;
    HighestLevel: dWord;
    GenuineIntel: ByteBool;
    VendorID: packed array[0..11] of Char;
    ProcessorSignature: dWord;
    MiscInfo: dWord;
    FeatureFlags: packed array[0..1] of dWord;
    Stepping: Byte;
    Model: Byte;
    Family: Byte;
    ProcessorType: Byte;
    ExtendedModel: Byte;
    ExtendedFamily: Byte;
    FPUPresent: ByteBool;
    TimeStampCounter: ByteBool;
    CX8Supported: ByteBool;
    FastSystemCallSupported: ByteBool;
    CMOVSupported: ByteBool;
    FCOMISupported: ByteBool;
    MMXSupported: ByteBool;
    SSESupported: ByteBool;
    SSE2Supported: ByteBool;
    SerialNumberEnabled: ByteBool;
    CacheDescriptors: packed array[0..47] of Byte;
    SerialNumber: packed array[0..1] of dWord;
  end;
  TCPUType = (ctOriginal, ctOverDrive, ctDualProcessor, ctUnknown);
  TCPUFamily = (cfUnknown, cf486, cfPentium, cfPentiumPro, cfPentium4);
  TCPUFeature = (ftFPU, ftTSC, ftCX8, ftFSC, ftCMOV, ftFCOMI, ftMMX, ftSSE,
    ftSSE2, ftSerialNumber);
  TCacheSize = (caSizeUnknown, caNoCache, Ca128K, Ca256K, ca512K, ca1M, ca2M);
  TCPUBrandID = (brUnsupported, brCeleron, brP3, brP3Xeon, brP4);
    {The TCPUID class is the class for the global CPUID instance
    that is created by this unit and that offers several properties
    usefull in identifying the CPU your application is running on.
    Application builders should use the global CPUID object since
    there is no need to create new, additional, instances of this
```

```
    class (because they would simply return an identical object).
    All properties are read-only since it is impossible to change
    the CPU characteristics.}
  TCPUID = class
  private
    FCPUIDResult: TCPUIDResult;
    function GetBooleanField(Index: Integer): Boolean;
    function GetCPUBrand: TCPUBrandID;
    function GetCPUFamily: TCPUFamily;
    function GetCPUType: TCPUType;
    function GetFeature(Index: TCPUFeature): Boolean;
    function GetIntegerField(Index: Integer): Integer;
    function GetLevel2Cache: TCacheSize;
    function GetProcessor: String;
    function GetSerialNumber: String;
    function GetVendorID: String;
  public
    constructor Create;
    property BrandID: TCPUBrandID read GetCPUBrand;
    property CanIdentify: Boolean index 0 read GetBooleanField;
    property CPUFamily: TCPUFamily read GetCPUFamily;
    property CPUID: TCPUIDResult read FCPUIDResult;
    property CPUType: TCPUType read GetCPUType;
    property Family: Integer index 3 read GetIntegerField;
    property Features[Index: TCPUFeature]: Boolean read GetFeature;
    property GenuineIntel: Boolean index 1 read GetBooleanField;
    property HighestIDLevel: Integer index 0 read GetIntegerField;
    property CacheSize: TCacheSize read GetLevel2Cache;
    property Model: Integer index 2 read GetIntegerField;
    property Processor: String read GetProcessor;
    property SerialNumber: String read GetSerialNumber;
    property Stepping: Integer index 1 read GetIntegerField;
    property VendorID: String read GetVendorID;
  end;

var
  CPUID: TCPUID;

implementation

const
  SizeOfTCPUIDResult = SizeOf(TCPUIDResult);

  {GetCPUIDResult is a basm routine that performs the actual
  CPUID calls and stores the results in a record of type
  TCPUIDResult. If the CPUID instruction is supported by the
  processor, this routine will call it for every value of
  eax allowed for that processor, making one call for each
  value and storing the results in the record. Only for eax=2
  is it possible that multiple calls are performed in order
  to get to all the cache descriptors.
  More information about the CPUID function can be found in
  the Intel Application Note AP-485.}

function GetCPUIDResult: TCPUIDResult;
var
  Counter: Byte;
asm
  push ebx  {changes all general registers...}
  push edi  {...so, make sure we save what needs to be preserved}
  push esi
  mov edi,eax {pointer to result in edi}
  mov ecx,SizeOfTCPUIDResult
  mov esi,edi {zero the entire record out}
```

