Руководство программиста Free Pascal

Руководство программиста для Free Pascal,
Версия 3.0.4
Версия документа 3.0

Сентябрь 2017 (оригинал)
(перевод на русский язык) Не окончен

Авторы:
Michaël Van Canneyt

Переводчики:
Поляков Андрей Валерьевич
Google переводчик
Чигрин Виталий Николаевич (адаптировал и переработал)

Содержание

ОБ ЭТОМ ДОКУМЕНТЕ

Этот документ представляет собой руководство программиста на Free Pascal.

Документ описывает некоторые особенности компилятора Free Pascal и коротко рассказывает о том, как компилятор генерирует код, а также о том, как вы можете изменить этот код. Достаточно подробно описана внутренняя работа компилятора, но не описано, как использовать компилятор *(для этого см. Справочное руководство Free Pascal)*. Здесь также не описана работа библиотеки времени выполнения (RTL). Лучший путь для изучения RTL – это изучение её исходных кодов.

Описанные здесь вещи могут оказаться полезными, когда требуется применение более гибких конструкций языка, чем имеется в стандартном языке Pascal *(см. Справочник пользователя Free Pascal)*.

Потому как компилятор непрерывно совершенствуется разработчиками, этот документ может не содержать описания некоторых новшеств. Если вы нашли ошибки в документе, свяжитесь с разработчиками.

От переводчика:
Я старался сохранить исходный вариант документации. Основную часть работы сделана **Поляковым Андреем Валерьевичем**, за что ему огромное спасибо. Однако он делал перевод документации для Free Pascal **2.4.2**. С тех пор "утекло много воды" и в исходную документацию внесены изменения, я попытался это учесть. Я пытался переработать документацию под Free Pascal **3.0.4**. Но мог пропустить или не заметить несоответствия. Так что если Вами будут замечены несоответствия или др замечания просьба присылать их по адресу vchigrin@mail.ru, я постараюсь это исправить. Даже простое "обращение внимание" имеет смысл.

1. ДИРЕКТИВЫ КОМПИЛЯТОРА

1.1 Введение

Free Pascal поддерживает директивы компилятора в исходном файле: в основном это те же самые директивы, что и в компиляторах Turbo Pascal, Delphi и Pascal для операционных систем Макинтош. Некоторые директивы используются только для совместимости и не имеют какого-либо эффекта.

Директивы компиляции всегда пишутся в фигурных скобках:

```
{$DIRECTIVE [value]}
```

Не должны использоваться скобки `(* *)` или `//` для директив компиляции, также ***не могут*** использоваться обозначения C++.

То что директива пишется как комментарий означает, что это не инструкция Pascal. Это значит что **если указать несуществующую директиву, то это не приведёт к ошибке**, компилятор просто дает предупреждение о недопустимой/неизвестной директиве.

Имеются различия между глобальными и локальными директивами:

- ***Локальные директивы*** эффективны с того момента, когда они встречаются в тексте и до того момента, когда они будут заменены другой директивой или такой же директивой с другими параметрами. Локальные директивы могут использоваться в файле более одного раза.
- ***Глобальные директивы*** действуют на весь компилируемый код. Они могут быть указаны только один раз для всего файла. Это означает также, что действие директивы заканчивается, когда откомпилирован текущий модуль. Действие глобальной директивы не распространяется на другие модули.

Некоторые директивы могут принимать только логические значения: знак «+» включает директиву, знак «−» отключает директиву. Такие директивы также называют переключателями. Многие переключатели также имеют короткую и длинную форму записи. В таких случаях приводятся обе формы записи *(как короткая, так и длинная)*.

Для длинной формы записи знаки «+» и «−» могут быть заменены ключевыми словами `ON` и `OFF` соответственно.

Таким образом, запись `{$I+}` эквивалентна записи `{$IOCHECKS ON}` или `{$IOCHECKS +}`, а `{$C-}` эквивалентна записи `{$ASSERTIONS OFF}` или `{$ASSERTIONS -}`.

Длинные формы переключателей являются такими же, как и их аналоги в Delphi.

1.2. Локальные директивы

Локальные директивы могут использоваться более одного раза в модуле или программе. Локальная директива будет влиять на поведение компилятора с того момента, когда она встречаются в тексте и до того момента, пока другая директива не отменит действие первой или не закончится текущий модуль или файл. Если локальная директива дублируется аргументом командной строки, то аргумент командной строки используется для каждого компилируемого файла *(по умолчанию)*.

1.2.1. $A или $ALIGN : Выравнивание данных

Директива `{$ALIGN}` может быть использована для выбора стратегии выравнивания данных компилятора для записей. Директива принимает числовой аргумент, который может быть **1**, **2**, **4**, **8**, **16** или **32**, указывающий границу выравнивания в байтах. Для этих значений директива имеет тот же эффект, что и директива `{$PACKRECORDS}` (см. раздел 1.2.60. $PACKRECORDS : Выравнивание элементов записи [35]).

Таким образом, следующая директива

```
{$ALIGN 8}
```

эквивалента директиве

```
{$PACKRECORDS 8}
```

И указывает компилятору, что все данные внутри записи должны быть выровнены на **8**-байтовой границе.

В режиме MACPAS дополнительно могут быть следующие значения:

MAC68K – устанавливает выравнивание следующих m68K ABI.
POWER – устанавливает выравнивание следующих PowerPC ABI.
POWERPC – устанавливает выравнивание следующих PowerPC ABI.
RESET – переустанавливает выравнивание по умолчанию.
ON – то же самое, что указать **4**.
OFF – то же самое, что указать **1**.

Эти значение не доступны с директивой `{$PACKRECORDS}`.

1.2.2. $A1, $A2, $A4 и $A8

Эти директивы имеют такое же действие, что и директива `$PACKRECORDS` (см. раздел 1.2.60. $PACKRECORDS : Выравнивание элементов записи [35]), но они имеют указатель на выравнивание уже в имени директивы. Таким образом, следующая директива:

```
{$A8 }
```

эквивалентна

{$PACKRECORDS 8}

Учтите, что специфические случаи $PACKRECORDS не могут быть представлены этим способом.

1.2.3. $ASMMODE : Режим ассемблера (только для Intel 80x86)

Директива {$ASMMODE XXX} сообщает компилятору, какой тип ассемблера он может ожидать в блоке asm. Здесь XXX должно быть одним из следующих вариантов:

att – указывает на то, что блок asm содержит синтаксис ассемблера AT&T.
intel – указывает на то, что блок asm содержит синтаксис ассемблера Intel.
direct – говорит компилятору, что блоки asm должны быть скопированы непосредственно в ассемблерный файл. Это невозможно для использования таких ассемблерных блоков, когда используется внутренний ассемблер компилятора.

Эти переключатели являются локальными, и сохраняют свои значения до конца компилируемого модуля, кроме случая, когда они будут заменены другой директивой данного типа. Переключатель командной строки -R ссылается на данную директиву.

По умолчанию установлен ассемблер AT&T.

1.2.4. $B или $BOOLEVAL : Полная проверка логических выражений

По умолчанию компилятор использует сокращённую проверку логических выражений, т. е. оценка логического выражения прекращается, как только результат выражения уже известен. Для изменения этого поведения может быть использован переключатель {$B }: если его аргумент равен +, то компилятор всегда будет оценивать все условия выражения. Если это - *(по умолчанию)*, то компилятор будет всегда поверять только некоторые *(необходимые)* условия в выражении.

Так, в следующем примере, логическая функция Bofu, никогда не будет вызвана *(если использована директива {$B-})*.

```
if False and Bofu then
…
```

Следовательно, любые действия, которые реализует функция Bofu, не будут выполнены. Если компиляция выполняется с директивой {$B+}, то функция Bofu будет вызывана в любом случае.

1.2.5. $C или $ASSERTIONS : Поддержка формальных утверждений

Переключатель {$ASSERTIONS} определяет, нужно ли операторы формальных утверждений компилировать в двоичный код. Если переключатель включен, то

```
Assert(BooleanExpression, AssertMessage);
```

будет компилироваться в двоичный код. Если BooleanExpression равно False, то RTL будет проверять, установлена ли AssertErrorProc. Если она установлена, она будет вызвана с параметрами сообщения AssertMessage, именем файла, LineNumber и адресом. Если она не установлена, то генерируется ошибка времени выполнения **227**.

AssertErrorProc определена как

```
Type
   TAssertErrorProc=procedure(Const msg, fname : String;
                              lineno, erroraddr : Longint);
Var
   AssertErrorProc = TassertErrorProc;
```

Она может быть использована, в основном, для отладки. Модуль system устанавливает AssertErrorProc для обработчика, который отображает сообщение на stderr и просто выходит с ошибкой времени выполнения 227. Модуль sysutils отлавливает ошибку времени выполнения **227** и генерирует исключение EAssertionFailed.

1.2.6. $BITPACKING : Включить битовую упаковку

Директива $BITPACKING указывает компилятору, нужно ли использовать битовую упаковку, если он обнаружил ключевое слово packed для структурного типа. Возможные значения ON и OFF. Если ON, то компилятор упаковывать структуры, если обнаружит ключевое слово Packed.

В следующем примере запись TMyRecord будет упакована:

```
{$BITPACKING ON}
Type
   TMyRecord = packed record
   B1,B2,B3,B4 : Boolean;
end;
```

Учтите, что:

- Директива $BITPACKING игнорируется в режиме macpas, где упакованные записи всегда имеют битовую упаковку.
- Ключевое слово bitpacked можно использовать всегда для форсирования поразрядной упаковки, не взирая на значение директивы $BITPACKING и не смотря на режим.

1.2.7. $CALLING : Определить соглашение о вызовах

Директива {$CALLING } указывает компилятору, какое соглашение о вызовах должно использоваться, если не указано:

```
{$CALLING REGISTER}
```

По умолчанию это **REGISTER**. Существуют следующие соглашения о вызовах:

```
default
register
cdecl
pascal
safecall
stdcall
oldfpccall
```

Более подробные разъяснения относительно соглашений о вызовах см. в разделе 6.3. Механизм вызова[93]. В специальных случаях можно использовать DEFAULT для записи соглашений о вызовах по умолчанию.

1.2.8. $CHECKPOINTER : Проверять значения указателя

Директива {$CHECKPOINTER} включает *(значение ON)* или выключает *(значение OFF)* проверку указателя «кучи». Если проверка указателя «кучи» включена, а код компилируется с включенной опцией -gh (трассировка кучи), то проверка выполняется при разыменовании указателя. Проверка будет подтверждать, что указатель содержит правильные данные, то есть указывает на размещение, которое доступно для чтения программой: стек или «куча». Если нет, то генерируется ошибка времени выполнения **216** или **204**.

Если код компилируется без переключателя -gh, то эта директива не имеет эффекта. Учтите, что она существенно замедляет код.

1.2.9. $CODEALIGN : Установить выравнивание кода

Этот переключатель устанавливает выравнивание кода. Он принимает аргумент, который определяет выравнивание в байтах *(выравнивание – это установка размера для разных типов данных)*.

```
{$CODEALIGN 8}
```

Имеется некоторое количество аргументов, которые можно указать для более гибкой настройки поведения компилятора.

Общий формат директивы:

```
{$CODEALIGN PARAM=VALUE}
```

где PARAM – это настраиваемый параметр, а VALUE – числовое значение, определяющее выравнивание. PARAM может содержать следующие строки:

PROC – установить выравнивание для точек вхождения процедур.

JUMP – установить выравнивание для переходов к назначенным местоположениям.
LOOP – установить выравнивание для циклов (`for, while, repeat`).
CONSTMIN – минимальное выравнивание для констант *(типизированных и нетипизированных)*.
CONSTMAX – максимальное выравнивание для констант *(типизированных и нетипизированных)*.
VARMIN – минимальное выравнивание для статичных и глобальных переменных.
VARMAX – максимальное выравнивание для статичных и глобальных переменных.
LOCALMIN – минимальное выравнивание для локальных переменных.
LOCALMAX – максимальное выравнивание для локальных переменных.
RECORDMIN – минимальное выравнивание для полей записи.
RECORDMAX – максимальное выравнивание для полей записи.

По умолчанию размер структуры данных определяет выравнивание:

- `SmallInt` будет выровнено на **2** байта.
- `LongInt` будет выровнено на **4** байта.
- `Int64` будет выровнено на **8** байтов.

С указанными выше переключателями можно определить минимальное и максимальное выравнивание. Максимально допустимое выравнивание имеет значение, только если оно меньше, чем натуральный размер. То есть максимальное выравнивание *(например, VARMAX)* равное **4**, будет приводить к выравниванию в **4** байта: `Int64` будет уменьшено также до **4** байтов, а `SmallInt` так и останется выровнено на **2** байта.

Эти значения можно также указать в командной строке как

```
-OaPARAM=VALUE
```

1.2.10. $COPERATORS : Разрешить C-подобные операторы

Этот логический переключатель определяет, разрешать ли использование C-подобных операторов. По умолчанию это запрещено. Если применить эту директиву:

```
{$COPERATORS ON}
```

То следующие операторы будет разрешено использовать:

```
Var
   I : Integer;
begin
 I := 1;
 I += 3; // Прибавить 3 к I и присвоить результат переменной I;
 I -= 2; // Вычесть 2 из I и присвоить результат переменной I;
 I *= 2; // Умножить I на 2 и присвоить результат переменной I;
 I /= 2; // Разделить I на 2 и присвоить результат переменной
I;
```

```
end;
```

1.2.11. $DEFINE или $DEFINEC : Определить идентификатор

Директива

```
{$DEFINE name}
```

Определяет имя *(name)* идентификатора. Определение этого идентификатора остаётся до конца текущего модуля *(то есть модуля или программы)*, или до тех пор, пока не будет применена директива `$UNDEF name`.

Если имя уже определено, то эта директива не имеет эффекта. Имя не чувствительно к регистру.

Идентификаторы, которые определены в модуле, не сохраняются в файле модуля, также они не экспортируются из модуля.

В режиме `Mac Pascal` директива `$DEFINEC` эквивалентна директиве `$DEFINE` и предоставляется для совместимости с Mac Pascal.

Директива `{$DEFINE}` может использоваться для определения макросов или констант во время компиляции:

```
{$DEFINE cdecl:=stdcall}
```

переопределим стандартный модификатор `cdecl` как `stdcall`.

Более подробную информацию о макросах и константах времени-компиляции, можно найти в разделе 2.2. Макросы [64].

Директива `{$ DEFINE}` имеет эквивалент командной строки `-d`

С помощью `-dNAME` можно определять символ `Name` *(в командной строке)*. Используя

```
-dcdecl:=stdcall
```

можно переопределить стандартный модификатор `cdecl` как `stdcall` в командной строке.

1.2.12. $ELSE : Переключатель условной компиляции

Директива `{$ELSE}` является переключателем между компилируемым и игнорируемым участками текста исходного кода. Этой директиве предшествует директива `{$IFxxx}`, а за директивой `{$ELSE}` должна быть директива `{$ENDIF}`. Любой текст после ключевого слова `ELSE` но перед фигурной скобкой игнорируется:

```
{$ELSE этот текст игнорируется}
```

Это то же самое, что

```
{$ELSE}
```

Это может оказаться полезным для описания директивы.

1.2.13. $ELSEC : Переключатель условной компиляции

В режиме MACPAS эта директива может быть использована как альтернатива директиве $ELSE. Она поддерживается для совместимости с существующими компиляторами Pascal для Mac OS.

1.2.14. $ELSEIF или $ELIFC : Переключатель условной компиляции

Эта директива может быть использована как сокращённая команда для новой директивы {$IF } внутри блока условной компиляции вместо {$ELSE }:

```
{$IF XXX}
// Здесь код XXX
{$ELSEIF YYY}
// Здесь код YYY
{$ELSE}
// А здесь код по умолчанию
{$ENDIF}
```

Приведённый выше пример является эквивалентным со следующим примером:

```
{$IF XXX}
// Здесь код XXX
{$ELSE }
{$IF YYY}
// Здесь код YYY
{$ELSE}
// А здесь код по умолчанию
{$ENDIF}
{$ENDIF}
```

Директива в вышеуказанном выражении, следует за директивой {$IF }.

Вариант {$ELIFC } допускается только в режиме MACPAS.

1.2.15. $ENDC : Завершение условной компиляции

В режиме MACPAS эта директива может быть использована как альтернатива директиве $ENDIF. Она поддерживается для совместимости с существующими компиляторами Pascal для Mac OS.

1.2.16. $ENDIF : Завершение условной компиляции

Директива{$ENDIF} завершает блок условной компиляции, инициализированный директивой {$IFxxx}. Любой текст после ключевого слова ENDIF но перед фигурной скобкой игнорируется:

```
{$ENDIF Этот текст игнорируется}
```

Это то же самое, что и

```
{$ENDIF}
```

Это может оказаться полезным для описания того, какой блок завершён.

