IdaPro v.6.1 demo: Серьезное испытание

by **Erfaren**
http://erfaren.narod.ru
erfaren@rambler.ru

Введение

Надо отдать должное Ильфаку Гильфанову, он успевает выпускать новые версии своей программы быстрее, чем мы тестировать их ☺. Кроме того, в новой версии сохранилась возможность использовать скрипты и плагины, что не может не радовать. Ведь именно благодаря этим средствам даже демо-версия «Иды» становиться практически полупрофессиональным инструментом. В профессиональной, коммерческой версии, расширяется только набор поддерживаемых средств и возможность долговременной работы с программной базой **idb**. Но для наших целей это пока не актуально, особенно учитывая стоимость программ Ильфака и трудность их приобретения частными лицами. Любая оплачиваемая программа должна рано или поздно приносить либо прибыль своему пользователю, либо удовольствие. Прибыль пока даже не просматривается, а удовольствие овладения коммерческой «Идой» слишком дорогостоящее, чтобы об этом думать всерьез. Я, конечно, не имею в виду пиратские копии «народного хакерского инструмента», ими все пользуются на собственный страх и риск. Причем это касается практически всего коммерческого программного обеспечения. Будь моя воля, я бы требовал выплаты за использование ПО только после того, как оно уже начало приносить прибыль своему владельцу (а не арендатору!), не принимая в расчет цифровой продукт, предназначенный для развлечений. А иначе за что платить, за возможность самообучения?

Однако перейдем к цели нашей статьи. Проведенные, в предыдущих статьях, тесты по **восстановлению исходного кода на уровне ассемблера** для простейших программ, логически приводят к мысли, а насколько хорош наш **IdaPro** для перекомпиляции больших бинарных программ? Конечно, иметь дело с бинарным кодом, защищенным различными навесными защитами либо не имеющим отладочных символов пока что малопривлекательно. Поэтому мы волей-неволей должны выбрать для теста потенциально хорошо восстановимые **dll**-ки или **exe**-шники и достаточно общедоступные, чтобы любой мог повторить наш тест. Естественно, что наиболее подходящие на эту роль файлы из относительной простой и еще достаточно распространенной операционной системы Майкрософт **Windows XP**, тем более, что для ее системных файлов нет проблем с отладочными символами. Ну, а для широко известных, «средней тяжести» программ Майкрософта, естественно предложить для наших исследований что-то вроде знаменитого **Windows Explorer** или **Проводника** по-русски. Если же вести речь о **dll**-ке, то также естественно взять библиотеку общесистемных контролов **comctl32.dll**, тем более, что ее интенсивно использует тот же **Проводник**.

Предварительный обзор объектов исследования

Размер файла **explorer.exe** из **Windows XP, sp.3**, версии **6.00.2900.5512 (xpsp.080413-2105)** составляет **1'034'240** байт, хотя есть сборки той же **ОС** и аналогичной **dll**, но размером **1'721'344** байт. Видимая разница состоит только в размерах ресурсов, например, одни и те же **bmp**-шки могут сильно отличаться своим размером. Интересно, кому понадобилось менять системные **dll**, только ради изменения размеров графических ресурсов ☺?

В любом случае это уже на порядок больше, чем ранее исследуемые нами файлы. Но зато уже это более похоже на реальный проект, хотя очень трудно себе представить, кому может быть интересно восстановление кода **Windows Explorer** ☺, кроме как ради учебных целей? Но если уж исследовать возможности свежайшего релиза «Иды», то чего мелочиться ☺, правда? Тем более, как мы увидим, весьма существенную часть в **Проводнике** составляют ресурсы.

Библиотека **comctl32.dll** под **Windows XP, sp.3** имеет версию **6.0** (**xpsp.080413-2105**) и размер **1'054'208** байт.

Часть 1. explorer.exe

Ресурсы программы explorer.exe

Загрузим нашего «подопытного кролика» в редактор ресурсов **ResourceBuilder v. 3.25** (это вполне функциональная версия в течение месяца работы). Также неплоха, для сравнительного анализа, похожая программа **Resource Tuner v. 1.99.6.1400** (тоже полнофункциональна в течение **30** дней). Но для наших целей первая программа более удобна, так как генерит общий скрипт, а не набор скриптов, как вторая.

После извлечения всех ресурсов из **Проводника**, мы с удивлением обнаруживаем, что они занимают порядка **720 Кб** и **1.4 Мб**, для двух его упоминавшихся вариантов. Таким образом, на бинарный код остается в любом случае примерно **300 Кб**. Ну, это совсем другое дело! Для нас это хороший знак, поэтому двинемся дальше. Забегая вперед, заметим, что **dll**-ка будет уже иметь бинарного кода раза в два больше, чем любой из наших **exe**-шников.

Короче говоря, извлекаем стандартным образом все бинарные ресурсы в папку **res_exe**, соответственно подправляя скрипт файла ресурсов **explorer.rc**. **ResourceBuilder** выругается на строки типа:

```
MENUITEM "", -1, MFT_STRING, MFS_DISABLED
```

Здесь ему не нравиться **-1** и он предлагает выбрать значение от **0** до **65535**. Смотрим, что нам говорит по этому поводу **Resource Tuner**? Он вместо **-1** пишет ее безнаковое двухбайтное значение, т.е. **65535**. Мы можем делать соответствующие переименования, а можем и не делать, так как компилятор ресурсов вполне распознает минус единицу как максимальное **16**-ти битное значение.

Другие замены, которые нам надо сделать, касаются двойных кавычек внутри строк уже окруженных двойными кавычками. Редакторы ресурсов предлагают нам спецсимволы вида **\"** , либо даже двойные кавычки без косой черты, но компилятор ресурсов подобных приколов не понимает. Зато понимает символы вида **\x22** , либо подряд идущие две одиночные кавычки **''** . Поэтому делаем соответствующие исправления «неправильных» кавычек.

Далее, интересный момент, **ResourceBuilder v. 3.25** формирует файл скрипта в формате текстового юникода, что не сразу то и заметишь, работая с современными текстовыми редакторами. А для компилятора ресурсов нужен обычный текст. Чтобы поменять кодировку я использую **ansi** версию великолепного бесплатного редактора **Notepad++ v. 5.9.2** (http://notepad-plus-plus.org/download). Для этого достаточно выбрать пункт меню **Encoding \ Convert to ANSI** и сохранить файл. Сразу видим, что объем файла уменьшился в два раза, как и должно быть.

Другие изменения в ресурсном скрипте касаются добавления туда всех необходимых констант из заголовочных файлов **MS Visual Studio C++**. Что касается манифеста, то этот блок можно оставить либо в шестнадцатеричном виде, либо сослаться на файл вроде **explorer.xml**, либо вообще от него отказаться (раз все работает и без манифеста). Файл **explorer.xml** можно скопировать вручную из **ResourceBuilder**'а либо переименовать соответствующий файл **123.xml**, непосредственно генерируемый (в **ansi**) **Resource Tuner**'ом.

Теперь мы можем скомпилировать наш ресурсный скрипт **explorer.rc** и получить без проблем бинарный файл **explorer.res**.

Подготовка ассемблерного кода

Это уже будет серьезная часть нашей работы. **300 Кб** бинарного кода это на порядок больше того, что мы восстанавливали раньше. Естественно, что появились новые нюансы, а старые существенно возросли.

Например, мы уже отмечали ранее такой бзик «Иды», как **дублирование глобальных имен и использование ключевых слов в качестве имен переменных**. При небольшом количестве этих

казусов с ними можно было справиться вручную. Но в данном случае вручную, за обозримое время не получиться, поэтому нам придется писать **IdaPro**'шные скрипты для решения этой и других проблем.

Другая существенная проблема, с которой нам еще не приходилось всерьез сталкиваться, это **проблема «чанков»** или **«чунков»** (**the function chunks**). Это относительно автономные «куски» функций, которые компилятор (надо думать, в целях оптимизации) распределяет за пределами тел их родных функций. Все это порождает ненужные проблемы при перекомпиляции, так как локальные имена компилятор отслеживает только в пределах основного тела функции.

Третья проблема касается **представления данных как кода** и «неправильного» представления данных, особенно, когда подобных (ручных) изменений очень много.

Как бороться с «неправильными» именами «Иды»? Делать это «точечно» очень затруднительно, ибо нужно будет вновь и вновь возвращаться к исходному листингу и вносить нужные изменения. Поэтому мы предлагаем очень радикальное решение. **Изменить все имена!** Это касается полей перечислений, структур и их полей, глобальных и локальных переменных функций и имен самих функций, что особенно полезно при организации **неманглированных экспортируемых имен**. Очень важное замечание, **делать эти изменения лучше всего в самой «Иде»**, а не в сформированном внешнем листинге кода. Ибо при этом сама «Ида» очень корректно отслеживает все связанные переименования, по всей своей базе. Подобная же работа по переименованию имен во внешнем листинге обычно ведет к ненужному геморрою.

Таким образом, мы наметили себе несколько скриптов, которые должны еще написать. А именно:

enums.idc – Добавляет префикс **"e_"** к полям перечислений;

structs.idc – Добавляет префикс **"s_"** к именам структур и префикс **"f_"** к полям структур. Кроме этого, в именах структур переименовывает все двойные двоеточия **"::"**, на двойные символы подчеркивания **"__"**;

globals.idc – Переименовывает префиксы импортируемых функций с **"__imp_"** на **"_imp_"** (**деманглирование импортируемых имен**) и **деманглирует экспортируемые** (точнее, все **внутренние**) **функции**, т.е. удаляет символ подчеркивания вначале имени и символы **"@##"** в конце, где **"##"** означает некоторое число (байтов). При этом **«особые имена»**, начинающиеся с символов **"_"** (а также имен имеющих символ подчеркивания на «расстоянии» менее **9** символов от начала), **"?"** и **"@"** мы оставляем без изменений. Все остальные (обычные) глобальные переменные мы наделяем префиксом **"g_"**, при условии, что у данного имени такого префикса еще нет. Отметим также, что в нашем служебном файле «Иды» **ida.cfg** переменная **ASCII_PREFIX = "g_"** и **MAX_NAMES_LENGTH = 64**. Данный скрипт генерит также файл протокола (имя запрашивается) проделанной работы.

locals.idc – Добавляет префикс **"l_"** к локальным переменным функций, в которых нет символа подчеркивания и пропускает скрытые служебные переменные с именами **" r"** и **" s"**. Также генерит файл протокола (имя запрашивается) проделанной работы.

