№ 19 1–15 октября 2010 *Основана в 1995 г* inf.1september.ru

УЧЕБНО-МЕТОДИЧЕСКАЯ ГАЗЕТА ДЛЯ УЧИТЕЛЕЙ ИНФОРМАТИКИ

ИНФОРМАТИКА

/ тема номера:

Эксперименты в памяти

Изучение средствами Delphi способов хранения в компьютерной памяти различных данных

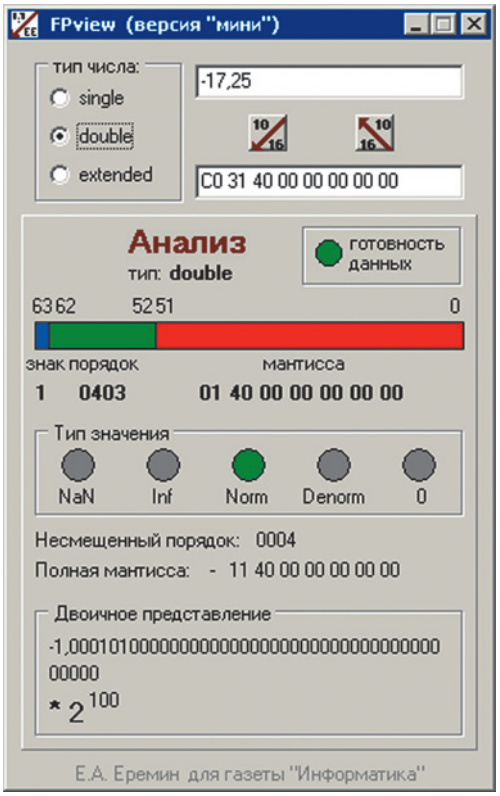
19

издательский дом Первое сентября

ИНФОРМАТИКА | Индексы подписки | Почта России – 79006 (инд.); – 79574 (орг.) | Роспечать – 32291 (инд.); – 32591 (орг.)

В НОМЕРЕ

НА ДИСКЕ



Электронные материалы к этому номеру размещены на диске, который вложен в № 20.

На диске находятся исходные коды и исполняемые модули всех рассматриваемых примеров. Автор всех программ — Е.А. Еремин.

ИНФОРМАТИКА

Учебно-методическая газета для учителей информатики
Основана в 1995 г.
Выходит два раза в месяц

РЕДАКЦИЯ:
гл. редактор С.Л. ОСТРОВСКИЙ
редакторы
Е.В. АНДРЕЕВА,
Д.М. ЗЛАТОПОЛЬСКИЙ
(редактор вкладки “В мир информатики”)
верстка Н.И. ПРОНСКАЯ
корректор Е.Л. ВОЛОДИНА
секретарь Н.П. МЕДВЕДЕВА
Фото: фотобанк Shutterstock
Газета распространяется по подписке
Цена свободная
Тираж 3000 экз.
Тел. редакции: (499) 249-48-96
E-mail: inf@1september.ru
http://inf.1september.ru

ИЗДАТЕЛЬСКИЙ ДОМ “ПЕРВОЕ СЕНТЯБРЯ”
Главный редактор:
Артем Соловейчик (Генеральный директор)
Коммерческая деятельность:
Константин Шмарковский (Финансовый директор)
Развитие, IT и координация проектов:
Сергей Островский (Исполнительный директор)
Реклама и продвижение:
Марк Сартан
Мультимедиа, конференции и техническое обеспечение:
Павел Кузнецов
Производство:
Станислав Савельев
Административно-хозяйственное обеспечение:
Андрей Ушков
Дизайн:
Иван Лукьянов, Андрей Балдин
Педагогический университет:
Валерия Арсланьян (ректор)

ГАЗЕТЫ ИЗДАТЕЛЬСКОГО ДОМА:
Первое сентября – Е.Бирюкова
Английский язык – А.Громушкина
Библиотека в школе – О.Громова
Биология – Н.Иванова
География – О.Коротова
Дошкольное образование – М.Аромштам
Здоровье детей – Н.Сёмина
Информатика – С.Островский
Искусство – М.Сартан
История – А.Савельев
Классное руководство и воспитание школьников – О.Леонтьева
Литература – С.Волков
Математика – Л.Рослова
Начальная школа – М.Соловейчик
Немецкий язык – М.Бузоева
Русский язык – Л.Гончар
Спорт в школе – О.Леонтьева
Управление школой – Я.Сартан
Физика – Н.Козлова
Французский язык – Г.Чесновицкая
Химия – О.Блохина
Школьный психолог – И.Вачков

УЧРЕДИТЕЛЬ:
ООО “ЧИСТЫЕ ПРУДЫ”

Зарегистрировано ПИ № 77-72230 от 12.04.2001 в Министерстве РФ по делам печати
Подписано в печать:
по графику 9.09.2010,
фактически 9.09.2010
Заказ №
Отпечатано в ОАО “Чеховский полиграфический комбинат”
ул. Полиграфистов, д. 1,
Московская область,
г. Чехов, 142300

АДРЕС ИЗДАТЕЛЯ:
ул. Киевская, д. 24,
Москва, 121165
Тел./факс: (499) 249-31-38

Отдел рекламы:
(499) 249-98-70
http://1september.ru

ИЗДАТЕЛЬСКАЯ ПОДПИСКА:
Телефон: (499) 249-47-58
E-mail: podpiska@1september.ru

Документооборот Издательского дома “Первое сентября” защищен антивирусной программой Dr.Web

ПОДПИСНЫЕ ИНДЕКСЫ: по каталогу «Газеты. Журналы» агентства «Роспечать» – 32291 (инд); – 32591 (орг.); по каталогу «Почта России» – 79006 (инд); – 79574 (орг.)

НОВОСТЬ № 1

Взять живым!

“Живой поиск” от Google: технологии впечатляют, к функциональности надо привыкнуть

В начале сентября особо внимательные пользователи обратили внимание на то, что логотип Google на странице поиска стал временами “пританцовывать”. Google частенько — по большим и не очень праздникам — радует нас вариантами логотипа, иногда весьма остроумными (коллекцию особо удачных можно посмотреть здесь http://www.google.com/logos/). Но в данном случае явного повода, казалось, не было, а логотип на странице поиска ожил. Конечно, неспроста...

Таким способом поисковик обращал внимание (намекал) на принципиально новую возможность, которую, видимо, ранее не предлагал никто. Теперь “на лету” — по мере набора пользователем поискового запроса — меняются не только варианты самого запроса, но и *результаты* поиска по нему. Нельзя не сказать, что с технологической точки зрения это очень круто. Не уверен, что всю степень крутизны могут оценить обычные пользователи, но IT-специалисты, понимающие, чего это должно стоить, обязаны снять шляпу.

Самими по себе AJAX-овыми технологиями уже давно никого не удивишь, и Google, возможно, первый, кто этому поспособствовал (если вы никогда не слышали про AJAX, имеет смысл почитать здесь http://ru.wikipedia.org/wiki/AJAX и здесь http://ru.wikibooks.org/wiki/AJAX). Но одно дело — относительно медленный обмен с сервером и совсем другое — поиск на лету, когда вместо одного поискового запроса фактически приходится обрабатывать несколько. Сколько, видимо, сейчас знают только программисты Google, но рискну предположить, что нагрузка на поисковую машину должна была вырасти не менее, чем раза в три.

Google не рискнул сразу оживить весь поиск и вводит новую опцию постепенно. Во-первых, чтобы вообще увидеть живой поиск, надо искать, будучи залогированным в аккаунт Google. Во-вторых, возможность живого поиска постепенно распространяется по национальным поисковым страницам (приятно, что Россия оказалась в авангарде этого процесса).

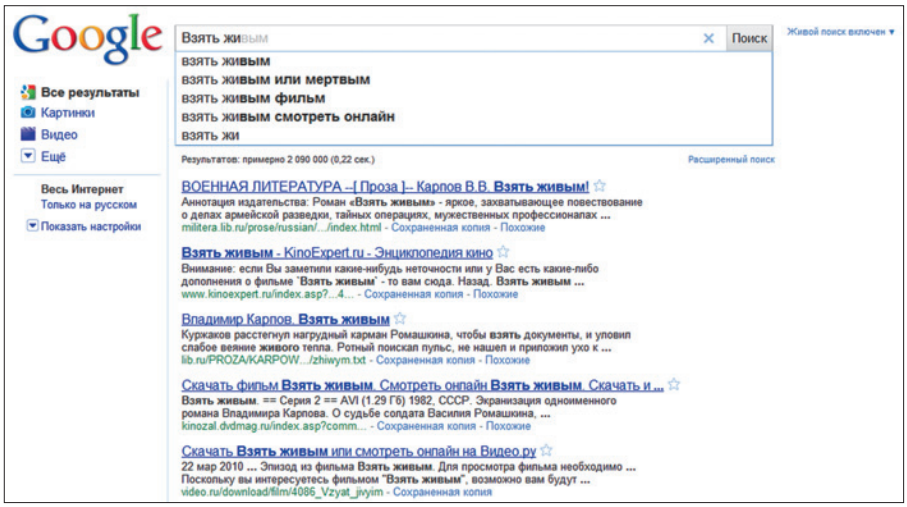
Но это все технологии, а что дает живой поиск с содержательной точки зрения? Делает ли он поиск удобнее?

Видимо, да, но использование такого поиска требует привычки. Если вы в точности знаете, *что* ищете и *как* это найти, то от новой опции пользы не так много. В подобной ситуации вы скорее всего способны сразу сформулировать запрос, приводящий к цели. А вот в том случае, когда вы знаете только *что* ищете, но не уверены в том, как это найти, живой поиск может помочь серьезно сэкономить время. Поскольку в такой ситуации чаще всего приходится экспериментировать с поисковыми запросами — а так попробую... а вот так... вот это слово добавлю... это уберу... вот так сформулирую. Для таких экспериментов живой поиск — прекрасный инструмент.

Интересные факты о живом поиске от Google

Источник: http://www.google.ru/instant

- В среднем на ввод одного поискового запроса требуется более 9 секунд.
- В живом поиске на одном запросе можно экономить от 2 до 5 секунд.
- Если все перейдут на живой поиск, экономия составит более 3,5 миллиардов секунд в день.



Эксперименты в памяти

Изучение средствами Delphi способов хранения в компьютерной памяти различных данных

Е.А. Еремин,
г. Пермь

...Мы интуитивно понимаем, что данные предшествуют алгоритму, ведь, прежде чем выполнять какие-либо операции, нужно иметь объекты, к которым они применяются.

Н.Вирт [2]

Введение

Вам когда-нибудь (хотя бы единожды!) хотелось заглянуть внутрь памяти компьютера и благодаря каким-то экстраординарным способностям суметь разглядеть, как там все хранится?

Мне кажется, большинство учителей информатики, не задумываясь, ответят на этот вопрос утвердительно. Почему? Да потому, что знание основных идей компьютерного представления данных очень помогает понять многие важные для практики особенности их обработки. То, что память состоит из отдельных битов, сразу же позволяет сделать важный вывод о дискретности *любых* компьютерных данных. Без представления о ячейках памяти невозможно до конца понять, какую роль играет в современном компьютере байт и почему все типы данных занимают в памяти только целое количество байт. Наконец, как вы объясните, что значит сообщение "floating point overflow", ничего не сказав о представлении чисел?

Вам когда-нибудь (хотя бы единожды!) хотелось заглянуть внутрь памяти компьютера и благодаря каким-то экстраординарным способностям суметь разглядеть, как там все хранится?

Существует и еще один, на мой взгляд, важный аспект. Как проверить достоверность и точность тех сведений, которые мы хотим дать ученикам (например, из Интернета)? Вот яркий пример, с которым я совсем недавно столкнулся в своей практике. В современных IBM-совместимых ПК существует специальный тип вещественных чисел, который называется `extended`. Он примечателен тем, что имеет максимально возможную аппаратную[1] разрядность — 80 бит. Иначе говоря, это самое "мощное" представление вещественных чисел, "штатно" предусмотренное в современных компьютерах (именно поэтому его хорошо знают те, кто часто принимает участие в олимпиадах по информатике). Зададим себе вопрос: *каково самое маленькое положительное вещественное число, которое может аппаратно сохранить компьютер IBM PC?* Найти внятный ответ на такой, казалось бы, простой и естественный вопрос оказалось на удивление непросто. Руководство по Borland Pascal [8] указывает для типа `extended` минимальное значение $3{,}4*10^{-4932}$, а известная книга [1] — 10^{-4951}. В Интернете даже при беглом поиске "выплыли" константы $1{,}9*10^{-4952}$, $3{,}6 * 10^{-4951}$

[1] Разумеется, программным способом можно организовать вычисления с разрядностью во много раз больше!

и $8*10^{-4933}$. Ну и какое же из них нам писать на доске? Видимо, перед нами один из случаев, когда самый надежный вариант проверки — спросить значение у самого компьютера, что мы позднее обязательно сделаем (см. эксперимент 6).

Одним из принципиальных препятствий изучения содержимого памяти служит, как ни странно, современное программное обеспечение. Заботясь о "невмешательстве" в свою работу, любая многозадачная операционная система "всеми силами" пресекает любые попытки получить доступ к конкретной ячейке памяти. Не случайно большинство экспериментов над данными производится в "старом добром" Турбо Паскале [4–6], рассчитанном на однопользовательский MS-DOS и полный контроль над памятью машины. Между прочим, сейчас и здесь результаты выглядят не так убедительно, ибо, будучи запущенным в среде Windows, Турбо Паскаль способен контролировать лишь некоторую виртуальную машину MS-DOS, которую создает для него Windows. Так что когда вы с помощью программы на Паскале записали код в видеопамять и увидели символ, ему соответствующий, — все это искусная симуляция, поскольку ОС никогда не даст вам доступа в "настоящую" видеопамять[2]! Да и сама среда Турбо Паскаля уже кажется такой архаичной, что возникают сомнения, надо ли в ней уметь работать нынешнему школьнику.

Так и хочется воскликнуть: "Вперед — в более современную среду Delphi!" Но мы только что говорили о современных программах: они стремятся как можно дальше отделить нас от памяти. Вот и из Delphi по сравнению с Турбо Паскалем изъяты функции определения адреса переменной `seg()` и `ofs()`, а главное, исключен предопределенный массив `mem`, с помощью которого можно было получить доступ к байтам памяти. Конечно, можно назвать множество убедительных причин, почему сделанные изменения логичны, но проблему доступа к ячейкам это решить не поможет.

Мне давно хотелось научиться исследовать содержимое переменных в Delphi, но целенаправленное чтение помощи долго не давало никаких результатов. Тем не менее терпение обычно бывает вознаграждено, и, наконец, удалось найти то, что я искал: некоторую специфическую конструкцию языка, с помощью которой удается извлечь байты, соответствующие любой интересующей переменной. Полученным опытом я и хотел бы поделиться с читателями[3].

Итак, **цели данной публикации** следующие:

1. Разработать простой метод изучения двоичного содержимого переменных в среде Delphi.

2. Используя этот метод, рассмотреть некоторые особенности представления данных в Delphi по сравнению с Турбо Паскалем.

3. Вычислить и уточнить некоторые характеристики хранения вещественных чисел в математическом сопроцессоре (как пример задачи, где Delphi выступает только инструментом исследования, поскольку здесь предмет изучения есть работа сопроцессора, а не особенности конкретного языка программирования).

4. Разработать учебную инструментальную программу, демонстрирующую для трех важнейших вещественных типов процесс перевода любого числа в его внутреннее представление и обратно.

5. Попутно рассказать о некоторых интересных проблемах хранения данных в ОЗУ, которые часто остаются за рамками рассмотрения в школьном курсе информатики.

1. Технология экспериментов

1.1. Оператор absolute

В основе всех наших экспериментов с памятью будет лежать оператор **`absolute`**. Изначально он предназначен для задания переменной абсолютного адреса, но в таком виде он нам бесполезен, поскольку одна из проблем как раз в том и заключается, что адрес переменной в ОЗУ неизвестен. Но у оператора есть еще частная форма, которая имеет синтаксис:

```
var a: <type1>;
 b: <type2> absolute a;
```

Описание означает, что адресом памяти для переменной `b` компилятор назначит *тот же самый адрес*, который имеет переменная `a`. Если переменные `a` и `b` к тому же имеют одинаковую длину в байтах (сама конструкция **`absolute`** этого, разумеется, не требует), то можно говорить о том, что они в памяти полностью совпадают и "наложены" друг на друга. Очень важно подчеркнуть, что типы переменных `<type1>` и `<type2>` совершенно произвольны, так что мы получаем поистине редкую возможность общаться с одной и той же областью памяти *разными способами*.

