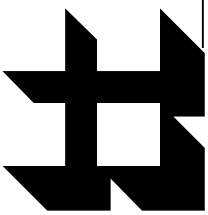


ГЛАВА 23

GDIPlus. Графический вывод

В этой главе мы обсудим способы печати изображений на принтере и рисование в окне формы. Вы также узнаете много нового (и неожиданного) о строении окон и научитесь перехватывать и обрабатывать оконные сообщения Windows.

Печать на принтере

Для того чтобы печатать на принтере, необходимо создать объект `Graphics`, связанный с графическим контекстом принтера. Кроме того, необходимо создать документ принтера, на страницах которого мы будем рисовать.

Создание графического контекста принтера

Один из вариантов создания графического контекста принтера для работы с ним из GDIPlus заключается в применении Windows API-функции `CreateDC`. Эта функция получает имя принтера, создает необходимую структуру и возвращает указатель на нее:

```
DECLARE Long CreateDC IN Gdi32.dll String lpszDriver, String lpszDevice,;
                                   String lpszOutput, String Devmode
hDC = CreateDC(NULL, Printer_Name, NULL, NULL)
```

Функция получает следующие параметры:

- *`lpszDriver`* — указатель на нуль-терминированную строку, содержащую имя драйвера. Должен быть `NULL`, если указан параметр *`lpszDevice`*;
- *`lpszDevice`* — указатель на нуль-терминированную строку, содержащую имя принтера;
- *`lpszOutput`* — не используется; должен быть `NULL`;
- *`Devmode`* — указатель на структуру `DEVMODE`, содержащую описание присущих устройству параметров, используемых при инициализации драйвера устройства.

Эта структура позволяет управлять положением листа, качеством графики и рядом других параметров; вы можете найти ее описание в MSDN.

Если функция `CreateDC` возвращает ноль, то это означает, что графический контекст не создан.

Для получения имени принтера можно воспользоваться встроенной функцией `GETPRINTER()`. Эта функция формирует диалоговое окно (рис. 23.1), в котором вы можете выбрать как локальный принтер, так и любой другой из доступных в вашей локальной сети, и возвращает строку с именем выбранного принтера:

```
Printer_Name = GETPRINTER()
```

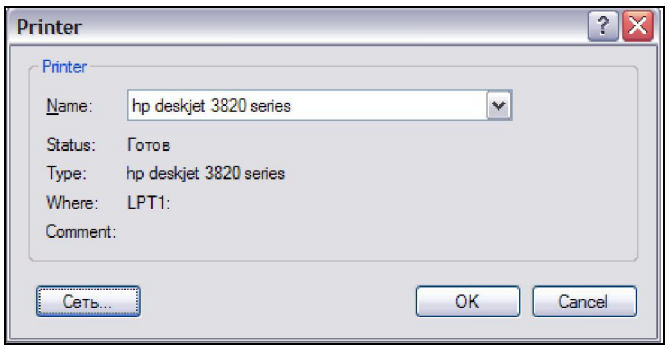


Рис. 23.1. Диалог для выбора принтера

Если вы хотите печатать на принтере, установленном по умолчанию, то воспользуйтесь функцией `SET("Printer")`:

```
Printer_Name = SET("Printer", nType)
```

где параметр `nType` может принимать одно из следующих значений:

- 2 — принтер по умолчанию в Windows;
- 3 — принтер по умолчанию в Visual FoxPro.

Так как Visual FoxPro "не видит" память, зарезервированную функциями Windows API, то вы должны самостоятельно удалить графический контекст принтера, вызвав функцию `DeleteDC`:

```
DECLARE Long DeleteDC IN Gdi32.dll Long hDC
= DeleteDC(hDC)
```

Создание связанного с принтером объекта *Graphics*

Для создания требуемой модификации объекта `Graphics` для вывода на принтер используется функция `GdipCreateFromHDC`. Вот ее объявление в Visual FoxPro:

```
DECLARE Long GdipCreateFromHDC IN Gdiplus.dll ;
        Long hdc, Long @ nativeGraphics
```

Функции передается два параметра. Параметр *hdc* — это указатель на структуру, содержащую графический контекст принтера, а в передаваемом по ссылке параметре *nativeGraphics* запоминается дескриптор созданного объекта Graphics.

В следующем фрагменте кода показано, как создать объект Graphics, связанный с принтером, используемым по умолчанию:

```
cPrinterName = SET("Printer", 2)
hDC = CreateDC(NULL, cPrinterName, NULL, NULL)
IF hDC != 0
   lnGraphics = 0
   Status = GdipCreateFromHDC(hDC, @lnGraphics)
ENDIF
```

Определение размера листа принтера

Справедливо предположить, что для рисования на принтере вы захотите использовать систему координат, единицей измерения в которой являются миллиметры. Поэтому неплохо было бы узнать, страницу какого размера ваш принтер может напечатать. Для определения размеров страницы используется функция GetDeviceCaps. Вот ее объявление:

```
DECLARE Long GetDeviceCaps IN Gdi32.dll Long hdc, Long nIndex
```

Параметр *hdc* — это указатель на структуру, содержащую графический контекст принтера, а *nIndex* — число, принимающее одно из значений, перечисленных в табл. 23.1.

Таблица 23.1. *Значения параметра nIndex функции GetDeviceCaps*

nIndex	Значение
4	Ширина листа в миллиметрах
6	Высота листа в миллиметрах
8	Ширина листа в пикселах
10	Высота листа в пикселах

В следующем фрагменте кода показано, как определить размеры листа (в мм) для принтера, установленного по умолчанию:

```
CLEAR
DECLARE Long CreateDC IN Gdi32.dll String, String, String, String
DECLARE Long GetDeviceCaps IN Gdi32.dll Long, Long
hDC = CreateDC(NULL, SET("Printer", 2), NULL, NULL)
? "Ширина", GetDeviceCaps(hDC, 4)
? "Высота", GetDeviceCaps(hDC, 6)
```

Напомню, что установить единицу измерения для печати на принтере можно функцией GdipSetPageUnit, описание которой приведено в начале предыдущей главы.

Документ принтера

При печати изображений на принтере первоначально весь графический вывод направляется в *документ* принтера. Документ может содержать произвольное количество *страниц*. Каждую страницу документа условно можно рассматривать как аналог растра с размерами, равными размерам листа бумаги. После того как все необходимые страницы документа сформированы, документ закрывается и направляется в очередь печати.

Создание документа принтера

Создает документ принтера функция `StartDoc`. Эта функция получает два параметра: указатель на графический контекст принтера и указатель на структуру `DOCINFO`. В общем случае структура `DOCINFO` может быть представлена символьной строкой длиной 20 байт, первый байт которой имеет значение 20.

В следующем фрагменте кода показан вариант создания документа принтера с использованием пустой структуры `DOCINFO`.

```
DECLARE Long StartDoc IN Gdi32.dll Long hdc, String Docinfo
cDocInfo = CHR(20) + REPLICATE(CHR(0), 19)     && Формируем структуру
Result = StartDoc(hDC, cDocInfo)               && Создаем документ
```

Если документ создан успешно, то функция возвращает значение, отличное от нуля, а в случае ошибки — ноль.

Заполнение структуры *DOCINFO*

Эта структура подробно рассматривалась в *главе 19*, в разделе "Распределение памяти для структур с указателями". Напомню, что она содержит пять четырехбайтовых полей, из которых интерес представляет второе поле, которое содержит указатель на строку с именем печатаемого документа. Это имя выводится в окне просмотра очереди печати (рис. 23.2) в колонке **Документ**.

Следующий фрагмент кода показывает, как документу принтера присвоить имя. Предполагается, что это имя хранится в переменной `cDocName`.

```
nLen = LEN(cDocName)
* Распределяем глобальную память для размещения строки с именем документа
hGlobal = GlobalAlloc(0x0040, nLen)
* Копируем строку в эту память
= SYS(2600, hGlobal, nLen, cDocName)
* Формируем структуру DOCINFO
cDOCINFO = BINTOC(20,"4RS") + BINTOC(hGlobal,"4RS") + ;
           REPLICATE(CHR(0),12)
```

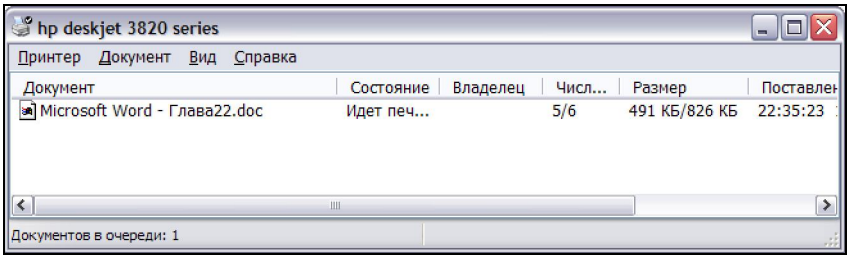


Рис. 23.2. Окно просмотра очереди печати

Не забудьте освободить распределенную функцией `GlobalAlloc` память после закрытия документа принтера!

Управление страницами документа

Функция `StartPage` добавляет в документ новую страницу и делает ее активной. Вы можете рисовать только на активной странице.

Функция получает только один параметр — указатель на графический контекст принтера.

Закрывает страницу функция `EndPage`. Закрытую страницу невозможно вновь сделать активной. Функция также получает только один параметр — указатель на графический контекст принтера.

После того как вы закрыли страницу, вы можете открыть следующую страницу и продолжить рисовать уже на ней.

Следующий фрагмент кода показывает, как организуется работа со страницами документа принтера.

```
DECLARE Long StartPage IN Gdi32.dll Long hDC
DECLARE Long EndPage IN Gdi32.dll Long hDC
* Предполагаем, что документ уже создан
Result = StartPage(hDC)
* Здесь — вызов функций GDIPlus для рисования на странице
Result = EndPage(hDC)
* Создаем следующую страницу документа принтера
Result = StartPage(hDC)
* и т. д.
```

Функции `StartPage` и `EndPage` при успешном выполнении возвращают отличное от нуля значение и ноль в случае ошибки.

Закрытие документа

Для того чтобы послать документ принтера в очередь печати, необходимо его закрыть. Делается это при помощи функции `EndDoc`. Функция получает только один параметр — указатель на графический контекст принтера:

```
DECLARE Long EndDoc IN Gdi32.dll Long hDC
```

```
Result = EndDoc(hDC)
```

Класс для печати на принтере

Создадим из функций, реализующих печать на принтере, класс с именем `GdipPrinter`. Добавьте этот класс в библиотеку VfpGdiPlus.vcx (мы по-прежнему продолжаем создавать классы для работы с GDIPlus!), в качестве класса-родителя выберите `Custom`.

Добавьте в класс следующие защищенные свойства:

- `Status` — для хранения состояния класса;
- `hDC` — для хранения указателя на графический контекст принтера;
- `nativeGraphics` — для хранения дескриптора объекта `Ghaphics`;
- `HGLOBAL` — для хранения указателя на распределенную для наименования документа память;
- `DocStatus` — для хранения статуса документа принтера ("истина", если документ существует).

Начальные значения всех этих свойств, кроме свойства `DocStatus`, установите равными нулю.

Метод *Init*

Поскольку класс `GdipPrinter` задуман как расширение класса `GdipImages`, то предполагается, что объект — экземпляр этого класса будет создаваться уже после того, как создан объект — экземпляр класса `GdipImages`. Таким образом, инициализация GDIPlus в этом методе не требуется.

В метод может быть передан один необязательный параметр: имя принтера, на котором вы хотите печатать. Если имя принтера опущено, то графический контекст будет создан для принтера, используемого на вашем компьютере по умолчанию.

Код метода показан в листинге 23.1.

```
LPARAMETERS tcPrinterName
LOCAL lcPrinter, lnGraphics, lhDC
IF VARTYPE(tcPrinterName) = "C"  && Если имя принтера передано методу
   lcPrinter = tcPinterName      && то используем его
ELSE                             && иначе —
   lcPrinter = SET("Printer", 2) && берем принтер по умолчанию
ENDIF
IF LEN(ALLTRIM(lcPrinter))
   DECLARE Long CreateDC IN Gdi32.dll String, String, String, String
   DECLARE Long GetDeviceCaps IN Gdi32.dll Long, Long
   lhDC = CreateDC(NULL, lcPrinter, NULL, NULL)
   IF lhDC != 0
```

```
      this.hDC = lhDC
      lnGraphics = 0
      DECLARE Long GdipCreateFromHDC IN Gdiplus.dll Long, Long @
      this.Status = GdipCreateFromHDC(lhDC, @lnGraphics)
      IF this.Status = 0
         this.nativeGraphics = lnGraphics
         RETURN .t.              && Объект создан успешно
      ENDIF                      && иначе —
      DECLARE Long DeleteDC IN Gdi32.dll Long
      = DeleteDC(lhDC)           && удаляем графический контекст принтера
   ENDIF
ENDIF
RETURN .f.
```

Переменная `lcPrinter` получает либо переданное в метод имя принтера, либо имя принтера, используемого на компьютере по умолчанию. Функция `CreateDC` создает графический контекст принтера и запоминает указатель на него в свойстве `hDC`. Далее создается связанный с принтером объект `Graphics`; дескриптор этого объекта сохраняется в свойстве `nativeGraphics`.

Функция `DeleteDC` удаляет графический контекст принтера в случае, если не удалось создать связанный с ним объект `Graphics`; в этом случае, а также в случае, если невозможно обнаружить принтер, объект — экземпляр класса `GdipPrinter` не создается.

В следующем фрагменте кода демонстрируется создание объектов — экземпляров класса `GdipPrinter`:

```
* Подключение к произвольному принтеру в вашей локальной сети
SET CLASSLIB TO vfpgdiplus.vcx
oPrint = CREATEOBJECT("GdipPrinter", GETPRINTER())
* Подключение к принтеру, используемому по умолчанию.
SET CLASSLIB TO vfpgdiplus.vcx
oPrint = CREATEOBJECT("GdipPrinter")
```

Метод *GetGraphics*

Этот метод возвращает дескриптор объекта `Graphics`, связанный с принтером. Дескриптор может использоваться функциями рисования, рассмотренными в предыдущей главе.