Именно в таком порядке и рекомендуется применять наши скрипты, к «распахнутым» (**Unhide all**) листингам перечислений, структур и кода. Для кода лучше предварительно выделить весь листинг, иначе возможно останутся десятки «свернутых» функций во внешнем листинге с бесполезными сообщениями типа: «**PRESS KEYPAD "+" TO EXPAND**».

Но, перед тем как применять данные скрипты, очень желательно «пройтись» по областям данных программы, которые обычно группируются в начале и в конце секции **.text**, в секциях типа **.data** и т.п. И внимательно посмотреть, нет ли среди данных «левого» кода и может быть какие-нибудь представления в байтах или словах и т.п. преобразовать в двойные слова, юникод, строки, структуры либо массивы. Труднее всего преобразовывать большие области данных в двойные слова (как наиболее общий случай). Для этого мы предлагаем скрипт **data.idc**, которые сначала «расформатирует» выделенную область данных, а затем преобразует неопределенные данные в двойные слова. На базе этого скрипта вы можете

писать свои варианты преобразований. Выделенный текст данных легко преобразуется через стандартное меню **IdaPro**, либо пиктограммы в строки, юникод, структуры и массивы. Главное для нас **преобразовать в данные неправильно распознанный код**. Ибо данные могут «работать» как код, а код (в виде ассемблерных инструкций) очень плохо (неоднозначно!) представляет данные, что может привести к ошибкам компиляции и времени выполнения. Также преобразованием данных можно **получить нераспознанные ссылки** на какие-нибудь виртуальные функции и т.п. Они также довольно часто располагаются связными «кусками». Для этого просто смотрим, не напоминают ли различные данные внутренние адреса нашей программы?

Далее применение этих скриптов мы рассмотрим более конкретно, а пока опубликуем их листинги. Думаю, что в этом есть смысл, так как поиск подобных прототипов в Интернете занимает очень много времени.

Скрипт data.idc

```
#include <idc.idc>

static main() {
  auto i, StartEA, FinishEA, Elems, Size;

  StartEA = SelStart();
  FinishEA = SelEnd();
  Size = FinishEA - StartEA;
  Elems = Size/4;

  MakeUnknown(StartEA, Size, DOUNK_SIMPLE);

  for(i = 0; i < Elems; i++) {
    MakeDword(StartEA + 4*i);
  }

  Message("All done!\n");
}
```

Скрипт enums.idc

```
#include <idc.idc>

static main() {
  auto i, j, pos, EnumId, EnumName, NewName, FieldName, EnumQty, EnumSize;

  EnumQty = GetEnumQty();

  for(i = 0; i < EnumQty; i++) {
    EnumId = GetnEnum(i);
    EnumName = GetEnumName(EnumId);
    EnumSize = GetEnumSize(EnumId);

    Message("Enum No. %d is %s. EnumSize = %d\n", i, EnumName, EnumSize);

    for(j = 1; j <= EnumSize; j++) {
      FieldName = GetConstName(EnumId + j);

      Message("  Enum Const No. %d is %s\n", j, FieldName);

      if(substr(FieldName, 0, 2) == "e_")
        continue;

      NewName = "e_" + FieldName;
```

```
      SetConstName(EnumId + j, NewName);
    }

  }

  Message("%d Enums are done!\n", EnumQty);
}
```

Скрипт structs.idc

```
#include <idc.idc>

static main() {
  auto i, pos, StructId, StructName, NewName, FieldOfs, FieldName, StructQty;

  StructQty = GetStrucQty();

  for(i = 0; i < StructQty; i++) {
    StructId = GetStrucId(i);

    StructName = GetStrucName(StructId);

    while(1) {
      pos = strstr(StructName, "::");

      if (pos > -1) {
        StructName = substr(StructName, 0, pos) + "__" + substr(StructName, pos + 2, -1);
        Message("StructName = '%s'\n", StructName);
        SetStrucName(StructId, StructName);
      } else
          break;
    }

    if(substr(StructName, 0, 2) == "s_")
        continue;

    NewName = "s_" + StructName;

    SetStrucName(StructId, NewName);

    for(FieldOfs = 0 ; FieldOfs != BADADDR ; FieldOfs = GetStrucNextOff(StructId, FieldOfs)) {
      FieldName = GetMemberName(StructId, FieldOfs);

      if(substr(FieldName, 0, 2) == "f_")
          continue;

      if(substr(FieldName, 0, 1) == "_")
          SetMemberName(StructId, FieldOfs, "f" + FieldName);
      else
          SetMemberName(StructId, FieldOfs, "f_" + FieldName);
    }
  }

  Message("%d structurs are done!\n", StructQty);
}
```

Скрипт globals.idc

```
#include <idc.idc>

static main() {
```

```
auto TxtFile, pFile, EA, EndEA, OldName, NewName, pos, OldNameLen;

EA = MinEA();

TxtFile = AskFile(1, "*.txt", "Choose a txt file for output...");
pFile = fopen(TxtFile, "w");

if(!pFile) {
  Message("Can't create file '%s'\n", TxtFile);
  return;
}

Message("MinEA() = 0x%8X\n", EA);
fprintf(pFile, "MinEA() = 0x%8X\n", EA);

while(EA != BADADDR) {
  OldName = Name(EA);

  if(OldName == "") {
    EA = NextAddr(EA);
    continue;
  }

  //*** Поиск имен функций

  //*** Изменяем функции импорта
  if(substr(OldName, 0, 6) == "__imp_") {
    NewName = "_imp_" + substr(OldName, 6, -1);

    MakeName(EA, NewName);

    Message("0x%8X : %s => %s\n", EA, OldName, NewName);
    fprintf(pFile, "0x%8X : %s => %s\n", EA, OldName, NewName);

    EA = NextAddr(EA);
    continue;
  }

  //*** Все остальные функции импорта пропускаем
  if(SegName(EA) == ".idata") {
    EA = NextAddr(EA);
    continue;
  }

  //*** В принципе, это условие не обязательно, если встречается только в секции .idata
  if(substr(OldName, 0, 5) == "_imp_") {
    EA = NextAddr(EA);
    continue;
  }

  OldNameLen = strlen(OldName);
  pos = strstr(substr(OldName, OldNameLen - 4, -1), "@");

  //*** Деманглируем локальные функции
  if(pos > 0) {
    if (substr(OldName, 0, 1) == "_") {
      NewName = substr(OldName, 1, OldNameLen - 4 + pos);

      MakeName(EA, NewName);

      Message("0x%8X : %s => %s\n", EA, OldName, NewName);
      fprintf(pFile, "0x%8X : %s => %s\n", EA, OldName, NewName);
```

```
      EA = NextAddr(EA);
      continue;
    }
  }

  //*** Поиск глобальных переменных

  //*** Пропускаем особые имена

  pos = strstr(OldName, "_");

  if((pos > -1) && (pos < 9)) {
    EA = NextAddr(EA);
    continue;
  }

  if(substr(OldName, 0, 1) == "?") {
    EA = NextAddr(EA);
    continue;
  }

  if(substr(OldName, 0, 1) == "@") {
    EA = NextAddr(EA);
    continue;
  }

  //*** Здесь нужно пропустить (деманглированные) имена функций

  if(OldName == GetFunctionName(EA)) {
    Message("0x%8X : Must be old function '%s'! \n", EA, OldName);
    fprintf(pFile, "0x%8X : Must be old function '%s'! \n", EA, OldName);

    EA = NextAddr(EA);
    continue;
  }

  //*** Пропускаем уже измененные имена

  if(substr(OldName, 0, 2) == "g_") {
    EA = NextAddr(EA);
    continue;
  }

  //*** Изменяем обычные имена

  NewName = "g_" + OldName;

  MakeName(EA, NewName);

  Message("0x%8X : %s => %s\n", EA, OldName, NewName);
  fprintf(pFile, "0x%8X : %s => %s\n", EA, OldName, NewName);

  EA = NextAddr(EA);
 }  // while(EA != BADADDR)

 fclose(pFile);

 Message("All done!\n");
 fprintf(pFile, "All done!\n");
}
```

Скрипт locals.idc

```
#include <idc.idc>

static main() {
 auto EA, FrameId, LastOffs, Offs, VarName, OldVarName, TxtFile, pFile, FuncName, NewName;

 TxtFile = AskFile(1, "*.txt", "Choose a txt file for output...");
 pFile = fopen(TxtFile, "w");

 if(!pFile) {
  Message("Can't create file '%s'\n", TxtFile);
  return;
 }

 for(EA = NextFunction(0); EA != BADADDR; EA = NextFunction(EA)) {
  FuncName = GetFunctionName(EA);

  Message("%8X: Function = '%s'\n", EA, FuncName);
  fprintf(pFile, "%8X: Function = '%s'\n", EA, FuncName);

  FrameId = GetFrame(EA);
  LastOffs = GetLastMember(FrameId);

  OldVarName = "";

  for(Offs = GetFirstMember(FrameId); Offs <= LastOffs; Offs = Offs + 4) {
   VarName = GetMemberName(FrameId, Offs);

   if(VarName == OldVarName)
     continue;

   OldVarName = VarName;

   if(VarName == "" || VarName == " r" || VarName == " s")
     continue;

   //*** Пропускаем локальные имена с уже имеющимся непустым префиксом
   if(strstr(VarName, "_") > 0)
     continue;

   //*** Создаем новое имя
   NewName = "l_" + VarName;

   //*** Применяем его вместо старого
   SetMemberName(FrameId, Offs, NewName);

   //Message("  VarName = '%s'\n", VarName);
   fprintf(pFile, "  VarName = '%s' => '%s', MemberSize = %d\n", VarName, NewName, GetMemberSize(FrameId, Offs));
  }
 }
}
```

Использование скриптов

Вооружившись этими несложными скриптами, мы уже можем существенно облегчить себе работу.