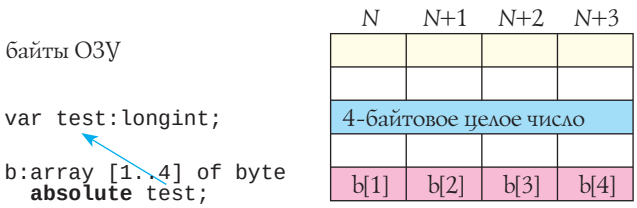


Рис. 1. Принцип работы оператора **`absolute`**

На *рис.* 1 приведен конкретный пример такого совмещения переменных. Изучаемая переменная `test` хранится в памяти начиная с адреса *N*. Мы не можем его узнать, но с помощью конструкции **`absolute`** имеем возможность "наложить" на `test` массив `b` из четырех байт, что позволит нам получить доступ к любому из них через элементы массива `b[1]–b[4]`. А это-то нам как раз и нужно!

Интересно, что аналог подобной конструкции существовал в Фортране и назывался `COMMON` — общая область памяти.

Желательно понимать, что применение оператора **`absolute`** для разных типов переменных потенциально опасно, поскольку, пользуясь операциями над

[2] К тому же и видеопамять давно работает по-другому: не в текстовом, а в графическом режиме!

[3] Не скрою, что оформить свои мысли в форме данной статьи меня первоначально подтолкнуло чтение недавней глубокой и интересной публикации [6], где для анализа способов хранения данных по-прежнему применялся Borland Pascal; но, как это часто бывает, при подготовке материала удалось найти много интересного вне связи с первоначальными замыслами.

<type1>, можно получить результат, недопустимый для <type2>. Но мы будем все делать внимательно и аккуратно, так что новый оператор не принесет нам ничего, кроме пользы.

1.2. Консольное приложение

Для многих простых экспериментов не требуется организовывать удобный пользовательский интерфейс. Например, если нас интересует минимальное значение типа `extended` и программа его как-то вычисляет, то достаточно вывести результат на экран, как в старых MS-DOS-программах. Конечно, можно было бы создать на форме метку или поле редактирования и вывести туда результат, но это сильно напоминает то, что в народе называют "стрельбой из пушки по воробьям".

В свете сказанного, многие наши простые эксперименты будем выполнять в виде так называемых *консольных приложений*, которые хотя и до боли напоминают черное окно исполнения Турбо Паскаля, но тем не менее пишутся на принятом в Delphi диалекте Паскаля и главное — в удобной оконной среде разработки.

Чтобы создать консольное приложение в Delphi 7, достаточно в меню **File** выбрать пункт **New** и далее, поскольку консольное приложение "не самый ходовой товар", пункт **Other...** (другие — см. *рис.* 2).

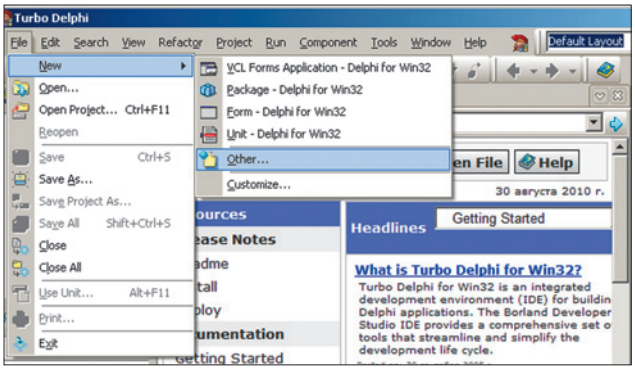


Рис. 2. Создание консольного приложения (начало)

В открывшемся диалоговом окне среди многочисленных вариантов приложений остается только найти и выбрать **Console Application** (*рис.* 3).

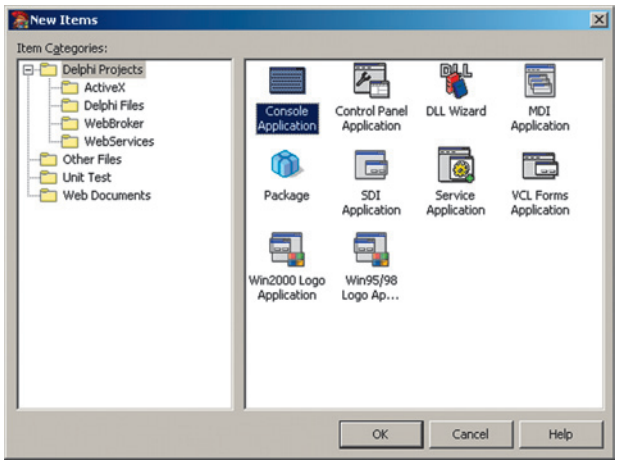


Рис. 3. Создание консольного приложения (выбор)

В итоге "волшебник" (wizard) создания приложений автоматически сгенерирует следующий небольшой фрагмент:

```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  { TODO -oUser -cConsole Main :
              Insert code here }
end.
```

Описания переменных в нем рекомендуется располагать непосредственно перед словом **begin**, а саму программу — между комментарием "Insert code here", т.е. "вставьте код (программу!) здесь", и словом **end**. Как это выглядит на практике, рассмотрим на конкретном примере.

1.3. Пример задачи: хранение в памяти многобайтовых данных

Пусть в процессоре лежит некоторое целое число, размер которого составляет несколько байт (для определенности возьмем 32-битное целое типа `longint`, хотя конкретное количество бит и тип несущественны). Думаю, все понимают, что процессор хранит это число в одном из своих регистров как единое целое[4]. Возникает вопрос: как сохранить это единое 4-байтовое число в ОЗУ, каждая ячейка которого имеет емкость 1 байт?

Пусть (опять-таки для определенности) число равно 12345678_{16}. Очевидно, что каждый байт числа — это две шестнадцатеричные цифры и разбиение на байты следующее: 12 34 56 78. Тогда возможно два одинаково логичных способа сохранения нашего числа в четыре последовательных байта ОЗУ. Они изображены на *рис.* 4.

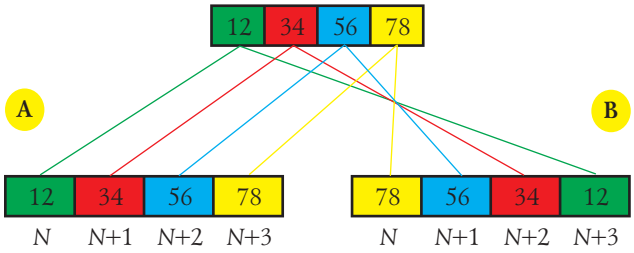


Рис. 4. Хранение многобайтовых данных в ОЗУ (*A* — big-endian, *B* — little-endian)

В варианте *A* байт с наиболее значащей частью ("big-end", в исходном числе он находится слева) сохраняется в память *по* ***наименьшему*** *адресу* (на рисунке это *N*). Такой способ принято называть "*big-endian*". В варианте *B* этот байт, напротив, сохраняется в память *по* ***наибольшему*** *адресу* (на рисунке это *N*+3). Следовательно, первым, наоборот, сохраняется байт с наименее значащей частью ("little-end", в исходном числе находится справа). Это, как понятно по аналогии, "*little-endian*". По-русски обычно используют не очень удачные эквиваленты терминов: big-endian — это *обратное размещение байтов*, а little-endian — *прямое*.

[4] Битов в регистрах современного процессора хватит даже на 8-байтовое число!

Хотя для математиков нумерация байтов "с младшего конца" более естественна, но язык не поворачивается, глядя на *рис.* 4, называть способ *B* прямым.

Кстати говоря, термины big-/little-endian были предложены в одной из статей, посвященных рассматриваемому вопросу, со ссылкой на книгу Джонатана Свифта "Путешествие Гулливера". Как вы, наверно, помните, в Лилипутии ради обсуждения проблемы, с какого конца — тупого или острого (по-английски "big side" или "little side") разбивать яйцо, были созданы две непримиримые политические партии.

Из литературы известно, что в компьютерах IBM PC принят прямой (little-endian, вариант *B* на *рис.* 4). Это утверждение мы и хотим экспериментально проверить.

Эксперимент 1. Порядок байтов данных в памяти IBM PC

Постановка задачи. Имеется некоторая величина, состоящая из нескольких байт, например, целое число. Рассмотрим для конкретности 4-байтовый тип `longint`, хотя все сделанные выводы распространяются и на другие числовые многобайтовые типы. Вопрос: как (в каком порядке) байты числа лежат в памяти ПК IBM PC?

Реализация эксперимента. Создадим консольное приложение, как описано выше, и дополним его небольшой программкой, в соответствии с листингом 1 (цветом выделен текст, который надо набрать в дополнение к автоматически сгенерированному Delphi).

Листинг 1

```
program byte_order;
{$APPTYPE CONSOLE}
uses
  SysUtils;

const Nb = 4;
type t = longint;
var  test: t;
     b: array [1..Nb] of byte absolute test;
     i: integer;

begin
  {TODO -oUser -cConsole Main :
                Insert code here }
  for i := 1 to Nb do b[i] := 0;
  b[Nb] := 1;
  writeln(test);
  for i := 1 to Nb do write(b[i]:4);
  readln;
end.
```

Верхний фрагмент описывает переменные `test` и `b`, которые "наложены" в памяти друг на друга в полном соответствии с *рис.* 1. (Конечно, без обозначения констант и типов вполне можно было обойтись, но более общая форма записи облегчит, если потребуется, расширение программы.)

Обратимся теперь ко второй части, описывающей собственно эксперимент. Суть его сводится к тому, что в массиве (и одновременно в числе) создается последовательность байт 0 0 0 1, которая затем выводится на экран. Если бы IBM PC использовал big-endian представление чисел, на экране мы увидели бы единицу. На практике результат другой — см. *рис.* 5.

Рис. 5. Результат работы листинга 1

Проверка на калькуляторе показывает, что число 16777216 есть 2^{24}, что прекрасно согласуется с переводом исходного числа $01\ 00\ 00\ 00_{16}$ в десятичную систему.

Зато если единицу положить не в последний, а в первый элемент массива (`b[1] := 1`), то в полном соответствии с теорией на экране появится единица.

Вывод. ПК IBM PC использует little-endian порядок байт. На практике это означает, что прочитанные из памяти байты для правильного формирования многобайтовых данных следует переставлять! (01 00 00 00 → 00 00 00 01 и т.п.)

Заметим, что данный вывод ни в коем случае не следует распространять на ASCII-строки, поскольку там каждый символ рассматривается по отдельности (сравните 4-байтовое число и 200-символьную строку, которая даже не поместится единовременно ни в один из регистров процессора!).

2. Данные: от Турбо Паскаля к Delphi

При переходе от Турбо Паскаля к Delphi с данными произошло довольно много изменений. Их основой, без сомнения, стали (как минимум) две важные причины: увеличение в 2–4 раза разрядности процессоров и существенное недовольство программистов некоторыми ограничениями Паскаля, например, принципиальной невозможностью иметь строку более 255 символов. В результате в Delphi не только появились новые типы данных, но и старые (`integer`, `string`) получили бóльшие возможности.

В подтверждение предлагаю вашему вниманию несколько очень простых экспериментов. При описании предполагается, что вы прочитали предыдущий раздел и уже сумеете создать консольное приложение.

2.1. Целые числа

Центральное новшество в этой области состоит в том, что в Delphi тип `integer` уже не двух-, а четырехбайтовый[5], т.е. фактически неотличим от `longint` в Турбо Паскале! Ради сохранения преемственности в Delphi введен тип `smallint`, который полностью эквивалентен `integer` в Турбо Паскале.

А еще для любителей вычислений с большими числами в Delphi добавлен 64-разрядный тип `int64`. Полная таблица целочисленных типов приведена на с. 8.

[5] Факториал от 8 теперь уже не будет отрицательным, эта неприятность "отложится" до 17!

Тип	Диапазон	Наличие знака	Длина (бит)
shortint	–128 .. 127	знаковый	8
smallint	–32768 .. 32767	знаковый	16
longint (integer)	–2147483648 .. 2147483647	знаковый	32
int64	-2^{63} .. $2^{63} - 1$	знаковый	64
byte	0 .. 255	беззнаковый	8
word	0 .. 65535	беззнаковый	15
longword (cardinal)	0 .. 4294967295	беззнаковый	32

Эксперимент 2. Количество байт в integer

Постановка задачи. Убедиться, что тип integer действительно занимает в памяти 4 байта.

Реализация эксперимента. Простейший способ проверить это — задать большое число, для убедительности — 2 147 483 647 (см. таблицу).

Описательная и исполняемая части программы могут быть, например, следующими (добавьте их к шаблону консольного приложения в соответствии с цветом так, как это сделано в листинге 1)[6].

Листинг 2

```
const Nb = 4;
type  t = integer;
var   test: t;
      b: array [1..Nb] of byte absolute test;
      i:integer;

  test := 2147483647; {2^31 – 1}
  writeln(test);
  for i := 1 to Nb do write(b[i]:4);
  readln;
```

Результат выполнения эксперимента на экране будет выглядеть следующим образом:

```
2147483647
255 255 255 127
```

Мы видим, что число "воспринимается" и сохраняется, а значит, значение уже не двухбайтовое, а минимум четырехбайтовое. Шестнадцатеричное представление числа (с учетом перестановки байт) — 7F FF FF FF (т.е. 127 255 255 255 в десятичном виде), подтверждает, что перед нами максимальное положительное значение 32-битного целого числа со знаком.

Стоит также провести еще одну дополнительную проверку: увеличить данное максимально допустимое число на единицу и записать `test:=2147483648;` При попытке трансляции в нижней части окна с текстом программы появится предупреждающее сообщение:

[Warning] int_len.dpr(17): Constant expression violates subrange bounds

(примерный перевод: [Предупреждение] проект int_len.dpr(строка 17): значение константы нарушает границу допустимого диапазона)

Таким образом, Delphi подсказывает нам, что предел четырехбайтовой величины превышен. Тем не менее задача запускается (предупреждение — это "небольшая" ошибка), а ответ, как и следовало ожидать, выдается отрицательным.

Наконец, есть и еще один (совсем простой) способ проверки — вывести на экран значение стандартной константы maxint и убедиться, что оно равно 2 147 483 647, что соответствует именно четырехбайтовому целому[7].

Еще рекомендую поэкспериментировать с другими типами целочисленных данных, указанных в таблице, например, longint и smallint.

Выводы. В Delphi значение типа integer хранится не в двух, как в Турбо Паскале, а в четырех байтах; longint в обеих системах программирования совпадают, следовательно, в Delphi integer и longint одно и то же. Smallint — новый двухбайтовый тип Delphi, введенный для совместимости с integer в старых реализациях Паскаля.

2.2. Вещественные числа

А в этой области нас ждут еще более радикальные изменения. Тип real, к которому мы так привыкли в классическом Паскале, вообще объявлен устаревшим и не рекомендуется к употреблению (хотя и поддерживается ради совместимости под псевдонимом real48).

Вместо real предлагается использовать другие вещественные типы — single, double и extended (они есть и в Турбо Паскале). Все они непосредственно опираются на одноименные форматы данных математического сопроцессора, поэтому фактически даже не являются чем-то "сугубо паскалевским". Поскольку все эти важные с практической точки зрения данные оказываются независимыми от языка программирования, есть смысл рассмотреть круг вопросов, которые связаны с вещественными числами, отдельно. Это будет сделано в разделе 3.

Напомню также читателям, что исторически тип real был реализован в виде специализированной математической библиотеки Паскаля и не имел аппаратной поддержки.

2.3. Строковые данные

История такова, что в классическом Паскале не было специального строкового типа, а для обработки текста создавался сложный тип данных array of char. В более поздних реализациях компиляторов появился тип string с большим ассортиментом специализированных процедур и функций для работы с текстами. Программисты по достоинству оценили новые возможности, хотя заложенное в Турбо Паскале принципиальное 255-символьное ограничение (объяснение его появления см., например, в [4]), конечно, создавало трудности для программирования реальных задач.

При переходе к Delphi максимальная длина текста было сдвинута с 255 байт до 2 гигабайт, что для большинства практических (и всех учебных!) задач позволяет вообще забыть о существовании этого ограничения.

[6] Удобно взять за основу файл с листингом 1: тогда в описательной части достаточно заменить тип longint на integer.

[7] Более "мощный" вариант — использовать конструкцию вроде high(smallint).

Важно понять, что обеспечить хранение переменных такой длины традиционным выделением заранее участка памяти из фиксированного числа байт просто невозможно: при таком подходе потребуется зарезервировать под каждую строковую переменную по 2 Гб ОЗУ, что абсурдно. Как же все-таки решается эта проблема в Delphi (да и в других компиляторах тоже)?

Существует два принципиально разных метода хранения переменных: *статический* и *динамический*. Для того чтобы лучше понять разницу, обратимся к *рис.* 6, где изображено выделение памяти под статическую (`stat`) и динамическую (`dyn`) переменные.