Метод содержит только одну команду:

```
RETURN this.nativeGraphics
```

Установка единицы измерения. Метод *SetPageUnit*

Вы можете указывать различные единицы измерения для каждой страницы документа принтера. Метод `SetPageUnit` получает три параметра, первый из которых (логического типа) определяет, какая единица измерения будет использоваться при рисовании на странице документа принтера (значение "ложь" определяет пикселы, значение "истина" — миллиметры), а в остальные передаваемые по ссылке параметры будет

записано значение ширины и высоты листа в выбранной единице измерения. Код метода приведен в листинге 23.2.

```
LPARAMETERS tlUnit, tnListWidth, tnListHeight
LOCAL lnUnit
IF VARTYPE(tlUnit) + VARTYPE(tnListWidth) + VARTYPE(tnListHeight) = "LNN"
   DECLARE Long GetDeviceCaps IN Gdi32.dll Long, Long
   IF tlUnit                        && Выбрано: миллиметры
      lnUnit = 6
         tnListWidth = GetDeviceCaps(this.hDC, 4)
         tnListHeight = GetDeviceCaps(this.hDC, 6)
   ELSE                             && Выбрано: пикселы
      lnUnit = 3
      tnListWidth = GetDeviceCaps(this.hDC, 8)
      tnListHeight = GetDeviceCaps(this.hDC, 10)
   ENDIF
   DECLARE Long GdipSetPageUnit IN Gdiplus.dll Long, Long
   this.Status = GdipSetPageUnit(this.nativeGraphics, lnUnit)
ELSE
   this.Status = 2
ENDIF
RETURN this.Status = 0
```

В следующем фрагменте кода показано, как использовать метод `SetPageUnit` для установки единицы измерения "миллиметры":

```
nPageWidth = 0
nPageHeight = 0
oPrint.SetPageUnit(.t., @nPageWidth, @nPageHeight)
```

После выполнения этого кода переменные `nPageWidth` и `nPageHeight` будут содержать значения ширины и высоты страницы принтера в миллиметрах.

Создание документа принтера. Метод *OpenDocument*

Создает документ принтера метод `OpenDocument`. Он получает только один параметр — имя печатаемого документа.

Если документ создан успешно, то в нем сразу открывается страница, на которой вы можете рисовать при помощи объекта `Ghaphics`. Код метода приведен в листинге 23.3.

```
LPARAMETERS tcDocumentName
LOCAL lcDOCINFO, lnLen
IF VARTYPE(tcDocumentName) = "C"
* Имя документа передано. Формируем структуру DOCINFO
   tcDocumentName = ALLTRIM(tcDocumentName)
   lnLen = LEN(tcDocumentName)
```

```
   IF lnLen > 0
      this.hGlobal = GlobalAlloc(0x0040, lnLen)
      = SYS(2600, this.hGlobal, lnLen, tcDocumentName)
   ENDIF
ENDIF
cDOCINFO = BINTOC(20,"4RS") + BINTOC(hGlobal,"4RS") + ;
           REPLICATE(CHR(0),12)
DECLARE Long StartDoc IN Gdi32.dll Long, String
IF StartDoc(this.hDC, lcDOCINFO) > 0   && Создаем документ
   this.DocStatus = .t.
   DECLARE Long StartPage IN Gdi32.dll Long
   IF StartPage(this.hDC) > 0          && Открываем страницу
      RETURN .t.
   ENDIF
ENDIF
RETURN .f.
```

Метод *NewPage*

Метод закрывает текущую страницу и открывает новую. Его код показан в листинге 23.4.

```
this.Status = IIF(this.DocStatus, 0, -1)
IF this.Status = 0
   DECLARE Long StartPage IN Gdi32.dll Long
   DECLARE Long EndPage IN Gdi32.dll Long
   IF EndPage(this.hDC) != 0
      IF StartPage(this.hDC) != 0
         RETURN .t.
      ELSE
         this.Status = -2  && Невозможно открыть / закрыть страницу
      ENDIF
   ELSE
      this.Status = -2
   ENDIF
ENDIF
RETURN .f.
```

После вызова этого метода вы уже не сможете вернуться на предыдущие страницы документа для правки.

Метод *CloseDocument*

Метод закрывает страницу документа принтера и сам документ и направляет его в очередь печати. Код метода приведен в листинге 23.5.

```
this.Status = IIF(this.DocStatus, 0, -1)
IF this.Status = 0
   DECLARE Long EndPage IN Gdi32.dll Long
   DECLARE Long EndDoc IN Gdi32.dll Long
   = EndPage(this.hDC)
   IF EndDoc(this.hdc) != 0
      this.DocStatus = .f.
      RETURN .t.
   ELSE
      this.Status = -3
   ENDIF
ENDIF
RETURN .f.
```

Метод проверяет, был ли создан документ принтера, и, если да, закрывает страницу и сам документ, после чего устанавливает свойство `DocStatus` класса в "ложь".

Метод *GetStatus*

Как и в классе `GdipImages`, назначение этого метода — вернуть значение свойства `Status`, содержащего коды завершения функций GDIPlus. Он содержит только одну команду:

```
RETURN this.Status
```

Метод *Destroy*

Так как этот метод вызывается при уничтожении объекта, мы должны включить в него команды для освобождения памяти, занимаемой графическим контекстом принтера, объектом `Graphics` и, возможно, строкой — наименованием документа. Кроме того, если документ принтера еще не закрыт, то в методе он закрывается и отправляется в очередь печати (листинг 23.6).

```
IF this.DocStatus
   this.CloseDocument()
ENDIF
DECLARE Long DeleteDC IN Gdi32.dll Long
DECLARE Long GdipDeleteGraphics IN Gdiplus.dll Long
= GdipDeleteGraphics(this.nativeGraphics)
= DeleteDC(this.hDC)
IF this.hglobal != 0
   DECLARE Long GlobalFree IN WIN32API Long
   = GlobalFree(this.hGlobal)
ENDIF
```

Тестирование

Тестовые примеры, приведенные ниже, демонстрируют способы печати на принтере графических примитивов и изображений.

Печатаем графические примитивы и тексты

За основу возьмем тестовый пример из файла test14.prg (если вы его не создали в процессе чтения предыдущей главы, ничего страшного). Создайте в проекте новый программный файл с именем test15.prg и введите в него код из листинга 23.7 (это всего лишь некоторая модификация кода из test14.prg).

```
LOCAL lcPath, loGP, llStatus, lcText, lnWidth, ;
      lnHeight, lnLeft, lnGraphics
lcPath = JUSTPATH(SYS(16))
SET DEFAULT TO (lcPath)
SET CLASSLIB TO vfpgdiplus
loGP = CREATEOBJECT("gdipimages")
loPrint = CREATEOBJECT("gdipPrinter")
IF VARTYPE(loPrint) != "O"
   = MESSAGEBOX("Не удалось подключиться к принтеру"
   RETURN
ENDIF
* Получаем дескриптор объекта Graphics, связанного с принтером
lnGraphics = loPrint.GetGraphics()
* Открываем документ принтера
llStatus = loPrint.OpenDocument()
IF llStatus
* Создаем перо
   llStatus = loGP.CreatePen(1)
ENDIF
IF llStatus
* Рисуем пять прямоугольников, чтобы отметить области, в которые выводится текст
   llStatus = loGP.DrawRectangle(10, 40, 300, 60, lnGraphics)
   llStatus = loGP.DrawRectangle(10, 110, 300, 60, lnGraphics)
   llStatus = loGP.DrawRectangle(10, 180, 300, 60, lnGraphics)
   llStatus = loGP.DrawRectangle(320, 40, 110, 180, lnGraphics)
   llStatus = loGP.DrawRectangle(440, 40, 80, 180, lnGraphics)
   llStatus = loGP.DrawRectangle(530, 40, 100, 180, lnGraphics)
ENDIF
IF llStatus
* Создаем одноцветную кисть коричневого цвета
   llStatus = loGP.CreateSolidBrush(0xFF882000)
ENDIF
IF llStatus
* Создаем шрифт высотой 16 пикселов
   llStatus = loGP.CreateFont("Times New Roman", 16)
ENDIF
```

```
IF llStatus
* Центрирование строки посредством вычисления ее размеров
   lcText = "Эта строка центрируется по горизонтали"
   lnWidth = 0
   lnHeight = 0
   llStatus = loGP.GetLengthString(lcText, @lnWidth, @lnHeight, ;
                                   lnGraphics)
   lnLeft = (650 — lnWidth) / 2  && Определение левой точки
ENDIF
IF llStatus
   llStatus = loGP.DrawString(lnLeft, 5, 0, 0, lcText, lnGraphics)
ENDIF
IF llStatus
* Создаем объект StringFormat
   llStatus = loGP.CreateStringFormat()
ENDIF
IF llStatus
   lcText = "Этот текст выводится в прямоугольную область без " + ;
            "форматирования"
   llStatus = loGP.DrawString(10, 40, 300, 60, lcText, lnGraphics)
ENDIF
IF llStatus
   lcText = "Строки этого текста центрируется внутри заданной " + ;
            "прямоугольной области"
   llStatus = loGP.SetStringFormatParameter(1)
   llStatus = loGP.DrawString(10, 110, 300, 60, lcText, lnGraphics)
ENDIF
IF llStatus
   lcText = "Этот текст выровнен по правому краю " + ;
            "заданной прямоугольной области"
   llStatus = loGP.SetStringFormatParameter(2)
   llStatus = loGP.DrawString(10, 180, 300, 60, lcText, lnGraphics)
ENDIF
IF llStatus
   lcText = "Этот текст выведен вертикально без форматирования " + ;
            "и коррекции качества начертания символов"
   llStatus = loGP.SetStringFormatParameter(0, .t.)
   llStatus = loGP.DrawString(320, 40, 110, 180, lcText, lnGraphics)
ENDIF
IF llStatus
* Улучшаем качество вертикально выводимого текста
        llStatus = loGP.SetTextRendering(4, lnGraphics)
ENDIF
IF llStatus
   lcText = "Этот текст выведен вертикально и центрирован"
   llStatus = loGP.SetStringFormatParameter(1, .t.)
   llStatus = loGP.DrawString(440, 40, 80, 180, lcText, lnGraphics)
ENDIF
IF llStatus
   lcText = "Этот текст выведен вертикально и прижат " + ;
            "к нижней границе прямоугольной области"
   llStatus = loGP.SetStringFormatParameter(2, .t.)
```

```
   llStatus = loGP.DrawString(530, 40, 100, 180, lcText, lnGraphics)
ENDIF
= MESSAGEBOX(IIF(llStatus, "ok","Ошибка" + STR(loGP.GetStatus())))
```

В начале тестового примера, после создания объекта — экземпляра класса `GdipImages`, создается объект — экземпляр класса `GdipPrinter` для принтера, используемого по умолчанию. Дескриптор объекта `Graphics`, связанный с принтером, запоминается в переменной *`lnGraphics`* и в дальнейшем используется *всеми функциями рисования* для вывода на принтер.

Запустите тест. Оцените качество печати (а ведь мы не устанавливали для принтера никаких единиц измерения; так работает единица измерения UnitWorld). Обратите внимание, что вызов метода `SetTextRendering` не оказывает на качество печатаемого текста никакого влияния; действительно, он эффективен только при выводе на жидкокристаллические мониторы.

Вставьте в код тестового примера команды вызова метода `SetPageUnit` и установите в качестве единицы измерения миллиметры. Измените значения координат и размеров графических примитивов так, чтобы они соответствовали этой единице измерения. Запустите тест на выполнение. Возьмите линейку и проверьте, действительно ли размеры прямоугольников соответствуют указанным в миллиметрах.

Печатаем изображения

Создайте в проекте еще один программный файл с именем test16.prg и введите в него код из листинга 23.8.

```
LOCAL lcPath, loGP, loPrint, llStatus, lcFile, lnGraphics
lcPath = JUSTPATH(SYS(16))
SET DEFAULT TO (lcPath)
SET CLASSLIB TO vfpgdiplus
lcFile = GETFILE('JPG|GIF|BMP')
IF EMPTY(lcFile)
   RETURN
ENDIF
* Создаем объект — экземпляр класса GdipImages
loGP = CREATEOBJECT("gdipimages")
* Загружаем графический файл
IF loGP.LoadFromFile(lcFile)
   loPrint = CREATEOBJECT("gdipPrinter")
   IF VARTYPE(loPrint) != "O"
      = MESSAGEBOX("Не удалось подключиться к принтеру"
      RETURN
   ENDIF
ELSE
   RETURN
ENDIF
* Получаем дескриптор объекта Graphics, связанного с принтером
```

```
lnGraphics = loPrint.GetGraphics()
lnWidth = 0
lnHeight = 0
loPrint.OpenDocument("Проверка графического вывода")
loGP.DrawImage(lnGraphics, 30, 30)
```

Обратите внимание на то, что документу при его создании присваивается имя. Запустите пример на выполнение и выберите любой файл, содержащий изображение, которое и будет распечатано на принтере.

Попробуйте вызвать метод `DrawImage`, указав ширину и высоту прямоугольной области. Изображение будет центрироваться внутри этой области, полностью вписываясь в нее.

Как и в случае с предыдущим тестом, выберите миллиметры как единицу измерения для страницы принтера и снова выполните тест. И, наконец, попробуйте создать несколько страниц принтера при помощи метода `NewPage` и что-нибудь нарисовать или напечатать на них.

На этом мы закончим изучение особенностей печати на принтере и перейдем к гораздо более интересному разделу — рисованию в окне формы.

Рисование в окне формы

Для того чтобы рисовать в окне формы, нужно иметь связанный с этим окном объект `Graphics`. Этот объект создается функцией `GdipCreateFromHWND`, которая как параметр получает дескриптор окна `HWnd`. Но, к сожалению, не всегда значение свойства `HWnd` формы, доступное в окне **Properties**, соответствует `HWnd` отображаемого окна — дело в том, что окна формы устроены гораздо сложнее, чем кажется на первый взгляд. Образно говоря, на форме может существовать несколько наложенных друг на друга окон (клиентские окна типа `WCLIENTWINDOW`), и только на одном из них, расположенном "на самом верху", нужно рисовать. Поэтому может возникнуть ситуация, когда вы успешно создаете объект `Graphics`, связанный с окном формы по его дескриптору (свойство `HWnd`), пытаетесь рисовать в этом окне, все функции отрабатывают нормально, но вы ничего не видите. А все дело в том, что вы рисуете в окне, которое "перекрыто" другим клиентским окном.

В каких случаях возникают такие клиентские окна? Например, если форма объявлена как форма верхнего уровня (As Top Level), то для того, чтобы мы могли размещать на ней различные управляющие элементы, создается клиентское окно со своим `HWnd`, расположенное над основным окном и "загораживающее" его. Иная ситуация возникает, когда форма имеет полосы прокрутки (и неважно, видны они в данный момент или нет). Полосы прокрутки являются самостоятельными окнами (со своими `HWnd`), но на них, перекрывая, наложено еще одно клиентское окно! И только тогда, когда ваша форма не является формой верхнего уровня и на ней отсутствуют полосы прокрутки, ее свойство `HWnd` является именно тем дескриптором, который нужно передавать объ-

екту `Graphics`. Во всех остальных случаях вы должны определить правильный `HWnd` окна.