Начнем как обычно, загрузим **explorer.exe** в **demo версию IdaPro 6.1** и подгрузим из Интернета или локально отладочные символы. Чтобы иметь возможность сохранять полученный листинг, мы воспользуемся нашим плагином **SaveListing.plw** из прошлой статьи. Затем «распахиваем» свернутый

код на вкладках **Перечисления**, **Структуры** и **Листинг** «Иды». Смотрим, чтобы на вкладке с кодом не было сообщений типа: «**PRESS KEYPAD "+" TO EXPAND**».

Заметим, что появилась новая для нас секция **HEADER**, **PE**-заголовка данного **exe**-шника, который конечно можно свернуть, но в теле листинга есть ссылки на метки этого заголовка. Ну да ладно, пока оставим все как есть. Если все у нас заработает, то и хорошо, а если нет, тогда и будем ломать голову, что делать с данной **PE**-секцией.

Поскольку мы не имеем возможности сохранять в демо-версии наши изменения в базу **IdaPro**, то все наши «телодвижения» мы должны аккуратно осуществить за один «присест».

Начнем, как и договаривались, с исследования областей данных практически по всему листингу. Для этого будем просто тупо вручную листать сформированный «Идой» код на предмет подозрительного представления в нем данных. Прежде всего, обращаем внимание на начало и конец секции **.text** и на области где определенно много данных. Вот образец найденной «шероховатости»:

.text:01029D24 _c_wzTaskbarVertTheme – похоже на «левый» код между переменными юникода. Выделяем этот «код» и преобразовываем его в юникод командой меню **Edit / Strings / Unicode** или соответствующей пиктограммкой. Получаем по тому же адресу вместо «кода» вполне нормальное выражение **unicode 0, <TaskbarVert>, 0** . Это уже известный нам макрос, которого нет в «Иде», но который легко написать самостоятельно.

Другой пример более изощренный. По адресу **0x01003B94** имеем

```
.text:01003B94
.text:01003B94 loc_1003B94:    ; DATA XREF: CTrayBandSite::QueryInterface(_GUID const &,void * *)+24o
.text:01003B94             ; IUnknownToCTrayBandSite(IUnknown *)+Fo
.text:01003B94   push es
.text:01003B95   icebp
.text:01003B96   mov bl, 69h
.text:01003B98   add al, 0Fh
.text:01003B9A   rcl dword ptr [ecx], cl
.text:01003B9C   scasb
.text:01003B9D   db 2Eh
.text:01003B9D   add al, al
.text:01003BA0   dec edi
.text:01003BA1   mov gs, dx
.text:01003BA3   cdq
.text:01003BA4   nop
.text:01003BA5   nop
.text:01003BA6   nop
.text:01003BA7   nop
.text:01003BA8   nop
```

Это уже на код никак не «смахивает», хотя он и лежит между подпрограммами. И действительно, если мы это место «проморгаем», то компилятор выругается на команду **icebp**. Хорошо, хоть компилятор заметил, а не то это могло бы быть ошибкой времени выполнения. Ладно, применим на это дело наш скрипт **data.idc**. Получаем

```
.text:01003B94 dword_1003B94   dd 69B3F106h          ; DATA XREF: CTrayBandSite::QueryInterface(_GUID const &,void * *)+24o
.text:01003B94          ; IUnknownToCTrayBandSite(IUnknown *)+Fo
.text:01003B98   dd 11D30F04h
.text:01003B9C   dd 0C0002EAEh
.text:01003BA0   dd 99EA8E4Fh
.text:01003BA4   dd 90909090h
.text:01003BA8   dd 90h ; P
```

Это уже лучше, только не смотрятся пять подряд идущих **nop**'ов. Сделаем-ка из них массив байтов. Для этого **dword** из **4-х nop**'ов превратим в байты (клавиша «**d**» на клавиатуре), а затем все **5** байтов

выделим и применим команду «*» (на правой цифровой клавиатуре), получим искомую строку

```
.text:01003BA4   db 5 dup(90h)
```

Других особых мотивов, кроме эстетических, для смены типа данных, мы пока не обнаружили. И особой необходимости в применении скрипта **data.idc** тоже. Но вот зато для компактификации данных он нам будет крайне необходим.

Далее все будет относительно просто. Идем на вкладку перечислений **Enums** и применяем наш скрипт **enums.idc** (через **Alt-F7**). Видим, что все имена перечислений изменились, как мы и предполагали. Аналогично, на вкладке структур **Structures** меняем имена структур и их полей с помощью скрипта **structs.idc**. Ну и, наконец, на вкладке с листингом кода запускаем по очереди скрипты **globals.idc** и **locals.idc**. Эти скрипты запросят имена текстовых файлов для сохранения результатов своей работы, вводим произвольные имена, после чего можно будет посмотреть эти файлы независимо.

Теперь можно применить наш плагин из прошлой статьи для сохранения полученного листига. Сохраняем (у нас для этого назначена комбинация **Ctrl**+**[**) все в файл **explorer.lst**, который мы уже можем непосредственно преобразовать в файл ассемблера (**explorer.asm**). Не забудем вручную, через буфер обмена, добавить в начало файла листинга содержимое вкладки перечислений. Там всего порядка **8 Кб** текста, поэтому такие небольшие объемы данных «Ида» демо нам позволяет переносить. Можно было бы подправить наш плагин для перечислений, но пока лень, так как перечисления встречаются редко, да и небольших объемов, для копирования которых вполне хватает даже ограниченных возможностей «Иды». Если хотите, отредактируйте этот плагин самостоятельно.

Глюк «Иды»

При работе с «Идой» внимательно следите за окном сообщений. Вы не должны ни в коем случае получать сообщения вроде:

```
.text:010035E6: Can't find references (hint: redo analisys)
```

(адрес может быть любой). Эти сообщения достаточно непредсказуемы, их может быть много и они «гамачат» код листинга, который уже будет безнадежно компилировать. Если вы все же столкнулись с этой «радостью», закройте «Иду» и начните все делать заново, рано или поздно удача вам улыбнется, и вы сохраните нужный листинг без искажений «Иды». Лично мне пришлось делать несколько попыток, чтобы получить нормальный листинг кода.

Получение asm файла

Итак, мы имеем файл листинга **explorer.lst** из которого нам надо получить ассемблерный код в файле **explorer.asm**. До сих пор для этого мы использовали внешний скрипт **Visual FoxPro lst2asm.prg**. К этому скрипту мы еще вернемся, но сейчас у нас появилась новая проблема, которую самое время решить, ибо позже делать это будет более затруднительно.

Новая группа ошибок, которую мы уже упоминали, связана с упоминавшимися уже **чанками** или **чунками функций**. Это попросту означает, что все эти куски функций, которые можно грубо обозвать «не пришей к одному месту рукав», нужно вернуть на их исконное место, т.е. внутрь тела их родительских функций. Например, чунк

```
.text:0101538F ; =============== S U B R O U T I N E =======================================
.text:0101538F
.text:0101538F ; Attributes: bp-based frame
.text:0101538F
.text:0101538F ; int __stdcall CTray___UpdateVertical(HDC hDC, int)
.text:0101538F ?_UpdateVertical@CTray@@IAEXIH@Z proc near
.text:0101538F             ; CODE XREF: CTray::v_WndProc(HWND__ *,uint,uint,long)+1739p
.text:0101538F             ; CTray::_InitBandsite(void)+1Dp
```

```
.text:0101538F
.text:0101538F l_hDC          = dword ptr  8
.text:0101538F arg_4          = dword ptr  0Ch
.text:0101538F
.text:0101538F ; FUNCTION CHUNK AT .text:01024CD6 SIZE 0000000A BYTES
.text:0101538F
.text:0101538F   mov     edi, edi
```

о котором сообщается в начале функции, нужно разместить в конце этой функции

```
.text:01015450   retn    8
.text:01015450 ?_UpdateVertical@CTray@@IAEXIH@Z endp
```

а именно, мы должны получить что-то вроде

```
.text:01015450   retn    8
.text:01024CD6 ; ---------------------------------------------------------------------------
.text:01024CD6 ; START OF FUNCTION CHUNK FOR ?_UpdateVertical@CTray@@IAEXIH@Z
.text:01024CD6
.text:01024CD6 loc_1024CD6:             ; CODE XREF: CTray::_UpdateVertical(uint,int)+48j
.text:01024CD6   mov     eax, offset _c_wzTaskbarVertTheme ; "TaskbarVert"
.text:01024CDB   jmp     loc_10153DD
.text:01024CDB ; END OF FUNCTION CHUNK FOR ?_UpdateVertical@CTray@@IAEXIH@Z
.text:01024CDB ; ---------------------------------------------------------------------------
.text:01015450 ?_UpdateVertical@CTray@@IAEXIH@Z endp
```

Естественно, проделывать вручную подобные телодвижения не очень «прикалывает», так как их до безобразия очень много. Нужно писать еще один, уже внешний скрипт, на нашем любимом **Visual FoxPro** ☺. И такой скрипт, **nochunks.prg**, мы написали. Только применять его надо до выполнения скрипта **lst2asm.prg**.