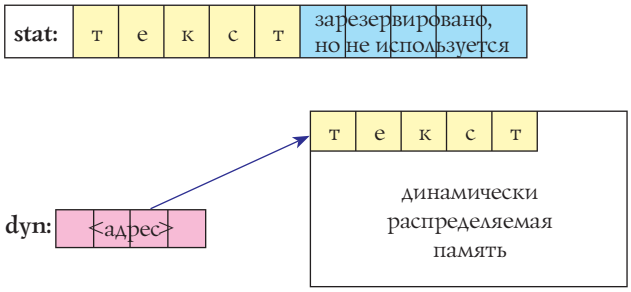


Рис. 6. Статические и динамические переменные

В первом случае *при трансляции* программы под переменную выделяется одна или несколько ячеек ОЗУ, и в этой области памяти позднее будут храниться значения переменной. Фактически везде, где в тексте программы имеется ссылка на данную переменную, компилятор подставляет команды обращения к адресам зарезервированной области памяти. Выделенная под переменную память не может быть использована иначе, чем для хранения значений этой переменной[8].

Динамический механизм работает совсем по-другому. В момент трансляции под переменную выделяется некоторое место в ОЗУ, которого достаточно только для хранения адреса (почувствуйте разницу: резервируется место не под сами данные, а под адрес, ссылающийся на участок памяти, где находятся данные). Память под данные в этот момент не выделяется — это делается *при исполнении* программы, когда переменной будет присваиваться конкретное значение. И в этом главный залог успеха: под значение будет отведено ровно столько динамической памяти, сколько требуется[9].

Применительно к текстам это выглядит так. В Турбо Паскале заранее резервируется память под строку максимально возможной длины, даже если при выполнении программы окажется, что переменная реально будет хранить всего один символ. (Сравните с бланком, где ширина колонки под фамилию всегда рассчитана на самый длинный текст, тогда как некоторые фамилии, вроде Ким или Усов, можно было бы поместить и в более узкую колонку.) В Delphi память под значение выделяется в процессе исполнения, поэтому переменная получает именно столько байтов, какова длина текста. Таким образом, в отличие от увеличения размера чисел `integer`, описанного в 2.1, удлинение строк потребовало еще и изменения способа организации памяти.

Новый "расширенный" тип `string` называется в Delphi `ANSIstring`. Для совместимости существуют также короткие строки `SHORTstring`, полностью воспроизводящие `string` в Турбо Паскале. Описание `string` по умолчанию трактуется как `ANSIstring`[10]. Описание `string[n]`, где $n \leqslant 255$, генерирует статическую форму `SHORTstring`.

Дополнительно в Delphi имеется еще тип `WIDEstring`, созданный для поддержки 16-битной кодировки Unicode.

Обогащенные столь подробными теоретическими знаниями, поэкспериментируем со строками.

Эксперимент 3. Изучение хранения строк

Постановка задачи. Изучить статическое размещение в памяти строковой переменной. Посмотреть, как выглядит содержимое памяти с динамической переменной.

Реализация эксперимента. В листинге 3 показаны описательная и исполняемая части программы для нашего эксперимента. Описание `string[3]` обеспечивает статическое выделение памяти под переменную `test`. В ходе выполнения программы ей присваивается значение `'A 1'`, а затем байты переменной выводятся на экран.

Листинг 3

```
const Nb = 8;
type t = string[3];
var  test: t;
   b: array [1..Nb] of byte absolute test;
   i:integer;
 test := 'A 1';
 for i := 1 to Nb do write(b[i]:4);
 readln;
```

В результате выполнения эксперимента на экране появятся числа:

```
   3  65  32  49   0   0   0   0
```

Первое из них — это фактическая длина текста. Три следующих — это десятичные(!) коды символов, а остальные нули показывают, что мы вывели слишком много байт. Полученная картина "один к одному" повторяет результаты для Турбо Паскаля [4].

Очень поучительно взглянуть, что получится, если в переменную **test** попробовать занести более трех символов: результат не изменится, поскольку Паскаль просто игнорирует лишние символы.

Наконец, заменим описание `string[3]` на `string`. Тем самым, вместо статической строки получим динамическую, и результат эксперимента будет другим. На моем экране появились числа:

```
 180   7 134   0   0   0   0   0
```

(у вас вполне могут быть другие значения). Это ссылка на строковую область памяти, найти которую в рамках нашего грубого эксперимента затруднительно.

Если хватит терпения, можете присвоить динамической переменной **test** строковое значение,

[8] Понимаете теперь, почему Delphi предупреждает вас, когда переменная описана, но ни разу не использовалась?

[9] Тот, кто пользовался в программах на Паскале типом "указатель", хорошо представляют данный механизм; возможно, преподаватели со стажем также вспомнили об области string space в языке MSX BASIC.

[10] Если установить директиву компилятора `$H-`, то установится совместимость со строками Турбо Паскаля.

длина которого превышает 255 символов, и напечатать его; для этого добавьте в программу строку `writeln(test);`

Выводы. В Delphi в дополнение к Турбо Паскалю добавляется принципиально новый строковый тип практически неограниченной длины; он реализован не статическим, а динамическим образом.

2.4. Тип variant

В Delphi появляется принципиально новый тип данных `variant`, который может помещать внутрь переменной значения *разного* типа: например, сначала `integer`, а затем `string`. В классическом Паскале такое вообще невозможно. Кроме того, действия над значениями типа `variant` выполняются особым образом. Например, если `V1` и `V2` имеют тип `variant`, и `V1 = 12.5, V2 = '12'`, то `V1 + V2` будет равно `24.4`, потому что строка `'12'` перед сложением будет автоматически преобразована в число.

Рассмотрим кратко, как же устроен этот необычный тип данных. Каждая переменная типа `variant` состоит из 16 байт, причем первая их половина хранит служебную информацию, а вторая — значение переменной или ссылку на это значение. Самые первые 2 байта переменной содержат целое число, характеризующее тип значения, которое в данный момент лежит в переменной. Полный список возможных типов содержит около 20 разных значений, начиная целыми и вещественными числами и кончая символьными значениями и даже массивами. Например, для числа типа `integer` характеристическое значение равняется 3, для вещественного числа типа `double` — 5, а для `string` — $100_{16} = 256_{10}$. Остальная часть служебной части пока не используется (зарезервирована) и заполняется нулями.

Эксперимент 4. Как устроен тип variant?

Постановка задачи. Изучить, как тип `variant` хранит в себе разные значения, например, `integer` и `string[n]`.

Реализация эксперимента. Программа для проведения эксперимента приведена в листинге 4.

Листинг 4

```
const Nb = 20;
type  t = integer {string[3]};
var   test: variant; e:t;
    b: array [1..Nb] of byte absolute test;
    i:integer;
```

```
  readln(e);
  test := e;
{тип}
  writeln(b[2],' ',b[1]);
{резерв}
  for i := 3 to 8 do write(b[i],' ');
  writeln;
{данные}
  for i := 9 to Nb do writeln(b[i],' ');
  readln;
```

В описательной части при первом запуске будем ставить тип `integer`, а при втором — `string[3]`. В программе по необходимости предусмотрена дополнительная переменная `e`, в которую вводится значение с клавиатуры. Дело в том, что непосредственно в переменную `variant` вводить с клавиатуры нельзя, поскольку компьютер не всегда способен однозначно определить, какого именно типа набранное значение (попробуйте, в частности, сами догадаться: введенная с клавиатуры последовательность символов "15 3" — это два целых числа, разделенных пробелом, или же строка?).

Хотя тип `variant` хранится в 16 байтах, в описании указано значение 20. По результатам второго запуска мы увидим, как пригодится этот запас.

Вывод на экран несколько усложнен, зато он легче читается. Так, при выводе характеристического числа байты предусмотрительно переставляются, что позволяет легче расшифровать значение константы для `string`. Все резервные нулевые байты для экономии места выводятся в одну строку, и лишь остальные байты не обрабатываются при выводе.

Итак, запустим программу для целочисленных данных. На экране получатся следующие результаты (для экономии места семь последних нулей не показаны):

```
65536
0 3
0 0 0 0 0 0
0
0
1
0
0
```

Расшифруем приведенную выдачу на экран. Первая строка содержит число, которое я вводил. Далее выдается число 3, которое, как мы знаем, означает тип `integer`. Пропуская строку резервных нулей, обращаем внимание на следующие 4 байта, которые и есть собственно число. Выпишем его, традиционно переставляя байты, и получим 00 01 00 00; после перевода из шестнадцатеричного представления в десятичное убедимся, что перед нами введенное число.

Теперь заменим описание `integer` на `string[3]` и запустим программу еще раз. "Протокол работы" выглядит следующим образом.

```
A 1
1 0
0 0 0 0 0 0
180
7
133
0
0
0
0
0
3
65
32
49
```

Прокомментируем содержимое выдачи. Текст `"A 1"` я набрал по аналогии с экспериментом 3. В следующей строке вывелась характерная для строкового типа константа 01 00 (байты уже переставлены программно!). Пропускаем резервные нули и видим странные числа. Вспомните предыдущий эксперимент: это не что иное,

как ссылка на то место ОЗУ, где лежит строка. И на этом бы все и закончилось, но нам сильно повезло: это самое место ОЗУ оказалось сразу после переменной типа `variant` и мы его "зацепили"! Отсчитайте 16 чисел, начиная с 1 во второй(!) строке (это 16 байт переменной `test`), и вы попадете на число 3, которое есть не что иное, как длина введенного текста. И далее знакомые нам десятичные коды введенных символов. Так что это то самое строковое значение, на которое ссылается тип `variant`.

Выводы. Переменные типа `variant` могут хранить значения разных типов. Компьютер распознает, какой именно тип имеет в данный момент содержимое переменной, по характеристическому числу из управляющей части структуры данных. Внутри переменной может лежать как само значение, так и ссылка на него.

Таким образом, очень простые программки позволили нам познакомиться с интересным новым типом данных в Delphi, а также с новыми свойствами старых типов, пришедших из Турбо Паскаля.

3. Вещественные числа в сопроцессоре

Представление вещественных чисел — одно из наиболее "темных" мест в современных учебниках информатики. Причин тут, видимо, много, в том числе не обходится без традиционного мотива "сложно это все, давайте объявим техническими деталями".

Считаю, что полноценный курс информатики невозможен без объяснения наиболее общих закономерностей представления вещественных чисел. Одного только знания принципов целочисленной арифметики недостаточно хотя бы потому, что целые числа дискретны, а вещественные непрерывны, так что вещественная арифметика будет обладать массой особенностей по сравнению с целочисленной!

Предлагаю читателям все-таки попробовать разобраться в основах представления вещественных чисел. Постараюсь со своей стороны сделать все, чтобы картина была максимально понятной, охватывала все наиболее важные идеи и, в то же время, не тонула в мелочах.

С точки зрения основной темы данной статьи рассматриваемая проблема есть отличная иллюстрация того, что можно использовать Delphi для изучения вопросов, не входящих непосредственно в область языка Паскаль.

3.1. Основные принципы представления вещественных чисел

Начнем с того, что форма представления вещественных чисел придумана не "компьютерщиками", а математиками. В науке для записи очень больших и очень маленьких чисел уже давно используется способ, основанный на том, что любое число A в системе счисления с основанием B можно записать в виде

$$A = (\pm M) * B^{\pm P},$$

где M называют *мантиссой*, а показатель степени P — *порядком* числа. Для десятичной системы это выглядит очень привычно, например: заряд электрона равен $-1{,}6*10^{-19}$ К, а скорость света в вакууме составляет $3*10^8$ м/сек. Обратите внимание, что мантисса — это по сути своей цифры числа, а порядок — величина, показывающая, в каком месте мантиссы надо поставить запятую.

Описанная выше система записи чисел называется представлением *с плавающей запятой*[11] (видимо, по той причине, что запятая "плавает" по мантиссе за счет порядка).

Некоторое неудобство вносит тот факт, что представление числа с плавающей запятой не является единственным:

$$3*10^8 = 30*10^7 = 0{,}3*10^9 = 0{,}03*10^{10} = \ldots$$

Поэтому договорились для выделения однозначного варианта записи считать, что

- мантисса ***всегда меньше единицы*** (иначе говоря, целая часть равна нулю);
- старший разряд мантиссы содержит ***отличную от нуля*** цифру.

Такое представление чисел называется *нормализованным* и является единственным — в нашем примере обоим требованиям одновременно удовлетворит только $0{,}3*10^9$. Любое число может быть легко нормализовано. Единственное, но важное исключение из правила составляет нуль: в его мантиссе нет ни одной ненулевой цифры. Ради такого важного случая было введено дополнительное соглашение: число с нулевыми мантиссой и порядком (все биты — нули) в качестве исключения считать нормализованным.

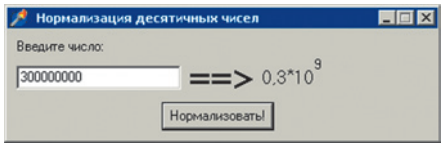


Рис. 7. Демоприложение "Нормализация чисел" (см. на диске)

Особо подчеркнем, что требования нормализации обеспечивают максимальную точность представления чисел. В частности, эта процедура эффективно убирает незначащие нули как из целой ($1000000 = 0{,}1*10^7$), так и из дробной ($0{,}0000001 = 0{,}1*10^{-6}$) частей.

Все эти хорошо известные *из математики*(!) рассуждения могут быть применены и к двоичной системе счисления, лежащей в основе компьютерного представления чисел:

$$A = (\pm M) * 2^{\pm P}$$

Например: $-3_{10} = -11*2^0 = -0{,}11*2^{10}$; отсюда $M = 0{,}11$ и $P = 10$.

В двоичной системе сформулированные выше требования нормализации приобретают новое, отсутствующее в десятичной системе, свойство. Согласно этим требованиям, первый разряд мантиссы не должен быть равен нулю, что приводит к выводу, что он *всегда равен единице* (других-то цифр в двоичной системе нет!).

[11] В англоязычных странах используется эквивалентный термин "плавающая точка" (`floating point`), так как там принято отделять дробную часть от целой точкой, а не запятой, как мы привыкли (а вы не задумывались, почему в Паскале полтора пишется как 1.5?).

Это позволяет при сохранении вещественных чисел в память не записывать заведомо известный старший разряд, а за счет освободившегося бита сохранить еще один дополнительный разряд мантиссы (разумеется, перед вычислениями в процессоре "отброшенная" "обязательная" единица восстанавливается). Такой метод хранения носит название *скрытая единица*.

В отличие от математики количество разрядов в изображении числа во всех вычислительных устройствах ограничено аппаратными возможностями (иначе говоря, разрядность хранения и обработки чисел конечна). При этом количество разрядов мантиссы (по сути, число знаков после запятой) влияет на точность вычислений, а разрядность порядка — на диапазон допустимых чисел. Мы подробнее изучим ограничения, которые возникают из-за конечной разрядности порядка и мантиссы вещественных чисел в экспериментах 6 и 7 соответственно.

И в заключение укажем на еще одну небольшую, но очень важную с практической точки зрения особенность устройства математического сопроцессора (без нее порядок всех переведенных "по большой науке" чисел всегда окажется на единицу больше). Дело в том, что при разработке сопроцессора было принято, что **нормализованное число имеет одну единицу в целой части мантиссы**[12] [9, 7], тогда как в "классическом" определении нормализации целая часть нулевая. Чтобы представить себе разницу, сравните две записи одного и того же числа; $0{,}1101*2^{101}$ и $1{,}101*2^{100}$. Сопроцессор использует второй вариант.

Таким образом, в основе машинного представления вещественных чисел лежат следующие важные теоретические положения.

- Вещественное число хранится в виде двух компонентов — мантиссы и порядка. В мантиссе хранятся цифры числа, а порядок указывает на положение запятой.
- Для выделения некоторого единственного представления числа используются требования нормализации.
- Поскольку в двоичной системе старшая цифра нормализованной мантиссы всегда равна единице, при хранении чисел в технических устройствах ее можно отбрасывать (метод "скрытой единицы").

3.2. Стандарт ANSI/IEEE 754-1985

Посмотрим теперь, как сформулированные теоретические принципы воплощаются в жизнь. Стандартом де-факто в этой области является Стандарт ANSI/IEEE 754–1985 [9], на базе которого, в частности, построен математический сопроцессор фирмы Intel. Как известно, сопроцессор этот является составной частью любого процессора, работающего внутри IBM-совместимого компьютера. (Попутный вывод: имеется "господствующая" система представления вещественных чисел, что повышает полезность ее изучения.)

Заметим, что в 2008 году ассоциация IEEE выпустила расширенный стандарт IEEE 754-2008 [10], который без изменений включил в себя стандарт IEEE 754–1985.

[12] Именно эта единица будет потом "скрыта"!

Согласно IEEE 754–1985, существует несколько форматов представления чисел. Все они показаны на *рис.* 8.