1.2.17. $ERROR или $ERRORC : Генерировать сообщение об ошибке

Следующий код

```
{$ERROR Этот код неправильный !}
```

Будет отображать сообщение об ошибке, если компилятор обнаружит её, и увеличивать счётчик ошибок компилятора. Компилятор продолжит компиляцию, но код будет пропущен.

Вариант `$ERRORC` используется для совместимости с Mac Pascal.

1.2.18 $ENDREGION: Конец разбираемого региона

Эта директива синтаксического анализа для совместимости с Delphi, но она игнорируется компилятором. В Delphi, она знаменует конец разборной области в IDE.

1.2.19. $EXTENDEDSYM: Игнорируемый

Эта директива синтаксического анализа для совместимости с Delphi, но она игнорируется. Если компилятор обнаружит эту директиву, то будет отображено предупреждение.

1.2.20 $EXTENDELSYM: Игнорируемый

Эта директива синтаксического анализа для совместимости с Delphi, но она игнорируется.

1.2.21. $F : Дальний или ближний вызов функций

Эта директива распознаётся для совместимости с `Turbo Pascal`. Для **32**-разрядных и **64**-разрядных программных моделей концепция ближних и дальних вызовов не имеет значения, поэтому директива игнорируется. Предупреждение выводится на экран как напоминание.

Например, следующий кусок кода:

```
{$F+}
```

```
Procedure TestProc;
begin
   Writeln ('Hello From TestProc');
end;
begin
   testProc
end.
```

Генерируется в следующий выход компилятора:

```
malpertuus: >pp -vw testf
Compiler: ppc386
Units are searched in: /home/michael;/usr/bin/;/usr/lib/
ppc/0.9.1/linuxunits
Target OS: Linux
Compiling testf.pp
testf.pp(1) Warning: illegal compiler switch
7739 kB free
Calling assembler…
Assembled…
Calling linker…
12 lines compiled,
1.00000000000000E+0000
```

Здесь можно видеть, что уровень «многословности» сообщений компилятора был установлен *(опция `-vw`)* для отображения предупреждений.

Когда функция объявляется как `Far` *(это имеет такой же эффект, как помещение функции между директивами `{$F+}` …`{$F-}`)*, компилятор также генерирует предупреждение:

```
testf.pp(3) Warning: FAR ignored
```

То же справедливо и для процедур, объявленных как `Near`. В этом случае отобразится предупреждение:

```
testf.pp(3) Warning: NEAR ignored
```

1.2.22. $FATAL : Генерировать сообщения о фатальных ошибках

Следующий код

```
{$FATAL Этот код неправильный !}
```

Будет отображать сообщение об ошибке, если компилятор обнаружит её, и компилятор немедленно прекратит процесс компиляции.

Это может оказаться очень полезным при совместном использовании с операторами `{$IFDEF}` или `{$IFOPT}`.

1.2.23. $FPUTYPE : Выбрать тип сопроцессора

Эта директива выбирает тип сопроцессора, используемого для вычислений с плавающей точкой. Эта директива должна сопровождать тип модуля

плавающей точки. Следующие значения определены для целевых процессоров:

all
: SOFT: эмуляция сопроцессора FPC (уже не применяется).

i386
: X87, SSE, SSE2: код компилируется с SSE, использующим SSE для вычислений выражений с плавающей точкой типа Single. Этот код выполняется только на Pentium III и выше, или AthlonXP и выше. Код компилируется с SSE2, используя модуль SSE для вычислений с типами данных single и double. Этот код выполняется только для PentiumIV и выше или Athlon64 и выше.

x86-64
: SSE64

powerpc
: STANDARD

arm
: LIBGCC, FPA, FPA10, FPA11, VFP.

Эта директива связана с опцией командной строки -Cf.

1.2.24. $GOTO : Поддерживать Goto и Label

Если указано {$GOTO ON}, то компилятор будет поддерживать операторы Goto и объявления меток *(Label)*.

По умолчанию установлено $GOTO OFF. Эта директива связана с опцией командной строки -Sg.

Например, следующий код будет откомпилирован:

```
{$GOTO ON}
label Theend;
begin
   If ParamCount=0 then
        GoTo TheEnd;
      Writeln ('Вы указали опции командной строки');
      TheEnd:
end.
```

ПРИМЕЧАНИЕ

Если компиляция ассемблерного кода использует встроенный ассемблер, любые метки, используемые в ассемблерном коде, должны быть объявлены, и должна использоваться директива {$GOTO ON}.

1.2.25. $H или $LONGSTRINGS : Использовать AnsiStrings

Если указана директива {$LONGSTRINGS ON}, то ключевое слово String *(не длина спецификатора)* будет обрабатываться как AnsiString, и компилятор

будет обрабатывать соответствующую переменную как **ansistring**, и будет генерировать соответствующий код. Это переключатель связан с опцией командной строки `-Sh`.

По умолчанию использование **ansistrings** отключено, что соответствует директиве `{$H-}`. Системный модуль откомпилирован без **ansistrings**, все его функции принимают аргументы **shortstring**. То же справедливо для всех модулей RTL, кроме модуля sysutils, который откомпилирован с **ansistrings**.

Однако оператор `{$MODE}` по умолчанию влияет на значение директивы `{$H}`: директива `{$MODE DELPHI}` подразумевает включение `AnsiString {$H+}`, все другие режимы отключают `AnsiString`. В результате вы всегда должны помещать директиву `{$H+}` после директивы режима, если хотите включить `AnsiString`. Это поведение можно изменить, кроме некоторых старших версий Free Pascal.

1.2.26. $HINT : Генерировать сообщение с подсказкой

Если генерация подсказок включена с помощью опции командной строки `-vh` или директивы[20] `{$HINTS ON}`, то сообщение с подсказкой

```
{$Hint Этот код должен быть оптимизирован }
```

будет отображаться на экране, когда компилятор найдёт эту директиву.

По умолчанию подсказки не генерируются.

1.2.27. $HINTS : Разрешить подсказки

Переключатель `{$HINTS ON}` разрешает генерацию подсказок *(см. 1.2.26. $HINT : Генерировать сообщение с подсказкой[20])*. Переключатель `{$HINTS OFF}` отключает генерацию подсказок. В отличие от опции командной строки `-vh`, это локальный переключатель, который может оказаться полезным для проверки участков кода.

1.2.28. $HPPEMIT: Игнорируется

Эта директива используется для совместимости с Delphi, но игнорируется в Free Pascal.

1.2.29. $IF : Начать условную компиляцию

Директива `{$IF expr}` продолжит компиляцию, если логическое выражение `expr` равно `True`. Если это выражение равно `False`, то участок исходного кода до первой директивы `{$ELSE}` или `{$ENDIF}`.

Компилятор должен быть способен вычислить выражение во время анализа.

Это означает, что переменные или константы, определённые в исходном коде, не могут быть использованы. Однако макросы и идентификаторы могут использоваться.

Более подробную информацию по этой теме вы можете найти в разделе 2.4.1. Определение[66] об условной компиляции.

1.2.30. $IFC : Начать условную компиляцию

В режиме MACPAS эта директива может использоваться как альтернатива директиве $IF. Она поддерживается для совместимости с существующими компиляторами Pascal для Mac OS.

1.2.31. $IFDEF Имя : Начать условную компиляцию

Если идентификатор Имя **определен**, то директива {$IFDEF Имя} выполнит компиляцию текста, который следует после неё до первой директивы {$ELSE} или {$ENDIF}.

Если Имя не определено, то компиляция выполнена не будет.

1.2.32. $IFNDEF : Начать условную компиляцию

Если идентификатор Имя **не определен**, то директива {$IFNDEF Имя} выполнит компиляцию текста, который следует после неё до первой директивы {$ELSE} или {$ENDIF}.

Если Имя определено, то компиляция выполнена не будет.

1.2.33. $IFOPT : Начать условную компиляцию

Директива {$IFOPT switch} будет компилировать текст, который находится после неё, если переключатель switch в текущий момент находится в установленном состоянии. Если он находится ***НЕ*** в установленном состоянии, то компиляция продолжится после соответствующей директивы {$ELSE} или {$ENDIF}.

Пример:

```
{$IFOPT M+}
   Writeln('Компилирование с типом информации');
{$ENDIF}
```

В этом примере компиляция оператора Writeln будет выполнена только в том случае, если включена генерация типа информации.

ПРИМЕЧАНИЕ

Директива {$IFOPT} допускает только сокращённые записи опций, то есть запись {$IFOPT TYPEINFO} не допускается.

1.2.34. $IMPLICITEXCEPTIONS : Неявное завершение генерации кода

Компилятор неявно генерирует блок try…finally вокруг каждой процедуры, которой необходима инициализация или завершение переменных, присваивание окончательных результатов переменным в блоке finally. Такое поведение замедляет работу процедур (иногда до **5-10%**). **С помощью этой директивы генерация таких блоков может быть отключена.** Вы должны осторожно обращаться с этой директивой, потому что она может привести к утечке памяти, если исключение случится внутри подпрограммы. По этой причине она установлена в **ON по умолчанию**.

1.2.35. $INFO : Генерировать информационное сообщение

Если генерация информационных сообщений включена с помощью опции командной строки -vi, то директива

```
{$INFO Этот код был написан в дождливый день программистом Bugs Bunny}
```

будет отображать сообщение в том месте, где её обнаружит компилятор.

Эту директиву полезно использовать совместно с директивой {$IFDEF} для отображения информации о том, какой участок кода был откомпилирован.

1.2.36. $INLINE : Разрешить встраиваемый код

Директива {$INLINE ON} указывает компилятору, что процедурный модификатор Inline должен быть разрешён. Код процедуры, объявленной как встраиваемая (Inline), ***копируется*** в то место, откуда она вызывается. То есть реального вызова процедуры не происходит, код процедуры только копируется там, где это необходимо. В результате можно получить более быструю скорость выполнения программы, если используется много функций или процедур.

По умолчанию встраиваемые процедуры не разрешаются. Эта директива должна указываться для использования встраиваемого кода. Опция командной строки -Si эквивалентна этой директиве. Более подробно о встраиваемых подпрограммах написано в Справочное руководство Free Pascal.

1.2.37. $INTERFACES : Указать тип интерфейса

Директива {$INTERFACES} указывает компилятору, что он должен

принимать как родительский интерфейс для интерфейса, в объявлении которого явно не указан родительский интерфейс. По умолчанию используется интерфейс Windows COM `IUnknown`. Другие виды интерфейсов (CORBA или Java) могут не иметь этот интерфейс, поэтому для таких случаев нужно использовать эту директиву. Директива допускает следующие значения:

COM
: интерфейс будет потомком от `IUnknown` и выполняться подсчёт ссылок.

CORBA
: интерфейсы не будут иметь предка и не будут выполнять подсчёт ссылок *(поэтому ответственность подсчёта ссылок ложится на программиста)*. Интерфейс Corba идентифицируется как простая строка, поэтому он совместим со строками и не совместим с `TGUID`.

DEFAULT
: на сегодняшний день это COM.

1.2.38. $I или $IOCHECKS : Проверка ввода/вывода

Директива `{$I-}` или `{$IOCHECKS OFF}` указывает компилятору, что не нужно генерировать проверку кода ввода/вывода в программе. По умолчанию компилятор генерирует проверку кода ввода/вывода. Это поведение можно настроить глобально с помощью переключателя `-Ci`.

Если компиляция использует переключатель `-Ci`, то компилятор Free Pascal вставляет проверку кода ввода/вывода после каждого вызова кода ввода/вывода. Если происходит ошибка во время ввода или вывода, то будет генерироваться ошибка времени выполнения. Это переключатель можно также использовать отмены проверки.

Если генерируется код проверки ввода/вывода, то проверить, если что-то пошло не так, можно с помощью функции `IOResult`.

Наоборот, директива `{$I+}` включает проверку вода/вывода до тех пор, пока не встретится директива выключения проверки.

Примечание:

> При возникновении ошибки ввода-вывода и отключенной генерации кода проверки все последующие операции ввода-вывода игнорируются до тех пор, пока не будет очищен код состояния этой ошибки.

При вызове функции `IOResult` из модуля system она возвращает текущий статус ввода-вывода и сбрасывает его.

Результатом этого является то, что `IOResult` должен быть проверен после операции ввода/вывода, перед выполнением следующих операций ввода/вывода.

Наиболее часто этот переключатель используется для того, чтобы убедиться,

что открытие файла прошло без проблем, как это показано в следующем коде:

```
assign(f, 'file.txt');
{$I-}
rewrite(f);
{$I+}
if IOResult<>0 then
   begin
      Writeln('Ошибка открытия файла: "file.txt"');
      exit
   end;
```

См. описание функции `IOResult` в Справочное руководство Free Pascal, где можно найти список всех возможных ошибок, которые могут произойти при проверке ввода/вывода.

1.2.39 $IEEEERRORS : Разрешить проверку IEEE констант

Это логическое директива включает *(или выключает)* проверку констант `IEEE` *(с плавающей запятой)* на ошибку. Она локальный эквивалент глобального параметра командной строки `-C3`.

Следующая директива включает проверку ошибок `IEEE` *(с плавающей запятой)* констант:

```
{$IEEEERRORS ON}
```

1.2.40. $I или $INCLUDE : Подключить файл

Директива `{$I ИмяФайла}` или `{$INCLUDE ИмяФайла}` указывает компилятору, что следующие операторы нужно читать из файла `ИмяФайла`. Операторы из этого файла будут вставлены в код, как если бы они имелись в текущем файле.

Если файл с указанным именем существует, то он будет включен в текст программы. Если расширение не указано, компилятор добавит к файлу `ИмяФайла` расширение .pp.Другие файлы с этим именем и с другими расширениями не ищутся.

Имя файла может быть помещено в одинарные кавычки, они не рассматриваются как часть имени файла. Если имя файла содержит пробел, то оно должно быть заключено в одинарные кавычки

Директива

```
{$I 'my file name'}
```

пытаться включить файл my file name или my file name.pp.

А директива

```
{$I my file name}
```

попытается включить файл my или my.pp.

Если вместо имени файла будет звездочка *(*)*, то компилятор будет использовать имя модуля или имя программы в качестве имени файла и попытаться добавить его содержимое. Следующий код

```
unit testi;

interface

{$I *}

implementation

end.
```

будет включает в себя файл testi или testi.pp, если они существуют.

```
Type
  A = Integer;
```

Следует быть осторожным с этим механизмом, потому что имя модуля уже должен соответствовать имени файла модуля, а это будет ***рекурсивное*** включение модуля.

Подключаемые файлы могут быть вложенными, но не бесконечно глубоко. Количество файлов ограничено количеством файловых дескрипторов, доступных компилятору Free Pascal.

В отличие от Turbo Pascal, подключаемые файлы могут составлять сквозные блоки, то есть блок кода может начаться в одном файле (с ключевого слова `Begin`), а закончиться в другом файле (ключевым словом `End`). Наименьший элемент в подключаемом файле должен быть лексемой, то есть идентификатор, ключевое слово или оператор.

Компилятор будет искать подключаемые файлы в следующих местах:

1. Путь, указанный в имени подключаемого файла
2. Каталог, где находится текущий исходный файл
3. Путь, указанный в настройках для подключаемых файлов

Каталоги, где будет выполняться поиск подключаемых файлов, можно добавить с помощью опции командной строки `-Fi`.

1.2.41. $I или $INCLUDE : Включать информацию компилятора

В следующем формате:

```
{$INCLUDE %XXX%}
```

директива ***{$INCLUDE}*** вставляет строковую константу в исходный код.

Здесь `XXX` может быть одним из следующих значений:

DATE

Вставляется текущая дата. Она будет отформатирована как YYYY/MM/DD.

FPCTARGET

Вставляется имя целевого процессора. (не рекомендуется, лучше используйте FPCTARGETCPU).

FPCTARGETCPU

Вставляется имя целевого процессора.

FPCTARGETOS

Вставляется имя целевой операционной системы.

FPCVERSION

Вставляется номер текущей версии компилятора.

FILE

Вставляется имя файла, в котором найдена директива.

LINE

Вставляется номер строки, на которой находится директива.

LINENUM

Вставляется номер строки, на которой находится директива. В этом случае, результат это целое число, а не строка.

TIME

Вставляется текущее время, в формате HH:MM:SS.

Если XXX имеет другое значение, то предполагается что это имя переменной окружения. Её значение будет получено из окружения, если переменная существует. Если нет, то будет вставлена пустая строка. В результате эта директива будет генерировать макрос со значением, определённым в XXX, как если бы это была строковая константа в исходном коде *(или, в случае LINENUM , целое число)*.