Скрипт nochunks.prg

```
*****************************************************************************
CLEAR
SET TALK OFF
SET SAFETY OFF
SET DATE german
SET CENTURY ON

outname = "explorer"

SET TEXTMERGE ON
SET TEXTMERGE TO (outname + "~.lst")

* .text:01001988

IF NOT FILE("lst.dbf")
  CREATE TABLE lst.dbf (txt c(6), addr c(8), line1 c(254), line2 c(254))

  SELECT lst
  APPEND FROM (outname + ".lst") TYPE SDF
ENDIF

IF NOT FILE("idxlst.dbf")
  CLOSE DATABASES

  COPY FILE lst.dbf TO idxlst.dbf
ENDIF
```

```
SELECT 0

IF FILE("idxlst.cdx")
  USE idxlst INDEX idxlst
ELSE
  USE idxlst
  INDEX ON addr TAG iaddr
ENDIF

SET ORDER TO iaddr

SELECT 0
USE lst

MAXCHUNKLINE = 1000  && Максимальное количество строк в чунке

MAXCHUNK = 250  && Максимальное количество чунков в функции
DIMENSION ChunkAddr[MAXCHUNK]

&& Это можно и не делать
FOR i = 1 TO MAXCHUNK
  ChunkAddr[i] = ""
ENDFOR

j = 0
FuncName1 = ""
FuncName2 = ""

ChunkText = "; FUNCTION CHUNK AT .text:"
ChunkTextLen = LEN(ChunkText)

ChunkLine = "; ---------------------------------------------------------------------------"
ChunkStart = "; START OF FUNCTION CHUNK FOR"
ChunkStartLen = LEN(ChunkStart)
ChunkEnd = "; END OF FUNCTION CHUNK FOR"
ChunkEndLen = LEN(ChunkEnd)

IsChunk = .F.

FOR i = 1 TO RECCOUNT()
  GO i
  SCATTER MEMVAR

*!*    && Проверяем нужна ли нам line2?
*!*    IF NOT EMPTY(m.line2)
*!*      \i = <<i>> : '<<TRIM(m.line2)>>'
*!*    ENDIF
*!*    LOOP

  pos = AT(" proc ", m.line1)  && Нашли начало функции

  IF pos > 0
    FuncName1 = LTRIM(LEFT(m.line1, pos - 1))

*    \i = <<i>> : FuncName1 = '<<FuncName1>>'

    FuncName2 = ""
  ENDIF

  pos = AT("endp", m.line1)  && Нашли конец функции

  IF pos > 0
```

```
   FuncName2 = LTRIM(LEFT(m.line1, pos - 2))

*    \i = <<i>> : FuncName2 = '<<FuncName2>>'

   FuncName1 = ""
  ENDIF

  if(EMPTY(FuncName1) AND EMPTY(FuncName2))  && Находимся вне тела некоторой функции
   IF LEFT(LTRIM(m.line1), ChunkStartLen) == ChunkStart
    IsChunk = .T.
    LOOP
   ENDIF

   IF LEFT(LTRIM(m.line1), ChunkEndLen) == ChunkEnd
    IsChunk = .F.
    LOOP
   ENDIF

   IF IsChunk
    LOOP
   ENDIF

   IF ALLTRIM(m.line1) <> ChunkLine
    \<<m.txt>><<m.addr>><<TRIM(m.line1 + m.line2)>>
   ENDIF
  ENDIF

*!*  .text:01001991 ; =============== S U B R O U T I N E =======================================
*!*  .text:01001991
*!*  .text:01001991 ; Attributes: bp-based frame
*!*  .text:01001991
*!*  .text:01001991 ; protected: void __thiscall CTray::_MessageLoop(void)
*!*  .text:01001991 ?_MessageLoop@CTray@@IAEXXZ proc near   ; CODE XREF: CTray::MainThreadProc(void *)+24_p
*!*  .text:01001991
*!*  .text:01001991 l_Msg          = s_tagMSG ptr -1Ch
*!*  .text:01001991
*!*  .text:01001991 ; FUNCTION CHUNK AT .text:01001A5F SIZE 0000007D BYTES
*!*  .text:01001991 ; FUNCTION CHUNK AT .text:01021E8C SIZE 00000024 BYTES
*!*  .text:01001991 ; FUNCTION CHUNK AT .text:010221C5 SIZE 0000002A BYTES
*!*  .text:01001991 ; FUNCTION CHUNK AT .text:01025521 SIZE 00000032 BYTES
*!*  .text:01001991
*!*  .text:01001991   mov     edi, edi
*!*  .text:01001993   push    ebp
*!*  .text:01001994   mov     ebp, esp
*!*  .text:01001996   sub     esp, 1Ch
*!*  .text:01001999   push    ebx
*!*  .text:0100199A   mov     ebx, ds:_imp__SendMessageW@16 ; imp__SendMessageW(x,x,x,x)

  IF NOT EMPTY(FuncName1) AND EMPTY(FuncName2)  && Находимся в теле некоторой функции
   && Ищем чунки
   pos = AT(ChunkText, m.line1)  && Нашли чунк

   IF pos > 0
    j = j + 1

    IF j > MAXCHUNK  && Переполнение массива чунков
     \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>
     \j > MAXCHUNK = <<MAXCHUNK>>

     MESSAGEBOX("MAXCHUNK overflow!")
     EXIT
    ENDIF
```

```
    ChunkAddr[j] = SUBSTR(m.line1, pos + ChunkTextLen, 8)

*     \i = <<i>> : ChunkAddr = '<<ChunkAddr>>'
   ENDIF

   \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>
  ENDIF

  IF EMPTY(FuncName1) AND NOT EMPTY(FuncName2)  && Находимся в конце некоторой функции
   && Пишем чунки

   SELECT idxlst

   DO ChunksWrite

   SELECT lst

   GO i
   SCATTER MEMVAR

   \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>

   j = 0
   FuncName1 = ""
   FuncName2 = ""
  ENDIF

  IF NOT EMPTY(FuncName1) AND NOT EMPTY(FuncName2)  && Должна быть недопустимой ситуацией
   \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>
   \i = <<i>> : FuncName1 = '<<FuncName1>>'; FuncName2 = '<<FuncName2>>'

   MESSAGEBOX("Function error!")
   EXIT
  ENDIF
ENDFOR

CLOSE ALL
QUIT
*******************************************************************************
*******************************************************************************
PROCEDURE ChunksWrite
  FOR k = 1 TO j
   SEEK ChunkAddr[k]

*!*  .text:01001A5F ; ---------------------------------------------------------------------------
*!*  .text:01001A5F ; START OF FUNCTION CHUNK FOR ?_MessageLoop@CTray@@IAEXXZ
*!*  .text:01001A5F
*!*  .text:01001A5F loc_1001A5F:             ; CODE XREF: CTray::_MessageLoop(void)+26_j
*!*  .text:01001A5F   cmp     [ebp+l_Msg.f_message], 12h
*!*  .text:01001A63   jz      loc_1025532
*!*  .text:01001A69   lea     eax, [esi+4C0h]
*!*  ...
*!*  .text:01001ACD   lea     eax, [ebp+l_Msg]
*!*  .text:01001AD0   push    eax             ; lpMsg
*!*  .text:01001AD1   call    ds:_imp__DispatchMessageW@4 ; imp__DispatchMessageW(x)
*!*  .text:01001AD7   jmp     loc_10019A6
*!*  .text:01001AD7 ; END OF FUNCTION CHUNK FOR ?_MessageLoop@CTray@@IAEXXZ
*!*  .text:01001AD7 ; ---------------------------------------------------------------------------

   IF FOUND()
    RNo = RECNO()
```

```
    SCATTER MEMVAR

*     IF ALLTRIM(m.line1) <> ChunkLine
    \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>
*     ENDIF

    I = 0

    DO WHILE .T.
     I = I + 1
     RNo = RNo + 1

     GO RNo
     SCATTER MEMVAR

*      IF ALLTRIM(m.line1) <> ChunkLine
     \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>
*      ENDIF

     IF LEFT(LTRIM(m.line1), ChunkEndLen) == ChunkEnd OR I > MAXCHUNKLINE
      EXIT
     ENDIF
    ENDDO

    IF I > MAXCHUNKLINE
     \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>
     \I > MAXCHUNKLINE = <<MAXCHUNKLINE>>

     MESSAGEBOX("MAXCHUNKLINE overflow!")

     CLOSE ALL
     QUIT
    ENDIF

    \<<m.txt>><<m.addr>> <<ChunkLine>>
   ELSE  && NOT FOUND()
    \<<m.txt>><<m.addr>><<TRIM (m.line1 + m.line2)>>
    \Not found ChunkAddr[<<k>>] = <<ChunkAddr[k]>>

    MESSAGEBOX("Not found ChunkAddr!")

    CLOSE ALL
    QUIT
   ENDIF
  ENDFOR
ENDPROC
********************************************************************************
********************************************************************************
```

Хочется сказать пару слов насчет этого скриптпа. Несмотря на его небольшой объем, он делает очень большую и серьезную работу. А именно создает два файла базы данных **VFP**, причем второй **dbf**-файл индексируется для быстрого поиска в нем адресов чунков. При сканировании первого **dbf**-файла (который получается в два хлопка за счет очень мощных команд **VFP**) извлекается вся информация в выходной поток, кроме внешних чунков, до тех пор, пока нет нужды делать вставку внутреннего чунка в конце тел соответствующих функций. До этого идет сбор адресов чунков, если они есть, из начала данной функции. После того, как определяется место вставки для внутренних чунков, идет извлечение этих самых чунков из второго индексированного **dbf**-файла и запись их в выходной поток. Эти **dbf**-файлы состоят из нескольких полей, два последних из которых имеют максимальную для **VFP** длину – **254** байта. Мы выбрали два таких поля вместо одного потому, что некоторые строки листинга могут иметь очень большие длины, порядка **300** и более символов. Т.е. одного поля мало, а два вполне

достаточно, хотя для листингов со сверхдлинными строками можно взять и три таких поля. Несмотря на то, что текстовый файл может иметь строки произвольной длины, он экспортируется в **dbf**-файл, фиксированной ширины, фактически одной командой, с правильной разбивкой на нужные поля. Это очень сильно упрощает работу с подобными текстовыми файлами в формате **dbf**. Добавим еще, что обработка других, кроме **.text**, секций может быть бесполезной, особенно если отличаются длины наименований этих секций, но там чунки мы и не ищем, а выходной поток сформируется, тем не менее, аккуратно.

Интересно, насколько усложнится подобный скрипт, если отказаться от идеи использования движка базы данных ☺?

Этот скрипт выполняется довольно долго – несколько минут и генерит весьма объемные дополнительные файлы, которые нужно затем удалять. Но поскольку процедура это разовая, то думаю, несколько минут можно и перекурить ☺.