Существует еще одна проблема. Дело в том, что, используя GDIPlus, вы рисуете непосредственно в окне формы, поэтому все нарисованное вами затирается, если над формой перемещается другое окно. Если вы пробовали рисовать на форме при помощи методов `Draw`, `Circle` или `Box`, то наверняка сталкивались с таким эффектом. Другое дело, если изображение выводится в управляющем элементе, например, в `ImageBox`. Все визуальные управляющие элементы устроены таким образом, что они при необходимости сами себя перерисовывают. При использовании GDIPlus вы должны перехватывать сообщение Windows, посылаемое окну в момент перемещения над ним другого окна, и повторно рисовать все нарисованное ранее.

В этом разделе вы узнаете, как решить перечисленные выше проблемы.

Определение "правильного" *HWnd*

Для того чтобы найти дескриптор "самого верхнего" окна в так называемой Z-последовательности клиентских окон, используется Windows API-функция `GetWindow`. Вы должны вызвать эту функцию в том случае, если ваша форма имеет полосы прокрутки (свойство `ScrollBars` имеет значение, большее нуля) или если это форма верхнего уровня (свойство `ShowWindow` равно двум). В следующем фрагменте кода демонстрируется последовательность действий для определения "правильного" `HWnd`:

```
#DEFINE GW_CHILD 5
DECLARE Long GetWindow IN Win32API Long, Long
hWnd = thisform.hWnd
IF thisform.ShowWindow = 2
   hWnd = GetWindow(hWnd, GW_CHILD)
ENDIF
IF thisform.ScrollBars > 0
   hWnd = GetWindow(hWnd, GW_CHILD)
ENDIF
```

Как видите, если форма является формой верхнего уровня, и на ней имеются полосы прокрутки, то функция `GetWindow` вызывается дважды. Последнее присвоенное переменной `hWnd` значение и будет правильным `HWnd` верхнего клиентского окна формы.

Перехват оконных сообщений

Возможно, вы знаете, что в Windows управление окнами происходит посредством обмена сообщениями. Например, когда вы щелкаете мышью по области экрана, то в ответ на событие "нажата левая кнопка мыши" Windows определяет, по какому окну вы щелкнули (т. е. в область какого окна попадают координаты мыши), и посылает этому окну сообщение о том, что по нему "щелкнули" левой кнопкой мыши. В классе, связанном с этим окном, существуют специальные методы, цель которых — обрабатывать полученные от Windows сообщения. Объект — экземпляр оконного класса имеет внутри себя скрытый от нас метод, получающий оконные сообщения и выпол-

няющий команду, аналогичную DO CASE, в которой в соответствии с кодом сообщения выбирается тот или иной метод класса для его обработки. Примерно то же происходит и при работе с клавиатурой — при нажатии на клавишу Windows "смотрит", какое окно в настоящий момент является активным, и посылает ему сообщение, в теле которого находится код нажатой клавиши.

Сообщение Windows представляет собой структуру, объявление которой в C++ выглядит следующим образом:

```
Typedef struct tag MSG {
   HWND    hwnd;
   UINT    msg;
   WPARAM  wParam;
   LPARAM  lParam;
   DWORD   time;
   POINT   pt;
} MSG
```

Описание полей структуры MSG:

- *hwnd* — это целое число, содержащее дескриптор окна, которому посылается сообщение;
- *msg* — идентификатор сообщения. Windows резервирует для собственных нужд 1024 сообщения с идентификаторами от 0x0000 до 0x03FF (обычно для обозначения значений идентификаторов сообщений используют шестнадцатеричный формат). Все сообщения, имеющие идентификаторы, начинающиеся с 0x0400, являются пользовательскими;
- *wParam* — целое число, которое может содержать дополнительный параметр, используемый обработчиком этого сообщения (например, код нажатой клавиши);
- *lParam* — дополнительный параметр, зависящий от типа сообщения; обычно это указатель на область памяти, в которой расположена дополнительная связанная с сообщением информация;
- *time* — значение системных часов во время генерации сообщения;
- *pt* — указатель на структуру POINT, содержащую координаты мыши.

В подавляющем числе достаточно анализировать первые четыре поля структуры.

Для того чтобы решить проблему с затиранием рисунка в окне формы, нам требуется перехватывать всего два сообщения Windows: WM_PAINT и WM_ERASEBKGND.

Сообщение WM_PAINT (его идентификатор — 0x000F) посылается форме после того, как Windows нарисовала пустое окно; получив это сообщение, обработчик события посылает сообщения всем расположенным на форме визуальным управляющим элементам команду "нарисовать себя". Это сообщение посылается после создания окна или его восстановления из свернутого состояния.

Сообщение WM_ERASEBKGND (его идентификатор — 0x0014) посылается форме, если над ней перемещается другое окно. Получив это сообщение, обработчик события перерисовывает ту часть окна, которая вновь стала видимой.

Перехват оконных сообщений стал возможен в девятой версии Visual FoxPro в результате появления нового синтаксиса встроенной функции `BINDEVENT`:

```
BINDEVENT(hWnd, nMessage, oEventHandler, cDelegate [, nFlags])
```

Функция перехватывает посланное окну сообщение, идентификатор которого указан в параметре *nMessage*, и вызывает метод *cDelegate* объекта *oEventHandler* для его обработки. Аргумент *nFlag* определяет тип связывания, его значение по умолчанию равно 0.

В момент получения сообщения оно будет направлено не оконному обработчику, а вашему пользовательскому методу, который содержится в вашем пользовательском объекте. Этому методу будет послано четыре параметра, содержащие значения полей *hWnd*, *Msg*, *wParam* и *lParam* структуры `MSG`.

В качестве объекта `oEventHandler`, обрабатывающего сообщения, может использоваться сама форма; соответственно, тогда метод `cDelegate` — это один из методов формы. Если в методе `Init` формы мы напишем код:

```
#DEFINE WM_ERASEBKGND  0x0014
BINDEVENT(this.hWnd, WM_ERASEBKGND, this, 'EventHandler', 0)
```

то в результате его выполнения посылаемое окну формы сообщение `WM_ERASEBKGND` будет перехватываться и для его обработки будет вызываться написанный вами метод `EventHandler` формы.

Если в методе `EventHandler` прописан код, перерисовывающий изображение — то проблема решена.

Но тут же возникает еще одна проблема: если вы перехватите сообщение `WM_PAINT` и не ретранслируете его после обработки оконному обработчику сообщений, то на вашей форме вообще ничего не будет нарисовано, ни одного управляющего элемента! С сообщением `WM_ERASEBKGND` можно быть менее осторожным, хотя его так же лучше ретранслировать. Поэтому ваш метод-делегат должен иметь возможность ретрансляции оконных сообщений.

Как же правильно нужно перехватывать, обрабатывать и ретранслировать оконные сообщения? Для этого нам понадобятся две Windows API-функции:

- `GetWindowLong` — функция возвращает адрес скрытого от нас обработчика сообщений окна;
- `CallWindowProc` — посылает поля структуры сообщения оконному обработчику сообщений.

В следующем фрагменте кода показан код, демонстрирующий перехват сообщений `WM_PAINT` и `WM_ERASEBKGND` самой формой:

```
#DEFINE GWL_WNDPROC -4
#DEFINE WM_PAINT 0x000F
#DEFINE WM_ERASEBKGND 0x0014
PUBLIC qnWndProc
qnWndProc = GetWindowLong(thisform.hWnd, GWL_WNDPROC)      BINDEVENT(thisform.hWnd,
WM_PAINT, thisform, 'EventHandler', 0)
BINDEVENT(thisform.hWnd, WM_ERASEBKGND, thisform, 'EventHandler', 0)
```

В форму должен быть добавлен метод с именем EventHandler, которому будет пересылаться перехваченное сообщение; он должен ретранслировать полученные сообщения и вызывать методы для повторного рисования. Этот метод должен содержать примерно такой код:

```
LPARAMETER hWnd, Msg, wParam, lParam
= CallWindowProc(qnWndProc, hWnd, Msg, wParam, lParam)
* Вызовы методов для рисования
```

Функция CallWindowProc вызывает оконный обработчик сообщений по его адресу, передаваемому ей в параметре *qnWmdProc*, и передает ему значения первых четырех полей структуры MSG.

Создание связанного с окном объекта *Graphics*

Если вы полагаете, что мы обошли все подводные камни, то должны вас разочаровать. Объект Graphics при создании "запоминает" размеры клиентской области окна; если вы впоследствии увеличите размеры этого окна, то появившиеся области будут недоступны объекту Graphics. Таким образом, объект Graphics должен создаваться заново всякий раз при изменении размеров окна.

Создается связанный с окном объект Graphics функцией GdipCreateFromHWND. Вот ее объявление в Visual FoxPro:

```
DECLARE Long GdipCreateFromHWND IN Gdiplus.dll ;
        Long hWnd, Long @ nativeGraphics
```

Первый передаваемый функции параметр содержит дескриптор окна HWnd, а во второй передаваемый по ссылке параметр записывается дескриптор созданного объекта Graphics.

Если окно, в котором вы собираетесь рисовать, имеет неизменяемые размеры, то вы можете создать объект Graphics в методе Init формы. Иначе — объект должен многократно создаваться в методе Resize.

Форма для рисования

Создайте новую форму с именем в том же проекте, в котором мы разрабатывали классы GdipImages и GdipPrinter. Это должна быть обычная немодальная форма без полос прокрутки. Добавьте в форму новые свойства:

- защищенное свойство WndProc, в котором будет храниться адрес обработчика сообщений окна;
- свойство oGP — для хранения ссылки на объект — экземпляр класса GdipImages;
- свойство nativeGraphics — для хранения дескриптора объекта Graphics.

Установите начальные значения этих свойств равными нулю. В метод Init введите код, показанный в листинге 23.9.

```
this.oGP = CREATEOBJECT("GdipImages")
this.WndProc = GetWindowLong(thisform.hWnd, -4)
BINDEVENT(this.hWnd, 0x000F, this, 'EventHandler', 0)
BINDEVENT(this.hWnd, 0x0014, this, 'EventHandler', 0)
```

Создавать объект `Graphics` будем в методе `Resize` формы. Вот его код (листинг 23.10).

```
LOCAL lnGraphics
lnGraphics = 0
DECLARE Long GdipCreateFromHWND IN Gdiplus.dll Long, Long @
= GdipCreateFromHWND(this.hwnd, @lnGraphics)
this.nativeGraphics = lnGraphics
```

Для рисования на форме создадим метод `ToDraw`. Этот метод будет вызывать метод `DrawImage` класса `GdipImages` для рисования на форме изображения. Следовательно, перед созданием объекта формы должен быть создан объект — экземпляр класса `GdipImages`, и в память компьютера должно быть загружено изображение.

Код метода `ToDraw` приведен в листинге 23.11.

```
IF this.nativeGraphics != 0
   this.oGP.DrawImage(this.nativeGraphics, 0, 0)
ENDIF
```

Добавьте в форму метод с именем `EventHandler`. Код этого метода показан в листинге 23.12.

```
LPARAMETER hWnd, Msg, wParam, lParam
DECLARE Long CallWindowProc IN WIN32API Long, Long, Long, Long, Long
= CallWindowProc(this.WndProc, hWnd, Msg, wParam, lParam)
this.ToDraw()
```

Теперь осталось решить, каким образом мы будем загружать в объект `GdipImages` изображение. Так как наш пример носит чисто демонстрационный характер, то поручим это методу `DblClick` формы. Таким образом, при двойном щелчке мышью по клиентской области формы должно появляться диалоговое окно `Open`, в котором вы сможете выбрать файл; затем этот файл загружается в память.

Код метода `DblClick` показан в листинге 23.13.

```
LOCAL lcFile
lcFile = GETFILE('JPG|BMP|GIF|PNG")
IF !EMPTY(lcFile)
   this.oGP.LoadFromFile(lcFile)
   this.Resize()
   this.ToDraw()
ENDIF
```

Сохраните форму, присвоив ее имя `Demo1`. Запустите форму на выполнение, убедившись, что по умолчанию установлена папка, в которой находится библиотека vfpgdiplus и файлы формы. Дважды щелкните по форме мышью и выберите файл. Если вы нигде не ошиблись, то изображение появится в клиентской области формы.

Попробуйте перемещать над формой другие окна. Изображение не должно затираться.

Модифицируйте код метода `ToDraw`, указав значения ширины и высоты отображаемой области в вызове метода `DrawImage` объекта `GdipImages`:

```
this.oGP.DrawImage(this.nativeGraphics, this.Width, this.Height)
```

Теперь изображение будет располагаться по центру окна. Правда, при изменении размеров формы будет возникать неприятный эффект "размножения", т. к. изображение перемещается, а его предыдущее отображение не стирается. Устранить этот эффект достаточно просто: добавьте в метод `Resize` команду

```
this.Cls
```

которая будет "стирать" все ранее нарисованное на форме.

Класс для рисования в окне формы

Согласитесь, что не очень-то удобно писать повторяющиеся фрагменты кода для каждой формы, на которой вы должны рисовать. Вероятно, проще иметь класс, который возьмет на себя решение этой задачи для *любой* формы.

Добавьте в библиотеку vfpgdiplus.vcx новый класс с именем GdipWindow, в качестве родительского класса выберите класс `Custom`. Этот класс предназначен для рисования в окне формы, используя методы класса `GdipImages`; поэтому объект — экземпляр этого класса должен создаваться после того, как создан объект — экземпляр класса `GdipImages`.

Добавьте в класс следующие свойства:

- `nativeGraphics` — для хранения дескриптора объекта `Graphics`, связанного с окном;
- `wndProc` — для хранения адреса метода оконного класса, обрабатывающего оконные сообщения;
- `oForm` — для хранения ссылки на форму.

Объявите эти свойства как защищенные и установите их начальные значения в нуль.

Метод *Init*

Этот метод получает ссылку на объект формы. Он определяет "правильный" *HWnd* окна и организует перехват оконных сообщений. Код метода приведен в листинге 23.14.

```
LPARAMETERS toForm
LOCAL hWnd
IF VARTYPE(toForm) = "O" .and. toForm.BaseClass = "Form"
   DECLARE Long GdipCreateFromHWND IN Gdiplus.dll Long, Long @
   DECLARE Long GetWindow IN WIN32API Long, Long
   DECLARE Long GetWindowLong IN WIN32API Long, Long
   DECLARE Long CallWindowProc IN WIN32API Long, Long, Long, Long, Long
   this.oForm = toForm
* Определение "правильного" HWND
   hWnd = toForm.HWND
   IF toForm.ShowWindow = 2
      HWnd = GetWindow(hWnd, 5)
   ENDIF
   IF toForm.ScrollBars > 0
      hWnd = GetWindow(hWnd, 5)
   ENDIF
   this.WndProc = GetWindowLong(hWnd, -4)
   this.hwnd = hWnd
* Перехват сообщений WM_PAINT и WM_ ERASEBKGND
   BINDEVENT(hWnd, 0x000F, this, 'EventHandler', 0) && WM_PAINT
   BINDEVENT(hWnd, 0x0014, this, 'EventHandler', 0) && WM_WM_ERASEBKGND
* Перехват события Resize
   BINDEVENT(toForm, "Resize", this, "ResizeEvent") && Событие Resize
ELSE
   RETURN .f.
ENDIF
```

Кроме сообщений Windows, в методе перехватывается событие `Resize` формы и вызывается метод `ResizeEvent` класса для его обработки.