```
  add esi,ecx
  neg ecx
@loop:
  mov BYTE PTR [esi+ecx],0
  inc ecx
  jnz @loop
  pushfd {test if bit 21 of extended flags}
  pop eax {can toggle. If yes, then cpuid is supported}
  mov ebx,eax
  xor eax,1 shl 21
  push eax
  popfd
  pushfd
  pop eax
  xor eax,ebx
  and eax,1 shl 21 {Only if bit 21 can toggle...}
  setnz TCPUIDResult(edi).IsValid {...are the results valid}
  jz @ending {don't continue if cpuid is not supported}
  xor eax,eax {eax=0: get highest value and Vendor ID}
  db $0f,$a2 {cpuid}
  mov DWORD PTR TCPUIDResult(edi).VendorID,ebx
  mov DWORD PTR TCPUIDResult(edi).VendorID+4,edx
  mov DWORD PTR TCPUIDResult(edi).VendorID+8,ecx
  xor ebx,$756e6547 {Check if Vendor is Intel...}
  xor edx,$49656e69
  xor ecx,$6c65746e
  or ebx,edx
  or ebx,ecx
  test ebx,ebx
  setz TCPUIDResult(edi).GenuineIntel {...and set boolean accordingly}
  mov TCPUIDResult(edi).HighestLevel,eax {also save highest level...}
  cmp eax,0
  setnz TCPUIDResult(edi).IsValid {...and if it is 0, results not valid}
  jz @ending {if level 1 is not supported, bail out}
  mov eax,1
  db $0f,$a2 {cpuid} {else get processor signature and feature flags}
  mov TCPUIDResult(edi).ProcessorSignature,eax
  mov TCPUIDResult(edi).MiscInfo,ebx
  mov DWORD PTR TCPUIDResult(edi).FeatureFlags,ecx
  mov DWORD PTR TCPUIDResult(edi).FeatureFlags+4,edx
  mov ebx,eax {Then isolate the various items from...}
  and al,$0f {...the processor signature into their fields}
  mov TCPUIDResult(edi).Stepping,al
  mov eax,ebx
  shr eax,4
  and al,$0f
  mov TCPUIDResult(edi).Model,al
  mov eax,ebx
  shr eax,8
  and al,$0f
  mov TCPUIDResult(edi).Family,al
  mov eax,ebx
  shr eax,12
  and al,$03
  mov TCPUIDResult(edi).ProcessorType,al
  mov eax,ebx
  shr eax,16
  and al,$0f
  mov TCPUIDResult(edi).ExtendedModel,al
  mov eax,ebx
  shr eax,20
  mov TCPUIDResult(edi).ExtendedFamily,al
  test edx,1 {Next, check various feature bits and set their...}
  setnz TCPUIDResult(edi).FPUPresent {...respective flags in the record}
```

```
  test edx,1 shl 4
  setnz TCPUIDResult(edi).TimeStampCounter
  test edx,1 shl 8
  setnz TCPUIDResult(edi).CX8Supported
  test edx,1 shl 11
  setnz TCPUIDResult(edi).FastSystemCallSupported
  test edx,1 shl 15
  setnz TCPUIDResult(edi).CMOVSupported
  mov eax,edx
  and eax,(1 shl 15) or 1
  cmp eax,(1 shl 15) or 1
  setz TCPUIDResult(edi).FCOMISupported
  test edx,1 shl 18
  setnz TCPUIDResult(edi).SerialNumberEnabled
  test edx,1 shl 23
  setnz TCPUIDResult(edi).MMXSupported
  test edx,1 shl 25
  setnz TCPUIDResult(edi).SSESupported
  test edx,1 shl 26
  setnz TCPUIDResult(edi).SSE2Supported
  cmp TCPUIDResult(edi).HighestLevel,2 {If eax=2 is not supported...}
  jl @ending {...then bail out}
  lea esi,TCPUIDResult(edi).CacheDescriptors
  mov eax,2 {else get the cache descriptors}
  db $0f,$a2 {cpuid}
  and al,3 {first time, al will hold a counter...}
  mov [Counter],al {...that tells us how often we should...}
  xor al,al {...call with eax=2 to get all descriptors...}
@nextdescriptor:
  test eax,1 shl 31 {If bit 31 is set, then no valid descriptors...}
  jnz @invalidA {...so skip this register}
  mov DWORD PTR [esi],eax
@invalidA:
  test ebx,1 shl 31
  jnz @invalidB
  mov DWORD PTR [esi+4],ebx
@invalidB:
  test ecx,1 shl 31
  jnz @invalidC
  mov DWORD PTR [esi+8],ecx
@invalidC:
  test edx,1 shl 31
  jnz @invalidD
  mov DWORD PTR [esi+12],edx
@invalidD:
  add esi,16
  dec [Counter] {...see if there are more descriptors...}
  jz @descriptorsfull {...if not, continue with next step}
  db $0f,$a2 {cpuid} {...else, get next serie of descriptors}
  jmp @nextdescriptor
@descriptorsfull:
  cmp TCPUIDResult(edi).HighestLevel,3 {see if serial number...}
  jl @ending {...is supported. If not, bail out.}
  mov eax,3 {else get lower 64 bits of serial number...}
  db $0f,$a2 {cpuid} {upper 32 bits = processor signature}
  mov DWORD PTR TCPUIDResult(edi).SerialNumber,ecx {...and store them}
  mov DWORD PTR TCPUIDResult(edi).SerialNumber+4,edx
@ending:
  pop esi
  pop edi
  pop ebx
end;