Скрипт lst2asm.prg

Опыт работы с большими ассемблерными файлами, сгенерированные «Идой», показывает, что очень удобно, когда в **asm**-файле присутствуют адреса строк соответствующего листинга. Поэтому мы слегка подправим наш файл **lst2asm.prg** (внешний скрипт **Visual FoxPro**), чтобы он сохранял эти адреса в виде комментариев, типа:

```
; =============== S U B R O U T I N E =======================================  ; .text:01001F05
; public: virtual unsigned long __stdcall CDeskTray::AddRef(void)  ; .text:01001F05
?AddRef@CDeskTray@@UAGKXZ proc near     ; DATA XREF: .text:01001EF0↓o  ; .text:01001F05
  push    2  ; .text:01001F05
  pop     eax  ; .text:01001F07
  retn    4  ; .text:01001F08
?AddRef@CDeskTray@@UAGKXZ endp  ; .text:01001F08
; ---------------------------------------------------------------------------  ; .text:01001F08
  align 10h  ; .text:01001F0B
```

Наверное, красивее было бы, если бы наш **asm**-файл был вида:

```
_(.text:01001F05) ; =============== S U B R O U T I N E =======================================
_(.text:01001F05) ; public: virtual unsigned long __stdcall CDeskTray::AddRef(void)
_(.text:01001F05) ?AddRef@CDeskTray@@UAGKXZ proc near     ; DATA XREF: .text:01001EF0↓o
_(.text:01001F05)            ; .text:0101ABE8↓o ...
_(.text:01001F05)   push    2
_(.text:01001F07)   pop     eax
_(.text:01001F08)   retn    4
_(.text:01001F08) ?AddRef@CDeskTray@@UAGKXZ endp
_(.text:01001F08) ; ---------------------------------------------------------------------------
_(.text:01001F0B)   align 10h
```

где макрофункция **_()** определена в виде:

```
_ MACRO txt:VARARG
   EXITM <>
ENDM
```

т.е. возвращает пустое значение при любом своем аргументе. Однако подобная реализация для многомегабайтных «простынь» кода имеет свою оборотную сторону. Компилятор на таком коде начинает непредсказуемо глючить, выдает ошибки случайным образом даже там, где их не может быть по определению. Поэтому мы не будем испытывать терпение компилятора и просто воспользуемся первым вариантом. Хотя для малых объемом кода можно применять и второй вариант, учитывая, однако, что стек под макросы у компилятора не безграничный.

Код скрипта **lst2asm.prg** для **Visual FoxPro** мы здесь не приводим, так как он похож на предыдущий, и вы можете посмотреть его, среди прилагаемых к этой статье файлов.

Итак, получив ассемблерный файл **explorer.asm**, мы займемся его предварительным «причесыванием». Во-первых, добавим строки

```
.686p
.mmx
.model flat

include headers.inc
```

в начало **asm**-файла и удалим аналогичные строки (в т.ч. «**include uni.inc»**) далее по тексту. Во-вторых, в файл **headers.inc** мы добавим ссылки на все необходимые для данного проекта библиотеки:

```
includelib lib\kernel32.lib
includelib Lib\user32.lib
includelib Lib\shell32.lib
includelib Lib\gdi32.lib
includelib Lib\advapi32.lib
includelib Lib\shlwapi.lib
includelib Lib\msvcrt.lib
includelib Lib\ntdll.lib
includelib Lib\imm32.lib
includelib Lib\oleacc.lib
includelib Lib\ole32.lib
includelib Lib\oleaut32.lib
includelib Lib\UxTheme.lib
includelib Lib\winmm.lib
includelib Lib\browseui.lib
includelib Lib\shdocvw.lib
includelib Lib\setupapi.lib
includelib Lib\winsta.lib
includelib Lib\userenv.lib
```

Некоторые из этих файлов у нас пока еще отсутствуют. Поэтому нам надо буде еще сгенерировать для них **def** и **lib** файлы по технологии, описанной в третьей статье.

Компиляция файла explorer.asm

Ну вот, теперь мы можем начать предварительную компиляцию нашего проекта с целью поиска и исправления имеющихся в нем ошибок. Применим для этого командный файл **asm_exe.bat**, аналогичный ранее использованному **asm.bat**, с небольшими изменениями. Компилируем и наблюдаем уже привычные нам ошибки.

Первая группа ошибок касается проблем инициализации структур. Например, если поле имеет структурный тип, то вместо инициализации вида:

```
s_ITEMIDLIST    struc ; (sizeof=0x3, standard type)  ; 00000000
f_mkid          s_SHITEMID ?  ; 00000000
s_ITEMIDLIST    ends  ; 00000003
```

должна быть запись типа:

```
s_ITEMIDLIST    struc ; (sizeof=0x3, standard type)  ; 00000000
f_mkid          s_SHITEMID <?>  ; 00000000
s_ITEMIDLIST    ends  ; 00000003
```

Можно, в принципе, все подобные знаки вопроса в структурах заменить на те же знаки в угловых скобках, поскольку для простых типов такая запись допустима. Но мы ограничимся только исправлением инициализации сложных типов полей структур. Кстати, для объединений (**union**) и некоторых видов структур компилятор не принимает инициализацию ни в виде **?** ни **<?>**, но «согласен» на пустые скобки **<>** .

Затем компилятор даст нам понять, что порядок структур ему не нравиться. Поскольку он однопроходной, то определения структур, встречающиеся далее по тексту, ему пока еще неизвестны, вот он и ругается по этому поводу. Нам нужно все определения таких структур вынести наверх, до их первого использования. В данном случае, это касается таких структур как **s_RECT** и **s_POINT**.

Вторая группа ошибок связано с выравниванием. Выравнивание типа **align 1000h** компилятор не принимает, только значения не более **10h**.

Третья группа ошибок связанна с недопустимыми символами в строках юникода, такими как процент или кавычка. В этом случае, просто делаем замену знака **%** на знаки **!%** и **">** на **!">** .

Четвертая группа ошибок связана с недопустимым переносом строк. Например, «Ида» может воспользоваться **си**-шной нотацией переноса строк:

```
stru_103B990    s__SCOPETABLE_ENTRY <0FFFFFFFFh, offset loc_103B96F, \  ; .text:0103B990
            ; DATA XREF: _IsDirectXExclusiveMode(void)+2↑o  ; .text:0103B990
        offset loc_103B978> ; SEH scope table for function 103B8FF  ; .text:0103B990
```

тогда как компилятору ассемблера такой вариант не подходит. Поэтому, просто пишем в одну строчку:

```
stru_103B990    s__SCOPETABLE_ENTRY <0FFFFFFFFh, offset loc_103B96F, offset loc_103B978>
            ; SEH scope table for function 103B8FF  ; .text:0103B990
            ; DATA XREF: _IsDirectXExclusiveMode(void)+2↑o  ; .text:0103B990
```

Может также встретиться ситуация, когда одна достаточно длинная строка разрезана на две части, даже без символа переноса строки. Такие строки мы просто объединяем и все.

Пятая группа ошибок связана с избыточной для компилятора инструкцией «**large** ». Заменяя ее на пустую строку, избавляем компилятор от лишних забот. Правда, делайте подобную замену аккуратно, а то ненароком измените правильное имя «**l_phiconLarge** » на неправильное «**l_phicon**».

Шестая группа ошибок связана с директивой «**rva**». Ранее мы эти области данных, связанные с импортом функций, попросту удаляли, но сейчас такой номер не прокатит, так как, по-видимому, это связано с отложенным импортом и наличием, в силу этого, ссылок на эту область данных. Поэтому мы поступим проще, сделаем замену «**rva** » на «**offset** ».

Седьмая группа ошибок связана с неспособностью компилятора понять некоторые инструкции «Иды», например,

```
jmp     far ptr 59Eh:11CF7F92h  ; .text:0103154F
```

Такие «прибамбасы» не нравятся даже «Иде», и эту инструкцию надо взять под особый контроль, но пока для целей компиляции мы воспользуемся универсальным выходом из подобных ситуаций. Вместо кода запишем его оригинальные данные в шестнадцатеричном формате, которые можно посмотреть на вкладке **Hex View** «Иды» (выделив всю инструкцию целиком). Берем оттуда предлагаемую последовательность байтов и пишем:

```
db 0EAh, 92h, 7Fh, 0CFh, 11h, 9Eh, 05h  ; jmp     far ptr 59Eh:11CF7F92h  ; .text:0103154F
```

На самом деле, тут проблема может быть глубже, возможно, весь объемлющий эту строчку локальный код (в диапазоне адресов **.text:0103154C – .text:0103154F**):

```
 retn    4  ; .text:01031549
; ---------------------------------------------------------------------------  ; .text:0103154C

loc_103154C:             ; DATA XREF: CTray::_LoadInProc(tagCOPYDATASTRUCT *)+15o  ; .text:0103154C
 add     [edi+48h], esp  ; .text:0103154C
 db 0EAh, 92h, 7Fh, 0CFh, 11h, 9Eh, 05h  ; jmp     far ptr 59Eh:11CF7F92h  ; .text:0103154F
?_LoadInProc@CTray@@IAEJPAUtagCOPYDATASTRUCT@@@Z endp  ; .text:0103154F

g_Dest    db 'DEST',0  ; .text:01031556
 align 4  ; .text:0103155B
 db 5 dup(90h)  ; .text:0103155C
```

является данными, следовательно, в данные нужно превратить и вышележащую инструкцию (по адресу **.text:0103154C**). Также, не исключено, что нужно будет еще передвинуть определение конца функции вверх, срезу после инструкции **retn 4**. Однако думается, в данном случае сойдет все и так, но помнить об этом проблемном месте стоит.

Аналогично решаем проблему со следующей инструкцией:

```
db 69h, 0C0h, 18h, 0FCh, 0FFh, 0FFh  ; imul    eax, 0FFFFFC18h  ; .text:01006B53
```

Здесь инструкция вполне законна, но, по-видимому, компилятор ее просто не «знает».

Естественно, что компилятор воспринимает машинные инструкции как родные 🙂.

Восьмая группа ошибок связана уже с перемещенными, на свои места, чунками. Из-за этого некоторые короткие переходы становятся уже длинными. В таких случаях нужно просто удалить директиву «**short**».

После указанных исправлений, мы наконец-то скомпилировали объектный модуль **explorer.obj**. Таким образом, очередное препятствие позади 🙂. Теперь настал момент «обслуживать» линковщик.

Разрешение проблем линковки

Проблемы линковки можно разделить на две большие группы. Первая связана с разрешением проблем экспортируемых символов из имеющихся **def** и **lib**-файлов и, вторая, с отсутствием самих **def** и **lib**-файлов.

Сначала решим проблему отсутствующих библиотек, а затем уже ошибки экспорта имен функций.

Генерация новых lib-файлов

Для этого просто используем технологию описанной в нашей третьей статье. Добавим только, что необходимые **def**-файлы мы уже подготовили, включив их наряду с уже имеющимися у нас. Для разовой компиляции всех запрашиваемых **lib**-файлов мы используем командный файл **def2lib.cmd**. Некоторые ненужные в данном проекте **def**-файлы закомментированы. Аналогичный файл **def2lib.bat** служит для пересоздания какого-то одного конкретного **lib**-файла. Заметим также, что некоторые **def**-файлы требуют определения своих функций через ординалы. При наличии «правильно» сгенерированных **lib**-файлов мы не должны иметь проблем с определением этих функций в нашем проекте.