Метод *EventHandler*

Метод получает перехваченное сообщение Windows и ретранслирует его обработчику оконных сообщений. Основное назначение метода — инициировать процесс перерисовки изображения на форме, для чего из него вызывается метод формы, выполняющий рисование. Вы должны создать этот метод в форме и написать в нем необходимый код.

Добавьте метод с именем `EventHandler` в класс. Код метода приведен в листинге 23.15.

```
LPARAMETER hWnd, Msg, wParam, lParam
= CallWindowProc(this.WndProc, hWnd, Msg, wParam, lParam)
IF this.nativeGraphics = 0
   this.ResizeEvent()
ENDIF
IF VARTYPE(this.oForm) = "O"
   this.oForm.ToDraw()
ENDIF
```

Как видно из листинга, для корректной работы метода вы должны создать на форме метод с именем `ToDraw`, который будет отвечать за рисование. Конечно, это не лучшее решение при создании класса, но, с другой стороны, оно обеспечивает большую гибкость, потому что лучше создать новый метод формы, чем каждый раз модифицировать класс.

Обратите внимание: если объект `Graphics` на момент перехвата сообщения не существует (например, форма только что создана и ей послано сообщение `WM_PAINT`), то вызывается метод `ResizeEvent` для его создания.

Метод *ResizeEvent*

Этот метод вызывается после выполнения метода — обработчика события `Resize` формы. В нем создается связанный с формой объект `Graphics`. Добавьте метод в класс. Его код приведен в листинге 23.16.

```
LOCAL lnGraphics
lnGraphics = 0
IF GdipCreateFromHWND(this.hwnd, @lnGraphics) = 0
   this.nativeGraphics = lnGraphics
ENDIF
```

Последний метод, который нужно добавить в класс, — это метод `GetGraphics`, который будет возвращать дескриптор связанного с формой объекта `Graphics`. Он содержит всего одну команду:

```
RETURN this.nativeGraphics
```

Тестирование

Создайте в проекте форму верхнего уровня (As Top Leval). В методе `Init` введите следующий код:

```
this.oGPWindow = CREATEOBJECT("GdipWindow", this)
IF VARTYPE(this.oGPWindow) != "O"
   RETURN .F.
ENDIF
```

В методе `Unload` очистите очередь сообщений командой

```
CLEAR EVENTS
```

Добавьте в форму метод `ToDraw` и введите в него следующий код:

```
qoGP.DrawImage(this.oGPWindow.GetGraphics(), 0, 0)
```

(предполагается, что перед вызовом формы будет создан объект — экземпляр класса `GdipImages`, ссылка на который будет храниться в глобальной переменной `qoGP`).

Сохраните форму, присвоив ей имя Demo2.

Для тестирования создайте новый программный файл с именем **test18.prg** и введите в него следующий код:

```
PUBLIC qoGP
SET DEFAULT TO (JUSTPATH(SYS(16)))
SET CLASSLIB TO vfpgdiplus
qoGP = CREATEOBJECT("GdipImages")
IF qoGP.LoadFromFile(GETFILE("JPG"))
   DO FORM Demo2
   READ EVENTS
ENDIF
RELEASE qoGP
```

Запустите тестовый пример на выполнение. В появившемся диалоговом окне **Open** выберите файл. После подтверждения выбора на экран будет выведена форма, и на ней нарисовано выбранное изображение.

Динамическое отображение диаграмм

Одна из возможных областей применения GDIPlus — это создание динамически изменяющихся диаграмм и графиков. Представьте себе такую ситуацию, когда диаграмма формируется одновременно с вводом или корректировкой данных в таблице, отображаемой в управляющем элементе `Grid`. Пользователь меняет данные и сразу видит результат в виде графика. В этом разделе мы рассмотрим один из способов решения этой задачи на примере формы с условным названием "Итоги по продажам".

Создайте таблицу с именем **demotable**. Записи этой таблицы должны содержать следующие поля:

- **monthname** — поле типа `Character` длиной 9 символов, предназначено для хранения наименования месяца;
- **year2005** — поле типа `Integer`, предназначено для хранения суммы продаж для каждого месяца 2005 года;
- **year2006** — поле типа `Integer`, предназначено для хранения суммы продаж для каждого месяца 2006 года.

Введите в таблицу двенадцать строк (по числу месяцев), в поле **montname** укажите наименования месяцев (январь, февраль и т. д.), а остальные поля заполните любыми

положительными числами; ограничим максимальное значение суммы продаж в месяце числом 99999 (предположим, что это тысячи рублей).

Создайте программный файл с именем demochart.prg. Введите в него код из листинга 23.17.

```
PUBLIC oDPI
cPath = JUSTPATH(SYS(16))
SET DEFAULT TO (cPath)
SET CLASSLIB TO vfpgdiplus
CLOSE TABLES ALL
* Открываем таблицу DEMOTABLE.DBF
USE demotable IN 0
* Создаем объект GdipImages и в нем — растр размером 510 на 265 пикселов
oGPI = CREATEOBJECT("GdipImages")
oGPI.CreateBitmap(510, 265, 0xFFEEFFEE)
* Рисуем на растре "фон": наименования месяцев и координатные оси
lnTop = 10                && Начальное смещение по вертикали
lnStep = 20               && Приращение
oGPI.CreateSolidBrush(0xFF000000) && Создаем черную кисть
oGPI.CreateFont("Arial", 13, 0)   && Создаем шрифт высотой 13 пикселов
SELECT demotable
* Рисуем на растре наименования месяцев
* с отступом 5 пикселов от левого края
SCAN
   oGPI.DrawString(5, lnTop, 0, 0, ALLTRIM(demotable.monthname))
   lnTop = lnTop + lnStep
ENDSCAN
* Создаем черное перо толщиной 1 пиксел ярко-синего цвета
llStatus = oGPI.CreatePen(1, 0xFF0000FF)
IF llStatus
* Рисуем координатные оси, отступив от левого края растра 70 пикселов
* (это место занято наименованиями месяцев)
   llStatus = oGPI.DrawLine(70,5,70,248)
   llStatus = oGPI.DrawLine(70,248,500,248)
ENDIF
IF llStatus
* Загружаем форму, на которой будем рисовать диаграмму
   DO FORM DemoChart
ELSE
* Если имела место ошибка, то сообщаем ее код
   = MESSAGEBOX("Ошибка" + STR(oGPI.GetStatus()))
ENDIF
RELEASE oGPI
```

В этом коде создается растр, который будет использоваться в качестве "фонового рисунка" диаграммы, а в глобальной переменной *oGPI* запоминается ссылка на объект — экземпляр класса `GdipImages`.

Следующий шаг — создание формы **DemoChart**. Эта форма должна быть модальной. Установите размеры клиентской области формы равными 725 (ширина) на 275 (высота) пикселов. В принципе вы можете выбрать любые другие размеры; в моем примере получилось так. Разместите на форме управляющий элемент `Grid`, расположенный вплотную у левой границы окна формы; установите его ширину равной 205 пикселам. На пять пикселов правее `Grid` будет располагаться область для вывода диаграммы. Свяжите `Grid` с таблицей demotable.dbf; запретите редактирование первой колонки (с именами месяцев).

Добавьте в форму свойство `oGPW`, которое будет использоваться для хранения ссылки на объект — экземпляр класса `GdipWindow`.

Откройте на редактирование метод `Init` формы и введите в него следующий код:

```
this.oGPW = CREATEOBJECT("GdipWindow", this)
IF VARTYPE(this.oGPW) != "O"
   RETURN .F.
ENDIF
```

Как вы помните, класс `GdipWindow` предполагает наличие у формы метода `ToDraw`, который и выполняет рисование. Добавьте этот метод в форму. Его код приведен в листинге 23.18.

```
LOCAL lnGraphics, llStatus, lnMax, lnValue, lnScale, ;
      lnTop, lnLeft, lnStep
* Копируем таблицу в курсор и находим
* максимальное значение для полей year2005 и year2006
STORE 0 TO lnMax, lnValue, lnScale, lnTop, lnLeft, lnStep
SELECT * FROM demotable INTO CURSOR t_demo
GO TOP
SCAN
   IF lnMax < t_demo.year2005
      lnMax = t_demo.year2005
   ENDIF
   IF lnMax < t_demo.year2006
      lnMax = t_demo.year2006
   ENDIF
ENDSCAN
* Ширина области для рисования гистограммы равна 420 пикселов
lnRegion = 420
lnScale = lnRegion / lnMax    && Коэффициент масштабирования
lnTop = 12                    && Начальное смещение от верхней границы
                              && области рисования
lnStep = 20                   && Шаг приращения
* Левая граница гистограмм вычисляется так:
* смещение до области вывода растра (210 пикселов) + смещение до оси
* координат X на растре (70 пикселов — см. demochart.prg) + 1 пиксел
lnLeftRegion = 281            && Левая граница гистограмм
```

```
* Копируем в lnGraphics дескриптор объекта Graphics, связанного с окном
lnGraphics = this.oGPW.GetGraphics()
* Рисуем в окне растр; он используется как фоновый рисунок
oGPI.DrawImage(this.oGPW.GetGraphics(), 210, 3)
* Изменяем цвет пера на зеленый; перо было создано ранее в demochart.prg
oGPI.SetPenColor(0xFF00AA00)
* Рисуем столбцы гистограммы; каждый столбец шириной 11 пикселов
GO TOP
SCAN
   oGPI.SetColorSolidBrush(0xFF0099FF) && Синяя кисть
   oGPI.FillRectangle(lnLeftRegion, lnTop, t_demo.year2005 * lnScale, ;
                      11, lnGraphics)
   oGPI.DrawRectangle(lnLeftRegion, lnTop, t_demo.year2005 * lnScale, ;
                      11, lnGraphics)
   oGPI.SetColorSolidBrush(0xCC55EE00) && Зеленая чуть прозрачная кисть
   oGPI.FillRectangle(lnLeftRegion, lnTop+6, t_demo.year2006 * lnScale, ;
                      11, lnGraphics)
   oGPI.DrawRectangle(lnLeftRegion, lnTop+6, t_demo.year2006 * lnScale, ;
                      11, lnGraphics)
   lnTop = lnTop + lnStep
ENDSCAN
* Рисуем вертикальные линии разметки.
* Будет выведено 10 вертикальных линий разметки
lnStep = lnRegion / 10
oGPI.SetPenColor(0x88AAAAAA) && Меняем цвет пера на светло-серый
FOR lnLeft = lnLeftRegion+lnStep TO 10*(lnLeftRegion+lnStep) STEP lnStep
   oGPI.DrawLine(lnLeft, 10, lnLeft, 250, lnGraphics)
ENDFOR
* Рисуем текст под разметкой (числовые значения)
oGPI.SetColorSolidBrush(0xFF000000) && Меняем цвет кисти на черный
* Создаем объект StringFormat и устанавливаем режим центрирования текста
oGPI.CreateStringFormat()
oGPI.SetStringFormatParameter(1)
* Определяем инкремент для значений разметки
lnMax = lnMax / 10
* В цикле выводим значения разметки
lnLeft = 255
FOR i = 1 TO 11
   oGPI.DrawString(lnLeft, 253, 50, 16, LTRIM(STR(lnValue)), lnGraphics)
   lnLeft = lnLeft + lnStep
   lnValue = lnValue + lnMax
ENDFOR
```

Сохраните форму и запустите файл demochart.prg на выполнение. Вы должны увидеть примерно следующее (рис. 23.3).

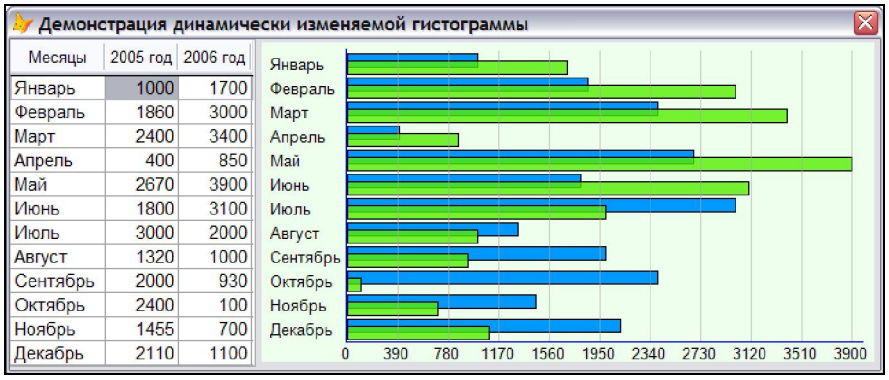


Рис. 23.3. Вид формы **DemoChart**

Измените несколько значений полей в таблице. Гистограмма тот час же изменится.

> ***Замечание***
>
> Управляющий элемент `Grid` при перемещении по его строкам постоянно посылает Windows многократные сообщения `WM_ERASEBKGND`, адресованные окну, в котором этот `Grid` отображается. Поэтому, с одной стороны, при перемещении по строкам таблицы гистограмма "мерцает" из-за того, что окно формы закрашивается фоновым цветом, а с другой стороны, не требуется вызывать метод `ToDraw` формы, например, из событий `LostFocus` текстовых боксов, потому что этот метод и так будет вызван из метода `EventHandler` объекта `GdipWindow`.

Применение GDIPlus в отчетах

В девятой версии Visual FoxPro система отчетов полностью переработана. Теперь во время выполнения отчет можно "связать" с объектом `ReportListener`. Этот объект не только привносит в отчет элементы объектно-ориентированного подхода; он позволяет динамически управлять отчетом. В принципе, при использовании этого объекта вы можете получить отчет, совершенно не похожий на тот, который был создан в Конструкторе отчетов.

Объект `ReportListener` "следит" за выводом отчета и генерирует различные события, обрабатывая которые можно управлять содержимым печатаемого документа. Но основное достоинство объекта `ReportListener` — это то, что он предоставляет нам дескриптор объекта `Graphics`!

События, методы и свойства объекта *ReportListener*

Из достаточно большого количества событий, методов и свойств объекта `ReportListener` мы рассмотрим только тот минимальный набор, который необходим для использования этого объекта совместно с GDIPlus.

Событие BeforeBand возникает перед обработкой каждой очередной записи FRX-файла. Метод BeforeBand получает два параметра:

- BandObjCode — содержит одно из значений, определяющих раздел отчета. Эти значения перечислены в табл. 23.2;
- FRXRecno — номер записи отчета (FRX файла).