Например, следующая программа

```
Program InfoDemo;
Const User = {$I %USER%};
begin
   Write('Эта программа была откомпилирована в ', {$I %TIME%});
   Writeln(' на ', {$I %DATE%});
   Writeln('пользователем ', User);
   Writeln('Версия компилятора: ', {$I %FPCVERSION%});
   Writeln('Целевой процессор: ', {$I %FPCTARGET%});
end.
```

выведет следующее:

```
Эта программа была откомпилирована в 17:40:18 на 1998/09/09
пользователем michael
Версия компилятора: 0.99.7
Целевой процессор: i386
```

Замечание:

Следует отметить, что включение DATE и TIME не заставит компилятор перекомпилировать файл каждый раз: используется дата и время последней компиляции.

1.2.42. $J или $WRITEABLECONST : Разрешить присваивание для типизированных констант

Этот логический переключатель указывает компилятору, нужно или нет разрешать присваивание для типизированных констант. По умолчанию - разрешено.

Следующий оператор отключить присваивание для типизированных констант:

```
{$WRITEABLECONST OFF}
```

После этого переключателя следующие операторы не будут компилироваться:

```
Const
   MyString : String = 'Некая подходящая строка';
begin
   MyString:='Некая другая строка';
end.
```

Но инициализация переменных, тем не менее, будет компилироваться:

```
Var
   MyString : String = 'Некая подходящая строка';
begin
   MyString:='Некая другая строка';
end.
```

1.2.43. $L или $LINK : Компоновать объектный файл

Директива `{$L ИмяФайла}` или `{$LINK ИмяФайла}` указывает компилятору, что файл с именем `ИмяФайла` должен быть скомпонован для программы. Она не используется для библиотек (см. раздел 1.2.46. $LINKLIB : Компоновать библиотеку[28]).

Компилятор будет искать этот файл в следующих местах:

1. Путь, указанный в имени объектного файла.
2. Каталог, где находится текущий исходный файл.
3. Путь, указанный в настройках для объектных файлов.

Каталоги, где будет выполняться поиск объектных файлов, можно добавить с помощью опции командной строки `-Fo`.

На системах LINUX и на операционных системах с чувствительной к регистру файловой системой *(таких, как системы UNIX)*, имя файл чувствительно к регистру и должно быть напечатано точно так же, как оно фигурирует в операционной системе.

ПРИМЕЧАНИЕ

Убедитесь, что объектный файл, который вы компонуете, имеет формат, известный компоновщику. Напечатайте в командной строке `ld` или `ld --help`, чтобы получить список форматов, известных компоновщику `ld`.

Другие файлы и опции можно передать в компоновщик, используя опцию командной строки -k. Можно использовать более одной опции. В этом случае они будут переданы в компоновщик в том порядке, в каком они указаны в командной строке, только перед именами объектных файлов, которые должны быть скомпонованы.

1.2.44 $LIBEXPORT : Ignored

Эта директива используется в компиляторе Darwin Pascal, и игнорируется в остальных случаях.

1.2.45. $LINKFRAMEWORK : Компоновать в структуру

Директива {$LINKFRAMEWORK Имя} будет компоновать в структуру с именем Имя. Этот переключатель доступен только для платформ Darwin.

1.2.46. $LINKLIB : Компоновать библиотеку

Директива {$LINKLIB Имя} выполняет компоновку в библиотеку с именем Имя. Она имеет тот же эффект, что и передача параметра -lимя в компоновщик.

Для примера рассмотрим следующий модуль:

```
unit getlen;
interface
{$LINKLIB c}
function strlen (P : pchar) : longint; cdecl;
implementation
function strlen (P : pchar) : longint; cdecl; external;
end.
```

Если будет выполнена компиляция командой

```
ppc386 foo.pp
```

здесь если foo.pp имеет описанный выше модуль в разделе uses, то компилятор будет компоновать программу в библиотеку C, передавая в компоновщик опцию -lc.

То же самое можно получить, удалив директиву linklib из описанного выше модуля, и указав -k -lc в командной строке:

```
ppc386 -k-lc foo.pp
```

Учтите, что компоновщик будет искать библиотеку по пути поиска библиотек для компоновщика: никогда не нужно указывать полный путь к библиотеке. Путь поиска библиотек для компоновщика можно установить опцией командной строки -Fl.

1.2.47. $M или $TYPEINFO : Генерировать информацию о типах

Для классов, которые компилируются в состоянии {$M+} или {$TYPEINFO ON}, компилятор будет генерировать Run-Time Type Information *(RTTI)*. Все классы-потомки от класса, который компилируется с состоянием {$M+}, также будут получать информацию RTTI. Любые классы, которые используются как поле или свойство, будут также получать информацию RTTI.

По умолчанию Run-Time Type Information генерируется для общедоступных разделов (published) создающих их эквиваленты для общедоступных (public) разделов. Только если класс (или один из его родительских классов) был откомпилирован в состоянии {$M+}, компилятор будет генерировать RTTI для методов и свойств в общедоступных разделах.

Объект TPersistent, который представлен в модуле classes *(часть RTL)*, сгенерирован в состоянии {$M+}. Генерация RTTI позволяет программистам получить доступ к общедоступным свойствам объекта, без необходимости знать актуальный класс объекта.

Run-Time Type Information доступна через модуль TypInfo, который является частью Free Pascal Run-Time Library.

ПРИМЕЧАНИЕ

Поточная система, реализованная на Free Pascal, требует, чтобы все потоковые компоненты были потомками от TPersistent. Возможно создание классов с общедоступными разделами, которые не являются потомками TPersistent, но эти классы не будут являться правильными потоками потоковой системы модуля Classes.

1.2.48. $MACRO : Разрешить использование макросов

В состоянии {$MACRO ON} компилятор позволяет использовать макросы в стиле C *(хотя не настолько детально)*. Макросы предоставляют средства для простого замещения текста. Эта директива эквивалентна опции командной строки -Sm. По умолчанию макросы запрещены.

Больше информации об использовании макросов вы можете найти в разделе 2.2. Макросы [64].

1.2.49. $MAXFPUREGISTERS : Максимальное количество регистров FPU для переменных

Директива {$MAXFPUREGISTERS XXX} указывает компилятору, сколько переменных с плавающей точкой можно хранить в регистрах плавающей точки

процессора Intel X86. Этот переключатель игнорируется, если используется переключатель оптимизации -Or (использовать регистровые переменные).

Это довольно сложно, так как стек Intel FPU имеет ограничение в **8** записей. Компилятор использует эвристический алгоритм, чтобы определить, сколько переменных необходимо поместить в стек: для видимых процедур это ограничение равно **3**, для невидимых – **1**. Но в случае глубокого дерева вызовов или, что ещё хуже, в рекурсивных процедурах, это всё равно может привести к переполнению стека FPU, поэтому пользователь может указать компилятору, сколько (плавающая точка) переменных должно сохраняться в регистрах.

Директива допускает следующие аргументы:

N

где N это максимальное количество регистров FPU для использования. В текущий момент это число может быть в диапазоне от **0** до **7**.

Normal

восстанавливает эвристическое и обычное поведение.

Default

восстанавливает эвристическое и стандартное поведение

ПРИМЕЧАНИЕ

Эта директива действительна до конца текущей процедуры.

1.2.50. $MESSAGE : Генерировать информационное сообщение

Директива $MESSAGE позволяет вставлять пользовательские предупреждения *(Warning)*, подсказки *(Hint)* или примечания *(Note)*. Она эквивалентна одной из директив $FATAL, $ERROR, $WARNING, $HINT или $NOTE. За ней должно следовать слово ERROR, WARNING, HINT или NOTE, а затем сообщение которое будет отображено, как показано ниже:

```
{$MESSAGE WARNING Это было закодировано в дождливый день, Баг
Кролика}
```

Эта директива выведет предупреждающее сообщение, когда компилятор её встретит. Эффект тот же, что и директивы {$WARNING}.

Это означает, что следующее:

```
{$MESSAGE WARNING Закодировано в дождливый день.}
{$MESSAGE NOTE Закодировано в очень, очень дождливый день.}
{$MESSAGE HINT Закодировано в очень дождливый день}
{$MESSAGE ERROR Это неправильно}
{$MESSAGE FATAL Это настолько неправильно, что компилятор
остановится сразу}
```

полностью эквивалентно

```
{$WARNING Закодировано в дождливый день.}
{$NOTE Закодировано в очень, очень дождливый день.}
{$HINT Закодировано в очень дождливый день}
{$ERROR Это неправильно}
{$FATAL Это настолько неправильно, что компилятор остановится
сразу}
```

Обратите внимание, что если используется переключатель `-Sw`, `WARNING` сообщение будет рассматриваться как ошибка, и компилятор остановится.

1.2.51. $MINENUMSIZE : Указать минимальный размер перечисления

Директива предоставляется для совместимости с Delphi: она имеет тот же эффект, что и директива `$PACKENUM` (см. раздел 1.2.59. $PACKENUM или $Z : Минимальный размер перечисляемого типа [34]).

1.2.52. $MINFPCONSTPREC : Указать точность констант с плавающей точкой

Этот переключатель эквивалентен переключателю командной строки `-CF`. Он устанавливает минимальную точность для констант с плавающей точкой. Этот переключатель можно использовать, чтобы компилятор никогда не снижал точность ниже заданного значения. Поддерживаемые значения **32**, **64** и `DEFAULT`. Значение **80** не поддерживается.

Учтите, что это не имеет ничего общего с реальной точностью, используемой при расчётах, где точность определяется типом используемых переменных. Этот переключатель определяет только с какой точностью будут храниться объявленные константы:

```
{$MINFPCONSTPREC 64}
Const
  MyFloat = 0.5;
```

Все константы будут иметь тип `double`, хотя они могут быть представлена как `single`.

Учтите, что значение **80** *(повышенная точность)* **не поддерживается**.

1.2.53. $MMX : Поддержка MMX (только Intel 80x86)

Free Pascal поддерживает оптимизацию для процессора MMX Intel (см. также раздел 5. ПОДДЕРЖКА INTEL MMX [83]).

Это оптимизирует некоторые части кода для процессора MMX Intel, что существенно повышает скорость. Повышение быстродействия наблюдается в

основном при перемещении больших объёмов данных. Вот всё, что изменяется при использовании этой директивы:

- Данные, размер которых кратен **8** байтам, перемещаются с помощью инструкции ассемблера `movq`, которая перемещает **8** байт за один раз

ПРИМЕЧАНИЕ

Поддержка MMX ***НЕ*** эмулируется на системах без MMX, то есть если процессор не имеет расширения MMX, то оптимизация MMX не может использоваться.

Если поддержка MMX включена, она не позволяет выполнять арифметические операции с плавающей точкой. Она позволяет перемещать данные с плавающей точкой, но не арифметические операции. Если операции с плавающей точкой всё равно должны быть выполнены, то поддержка MMX сначала должна быть отключена и регистры FPU должны быть очищены с помощью функции `emms` модуля cpu.

Следующий пример поможет вам представить это более ясно:

```
Program MMXDemo;
uses mmx;
var
   d1 : double;
   a : array[0..10000] of double;
   i : longint;
begin
   d1:=1.0;
{$mmx+}
{ данные с плавающей точкой используются, но нет арифметики }
   for i:=0 to 10000 do a[i]:=d2; { это 64-разрядное
перемещение }
{$mmx-}
   emms; { очистить fpu }
{ сейчас мы можем выполнять арифметические операции с плавающей
точкой }
…
end.
```

См. также раздел по `MMX` (5. ПОДДЕРЖКА INTEL MMX)[83], чтобы получить более подробную информацию по этой теме.

1.2.54. $NODEFINE : Игнорируется

Эта директива предоставляется для совместимости с Delphi, но игнорируется.

1.2.55. $NOTE : Генерировать примечание

Если включена генерация примечаний с помощью опции командной строки

-vn или директивой {$NOTES ON}, то директива

```
{$NOTE Спросите Санта Клауса, чтобы посмотреть этот код}
```

выведет сообщение с примечанием, когда компилятор обнаружит её.

1.2.56. $NOTES : Выводить примечания

Директива {$NOTES ON} разрешает генерировать примечания. Директива {$NOTES OFF} запрещает генерировать примечания. В отличие от опции командной строки -vn – это локальный переключатель, который может оказаться полезным для проверки участков кода.

По умолчанию {$NOTES} примечания отключены.

1.2.57. $OBJECTCHECKS : Проверять объект

Этот логический переключатель определяет, надо ли проверять, вставленый в метод указатель SELF. По умолчанию он ***выключен***. На пример если есть

```
{$OBJECTCHECKS ON}
```

и если указатель SELF равен NIL, то будет сгенерирована ошибка времени выполнения **210** *(поверка диапазона)*.

Этот переключатель также активируется опцией командной строки -CR.

1.2.58. $OPTIMIZATION : Включить оптимизацию

Этот переключатель включает оптимизацию. Он может иметь следующие возможные значения:

ON
: включает оптимизацию, соответствующую уровню оптимизации **2**.

OFF
: выключает все оптимизации.

DEFAULT
: возврат к оптимизациям по умолчанию *(то есть из командной строки или конфигурационного файла)*.

XYZ
: анализирует строку и включает оптимизации, имеющиеся в строке, как если бы они были переданы с помощью опции командной строки -Oo. Оптимизации должны быть разделены запятыми.

Поддерживаются следующие строки:

LEVEL1
: оптимизация уровня 1

LEVEL2
: оптимизация уровня 2

LEVEL3
оптимизация уровня 3
REGVAR
использовать регистровые переменные.
UNCERTAIN
использовать неточную оптимизацию.
SIZE
оптимизировать по размеру.
STACKFRAME
пропускать границы стека.
PEEPHOLE
оптимизация Peephole.
ASMCSE
использовать общие подвыражения исключения на уровне ассемблера.
LOOPUNROLL
разворачивать циклы.
TAILREC
изменять хвост рекурсии для регулярного `while`
ORDERFIELDS
Изменить порядок полей, если это приводит к лучшему выравниванию.
FASTMATH
Быстрое выполнение математических операций.
REMOVEEMPTYPROCS
удалить вызов пустых процедур.
CSE
использовать общие подвыражения исключения.
DFA
использовать DFA *(Анализ Потока Данных)*.

Например:

```
{$OPTIMIZATION ON}
```

Это эквивалентно директиве

```
{$OPTIMIZATION LEVEL2}
```

Могут быть указаны несколько оптимизаций:

```
{$OPTIMIZATION REGVAR,SIZE,LEVEL2}
```

Этот переключатель также активируется опцией командной строки `-Ooxxx`. Учтите, что маленькая `'o'`: следует за именем переключателя `-Oo`.

1.2.59. $PACKENUM или $Z : Минимальный размер перечисляемого типа

Эта директива указывает компилятору минимальное количество байтов, которое он должен использовать при хранении перечисляемых типов. Это требует следующего формата:

```
{$PACKENUM xxx}
```

```
{$MINENUMSIZE xxx}
```

где формат $MINENUMSIZE используется для совместимости с Delphi. xxx может быть одним из значений **1, 2, 4, NORMAL** или **DEFAULT**.

По умолчанию размер перечисляемых типов определяется режимом компилятора:

- В режимах Delphi и TP размер равен **1**.
- В режиме MacPas размер равен **2**.
- Во всех остальных режимах, размер по умолчанию равен **4**.

В качестве альтернативного формата можно использовать {$Z1}, {$Z2} или {$Z4}. Формат {$Z} принимает логический аргумент, где ON эквивалентно {$Z4}, а OFF эквивалентно {$Z1}.

В следующем коде

```
{$PACKENUM 1}
Type
   Days = (monday, tuesday, wednesday, thursday, friday,
saturday, sunday);
```

Будет использоваться **1** байт для хранения переменной типа Days, хотя в нормальном режиме должно использоваться **4** байта. Описанный выше код эквивалентен следующему:

```
{$Z1}
Type
   Days = (monday, tuesday, wednesday, thursday, friday,
saturday, sunday);
```

или эквивалентен

```
{$Z OFF}
Type
   Days = (monday, tuesday, wednesday, thursday, friday,
saturday, sunday);
```

1.2.60. $PACKRECORDS : Выравнивание элементов записи

Эта директива управляет байтовым выравниванием элементов в записи, объекте или классе.

Она имеет следующий формат:

```
{$PACKRECORDS n}
```

где n – это одно из **1, 2, 4, 8, 16, C, NORMAL** или **DEFAULT**. Это означает, что элементы записи, которые имеют размер больше, чем n, будут выровнены по границе в n байтов. Элементы, размер которых меньше или равен n, будут выровнены по натуральной границе, то есть по степени числа **2**, которая равна или больше, чем размер элемента. Специальное значение C используется для указания выравнивания, как это принято в компиляторе GNU CC. Оно должно использоваться только в том случае, если выполняется импорт модулей для

процедур C.