{TCPUID}
```

```
resourcestring
  rsUnknownVendor = 'UnknownVendor';

constructor TCPUID.Create;
begin
  inherited;
  FCPUIDResult:=GetCPUIDResult;
end;

function TCPUID.GetBooleanField(Index: Integer): Boolean;
begin
  case Index of
    0: {CanIdentify} Result:=FCPUIDResult.IsValid;
    1: {GenuineIntel} Result:=FCPUIDResult.GenuineIntel;
  else
    Result:=False;
  end;
end;

function TCPUID.GetIntegerField(Index: Integer): Integer;
begin
  case Index of
     0: {HighestLevel} Result:=FCPUIDResult.HighestLevel;
     1: {Stepping} Result:=FCPUIDResult.Stepping;
     2: {Model} if FCPUIDResult.Model=15 then
          Result:=FCPUIDResult.ExtendedModel
        else Result:=FCPUIDResult.Model;
     3: {Family} if FCPUIDResult.Family=15 then
          Result:=15+FCPUIDResult.ExtendedFamily
        else Result:=FCPUIDResult.Family;
  else
     Result:=0;
  end;
end;

function TCPUID.GetVendorID: String;
begin
  if CanIdentify then Result:=FCPUIDResult.VendorID
  else Result:=rsUnknownVendor;
end;

function TCPUID.GetCPUType: TCPUType;
begin
  case FCPUIDResult.Processortype of
    1: Result:=ctOverdrive;
    2: Result:=ctDualProcessor;
    3: Result:=ctUnknown;
  else
    Result:=ctOriginal;
  end;
end;

function TCPUID.GetCPUFamily: TCPUFamily;
begin
  case FCPUIDResult.Family of
    4: Result:=cf486;
    5: Result:=cfPentium;
    6: Result:=cfPentiumPro;
    15: case FCPUIDResult.ExtendedFamily of
          0: Result:=cfPentium4;
        else
          Result:=cfUnknown;
        end;
  else
```

```
    Result:=cfUnknown;
  end;
end;

function TCPUID.GetFeature(Index: TCPUFeature): Boolean;
begin
  case Index of
    ftFPU: Result:=FCPUIDResult.FPUPresent;
    ftTSC: Result:=FCPUIDResult.TimeStampCounter;
    ftCX8: Result:=FCPUIDResult.CX8Supported;
    ftFSC: Result:=FCPUIDResult.FastSystemCallSupported;
    ftCMOV: Result:=FCPUIDResult.CMOVSupported;
    ftFCOMI: Result:=FCPUIDResult.FCOMISupported;
    ftMMX: Result:=FCPUIDResult.MMXSupported;
    ftSSE: Result:=FCPUIDResult.SSESupported;
    ftSSE2: Result:=FCPUIDResult.SSE2Supported;
    ftSerialNumber: Result:=FCPUIDResult.SerialNumberEnabled;
  else
    Result:=False;
  end;
end;