Таблица 23.2. *Значения параметра* BandObjCode

BandObjCode	Раздел отчета	BandObjCode	Раздел отчета
0	Title	6	Column Footer
1	Page Header	7	Page Footer
2	Column Header	8	Summary
3	Group Header	9	Detail Header
4	Detail	10	Detail Footer
5	Group Footer		

Событие EvaluateContents возникает для каждого объекта отчета (типа полей таблицы, выводимой в отчет). Так, если вы выводите в отчет по десять полей в строке, то на одно событие BeforeBand произойдет десять событий EvaluateContents.

Метод EvaluateContents получает два параметра:

- FRXRecno — номер записи отчета (FRX-файла);
- ObjProperties — объект класса Empty, содержащий ряд свойств, которые перечислены в табл. 23.3.

Таблица 23.3. *Свойства объекта* ObjProperties

Свойство	Описание
reload	Если вы собираетесь изменить способ вывода поля, то для сообщения об этом объекту ReportListener установите его в "истину" (.T.)
text	Это свойство содержит текст, если тип данных для свойства value — символьный

Таблица 23.3 *(окончание)*

Свойство	Описание
value	Это свойство содержит значение типа Variant (вы можете определить тип переменной, используя функцию VARTYPE). Если value имеет символьный тип, то это означает, что текст нужно получить из свойства text объекта
fontname	Строка. Содержит имя шрифта

`Fontstyle, fontsize`	Целочисленные значения. Определяют стиль и высоту шрифта
`fillred, fillblue, fillgreen, penred, penblue, pengreen`	Свойства определяют цвет кистей и перьев, используемых для рисования поля отчета. Могут принимать значения от 0 до 255
`fillalpha`	Свойство определяет прозрачность кисти
`penalpha`	Свойство определяет прозрачность пера

Метод `Render` выполняет построчное рисование элементов отчета. Вызывается для каждого поля рисуемой строки. Метод получает большое количество параметров, в том числе номер записи в FRX-файле, координаты левого верхнего угла, ширины и высоты области рисования поля и некоторые другие. Если вы дадите в этом методе команду `NODEFAULT`, то в отчете ничего нарисовано не будет. При вводе своего кода в этот метод не забудьте вызвать функцию `DODEFAULT()` для выполнения кода класса-родителя.

Методы `GetPageWidth` и `GetPageHeight` возвращают размеры страницы отчета (в точках). Документ отчета имеет разрешение 960 точек на дюйм, или около сорока точек в миллиметре. Методы не имеют параметров.

Свойство `GdiPlusGraphics` содержит дескриптор объекта `Graphics` отчета.

Свойство `ListenerType` определяет, каким образом `ReportListener` управляет выводом отчета. Может принимать значения от –1 до 5. Подробности вы можете найти в справочной документации.

Создание отчета

Запустите Мастер отчетов. В окне **Wizard Selection** выберите **Report Wizard**. На первом шаге мастера выберите таблицу demotable (если вы еще не создали эту таблицу, то сделайте это сейчас, воспользовавшись рекомендациями из разд. *"Динамическое отображение диаграмм" в этой главе*). Включите в отчет все поля таблицы. На втором шаге все оставьте без изменений, на третьем шаге выберите стиль **Ledger**. Завершите выполнение мастера и сохраните отчет в файле demoreport.frx.

Запустите Конструктор отчетов и загрузите в него только что созданный отчет. Добавьте полосу **Summary**. Придайте отчету вид, показанный на рис. 23.4.

Для чего нужна полоса **Summary**? Получив сообщение о начале этой полосы, мы будем рисовать нашу гистограмму. Почему? Потому что эта полоса обрабатывается сразу после завершения полосы **Detail**. Конечно, мы все привыкли, что полоса **Summary** выводится *после* полосы **Detail**. Но это совсем не обязательно! Работая с GDIPlus, мы рисуем *на листе отчета*, и поэтому можем разместить диаграмму в любом месте (но в пределах листа, конечно)! В методе `BeforeBand` объекта `ReportListener` мы всегда имеем возможность перехватить и обработать событие, извещающее нас о том, что пришло время печатать полосу **Summary** — и приступить к рисованию графика или еще чего-нибудь.

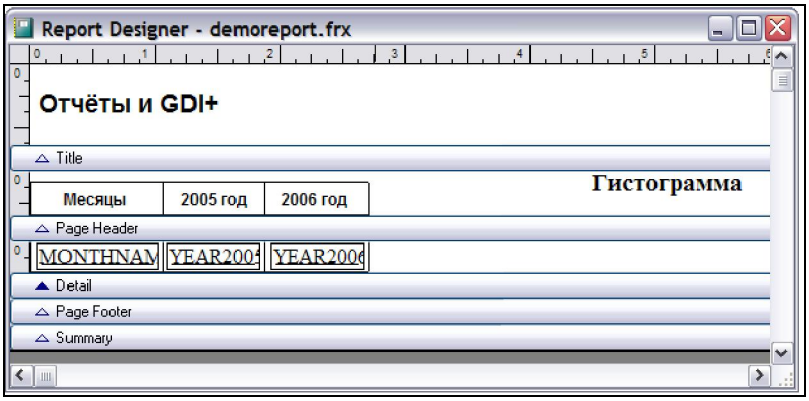


Рис. 23.4. Редактирование отчета **demoreport** в Конструкторе отчетов

Процедура запуска отчета

Во-первых, мы должны связать отчет с объектом `ReportListener`. На самом деле мы свяжем отчет с объектом, созданным на базе нашего собственного класса, который является потомком класса `ReportListener`.

Во-вторых, мы должны создать объект — экземпляр класса `GdipImages` для рисования на листе отчета.

Создайте программный файл с именем demoreport.prg и введите в него код из листинга 23.19.

```
#DEFINE DETAIL_BAND      4
#DEFINE SUMMARY_BAND     8
PUBLIC oGP, oReportListener
cPath = JUSTPATH(SYS(16))
SET DEFAULT TO (cPath)
SET CLASSLIB TO vfpgdiplus
* Создаем объект GdipImages
oGP = CREATEOBJECT("gdipimages")
* Создаем объект ReportListener из нашего класса GdipDemoReport
oReportListener = CREATEOBJECT("GdipReport")
oReportListener.ListenerType = 1
REPORT FORM demoreport PREVIEW OBJECT oReportListener
RELEASE oGP, oReportListener
```

Отчет будет выводиться в окно предварительного просмотра.

Класс *GdipDemoReport*

Если вы хотите управлять отчетом при помощи объекта `ReportListener`, то, как правило, вам придется создавать специализированные классы, наследуемые от класса `ReportListener`, и "заточенные" под конкретный отчет. Попытка создать универсальный класс "на все случаи" вряд ли увенчается успехом. Так, в нашем случае мы будем рисовать гистограмму (подобную показанной на рис. 23.3). В другом отчете вы захотите выводить строки заголовка вертикально или под углом, в третьем отчете... Ну мало ли чего вы еще захотите!

Поэтому приступим к созданию пользовательского класса `GdipDemoReport`. Так как это не универсальный класс, то мы допишем его код в программный модуль demoreport.prg — в тот самый, где мы уже поместили процедуру запуска отчета.

Итак, добавьте в этот файл команду:

```
DEFINE CLASS GdipReport As ReportListener
```

Свойства класса *GdipDemoReport*

Вот описание необходимых свойств:

```
lDetail = .f.   && Флаг, определяющий, что обрабатывается полоса Detail
nLines = 0      && Счетчик количества строк в полосе Detail
nColumns = 0    && Счетчик количества объектов в строке
DIMENSION aDates[12,3] && Массив, в который копируются значения полей
```

> ***ЗАМЕЧАНИЕ***
>
> Вероятно, вы обратили внимание на то, что размеры массива указаны при его объявлении. Этот массив должен содержать то количество строк, которые выводятся в полосе **Detail**, а количество столбцов равно количеству полей. К сожалению, изменение размеров этого массива внутри метода `EvaluateContents` (там в этот массив заносятся данные) дает некорректный результат. В общем случае, размеры этого массива должны соответствовать размеру печатаемой в отчете таблицы.

Следующая группа свойств используется для определения положения области рисования гистограммы

```
nTop = 0        && Верхняя граница области рисования
nLeft = 0       && Левая граница области рисования
lStart = .f.    && Флаг, показывающий, что верхняя граница определена
```

Метод *BeforeBand*

Этот метод будет определять тип обрабатываемой полосы отчета. Если это `Detail`, то устанавливается в "истину" свойство `lDetail`. Если это `Summary`, то вызывается метод `ToDraw` для рисования гистограммы. Для всех остальных полос свойство `lDetail` устанавливается в "ложь" (листинг 23.20).

```
PROCEDURE BeforeBand(tnBandObjCode, tnFRXRecno)
   DO CASE
      CASE tnBandObjCode = DETAIL_BAND  && Если это полоса Detail
         this.nLines = this.nLines + 1  && Инкремент счетчика строк
         this.nColumns = 0              && Сброс счетчика полей строки
         this.lDetail = .t.
      CASE tnBandObjCode = SUMMARY_BAND && Если это полоса Summary
         this.lDetail = .f.
         this.ToDraw()
      OTHERWISE                         && Все остальные полосы
         this.lDetail = .f.
   ENDCASE
ENDPROC
```

Такое построение метода позволяет вносить изменения в полосу **Detail**, но никак не влияет на вывод в отчет содержимого других полос.

Метод *EvaluateContents*

В методе проверяется значение свойства `lDetail`, т. е. относятся ли полученные данные к полосе **Detail**. Если да, то вычисляется номер поля; его значение заносится в массив **aDates** (листинг 23.21).

```
PROCEDURE EvaluateContents(tnFRXRecno, toProps)
   LOCAL lcValue
   IF this.lDetail
      this.nColumns = this.nColumns + 1 && Инкремент счетчика полей
      DO CASE
         CASE VARTYPE(toProps.value) = "C"
            this.aDates[this.nLines,this.nColumns] = toProps.text
         CASE VARTYPE(toProps.value) = "N"
            this.aDates[this.nLines,this.nColumns] = toProps.value
      ENDCASE
   ENDIF
ENDPROC
```

Метод *Render*

Напомним, что именно этот метод "рисует" отчет. В нашем классе он используется для вычисления границы области рисования гистограммы.

Если вы хотите полностью изменить вид полосы, то дайте в нем команду `NODEFAULT` — в этом случае данные из полосы, сформированные в Конструкторе отчетов, рисоваться не будут. В нашем примере мы "разрешаем" нарисовать таблицу в полосе **Detail**, поэтому вызывается функция `DODEFAULT()`, которая заставляет выполнить код класса-родителя (листинг 23.22).

```
PROCEDURE Render(nFRXRecNo, nLeft, nTop, nWidth, nHeight, ;
                 nObjectContinuationType, cContentsToBeRendered, ;
                 GDIPlusImage)
   DODEFAULT(nFRXRecNo, nLeft, nTop, nWidth, nHeight, ;
             nObjectContinuationType, cContentsToBeRendered, ;
             GDIPlusImage)
   IF this.lDetail         && Если полоса — Detail
      IF !this.lStart      && и если это первое поле первой строки
         this.nTop = nTop  && то запоминаем его верхнюю границу
         this.lStart = .t.
      ENDIF
* Далее определяем значение координаты по X границы печатаемой таблицы
      IF this.nLeft < nLeft + nWidth
         this.nLeft = nLeft + nWidth
      ENDIF
   ENDIF
ENDPROC
```

Как и все предыдущие методы, этот метод автоматически вызывается для каждого объекта отчета. Помимо прочего, в метод передаются координаты и размеры областей для рисования объекта.

Метод *ToDraw*

А это уже наш, пользовательский, метод. Он вызывается из метода `BeforeBand`, когда возникает событие начала печати полосы **Summary**. В этом методе мы должны вычислить координаты всех столбцов гистограммы и нарисовать ее (листинг 23.23).

```
PROCEDURE ToDraw
   LOCAL i, lnRow, lnCol, lnMax, lnPageWidth
   lnRow = ALEN(this.aDates,1)
   lnCol = ALEN(this.aDates,2)
   lnMax = 0
* Определение максимального значения
   FOR i = 1 TO lnRow
      IF lnMax < this.aDates[i,2]
         lnMax = this.aDates[i,2]
      ENDIF
      IF lnMax < this.aDates[i,3]
         lnMax = this.aDates[i,3]
      ENDIF
   ENDFOR
* Метод GetPageWidth возвращает значение ширины страницы в точках
* (для отчета используется DPI 960 точек в дюйме)
* Здесь мы отнимаем от полученного значения 500 точек; это будет
* отступ от правой границы страницы
```

```
   lnPageWidth = this.GetPageWidth() — 500
   lnDirect = lnPageWidth — this.nLeft  && Ширина области для рисования
* Левая граница области для рисования должна располагаться
* правее правого края таблицы. Устанавливаем сдвиг в 100 точек (2,5 мм)
   lnLeft = this.nLeft + 100            && Левая граница области
* Определение нижней границы области рисования. Высота области
* принимается равной 2850 точкам. В принципе, ее можно увязать
* с размерами напечатанной таблицы, выполнив необходимые вычисления
* в методе Render
   lnBottom = this.nTop + 2850
* Рисуем координатные оси
   lnGraphics = this.GdiPlusGraphics
   oGP.CreatePen(10)
   oGP.DrawLine(lnLeft, this.nTop — 100, lnLeft, lnBottom, lnGraphics)
   oGP.DrawLine(lnLeft, lnBottom, lnLeft + lnDirect + 200, ;
                lnBottom, lnGraphics)
   lnStep = 2850 / lnRow    && Шаг для размещения 12 столбцов гистограммы
   lnWidthRect = 0.6 * lnStep        && Толщина столбца
   lnScale = lnDirect / lnMax        && Масштабный коэффициент
   oGP.CreateSolidBrush(0xFF0088EE)  && Создаем кисть (любого цвета)
   lnTop = this.nTop
   oGP.SetPenColor(0xFF00AA00)       && Делаем перо зеленым
   FOR i = 1 TO lnRow
      oGP.SetColorSolidBrush(0xFF0088EE)  && Делаем кисть синей
      oGP.FillRectangle(lnLeft, lnTop, this.aDates[i,2] * lnScale, ;
                        lnWidthRect, lnGraphics)
      oGP.SetColorSolidBrush(0xAAAAFF00)  && Делаем кисть зеленой
                                          && и чуть прозрачной
      oGP.FillRectangle(lnLeft, lnTop + 0.5 * lnWidthRect, ;
                    this.aDates[i,3] * lnScale, lnWidthRect, lnGraphics)
      oGP.DrawRectangle(lnLeft, lnTop + 0.5 * lnWidthRect, ;
                    this.aDates[i,3] * lnScale, lnWidthRect, lnGraphics)
      lnTop = lnTop + lnStep
   ENDFOR
   oGP.SetPenColor(0xFFBBBBBB)       && Делаем перо светло-серым
* Рисуем вертикальные линии разметки
   lnStep = lnDirect / 10
   FOR i = lnLeft + lnStep TO lnDirect + lnLeft STEP lnStep
      oGP.DrawLine(i, this.nTop — 100, i, lnBottom, lnGraphics)
   ENDFOR
* Подготовка к рисованию текста
   oGP.CreateStringFormat()          && Создаем объект StringFormat
   oGP.SetStringFormatParameter(1)   && Режим центрирования строки
   oGP.CreateFont("Arial", 120)      && Фонт Arial высотой 4,9 мм
   oGP.SetColorSolidBrush(0xFF000000)&& Делаем кисть черной
   lnLeft = lnLeft — 300
   lnValue = 0
* Рисуем значения разметки
   FOR i = 1 TO 11
      oGP.DrawString(lnLeft, lnBottom + 50, 600, 120, ;
                    LTRIM(STR(lnValue)), lnGraphics)
```