Выравнивание по умолчанию *(которое можно выбрать с помощью значения **DEFAULT**)* – это естественное выравнивание.

Больше информации и примеры по этой теме можно найти в Справочное руководство Free Pascal, в разделе, описывающем типы записей.

Следующие сокращения могут быть использованы для этой директивы:

```
{$A1 }
{$A2 }
{$A4 }
{$A8 }
```

1.2.61. $PACKSET : Указать размер множества

Директива `$PACKSET` принимает числовой аргумент **1, 2, 4** или **8**. Это число определяет количество байтов, используемых для хранения множества: компилятор округляет количество байтов, необходимое для хранения множества до ближайшего множителя настроек `PACKSET`, за исключением случая **3**-байтного множества, которое всегда округляется до **4** байтов.

Другие возможные значения – это `FIXED`, `DEFAULT` *(или 0)* или `NORMAL`. С этими значениями компилятор хранит множества с менее чем **32** элементами в **4** байтах, а множества с менее чем **256** элементами в **32** байтах.

1.2.62. $POP : Перезаписать настройки компилятора

Директива `$POP` перезаписывает значения всех локальных директив компилятора последними значениями, которые были записаны в стек настроек. То есть берёт настройки из стека и записывает в локальные директивы. После этого последние настройки из стека удаляются.

Настройки могут быть записаны в стек директивой `$PUSH` *(см. раздел 1.2.63. $PUSH : Сохранить настройки компилятора[36])*.

Учтите, что глобальные настройки не перезаписываются этой директивой.

1.2.63. $PUSH : Сохранить настройки компилятора

Директива `$PUSH` сохраняет значения всех локальных директив компилятора, которые были записаны, в стек настроек. До **20** наборов настроек можно записать в стек.

Текущие настройки можно перезаписать из стека, используя директиву `$POP` *(см. раздел 1.2.62. $POP : Перезаписать настройки компилятора[36])*.

Учтите, что глобальные настройки *(такие как пути и т.п.)* не сохраняются

этой директивой.

Стек настроек сохраняется в процессе компиляции модулей, то есть если компилятор начинает компиляцию нового модуля, стек не очищается.

1.2.64. $Q или $OV или $OVERFLOWCHECKS: Проверка переполнения

Директива `{$Q+}` или `{$OV+}` (только режим `MACPAS`) или `{$OVERFLOWCHECKS ON}` включает проверку целочисленного переполнения. Это означает, что компилятор вставляет код для проверки на переполнение, когда выполняются вычисления с целыми числами. Если случается переполнение, то библиотека времени исполнения генерирует ошибку **215**: она печатает сообщение `Overflow at xxx` и завершает программу с кодом ошибки **215**.

ПРИМЕЧАНИЕ

Проверка переполнения выполняется не так, как в Turbo Pascal, потому что все арифметические операции выполняются с 32- или с **64**-разрядными значениями. Кроме того, стандартные процедуры `Inc` и `Dec` проверяются на переполнение в Free Pascal, в то время как в Turbo Pascal они не проверяются.

Используя переключатель `{$Q-}` (или `{$OV-}` в режиме `MACPAS`), можно отключить генерацию кода проверки переполнения.

Генерация кода проверки переполнения может быть также отключена с помощью опции командной строки `-Co` *(см. Справочник пользователя Free Pascal)*.

В Delphi проверка переполнения работает только на уровне процедур. В Free Pascal директива `{$Q }` может применяться на уровне выражений.

1.2.65. $R или $RANGECHECKS : Проверка диапазона

По умолчанию компилятор не генерирует код для проверки диапазонов индексов массивов, перечисляемых типов, поддиапазонов и т.п. Переключатель `{$R+}` указывает компилятору, чтобы он генерировал код для проверки этих индексов. Если во время выполнения программы индекс или перечисляемый тип выходит за пределы указанного диапазона, то генерируется ошибка времени выполнения, а программа завершается с кодом **201**. Это может случиться, когда выполняется преобразование типов *(явное или неявное)* с перечисляемыми типами или поддиапазонами.

Переключатель `{$RANGECHECKS OFF}` указывает компилятору, что ***НЕ*** требуется генерировать код проверки диапазона. Это, возможно, приведёт к

неправильной работе программы, но ошибки времени выполнения не будут генерироваться.

ПРИМЕЧАНИЕ

Стандартные функции `val` и `Read` будут также проверять диапазоны, если их вызов компилируется в режиме `{$R+}`.

В Delphi проверка диапазона работает только на уровне процедур. В Free Pascal директива `{$R }` может применяться на уровне выражений.

1.2.66 $REGION : Отметить начало вложенного региона

Директива `$REGION` оставлена для совместимости с Delphi. В IDE Delphi это служит, чтобы отметить начало вложенного региона.

1.2.67. $R или $RESOURCE : Подключить ресурс

Эта директива включает ресурс в двоичный файл. Аргументом для этой директивы является файл ресурса, например

```
{$R icons.res}
```

Включить файл icons.res как ресурс в двоичный файл. До версии **2.2.N** ресурсы поддерживались только для Windows *(которая использует ресурсы)* и для платформ, использующих ELF (linux, BSD). Начиная с версии **2.3.1**, ресурсы выполняются для всех поддерживаемых платформ.

В asterix могут использоваться заполнители для текущего имени файла модуля/программы

```
unit myunit;
{$R *.res}
```

подключит файл myunit.res.

1.2.68. $SATURATION : Насыщенность операций (только Intel 80x86)

Это работает только на компиляторах intel, при поддержке MMX (`{$MMX +}`) для получения какого-либо эффекта. См. раздел о поддержке насыщенности (раздел 5.2. Поддержка насыщенности [83]), чтобы получить более подробную информацию об этой директиве.

1.2.69 $SAFEFPUEXCEPTIONS Ждаать сохранения значений FPU на Intel x86

Эта логическая директива управляет тем, как компилятор генерирует код для хранения значений FPU. Если установлено значение ON, компилятор вставляет опкод FWAIT после записи значения с плавающей точкой, так что любые ошибки будут видны сразу. Это замедляет код, но гарантирует, что сообщение об ошибке генерируется в том месте, где была выполнена команда.

1.2.70 $SCOPEDENUMS Управление использованием перечисляемого типа

Логическая директива $SCOPEDENUMS управляет тем, как используется перечисляемый тип. По умолчанию *(OFF)*, значения перечисляемого типа используются непосредственно. В состоянии ON, значения будут использоваться с уточнением имени перечисляемого типа.

Практически это означает, следующее поведение по умолчанию:

```
{$SCOPEDENUMS OFF}
Type
  TMyEnum = (one,two,three);

Var
  A : TMyEnum;

begin
  A:=one;
end.
```

Значение можно применять непосредственно. Следущее присвоение выдаст сообщение об ошибке:

```
begin
  A:=TMyEnum.one;
end.
```

Если директива SCOPEDENUMS установлен в состояние ON, то присвоение производится следующим образом:

```
{$SCOPEDENUMS ON}
Type
  TMyEnum = (one,two,three);

Var
  A : TMyEnum;

begin
  A:=TMyEnum.one;
end.
```

т.е. значение должно иметь префикс с именем типа.

1.2.71. $SETC : Определить и присвоить значение идентификатору

В режиме MACPAS эту директиву можно использовать для определения идентификаторов компилятора. Данная директива является альтернативой директиве $DEFINE для макросов. Она поддерживается для совместимости с существующими компиляторами Mac OS Pascal. Директива определяет идентификатор с предустановленным значением *(называется переменное выражение компилятора)*.

Синтаксис выражения похож на синтаксис, используемый в макросах, но выражение должно быть вычислено во время компиляции. Это означает, что только некоторые основные арифметические и логические операции могут использоваться, а также некоторые дополнительные возможности, такие как операторы TRUE, FALSE и UNDEFINED:

```
{$SETC TARGET_CPU_PPC := NOT UNDEFINED CPUPOWERPC}
{$SETC TARGET_CPU_68K := NOT UNDEFINED CPUM68K}
{$SETC TARGET_CPU_X86 := NOT UNDEFINED CPUI386}
{$SETC TARGET_CPU_MIPS := FALSE}
{$SETC TARGET_OS_UNIX := (NOT UNDEFINED UNIX) AND (UNDEFINED
DARWIN)}
```

Символ присваивания := можно заменить символом =.

Учтите, что эта команда работает только в режиме MACPAS, но не зависима от опции командной строки -Sm или директивы {$MACRO }.

1.2.72. $STATIC : Разрешить использование ключевого слова Static

Если вы укажете директиву {$STATIC ON}, то для объектов будут разрешены методы Static. Методы Static объектов не используют переменную Self. Они эквиваленты методам Class для классов. По умолчанию методы Static запрещены. Методы класса разрешены всегда. Учтите, что также могут быть определены статические поля.

Эта директива эквивалентна опции командной строки -St.

1.2.73. $STOP : Генерировать сообщение о фатальной ошибке

Следующий код

```
{$STOP Этот код содержит ошибку!}
```

отобразит сообщение об ошибке, если компилятор обнаружит её. **Компилятор немедленно прекратит процесс компиляции.**

Эта директива имеет тот же эффект, что и директива {$FATAL}.

1.2.74 $STRINGCHECKS : Ignored

Эта директива применяется для совместимости Delphi, но в настоящее время игнорируется. В Delphi, она контролирует генерацию кода, который проверяет здравомыслие строковых переменных и аргументов.

1.2.75. $T или $TYPEDADDRESS : Тип оператора адреса (@)

В состоянии {$T+} или {$TYPEDADDRESS ON}, оператор @, если он имеется в имени переменной, возвращает результат типа ^T, если тип переменной – это T. В состоянии {$T-}, возвращаемый результат – это всегда нетипизированный указатель, который всегда совместим с другими типами указателей.

Например, следующий код не будет компилироваться:

```
{$T+}

Var
   I : Integer;
   P : Pchar;
begin
   P:=@I;
end.
```

Компилятор выдаст ошибку несоответствия типов:

```
testt.pp(8,6) Error: Incompatible types: got "^SmallInt" expec-
ted "PChar"
                (Ошибка: несовместимые типы: получено "^SmallInt",
ожидалось "Pchar")
```

По умолчанию оператор адреса возвращает **не типизированный** указатель.

1.2.76. $UNDEF или $UNDEFC : Разыменовать идентификатор

Директива

```
{$UNDEF имя}
```

переопределяет имя, если оно было определено ранее. Имя нечувствительно к регистру.

В режиме Mac Pascal, $UNDEFC эквивалентно $UNDEF, и предоставляется для совместимости с Mac Pascal.

1.2.77. $V или $VARSTRINGCHECKS : Проверка Var-строки

Директива {$VARSTRINGCHECKS } определяет, насколько строго компилятор будет проверять совместимость строковых типов для строк, переданных в подпрограмму по ссылке. Если эта директива в состоянии + или ON, то

компилятор проверяет, является ли строка, переданная как параметр, точным соответствием типу параметра, объявленного в процедуре.

По умолчанию компилятор предполагает, что все короткие строки являются совместимыми типами. То есть, следующий код будет компилироваться:

```
Procedure MyProcedure(var Arg: String[10]);
   begin
      Writeln('Arg ',Arg);
   end;
Var
   S : String[12];
begin
   S:='123456789012';
   MyProcedure(S);
end.
```

Типы `Arg` и `S` не являются строго совместимыми: параметр `Arg`- это строка длиной **10** символов, а переменная `S` – строка длиной **12** символов. Строка `S` будет усечена до **10** символов.

В состоянии `{$V+}` этот код приведёт к ошибке компиляции:

```
testv.pp(14,16) Error: string types doesn't match, because of
$V+ mode
(Ошибка: строковые типы несовместимы, потому что включен режим
$V+)
```

Учтите, что это имеет эффект только для строк, передаваемых по ссылке, а ***НЕ*** по значению.

1.2.78. $W или $STACKFRAMES : Генерировать кадры стека

Директива `{$W}` – это переключатель, который управляет генерацией кадров стека. Во включенном состоянии переключателя компилятор будет генерировать кадр стека для каждой процедуры или функции.

В выключенном состоянии компилятор будет пропускать генерацию кадров стека, если выполняются следующие условия:

- Процедура не имеет параметров
- Процедура не имеет логических переменных
- Если процедура не является ассемблерной (`assembler`), она не должна иметь блоков `asm ... end`;
- Процедура не является конструктором или деструктором.

Если эти условия выполняются, то кадр стека будет пропущен.

1.2.79. $WAIT : Ожидать нажатия клавиши ENTER

Если компилятор обнаружит директиву

```
{$WAIT}
```

он приостановит компиляцию и продолжит её только после того, как пользователь нажмёт клавишу ENTER. Если включена генерация сообщений, то компилятор отобразит следующее сообщение:

```
Press <return> to continue
Нажмите <return> для проболжения
```

перед тем, как начать ожидание нажатия клавиши.

ПРИМЕЧАНИЕ

Это может помешать процессам автоматической компиляции. Данная функция должна использоваться только во время отладки.

1.2.80 $WARN : Контроль генерации предупреждений

Эта директива позволяет выборочно включить или выключить генерацию предупреждений. Она имеет следующий вид

```
{$WARN IDENTIFIER ON}
{$WARN IDENTIFIER OFF}
{$WARN IDENTIFIER +}
{$WARN IDENTIFIER -}
{$WARN IDENTIFIER ERROR}
```

Значения ON или + включает генерацию предупреждений. Значения OFF или − подавляют генерацию предупреждений. ERROR генерирует предупреждение об ошибке, и компилятор примет это во внимание.

IDENTIFIER - это имя предупреждения. Можно применять следующие имена:

CONSTRUCTING_ABSTRACT
Создание экземпляра класса с абстрактными методами.

IMPLICIT_VARIANTS
Неявные использование модуля variants .

NO_RETVAL
Результат ***функции*** не определён.

SYMBOL_DEPRECATED
Устаревший символ.

SYMBOL_EXPERIMENTAL
Испытывемый символ

SYMBOL_LIBRARY
Не используется.

SYMBOL_PLATFORM
Символ зависимый от платформы.

SYMBOL_UNIMPLEMENTED
Нереализованный символ.

UNIT_DEPRECATED
Устаревший модуль.

UNIT_EXPERIMENTAL
Итпытываемый модуль.
UNIT_LIBRARY
UNIT_PLATFORM
Модуль зависимый от платформы.
UNIT_UNIMPLEMENTED
Нереализованный модуль.
ZERO_NIL_COMPAT
Преобразует **0** в **NIL**.
IMPLICIT_STRING_CAST
Неявное преобразование строковых типов.
IMPLICIT_STRING_CAST_LOSS
Неявное приведение строк с потенциальной потерей данных от "$1" до "$2".
EXPLICIT_STRING_CAST
Явное преобразование строковых типов.
EXPLICIT_STRING_CAST_LOSS
Явное приведение строк с потенциальной потерей данных от "$1" до "$2".
CVT_NARROWING_STRING_LOST
Преобразование Unicode констант с потенциальной потерей данных.

Кроме указанных выше текстовых идентификаторов, может быть использован и номер сообщения. Номер сообщения, который отображаются при использовании опции командной строки `-vq`.

1.2.81. $WARNING : Генерировать предупреждение

Если генерация предупреждений включена с помощью опции командной строки `-vw` или директивы `{$WARNINGS ON}`, то директива

```
{$WARNING Это сомнительный код}
```

выведет на экран указанное предупреждение там, где компилятор обнаружит эту директиву.

1.2.82. $WARNINGS : Выводить предупреждения

Директива `{$WARNINGS ON}` включает вывод предупреждений. Директива `{$WARNINGS OFF}` отключает вывод предупреждений. В отличие от опции командной строки `-vw` данная директива является локальной, что может оказаться полезным для проверки участков вашего кода.

По умолчанию предупреждения не выводятся.

1.2.83. $Z1, $Z2 и $Z4

Эти переключатели эквиваленты некоторым параметрам директивы `{$PACKENUM }` (см. раздел 1.2.59. $PACKENUM или $Z : Минимальный размер

перечисляемого типа[34]).

1.3. Глобальные директивы

Глобальные директивы влияют на весь процесс компиляции *(`program` (прогрммы), `unit` (модуля), `library` (библиотеки))*. Поэтому для каждой глобальной директивы также имеется свой параметр командной строки. ***Директивы*** должны быть указаны ***перед*** ключевыми словами `unit`, `program` или `library` в исходном файле, **иначе они не будут иметь никакого эффекта**.