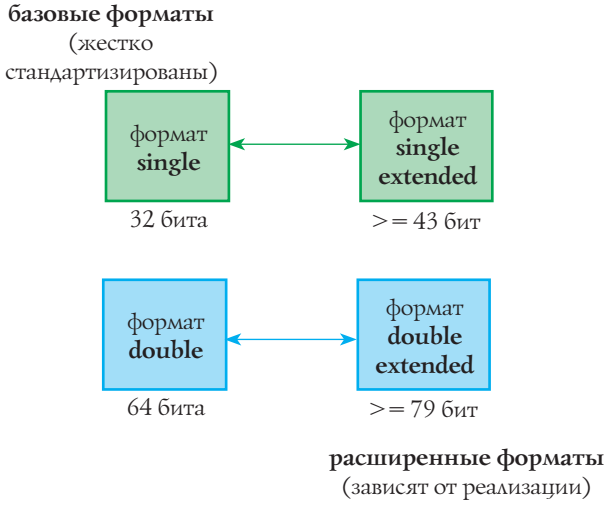


Рис. 8. Форматы вещественных чисел согласно Стандарту IEEE 754-1985

Базовые форматы служат для сохранения (и передачи) чисел. Для обеспечения максимальной совместимости данных базовые форматы стандартизированы очень жестко (вплоть до бита). Что касается расширенных форматов, то они служат в основном для временного представления при обработке чисел[13], поэтому они объявлены зависящими от конкретной реализации устройств (implementation-dependent) и для них в Стандарте описаны лишь самые общие требования.

Опираясь на полный перечень типов Стандарта IEEE 754, инженеры фирмы Intel при разработке своего математического сопроцессора упростили схему *рис.* 8 (не нарушая, разумеется, при этом Стандарта!). В итоге получилась схема взаимодействия типов, представленная на *рис.* 9.

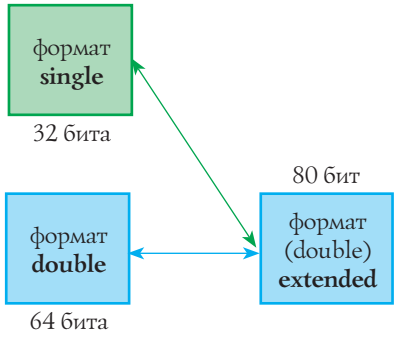


Рис. 9. Форматы вещественных чисел в математическом сопроцессоре фирмы Intel

Поскольку "младший" из расширенных форматов теперь упразднен, слово "double" в названии double extended становится излишним и постепенно отмирает.

Рассмотренная схема интересна для нашего обсуждения вот почему. Во-первых, она показывает, что существует два типа форматов: для *хранения* чисел (`single` и `double`) и для их *обработки* (`extended`). То есть эти две группы форматов "не совсем родные" и могут иметь отличия. В сопроцессоре Intel, как оказалось,

[13] При проведении вычислений математика для повышения точности рекомендует всегда использовать дополнительные разряды.

Тип числа	Всего битов (*B*)	Знаковый бит	Битов в порядке (*P*)	Битов в мантиссе (*M*)	Применение “скрытой единицы”
`single`	32	1	8	23	да
`double`	64	1	11	52	да
`extended`	80	1	15	64	нет

они действительно есть. Это должно повысить нашу подозрительность при чтении фраз типа “мы рассмотрели формат `single`, остальные форматы устроены совершенно аналогично” — совсем не обязательно, что *совершенно* аналогично. Во-вторых, тип `extended` в Стандарте описан очень поверхностно и для его подробного изучения надо обратиться к документации фирмы Intel (например, [11]). По моему мнению, перед нами возможные причины (хотя, может быть, и не все), по которым трудно найти хорошее и полное описание проблемы.

3.3. Вопросы разрядности

Согласно IEEE 754, любое вещественное число в форме с плавающей запятой состоит из трех компонентов: знака числа, порядка и мантиссы (*рис.* 10). Посмотрим, как распределяются биты между компонентами числа.

знак	порядок		мантисса	
B–1	B–2	*M*	*N*	0

Рис. 10. Структура числа с плавающей запятой

Пусть общее количество битов *B*, причем нумеруются они по традиции с нуля. Если обозначить номер самого старшего бита мантиссы *N*, то количество битов мантиссы $M = N + 1$. Количество битов порядка *P* при этом уже не может быть произвольным: оно вычисляется по формуле $(B–1) – M$. Распределение битов для всех типов рассматриваемых чисел представлено в таблице (на последний столбец пока не обращайте внимание). Напомним, что значений последней строки таблицы в IEEE 754 нет.

Знак всегда хранится в самом старшем бите и задается по правилу: 0 для положительных и 1 для отрицательных значений. При этом мантисса для положительных и для отрицательных значений кодируется одинаково (вспомним, что для отрицательных *целых* значений применялся специальный дополнительный код!).

Порядок также не лишен особенностей. Для того чтобы избавиться от знака, его “сдвигают”, прибавляя достаточно большое положительное число. В результате все отрицательные значения порядка исчезают, и он меняется от 0 до максимального значения[14]. Для чисел типа `double`, например, величина сдвига выбирается равной $1023_{10} = 3FF_{16}$ (10 единичек из 11 разрядов порядка). Примеры расчета порядка с учетом сдвига мы рассмотрим позднее в 3.4 и 3.6.

[14] Обычно для объяснения выбора именно такого метода хранения порядка приводятся весьма обтекаемые фразы о каких-то “технологических удобствах”; с теоретической точки зрения использование трех разных способов представления отрицательных чисел (целые числа, мантисса и порядок) изящным назвать трудно!

А теперь поговорим о последнем столбике таблицы, ибо он того заслуживает. В очередной раз вернемся к мысли о том, что IEEE 754 строго регламентирует только базовые стандарты `single` и `double`. В них метод “скрытой единицы” обязан применяться. Что же касается `extended`, то его детали оставлены на усмотрение разработчиков. Ну и как вы считаете, используется в формате `extended` “скрытая единица” или нет? Вот и я, честно говоря, до подготовки материалов статьи считал, что используется[15]. А оказывается — **нет**! И вывод этот у меня возник в ходе одного из экспериментов.

Сейчас, “задним числом”, я могу даже сказать, что сохранение всех разрядов мантиссы в рабочем формате `extended` даже объяснимо, поскольку это ускоряет вычисления... Зато неделю назад я верил тем, кто говорил, что “остальные форматы устроены совершенно аналогично”!

Эксперимент 5. Определение разрядности мантиссы

Проверим данные по распределению битов в вещественных числах, приведенные ранее в таблице. Особый интерес вызывает проверка разрядности мантиссы, поскольку размер порядка после этого однозначно определяется. Данный эксперимент кажется мне одним из наиболее изящных среди описанных в статье.

Постановка задачи. Экспериментально определить двоичную разрядность мантиссы для всех трех рассматриваемых типов вещественных чисел. Удобнее переформулировать задачу так: найти номер старшего бита мантиссы *N*. Если следовать общепринятой традиции нумерации битов с нуля, тогда разрядность мантиссы $M = N + 1$ (см. *рис.* 10).

Реализация эксперимента. Программа, решающая поставленную задачу, с некоторыми сокращениями приведена в листинге 5 (полный текст можно найти на диске). Исключена та часть, которая соответствует типу `double`, поскольку она абсолютно аналогична разобранной части для `single`.

Листинг 5

```
var s: single;
    sb: integer absolute s;
    smask,n: integer;
{здесь стояло несколько описаний
 для типа double}
    dmask: int64; {для double и extended}
    ex: extended;
    exb: int64 absolute ex;
```

[15] Один мой коллега часто говорил, что лучший способ изучить предмет — это начать его преподавать.

```
{мантисса у single}
  s := 0.75;
  {старший сохраненный бит мантиссы — 1,
   остальные — 0}
  smask := 1; n := 0;
  while (sb and smask) = 0 do
        begin smask := smask shl 1;
              n := n + 1;
        end;
  writeln('single: bit N ',n);
{здесь стояла аналогичная программа
 для типа double;
 ее можно посмотреть на диске!}
{мантисса у extended}
  ex := 0.5;
  {старший бит мантиссы — 1, остальные — 0}
  dmask := 1; n := 0;
  while (exb and dmask) = 0 do
        begin dmask := dmask shl 1;
              n := n + 1;
        end;
  writeln('extended: bit N ',n);
  readln;
```

Разбор начнем с описательной части. Она отличается от всех предыдущих экспериментов тем, что на вещественные числа накладывается не массив из байтов, а подобранные по длине целые числа. Так, переменная `s` типа `single` совмещена с переменной `sb` типа `integer`, так как оба типа 32-разрядные. Аналогичным образом `double` хорошо совмещается с `int64`. Труднее с типом `extended`: он 10-байтовый, а целого с такой разрядностью в Delphi нет[16]. Воспользуемся за неимением лучшего `int64`. Подумаем, какую часть переменной `ex` он "накроет"? Младшие 8 байт, т.е. как раз всю ожидаемую мантиссу! (Надеюсь, вы не забыли об эффекте little-endian, благодаря которому байты в ОЗУ лежат "задом наперед", так что мантисса оказывается *раньше* порядка?) Итак, если мантисса не превысит 64 бит (а она теоретически не должна), наше решение сработает.

При объяснении программы, оказывается, гораздо проще начать обсуждение с последнего типа — `extended`, причем именно благодаря тому, что там "никто никуда не прячет" старший бит мантиссы. Напомню, мы хотим определить его номер.

План действий таков. Присвоим вещественной переменной значение $0{,}5_{10} = 0{,}100..00_2$. Очевидно, что его двоичная мантисса содержит все нули и единицу в старшем бите. Остается только найти номер этого бита.

Нам удалось свести задачу к хорошо известной. Она легко решается с помощью константы-маски с единственным ненулевым битом `dmask` и логической операции И. Из алгебры логики известно, что если в маске бит единичный, то результат равен значению бита числа, на которое эту маску накладывают. Для нулевого бита маски результат всегда нулевой и никак не зависит от исходного числа. Короче говоря, описанная операция выделит из всего кода именно тот бит, для которого в маске установлена единица.

[16] Вообще 10-байтовый тип в 64-битовом компьютере порождает много вопросов!

Первоначально маске `dmask` присваивается значение 1, что настраивает ее на самый младший (самый правый, с `n = 0`) бит. Далее маска циклически накладывается на число `exb` и каждый раз проверяется равенство результата нулю (т.е. найдена единица или нет). Если бит оказался нулевым, то маска сдвигается на один разряд влево, номер бита *n* увеличивается на единицу и процесс повторяется еще раз. Так продолжается до тех пор, пока не будет обнаружена единица.

Подчеркнем, что операция И не может применяться к вещественным числам, так что если бы не наш трюк с **`absolute`**, ничего бы вообще не вышло!

Вернемся теперь к началу программы и поговорим о типе `single`. Здесь описанный выше алгоритм для числа 0,5 применить не удастся, поскольку старший бит мантиссы скрыт — просто нечего искать! Тем не менее, достаточно взять другое число — $0{,}75_{10} = 0{,}110..00_2$, и все сразу встанет на свои места: в мантиссе у 0,75 после того, как старший бит будет скрыт, останется единственная единица, причем она опять окажется в старшем бите.

Программа для `double` устроена совершенно аналогично, изменятся только типы переменных. Особо подчеркнем, что алгоритм для `single` и `double` ищет размер "сохраняемой" мантиссы, т.е. той, которая хранится в ОЗУ. Реальная мантисса числа на единицу ("скрытую") больше.

Запустив программу, получаем следующие результаты:

```
single: bit N 22
double: bit N 51
extended: bit N 63
```

Все значения полностью согласуются с приведенными ранее в таблице, если учесть, что $M = N + 1$.

Вывод. Полученные экспериментальным путем величины двоичной разрядности мантисс совпадают с приведенными в документации значениями.

3.4. Границы диапазона вещественных чисел

А помните, во введении речь шла о проблеме "самого маленького положительного вещественного числа", которое можно сохранить в сопроцессоре и которое по разным данным имело самые разные значения? Давайте попробуем разобраться с этим. Как оказывается, тут есть о чем поговорить!

Мы уже знаем, что диапазон представления чисел зависит от разрядности порядка. Кроме того, нам известно, что порядок хранится со сдвигом, так что сохраняемое в ОЗУ значение порядка $E_s = E + S$, где E — значение порядка числа, а S — величина смещения, определяемая по формуле $S = 2^{p-1} - 1$. Если внимательно проанализировать последнюю формулу, то станет понятным, что смещение S равно примерно половине максимального значения порядка, так что в результате порядок меняется не от –Max до +Max, а от 0 до 2*Max (точнее будет сказать — до 2*Max + 1).

Характеристики порядка для разных вещественных типов приведены в следующей таблице.

Будем далее для примера рассматривать тип `single`; порядок для остальных типов устроен совершенно ана-

	single	double	extended
Разрядность порядка *P*, бит	8	11	15
Диапазон порядка (IEEE 754)	–126..127	–1022..1023	–16 382..16 383
Смещение S_{10}	127	1023	(16383)[17]
Смещение S_2	111 1111	11 1111 1111	11 1111 1111 1111
Смещение S_{16}	7F	3FF	3FFF
Количество бит смещения	7	10	14

логично. Согласно IEEE 754, порядок для типа `single` может принимать значения от –126 до +127, так что смещенный порядок будет меняться от 1 до 254. Пусть, скажем, порядок некоторого числа равен 9. Тогда при сохранении в память его значение будет смещено и равно 9 + 127 = 136.

Из изучения двоичной системы мы четко помним, что беззнаковое положительное 8-битное число может принимать значения от 0 до 255. Между тем смещенный порядок, как мы видели, охватывает диапазон от 1 до 254. Оставшиеся два значения (0 и 255) являются служебными и обрабатываются особо. Они имеют самое непосредственное отношение к нашему разговору о диапазоне представления чисел.

Рассмотрим, что происходит с числами, когда они становятся очень большими. Очевидно, что порядок этих чисел растет и в конце концов достигает 127 (смещенный порядок при этом стремится к значению 254). Когда мантисса достигнет своего максимального значения (все единицы), то число уже нельзя увеличить даже на единицу, не попадая в "запрещенное" значение смещенного порядка 255. Ситуация, когда число настолько велико, что не может поместиться в рамках существующей разрядной сетки, называется *переполнением*. Таким образом, появление в порядке значения 255 является для сопроцессора признаком переполнения. При этом значение, у которого порядок равен 255, а мантисса нулевая, в IEEE 754 условно считается *бесконечностью*. Все же остальные числа с таким порядком обозначаются *NaN* (Not a Number — не число). Для работы с этими значениями в сопроцессоре приняты отдельные правила; мы не будем в них углубляться, а лучше поговорим об очень маленьких числах.

Итак, пусть теперь положительные числа уменьшаются. При этом их порядок становится все меньше и приближается к –126 (т.е. со смещением — к 1). В какой-то момент смещенный порядок равен 1, а мантисса содержит единственную единицу (по условиям нормализации — это старшая, она же — "скрытая" единица). Любая попытка еще уменьшить число (хотя бы на 1) приведет к появлению "запрещенного" нулевого значения в разрядах порядка. Казалось бы, результат придется обнулить, но тут инженеры Intel приготовили нам еще один сюрприз: числа с нулевым смещенным порядком продолжают обрабатываться, но по-другому! Поскольку порядок больше "не работает", то нормализацию приходится блокировать, а мантиссу обрабатывать "как есть". Фактически для очень маленьких чисел (по понятным причинам их называют *денормализованными* — нормализация не может использоваться!) сопроцессор переходит на другую модель вычислений: от *плавающей* запятой переходит к *фиксированной*. Это позволяет ему еще какое-то время "продержаться" и продолжать фиксировать ненулевые значения, но в конце концов мантисса обнулится, а число с нулевым порядком и мантиссой есть по определению машинный ноль.

Таким образом, мы получаем следующую последовательность чисел в сопроцессоре (от больших значений к меньшим):

> NaN — бесконечность — нормализованные числа — денормализованные числа — ноль

Наличие отрицательных значений позволяет симметричным образом продлить эту последовательность:

> –0 (минус ноль) — денормализованные и нормализованные числа — –бесконечность — –NaN

Минус ноль — это забавный казус: сопроцессор гарантирует, что –0 = +0.

Прекрасная схема перечисленных типов с указанием их границ для типа `single` приведена в статье [7] (см. *рис.* 11).

Остается только научиться вычислять границы между типами, причем не только для типа `single`, но и для остальных.

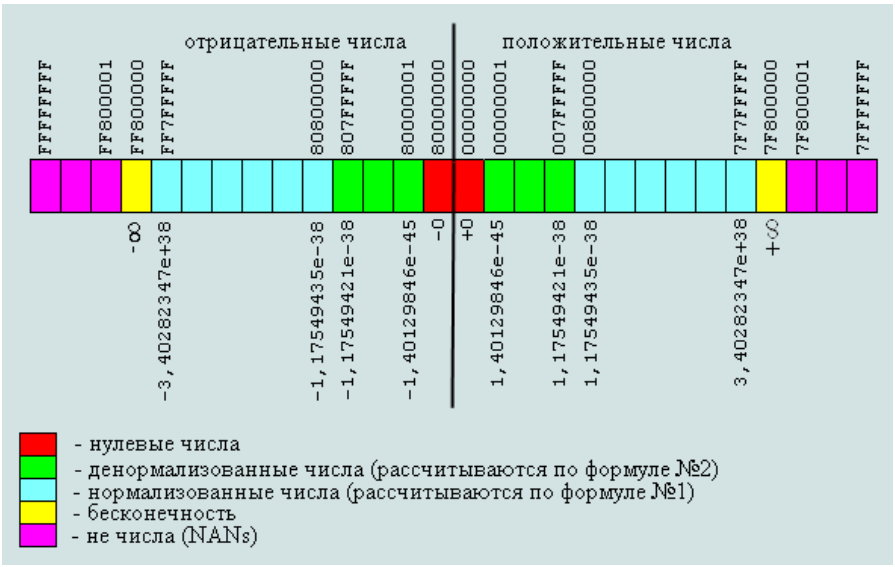


Рис. 11. Числа типа `single` в сопроцессоре [7]

[17] В IEEE 754–1985 не нормируется.