Ошибки экспорта имен функций

Линковщик выдал нам теперь следующие ошибки:

```
shell32.lib(shell32.dll) : error LNK2005: _CheckWinIniForAssocs@0 already defined in explorer.obj
shell32.lib(shell32.dll) : error LNK2005: _CheckDiskSpace@0 already defined in explorer.obj
```

```
shell32.lib(shell32.dll) : error LNK2005: _CheckStagingArea@0 already defined in explorer.obj
shlwapi.lib(shlwapi.dll) : error LNK2005: _StopWatchMode@0 already defined in explorer.obj
shlwapi.lib(shlwapi.dll) : error LNK2005: _GetPerfTime@0 already defined in explorer.obj
ole32.lib(ole32.dll) : error LNK2005: _CoUninitialize@0 already defined in explorer.obj
winmm.lib(winmm.dll) : error LNK2005: _waveOutGetNumDevs@0 already defined in explorer.obj
shdocvw.lib(shdocvw.dll) : error LNK2005: _RunInstallUninstallStubs@0 already defined in explorer.obj
```

На самом деле это означает, что эти имена в нашем ассемблерном файле продублированы, так как они уже существуют в соответствующих файлах библиотек. Все, что нам нужно сделать, это переименовать вручную неманглированные части этих имен так, чтобы они не совпадали с их библиотечными аналогами. Я, например, просто добавил символ подчеркивания в начало имени таких функций и изменил ссылки на них.

Исправив, наконец, эти последние ошибки мы, с разочарованием, обнаруживаем, что наш сгенерированный файл **explorer.exe** не запускается. И знаете почему? Помните, что выше мы писали о возможности игнорирования в файле ресурсов манифеста? Так вот без манифеста все работает! Ура товарищи ☺!!! Однако при ближайшем рассмотрении оказывается, что проблема манифеста, определенного в наших ресурсах как:

```
123 24
MOVEABLE PURE LOADONCALL DISCARDABLE
"explorer.xml"
```

заключается в идентификаторе **123**! Любой другой идентификатор **1**, **122** или **124** подходит (**exe**-шник запускается), а **123** нет. Ну и как это называется литературным словом ☺? Ошибка времени выполнения достаточно неочевидная и теоретически может быть непробиваемой. К счастью нам повезло, что мы решили сначала поэкспериментировать с ресурсами, а не искать ошибки в отладчике. Как бы там ни было, пишем единицу вместо **123** и любуемся нашим творением ☺ (рис. 1).

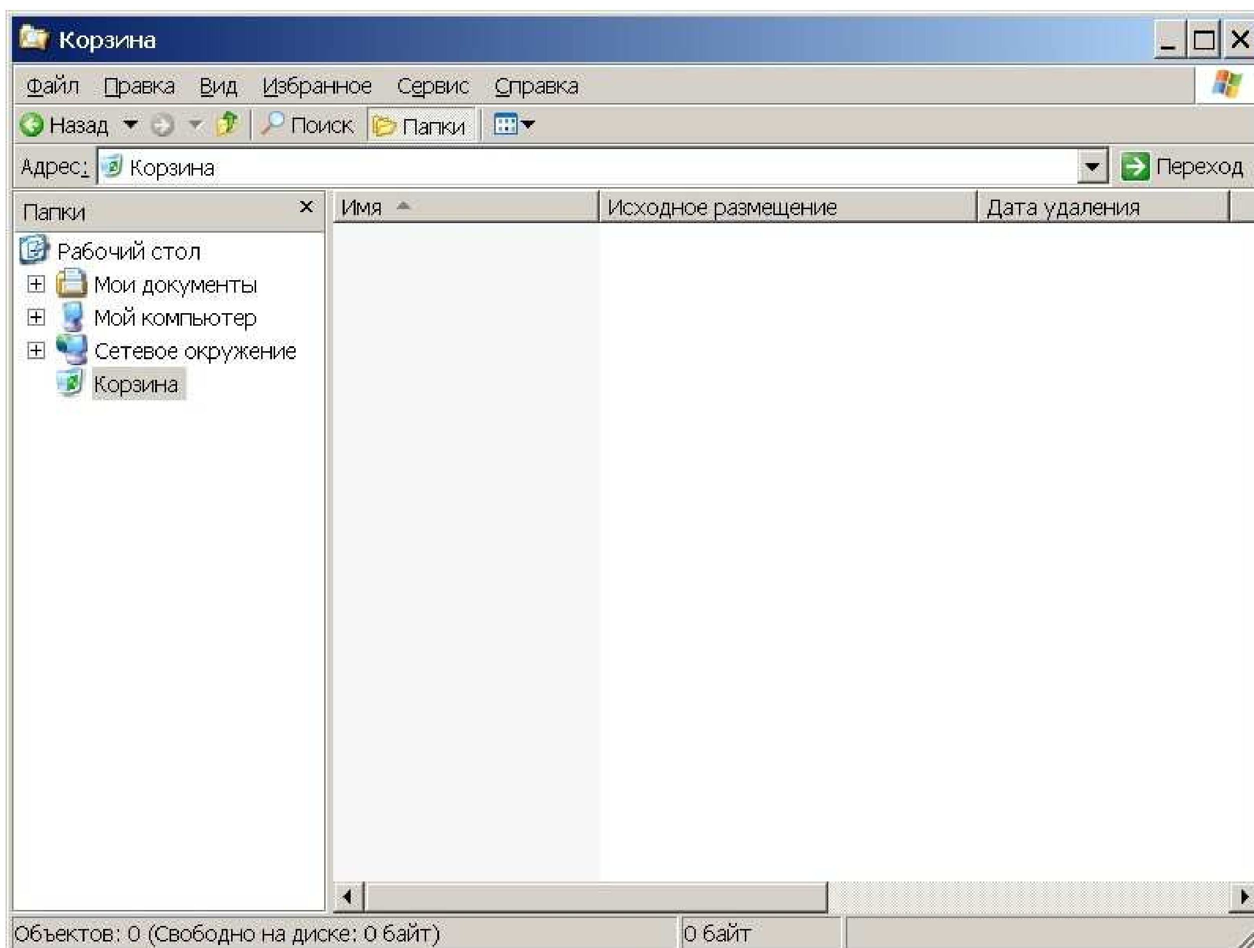

Рис. 1. Вид перекомпилированного файла **explorer.exe**.

Беглое тестирование показывает, что вроде бы перекомпилированный **Проводник** работает без проблем. Впрочем, ошибки времени выполнения не исключены и, если вы их обнаружите, то будет интересно узнать о них. Мы же считаем свою первую часть работы выполненной и можем теперь приступить ко второй ее части – перекомпиляции системной библиотеки **comctl32.dll** контролов общего назначения.

Часть 2. comctl32.dll

Ресурсы библиотеки comctl32.dll

Загрузим шестую версию **comctl32.dll,** под **Windows XP, sp.3,** в редактор ресурсов. Объем бинарного кода (без ресурсов) здесь примерно в два раза больше, чем для проводника **explorer.exe**, так что процесс перекомпиляции проще не будет. Исследуя ресурсы, мы видим, что имеется большее количество поддерживаемых языков интерфейса. Однако при перекомпиляции ресурсов мы удалим «лишние» языки интерфейса, так как мы работаем с текстовой (а не юникодовской) версией файла ресурсов и многие языки, типа арабского или китайского там будут отображаться неадекватно. А поскольку, на данном этапе, количество поддерживаемых языков интерфейса для нас несущественно, то мы оставим только английский и русский языки 🙂. Папку для ресурсов данной **dll**-ки мы выберем **res_dll**, чтобы отличать от аналогичной папки **exe**-шника. Дальнейшее редактирование ресурсного скрипта будет происходить как обычно. Таким образом, мы получаем достаточно компактный скомпилированный файл ресурсов **comctl32.res**, размером примерно **160 Кб**.

Подготовка ассемблерного кода

Начнем, как и раньше, с исследования областей данных по всему листингу. Для этого будем просто тупо вручную листать сформированный «Идой» код на предмет подозрительного представления в нем данных. Прежде всего, обращаем внимание на начало и конец секции **.text** и на области где определенно много данных. Вот образцы найденных «шероховатостей»:

.text:773C1A78÷773C1A96 – похоже на «левый» код между переменными юникода. Ставим курсор в начало этого «кода» (или, лучше, выделяем весь блок, так надежней) и жмем пимпочку юникода. Получаем определение соответствующей строки для глобальной переменой **g_Toolbarwindow32**.

.text:773C2220÷773C2238 – аналогично, для переменной **_c_szComboBoxEx**.

Далее мы ограничимся указанием только начала «кода» юникода и именем «правильной» переменной.

.text:773C2A18 – **_c_szSpace** (юникод);
.text:773C2A1C – **_c_szTabControlClass** (юникод);
.text:773C2A3C – **_c_szListViewClass** (юникод);
.text:773C2A58 – **_c_szHeaderClass** (юникод);
.text:773C2A70 – **_c_szTreeViewClass** (юникод);
.text:773C2AB4 – **_c_szSToolTipsClass** (юникод);
.text:773C2AF8 – **_c_szReBarClass** (юникод);
.text:773C2B14 – **_c_szEllipses** (юникод);
.text:773C2B1C – **_c_szShell** (юникод);
.text:773C2B28 – **_c_szEdit** (юникод);
.text:773C2B34 – **_c_szSelect** (юникод);
.text:773C2BD0 – **_c_szCC32Subclass** (юникод);
.text:773C2488 – **_mpStyleCbr** (двойные слова);

Имеется также много данных представленных в виде длинных последовательностей байтов. Их можно, хотя бы в целях компактификации, представить в виде **dword**’ов. Для этого можно применять наш скрипт **data.idc**. Но это уже оставим на ваше усмотрение.

Получение asm файла

Применяя наши скрипты, в той же последовательности, что и в первом случае, мы получаем искомый файл листинга **comctl32.lst**. Обращаем внимание на базу (**Imagebase**): **0x773C0000**. Ее надо учесть в командном файле **asm_dll.bat**.