1.3.1. $APPID : Указать ID приложения

Используется только на OC PALM. Можно указать имя приложения, которое можно посмотреть только на Palm. Эта директива имеет смысл только в исходных файлах программы, но не в модулях.

```
{$APPID MyApplication}
```

1.3.2. $APPNAME : Указать имя приложения

Используется только с OC PALM. Для указанного приложения можно установить имя, которое можно увидеть только в OC Palm. Эта директива работает только в исходных файлах программ, но не в модулях

```
{$APPNAME Моё приложение, откомпилировано с использованием Free Pascal.}
```

1.3.3. $APPTYPE : Указать тип приложения

В текущий момент эта директива поддерживается только для следующих целевых платформ: Win32, Mac, OS2 и AmigaOS. На других целевых платформах эта директива игнорируется.

Директива `{$APPTYPE XXX}` принимает один аргумент, который указывает, какого типа приложение нужно компилировать. Этот аргумент может принимать следующие значения:

CONSOLE	Консольное приложение. Будет создан терминал, а дескрипторы файлов стандартного ввода-вывода и ошибок будут инициализированы. В Windows будет создано окно терминала. Это значение по умолчанию. Учтите, то в Mac OS такие приложения не могут принимать параметры командной строки и возвращать код результата. Они запускаются в специальном терминальном окне,

	выполняемом как SIOWapplication, подробности см. в документации MPW. На OS/2 эти приложения могут запускаться как в полноэкранном, так и в терминальном окнах. Приложения LINUX – это всегда консольные приложения. Однако само приложение может принять решение о закрытии стандартных файлов.
FS	Задаёт полный экран приложения VIO на OS/2. Эти приложения используют специальные BIOS-подобные API для программирования экрана. OS/2 запускает эти приложения всегда в полноэкранном режиме.
GUI	Директива {$APPTYPE GUI} помечает приложение как графическое. Консольное окно не будет открыто при запуске приложения. Стандартные файловые дескрипторы не будут инициализированы, их использование (например, использование оператора writeln) приведёт к ошибке во время выполнения. При запуске из командной строки произойдёт немедленный возврат в командную строку после запуска приложения. На OS/2 и Mac OS приложения типа GUI создают GUI-приложение как на Windows. На OS/2 это реальное приложение Presentation Manager.
TOOL	Это специальная директива для Mac OS. Она указывает компилятору, что требуется создать приложение-инструмент, которое инициализирует файлы Input, Output и StdErr и может принимать параметры и возвращать код результата. Оно выполняется как инструмент MPW, который может запускаться только с помощью MPW или ToolServer.

Следует соблюдать осторожность при компиляции приложений с графическим интерфейсом (GUI-приложений). Файлы Input и Output не доступны в GUI-приложении, и попытка чтения или записи этих файлов приведёт к ошибке времени выполнения

Можно определить тип приложения WINDOWS или AMIGA во время выполнения. Константа IsConsole, объявленная в Win32 и Amiga системных модулях как

```
Const
IsConsole : Boolean;
```

Имеет значение True, если приложение является консольным, и False, если это GUI-приложение.

1.3.4. $CODEPAGE : Установить кодовую страницу

Этот переключатель задаёт кодовую страницу, не изменяя исходный файл. Кодовая страница применима только к текстовым строкам, фактически код должен быть в US-ASCII. Аргументом для этого переключателя является имя кодовой страницы, которую требуется использовать.

```
{$CODEPAGE UTF8}
```

Кодовая страница «UTF-8» может быть указана как «UTF-8» или «UTF8». Список поддерживаемых кодовых страниц – это список поддерживаемых кодовых страниц в модуле charset библиотеки RTL.

1.3.5. $COPYRIGHT: Указать сведения об авторских правах

Эта директива предназначена для версии компилятора NETWARE: она указывает информацию об авторских правах, которая может быть просмотрена в модуле для ОС Netware OS.

Например:

```
{$COPYRIGHT GNU copyleft. compiled using Free Pascal}
```

1.3.6. $D или $DEBUGINFO : Отладочные символы

Если этот переключатель включен, то компилятор вставляет отладочную информацию GNU в исполняемый файл. Действие этого переключателя аналогично опции командной строки `-g`.

По умолчанию вставка отладочной информации отключена.

1.3.7. $DESCRIPTION : Описание приложения

Этот переключатель распознаётся только для совместимости, но полностью игнорируется компилятором. В будущих версиях этот переключатель может быть задействован.

1.3.8. $E : Эмуляция сопроцессора

Эта директива управляет эмуляцией сопроцессора. У этой директивы не аналога для командной строки.

Intel 80x86 version

Если этот переключатель включен, то все инструкции с плавающей точкой, которые не поддерживаются стандартными эмуляторами сопроцессора, будут выдавать предупреждения.

Компилятор сам не выполняет эмуляцию сопроцессора.

Для использования эмуляции сопроцессора под DOS *(go32v2)* вы должны использовать модуль emu387, который содержит правильный код инициализации для эмулятора.

Под LINUX и другими UNIX-ами, ядро заботится о поддержке сопроцессора, поэтому данный переключатель нет необходимости использовать на этих платформах.

Motorola 680x0 version

Если этот переключатель включен, никакие коды операций с плавающей точкой не создаются генератором кода. Вместо этого процедуры встроенной библиотеки реального времени вызываются для необходимых вычислений. В этом случае все вещественные типы привязаны к одному типу IEEE с плавающей точкой.

ПРИМЕЧАНИЕ

По умолчанию эмуляция включена, если выполняется компиляция ***НЕ*** для unix-платформ. Для unix-платформ операции с плавающей точкой обрабатываются операционной системой, поэтому по умолчанию этот переключатель выключен.

1.3.9 $EXTENSION : Расширение генерируемого двоичного файла.

Эта директива может появиться в исходном файле программы или библиотеки: директива устанавливает расширение генерируемого двоичного файли или библиотеки. Это не исполняемый файл, то есть нет точки *(связывается с расширением)* входа.

Пример:

```
{$EXTENSTION 'cgi'}
```

установит расширение 'CGI'.

1.3.10 $FRAMEWORKPATH : Путь к файлам среды.

Этот параметр служит для задания пути поиска в среде разработки Darwin, компилятор будет там искать файлы. Директива

```
{$FRAMEWORKPATH XXX}
```

использует XXX в качестве пути к файлам среды разработки. XXX может содержать один или несколько путей, разделенных точкой с запятой или двоеточием.

1.3.11. $G : Генерировать код 80286

Эта опция имеется для совместимости с Turbo Pascal, но игнорируется, так как компилятор работает с **32**-х и **64**-х разрядными процессорами.

1.3.12 $IMAGEBASE : Указание начального адреса в DLL

Этот параметр используется для задания местоположения начальной точки отсчёта в библиотеке DLL в системах на Windows. Директива требует положение в памяти в качестве опции *(для задания шестнадцатеричного значения используется символ **$**)*. Это эквивалент опции командной строки `-WB`.

Ниже задаётся базовый адрес *(место)* **0x00400000**

```
{$IMAGEBASE $00400000}
```

1.3.13. $INCLUDEPATH : Указать путь подключений

Эта опция служит для указания пути подключаемых файлов, где компилятор ищет эти файлы. Используется как

```
{$INCLUDEPATH XXX}
```

где `XXX` – путь для подключаемых файлов. Значение `XXX` может содержать один или более путей, разделённых точкой с запятой или двоеточием.

Например:

```
{$INCLUDEPATH ../inc;../i386}
{$I strings.inc}
```

добавит директории **`../inc`** и **`../i386`** в путь подключений компилятора. Компилятор будет искать файл strings.inc в обеих этих директориях, и подключит первый найденный файл. Эта директива эквивалентна переключателю командной строки `-Fi`.

Будьте внимательны при использовании этой директивы: если вы распространяете файлы, то размещение подключаемых файлов на компьютере пользователя может быть не таким, как на вашем компьютере. Кроме того, может отличаться структура каталогов. В общем случае вы должны избегать использовать ***абсолютные*** пути. Вместо этого следует использовать ***относительные*** пути, как показано выше в примере.

1.3.14. $L или $LOCALSYMBOLS : Локальная символьная информация

Этот переключатель *(не путать с локальной директивой компоновщика `{$L file}`)* распознаётся для совместимости с Turbo Pascal, но игнорируется.

Генерация символьной информации управляется переключателем[47] $D.

1.3.15 $LIBPREFIX : Задать имя файла библиотеки

Директива LIBPREFIX аналогична, директивам компилятора {$EXTENSION } и {$LIBSUFFIX }: Она задает ***префикс*** библиотеки. По умолчанию это 'lib' на Unix, и ничего на Windows. Имя после директивы добавляется к имя файла библиотеки.

Пример:

```
library tl;
{$LIBPREFIX 'library'}

begin
end.
```

сгенерирует файл с именем librarytl.so на Linux, или librarytl.dll на Windows.

1.3.16. $LIBRARYPATH : Указать путь библиотек

Эта опция служит указания пути библиотек, где компоновщик ищет статические или динамические библиотеки. Директива {$LIBRARYPATH XXX} добавляет XXX в путь библиотек. XXX может содержать один или более путей, разделённых точкой с запятой или двоеточием.

Например:

```
{$LIBRARYPATH /usr/X11/lib;/usr/local/lib}
{$LINKLIB X11}
```

добавит директории **/usr/X11/lib** и **/usr/local/lib** в путь библиотек компоновщика. Компоновщик будет искать библиотеку libX11.so в обеих директориях, и использует первый найденный файл. Эта директива эквивалента переключателю командной строки -Fl.

Будьте внимательны при использовании этой директивы: если вы распространяете файлы, то размещение подключаемых файлов на компьютере пользователя может быть не таким, как на вашем компьютере. Кроме того, может отличаться структура каталогов. В общем случае вы должны избегать использовать эту директиву. Если вы не уверены, то лучше использовать переменные makefiles и makefile.

1.3.17 $LIBSUFFIX : Задать суффикс библиотеки

Директива LIBSUFFIX аналогична, директивам компилятора {$EXTENSION } и {$LIBSUFFIX }: Она задает ***суффикс*** библиотеки. Она, как правило, используется для номера версии: суффикс добавляется к ***OutputFileName***, перед расширением.

Пример:

```
library tl;
{$LIBSUFFIX '-1.2.3'}

begin
end.
```

сгенерирует файл с именем libtl-1.2.3.so на Linux, или tl-1.2.3.dll на Windows.

1.3.18 $MAXSTACKSIZE : Установить максимальный размер стека

Директива {$MAXSTACKSIZE} устанавливает максимальный размер стека для исполняемого файла в системе на базе Windows. Аргумент директивы используется как размер стека *(в байтах)*. Максимальное значение будет **$7FFFFFFF**, минимальное значение равно **2048** или $MINSTACKSIZE *(если он был указан)*.

Следующий пример устанавливает максимальный размер стека **$FFFFFF** байт:

```
{$MAXSTACKSIZE $FFFFFF}
```

1.3.19. $M или $MEMORY : Размер памяти

Этот переключатель может использоваться установки размеров кучи и стека. Формат переключателя:

```
{$M РазмерСтека, РазмерКучи}
```

где РазмерСтека и РазмерКучи должны быть двумя целыми числами более **1024**. Первое число устанавливает размер стека, второе – размер кучи. Установка размера стека игнорируется на Unix-платформах, если включена функция проверки стека: в этом случае код проверки стека будет использовать размер, установленный здесь как максимальный размер стека.

На других системах, в добавок к установленному здесь размеру стека, операционная система или среда окружения могут установить другой *(возможно, более строгий)* лимит на размер стека, используя вызовы операционной системы.

Эти два числа можно установить, используя опции командной строки -Ch и -Cs.

1.3.20 $MINSTACKSIZE : Установить минимальный размер стека

Директива {$MINSTACKSIZE} устанавливает минимальный размер стека для исполняемого файла в системах на базе Windows. Аргумент директивы используется как размер стека *(в байтах)*. Он должен быть больше, чем **1024**.

В следующем примере устанавливается минимальный размер стека **2048** байт:

```
{$MINSTACKSIZE 2048}
```

1.3.21. $MODE : Установить режим совместимости компилятора

Директива `{$MODE}` устанавливает режимы совместимости компилятора. Она эквивалентна одной из опций командной строки `-So`, `-Sd`, `-Sp` или `-S2`. Она имеет следующие аргументы:

Default	Режим по умолчанию. Возвращает обратно в режим, который был задан в командной строке.
Delphi	Режим совместимости с Delphi. Включает все расширения языка Object Pascal. Выполняет те же действия, что и опция командной строки `-Sd`. Учтите, что это также подразумевает применение директивы `{$H ON}` *(то есть в режиме Delphi строки `ansistrings` установлены по умолчанию)*.
TP	Режим совместимости Turbo Pascal. Расширения Object Pascal отключены, кроме ansistrings, которое остаётся в силе. Выполняет те же действия, что и опция командной строки `-So`.
FPC	Режим FPC. Это режим по умолчанию, если не использована соответствующая опция командной строки.
OBJFPC	Режим Object Pascal. Выполняет те же действия, что и опция командной строки `-S2`.
MACPAS	Режим `MACPAS`. В этом режиме компилятор пытается быть более совместимым с широко используемыми диалектами Pascal на операционных системах Mac, такими как Think Pascal, Metrowerks Pascal, MPW Pascal.
ISO	Стандартный режим Pascal *(ISO 7185)*. В этом режиме компилятор соответствует требованиям уровня 0 и 1 стандарта ISO/IEC 7185.

Точное описание каждого из этих режимов имеется в приложении ПРИЛОЖЕНИЕ D: РЕЖИМЫ КОМПИЛЯТОРА[177].

1.3.22. $MODESWITCH : Выбор функций режима

Как в FPC **2.3.1**, директива `{$MODESWITCH}` выбирает некоторые функции режима, который был выбран директивой `{$MODE }`.

Она может быть использована для подключения функций, которые недоступны в текущем режиме. Например, вы хотите программировать в режиме TP, но в

то же время хотите использовать параметр 'Out', который доступен только в режиме Delphi.

Директива {$MODESWITCH } позволяет активировать или деактивировать некоторые специфические функции режима, не изменяя текущий режим компилятора.

Это глобальный переключатель, поэтому его можно использовать везде, где применяется переключатель {$MODE}.

Синтаксис следующий:

```
{$MODESWITCH XXX}
{$MODESWITCH XXX+}
{$MODESWITCH XXX-}
```

Первые два случая включают функцию XXX, последний – выключает. Функция XXX может быть одним из следующих значений:

CLASS	Использовать классы Object Pascal.
OBJPAS	Автоматически подключать модуль ObjPas.
RESULT	Включить идентификатор Result для результата в функциях.
PCHARTOSTRING	Разрешить автоматическое преобразование строк с нулевым окончанием в строки Pascal.
CVAR	Разрешить использование ключевого слова CVAR.
NESTEDCOMMENTS	Разрешить использование вложенных комментариев.
CLASSICPROCVARS	Использовать классические процедурные переменные.
MACPROCVARS	Использовать процедурные переменные стиля MAC.
REPEATFORWARD	Объявления Implementation и Forward должны полностью совпадать.
POINTERTOPROCVAR	Разрешить «тихое» преобразование указателей в процедурные переменные.
AUTODEREF	Автоматическое *(тихое)* разыменование типизированных указателей.
INITFINAL	Разрешить использование Initialization и Finalization.
POINTERARITHMETICS	Разрешить использование арифметических операций с указателями.

ANSISTRINGS	Разрешить использование `ansistrings`.
OUT	Разрешить использовать параметр типа `out`.
DEFAULTPARAMETERS	Разрешить использовать по умолчанию значения параметров.
HINTDIRECTIVE	Поддерживать директивы-подсказки *(`deprecated`, `platform` и т.п.)*
DUPLICATELOCALS	Разрешить локальным переменным методов классы иметь такие же имена, что и имена свойств класса.
PROPERTIES	Разрешить использовать глобальные свойства.
ALLOWINLINE	Разрешить встроенные процедуры.
EXCEPTIONS	Разрешить использовать исключения.
ADVANCEDRECORDS	Разрешить использование расширенных записей *(т.е. записей с методами)*.
UNICODESTRINGS	Тип **string** по умолчанию является строкой unicode.
TYPEHELPERS	Разрешить использование помощников типа.
CBLOCKS	C тип блоков.
ISOIO	Ввод/вывод в соответствии с требованиями ISO pascal.
ISOPROGRAMPARAS	Параметры программы согласно требованиям ISO pascal.
ISOMOD	Операция **mod** согласно требованиям ISO pascal.
ISOUNARYMINUS	Унарный минус, как требуется ISO pascal.

Таким образом, следующий пример:

```
{$MODE TP}
{$MODESWITCH OUT}
```

Включит поддержку параметра `out` в режиме TP. Это эквивалентно следующей записи:

```
{$MODE TP}
{$MODESWITCH OUT+}
```

1.3.23. $N : Цифровая обработка

Этот переключатель распознаётся для совместимости с Turbo Pascal, но игнорируется, потому что компилятор всегда использует сопроцессор для

математики с плавающей точкой.