Эксперимент 6. Границы диапазонов для вещественных чисел

Постановка задачи. Научиться вычислять изображенные на *рис.* 11 границы чисел для всех трех рассматриваемых вещественных типов.

Реализация эксперимента. Любая задача почти всегда может быть решена несколькими способами. В частности, в статье [7] использован метод десятичных вычислений через степени двойки. Мне хочется предложить более простой метод, базирующийся на активно эксплуатируемой конструкции **absolute**.

Идея решения состоит в следующем. Все вычисляемые границы изначально пишутся в шестнадцатеричном виде, а нам надо перевести их в десятичный. Для этого совместим, как мы это уже неоднократно делали, вещественное число с байтовым массивом. Заполним массив необходимыми шестнадцатеричными значениями байтов, и нам останется просто напечатать вещественную переменную. Никаких формул и вычислений — сразу выводим ответ!

Решения для типов `single` и `double` одинаковы с точностью до числовых значений, так что в листинге 6 оставлена только та часть, которая относится к `single`.

Листинг 6

```
var s: single;
    sb: array [1..4] of byte absolute s;
{здесь стояли аналогичные описания
 для double}
    i:integer;

    writeln('single:');
{минимальное денормализованное значение}
  {00 00 00 01}
  for i := 4 downto 2 do sb[i] := 0;
  sb[1] := 1;
  writeln('min denormalized: ',s);
{максимальное денормализованное значение}
  {00 7F FF FF}
  sb[3] := $7F;
  for i := 2 downto 1 do sb[i] := $FF;
  writeln('max denormalized: ',s);
{минимальное нормализованное значение}
  {00 80 00 00}
  sb[3] := $80;
  for i := 2 downto 1 do sb[i] := 0;
  writeln('min normalized:   ',s);
{максимальное нормализованное значение}
  {7F 7F FF FF}
  sb[4] := $7F; sb[3] := $7F;
  for i := 2 downto 1 do sb[i] := $FF;
  writeln('max normalized:   ',s);
{здесь была аналогичная программа
 для типа double;
 ее можно посмотреть на диске!}
  readln;
```

Посмотрим, как устроен поиск границ. Самое маленькое денормализованное число, граничащее с нулем, есть единица в младшем разряде (00 00 00 01 для `single`). Самое большое денормализованное, напротив, содержит в мантиссе все единицы (но смещенный порядок остается нулевой: 00 7F FF FF). При переходе к нормализованным числам имеем порядок 1, а в мантиссе — единственную единицу, которую сопроцессор "скрывает"; проще говоря, сохраненная в память мантисса будет нулевой (00 80 00 00). Наконец, максимальное нормализованное число — это порядок $254_{10} = FE_{16}$[18] и мантисса из единиц (7F 7F FF FF).

Для типа `double` из-за различий в разрядности константы будут другие: 00 00 00 00 00 00 00 01, 00 0F FF FF FF FF FF FF, 00 01 00 00 00 00 00 00 и 7F EF FF FF FF FF FF FF соответственно. Сама программа устроена аналогично и поэтому в листинге не приведена.

После запуска программы на экране появятся следующие результаты.

```
single:
min denormalized:  1.40129846432482E-0045
max denormalized:  1.17549421069244E-0038
min normalized:    1.17549435082229E-0038
max normalized:    3.40282346638529E+0038

double:
min denormalized:  4.94065645841247E-0324
max denormalized:  2.22507385850720E-0308
min normalized:    2.22507385850720E-0308
max normalized:    1.79769313486232E+0308
```

Сразу подчеркнем полное согласие полученных значений для `single` с приведенными на *рис.* 11 данными [7].

Отчетливо видно, что в качестве абсолютного минимума следует брать денормализованное значение. Денормализованный максимум и нормализованный минимум почти одинаковы, но последнее значение чуть-чуть больше. Между прочим, разница между ними не какая-нибудь, а равна именно абсолютному минимальному значению (см. на диске проект cmp_max_min). Таким образом, минимальное денормализованное значение оказывается важной характеристикой. С другой стороны, абсолютный максимум ("значение перед самой сопроцессорной бесконечностью") есть максимальное нормализованное число.

Таким образом, границы типа `single` оказываются приблизительно равными $1{,}4*10^{-45}..3{,}4*10^{38}$, а для `double` — $4{,}9*10^{-324}.. 1{,}8*10^{308}$. Полученные степени полностью совпадают с приведенными в [8], хотя коэффициенты почему-то чуть-чуть отличаются. Еще раз подчеркнем, что наши коэффициенты очень хорошо совпадают с результатами [7].

Перейдем теперь к типу `extended`.

Главное отличие — это то, что старший бит мантиссы в граничных значениях требуется хранить. Поэтому программа для этого типа данных была написана отдельно (см. проект `extended` на диске), а значения шестнадцатеричных констант в ней были следующими: минимальное денормализованное значение 00 00 00 00 00 00 00 00 00 01, максимальное — 00 00 7F FF FF FF FF FF FF FF, минимальное нормализованное — 00 01 80 00 00 00 00 00 00 00 и, наконец, максимальное — 7F FF FE FF FF FF FF FF FF FF.

Ответы следующие:

```
extended:
min denormalized:  3.64519953188247E-4951
max denormalized:  3.36210314311209E-4932
min normalized:    3.36210314311209E-4932
max normalized:    1.18973149535723E+4932
```

[18] Учтите, что при сохранении в биты порядка данное значение будет сдвинуто на один разряд вправо (см. *рис.* 10).

Полученный диапазон чисел: $3{,}6*10^{-4951}..1{,}2*10^{4932}$. Сравнивая с таблицей [8], видим, что там в качестве нижней границы указано не минимальное ненормализованное, а минимальное нормализованное значение. Нет сомнения в том, что в Borland хорошо понимают, что такое денормализованные числа: просто Турбо Паскаль с ними не работает! (Подробнее об этом см. также в 3.7.)

Кстати, значение степени, равное –4951, хорошо согласуется с оценкой в книге [1].

Чтобы быть еще больше уверенным в правильности вычислений, отдельные значения я пересчитал на встроенном в Windows калькуляторе. Оказалось, его точность выше(!), чем у Delphi и, соответственно, математического сопроцессора[19]. Вот, например, результат вычисления минимального нормализованного числа `extended`: $2^{-16382} = 3{,}3621031431120935062626778173218e^{-4932}$ (внушает?), т.е. 32 знака против 15 в Паскале. Зато все эти 15 совпадают в обоих ответах!

Выводы. В математическом сопроцессоре существуют две нижних границы для диапазона вещественных чисел — нормализованная и денормализованная (последняя ниже); денормализованные числа могут не поддерживаться программным обеспечением. Полученные результаты эксперимента тщательно проверены и имеют хорошую точность. Самое маленькое вещественное число приблизительно равно $3{,}6*10^{-4951}$.

3.5. Точность представления вещественных чисел

В предыдущем примере нам удалось экспериментально определить диапазон представления чисел, используя знание разрядности порядка. Аналогичным образом стоит изучить и разрядность мантиссы, что позволит оценить точность вычислений каждого из вещественных типов в десятичных знаках.

Прекрасная идея эксперимента была описана в недавней публикации [6]. Его суть состояла в следующем. Берем вещественное число, равное 1, и прибавляем к нему некоторую добавку, вычисляемую по формуле 2^{-n}: ½, ¼, ... При этом бинарная мантисса числа ведет себя так: 1,1, 1,01, ..., 1,00..01. Очевидно, что в какой-то момент младшая единица "вылезет" за разрядную сетку и "потеряется". Сумма перестанет отличаться от единицы, что и зафиксирует программа.

В [6] было продемонстрировано, что погрешность вычислений не зависит от типа вещественных данных, поскольку все они ведутся в формате `extended`. Немного модифицировав предложенную авторами программу, все-таки можно измерить разницу в точности использования вещественных переменных разных типов: для этого надо обязательно *сохранять* результат суммирования в память. Думается, эксперимент 7 стоит сделать хотя бы для того, чтобы избежать неверного вывода о том, что для всех типов вещественных переменных погрешность одинакова (если бы это было так, то все бы пользовались самым коротким типом `single`!).

[19] Я вижу только одно объяснение этому факту: внутри калькулятора программным путем реализована собственная многоразрядная арифметика!

Эксперимент 7. Определение десятичной разрядности мантиссы

Постановка задачи. Определить, с какой погрешностью работают программы, построенные на каждом из рассматриваемых вещественных типов данных. Результат погрешности вычислить в десятичной системе.

Реализация эксперимента. Фрагмент модифицированной программы из [6], относящийся к типу `single`, воспроизведен в листинге 7. Программа была реализована в системах Free Pascal (файл epsilon2.pas на диске) и Delphi (проект также есть на диске).

Листинг 7

```
program for_Free_Pascal
var   s,es:single;
      d,ed:double;
      e,ee,n:extended;
begin
  n := 1; {по n был цикл}
  writeln('n=',n:2:0); writeln('single');
  s := 1;
  repeat s := s/2; es := n + s
  until es = n;
  writeln(s/n);
{здесь были аналогичные программы для
double и extended;  их можно посмотреть
на диске}
  readln;
end.
```

При запуске программы на экране появились следующие результаты:

```
n = 1
single
 5.9604644775390625E-0008
double
 1.1102230246251565E-0016
extended
 5.4210108624275222E-0020
```

Видно, что точность вычислений с применением переменных, разумеется, зависит от разрядности вещественного типа (часть разрядов теряется при сохранении чисел в ОЗУ). Хочется отметить прекрасное согласие полученных результатов с уже упоминавшейся ранее таблицей из [8]. Там, в частности, сказано, что тип `single` гарантирует 7–8 значащих цифр, `double` — 15–16, а `extended` — 19–20.

Отдельно надо сказать о роли переменной `n`. С ее помощью можно прекрасно продемонстрировать, что относительная погрешность вычислений с плавающей запятой постоянна [6]. Вычисления показывают, что при `n` = 1, 2, 4, ... (степени двойки) отношения погрешности к величине числа `n` получаются абсолютно одинаковыми. Да и как может быть иначе, если фактически речь идет о числах с идентичной мантиссой, различающихся только порядком!

Возможно, некоторые читатели спросят: а почему бы и здесь не использовать нашу "абсолютную" техно-

логию (в смысле, с оператором **absolute**)? Как сейчас принято говорить в молодежных кругах — легко!

Для этого достаточно составить шестнадцатеричное представление числа 1,0 и в самом последнем его бите дописать единицу. Разность между этим числом и единицей и есть искомый результат.

К сказанному остается только добавить, что в формате `single` вещественная единица имеет код 3F 80 00 00, `double` — 7F F0 00 00 00 00 00 00, и, наконец, в `extended` — 3F FF 80 00 00 00 00 00 00 00.

Фрагмент программы, который относится к типу `single` (для остальных типов программа устроена аналогично), приведен на листинге 8. Как обычно, "желтую" описательную часть вставляем в текст проекта до `begin`, а операторную — после него.

Листинг 8

```
var s: single;
    sb: array [1..4] of byte absolute s;
{здесь стояли описания для double и extended}
```

```
  writeln('single:');
  {3F 80 00 01 единица "с хвостиком"}
  sb[4] := $3F; sb[3] := $80; sb[2] := 0;
  sb[1] := 1;
  writeln(s-1);
{здесь были аналогичные программы для
double и extended; их можно посмотреть на
диске!}
```

Запускаем и получаем те же самые... А вот и нет! Результаты оказываются ровно вдвое больше:

```
single:
 1.19209289550781E-0007
double:
 2.22044604925031E-0016
extended:
 1.08420217248550E-0019
```

Впрочем, неожиданно это только на первый взгляд. На самом деле это понятный эффект, связанный с устройством алгоритмов листингов 7 и 8. В первом из них печатается первое число, для которого не была зафиксирована разница с единицей[20], а во втором — последнее, которое еще отличается от единицы.

Полученные степени сравним по порядку величины со степенями "исчезновения порядка" в предыдущем эксперименте: последние значительно выше! Это означает, что ограничение точности, связанное с мантиссой, проявляется существенно раньше, чем для порядка. Страшно ли это для вычислений? Вспомните, к примеру, типовые задачи, которые вы решали на уроках физики. Как правило, вычисления там ведутся по формулам, которые представляют собой дробь с произведениями величин в числителе и знаменателе. Из математической теории известно (см., например, [3]), что погрешность в такой ситуации определяется *исключительно числом с минимальным количеством знаков после запятой*! Проще говоря, если в одном из чисел достоверно известны только 2 знака, а у всех остальных — 8, в ответе все равно будут верными только 2 цифры после запятой. А теперь вспомните — сколько знаков вы используете в вычислениях на физике? Едва ли больше 3–4, а то и вовсе 1–2! Так что погрешность мантиссы в самом плохом случае в восьмом десятичном знаке едва ли вас должна настораживать. Зато диапазон значений вам требуется немалый: двузначный десятичный порядок — это не исключение, а, скорее, правило в микромире (отрицательные порядки) и в астрономических масштабах (положительные). Таким образом, оказывается, что принятое в компьютере распределение разрядов под мантиссу и порядок вполне разумно.

Выводы. Точность хранения в памяти переменных разных типов различна, хотя вычисления в сопроцессоре для всех ведутся с одинаковой точностью. Полученные в эксперименте 7 значения хорошо согласуются с известными из других источников данными.

3.6. Как перевести число?

В предыдущих разделах мы подробно изучили общие принципы представления вещественных чисел в компьютере. Видимо, настала пора применить эти принципы на практике. Сначала попробуем на листке бумаги, а потом перейдем к программному обеспечению.

Допустим, требуется представить в двоичном виде вещественное число –17,25. Выберем для простоты тип `single` как имеющий самый короткий по длине записи формат.

Прежде всего переведем модуль числа, т.е. 17,25, в двоичную форму, как этому учит математическая теория: отдельно целую и отдельно дробную части. В итоге получим:

$$10001{,}01$$

Нормализуем полученное число. Для этого, согласно общепринятому определению, потребовалось бы передвинуть запятую на $5_{10} = 101_2$ разрядов влево, но в сопроцессоре принято старшую единицу выносить в целую часть (см. 3.1), поэтому получим:

$$10001{,}01*2^0 = 0{,}1000101*2^{101} = 1{,}000101*2^{100}$$

Остается сформировать мантиссу, "скрыв" старшую единицу:

$$M = 0{,}\mathbf{000101}$$

Как мы помним, в отличие от целых чисел, где отрицательные числа представляются в дополнительном коде, мантисса вещественных чисел хранится в прямом коде. Следовательно, больше никаких преобразований мантиссы не требуется.

Знаковый разряд необходимо установить в 1, поскольку число отрицательное: $Z = \mathbf{1}$.

Теперь остается закодировать двоичный порядок 100. Для хранения порядка в числах типа `single` смещение равно 7F (см. таблицу в 3.4), так что смещенный порядок будет равен

$$E_S = 100 + 111\ 1111 = \mathbf{1000\ 0011}$$

Собирая теперь Z, E_S и M в единое 32-разрядное число, окончательно получаем:

$$1\ \mathbf{10000011}\ 0001010\ 00000000\ 00000000$$

[20] Исправление `es := n + s/2` в листинге 7 полностью согласует результаты двух методов, но делает алгоритм менее наглядным.

Или более компактно в шестнадцатеричной системе — **C1 8A 00 00**. Этот код и будет записан в ОЗУ, правда, байты при этом, как мы знаем, будут переставлены.

Не думаю, что описанная процедура сложна по своей сути, но технической работы требуется много. А значит, велика вероятность ошибиться. Поэтому для проверки неплохо бы иметь специальное учебное программное обеспечение, способное для введенного десятичного числа выдать его внутренне представление и, наоборот, расшифровать прочитанные из ячеек ОЗУ значения. Этим мы сейчас и займемся.

3.7. Инструмент для изучения представления вещественных чисел

Я очень боюсь, что разочаровал читателей. В заголовке стоит Delphi, и многие, вероятно, ожидали многочисленных окон, кнопок, флажков и прочих визуальных объектов. А вместо этого им предложили многочисленные консольные приложения в черном окне, которое многие сейчас воспринимают как призрак умирающей MS-DOS. Еще раз подчеркну, что все рассмотренные ранее примеры не требовали практически никакой интерактивности и просто выводили на экран некоторые числа. Кроме того, как случайно оказалось (и скоро мы это увидим), в графических приложениях менее точно, чем в консольных, реализована работа с числами типа `extended`. А это уже для нас принципиально.