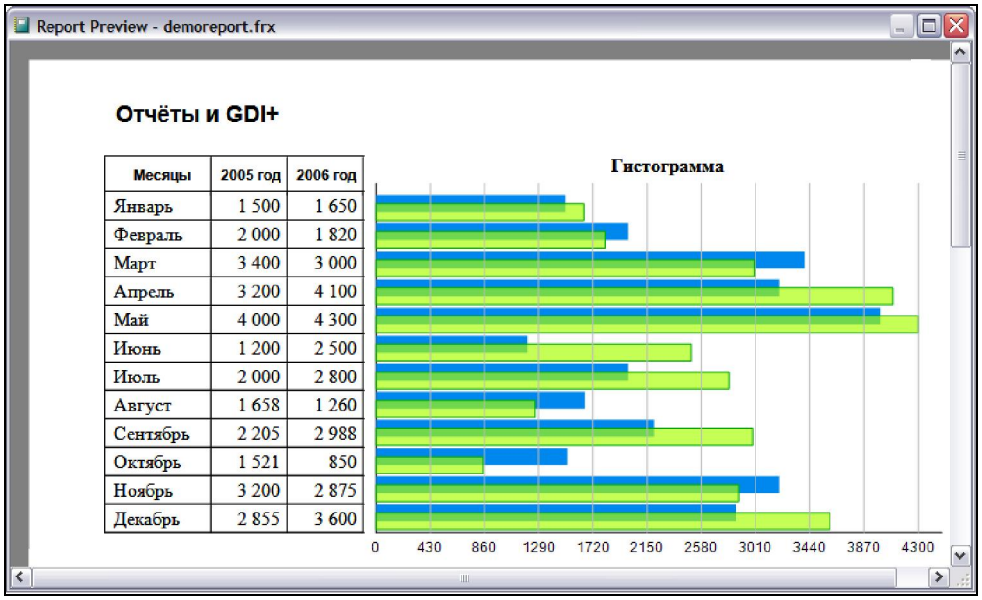
```
      lnLeft = lnLeft + lnStep
      lnValue = lnValue + lnMax / 10
   ENDFOR
ENDPROC
```

Для упрощения кода в отчете не выводятся наименования месяцев; вместо этого столбцы гистограммы располагаются напротив строк таблицы.

Вот и весь класс. Добавьте завершающий аккорд, допишите в файл команду

```
ENDDEFINE
```

Запустите файл demoreport.prg на выполнение. Появится окно просмотра отчета, в котором вы увидите примерно то, что показано на рис. 23.5.



Месяцы	2005 год	2006 год
Январь	1 500	1 650
Февраль	2 000	1 820
Март	3 400	3 000
Апрель	3 200	4 100
Май	4 000	4 300
Июнь	1 200	2 500
Июль	2 000	2 800
Август	1 658	1 260
Сентябрь	2 205	2 988
Октябрь	1 521	850
Ноябрь	3 200	2 875
Декабрь	2 855	3 600

Рис. 23.5. Окно **Report Preview** с загруженным отчетом **demoreport**

Координатные преобразования

В начале предыдущей главы уже были описаны функции `GdipRotateWorldTransform` и `GdipTranslateWorldTransform`, при помощи которых можно выполнять координатные преобразования, а именно: переносить точку начала координат в произвольное место растра и поворачивать координатные оси на заданный угол.

В этом разделе мы рассмотрим простой пример, демонстрирующий применение координатных преобразований. Мы создадим прообраз часов, ограничившись наличием только секундной стрелки.

Создайте в проекте форму с именем **DemoTimer**. Установите ее размеры равными 300 (ширина) на 250 (высота) пикселов. Разместите на форме управляющий элемент `Timer`, установите его свойство `Enabled` в "ложь", а свойству `Interval` присвойте значение 1000, чтобы таймер срабатывал один раз в секунду.

Зафиксируйте размеры формы и удалите кнопки **MinButton** и **MaxButton**.

Добавьте в форму следующие свойства:

- `nativeGraphics` — для хранения дескриптора объекта `Graphics`;
- `nativePen` — для хранения дескриптора пера (им мы будем рисовать шкалу часов и секудную стрелку);
- `nativeBrush` — для хранения дескриптора кисти (ею мы нарисуем кружочек в центре циферблата);
- `cPointsHour` — структура для хранения массива координат точек часовой стрелки;
- `cPointsMinute` — структура для хранения массива координат точек минутной стрелки.

Установите начальные значения свойств `nativeGraphics`, `nativePen` и `nativeBrush` равными нулю.

В этом примере мы не будем пользоваться какими-либо классами, поэтому объявите в методе `Init` формы все необходимые функции (листинг 23.24).

```
DECLARE Long GdipCreateFromHWND IN Gdiplus.dll Long, Long @
DECLARE Long GdipGraphicsClear IN Gdiplus.dll Long, Long
DECLARE Long GdipTranslateWorldTransform IN Gdiplus.dll Long, Single, ;
        Single, Long
DECLARE Long GdipRotateWorldTransform IN Gdiplus.dll Long, Single, Long
DECLARE Long GdipDrawLineI IN Gdiplus.dll Long, Long, Long, Long, ;
        Long, Long
DECLARE Long GdipFillEllipseI IN Gdiplus.dll Long, Long, Long, ;
        Long, Long, Long
DECLARE Long GdipCreatePen1 IN Gdiplus.dll Long, Single, Long, Long @
DECLARE Long GdipSetPenColor IN Gdiplus.dll Long, Long
DECLARE Long GdipSetPenWidth IN Gdiplus.dll Long, Single
DECLARE Long GdipCreateSolidFill IN Gdiplus.dll Long, Long @
DECLARE Long GdipDeleteGraphics IN Gdiplus.dll Long
DECLARE Long GdipDeletePen IN Gdiplus.dll Long
DECLARE Long GdipDeleteBrush IN Gdiplus.dll Long
DECLARE Long GdipDrawClosedCurveI IN Gdiplus.dll Long, Long, String, Long
DECLARE Long GdipSetSmoothingMode IN Gdiplus.dll Long, Long
DECLARE Long GetLocalTime IN WIN32API String @
* Структуры, описывающие массивы размеров часовой и секундной стрелок
thisform.cPointsHour = BINTOC(0, "4RS") + BINTOC(-90, "4RS") + ;
                       BINTOC(-10, "4RS") + BINTOC(10, "4RS") + ;
                       BINTOC(10, "4RS") + BINTOC(10, "4RS")
```

```
thisform.cPointsMinute = BINTOC(0, "4RS") + BINTOC(-115, "4RS") + ;
                         BINTOC(-10, "4RS") + BINTOC(5, "4RS") + ;
                         BINTOC(10, "4RS") + BINTOC(5, "4RS")
```

Последняя объявленная функция, `GetLocalTime`, подробно рассмотрена в *главе 19*. Она необходима для получения значений часов, минут и секунд.

В свойствах `cPoinsHour` и `cPoinsMinute` запоминаются координаты точек для рисования часовой и минутной стрелок. Каждая стрелка описывается тремя точками, по которым рисуется замкнутый сплайн; в системе координат стрелки направлены вертикально вверх. Перемещение стрелок по циферблату часов будет выполняться не за счет пересчета значений их координат, а за счет поворота самих осей координат.

Событие `Destroy` возникает перед уничтожением формы. В методе, обрабатывающем это событие, необходимо удалить все созданные объекты GDIPlus (листинг 23.25).

```
= GdipDeleteGraphics(this.nativeGraphics)
= GdipDeletePen(this.nativePen)
= GdipDeleteBrush(this.nativeBrush)
```

Где лучше всего создавать связанный с формой объект `Graphics`? Вероятно, в методе, обрабатывающем событие `Paint`. Событие `Paint` происходит, когда форма уже выведена на экран. Одновременно с созданием объекта `Graphics` также создадим перо и кисть. В этом же методе при помощи функции `GdipTranslateWorldTransform` перенесем оси координат в центр формы (листинг 23.26).

```
LOCAL lnGraphics, lnPen, lnBrush
lnGraphics = 0
IF GdipCreateFromHWND(this.HWnd, @lnGraphics) = 0
   this.nativeGraphics = lnGraphics
   = GdipSetSmoothingMode(lnGraphics, 4)
   lnPen = 0
   IF GdipCreatePen1(0, 1, 0, @lnPen) = 0
      this.nativePen = lnPen
      lnBrush = 0
      IF GdipCreateSolidFill(0xFF00FF00, @lnBrush) = 0
         this.nativeBrush = lnBrush
         = GdipTranslateWorldTransform(lnGraphics, this.width/2, ;
                                       this.height/2, 0)
         this.Timer1.Timer()
         this.Timer1.Enabled = .t.
      ENDIF
   ENDIF
ENDIF
```

В методе функцией GdipCreateFromHWND создается объект Graphics, связанный с окном формы. Так как мы создали обычную немодальную форму без полос прокрутки, то нам не требуется определять "истинный" hWnd верхнего окна.

При помощи функции GdipSetSmoothingMode включаем режим антиалиасинга — все рисуемые линии не будут иметь характерных зубцов.

Функция GdipCreatePen1 создает перо, которым мы будем рисовать контуры стрелок часов, а функция GdipCreateSolidFill создает кисть ярко-зеленого цвета; этой кистью мы будем рисовать ось, на которой "насажены" стрелки часов.

Теперь самое главное: при помощи функции GdipTranslateWorldTransform центр осей координат переносится в центр окна формы. Надеемся, вы обратили внимание на то, что в методе Init координаты точек стрелок заданы отрицательными числами; сделано это как раз с учетом того, что центр оси координат будет смещен, и поэтому стрелки будут находиться внутри окна формы.

Далее вызывается метод Timer объекта-таймера Timer1. В этом методе собственно и выполняется рисование циферблата и стрелок, а вызываем мы его здесь только для того, чтобы после появления формы на экране целую секунду не ждать, когда же на ней появится изображение часов.

Откройте для редактирования метод Timer управляющего элемента Timer1 и введите в него код из листинга 23.27.

```
LOCAL i, lnHour, lnMinute, lnSecond, lcSystemTime
* Все стираем и заливаем форму однотонным цветом
= GdipGraphicsClear(thisform.nativeGraphics, 0xFFEEFFEE)
STORE 0 TO lnHour, lnMinute, lnSecond
* Формируем структуру SystemTime и вызываем функцию GetLocalTime
* для получения текущего значения часов, минут и секунд
lcSystemTime = REPLICATE(CHR(0),16)
= GetLocalTime(@lcSystemTime)
lnHour = CTOBIN(SUBSTR(lcSystemTime,9,2),'2RS')
lnMinute = CTOBIN(SUBSTR(lcSystemTime,11,2),'2RS')
lnSecond = CTOBIN(SUBSTR(lcSystemTime,13,2),'2RS')
* Корректируем значение часов
IF lnHour > 12
   lnHour = lnHour - 12
ENDIF
lnHour = INT(5 * lnHour + lnMinute / 12)
* Цикл на 60 итераций
FOR i = 0 TO 59
* Рисуем минутные отметки шкалы циферблата
   = GdipSetPenWidth(thisform.nativePen, 4)
   = GdipSetPenColor(thisform.nativePen, 0xFF00AA00)
   = GdipDrawLineI(thisform.nativeGraphics, thisform.nativePen, ;
                   117, 0, 120, 0)
   IF MOD(i, 5) = 0 && Рисуем часовые отметки циферблата
```

```
      = GdipSetPenWidth(thisform.nativePen, 3)
      = GdipSetPenColor(thisform.nativePen, 0xFF0000AA)
      = GdipDrawLineI(thisform.nativeGraphics, thisform.nativePen, ;
                      110, 0, 125, 0)
   ENDIF
   IF lnHour = i   && Рисуем часовую стрелку
      = GdipSetPenWidth(thisform.nativePen, 2)
      = GdipSetPenColor(thisform.nativePen, 0xFF0000AA)
      = GdipDrawClosedCurveI(thisform.nativeGraphics, ;
                             thisform.nativePen, thisform.cPointsHour, 3)
   ENDIF
   IF lnMinute = i && Рисуем минутную стрелку
      = GdipSetPenWidth(thisform.nativePen, 2)
      = GdipSetPenColor(thisform.nativePen, 0xFF0000AA)
      = GdipDrawClosedCurveI(thisform.nativeGraphics, ;
                          thisform.nativePen, thisform.cPointsMinute, 3)
   ENDIF
   IF lnSecond = i && Рисуем секундную стрелку
      = GdipSetPenWidth(thisform.nativePen, 2)
      = GdipSetPenColor(thisform.nativePen, 0xFFAA6600)
      = GdipDrawLineI(thisform.nativeGraphics, thisform.nativePen, ;
                      0, -125, 0, 30)
   ENDIF
* Поворачиваем оси координат на 6 градусов (360 / 60 = 6)
   = GdipRotateWorldTransform(thisform.nativeGraphics, 6, 0)
ENDFOR
* Рисуем часовую ось
= GdipFillEllipseI(thisform.nativeGraphics, thisform.nativeBrush, ;
                   -6, -6, 12, 12)
```

Сохраните форму и запустите ее на выполнение. Вы должны увидеть нечто подобное показанному на рис. 23.6. Часы должны "ходить".

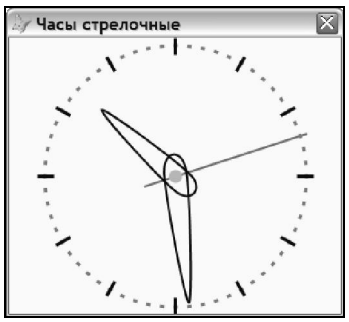


Рис. 23.6. Форма **DemoTimer** в работе

Обратите внимание на то, для всех графических примитивов указаны одни и те же координаты, но выводятся эти примитивы в разных позициях в результате поворота координатных осей функцией `GdipRotateWorldTransform`.