1.3.24. $O : Второй уровень оптимизации

В более ранних версиях FPC этот переключатель распознавался для совместимости с Turbo Pascal, но игнорировался: концепция перекрытия кода не актуальна в **32**-х и **64**-х разрядных программах.

В новых версиях FPC (начиная с **2.0.0**) это переключатель стал совместимым с Delphi: он имеет то же значение, что и переключатель `{$OPTMIZATIONS ON/OFF}`, который включает/выключает второй уровень оптимизации.

См. раздел 1.2.58. $OPTIMIZATION : Включить оптимизацию[33], где можно найти более подробные объяснения и описание настроек оптимизации.

1.3.25. $OBJECTPATH : Указать пути для объектных файлов

Эта опция служит для указания пути к объектным файлам, по которым компилятор будет искать эти файлы. Директива `{$OBJECTPATH XXX}` добавляет `XXX` в путь к объектным файлам. Значение `XXX` может содержать один или более путей, разделённых точкой с запятой или двоеточием. Например:

```
{$OBJECTPATH ../inc;../i386}
{$L strings.o}
```

добавит директории **`../inc`** и **`../i386`** в пути объектных файлов компилятора. Компилятор будет искать файл strings.o в обеих этих директориях, и выполнит компоновку первого найденного файла. Эта директива эквивалентна переключателю командной строки `-Fo`.

Будьте внимательны при использовании этой директивы: если вы распространяете файлы, то размещение объектных файлов на компьютере пользователя может быть не таким, как на вашем компьютере. Кроме того, может отличаться структура каталогов. В общем случае вы должны избегать использовать ***абсолютные*** пути. Вместо этого следует использовать ***относительные*** пути, как показано выше в примере. Используйте эту директиву только в том случае, если вы хотите указать точные пути к объектным файлам. Если вы не уверены, то лучше использовать переменные `makefiles` и `makefile`.

1.3.26. $P или $OPENSTRINGS : Использовать открытые строки

Если этот переключатель включен, то все строковые параметры процедур и функций считаются параметрами открытой строки. Этот переключатель имеет эффект только с короткими строками, но не с **ansistrings**.

Если используются открытые строки, то объявленный тип строки может отличаться от типа строки, которая передаётся по ссылке. Объявленный размер передаваемой строки можно получить с помощью функции `High(P)`.

По умолчанию использование открытых строк отключено.

1.3.27. $PASCALMAINNAME : Установить имя точки ввода

Директива `{$PASCALMAINNAME NNN}` устанавливает имя точки ввода ассемблерного идентификатора для программы или библиотеки в `NNN`. Эта директива эквивалентна опции командной строки `-XM`.

При нормальных обстоятельствах нет необходимости использовать эту директиву.

1.3.28. $PIC : Генерировать код PIC

Директива `{$PIC }` принимает логический аргумент и указывает компилятору, надо или нет генерировать код PIC *(Position Indepedent Code)*. Эта директива эквивалентна переключателю командной строки `-Cg`.

Эта директива полезна только для платформ Unix: модули должны компилироваться с использованием кода PIC, если они должны быть в библиотеке. Для программ использовать код PIC нет необходимости, но это возможно *(хотя работать они будут с кодом PIC медленнее)*.

В следующем участке кода

```
{$PIC ON}
unit MyUnit;
```

компилятору дана команда компилировать модуль `myunit`, используя код PIC.

1.3.29 $POINTERMATH : Разрешить использование математики с указателями

Это логическая директива разрешает или запрещает использование указателей в арифметических выражениях *(связанных с указателями)*. Если директива включена, то можно вычесть два указателя, или прибавить целое значение к указателю. По умолчанию `POINTERMATH` включена.

В следующий примере

```
{$POINTERMATH OFF}
unit MyUnit;
```

компилятор выдаст ошибку каждый раз, когда указатель появляется в математическом выражении.

1.3.30. $PROFILE : Профилирование

Эта директива включает генерацию профилирующего кода *(или выключает)*. Она эквивалента опции командной строки `-gp`. По умолчанию профилирование `OFF` *(выключено)*. Эта директива работает только с исходными кодами программ, но не с модулями.

1.3.31. $S : Проверка стека

Директива `{$S+}` указывает компилятору, что нужно генерировать код проверки стека. Она создаёт код для проверки переполнения стека, то есть код, который проверяет, не превысил ли стек максимально допустимый размер. Если стек превысил максимально допустимый размер, то генерируется ошибка времени выполнения, а программа будет завершена с кодом **202**.

Директива `{$S-}` отключает генерацию кода проверки стека.

Переключатель командной строки `-Ct` имеет тот же эффект, что и директива `{$S+}`.

По умолчанию проверка стека не выполняется.

ПРИМЕЧАНИЕ

Проверка стека предоставляется только для использования во время отладки, чтобы попытаться отследить процедуры, которые используют слишком много локальной памяти. Она не предназначена и не может быть использована для безопасной обработки таких ошибок. Неважно, является ли это обработкой исключений или чем-то иным.

Если произошла ошибка стека, это является фатальной ошибкой и не может выполняться корректно, независимо от того, работает оно в реальной среде или в процессе отладки.

1.3.32. $SCREENNAME : Указать имя экрана

Эту директиву можно использовать для целевой платформы Novell Netware, чтобы указать имя экрана. В качестве аргумента принимается имя экрана, которое будет использоваться.

```
{$SCREENNAME Мой Экран}
```

В этом примере имя экрана текущего приложения будет установлено как «Мой Экран».

1.3.33 $SETPEFLAGS : Задать флаг PE для исполняемых файлов

Директива `$SETPEFLAGS` устанавливает флаг PE в исполняемом файле Windows. Флаг PE записываются в заголовке двоичного файла. В качестве аргумента ожидается числовая константа, она будет записана в блок заголовка файла.

По умолчанию, компилятор сам определит значение флага PE для записи в двоичный файл, однако эта директива может использоваться, чтобы изменить поведение компилятора.

1.3.34. $SMARTLINK : Использовать «умную компоновку»

Модуль, который компилируется в состоянии `{$SMARTLINK ON}`, будет откомпилирован таким образом, чтобы его можно было использовать для **«умной компоновки»**. Это означает, что модуль будет разделён логически на части: каждая процедура будет помещена в собственный объектный файл, а все объектные файлы будут помещены вместе в большой архив. Если использовать такой модуль в программе, то только те части кода, которые вам реально необходимы *(то есть только те процедуры, которые вызываются из вашей программы)* будут скомпонованы в вашу программу, что существенно уменьшит размер исполняемого файла.

Будьте внимательны: использование **«умной компоновки»** модулей замедляет процесс компиляции, потому что отдельные объектные файлы должны создаваться для каждой процедуры. Если вы имеете модули с большим количеством функций и процедур, это может быть трудоёмким процессом, тем более, если вы используете внешний ассемблер *(ассемблер вызывается для ассемблирования каждой отдельного блока кода процедуры или функции)*.

Директива **«умной компоновки»** должна быть указана ***перед*** объявлением модуля:

```
{$SMARTLINK ON}
Unit MyUnit;
Interface
…
```

Эта директива эквивалентна переключателю командной строки `-CX`.

1.3.35 $SYSCALLS : Select system calling convention on Amiga/MorphOS

Эта директива устанавливает систему соглашение о вызовах для использования на системе MorphOS или Amiga. Директиве необходим один из следующих аргументов:

LEGACY

SYSV
SYSVBASE
BASESYSV
R12BASE

Директива будет генерировать предупреждение, если используется на другом компьютере.

1.3.36. $THREADNAME : Установить имя потока в Netware

Эта директива может быть использована для указания имени потока при компиляции для Netware.

1.3.37. $UNITPATH : Указать путь модулей

Эта опция служит для указания пути к файлам модулей, по которым компилятор будет искать эти файлы. Директива `{$UNITPATH XXX}` добавляет `XXX` в путь к файлам модулей. Значение `XXX` может содержать один или более путей, разделённых точкой с запятой или двоеточием.

Например:

```
{$UNITPATH ../units;../i386/units}
Uses strings;
```

добавит директории **`../units`** и **`../i386/units`** в пути файлов модулей компилятора. Компилятор будет искать файл strings.ppu в обеих этих директориях, и подключит первый найденный файл к программе. Эта директива эквивалентна переключателю командной строки `-Fu`.

Будьте внимательны при использовании этой директивы: если вы распространяете файлы, то размещение файлов модулей на компьютере пользователя может быть не таким, как на вашем компьютере. Кроме того, может отличаться структура каталогов. В общем случае вы должны избегать использовать ***абсолютные*** пути. Вместо этого следует использовать ***относительные*** пути, как показано выше в примере. Используйте эту директиву только в том случае, если вы хотите указать точные пути к объектным файлам. Если вы не уверены, то лучше использовать переменные `makefiles` и `makefile`.

Надо учесть, что действие переключателя не распространяется на другие модули *(то есть его действие ограничено текущим модулем)*.

1.3.38 $VARPROPSETTER : Разрешить использование var/out/const параметров для установки свойств

Эта логическая директива предназначена для импорта COM-интерфейсов. Иногда COM интерфейсы имеют уставщики свойств, принимающие аргументы,

не по значению, а по ссылке. Использование этих установщиков, как правило, запрещено. Эта директива позволяет использовать установщики свойств с аргументами `var`, `const`, `out`. По умолчанию она выключена. Эффективна при объявлении интерфейсов, но не для определения классов.

Следующий пример компилируется ***только*** в состоянии `ON`:

```
{$VARPROPSETTER ON}
Type
  TMyInterface = Interface
    Procedure SetP(Var AValue : Integer);
    Function GetP : Integer;
    Property MyP : Integer Read GetP Write SetP;
  end;
```

В выключенном *(OFF)* состоянии, будет сгенерировано сообщение об ошибке:

```
testvp.pp(7,48) Error: Illegal symbol for property access
testvp.pp (7,48) Ошибка: Недопустимый символ для доступа к
свойству
```

1.3.39. $VERSION : Указать версию DLL

На WINDOWS это можно использовать, чтобы указать номер версии для библиотеки. Этот номер версии будет использоваться при установке библиотеки, и может быть просмотрен в Windows Explorer при открытии окна свойств библиотеки DLL на вкладке «Версия». Номер версии содержит не менее **1** и не более **3** чисел:

```
{$VERSION 1}
```

или:

```
{$VERSION 1.1}
```

и даже:

```
{$VERSION 1.1.1}
```

Эта директива не может использоваться для исполняемых файлов в Windows, но может использоваться в будущем.

1.3.40. $WEAKPACKAGEUNIT : Игнорируется

Этот переключатель предоставляется для совместимости с Delphi, но игнорируется. Компилятор выдаст предупреждение при обнаружении этой директивы.

1.3.41. $X или $EXTENDEDSYNTAX : Расширенный синтаксис

Расширенный синтаксис позволяет вам пропустить результат функции. Это означает, что вы можете использовать вызов функции, как если бы это была

процедура. По умолчанию эта опция включена. Вы можете её отключить, используя директиву {$X-} или {$EXTENDEDSYNTAX OFF}.

Например, следующий код будет компилироваться:

```
function Func (var Arg : sometype) : longint;
begin
… { объявление функции }
end;
…
{$X-}
Func (A);
```

Эта конструкция поддерживается по той причине, что вы можете вызвать функцию в каких-то случаях, и при этом вам не нужен будет результат функции. В таком случае вам нет надобности присваивать результат функции внешней переменной.

Переключатель командной строки -Sa1 имеет тот же эффект, что и директива {$X+}.

По умолчанию предполагается расширенный синтаксис.

1.3.42. $Y или $REFERENCEINFO : Вставить информацию обозревателя

Этот переключатель управляет генерацией информацией браузера. Он распознаётся только для совместимости с Turbo Pascal и Delphi, но генерация информации браузера пока не поддерживается в полной мере.

2. ИСПОЛЬЗОВАНИЕ УСЛОВНЫХ ОПЕРАТОРОВ, СООБЩЕНИЙ И МАКРОСОВ

Компилятор Free Pascal поддерживает условные операторы, как и Turbo Pascal, Delphi или Mac OS Pascal. Однако это более развито. Он позволяет вам создавать макросы, которые можно использовать в вашем коде, а также определять сообщения или ошибки, которые могут отображаться во время компиляции. Он также поддерживает переменные и выражения времени компиляции, как это в общем случае делают компиляторы Mac OS.

Различные условные директивы компиляции `$IF`, `$IFDEF`, `$IFOPT` используются в комбинации с `$DEFINE` для предоставления программисту выбора во время компиляции, какие участки кода должны компилироваться. Это может использоваться, например, в следующих случаях:

- Для выбора реализации для одной или другой операционной системы
- Для выбора демо-версии или полной версии
- Для выбора между отладочной версией и версией для продажи

Эти опции затем можно выбрать при компиляции программы, включая или исключая необходимые части кода. В отличие от использования обычных переменных, компиляция с выбором участков кода включает в исполняемый файл только тот код, который необходим для конкретного случая.

2.1. Условные операторы

Правила использования условных операторов такие же, как в Turbo Pascal или Delphi. Объявление идентификатора должно быть следующим:

```
{$define Symbol}
```

С этого момента в вашем коде компилятору известен идентификатор `Symbol`. Также как и в `Pascal`, идентификаторы не чувствительны к регистру символов.

Вы можете также определить идентификатор в командной строке. Опция `-dSymbol` определяет идентификатор `Symbol`. Вы можете указать множество идентификаторов в командной строке.

Отменить объявление существующего идентификатора можно следующим образом:

```
{$undef Symbol}
```

Если идентификатор не существует, то эта директива ничего не делает. Если идентификатор предварительно объявлен, то идентификатор будет удалён, и не будет больше распознаваться в коде, который последует за оператором `{$undef …}`.

Вы можете также удалить объявленный идентификатор с помощью

переключателя командной строки -u.

Для использования условной компиляции, в зависимости от того, объявлен идентификатор или нет, вы можете заключить участок кода в пару операторов {$ifdef Symbol} … {$endif}.

Например, следующий коде никогда не будет компилироваться:

```
{$undef MySymbol}
{$ifdef Mysymbol}
DoSomething;
…
{$endif}
```

Аналогичным образом вы можете заключить участок кода в пару операторов {$ifndef Symbol} … {$endif}. Тогда код, находящийся между этой парой операторов, будет компилироваться только в том случае, если используемый идентификатор **не существует**. Например, в следующем коде вызов DoSomething будет всегда компилироваться:

```
{$undef MySymbol}
{$ifndef Mysymbol}
DoSomething;
…
{$endif}
```

Вы можете совместить две альтернативы в одной структуре, например:

```
{$ifdef Mysymbol}
DoSomething;
{$else}
DoSomethingElse
{$endif}
```

В этом примере, если MySymbol существует, то вызов DoSomething будет компилироваться. Если не существует, то будет компилироваться вызов DoSomethingElse.

2.1.1. Предопределённые идентификаторы

Компилятор Free Pascal определяет некоторые идентификаторы перед началом компиляции вашей программы или модуля. Вы можете использовать эти идентификаторы для определения вида компилятора и версий компилятора. Список предопределённых идентификаторов можно найти в приложении ПРИЛОЖЕНИЕ G: ОПРЕДЕЛЕНИЯ КОМПИЛЯТОРА ВО ВРЕМЯ КОМПИЛЯЦИИ[204].

ПРИМЕЧАНИЕ

Идентификаторы, даже если они определены в интерфейсной части модуля, недоступны за пределами этого модуля.

2.2. Макросы

Синтаксис макросов поход на синтаксис идентификаторов *(которые через* `$define` *объявляются)* или переменных. Разница в том, что макрос имеет какое-либо значение, тогда как идентификатор может быть определён или нет *(для использования в* `$ifdef`*)*.

Кроме того, после объявления макроса, любое появление макроса в исходном коде Pascal будет заменено значением этого макроса *(подобно тому, как выполняется поддержка макросов в препроцессоре* C *)*. Если требуется поддержка макросов, то должен быть использован переключатель командной строки `-Sm` *(должен быть включен)*, или вместо него можно использовать директиву:

```
{$MACRO ON}
```

В противном случае макрос будет рассматриваться как идентификатор.

Определение макроса в программе выполняется также, как определение идентификатора: в операторе препроцессора **`{$define}`**.

```
{$define ident:=expr}
```

Учтите, что в компиляторах версии до ***0.9.8*** *оператор присваивания для макросов был не* `:=`*, а* `=`.

Если компилятор обнаружит `ident` в последующей части исходного кода, то он заменит его на `expr`. Эта замена работает рекурсивно, то есть если компилятор выполнит один макрос, он будет смотреть результат выражения и определять, не нужно ли выполнить ещё одну замену. Это означает, что нужно осторожно использовать макросы, так как может получиться бесконечный цикл.