Процесс получения ассемблерного файла из файла листинга происходит аналогично предыдущему случаю. Некоторым облегчением для нас служит отсутствие чунков в файле листинга, поэтому нам нет необходимости запускать скрипт **nochunks.prg**. Ограничимся только скриптом **lst2asm~.prg** (вместо скрипта **lst2asm.prg**, от которого он отличается всего парой строчек). Все остальные изменения по получению корректного **asm**-файла уже достаточно стандартны, так что мы вполне можем пропустить их описание.

Экспортирование функций

Мало получить искомую **dll**, нужно еще, чтобы она экспортировала все требуемые нам функции. Кстати, многие функции **comctl32.dll** экспортируются по ординалам. Нам же нет никакой необходимости следовать этим путем, тем более что все экспортируемые функции описаны в отладочных символах и потому могут быть представлены только своими именами. Отметим, что скрипт **globals.idc** уже проделал для нас серьезную работу по деманглированию экспортируемых функций, что сильно упростит нам составление файла экспортируемых функций **comctl32.def**. Для его составления необходимо только решить вопрос по переименованию некоторых функций, которые являются телом для нескольких экспортируемых функций и имеют весьма экзотический вид, непригодный для непосредственного экспорта. Например,

```
; Exported entry  32. FlatSB_GetScrollProp  ; .text:7196EF54
; Exported entry  80. ImageList_SetFilter  ; .text:7196EF54

; =============== S U B R O U T I N E =======================================  ; .text:7196EF54

; protected: virtual int __thiscall CControl::v_OnNCCalcSize(unsigned int, long, long *)  ; .text:7196EF54
;  public ?v_OnNCCalcSize@CControl@@MAEHIJPAJ@Z  ; .text:7196EF54
  public CControl__v_OnNCCalcSize
;?v_OnNCCalcSize@CControl@@MAEHIJPAJ@Z proc near  ; .text:7196EF54
CControl__v_OnNCCalcSize proc near  ; .text:7196EF54
            ; CODE XREF: ListView_GetCxScrollbar+14p  ; .text:7196EF54
            ; ListView_GetCyScrollbar+14p ...  ; .text:7196EF54
  xor     eax, eax        ; FlatSB_GetScrollProp  ; .text:7196EF54
  retn    0Ch  ; .text:7196EF56
CControl__v_OnNCCalcSize endp  ; .text:7196EF56
;?v_OnNCCalcSize@CControl@@MAEHIJPAJ@Z endp  ; .text:7196EF56
```

Здесь «родное» имя **?v_OnNCCalcSize@CControl@@MAEHIJPAJ@Z** заменено на имя **CControl__v_OnNCCalcSize**, чтобы иметь возможность в файле **comctl32.def** сделать запись вида:

```
CControl__v_OnNCCalcSize
FlatSB_GetScrollProp = CControl__v_OnNCCalcSize  ; @32
ImageList_SetFilter = CControl__v_OnNCCalcSize  ; @80
```

Естественно, что все ссылки на старое имя должны быть изменены. Всего таких исправлений у нас будет порядка десяти.

Однако если мы намереваемся в будущем тестировать нашу **dll**-ку вместо системной (путем ее подмены), то нам надо иметь немного другой **def**-файл. Для этого мы слегка модифицировали скрипт **lst2def.prg** в **lst2def~.prg** для файла листинга **comctl32.lst**, так чтобы в файле определений отображались **все ординалы**. Здесь мы пользуемся **нетривиальной информацией** о том, что другие системные файлы **Windows** часто обращаются к библиотеке **comctl32.dll** по ординалам даже тогда, когда ее

экспортируемые функции определены по именам. Видимо, это наследие старого прошлого, когда **Майкрософт** не так щедро делился именами экспортируемых функций 🙂. Таким образом, если мы желаем существенно уменьшить для себя возможные проблемы с вызовом экспортируемых функций по неопределенным ординалам, то тогда должны в нашем **def**-файле показать как все имена, так и все ординалы. Такой файл (**comctl32.def**) мы подготовили, с которым вы также можете ознакомиться. Заметим, что частично его пришлось «доводить до ума» вручную.

Проблема «входной точки» для перекомпилируемых dll

Следует еще сказать про проблему «входной точки» для перекомпилируемых **dll**. В нашем случае это будет функция **LibMain**. Во-первых, компилятору не «нравиться» ее определение (он «требует» фиксированного прототипа для «входной точки», а у нас она «безпрототипная»), и, во-вторых, компилятор дублирует для «входной точки» стековый фрейм

```
push   ebp
mov    ebp, esp
```

несмотря на то, что он уже у нас есть. В данном случае, компилятор оказывает нам «медвежью услугу», проявляя собственную инициативу. Поэтому приходится наш стековый фрейм удалять, если мы желаем корректной работы библиотеки. Приведем окончательную правку входной функции **LibMain**.

```
; =============== S U B R O U T I N E =======================================  ; .text:773C4256
; Attributes: bp-based frame  ; .text:773C4256
; BOOL __stdcall LibMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)  ; .text:773C4256
public    LibMain  ; .text:773C4256
LibMain    proc near l_hLibModule:DWORD, l_fdwReason:DWORD, l_lpReserved:DWORD  ; .text:773C4256
   ; l_hLibModule    = dword ptr  8  ; .text:773C4256
   ; l_fdwReason     = dword ptr  0Ch  ; .text:773C4256
   ; l_lpReserved    = dword ptr  10h  ; .text:773C4256

   mov     edi, edi  ; .text:773C4256
   ; push    ebp  ; .text:773C4258
   ; mov     ebp, esp  ; .text:773C4259
   mov     eax, l_fdwReason  ; [ebp+l_fdwReason]  ; .text:773C425B
   sub     eax, 0  ; .text:773C425E
   jz      short loc_773C4279  ; .text:773C4261
   dec     eax  ; .text:773C4263
   jnz     short loc_773C4281  ; .text:773C4264
   push    l_hLibModule  ; [ebp+l_hLibModule] ; hLibModule  ; .text:773C4266
   call    ds:_imp__DisableThreadLibraryCalls@4  ; imp__DisableThreadLibraryCalls(x)  ; .text:773C4269
   push    l_hLibModule  ; [ebp+l_hLibModule]  ; .text:773C426F
   call    _ProcessAttach  ; .text:773C4272
   jmp     short loc_773C4284  ; .text:773C4277
; ---------------------------------------------------------------------------  ; .text:773C4279

loc_773C4279:            ; CODE XREF: LibMain+Bj  ; .text:773C4279
   push    l_hLibModule  ; [ebp+l_hLibModule]  ; .text:773C4279
   call    _ProcessDetach  ; .text:773C427C

loc_773C4281:            ; CODE XREF: LibMain+Ej  ; .text:773C4281
   xor     eax, eax  ; .text:773C4281
   inc     eax  ; .text:773C4283

loc_773C4284:            ; CODE XREF: LibMain+21j  ; .text:773C4284
   pop     ebp  ; .text:773C4284
   retn    0Ch  ; .text:773C4285
LibMain        endp  ; .text:773C4285
; ---------------------------------------------------------------------------  ; .text:773C4285
   db 5 dup(0CCh)  ; .text:773C4288
```

Перекомпиляция comctl32.dll

Завершив, наконец, все исправления мы получаем наш искомый библиотечный файл **comctl32.dll** ☺, вместе со всеми причитающимися, экспортируемыми (по именам и ординалам) функциями. Правда, протестировать библиотечный файл уже не так легко, как **exe**-шник. Можно, тем не менее, написать небольшой тестовый пример, который вызывает некоторые функции из этой библиотеки. Например, к результатам исследований приложен файл **mb.asm**, который вызывает достаточно сложную функцию **DllInstall**, а та, в свою очередь, вызывает не менее сложную функцию **InitCommonControlsEx** из **comctl32.dll**. При неудачном завершении **DllInstall** будет сообщение об ошибке, иначе сообщение об удачном запуске ☺.

Тестовая программа mb.asm

```
.686p
.mmx
.model flat

includelib Lib\kernel32.lib
includelib Lib\user32.lib

; ============================================================================
; Imports from Kernel32.dll
; ============================================================================

; HMODULE __stdcall LoadLibraryA(LPCWSTR lpLibFileName)
extrn _imp__LoadLibraryA@4 : dword

; FARPROC __stdcall GetProcAddress(HMODULE hModule, LPCSTR lpProcName)
extrn _imp__GetProcAddress@8 : dword

; BOOL __stdcall FreeLibrary(HMODULE hLibModule)
extrn _imp__FreeLibrary@4 : dword

; void __stdcall ExitProcess(UINT uExitCode)
extrn _imp__ExitProcess@4 : dword

; ============================================================================
; Imports from User32.dll
; ============================================================================

; int __stdcall MessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT
uType)
extrn _imp__MessageBoxA@16 : dword

MB_OK           equ 0h

; ============================================================================

.data

hOurDll         dd 0   ; HMODULE hOurDll

szTestDll       db "comctl32.dll", 0
szFuncName      db "DllInstall", 0   ; Вызывает также InitCommonControlsEx

Header          db "Info", 0
ErrHeader       db "Error", 0
ErrMsgDll       db "Load comctl32.dll is wrong!", 0
MsgInfo         db "Function DllInstall is Ok!", 0
ErrMsgFunc      db "Function DllInstall is wrong!", 0

; ============================================================================

.code

_Start:
  push offset szTestDll
```

```
  call ds:_imp__LoadLibraryA@4   ; LoadLibraryA(x)

  mov hOurDll, eax

  test eax, eax
  jz short ErrorDll

  push offset szFuncName ; "DllInstall"
  push eax   ; hOurDll

  call ds:_imp__GetProcAddress@8 ; GetProcAddress(x,x)

  test eax, eax
  jz short ErrorFunc

  push 0
  push 1

  call eax   ; szFuncName

  push MB_OK
  push offset Header
  push offset MsgInfo
  push 0

  call _imp__MessageBoxA@16   ; Положительное сообщение
  jmp short Free

ErrorDll:
  push MB_OK
  push offset ErrHeader
  push offset ErrMsgDll
  push 0

  call _imp__MessageBoxA@16   ; Отрицательное сообщение
  jmp short Quit

ErrorFunc:
  push MB_OK
  push offset ErrHeader
  push offset ErrMsgFunc
  push 0

  call _imp__MessageBoxA@16   ; Отрицательное сообщение

Free:
  push     hOurDll

  call     ds:_imp__FreeLibrary@4   ; FreeLibrary(x)

Quit:
  push 0

  call _imp__ExitProcess@4

end _Start
```

Запускаем тест, получаем что надо (рис. 2) ☺.