Манипуляции с цветом

В *главе 21* вы уже встречались с описанием функций `GdipDrawImageRectRect` и `GdipDrawImageRectRectI`, которые копируют выделенную область изображения на указанное место растра. Особенностью этих функций является то, что в процессе рисования они используют объект `ImageAttributes`. В этом разделе мы рассмотрим, как с помощью этого объекта можно менять баланс цветов изображения.

Объект *ImageAttributes*

Создается объект `ImageAttributes` функцией `GdipCreateImageAttributes`. Вот ее объявление:

```
DECLARE Long GdipCreateImageAttributes IN Gdiplus.dll ;
        Long @ nativeImageAttributes
```

где `nativeImageAttributes` — возвращаемый дескриптор объекта `ImageAttributes`.

Функция `GdipDisposeImageAttributes` уничтожает этот объект. Вот ее объявление:

```
DECLARE Long GdipDisposeImageAttributes IN Gdiplus.dll ;
        Long ImageAttributes
```

где `LongImageAttributes` — дескриптор объекта `ImageAttributes`.

Объект `ImageAttributes` позволяет выполнять множество операций над изображением, позволяя создавать многофункциональные графические редакторы. Наша же задача предельно проста: научиться изменять цветовую гамму и степень прозрачности для конкретного изображения. Для этого мы должны при помощи функции `GdipSetImageAttributesColorMatrix` передать этому объекту цветовую матрицу размером 5×5, представляющую собой массив, состоящий из пяти строк и пяти столбцов вещественных чисел. Структура матрицы показана на рис. 23.7.

	Red	Green	Blue	Alpha	Доп.
1-я строка	Val	0	0	0	0
2-я строка	0	Val	0	0	0
3-я строка	0	0	Val	0	0
4-я строка	0	0	0	Val	0
5-я строка	Val	Val	Val	Val	1

Рис. 23.7. Матрица преобразования цветов

Каждый столбец матрицы (кроме последнего) определяет, как будет изменена цветовая компонента (Red, Green, Blue) или прозрачность (Alpha). Последний столбец выполняет служебные функции. Ячейки матрицы, в которых на рис. 23.7 указаны числовые значения (0 или 1), не могут содержать никаких других значений, кроме указанных. А вот ячейки, в которых написано слово Val, могут содержать различные

значения (конечно, в допустимых пределах — но об этом чуть позже) и предназначены они для вычисления нового значения каждой цветовой компоненты.

В строках матрицы с первой по четвертую значения Val определяют кратность изменения цвета; значение соответствующей цветовой компоненты умножается на значение ячейки (это значение не может быть отрицательным). Так, если вы хотите в два с половиной раза увеличить значение красной цветовой компоненты, не изменяя значения остальных компонент, то запишите в ячейку на пересечении первой строки и столбца Red значение 2.5, а в ячейки второй, третьей и четвертой строки, там, где на рисунке указано "Val", — единицы. А вот ячейки в пятой строке матрицы (кроме ячейки дополнительного столбца) могут принимать вещественные значения в интервале ±1. Для понимания того, как используется пятая строка матрицы, нужно дать определение цветового вектора.

Значение каждой цветовой компоненты может изменяться в интервале от 0 до 255. Например, пиксел со значениями компонент (0, 128, 0, 255), где три первых числа соответствуют соответственно красному, зеленому и синему цветам, а последнее число указывает степень прозрачности, будет темно-зеленым и непрозрачным. Если мы применим относительное представление цветовых компонент, то параметры этого пиксела можно было бы записать так: (0, 0,5, 0, 1), где минимальное значение компоненты равно нулю, а максимальное — единице. Все остальные значения лежат внутри этого интервала.

Такое представление цвета и называется *цветовым вектором*. Значения из пятой строки цветовой матрицы (без учета дополнительного столбца) суммируются с цветовым вектором каждого пиксела растра.

Следовательно, значения матрицы в строках с первой по четвертую указывают, *во сколько раз* должно измениться значение компоненты, а значения в пятой строке — *на сколько* должно измениться значение компоненты.

Как вы полагаете, как изменится цветовая гамма изображения, если массив будет отображать матрицу, представленную на рис. 23.8?

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Рис. 23.8. Стандартная цветовая матрица

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0.2 & 0.2 & 0.2 & 0 & 1 \end{bmatrix}$$

Рис. 23.9. Матрица, корректирующая цветовую гамму изображения

При использовании такой матрицы цветовая гамма изображения никак не изменится: действительно, значение каждой цветовой компоненты должно быть умножено на единицу, а цветовые векторы пикселов суммируются с нулевыми значениями пятой строки матрицы.

А как изменится цветовая гамма после обработки вот такой цветовой матрицы, как показано на рис. 23.9?

Сначала вычисления выполняются над первыми четырьмя строками матрицы, в результате чего значение красной цветовой компоненты удваивается (но оно не может быть более чем 255), значения цветов остальных компонент и прозрачность остаются без изменения.

При обработке пятой строки матрицы:

- красная компонента цветового вектора увеличивается на 0.2;
- зеленая компонента также увеличивается на 0.2;
- синяя компонента также увеличивается на 0.2;
- прозрачность пиксела не изменяется.

Как применить все вышеизложенное на практике? Нужно создать объект `ImageAttributes` и при помощи функции `GdipSetImageAttributesColorMatrix` передать ему массив, содержащий цветовую матрицу.

Вот объявление функции `GdipSetImageAttributesColorMatrix`:

```
DECLARE Long GdipSetImageAttributesColorMatrix IN Gdiplus.dll ;
        Long nativeImageAttributes, Long type, Long enable, ;
        String colorMatrix, String grayMatrix, Long flags
```

Передаваемые функции параметры:

- *nativeImageAttributes* — дескриптор объекта `ImageAttributes`;
- *type* — этот параметр при работе с изображениями равен 1;
- *enable* — если функции передается цветовая матрица, параметр равен 1; иначе — ноль;
- *colorMatrix* — указатель на массив, содержащий цветовую матрицу;
- *grayMatrix* — указатель на массив, содержащий матрицу оттенков серого (полутона). Если не используется, то должен быть `NULL`;
- *flags* — параметр, определяющий типы изображений и цветов, на которые будут воздействовать цветовые и полутоновые установки. Его значения приведены в табл. 23.4.

Таблица 23.4. *Значения параметра* `flags` *функции* `GdipSetImageAttributesColorMatrix`

Значение `flags`	Описание
0	Полное преобразование
1	Цвета корректируются, но оттенки серого не корректируются. Оттенком серого является любой цвет, цветовые компоненты которого имеют одинаковые значения

2	Цвета корректируются одной матрицей (colorMatrix), а оттенки серого корректируются другой матрицей (grayMatrix)

Пример корректировки цветовой гаммы изображения

В листинге 23.28 показан код примера, позволяющего изменить цветовую гамму изображения (предполагается, что в переменной *nativeImage* находится указатель на область памяти, в которую загружено изображение).

```
DECLARE Long GdipDrawImageRectRectI IN Gdiplus.dll ;
        Long, Long, Long, Long, Long, Long, Long, Long, Long, Long, ;
        Long, Long, Long, Long
DECLARE Long GdipGetImagePixelFormat IN Gdiplus.dll Long, Long @
DECLARE Long GdipCreateImageAttributes IN Gdiplus.dll Long @
DECLARE Long GdipSetImageAttributesColorMatrix IN Gdiplus.dll ;
        Long nativeImageAttributes, Long, Long, String, String, Long
DECLARE Long GdipDisposeImageAttributes IN Gdiplus.dll Long
DECLARE Long GdipGetImageWidth IN Gdiplus.dll Long, Long @
DECLARE Long GdipGetImageHeight IN Gdiplus.dll Long, Long @
DECLARE Long GdipDeleteGraphics IN Gdiplus.dll Long
* Получаем размеры исходного изображения
nSrcX = 0
nSrcY = 0
nSrcWidth = 0
nSrcHeight = 0
= GdipGetImageWidth(nativeImage, @nSrcWidth)
= GdipGetImageHeight(nativeImage, @nSrcHeight)
* Размеры конечного изображения должны быть равны
* размерам исходного изображения
nDstX = nSrcX
nDstY = nSrcY
nDstWidth = nSrcWidth
nDstHeight = nSrcHeight
* Создаем объект Graphics, связанный с исходным изображением
nativeGraphics = 0
GdipGetImagePixelFormat(nativeImage, @nativeGraphics)
* Создаем массив, содержащий цветовую матрицу.
DIMENSION aColorMatrix[5,5]
STORE 0 TO aColorMatrix
aColorMatrix[1,1] = 1      && Кратность для цветов не меняется
aColorMatrix[2,2] = 1
aColorMatrix[3,3] = 1
aColorMatrix[4,4] = 1
aColorMatrix[5,5] = 1
* Установка приращений для цветов
aColorMatrix[5,1] = -0.2   && Уменьшаем интенсивность красного на 20%
aColorMatrix[5,2] = 0      && Зеленый цвет не изменяем
aColorMatrix[5,3] = 0.25   && Увеличиваем интенсивность синего на 25%
```

```
aColorMatrix[5,4] = 1      && Растр полностью непрозрачный
* Заносим массив в строку
cColorMatrix = ''
FOR i = 1 TO 25
   cColorMatrix = cColorMatrix + BINTOC(aColorMatrix[i], 'F')
ENDFOR
* Создаем объект ImageAttributes
nativeImageAttributes = 0
GdipCreateImageAttributes(@nativeImageAttributes)
* Передаем объекту цветовую матрицу
GdipSetImageAttributesColorMatrix(nativeImageAttributes, 1, 1, ;
                                  cColorMatrix, 0, 0)
* Теперь можно рисовать
GdipDrawImageRectRectI(nativeGraphics, nativeImage, ;
                       nDstX, nDstY, nDstWidth, nDstHeight, ;
                       nSrcX, nSrcY, nSrcWidth, nSrcHeight, 0, ;
                       cColorMatrix, 0, 0)
* Убираем "мусор"
GdipDisposeImageAttributes(nativeImageAttributes)
GdipDeleteGraphics(nativeGraphics)
```

Еще несколько функций GDIPlus...

Вы уже познакомились с достаточно большим количеством функций GDIPlus. Но значительно большее их количество осталось за рамками изложенного материала. В этом разделе вы познакомитесь еще с несколькими функциями, которые позволяют:

- создать перо, рисующее кистью;
- изменить форму начала или конца линии (например, нарисовать стрелку);
- получать и изменять цвет отдельного пиксела растра.

Перо, рисующее кистью

До сих пор мы рассматривали перо как инструмент для рисования линий. Возможность рисовать линии кистью позволяет получить очень интересные эффекты. Например, представьте себе перо, рисующее градиентной кистью. Нарисованные им линии будут переливаться всеми цветами радуги!

Создается такое перо функцией `GdipCreatePen2`. Вот ее объявление в Visual FoxPro:

```
DECLARE Long GdipCreatePen2 IN diplus.dll ;
        Long nativeBrush, Single width, Long unit, Long @ nativePen
```

Передаваемые функции параметры:

- *nativeBrush* — дескриптор кисти;
- *width* — толщина пера;
- *unit* — код используемой единицы измерения; как обычно, равен нулю;

- *nativePen* — передаваемый по ссылке параметр, в который записывается дескриптор созданного пера.

Изменение вида начала или конца линии

Функция GdipSetPenStartCap "пририсовывает" элемент к началу линии, а функция GdipSetPenEndCap — к концу. Вот их объявление в Visual FoxPro:

```
DECLARE Long GdipSetPenStartCap IN Gdiplus.dll ;
        Long nativePen, Long startCap
DECLARE Long GdipSetPenEndCap IN Gdiplus.dll ;
        Long nativePen, Long endCap
```

Передаваемые функциям параметры:

- *nativePen* — дескриптор пера;
- *startCap* и *endCap* — вид "наконечника". Возможные значения параметров приведены в табл. 23.5.

Таблица 23.5. *Виды "наконечников" линий*

Значения startCap, endCap	**Описание**
3	Начало (конец) линии заостряется в виде треугольника
17	К началу (концу) линии пририсовывается прямоугольник
18	К началу (концу) линии пририсовывается окружность
19	К началу (концу) линии пририсовывается ромб
20	Начало (конец) линии заканчивается стрелкой

ЗАМЕЧАНИЕ

Для того чтобы эффект проявился, перо должно иметь достаточную толщину (по крайней мере, больше одного пиксела).

Изменение и установка цвета пикселов

Возвращает цвет пиксела функция GdipBitmapGetPixel, а устанавливает — функция GdipBitmapSetPixel. Вот их объявление в Visual FoxPro:

```
DECLARE Long GdipBitmapGetPixel IN Gdiplus.dll ;
        Long nativeImage, Long x, Long y, Long @ color
DECLARE Long GdipBitmapSetPixel IN Gdiplus.dll ;
        Long nativeImage, Long x, Long y, Long color
```

Параметры функций:

- *nativeImage* — указатель на область памяти, в которую загружен растр;
- *x*, *y* — координаты пиксела в растре;

- *color* — получает или передает значение цвета и прозрачности пиксела.

Обзор классов

В процессе изучения GDIPlus мы совместно создали три класса, разместив их в библиотеке классов vfpgdiplus.vcx. В этом разделе приводятся краткие сведения о свойствах и методах этих классов.

Класс *GdipImages*

Создание объекта — экземпляра класса:

```
oGP = CREATEOBJECT("GdipImages" [, lGdipInit])
```

Необязательный параметр логического типа *lGdipInit* определяет необходимость инициализации среды GDIPlus. Если его значение "истина", то GDIPlus инициализируется.

Свойства

`InterpolationMode`

Определяет режим интерполяции. Может принимать значения в интервале от 0 до 7.

`JpegQuality`

Определяет значение качества при сохранении изображения в формате JPEG. Может принимать значения в интервале от 20 до 90.