Здесь приводятся два примера, которые иллюстрируют работу макросов:

```
{$define sum:=a:=a+b;}
…
sum     {Будет заменено на 'a:=a+b;'
        Обратите внимание на отсутствие точки с запятой}
…
{$define b:=100}
sum     {Будет выполнено рекурсивно a:=a+100;}
…
```

Предыдущий пример может быть выполнен иначе:

```
{$define sum:=a:=a+b;}
…
sum     {Будет заменено на 'a:=a+b;'
        Обратите внимание на отсутствие точки с запятой}
…
{$define b=sum} { НЕ ДЕЛАЙТЕ ЭТОГО !!!}
sum     { Будет бесконечный рекурсивный цикл… }
…
```

На моей системе последний пример привёл к ошибке «кучи», вызвав завершение работы компилятора с ошибкой **203**.

ПРИМЕЧАНИЕ

Макросы, объявленные в интерфейсной части модуля, **не доступны за пределами этого модуля**! Они могут использоваться только как нотации или в условной компиляции.

По умолчанию компилятор предоставляет три макроса, содержащих номер версии, номер выпуска («релиза») и номер «патча». Они перечислены в таблице 2.1.

Таблица 2.1. Предопределённые макросы

Идентификатор	Содержание
FPC_FULLVERSION	Целочисленный номер версии компилятора.
FPC_VERSION	Номер версии компилятора.
FPC_RELEASE	Номер выпуска.
FPC_PATCH	Номер «патча».

Макрос FPC_FULLVERSION содержит номер версии, который всегда использует две цифры для номеров версий «релиза» и «патча». Это означает, что версия **2.3.1** в результате будет FPC_FULLVERSION=20301. Это число позволяет более просто определить минимальный номер версии.

ПРИМЕЧАНИЕ

Не забывайте, что поддержка макросов выключена по умолчанию. Она должна быть включена опцией командной строки -Sm или директивой {$MACRO ON}.

Начиная с версии **3.1.1**, для платформ *(таких как AVR)* компилятор определяет несколько встроенных макросов, определяющих компановку памяти для платформы, они перечислены в таблице (2.2). Эти макросы являются целыми значениями и могут использоваться в коде и в операторах *$IF*.

Таблица 2.2: Компановка встроенной памяти для платформы

Идентификатор	Содержание
FPC_FLASHBASE	Базовый адрес флэш-памяти.
FPC_FLASHSIZE	Размер флэш-памяти.
FPC_RAMBASE	Базовый адрес оперативной памяти.
FPC_RAMSIZE	Размер оперативной памяти.

FPC_BOOTBASE	Базовый адрес загрузочной памяти.
FPC_BOOTSIZE	Размер загрузочной памяти.
FPC_EEPROMBASE	Базовый адрес памяти EEPROM.
FPC_EEPROMSIZE	Размер памяти EEPROM.

2.3. Переменные времени компиляции

В режиме MacPas могут быть определены переменные времени компиляции. Они отличаются от идентификаторов тем, что могут иметь значение, а в отличие от макросов их нельзя использовать для замещения участков исходного кода их значением. Поведение этих переменных совместимо с поведением таких переменных с популярными компиляторами Pascal для Macintosh.

Переменные времени компиляции объявляются подобным образом:

```
{$SETC ident:= expression}
```

Это так называемое **выражение времени компиляции**, которое вычисляется один раз в тот момент, когда директива {$SETC } обнаружена в исходном коде. Значение результата затем присваивается переменной времени компиляции.

Вторая директива {$SETC } для той же переменной перезаписывает предыдущее значение.

В отличие от макросов и идентификаторов, переменные времени компиляции, определённые в интерфейсной части модуля, являются экспортируемыми. Это означает, что их значения будут доступны в модулях, которые используют модуль где объявлена переменная. Для этого требуется, чтобы оба модуля компилировались в режиме macpas.

Большое отличие между макросами и переменными времени компиляции заключается в том, что механизм замещения текста *(подобно C)*, который похож на обычный язык программирования, но доступен только для компилятора.

В режиме MacPas переменные времени компиляции всегда включены.

2.4. Выражения времени компиляции

2.4.1. Определение

Кроме обычных конструкций Turbo Pascal для условной компиляции,

компилятор Free Pascal также поддерживает расширенный механизм условной компиляции: конструкцию {$IF}, которая может быть использована для вычисления выражений времени компиляции.

Прототип этой конструкции следующий:

```
{$if expr}
   CompileTheseLines;
{$else}
   BetterCompileTheseLines;
{$endif}
```

Содержание выражения ограничено тем, что может быть вычислено на этапе компиляции:

- Константы (строки, числа)
- Макросы
- Переменные времени компиляции *(только для режима MacPas)*
- Выражения Pascal с константами *(только для режима Delphi)*

Идентификаторы замещаются их значениями. Для макросов может произойти рекурсивная замена. Следующие логические операторы доступны:

=, <>, >, <, >=, <=, AND, NOT, OR, IN

Оператор IN проверяет наличие переменной времени компиляции в множестве.

Доступны следующие функции:

Функция	Описание
TRUE	Определена только в режиме MacPas, где она равна True. В других режимах может использоваться 1.
FALSE	Определена только в режиме MacPas, где она равна False. В других режимах может использоваться 0.

DEFINED(sym)	Возвращает TRUE, если идентификатор времени компиляции определён. В режиме MacPas скобки не обязательны, то есть
	`{$IF DEFINED(MySym)}`
	Эквивалентно
	`{$IF DEFINED MySym}`

UNDEFINED sym	Возвращает TRUE, если идентификатор времени компиляции *НЕ* определён, и FALSE, если определён *(только в режиме MacPas)*.
OPTION(opt)	Возвращает TRUE, если опция компилятора установлена *(только в режиме MacPas)*. Это эквивалентно директиве {$IFOPT }.

SIZEOF(passym)	Возвращает размер типа Pascal, переменную или константу.
DECLARED(passym)	Возвращает TRUE, если идентификатор Pascal объявлен на текущий момент в исходном коде, иначе возвращает FALSE.

В выражениях при вычислениях используются следующие правила:

- Если все части выражения могут быть вычислены как логические (числа 1 и 0 представляют соответственно TRUE и FALSE), то выражение вычисляется с использованием логических операторов.
- Если все части выражения могут быть вычислены как числовые, то выражение вычисляется с использование чисел.
- Во всех других случаях выражение вычисляется, используя строки

Если полное выражение имеет значение «0», то оно считает ложным и отклоняется. Иначе оно считается истинным и принимается. Это может иметь неожиданные последствия:

```
{$if 0}
```

Будет вычислено как False и отклонено, в то время как

```
{$if 00}
```

Будет вычислено как True.

2.4.2. Использование

Основное использование выражений времени компиляции следующее:

```
{$if expr}
   CompileTheseLines;
{$endif}
```

Если expr возвращает TRUE, то CompileTheseLines будет включено в исходный код.

Подобно обычному Паскалю, можно использовать {$ELSE }:

```
{$if expr}
   CompileTheseLines;
{$else}
   BetterCompileTheseLines;
{$endif}
```

Если expr возвращает TRUE, то будет компилироваться CompileTheseLines. Иначе будет компилироваться BetterCompileTheseLines.

Кроме того, можно использовать {$ELSEIF}

```
{$IF expr}
// …
```

```
{$ELSEIF expr}
// …
{$ELSEIF expr}
// …
{$ELSE}
// …
{$ENDIF}
```

В добавок к описанным выше конструкциям, которые также поддерживаются Delphi, можно создавать эквивалентные конструкции в режиме MacPas:

```
{$IFC expr}
//…
{$ELIFC expr}
…
{$ELIFC expr}
…
{$ELSEC}
…
{$ENDC}
```

То есть `IFC` соответствует `IF`, `ELIFC` соответствует `ELSEIF`, `ELSEC` эквивалентно `ELSE`, а `ENDC` эквивалентно `ENDIF`. Дополнительно `IFEND` эквивалентно `ENDIF`:

```
{$IF EXPR}
   CompileThis;
{$ENDIF}
```

В режиме MacPas возможно комбинировать эти конструкции.

В следующем примере показаны некоторые возможности:

```
{$ifdef fpc}
var
   y : longint;
{$else fpc}
var
   z : longint;
{$endif fpc}
var
   x : longint;
begin
{$IF (FPC_VERSION > 2) or
   ((FPC_VERSION = 2)
   and ((FPC_RELEASE > 0) or
      ((FPC_RELEASE = 0) and (FPC_PATCH >= 1))))}
   {$DEFINE FPC_VER_201_PLUS}
  {$ENDIF}
{$ifdef FPC_VER_201_PLUS}
{$info At least this is version 2.0.1}
{$else}
{$fatal Problem with version check}
{$endif}

{$define x:=1234}
```

```
{$if x=1234}
{$info x=1234}
{$else}
{$fatal x should be 1234}
{$endif}

{$if 12asdf and 12asdf}
{$info $if 12asdf and 12asdf is ok}
{$else}
{$fatal $if 12asdf and 12asdf rejected}
{$endif}

{$if 0 or 1}
{$info $if 0 or 1 is ok}
{$else}
{$fatal $if 0 or 1 rejected}
{$endif}

{$if 0}
{$fatal $if 0 accepted}
{$else}
{$info $if 0 is ok}
{$endif}

{$if 12=12}
{$info $if 12=12 is ok}
{$else}
{$fatal $if 12=12 rejected}
{$endif}

{$if 12<>312}
{$info $if 12<>312 is ok}
{$else}
{$fatal $if 12<>312 rejected}
{$endif}

{$if 12<=312}
{$info $if 12<=312 is ok}
{$else}
{$fatal $if 12<=312 rejected}
{$endif}

{$if 12<312}
{$info $if 12<312 is ok}
{$else}
{$fatal $if 12<312 rejected}
{$endif}

{$if a12=a12}
{$info $if a12=a12 is ok}
{$else}
{$fatal $if a12=a12 rejected}
{$endif}
```

```
{$if a12<=z312}
{$info $if a12<=z312 is ok}
{$else}
{$fatal $if a12<=z312 rejected}
{$endif}

{$if a12<z312}
{$info $if a12<z312 is ok}
{$else}
{$fatal $if a12<z312 rejected}
{$endif}

{$if not(0)}
{$info $if not(0) is OK}
{$else}
{$fatal $if not(0) rejected}

{$endif}{$IF NOT UNDEFINED FPC}
// Обнаружить элементы FPC при компиляции на MAC.
{$SETC TARGET_RT_MAC_68881:= FALSE}
{$SETC TARGET_OS_MAC := (NOT UNDEFINED MACOS)
                OR (NOT UNDEFINED DARWIN)}
{$SETC TARGET_OS_WIN32 := NOT UNDEFINED WIN32}
{$SETC TARGET_OS_UNIX := (NOT UNDEFINED UNIX)
                AND (UNDEFINED DARWIN)}
{$SETC TYPE_EXTENDED := TRUE}
{$SETC TYPE_LONGLONG := FALSE}
{$SETC TYPE_BOOL := FALSE}
{$ENDIF}

{$info
**********************************************************}
{$info * Теперь должно следовать не менее 2 сообщений об
ошибке: *}
{$info
**********************************************************}

{$if not(0}
{$endif}
{$if not(<}
{$endif}
end.
```

Как вы можете видеть в этом примере, эти конструкции не приносят пользы, если используются с обычными идентификаторами. Только если вы используете макросы, которые описаны в разделе 2.2. Макросы [64], они могут оказаться очень полезны. Чтобы проверить этот пример, вы должны включить поддержку макросов с помощью переключателя командной строки `-Sm`.

Следующий пример работает только в режиме MacPas:

```
{$SETC TARGET_OS_MAC := (NOT UNDEFINED MACOS) OR (NOT UNDEFINED
DARWIN)}
```

```
{$SETC DEBUG := TRUE} {$SETC VERSION := 4}
{$SETC NEWMODULEUNDERDEVELOPMENT := (VERSION >= 4) OR DEBUG}
{$IFC NEWMODULEUNDERDEVELOPMENT}
   {$IFC TARGET_OS_MAC}
    //  … новый код mac
   {$ELSEC}
    //  … новый другой код
   {$ENDC}
{$ELSEC}
   … старый код
{$ENDC}
```

2.5. Сообщения

Free Pascal позволяет вам определить предупреждения и сообщения об ошибках в вашем коде. Сообщения можно использовать для отображения полезной информации, такой как примечания о правообладателе, список идентификаторов, на которые реагирует ваш код и т. п.

Предупреждения могут быть использованы, если вы думаете, что какая-то часть вашего кода содержит ошибки, или если вы думаете, что некоторые идентификаторы бесполезны.

Сообщения об ошибках могут быть полезны, если вам нужен определённый идентификатор, чтобы предупредить о том, что какая-то переменная не определена, или если версия компилятора не соответствует вашему коду.

Компилятор воспринимает эти сообщения, как если бы они были сгенерированы компилятором. Это означает, что если предупреждающие сообщения отключены, то эти сообщения не будут отображаться. Ошибки отображаются всегда, а компилятор останавливается, если обнаружит **50** ошибок. После фатальной ошибки компилятор останавливается сразу.

Для сообщений используется следующий синтаксис:

```
{$Message TYPE Текст сообщения}
```

Где TYPE - это:

TYPE	Описание
NOTE	Выдается сообщение-замечание. Эквивалентно $NOTE.
HINT	Выдается сообщение-подсказка. Эквивалентно $HINT.
WARNI	Выдается сообщение-предупреждение. Эквивалентно $WARNING.

NG	
ERROR	Выдается сообщение-ошибка. Эквивалентно $ERROR.
FATAL	Выдается сообщение-фатальная ошибка. Эквивалентно $FATAL.

В качестве альтернативы можно использовать следующий вариант:

```
{$Info Текст сообщения}
```

Для примечаний:

```
{$Note Текст сообщения}
```

Для предупреждений:

```
{$Warning Текст предупреждения}
```

Для подсказок:

```
{$Hint Текст предупреждения}
```

Для ошибок:

```
{$Error Текст ошибки}
```

Для фатальных ошибок:

```
{$Fatal Текст ошибки}
```

или

```
{$Stop Текст ошибки}
```

Разница между сообщениями `$Error` и `$FatalError` или `$Stop` заключается в том, что при обнаружении ошибки компилятор продолжает работу. При обнаружении фатальной ошибки компилятор останавливается.

ПРИМЕЧАНИЕ

Вы не можете использовать символ `'}'` в ваших сообщениях, так как он воспринимается как закрывающая скобка сообщения.

Например, следующий фрагмент кода будет генерировать ошибку, если идентификаторы `RequiredVar1` или `RequiredVar2` определены:

```
{$IFNDEF RequiredVar1}
{$IFNDEF RequiredVar2}
{$Error Должен быть определён хотя-бы один параметр из Re-
quiredvar1 или Requiredvar2 }
{$ENDIF}
{$ENDIF}
```

Но компилятор продолжит компиляцию. Однако при этом не будет создан файл модуля или программы, так как произошла ошибка.

3. ИСПОЛЬЗОВАНИЕ ЯЗЫКА АССЕМБЛЕРА

Free Pascal поддерживает операторы встроенного ассемблера в коде Pascal. Механизм использования ассемблера такой же, как Turbo Pascal и Delphi. Имеются, однако, существенные различия, что будет разъяснено в следующих разделах.

3.1. Использование ассемблера в исходных кодах

Существует два способа вставить код ассемблера в исходный текст Pascal. Первый способ простой – использование блока `asm`:

```
Var
   I : Integer;
begin
   I:=3;
   asm
     movl I, %eax
   end;
end;
```

Всё, что находится между `asm` и `end` – это блок, который вставляется как ассемблер в генерируемый код. В зависимости от режима ассемблера, компилятор выполняет замену имён их адресами.

Второй способ – это использование функции или процедуры на языке ассемблера. Это делается путём добавления модификатора `assembler` в заголовок функции или процедуры:

```
function geteipasebx : pointer; assembler;
asm
   movl (%esp),%ebx ret
end;
```

Также можно объявлять переменные в ассемблерной процедуре:

```
procedure Move(const source; var dest; count:SizeInt);
assembler;
var
   saveesi,saveedi : longint;
asm
   movl %edi,saveedi
end;
```

Компилятор резервирует место в стеке для этих переменных, для чего он вставляет некоторые команды.