Но тем не менее обманывать ожидания читателей нехорошо. К тому же мы подошли к достойной задаче, в которой возможностью удобно вводить данные пренебрегать нельзя. И сейчас мы наконец(!) напишем настоящее графическое приложение.

Учебная программа “FPview”

Постановка задачи. Реализовать учебную программу, которая по введенному десятичному числу выдает его внутренне представление и, наоборот, расшифровывает введенный шестнадцатеричный код в виде вещественного числа.

Проект. Для простоты сначала реализуем простейшую версию программы. Назовем ее **FPview_micro** — просмотрщик вещественных (`floating-point`) чисел. Эту версию программы мы разберем, хотя и с некоторыми сокращениями. Зато более мощную (миниверсию) мы обсудим только с пользовательских позиций. Желающие смогут заглянуть в “исходники” этого проекта на диске и изучить его.

Итак, создадим настоящее Windows-приложение (application) и сконструируем для него простейшую заготовку окна (форму `Form1`), примерный вид которой приведен на *рис.* 12.

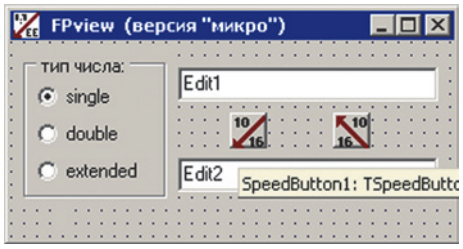


Рис. 12. Проект формы проекта

На форме созданы следующие компоненты.

Во-первых, два поля редактирования — `Edit1` и `Edit2`, в которые будем вводить десятичное число или шестнадцатеричный код соответственно. В этих же полях будут отображаться результаты перевода. Например, когда мы введем в `Edit1` интересующее нас десятичное число, приложение в `Edit2` выведет его машинное представление и наоборот.

Во-вторых, предусмотрим две кнопки типа `TSpeedButton` для перевода из одной формы в другую. Будем условно обозначать 10 →16 перевод из вещественного числа в машинное представление, а 16 → 10 — обратный процесс.

Наконец, слева на форме находится переключатель типов чисел `RadioGroup1`, на котором перечислены все три изучаемых нами типа.

Учитывая простоту проекта, в нем будет всего два обработчика событий — на каждую из кнопок.

Перейдем к чтению и анализу текста программы. С некоторыми сокращениями он приведен на листинге 9. Сокращению подверглись в основном второстепенные с точки зрения основного алгоритма функции и процедуры. Я надеюсь, что функция, которая преобразует число типа `byte` в два шестнадцатеричных символа, и обратная ей — это не то, что стоит сейчас подробно обсуждать. К тому же недавно появившееся в газете приложение в виде компакт-диска — это спасательный круг для меня как для автора: теперь, поместив полный текст программы на диске, я могу больше не беспокоиться о том, что без не попавших на страницы газеты кусков программа работать не будет!

Листинг 9

```
type  float_type = (fs,fd,fe); {single, double, extended}
{количество байт в представлении типов}
const NB: array [float_type] of integer = (4, 8, 10);
var   s: single;
      sa: array [1..4] of byte absolute s;
      d: double;
      da: array [1..8] of byte absolute d;
      e: extended;
      ea: array [1..10] of byte absolute e;
      errFlag: boolean; {флаг ошибок при переводе 16 ==> 10}
function MyByteToHex(n:byte):string;
{выдает 16-ричный код заданного байта}
```

```
begin {текст функции см. на диске}
end;
procedure TForm1.SpeedButton1Click(Sender: TObject);
{перевод 10 ==> 16}
var   w: array [1..10] of byte; {копия байтов числа}
      hex,sobr: string;
      i,n: integer;
begin case radioGroup1.ItemIndex{single, double, extended} of
        0: begin s := StrToFloat(edit1.Text); {введенное число}
                 n := Nb[fs]; {количество байт}
                 {копируем байты числа в рабочий массив}
                 for i := 1 to n do w[i] := sa[i];
           end; {0}
        1: begin d := StrToFloat(edit1.Text);
                 n := Nb[fd];
                 for i := 1 to n do w[i] := da[i];
           end; {1}
        2: begin e := StrToFloat(edit1.Text);
                 n := Nb[fe];
                 for i := 1 to n do w[i] := ea[i];
           end; {2}
      end; {case}
      sobr := '';
      for i := 1 to n do {побайтная обработка}
          begin {перевести в 16-ричный код очередной байт}
                hex := MyByteToHex(w[i]);
                {значения байтов в обратном порядке}
                sobr := hex + ' ' + sobr;
          end;
      edit2.Text := sobr;
end;

function MyHexToByte(s:string):byte;
{выдает целое значение, соответствующее 16-ричному коду байта}
begin {текст функции см. на диске}
end;

function check_length(var w: string; b: integer):boolean;
{проверяет длину текста, сравнивая с удвоенным числом байт;
 дополняет нули слева или обрезает строку, всегда спрашивает согласия пользователя}
begin {текст функции см. на диске}
end;

procedure TForm1.SpeedButton2Click(Sender: TObject);
{перевод 16 ==> 10}
var   w1,w2: string;
      i,n: integer;
      t: float_type;
begin w1 := edit2.Text;
{удалить пробелы}
      while pos(' ',w1) <> 0
            do delete(w1,pos(' ',w1),1);
{перебираем типы, пока номер не совпадет с itemIndex}
      t := fs;
      while ord(t) <> radioGroup1.ItemIndex
            do t := succ(t);
      n := Nb[t]; {количество байт для данного типа}
{проверим длину}
      errFlag := not check_length(w1,n);
      if errFlag then
         begin Edit1.Text := ('Terminated by user'); exit
         end;
{пакуем байты из строки в память в обратном порядке}
      for i := 1 to n do {перебираем байты}
          begin {берем очередной и удаляем из исходной строки}
                w2 := copy(w1,1,2); delete(w1,1,2);
                case t of
```

```
                    fs:begin {пакуем в память}
                             sa[n + 1 - i] := MyHexToByte(w2);
                             {отображаем результат}
                             edit1.Text := FloatToStr(s);
                       end;
                    fd:begin da[n + 1 - i] := MyHexToByte(w2);
                             edit1.Text := FloatToStr(d);
                       end;
                    fe:begin ea[n + 1 - i] := MyHexToByte(w2);
                             edit1.Text := FloatToStr(e);
                       end;
                end;
            end;
        if errFlag then edit1.Text := ('Error!')
end;
```

Программа начинается с описаний исследуемых переменных `single`, `double` и `extended`, на которые, как мы это уже неоднократно делали, наложены байтовые массивы соответствующей длины. В первый момент я хотел совместить их сразу все, но затем осторожность взяла верх, и я объединил их попарно.

Далее в тексте программы помещен обработчик первой кнопки, при нажатии на которую происходит перевод 10 → 16.

В соответствии с состоянием `RadioGroup1` (а точнее, со значением номера нажатой радиокнопки `ItemIndex`) из трех ветвей алгоритма выбирается та, которая соответствует выбранному вещественному типу. Пусть, например, выбран тип `single`, которому соответствует нулевое значение индекса. Тогда считанная с поля редактирования `Edit1` строка преобразуется в значение числа стандартной функцией `StrToFloat()` и сохраняется в переменную `s`. Заметим, что в использованной функции уже заложен контроль возможных ошибок в записи числа, так что никаких дополнительных действий с нашей стороны не требуется. Как только переведенное значение попало в переменную, можно сразу же пользоваться содержимым байтового массива `sa`, с ней совпадающим. Для того чтобы в дальнейшем работать с одним массивом, а не выбирать все время нужный из трех, скопируем все `n` байтов массива `sa` в рабочий массив `w`. После этого все три ветви алгоритма "сливаются" в одну общую.

Завершающая часть обработчика преобразует набор целых чисел из массива `w` в текстовую строку `sobr`, куда шестнадцатеричные эквиваленты байтов заносятся *в обратном порядке* ("компенсация" эффекта little-endian). Окончательную строку заносят в свойство `Text` компонента `Edit2`, и пользователь видит результат перевода на экране.

Перейдем теперь ко второму (обратному) обработчику. Он начинается с того, что считывает исходную строку из свойства `Text` компонента `Edit2` в переменную `w1`, с которой программа в дальнейшем и работает. Сначала производится целый ряд несложных, но полезных вспомогательных действий. В частности, из строки удаляются все пробелы, что уменьшает вероятность появления ошибок из-за невнимательности пользователя. Далее располагается немного "непрозрачный" цикл, суть которого крайне проста: перевести числовой номер `RadioGroup1.ItemIndex` в значение типа `fs`, `fd` или `fe`. Для этого с помощью функции `succ()`, т.е. следующий, они по очереди перебираются и номер текущего значения сравнивается с `ItemIndex`. В момент совпадения (а его не может не быть, поскольку и радиокнопок, как и типов, ровно три!) цикл немедленно завершается и в переменной `t` сохраняется наименование выбранного пользователем типа. Зная это значение, можно считать из массива количество байт `Nb`, а также в дальнейшем обращаться к одной из трех переменных, `s`, `d` или `e`, которая имеет необходимый тип.

После этого вызывается еще одна проверочная процедура, цель которой — сравнить длину полученной строки и количество байт для выбранного типа данных. В идеале длина в точности равна удвоенному количеству байт, так как каждый из них записывается двумя шестнадцатеричными цифрами. Если оказывается, что реальная строка короче, компьютер предлагает пользователю автоматически дописать слева необходимое количество нулей. В противном случае предлагается "укоротить" строку до необходимой длины. В обоих случаях с помощью очень удобной функции `MessageDlg()` у пользователя запрашивается подтверждение на изменение строки. Пользователь может отказаться от предложения "испортить" его строку; в этом случае функция `check_length()` возвращает значение `false` и выполнение обработчика прерывается.

Если же с длиной все прошло удачно, программа продолжает свою работу. Из строки `w1` последовательно извлекается по две шестнадцатеричных цифры, которые преобразуются в целочисленное значение типа `byte` и пакуются в нужный массив (выбор массива соответствующего типа происходит по переменной `t`, о которой подробно рассказывалось выше). Главная "хитрость" данного цикла состоит в том, что заполнять массив приходится "с конца", с тем чтобы изменить порядок байтов на противоположный.

В ходе перевода какой-то из обрабатываемых символов может оказаться некорректным. В этом случае в одной из внутренних процедур переменная `errFlag` получит значение `false`. Последний оператор обработчика обеспечит в этом случае индикацию ошибки.

Вот так вкратце и устроена программа.

Запустим ее и протестируем. В качестве тестов можно использовать результаты, полученные ранее в 3.4 и 3.6. Кроме того, полезно заранее "вручную" перевести 2–3 значения чисел и потом их проверить. Все должно прекрасно получаться, кроме одного: любое денормализован-

ное число в режиме `extended` программа принципиально сбрасывает в ноль[21]! Очень любопытный результат! Как тут не вспомнить таблицу границ значений из [8], о которой мы уже столько говорили. В ней честно сказано, что нижняя граница `extended` совпадает с наименьшим нормализованным числом. То есть это не ошибка, а так задумано.

А как же все-таки в Delphi обстоит дело с ненормализованными значениями? С чувством глубокого удовлетворения сообщаю, что наша программа в состоянии помочь ответить на этот вопрос. Давайте дадим ей некоторое десятичное значение, скажем, 1e-4949, которое заведомо является денормализованным (проверьте по данным 3.4). Ответ, воспроизведенный на *рис.* 13, приятно радует: значение будет ненулевым! (Правда, при обратном переводе десятичное число немедленно обнуляется.) Таким образом, эксперимент убедительно показывает, что в памяти такие значения хранятся, но при выводе обнуляются. Тем более забавно, что в компилируемых в Delphi консольных приложениях оператор `WRITE` прекрасно справляется с выводом обсуждаемых значений на экран[22]!

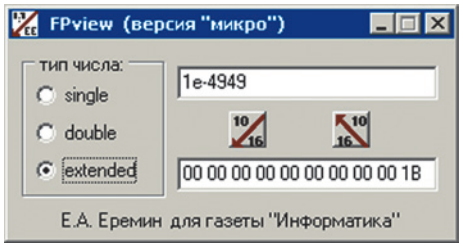


Рис. 13. А денормализованные значения все-таки есть!!!

Полученная в результате трансляции листинга 9 программа уже существенно помогает в изучении представления вещественных чисел. В частности, она позволяет проверить правильность "ручного" перевода при подготовке примеров, а также незаменима при составлении и проверке решения задач. Тем не менее выводимый шестнадцатеричный код нужно еще расшифровывать. Неплохо было бы, если бы программа помогала и здесь. Нужна новая, более мощная версия, способная делать это. И она тоже предлагается вашему вниманию, правда, на уровне пользовательского описания. Полный текст проекта содержится на диске. Я бы посоветовал заглянуть в текст проекта: если вам нравится программирование, вы можете найти там для себя кое-что интересное (скажем, массив из визуальных компонентов или широкое применение логических операций при формировании чисел).

Новая версия, вид окна которой показан на *рис.* 14, выглядит существенно привлекательнее, чем аскетичная версия "микро".

Сразу бросается в глаза, что большую часть окна в новой версии занимает панель, на которой отображаются результаты анализа кода числа. На ней можно прочесть:

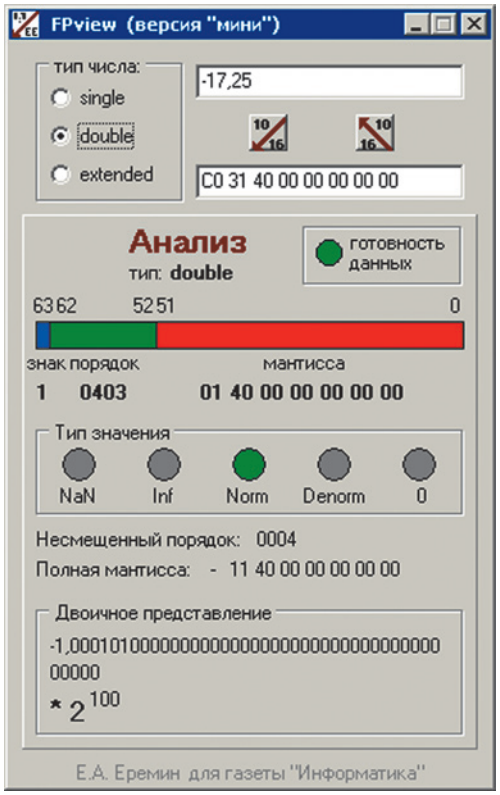


Рис. 14. Расширенная версия программы

- тип вещественного числа;
- распределение битов кода для этого типа;
- значения знакового бита и битов порядка и мантиссы, выделенные из кода введенного числа;
- тип значения (NaN, бесконечность, нормализованное, денормализованное или машинный ноль);
- истинное значение порядка, полученное после вычитания смещения;
- полное значение мантиссы, включая восстановленную "скрытую единицу" (если она была скрыта);
- окончательно расшифрованное двоичное число.

На *рис.* 14 изображено окно с информацией о числе –17,25 формата `double` (ранее в 3.6 оно переводилось в `single`). Легко убедиться, что отображаемое в нижней части окна расшифрованное двоичное число действительно соответствует результатам перевода указанного десятичного числа в двоичную систему счисления.

Пара слов по поводу "таинственного" индикатора готовности данных в правой верхней части панели. Все данные панели автоматически отображаются после нажатия любой из двух кнопок перевода. Представим теперь, что мы изменили состояние радиокнопок или начали изменять содержимое поля редактирования с десятичным числом. Данные на панели анализа пока сохраняются без изменения. В результате содержимое верхней и нижней частей окна перестает соответствовать друг другу, о чем будет напоминать пользователю красный цвет индикатора. После завершения ввода изменений и нажатия на любую из кнопок перевода индикатор снова "по-

[21] Excel, кстати говоря, поступает так гораздо раньше — уже при 1e-308.

[22] Маленькая, но важная деталь: оператор `write` в консольных приложениях при выводе разделяет целую и дробную части точкой, а функция `FloatToStr()` в оконном приложении использует национальные настройки, т.е. выдает запятую!

зеленеет", что будет свидетельствовать об обновлении данных на панели анализа[23].

Возможно, читателей заинтересовало, почему версия названа "мини" и чего же еще в ней не хватает. Как показало тестирование, Delphi не всегда корректно обрабатывает "неправильные" значения двоичных кодов, занесенные в качестве вещественных значений. Например, программа "ругается" на значение single-NaN, равное FF 90 00 00, или некорректное extended-значение 40 00 00 00 00 00 00 00 00 00 (в нем отсутствует "нескрываемая" единица в мантиссе, чего быть не должно). К тому же, как указывалось ранее, Delphi не поддерживает ненормализованные extended-значения, принудительно их обнуляя при выводе. И хотя все это весьма второстепенные ограничения, полноценная программа все это должна как-то делать. Еще возникало желание добавить возможность ввода двоичного числа в виде $X * 2^Y$ в нижней части окна. В общем, нет в мире совершенства...