function TCPUID.GetProcessor: String;
begin
  if GenuineIntel then Result:='Intel ' else Result:='';
  case CPUFamily of
    cf486: case Model of
      0..1: Result:=Result+'80486DX';
      2: Result:=Result+'80486SX';
      3: Result:=Result+'80486DX2';
      4: Result:=Result+'80486SL';
      5: Result:=Result+'80486SX2';
      7: Result:=Result+'80486DX2/WB-Enh';
      8: Result:=Result+'80486DX4';
    else
      Result:=Result+'80486';
    end;
    cfPentium: if Features[ftMMX] then Result:=Result+'Pentium MMX' else
    Result:=Result+'Pentium';
    cfPentiumPro: case Model of
      1: Result:=Result+'Pentium Pro';
      3..4: Result:=Result+'Pentium II, Model '+IntToStr(Model);
      5: case CacheSize of
          caNoCache: Result:='Celeron, Model 5';
          ca1M, ca2M: Result:='Pentium II Xeon, Model 5';
         else
          Result:=Result+'Pentium II, Model 5';
         end;
      6: Result:=Result+'Celeron, Model 6';
      7: case CacheSize of
          ca1M, ca2M: Result:=Result+'Pentium III Xeon, Model 7';
         else
          Result:=Result+'Pentium III, Model 7';
         end;
      8: case BrandID of
          brCeleron: Result:=Result+'Celeron, Model 8';
          brP3Xeon: Result:=Result+'Pentium III Xeon, Model 8';
         else
          Result:=Result+'Pentium III, Model 8';
         end;
      9: Result:=Result+'Pentium III Xeon, Model A';
    else
      Result:=Result+'Pentium Pro';
    end;
```

```
    cfPentium4: Result:=Result+'Pentium 4';
  else
    Result:=Result+'Unknown CPU';
  end;
  case CPUType of
    ctOverDrive: Result:=Result+' OverDrive';
    ctDualProcessor: Result:=Result+' Dual CPU';
  end;
  Result:=Result+' (Stepping '+IntToStr(Stepping)+')';
end;

function TCPUID.GetLevel2Cache: TCacheSize;
var
  I,M,S: Integer;
  F: Boolean;
begin
  Result:=caSizeUnknown;
  M:=0;
  S:=0;
  F:=False;
  for I:=Low(FCPUIDResult.CacheDescriptors) to
    High(FCPUIDResult.CacheDescriptors) do begin
    if FCPUIDResult.CacheDescriptors[I]<>0 then F:=True;
    case FCPUIDResult.CacheDescriptors[I] of
      $40: begin
             M:=0;
             break;
           end;
      $41,$79: S:=128;
      $42,$7a,$82: S:=256;
      $43,$7b: S:=512;
      $44,$7c,$84: S:=1024;
      $45,$85: S:=2048;
    end;
    if S>M then M:=S;
  end;
  if F then case M of
    0: Result:=caNoCache;
    128: Result:=ca128K;
    256: Result:=ca256K;
    512: Result:=ca512K;
    1024: Result:=ca1M;
    2048: Result:=ca2M;
  end;
end;

function GetNibbleGroup(I: Integer): String;
var
  T: Integer;
begin
  T:=(I and $FFFF0000) shr 16;
  Result:=IntToHex(T,4);
  T:=(I and $FFFF);
  Result:=Result+'-'+IntToHex(T,4);
end;

function TCPUID.GetSerialNumber: String;
begin
  if Features[ftSerialNumber] then begin
    Result:=GetNibbleGroup(FCPUIDResult.ProcessorSignature);
    Result:=Result+'-'+GetNibbleGroup(FCPUIDResult.SerialNumber[1]);
    Result:=Result+'-'+GetNibbleGroup(FCPUIDResult.SerialNumber[0]);
  end else Result:='';
end;
```