Рис. 2. Результат выполнения простого теста для перекомпилированной **comctl32.dll**.

Интересно, что если удалить наш файл **comctl32.dll**, то вызов функции **DllInstall** также пройдет успешно. Поэтому изменим слегка **mb.asm** в **mb~.asm**, за счет правки в нем строки «**comctl32.dll**» на «**comctl32~.dll**». Тогда если будет файл **comctl32~.dll**, то вызов пройдет успешно, а если нет, то появится сообщение об ошибке.

Радикальное тестирование скомпилированной библиотеки

Однако чтобы не ходить вокруг да около, можно просто подменить данную **dll**-ку, в операционной системе, в одной из подпапок **\WINDOWS\WinSxS**, там, где лежит реально используемая, практически всеми приложениями, библиотека **comctl32.dll**. Чтобы узнать ее точный путь, достаточно посмотреть полную ссылку в каком-нибудь **exe**-файле, который использует данную **dll**-ку, Например, в файле **regedit.exe** (рис. 3).

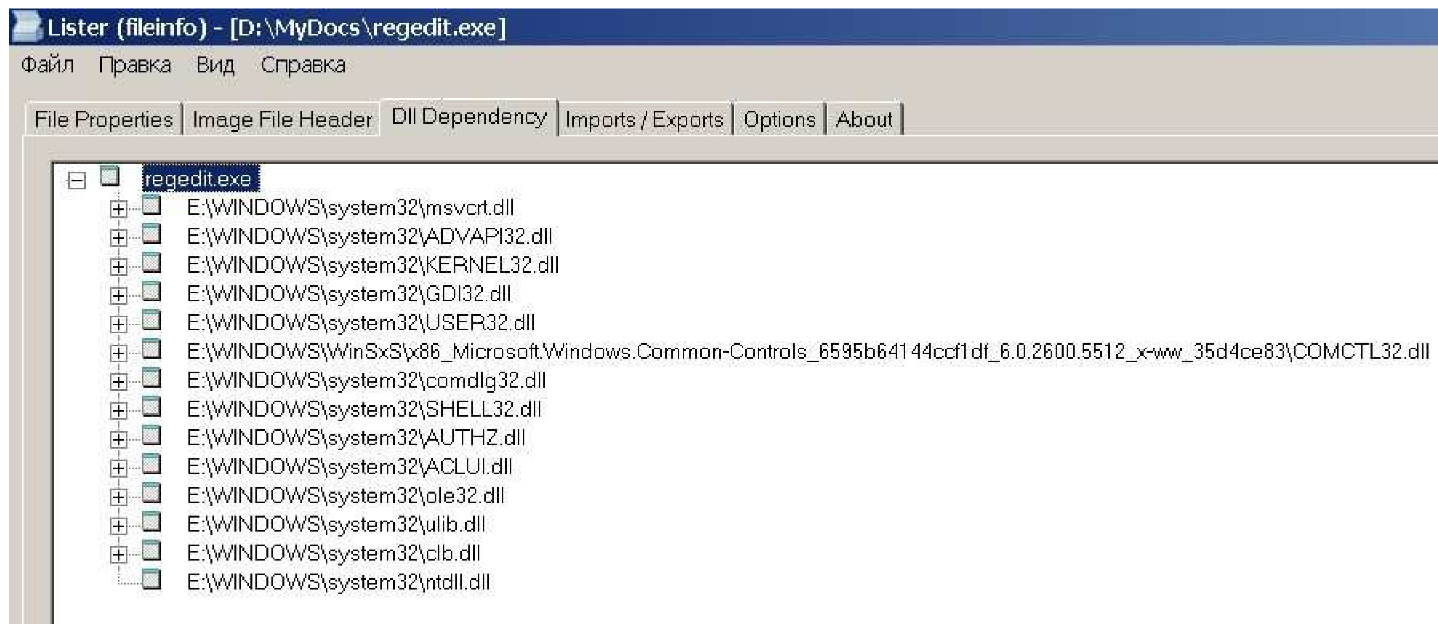

Рис. 3. Определение полного пути расположения системной **comctl32.dll**.

Результат такой подмены может быть катастрофическим. Он и был таковым на первых порах ☺. Однако мы не зря заранее уделили много времени экспорту функций, как по именам, так и по ординалам. Это помогло, система относительно корректно запустилась, если не считать потерю иконок на рабочем столе и в проводнике **Windows** (рис. 4) ☺. Впрочем, это настолько непринципиально, по сравнению с достигнутым результатом, что на их поиск и не хочется пока тратить время. Во всем остальном система работает достаточно стабильно. Хотя, абсолютной гарантии никто не даст ☺.

Выводы

Не хочется сильно перехваливать **IdaPro v.6.1 demo**, но полученные результаты, откровенно говоря, превысили ожидаемые ☺. Думал, что реально потребуется больше сил на тот же эффект. Сейчас самое время поразмышлять о непосредственной работе с полученным ассемблерным кодом, чтобы, с одной стороны, повысить свою квалификацию в нижнеуровневом программировании, а с другой, научиться эффективно манипулировать большими объемами **asm**-кода для получения нужных практических результатов. Более конкретно об этом можно будет поговорить в следующих статьях.

Примечание

К данному тексту приложен файл **IdaPro61DemoTest.005** (http://erfaren.narod.ru/Asm/IdaPro61DemoTest.005 - измените расширение в **zip**), с результатами исследований. Также приложены копии этого **zip**-файла, но с другими расширениями: http://erfaren.narod.ru/Asm/IdaPro61DemoTest.pdf и http://erfaren.narod.ru/Asm/IdaPro61DemoTest.txt . Их

также следует переименовывать в **zip**-файл. Сделано это из-за неудобств скачивания файлов, с определенными расширениями, различными браузерами.

Можно также посмотреть **html** версию этой статьи (http://erfaren.narod.ru/Asm/Erfaren005.htm) либо ее **pdf**-файл (http://erfaren.narod.ru/Asm/Erfaren-005-Test-IdaPro61-demo.pdf).

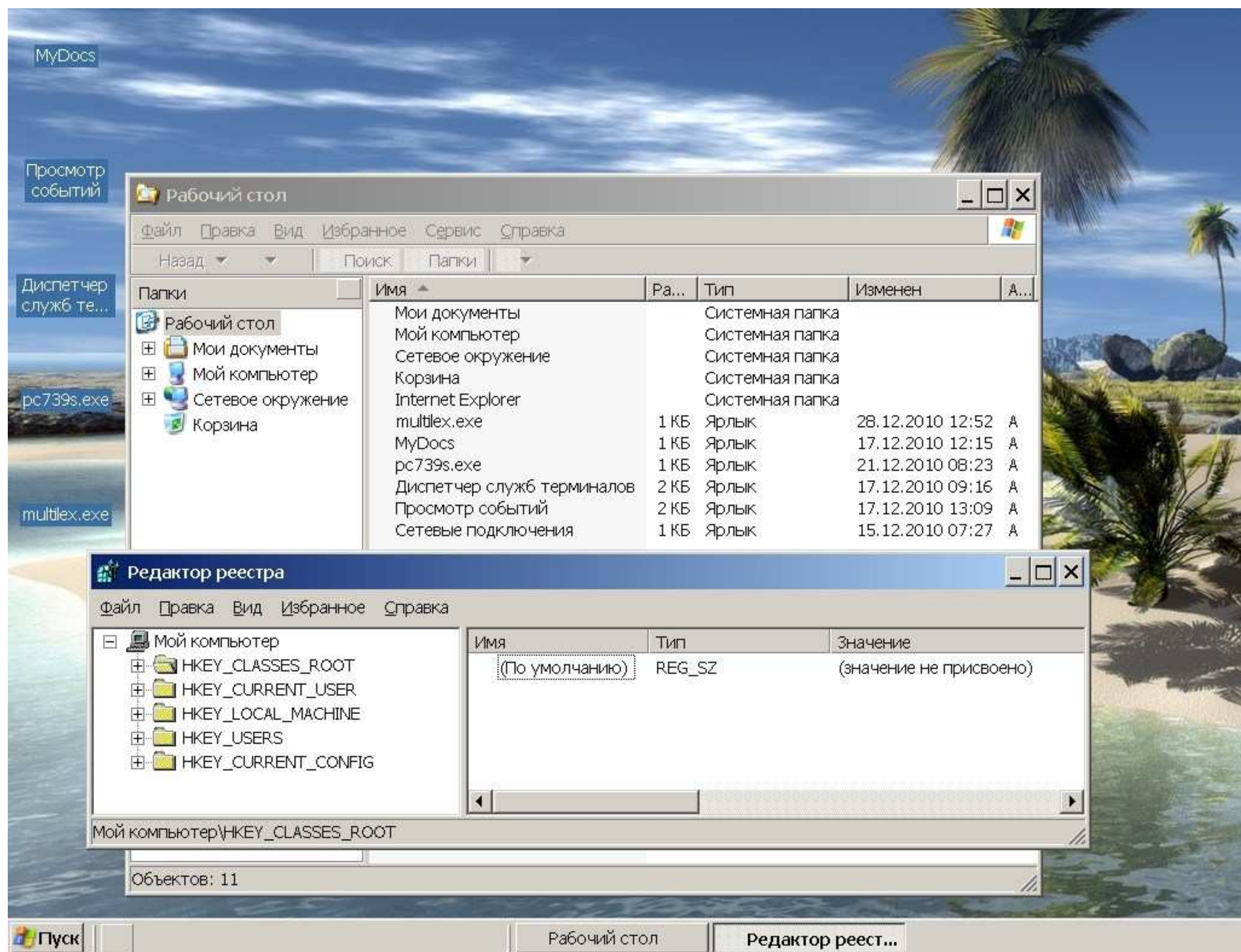

Рис. 4. Результат подмены перекомпилированной **comctl32.dll** вместо системной библиотеки.