Возможно, вам покажется интересным сравнить возможности описанного в статье инструмента с уже имеющимися, например, с online-версией конвертера [12].

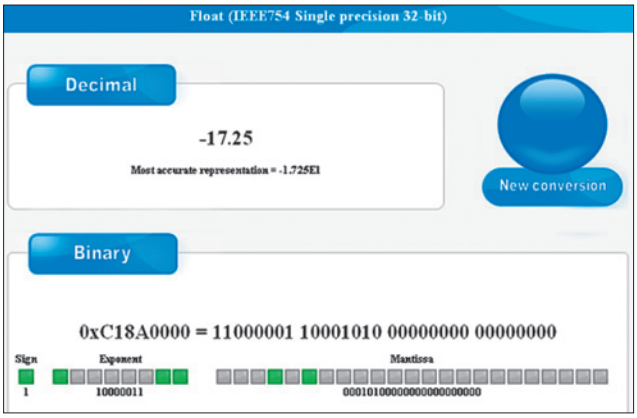


Рис. 15. Online-конвертер для вещественных чисел [12]

3.8. "Бонусный" эксперимент

Часто в компьютерных играх после прохождения серии трудных уровней предлагается так называемый "бонус" (призовая стадия, которая позволяет расслабиться, прежде чем двигаться дальше). Мне хочется предложить тем, кто добросовестно разбирался во всех хитросплетениях вещественных чисел, проделать следующее простое и забавное упражнение. Откройте стандартный калькулятор Windows (в инженерном режиме) и попробуйте получить вещественное число, которое изображено в копии экрана на *рис.* 16.

Трудность в том, что набор порядка ограничен четырьмя цифрами, после чего компьютер начинает "возмущенно пищать" и игнорирует дальнейший набор. Не сомневаюсь, что, используя свои математические знания, читатели легко справятся с предложенным "бонусным" упражнением. И тем, кто это сделает, — "призовая картинка". Возведите число, изображенное на экране, в квадрат 5–6 раз. Забавное диалоговое окно, которое вы

[23] В Delphi подобный контроль за изменением данных осуществляется очень легко и естественно (см. текст программы на диске).

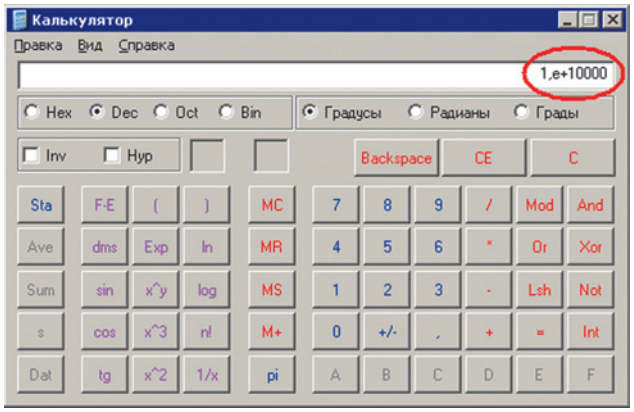


Рис. 16. Калькулятор Windows способен обрабатывать огромные вещественные числа

увидите, вполне способно вызвать положительные эмоции. Во всяком случае, я хорошо посмеялся.

Приложение на диске

На диске прилагаются все описанные в статье проекты, а также несколько дополнительных, о которых в тексте только упоминалось. Также с исходными текстами приведены проекты FPview_micro и FPview_mini, описанные в 3.7. Наконец, в папке normal вы найдете полный проект, демонстрирующий процесс нормализации десятичных чисел (см. *рис.* 7 на с. 11). Буду рад, если что-нибудь окажется вам полезным.

Литература

1. *Андреева Е., Фалина И.* Информатика: системы счисления и компьютерная арифметика. М.: Лаборатория Базовых Знаний, 1999.
2. *Вирт Н.* Алгоритмы и структуры данных. СПб.: Невский Диалект, 2001.
3. *Демидович Б.П., Марон И.А.* Основы вычислительной математики. М.: Государственное издательство физико-математической литературы, 1963.
4. *Еремин Е.А.* Представление информации в ЭВМ средствами Turbo Pascal / Информатика и образование, 1999, № 3.
5. *Еремин Е.А.* Популярные лекции об устройстве компьютера. СПб.: BHV-Петербург, 2003.
6. *Русаков С.В., Миндоров Н.И., Соловьева Т.Н.* Вычислительная погрешность / Информатика № 11, 2010.
7. *Яшкардин В.* IEEE 754 — стандарт двоичной арифметики с плавающей точкой. http://softelectro.ru/ieee754.html
8. Borland Pascal With Objects. Руководство по языку программирования.
9. IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754–1985).
10. IEEE Standard for Floating-Point Arithmetic (IEEE Std 754–2008).
11. Intel 64 and IA_32 Architectures Software Developer's Manual. Volume 1. Basic Architecture.
12. Online Binary-Decimal Converter. http://www.binaryconvert.com/convert_float.html

В МИР ИНФОРМАТИКИ // № 149

Газета для пытливых учеников
и их талантливых учителей
vmi@1september.ru

ШКОЛА ПРОГРАММИРОВАНИЯ

Решение линейного диофантового уравнения без лишних операций

В.Е. Гончаренко,
кандидат технических наук,
доцент Ивановского государственного университета

В программировании, как и в спорте, необходимы постоянные тренировки, оттачивающие мастерство и навыки. Проведем очередную тренировку в программировании на примере задачи, не очень сложной для обычной тренировки, и сконцентрируем внимание на избавлении программ от лишних действий. В рамках рассматриваемой нами задачи стремление избежать несколько десятков операций может показаться излишним при современных показателях быстродействия компьютеров, но мы говорим о принципиальном подходе к избавлению от лишнего и отрабатываем приемы на простых примерах. Современные программные проекты содержат сотни тысяч строк программного кода, и выполнение ненужных операций в них может вылиться в существенные задержки во времени выполнения программы. Сегодня только ленивый пользователь не жалуется на медлительность некоторых программ, которыми он повседневно пользуется на своем компьютере. Для тех, кто не хочет, чтобы в будущем на его программы жаловались ☺, и предназначена данная статья.

Лишние операции программу не красят

Рассмотрим достаточно известную старинную задачу на тему линейного диофантового уравнения[1]. Ее формулировки могут меняться, но суть остается неизменной.

"Крестьянин потратил 100 сольдо на покупку 100 различных животных: коров по 12 сольдо, овец по 4 сольдо и кроликов по 0,5 сольдо. Сколько голов каждого вида животных крестьянин купил при условии, что по крайней мере по одной голове каждого вида было куплено?"

Первоначальный подход к задаче очень простой: необходимо перебрать все возможные сочетания количества различных животных, и ответом задачи будут те варианты, для которых потраченное количество денег равно 100 сольдо и количество приобретенных голов также равно 100 единицам.

В программе на школьном языке программирования, реализующей эту идею решения задачи, используем следующие величины:

кор, *ов*, *пар_кр* — соответственно, количество купленных голов коров и овец и пар кроликов (понятно, что кроликов необходимо покупать парами, иначе ровно 100 сольдо не потратить);

ст_кор = 12, *ст_ов* = 4, *ст_кр* = 1 — соответственно, стоимость коровы, овцы и пары кроликов;

макс_кор, *макс_ов*, *макс_крол* — максимальное количество коров, овец и пар кроликов, которые можно купить.

Соответствующая программа имеет вид:

```
алг Покупки_крестьянина_1
нач цел ст_кор, ст_ов, ст_крол, кор, ов, пар_кр, макс_кор,
  макс_ов, макс_крол
  ст_кор := 12; ст_ов := 4
  ст_крол := 1
  макс_кор := div(100 - ст_крол - ст_ов, ст_кор)[2]
  макс_ов := div(100 - ст_кор - ст_крол, ст_ов)
```

[1] "Диофантовыми" называют алгебраические уравнения, отличающиеся тем, что количество неизвестных в них больше, чем количество уравнений, и их решение должно быть найдено в целых положительных числах. (Статью о Диофанте — древнеегипетском ученом, в честь которого названы уравнения, а также занимательную задачу, связанную с его именем, см. в газете "В мир информатики" № 145. — **Прим. ред.**)

[2] Напомним, что согласно условию как минимум по одной голове каждого вида животных было куплено. — **Прим. ред.**

```
  макс_крол := div(100 - ст_кор - ст_ов,
                   ст_крол)
  нц для кор от 1 до макс_кор
    нц для ов от 1 до макс_ов
      нц для пар_кр от 1 до макс_крол
        если кор * ст_кор + ов * ст_ов +
             пар_кр * ст_крол = 100
             и кор + ов + 2 * пар_кр = 100
        то
          вывод "Крестьянин купил следующих
                 животных:"
            вывод нс, "коров - ",кор, ",
                      овец - ",ов,",
                    кроликов - ",2 * пар_кр
        все
      кц
    кц
  кц
кон
```

Видно, что используются три вложенных оператора цикла с параметром (цикл для). Если включить в программу счетчик числа выполнений тела "внутреннего" оператора цикла, то можно установить, что для приведенных исходных данных это число равно 12 348.

Начинаем убирать лишнее из программы

В [1–2] основное внимание уделено исключению третьего вложенного оператора цикла. Это решение достаточно очевидно — ведь если из предыдущих двух операторов определено количество коров и овец, то этому количеству может соответствовать единственное количество пар кроликов, на которые необходимо потратить оставшиеся деньги:

```
пар_кр := div(100 - кор * ст_кор -
              ов * ст_ов, ст_крол)
```

После исключения третьего оператора цикла общее количество итераций сократится до 147, что с практической точки зрения вполне достаточно. Но, в соответствии с поставленной целью, рассмотрим все остальные возможности по устранению лишних операций.

Можно отметить, что логическое выражение в команде если (в условном операторе) довольно громоздкое. Рассмотрим систему уравнений к нашей задаче, что математически докажет ненужность третьего вложенного оператора цикла и даст возможность составить более простое логическое выражение.

Условие задачи записывается в виде двух уравнений:

$$ст\_кор * кор + ст\_ов * ов + крол/2 = 100 \quad (1)$$
$$кор + ов + крол = 100 \quad (2)$$

— где *крол* — количество кроликов (а не пар кроликов).

Умножим первое уравнение на 2 для того, чтобы избавиться от знаменателя, и, вычтя затем из него уравнение (2), получим

$$(2 * ст\_кор - 1) * кор + (2 * ст\_ов - 1) * ов = 100$$

В полученном уравнении остались только две переменные — *кор* и *ов*, что подтверждает возможность исключения третьего оператора цикла, а само полученное выражение можно использовать в команде если как более компактное.

Рассмотрим следующую возможность сокращения лишних итераций. Если в первом цикле задается очередное значение *кор*, то это значение можно учесть в определении максимального значения *ов* в следующем операторе цикла. С учетом этого начальную конструкцию второго оператора цикла можно записать в следующем виде:

```
нц для ов от 1 до div(100 - ст_кор * кор -
                      ст_крол, ст_ов)
```

Необходимо отметить, что выражения для начального и конечного значений параметров оператора цикла рассчитываются один раз до начала работы цикла, поэтому более громоздкая запись в структуре последнего оператора не приведет к дополнительным потерям времени его работы[3]. Использование рассмотренного усовершенствования сокращает общее количество итераций с 147 до 84. Конечно — не столь внушительное сокращение, как в первом случае, но, с другой стороны, зачем транслятору выполнять 63 лишние итерации?

На этом этапе создается впечатление, что все возможности удаления "лишнего" исчерпаны и можно представить перечень реализованных идей:

1) мы устранили третий вложенный оператор цикла, так как количество *пар_кр* можно рассчитать по определившимся значениям *кор* и *ов* предшествующих операторов;

2) во втором вложенном цикле оператора значение счетчика корректируется по значению *кор* из предшествующего оператора;

3) логическое выражение в команде если оператора мы упростили — оно стало простым, а не составным.

Казалось бы — все! Нет! К перечисленным достижениям можно еще добавить идею проверки наибольшего значения *кор*, не противоречащего условию задачи. Идея здесь такая. Самыми большими могут оказаться расходы на коров, что учитывается в первоочередном порядке, остальная "мелочь" добавляется позже. Но не получится ли так, что, потратив слишком много денег на коров, добрать до 100 голов "мелочью" уже не получится? Обозначим через x количество приобретенных коров, при котором еще есть возможность купить не менее 100 голов животных. Самый легкий вариант — увеличивать количество купленных голов, покупая кроликов, но при этом не купить хотя бы одну овцу нельзя. С учетом сделанных рассуждений можно записать неравенство:

$$x + 1 + 2 * (100 - x * ст\_кор - ст\_ов) \geqslant 100,$$

— где 1 учитывает одну овцу, а количество кроликов рассчитывается исходя из оставшейся суммы денег. После простых преобразований получим:

$$x \leqslant (100 - 2\,x * ст\_ов + 1) / (2 * ст\_кор - 1)$$

Полученное значение необходимо преобразовать к целому типу, отбрасывая дробную часть, что не нарушит условие неравенства. Соответствующий фрагмент программы будет выглядеть следующим образом:

[3] Можно также предварительно один раз рассчитать по громоздкой формуле конечное значение параметра цикла, как это сделано в первом варианте программы и будет сделано ниже. — **Прим. ред.**

```
  макс_кор := int((100 - 2 * ст_ов + 1)/
                  (2 * ст_кор - 1))
  нц для кор от 1 до макс_кор
  ...
```

В итоге, для исходных данных в условии задачи тело оператора цикла с параметром кор будет выполняться уже не 7, а 4 раза, а общее количество итераций сократится с 84 до ... — впрочем, предлагаю установить новое значение читателям (см. задания для самостоятельной работы).

Окончательный вариант исходного кода программы выглядит так:

```
алг Покупки_крестьянина_2
нач цел ст_кор, ст_ов, ст_крол, кор, ов,
         пар_кр, макс_кор
   ст_кор := 12; ст_ов := 4; ст_крол := 1
   макс_кор := int((100 - 2 * ст_ов + 1)/
                   (2 * ст_кор - 1))
   нц для кор от 1 до макс_кор
     нц для ов от 1 до div(100 - ст_кор *
                     кор - ст_крол, ст_ов)
        пар_кр := div(100 - кор *
                     ст_кор - ов *
                     ст_ов, ст_крол)
        если (2 * ст_кор - 1) * кор +
             (2 * ст_ов - 1) * ов = 100
        то
           вывод "Крестьянин купил
                  следующих животных:"
           вывод нс, "коров - ", кор, ",
                     овец - ", ов, ",
                     кроликов - ", 2 *
                     пар_кр
        все
     кц
   кц
кон
```

Задание для самостоятельной работы

1. Разработав программу на языке программирования, который вы изучаете, определите:

1) сколько каких животных купил крестьянин;

2) сколько раз будет выполняться тело “внутреннего” оператора цикла в последнем варианте программы, приведенном в статье.

2. Разработайте вариант программы, в котором учитывается тот факт, что число пар кроликов не должно быть больше 49 (иначе общее количество голов окажется больше 100). Повлияет ли учет этого факта на количество выполнений тела “внутреннего” оператора цикла?

3. Разработайте вариант программы, которая будет обладать свойством массовости (рассмотренная программа по этому показателю “хромает”). Значения потраченной общей суммы денег и количество планируемых к приобретению голов должен вводить сам пользователь, ну и, конечно, на разных рынках и в разное время будут меняться и цены на животных. Пожалуй, стоит оставить цену на кроликов 0,5 сольдо, так как это можно отнести к особым условиям данной задачи.

4. А самым отважным читателям ☺ предлагаю взяться за аналогичную задачу, когда крестьянин купит не три вида животных, а четыре (и более). Можно ли в этом случае предложить одну и ту же универсальную программу для решения класса задач по данной теме?

5. Разработайте программу решения еще одной задачи, которое приводит к необходимости использования диофантовых уравнений: “Требуется на 100 рублей купить 18 штук почтовых марок: шестирублевых, четырехрублевых и трехрублевых. Сколько окажется марок каждого достоинства?”

Ответы и/или программы, пожалуйста, присылайте в редакцию.

Литература

1. *Окулов С.М.* Основы программирования. М.: ЮНИМЕДИАСТАЙЛ, 2002.

2. *Немнюгин С.А.* Turbo Pascal. СПб.: Питер, 2001.

“ЛОМАЕМ” ГОЛОВУ

Числовой ребус с КУБом

Решите, пожалуйста, числовой ребус:

$$\text{КУБ} = \text{Б}^{\text{У}+\text{К}}$$

Одинаковыми буквами зашифрованы одинаковые цифры, разными буквами — разные цифры.

Еще одна задача “Прогноз погоды”

Четыре различных метеостанции дали прогноз погоды на завтра:

1-я: слабый ветер, снегопад, гололед, $t^o = 1$;

2-я: сильный ветер, гололед, отрицательная температура;

3-я: ясно, ветер слабый, $t^o = 0$;

4-я: ясно, безветренно, гололед, $t^o = 2$.