```
function TCPUID.GetCPUBrand: TCPUBrandID;
var
  I: Integer;
begin
  if (Family>6) or ((Family=6) and (Model>7)) then begin
    I:=FCPUIDResult.MiscInfo and 255;
    case I of
      1: Result:=brCeleron;
      2: Result:=brP3;
      3: Result:=brP3Xeon;
      8: Result:=brP4;
    else
      Result:=brUnsupported;
    end;
  end else Result:=brUnsupported;
end;

initialization
  CPUID:=TCPUID.Create;
finalization
  CPUID.Free;
end.
```

Таблица 1: Использование регистров процессора

В данной таблице приведены сведения по использованию регистров процессора в 32-битных приложениях Дельфи. В первой колонке - список регистров. Во второй колонке - что содержится в регистре в секции входа в процедуру, а в третьей - что при выходе. В четвертой колонке - возможность использования регистра в коде и в последней колонке - необходимость сохранения регистра (сохранять при входе и восстанавливать при выходе).

Регистр	Код входа	Код выхода	Можно ли использовать?	Нужно ли сохранять
EAX	Self (1), Первый параметр (2) или не определен (3)	Результат функции (4)	Да	Нет
EBX	Неизвестно	Не используется	Да	Да
ECX	Второй параметр (1), третий параметр (2) или не определен (3)	Не используется	Да	Нет
EDX	Первый параметр (1), второй параметр (2) или не определен (3)	Для Int64 старшее двойное слово результата, или не используется	Да	Нет
ESI	Не определен	Не используется	Да	Да
EDI	Не определен	Не используется	Да	Да
EBP	Указатель фрейма стека	Указатель фрейма стека	Да	Да
ESP	Указатель стека	Указатель стека	Да	n/a
cs	Кодовый сегмент (5)	Не используется	Нет	Да
ds	Сегмент модели памяти (5)	Не используется	Нет	Да
es	Сегмент модели памяти (5)	Не используется	Нет	Да
fs	Резервировано для Windows	Резервировано для Windows	Нет	Да
gs	Резервировано	Резервировано	Нет	Да
ss	Сегмент стека (5)	Не используется	Нет	Да

(1) Для метода, когда используется соглашение Register

(2) Для автономных функций и процедур, когда используется соглашение Register

(3) Для всех других случаев при всех соглашенияч о вызове

(4) Только для результата, который полностью помещается в регистр. См. таблицу для полного обзора как результаты возвращаются из функции.

(5) В плоской 32-битной модели памяти все сегментные регистры нормально указывают на один и тот же сегмент памяти. Тем не менее, при анализе поведения Дельфи, оказывается, что регистр cs имеет различное значение.

Таблица 2: Передача параметров в функции и процедуры

В следующей таблице приведены сведения о передаче параметров по значению (включая, директиву const) в процедуры и функции Дельфи. При передаче по ссылке (директива var), все параметры передаются как 32-битные указатели.

Тип	Размер	Регистр (1)
ShortInt	1 байт (2)	Да
SmallInt	1 слово (2)	Да
LongInt	1 двойное слово	Да
Byte	1 байт (2)	Да
Word	1 слово (2)	Да
Dword	1 двойное слово	Да
Int64	8 байт	Нет
Boolean	1 байт (2)	Да
ByteBool	1 байт (2)	Да
WordBool	1 слово (2)	Да
LongBool	1 двойное слово	Да
Char	1 байт (2)	Да
AnsiChar	1 байт (2)	Да
WideChar	1 слово (2)	Да
ShortString	32-битный указатель	Да
AnsiString	32-битный указатель	Да
WideString	32-битный указатель	Да
Variant	32-битный указатель	Да
Pointers	1 двойное слово	Да
Objects	32-битный указатель	Да
Class and Class reference	32-битный указатель	Да
Procedure pointer	1 двойное слово	Да
Method pointers	Два 32-битных указателя (3)	Нет
Sets	Значение типа байт/слово/двойное слово или 32-битный указатель (4)	Да (4)
Records	Значение типа байт/слово/двойное слово или 32-битный указатель (4) (5)	Да (4)
Static Arrays	Значение типа байт/слово/двойное слово или 32-битный указатель (4)	Да (4)

Dynamic arrays	32-битный указатель	Да
Open array	Два 32-битных значения (6)	Нет
Single	4 байта	Нет
Double	8 байт	Нет
Extended	12 байт (7)	Нет
Real48	8 байт (8)	Нет
Currency	8 байт	Нет

(1) Если указано, то тип передается через регистр. Типы, которые не указаны, всегда передаются через стек.

(2) Когда эти типы занимают менее 32 бит, тогда при передаче на стек они всегда занимают 32 бита, и значение находится в младшей части, содержимое оставшей части неопределено.

3) Указатели на метод передаются через стек, как два 32-битных указателя, указатель на экземпляр помещается перед указателем на метод, так что позже это становится младшим адресом.