Методы

`ARGB`

Преобразует значение цвета, возвращаемое функцией `RGB()`, в формат, используемый в GDIPlus. Код метода приведен в листинге 22.1.

```
nGdipColor = oGP.ARGB(RGB(Red,Green,Blue) [, Alpha])
```

`ClipImage`

Вырезает из изображения заданную прямоугольную область и либо копирует ее в файл, либо замещает ею исходное изображение. Код метода приведен в листинге 21.20.

```
lReturn = oGP.ClipImage(Left, Top, Width, Height [, cOutputFileName])
```

`CopyFromClipboard`

Копирует растр из буфера обмена в память. Код метода приведен в листинге 21.14.

```
lReturn = oGP.CopyFromClipboard()
```

`CopyToClipboard`

Копирует изображение из памяти в буфер обмена. Код метода приведен в листинге 21.13.

```
lReturn = oGP.CopyToClipboard()
```

`CreateBitmap`

Создает растр указанного размера и заливает его заданным цветом. Код метода приведен в листинге 21.23.

```
lReturn = oGP.CreateBitmap(width, height [, color])
```

`CreateFont`

Создает шрифт заданной высоты и стиля по его имени. Код метода приведен в листинге 22.37.

```
lReturn = oGP.CreateFont(FontName, width [, style])
```

`CreateGradientBrush`

Создает линейную градиентную кисть. Код метода приведен в листинге 22.10.

```
lReturn = oGP.CreateGradientBrush(PointArray [, WrapMode][, Mode])
```

`CreateHatchBrush`

Создает штриховую кисть. Код метода приведен в листинге 22.13.

```
lReturn = oGP.CreateHatchBrush([Style][, ForeColor][, BackColor]
```

`CreatePen`

Создает перо. Код метода приведен в листинге 22.3.

```
lReturn = oGP.CreatePen(width [, color])
```

`CreateSolidBrush`

Создает одноцветную кисть. Код метода приведен в листинге 22.8.

```
lReturn = oGP.CreateSolidBrush(color)
```

`CreateStringFormat`

Создает объект `StringFormat`, управляющий форматированием текста. Код метода приведен в листинге 22.43.

```
lReturn = oGP.CreateStringFormat([lDirect])
```

`CreateTextureBrush`

Создает текстурированую кисть. Код метода приведен в листинге 22.12.

```
lReturn = oGP.CreateTextureBrush(TextureFileName [, WrapMode])
```

`DeleteBrush`

Удаляет кисть.

```
= oGP.DeleteBrush()
```

`DeleteFont`

Удаляет шрифт.

```
= oGP.DeleteFont()
```

`DeletePen`

Удаляет перо.

```
= oGP.DeletePen()
```

`DeleteStringFormat`

Удаляет объект `StringFormat`. После удаления этого объекта текст не форматируется.

```
= oGP.DeleteStringFormat()
```

`DrawClosedCurve, FillClosedCurve`

Рисует (закрашивает) замкнутый сплайн. Код методов приведен в листингах 22.20 и 22.21.

```
lReturn = oGP.DrawClosedCurve(PointsArray [, ObjGraphics])
lReturn = oGP.FillClosedCurve(PointsArray [, ObjGraphics])
```

`DrawCurve`

Рисует сплайн. Код метода приведен в листинге 22.19.

```
lReturn = oGP.DrawCurve(PointsArray [, ObjGraphics])
```

`DrawEllipse, FillEllipse`

Рисует (закрашивает) эллипс (окружность). Код метода приведен в листинге 22.29.

```
lReturn = oGP.DrawEllipse(x, y, Diameter, Diameter1 [, ObjGraphics])
lReturn = oGP.FillEllipse(x, y, Diameter, Diameter1 [, ObjGraphics])
```

`DrawImage`

Рисует изображение на внешнем устройстве графического вывода или растре. Код метода приведен в листинге 22.46.

```
lReturn = oGP.DrawImage(ObjGraphics, Left, Top [, Width, Height])
```

`DrawImageFromFile`

Рисует на растре изображение, считанное из файла. Код метода приведен в листинге 22.47.

```
lReturn = oGP.DrawImageFromFile(FileName, Left, Top)
```

`DrawLine`

Рисует линию по заданным координатам точек ее начала и конца. Код метода приведен в листинге 22.16.

```
lReturn = oGP.DrawLine(x1, y1, x2, y2 [, ObjGraphics])
```

`DrawLines`

Рисует ломаную линию по координатам, заданным в массиве точек координат. Код метода приведен в листинге 22.17.

```
lReturn = oGP.DrawLines(PointsArray [, ObjGraphuvs])
```

`DrawPie`

Рисует сектор или дугу. Код метода приведен в листинге 22.31.

```
lReturn = oGP.DrawPie(x, y, Diameter, Diameter1, startAngle, ;
                      sweepAngle [, ArcFlag][, ObjGraphics])
```

`FillPie`

Закрашивает сектор. Код метода приведен в листинге 22.32.

```
lReturn = oGP.DrawPie(x, y, Diameter, Diameter1, startAngle, ;
                      sweepAngle [, ObjGraphics])
```

`DrawPolygon, FillPolygon`

Рисует (закрашивает) многоугольник. Код метода приведен в листинге 22.27.

```
lReturn = oGP.DrawPolygon(PoinsArray [, ObjGraphics])
lReturn = oGP.FillPolygon(PoinsArray [, ObjGraphics])
```

`DrawRectangle, FillRectangle`

Рисует (закрашивает) прямоугольник. Код метода приведен в листинге 22.23.

```
lReturn = oGP.DrawRectangle(Left, Top, Width, Height [, ObjGraphics])
lReturn = oGP.FillRectangle(Left, Top, Width, Height [, ObjGraphics])
```

`DrawRectangles, FillRectangles`

Рисует (закрашивает) множество прямоугольников. Код метода приведен в листинге 22.25.

```
lReturn = oGP.DrawRectangles(PointsArray [, ObjGraphics])
```

`DrawString`

Рисует текстовую строку в заданной прямоугольной области. Код метода приведен в листинге 22.38.

```
lReturn = oGP.DrawLine(Left,Top,Width,Height,TextString [, ObjGraphics])
```

`FillImage`

Заливает растр указанным цветом.

```
lReturn = oGP.FillImage([Color])
```

`GetImageResolution`

Возвращает разрешение растра (dpi). Код метода приведен в листинге 21.16.

```
lReturn = oGP.GetImageResolution(@HorDPI, @VertDPI)
```

`GetImageSize`

Возвращает размеры изображения. Код метода приведен в листинге 21.15.

```
lReturn = oGP.GetImageSize(@Widtch, @Height)
```

`GetMeasureString`

Вычисляет размеры строки. Код метода приведен в листинге 22.41.

```
lReturn = oGP.GetMeasureString(String, @Width, @Height [, ObjGraphics])
```

`GetRawFormat`

Определяет графический формат файла. Код метода приведен в листинге 21.17.

```
cFormatName = oGP.GetRawFormat()
```

`GetStatus`

Возвращает код состояния после выполнения метода класса. Положительные значения идентифицируют ошибки, возникшие при выполнении функций GDIPlus, отрицательные — добавленные нами в класс коды ошибок. Перечень добавленных ошибок приведен в табл. 23.6. Если метод возвращает ноль, то метод выполнился без ошибок.

```
nStatusCode = oGP.GetStatus()
```

`GraphicsFromImage`

Создает объект `Graphics`, связанный с загруженным в память изображением.

```
lReturn = oGP.GraphicsFromImage()
```

`LoadFromFile`

Загружает изображение из файла в память. Код метода приведен в листинге 21.2.

```
lReturn = oGP.LoadFromFile(FileName)
```

`LoadFromField`

Загружает изображение из поля таблицы или переменной Visual FoxPro. Код метода приведен в листинге 21.11.

```
lReturn = oGP.LoadFromField(table.fieldname)
```

или

```
lReturn = oGP.LoadFromField(FoxVariable)
```

`ResizeImage`

Изменяет размеры изображения либо сохраняя новое изображение в файле, либо замещая им исходное изображение. Код метода приведен в листинге 21.25.

```
lReturn = oGP.ResizeImage(NewWidth, NewHeight [, OutputFileName])
```

`RotateFlip`

Выполняет отражения и повороты изображения на угол, кратный 90°. Код метода приведен в листинге 21.19.

```
lReturn = oGP.RotateFlip(RotateFlipCode)
```

`SaveToField`

Сохраняет изображение в поле таблицы или переменной Visual FoxPro. Код метода приведен в листинге 21.12.

```
Replace table.fieldname with oGP.SaveToField(GraphicsFormatName)
```

или

```
FoxVariable = oGP.SaveToField(GraphicsFormatName)
```

`SaveToFile`

Сохраняет изображение в файле. Код метода приведен в листинге 21.7.

```
lReturn = oGP.SaveToFile(OutputFileName)
```

`SetColorGradient`

Изменяет начальный и конечный цвета для существующей линейной градиентной кисти. Код метода приведен в листинге 22.11.

```
lReturn = oGP.SetColorGragient(StartColor, EndColor)
```

`SetColorSolidBrush`

Изменяет цвет существующей одноцветной кисти. Код метода приведен в листинге 22.9.

```
lReturn = oGP.SetColorSolidBrush(NewColor)
```

`SetPenColor`

Изменяет цвет существующего пера. Код метода приведен в листинге 22.5.

```
lReturn = oGP.SetPenColor(NewColor)
```

`SetPenStyle`

Изменяет стиль существующего пера. Код метода приведен в листинге 22.4.

```
lReturn = oGP.SetPenStyle(StyleCode)
```

`SetPenWidth`

Изменяет толщину существующего пера. Код метода приведен в листинге 22.6.

```
lReturn = oGP.SetPenWidth(Width)
```

`SetSmoothing`

Устанавливает или отменяет антиалиасинг при рисовании графических примитивов. Код метода приведен в листинге 22.15.

```
lReturn = oGP.SetSmoothing(Flag [, ObjGraphics])
```

`SetStringFormatParameter`

Устанавливает режим выравнивания текста и направление вывода в заданной прямоугольной области. Код метода приведен в листинге 22.44.

```
lReturn = oGP.SetStringFormatParameter(Align [, Direct])
```

`SetTextRendering`

Определяет качество рисования символов (например, можно установить режим `Clear Type`). Код метода приведен в листинге 22.40.

```
lReturn = oGP.SetextRendering(mode [, ObjGraphics])
```

Коды ошибок

Код ошибки, возникшей при выполнении метода класса, можно получить при помощи метода `GetStatus`.

Если `GetStatus` возвращает положительное число, то это означает, что ошибка возникла при выполнении функции GDIPlus. Коды этих ошибок перечислены в табл. 21.1.

Если `GetStatus` возвращает отрицательное число, то эта ошибка обнаружена методом. Коды этих ошибок приведены в табл. 23.6.

Таблица 23.6. *Коды ошибок, обнаруженные методами класса*

Код ошибки	Описание
–1	Нет загруженного в память изображения
–2	Невозможно создать поток для считывания изображения из поля таблицы или переменной Visual FoxPro
–3	Ошибка при работе с буфером обмена Windows
–4	Объект `Graphics` не существует
–5	Перо не существует
–6	Кисть не существует
–7	В метод должен быть передан массив (по ссылке)
–8	Недопустимое количество элементов переданного в функцию массива
–9	Шрифт не существует

Таблица 23.6 *(окончание)*

Код ошибки	Описание
–10	Объект `StringFormat` не существует
–11	Изображение не имеет атрибутов DPI

Класс *GdipPrinter*

Создание объекта — экземпляра класса:

```
oPrinter = CREATEOBJECT("GdipPrinter" [, PrinterName])
```

Если имя принтера опущено, то объект связывается с принтером, установленным по умолчанию.

Метод не имеет открытых свойств.

Методы

`CloseDocument`

Закрывает документ принтера и отправляет его в очередь печати. Код метода приведен в листинге 23.5.

```
lReturn = oPrinter,CloseDocument()
```

`GetGraphics`

Возвращает дескриптор объекта `Graphics`, связанного с принтером.

```
nObjGraphics = oPrinter.etGraphics()
```

`GetStatus`

Возвращает код состояния после выполнения метода класса. Положительные значения идентифицируют ошибки, возникшие при выполнении функций GDIPlus, отрицательные — добавленные нами в класс коды ошибок. Перечень добавленных ошибок приведен в табл. 23.7. Если метод возвращает ноль, то метод выполнился без ошибок.

```
nStatus = oPrinter.GetStatus()
```

`NewPage`

Создает новую страницу в документе принтера. Код метода приведен в листинге 23.4.

```
Leturn = oPrinter.NewPage()
```

`OpenDocument`

Открывает документ принтера. Код метода приведен в листинге 23.3.

```
lReturn = oPrinter.OpenDocument([DocumentName])
```

`SetPageUnit`

Устанавливает единицу измерения для печати на принтере и возвращает размеры листа принтера в выбранных единицах.

```
lReturn = oPrinter(FlagUnit, @WidthPage, @HeightPage)
```

Коды ошибок

Код ошибки, возникшей при выполнении метода класса, можно получить при помощи метода `GetStatus`.

Если `GetStatus` возвращает положительное число, то это означает, что ошибка возникла при выполнении функции GDIPlus. Коды этих ошибок перечислены в табл. 21.1.

Если `GetStatus` возвращает отрицательное число, то эта ошибка обнаружена методом. Коды этих ошибок приведены в табл. 23.7.

Таблица 23.7. *Коды ошибок, обнаруженные методами класса*

Код ошибки	Описание
–1	Нет открытого документа принтера
–2	Не удалось создать страницу в документе принтера
–3	Не удается корректно закрыть документ принтера

Класс *GdipWindow*

Создание объекта — экземпляра класса:

```
oWnd = CREATEOBJECT("GdipWindow", ObjForm)
```

При создании объекта ему передается ссылка на объект формы (thisform).

Для корректной работы класса форма должна иметь метод с именем `ToDraw`, в котором должен быть прописан код для рисования в окне формы.

Класс не имеет открытых свойств.

Единственный открытый метод класса, `GetGraphics`, возвращает дескриптор объекта `Graphics`, связанного с окном формы.

Заключение

Эта глава завершает тему использования GDIPlus в приложениях Visual FoxPro. Ограниченный объем книги не позволил раскрыть все возможности, предоставляемые этой средой. Если вы хотите углубить свои знания в этой области, то для получения необходимой информации следует обратиться к Visual Studio .NET или более поздней; кроме того, вы должны иметь навыки программирования на C++ (или хотя бы уметь понимать коды).

В MSDN для Visual Studio описание GDIPlus находится в разделе **GDI+**, доступ к этому разделу можно получить, открыв раздел **Win32 and COM Development** и в нем — раздел **Graphics and MultiMedia**. Там вы найдете описание классов C++, ориентированных на работу с GDIPlus.

Кроме этого, у вас должны быть заголовочные файлы типа GdiplusXXXX.h, которые вы можете найти в папке Visual Studio: ..\VC\PlatformSDK\Include (для Visual Studio 2005).

Сначала вы изучаете возможности класса C++, а затем открываете соответствующий H-файл и находите в нем описание этого класса. Имейте в виду, что все методы классов C++ многократно перегружены.

Например, для того чтобы определить, какие функции GDIPlus вызываются из методов класса `Graphics C++`, вы должны найти описание этих методов в файле GdiplusGraphics.h.

Все тестовые примеры, а также библиотеку классов `vfpgdiplus` вы сможете найти на прилагаемом к книге компакт-диске.