Известно, что каждая метеостанция может достоверно предсказать не более двух природных явлений (наличие осадков, температуру воздуха и др.), которые или не противоречат предсказаниям других станций, или подтверждаются хотя бы еще одной. Это означает, что завтра будет:

1) ясно, слабый ветер, гололед;

2) сильный ветер, гололед, $t^o = 2$;

3) снегопад, гололед, $t^o = 1$;

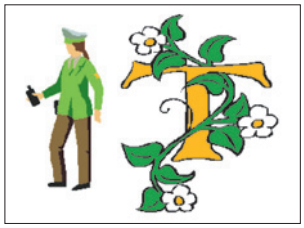
4) ясно, безветренно, $t^o < 0$.

Напомним, что методика решения подобных задач, которую предложил Саттаров Алмаз, ученик гимназии № 4 г. Бавлы, Республика Татарстан (учитель **Шафиков Н.Р.**), описана в газете “В мир информатики” № 127 (“Информатика” № 9/2009).

10 ребусов на одну тему

Приведенные ниже ребусы разработали ученики школы № 2 им. Н.П. Массонова, г. Свислочь Гродненской обл., Республика Беларусь, а подготовила их к публикации учитель **Синица А.А.**

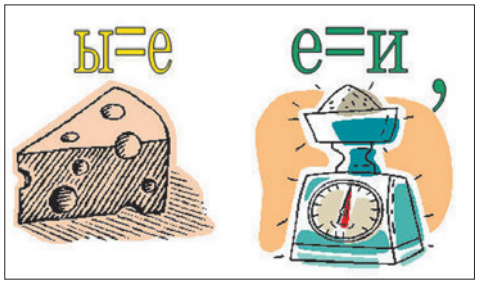
Ребус № 1



Ребус № 2

Ребус № 3

Ребус № 4



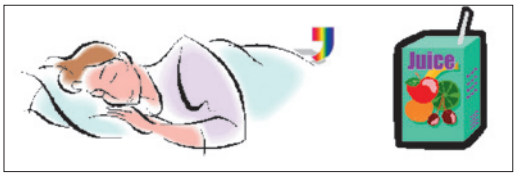
Ребус № 5

Ребус № 6

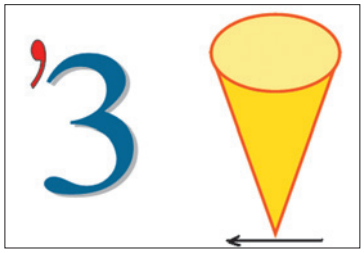
Ребус № 7

Ребус № 8

Ребус № 9



Ребус № 10



Решите, пожалуйста, эти ребусы. Определите также, с какой темой они связаны.

ЭКСПЕРИМЕНТЫ

И вновь — Debug

Н.Д. Тамошина, Москва

В газете-вкладке "В мир информатики" были опубликованы статьи [1–2], посвященные программе-отладчику Debug, некогда разработанной еще для операционной системы (ОС) MS-DOS, но по-прежнему входящей в состав ОС Windows, даже в ее версию Vista. Основные функции этой программы:

— загрузка программ в оперативную память, их просмотр и тестирование;

— вывод на экран содержимого участков памяти;

— изменение содержимого участков памяти и портов;

— перенос блоков данных из одного места памяти в другое;

— вывод на экран и корректировка содержимого регистров процессора;

— выполнение с помощью встроенного калькулятора шестнадцатеричного сложения и вычитания;

— сохранение содержимого участков памяти на гибких и жестких дисках;

— генерация и вывод на экран команд ассемблера по коду, загруженному в память компьютера.

В данной статье мы расскажем о выводе на экран информации из оперативной памяти.

Чтобы запустить программу Debug, щелкните на кнопке **Пуск** и выберите пункт **Выполнить** — появится диалоговое окно (*рис.* 4), в котором необходимо набрать имя программы и щелкнуть на кнопке **OK**.

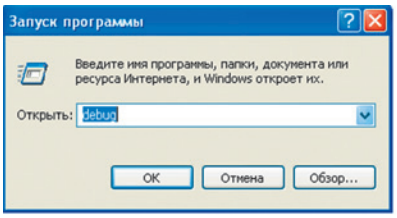


Рис. 1

После этого появится пустое черное окно, в верхней строчке которого будет высвечен символ "–", который является признаком готовности отладчика принять команду. Некоторые из команд Debug'а мы рассмотрим позже, а здесь заметим, что для выхода из программы следует ввести команду `q` (или `quit`).

Итак, запустите программу, посмотрите на ее внешний вид и выйдите из нее, набрав соответствующую команду.

Получилось? ☺ Тогда начинаем работать.

Для вывода на экран содержимого оперативной памяти в программе-отладчике Debug используется команда `d` (или `dump`). Ее формат:

```
- d [<на>] [<ка>]
```

или

```
- d [<на>] [L <длина>]
```

— где `<на>` и `<ка>` — соответственно начальный и конечный адреса выводимого участка памяти; `<длина>` — длина выводимого участка в байтах (после буквы L, которая может быть и строчной, пробел можно не ставить).

Обратим внимание также, что в квадратные скобки заключены необязательные параметры команды.

Адреса байтов памяти в команде `dump` указываются в так называемой "сегментной адресации" (см. [3]) и состоят из номера сегмента (СГ) и величины смещения (СМ). Сегментная часть стоит впереди и отделяется от смещения двоеточием:

СГ:СМ

а сами значения СГ и СМ указываются в виде 4-разрядных шестнадцатеричных чисел, например, ABCD:1234 (адрес байта в таком виде называют *логическим адресом*, а адрес как порядковый номер байта — *абсолютным адресом*). Другие примеры логических адресов:

5A1:DD
20:FFFF
0:ABCD
20:0
0000:0000
0:0

Видно, что начальные нули в значениях СГ и СМ можно не указывать, но нулевые значения приводить необходимо.

Прежде чем идти дальше, заметим, что информацию, прочитанную из памяти, называют *дампом* (это может быть файл, перенесенный на другой диск, текст на экране или текст, отправленный на печать прямо с экрана). Этот термин мы будем использовать часто.

Итак, проведем первый эксперимент — после запуска Debug'а введем команду:

```
d 10D4:0100.
```

Результат выполнения данной команды:

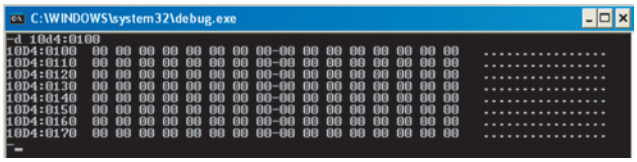


Рис. 2

Проанализируем отклик отладчика на нашу команду.

Изучим первую строку. Первое число 10D4:0100 — это начальный адрес дампа, где сегментная часть равна 10D4h[4], а смещение относительно начала сегмента — 0100h. Следующие за адресом два нуля означают, что в байте памяти с адресом 10D4:0100 записано шестнадцатеричное значение 00h. Следующие два нуля означают, что значение байта с адресом, на единицу большим предыдущего, — 10D4:0101, также равно 00h, и т.д. Дефис в середине строки разделяет байты на две равные группы. Подсчитав общее число таких пар в строке, убедимся, что в ней отображены значения 16 байтов, последовательно расположенных в памяти. При этом адрес начального байта первой строки равен 10D4:0100, а адрес последнего — 10D4:010F, т.е. в строке выведено содержимое одного параграфа (см. [3]).

В конце первой и других строк представлены также точки (их тоже 16). О них мы поговорим чуть позже.

Следующие семь строк, каждая из которых начинается с адреса первого для этой строки байта, представляют оставшуюся часть дампа. Начальный адрес дампа — `10D4:0100`, а конечный — `10D4:017F`.

Таким образом, весь дамп содержит 128 последовательно расположенных байтов (8 параграфов).

Введем команду `d` еще раз[5]:

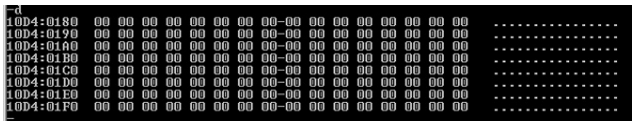


Рис. 3

Последовательность символов отличается начальным адресом дампа `10D4:0180`, который превышает на единицу конечный адрес предыдущего дампа. Таким образом, если вводить команду `d` без параметров, программа Debug будет последовательно выводить по 128 байтов памяти, начиная с очередного байта.

Еще пример. Выведем на экран содержимое участка памяти, начиная с адреса DDD1:234. Для этого введем соответствующую команду:

```
-d DDD1:234
```

Результат ее выполнения[6]:

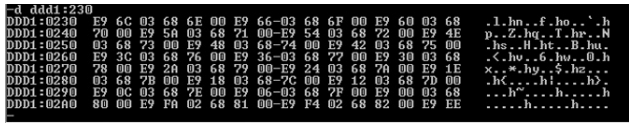


Рис. 4

[4] Буква *h* указывает на шестнадцатеричную систему.

[5] Видно, что она вводится без параметров. — **Прим. ред.**

[6] На вашем компьютере дамп может быть другим. Если на нем представлены одни нули, то попробуйте указывать другой начальный адрес до тех пор, пока не будут получены ненулевые значения. — **Прим. ред.**

Отметим, что дамп начинается с адреса, указанного в команде, и состоит, как обычно, из 128 последовательно расположенных байтов памяти. Но на этот раз дамп содержит не только нули. Например, значение байта с адресом DDD1:235 равно 6Ch.

Справа от каждой строки вместо некоторых из 16 точек появились символы. Это символы, коды которых в таблице кодировки ASCII равны значению соответствующих байтов той же строки. При этом выводятся только символы с кодами в первой половине (в стандартной части) таблицы ASCII[7]. Те же символы, ASCII-коды которых не являются стандартной частью таблицы ASCII, т.е. не входят в диапазон от 0 до 7Fh (от 0 до 127 в десятичной системе счисления), обозначаются на экране точкой. Так, все буквы русского алфавита будут представлены на экране точками, поскольку их коды находятся во второй половине таблицы ASCII.

В нашем примере, в полученном дампе значение байта DDD1:235 равно 6Ch. Данному коду в таблице ASCII соответствует символ "l", который выведен в соответствующей позиции символьной части строки. Байт DDD1:237 содержит значение 68h, являющееся ASCII-кодом символа "h", и т.д.

Как следует из формата команды `d` (`dump`), в качестве ее параметра можно указывать также количество выводимых байтов.

С помощью команды `d` на экран можно вывести содержимое (байты) конкретного файла. Подтвердим это экспериментально.

В текстовом редакторе **Блокнот** создадим файл *primer.txt*, состоящий из символов, ASCII-коды которых принадлежат стандартной части. Например, следующий текст:

```
++++++++++++++++++++++++++++++++++++
This text file will be used to demonstrate DEBUG.
This file will be loaded into memory when DEBUG is started.
DEBUG is easy to use after some practice.
++++++++++++++++++++++++++++++++++++
```

Файл сохраним в корневом каталоге (корневой папке) диска C:.

Запустим на исполнение программу Debug, указав в качестве параметра полное имя созданного файла (*рис.* 5).

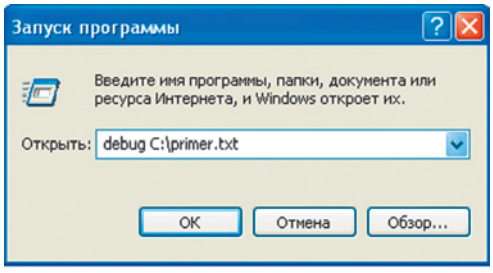


Рис. 5

Для вывода на экран дампа, в котором содержится файл *primer.txt*, введем команду `d` без параметров и, дождавшись результата, введем эту же команду повторно и получим следующее:

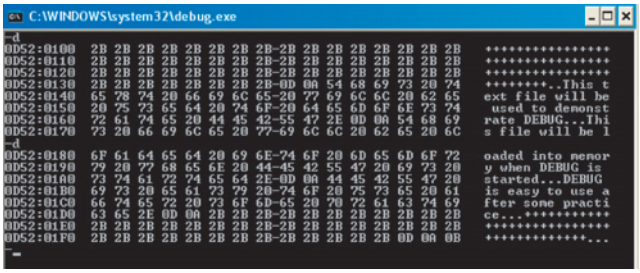


Рис. 6

Мы видим содержимое файла *primer.txt*, размещенное в оперативной памяти, начиная с адреса 0D524:0100 (на вашем компьютере этот адрес может быть другим). Причем в левой части представлены ASCII-коды, записанные в шестнадцатеричной системе счисления, а в правой части — строки символов, соответствующие этим кодам.

Задания для самостоятельной работы

1. Определите, как будет выглядеть дамп на экране, если начальный адрес выводимого участка памяти будет не кратным 16.

2. Установите, можно ли в команде dump:

1) в конечном адресе выводимого участка памяти не указывать значение сегментной части, если оно совпадает с соответствующим значением начального адреса этой же команды;

2) в начальном адресе выводимого участка памяти не указывать значение сегментной части, если оно совпадает с соответствующим значением, использованным в предыдущей команде dump.

3. Определите, что записано в оперативной памяти в 48 байтах, начинающихся с байта с адресом 10D4:0AD0.

4. В оперативную память записывается информация о дате выпуска версии постоянного запоминающего устройства (ПЗУ, ROM) компьютера. Эта информация хранится в формате ММ/ЧЧ/ГГ (месяц/число/год[8]), начиная с абсолютного адреса FFFF5. Определите указанную дату для своего компьютера.

5. Ответьте, пожалуйста, на вопрос, почему в третьей строке второй части дампа, приведенного на *рис.* 6, символы, коды которых равны 2E, 0D и 0A, справа представлены точками, хотя десятичные значения этих кодов меньше 127.

Ответы присылайте в редакцию (можно выполнять не все задания).

Литература

1. *Заславская О.Ю.* Работа с регистрами центрального процессора. / "В мир информатики" № 48 ("Информатика" № 1/2005).

2. *Еремин Е.А.* Еще раз о программе Debug. / "В мир информатики" № 82, 84 ("Информатика" № 23, 25/2006).

3. *Тамошина Н.Д.* Об адресации оперативной памяти. / "В мир информатики" № 147 ("Информатика" № 17/2010).

От редакции. В файле *primer.txt* имеется текст, смысл которого на русском языке следующий: "Debug легок в использовании после приобретения некоторого опыта работы с ним". Обращаем внимание читателей на это.

[7] Исключение составляют так называемые "управляющие символы", не имеющие представления в таблице ASCII (с десятичными кодами 0–31). Для них справа также выводится точка. — **Прим. ред.**

[8] Это так называемый "американский стандарт даты". — **Прим. ред.**

Непримитивные примитивы

“Мое любимое занятие — заниматься огородом”, — говорит сегодня Джек Брезенхем (Jack Elton Bresenham, родился в 1937 г.). Этой цитатой заканчивается редкая публикация об ученом, имя которого носят алгоритмы, реализованные практически во всех программах и аппаратных устройствах компьютерной графики. Надо сказать, что найти информацию о Брезенхеме действительно непросто. Даже англоязычная Википедия содержит лишь очень короткую и не слишком информативную статью. Фактически нам удалось найти только заметку (http://www.mytjnow.com/former-winthrop-professor-resp/author/thejohnsonian) в студенческой газете университета Winthrop (США), в котором Брезенхем преподавал до выхода в отставку. Про огород — это оттуда.

Что же такое придумал Брезенхем (как он сам подчеркивает, не один — с коллегами), что алгоритмы, носящие его имя, входят практически во все классические учебники? Всего лишь научился строить отрезки и окружности! Подумаешь! Подумаем...

Понятно, что и отрезок, и окружность совсем просто построить, например, посредством параметрических уравнений. Но у такого способа построения есть очень серьезный недостаток — он требует использования вещественной арифметики. А с окружностью — вообще беда, там еще и синусы с косинусами. Дело не в том, что кого-то пугают вещественные числа и тригонометрические функции, а в том, что алгоритмы, основанные на их использовании, работают жутко медленно. Причем медленно они работали и в далеком 1962 году, когда сотрудник IBM Джек Брезенхем разработал свои алгоритмы, и сейчас. Абсолютный смысл слов “жутко медленно” с тех пор, конечно, изменился, относительный — нет.

Брезенхем разработал алгоритмы построения отрезков и окружностей (последний позволяет строить и эллипсы), основанные исключительно на использовании целочисленной арифметики. Никаких вещественных чисел, никаких синусов и косинусов. Только целые числа и целочисленные операции. Сложение и вычитание целых чисел выполняются достаточно быстро, а умножать в этих алгоритмах требуется только на 2. Поскольку умножение на 2 реализуется посредством быстрой операции — битового сдвига, алгоритмы работают очень эффективно.

Имеются не только многочисленные (бесчисленные) программные, но и аппаратные реализации алгоритмов Брезенхема. И хотя более ничем их автор не знаменит, всего лишь двух примитивов — line и circle — хватило, чтобы навсегда вписать свое имя в историю компьютерных наук.