(4) Если тип помещается в байт/слово/двойное слово, то он передается непосредственно. Иначе, передается 32-битный указатель на память, где хранится этот тип.

(5) При использовании соглашения по вызову типа cdecl, stdcall или safecall, записи всегда передаются через стек и их размер округляется в сторону большего двойного слова.

(6) Первое значение это 32-битный указатель на массив, а второе значение содержит количество элементов в массиве.

(7) Используются только младшие 10 байт.

(8) Используется только младшие 6 байт.

Таблица 3: Результаты возврата функций

В следующей таблице приведен обзор того, как результаты возвращаются из функции в программу. Для более подробной информации насчет каждого типа, читайте соответствующий раздел.

Тип Дельфи	Результат	Размер
ShortInt	al	8-битное значение
SmallInt	ax	16-битное значение
LongInt	EAX	32-битное значение
Byte	al	Значение типа байт
Word	ax	Значение типа слово
Dword	EAX	Значение типа двойное слово
Int64	EDX:EAX	64-битное значение
Boolean	al	Значение типа байт
ByteBool	al	Значение типа байт
WordBool	ax	Значение типа слово
LongBool	EAX	Значение типа двойное слово
Char	al	Значение типа байт
AnsiChar	al	Значение типа байт
WideChar	ax	Значение типа слово
ShortString	Указатель в Result (1)	32-битный указатель
AnsiString	Указатель в Result (1)	32-битный указатель
WideString	Указатель в Result (1)	32-битный указатель
Variant	Указатель в Result (1)	32-битный указатель
Pointers	EAX	32-битный указатель
Objects	EAX	32-битный указатель
Class and Class reference	EAX	32-битный указатель
Procedure pointer	EAX	32-битный указатель
Method pointers	Указатель в Result (2)	2 x 32-битных указателя
Sets	EAX или Result (3)	Непосредственно или как 32-битный указатель (3)
Records	EAX или Result (3)	Непосредственно или как 32-битный указатель (3)
Static Arrays	EAX или Result (3)	Непосредственно или как 32-битный указатель (3)
Dynamic arrays	Указатель в Result (1)	32-битный указатель
Single	ST(0)	n/a
Double	ST(0)	n/a
Extended	ST(0)	n/a
Real48	ST(0)	n/a
Currency	ST(0) (4)	n/a

(1) Переменная Result в действительности передается в функцию, как дополнительный **var** параметр. Эта переменная Result содержит 32-битный указатель на область результата в памяти. Подлинное местонахождение

зависит от типа использованного соглашения о вызове: Для соглашения register это может быть EAX, EDX или ECX, в зависимости от количества переданных параметров. В других случаях Result это 32-битный указатель на стеке.

(2) Переменная Result указывает на адрес памяти где расположены два 32-битных указателя. Этот указатель передается так, как если бы он был действительно объявлен, и его точное местонахождение зависит от типа используемого соглашения о вызове.

(3) Если подлинный тип помещается в 32 бита, то он возвращается напрямую через регистр **al/ax/EAX**. Иначе, Result содержит 32-битный указатель на переменную памяти, и он передается в функцию, как если бы он был объявлен как дополнительный 32-битный **var** параметр. Этот параметр (точное местонахождение зависит от типа использованного соглашения о вызове) должен содержать указатель на действительные данные в памяти

(4) Значение в ST(0) является маштабированным значением (x10000). Для примера, значение 5,8745 возвращается как 58745.

Как связаться с Гуйдо Гайбелсом

Я – глава ICT, подразделения Королесвкого Национального Института Глухих (Royal National Institute for Deaf people) (RNID), крупного благотворительного учреждения в Великобритании. Моя основная задача – возглавлять группу ICT с целью определения и использования преимуществ технологий для нужд глухих и плохо слышащих, особенно в сфере информации и коммуникационных технологий. Под моим управлением группа ICT будет проводить как исследования, так и разработки в данной области и моя задача сделать эту группу основным распространителем передовых технологий в этой области, более тесно работа с партнерами, организациями, определяющими различные технические стандарты, сообществом глухих и всеми, кто могут и желают реально помочь. Я сам из Бельгии, но в данное время проживаю и работаю в Лондоне.

Вы можете мне писать по адресу guido.gybels@rnid.org.uk