Учтите, что ассемблерное имя ассемблерной функции будет «искалечено» компилятором, то есть метка для этой функции будет иметь не такое имя, как указано при объявлении функции. Чтобы изменить это, нужно использовать модификатор `Alias`:

```
function geteipasebx : pointer; assembler;
```

```
[alias:'FPC_GETEIPINEBX'];
asm
   movl (%esp),%ebx ret
end;
```

Чтобы сделать функцию доступной в ассемблерном коде вне текущего модуля, нужно добавить модификатор Public:

```
function geteipasebx : pointer;assembler; [public,
alias:'FPC_GETEIPINEBX'];
asm
   movl (%esp),%ebx ret
end;
```

3.2. Встроенный ассемблер Intel 80x86

3.2.1. Синтаксис Intel

Free Pascal в своих блоках asm поддерживает синтаксис Intel для семейства процессоров Intel Ix86.Синтаксис Intel в вашем блоке asm преобразуется компилятором в синтаксис AT&T, после чего ассемблерный блок вставляется в компилируемый исходный код. Поддерживаемые конструкции ассемблера являются поднабором обычного синтаксиса ассемблера. Из этого следует, что специфические конструкции не поддерживаются в Free Pascal, но существуют в Turbo Pascal:

- Квалификатор TBYTE не поддерживается.
- Идентификатор & не поддерживается.
- Оператор HIGH не поддерживается.
- Оператор LOW не поддерживается.
- Операторы OFFSET и SEG не поддерживаются. Используйте LEA и вариации инструкции Lxx вместо него.
- Выражения со строковыми константами не допускаются.
- Доступ к полям записей через скобки не допускается.
- Преобразование типов с обычными типами Pascal не допускается, только распознаваемые типы ассемблера допускается преобразовывать.
  Например:
  ```
  mov al, byte ptr MyWord – допускается,
  mov al, byte(MyWord) – допускается,
  mov al, shortint(MyWord) -- НЕ допускается.
  ```
- Преобразование типов с константами не допускается.
  Например
  ```
  const s= 10; const t = 32767;
  ```
  в Turbo Pascal:
  ```
  mov al, byte(s) -- бесполезное преобразование типов.
  mov al, byte(t) -- ошибка синтаксиса!
  ```

Анализатор в обоих случаях выдаст ошибку.

- Ссылки на выражения, содержащие только константы, не допускаются *(во всех случаях они не работают в защищённом режиме, например, под LINUX i386)*.
  Примеры:
  ```
  mov al,byte ptr ['c'] -- НЕ допускается.
  mov al,byte ptr [100h] -- НЕ допускается.
  ```
  (Это связано с ограничениями GNU Assembler).
- Скобки внутри квадратных скобок не допускаются.
- Выражения с сегментами, находящиеся полностью в квадратных скобках, на текущий момент не поддерживаются, но они при необходимости могут быть реализованы в `BuildReference`.
  Пример:
  ```
  mov al,[ds:bx] -- НЕ допускается
  ```
  используйте вместо этого:
  ```
  mov al,ds:[bx]
  ```
- Допустимые способы индексации:
  - `Sreg:[REG+REG*SCALING+/-disp]`
  - `SReg:[REG+/-disp]- Sreg:[REG]`
  - `Sreg:[REG+REG+/-disp]`
  - `SReg:[REG+REG*SCALING]`

  Где `Sreg` является не обязательным и определяет сегмент.
  Примечание:
  1. В отличие от Turbo Pascal порядок следования имеет значение.
  2. Значение `Scaling` должно быть числом, а не идентификатором или символом.

  Примеры:
  ```
  const myscale = 1;
  …
  mov al,byte ptr [esi+ebx*myscale] -- НЕ допускается.
  ```
  используется:
  ```
  mov al, byte ptr [esi+ebx*1]
  ```
- Допустимы синтаксис идентификатора переменной выглядит следующим образом *(`ID` = идентификатор переменной или типизированной константы)*:
  ```
  1. ID
  2. [ID]
  3. [ID+expr]
  4. ID[expr]
  ```
  Допустимые поля записей следующие:
  ```
  1. ID.subfield.subfield …
  2. [ref].ID.subfield.subfield …
  3. [ref].typename.subfield …
  ```
- Локальные метки. В отличие от Turbo Pascal, локальные метки должны

содержать, по крайней мере, один символ после указателя локальной метки.
Например:
```
@: -- НЕ допускается
```
используйте вместо этого:
```
@1: -- допускается
```

- В отличие от Turbo Pascal, локальные ссылки не могут использоваться как ссылки, только как перемещения.
  Например:
  ```
  lds si,@mylabel -- НЕ допускается
  ```
- В отличие от Turbo Pascal, сегменты `SEGCS`, `SEGDS`, `SEGES` и `SEGSS` в настоящее время не поддерживаются. *(Их планируется добавить в будущем)*.
- В отличие от Turbo Pascal, где спецификаторы размера памяти могут быть практически везде, встроенный ассемблер Free Pascal Intel требует использовать спецификаторы размера памяти внутри квадратных скобок.
  Пример:
  ```
  mov al,[byte ptr myvar] -- НЕ допускается.
  ```

Используйте:

```
mov al,byte ptr [myvar] -- допускается.
```

- Регистры базы и индекса должны быть 32-разрядными *(ограничение GNU Assembler)*.
- `XLAT` является эквивалентным `XLATB`.
- Поддерживаются только опкоды Single и Double FPU.
- Опкоды плавающей точки на текущий момент не поддерживаются *(за исключением тех, которые связаны только с регистрами плавающей точки)*.

Встроенный ассемблер `Intel` поддерживает следующие макросы:

`@Result` - представляет результат работы функции и возвращает значение.
`Self` - представляет указатель на метод объекта в методах.

3.2.2. Синтаксис AT&T

В ранних версиях Free Pascal использовал только GNU в качестве ассемблера при генерации объектных файлов для процессоров Intel x86. только спустя некоторое время был создан встроенный ассемблер, который непосредственно создаёт объектный файл.

Так как ассемблер GNU использует синтаксис AT&T, написанный вами код должен использовать этот же синтаксис. Отличия между синтаксисом AT&T и Intel, который используется в Turbo Pascal, приведены ниже:

- Имя кода операции включает в себя размер операнда. В общем, можно сказать, что имя кода операции AT&T является именем кода операции Intel с суффиксом `l`, `w` или `b` для, соответственно, типов `longint` (**32** бита),

word (**16** бит) и byte (**8** бит) памяти или регистров. Например, конструкция Intel `mov al, bl` эквивалентна инструкции стиля AT&T `movb %bl,%al`.

- В AT&T непосредственные операнды обозначаются с `$`, в то время как в синтаксисе Intel не используется префикс для таких операндов. Таким образом, конструкция Intel `mov ax, 2` становится `movb $2, %al` в синтаксисе AT&T.
- В AT&T имена регистров начинаются с префикса `%`. Этого нет в синтаксисе Intel.
- В AT&T абсолютные операнды `jump/call` обозначаются знаком `*`, синтаксис Intel не разграничивает эти адреса.
- Порядок следования операндов источника и приёмника меняются местами. Синтаксис AT&T использует `Источник, Приёмник`, в то время как синтаксис Intel использует `Приёмник, Источник`. То есть конструкция Intel `add eax, 4` трансформируется в `addl $4, %eax` на диалекте AT&T.
- Непосредственные длинные переходы сопровождаются префиксом `l`. То есть конструкция Intel `call/jmp section:offset` трансформируется в `lcall/ljmp $section,$offset`. Аналогично для дальних возвратов – `lret` вместо `ret far` на Intel.
- Ссылки на память определяются по разному в ассемблерах AT&T и Intel. Косвенная ссылка на память в Intel:

  ```
  Section:[Base + Index*Scale + Offs]
  ```

  В синтаксисе AT&T будет как:

  ```
  Section:Offs(Base,Index,Scale)
  ```

  где `Base` и `Index` являются не обязательными **32**-разрядными регистрами базы и индекса, а `Scale` используется как множитель для `Index`. Он может принимать значения **1, 2, 4** и **8**. `Section` используется для указания дополнительного регистра для операнда памяти.

Больше информации о синтаксисе AT&T можно найти в руководстве, хотя нужно принять во внимание следующие отличия от обычного ассемблера AT&T:

- Поддерживаются только следующие директивы:

  ```
  .byte
  .word
  .long
  .ascii
  .asciz
  .globl
  ```

- Следующие директивы распознаются, но не поддерживаются:

  ```
  .align
  .lcomm
  ```

В будущем они будут поддерживаться.

- Директивы чувствительны к регистру, остальные идентификаторы ***НЕ*** чувствительны к регистру.
- В отличие от gas, локальные метки/идентификаторы должны начинаться с `.L`.
- Оператор `!` не поддерживается.
- Строковые выражения в операндах не поддерживаются.
- `CBTW`, `CWTL`, `CWTD` и `CLTD` не поддерживаются, используйте вместо них обычные эквиваленты Intel.
- Выражения с константами, которые представляют ссылки на память, не допускаются, даже если непосредственно значение константы поддерживается. Пример:

```
const myid = 10;
…
movl $myid,%eax -- допускаетсямovl
myid(%esi),%eax – НЕ допускается.
```

- Если найдена директива `.globl`, то идентификатор, следующий непосредственно за ней, становится общедоступным и сразу же выделяется. Поэтому имена меток с этим именем будут игнорироваться.
- Только коды операций Single и Double FPU поддерживаются.

Встроенный ассемблер AT&T поддерживает следующие макросы:

`__RESULT` представляет результат работы функции и возвращает значение.
`__SELF` представляет указатель на метод объекта в методах.
`__OLDEBP` представляет старый указатель базы в рекурсивных процедурах.

3.3. Встроенный ассемблер Motorola 680x0

Встроенный ассемблер для процессоров семейства Motorola 680x0 использует синтаксис ассемблера Motorola (`q.v`). Существует несколько отличий:

- Локальные метки начинаются с символа `@`, как в примере:

```
@MyLabel:
```

- Директива `XDEF` в ассемблерном блоке сделает идентификатор общедоступным с указанным именем *(это имя чувствительно к регистру)*.
- Директивы `DB`, `DW`, `DD` могут использоваться только для объявления констант, которые будут записаны в сегмент кода.
- Директива `Align` не поддерживается.
- Арифметические операции с выражениями, содержащими константы, используют те же операнды, что и в версии intel, например, `AND`, `XOR` …
- Сегментные директивы не поддерживаются.
- В настоящее поддерживаются время только коды операций 68000 и поднабора 68020.

Встроенный ассемблер поддерживает следующие макросы:

`@Result` - представляет результат работы функции и возвращает значение.
`Self` - представляет указатель на метод объекта в методах.

3.4. Сигнализация изменения регистров

Если компилятор использует переменные, он иногда записывает их или результат каких-либо вычислений в регистры процессора. Если вы вставляете ассемблерный код в вашу программу, который изменяет регистры процессора, то это может помешать компилятору работать с регистрами. Чтобы избежать этой проблемы, Free Pascal позволяет вам указать компилятору, какие регистры изменялись в ассемблерном блоке (`asm`). Тогда компилятор сохранит и восстановит эти регистры, если он их использовал. Указать компилятору на регистры, которые изменяются, можно путём указания набора имен регистров, имеющихся в ассемблерном блоке, как показано ниже:

```
asm
…
end ['R1', … ,'Rn'];
```

Здесь набор от `R1` до `Rn` – это имена регистров, которые вы изменяете в вашем ассемблерном коде.
Например:

```
asm
movl BP,%eax
movl 4(%eax),%eax
movl %eax,__RESULT
end ['EAX'];
```

В этом примере компилятору указано, что регистр `EAX` был изменён.

Для ассемблерных процедур, то есть для процедур, которые написаны полностью на ассемблере, ABI процессора и платформы должны соблюдаться, то есть процедура сама должна знать, какие регистры сохранять, а какие нет, но она может указать компилятору, используя тот же метод, какие регистры были изменены или нет. Компилятор сохранит указанные регистры в стек на входе и восстановит их на выходе из процедуры.

Единственное, что компилятор обычно делает, это создаёт минимальный кадр стека, если это необходимо *(например, при объявлении переменных)*. Всё остальное зависит от программиста.

4. СГЕНЕРИРОВАННЫЙ КОД

Как упоминалось выше, ранние версии компиляторов Free Pascal использовали ассемблер GNU для создания объектных файлов. Компилятор только генерировал файл на языке ассемблера, который передавался в ассемблер. В следующих двух разделах мы обсудим, что генерируется при компиляции модулей и программ.

4.1. Модули

Когда вы компилируете модуль, компилятор Free Pascal генерирует два файла:

1. Файл описания модуля
2. Файл на языке ассемблера

Файл на языке ассемблера содержит актуальный исходный код для операторов вашего модуля, и выделяет необходимую память для всех переменных, используемых в вашем модуле. Этот файл преобразуется при помощи ассемблера в объектный файл *(с расширением .o)*, который может затем быть скомпонован в другие модули и вашу программу, для формирования исполняемого файла.

По умолчанию ассемблерный файл удаляется после завершения компиляции. Только в случае использования опции командной строки `-s` ассемблерный файл будет оставлен на диске, что позволит вызвать ассемблер позже. Вы можете отключить удаление ассемблерных файлов с помощью переключателя `-a`.

Файл модуля содержит всю информацию, необходимую компилятору для использования модуля:

1. Другие используемые модули, как в разделе `interface`, так и в разделе `implementation`.
2. Типы и переменные из раздела `interface` модуля.
3. Объявления функций из раздела `interface` модуля.
4. Некоторую отладочную информацию, если компиляция выполняется с отладочной информацией.

Подробное описание содержимого и структуры этого файла описано в приложении A. Вы можете изучить описание модуля, используя программу ppudump, которая показывает содержимое файла.

Если вы хотите распространять модуль без исходного кода, вы должны предоставить оба файла модуля: файл описания и объектный файл.

Вы можете также предоставлять заголовочный C для связи с объектным файлом. В этом случае ваш модуль могут использовать программисты, пишущие программы на C. Однако, вы должны создать этот файл самостоятельно, так как компилятор Free Pascal не сделает это за вас.

4.2. Программы

Если вы компилируете программу, компилятор также создаёт два файла:

1. Файл на языке ассемблера, содержащий операторы вашей программы, и распределяющий память для всех используемых переменных.
2. Файл связей компоновщика. Этот файл содержит список объектных файлов, которые компоновщик должен скомпоновать вместе.

По умолчанию файл связей компоновщика удаляется с диска после компиляции. Только если вы укажете опцию командной строки `-s` или если компоновка завершится неудачно, файл останется на диске. Он называется link.res.

Файл на языке ассемблера преобразуется ассемблером в объектный файл, а затем компонуется вместе с остальными модулями и заголовком программы, чтобы окончательно сформировать вашу программу.

Заголовочный файл программы – это маленькая программа на ассемблере, которая предоставляет точку входа для программы. Точка, где начинается ваша программа, зависит от операционной системы, потому что разные операционные системы передают параметры в программы по разному.

По умолчанию его имя prt0.o, а исходный файл находится в prt0.as или в каком-либо варианте этого имени: какой файл используется реально, определяется операционной системой, а на системах LINUX определяет библиотека C, использовать этот файл или нет.

Обычно этот файл находится там, где расположен исходный код системного модуля для вашей системы. Его основная функция заключается в том, чтобы сохранить окружение и аргументы командной строки и установить стек. Затем он вызывается основной программой.

5. ПОДДЕРЖКА INTEL MMX

5.1. О чем это?

Free Pascal поддерживает новые инструкции MMX *(Multi-Media extensions – мультимедийные расширения)* процессоров Intel. Основная идея MMX – это обработка большого количества данных в одной инструкции. Например, процессор может сложить одновременно 4 слова. Для реализации этого эффекта, язык Pascal должен быть расширен. Так, Free Pascal позволяет складывать, например, два массива `array[0..3] of word`, если поддержка MMX включена. Операция выполняется с помощью модуля MMX, который позволяет людям без знания ассемблера использовать преимущества расширений MMX.

Пример:

```
uses
   MMX;{ подключить некоторые предопределённые типы данных }
const
   { tmmxword = array[0..3] of word;, объявлено при помощи
модуля MMX }
   w1 : tmmxword = (111, 123, 432, 4356);
   w2 : tmmxword = (4213, 63456, 756, 4);
var
   w3 : tmmxword;
   l: longint;
begin
   if is_mmx_cpu then { is_mmx_cpu экспортировано из модуля mmx
}
      begin
{$mmx+}  { включить поддержку mmx }
         w3:=w1+w2;
{$mmx-}
      end
   else
      begin
         for i:=0 to 3 do
         w3[i]:=w1[i]+w2[i];
      end;
end.
```

5.2. Поддержка насыщенности

Одним из важных моментов MMX является поддержка насыщенных операций. Если операция приведёт к переполнению, то значение останется наибольшим или наименьшим из всех возможных значений для данного типа данных: Если вы используете тип `byte`, то в нормальном режиме вы получите **250+12=6**.

Это очень раздражает, когда выполняются операции с цветом или звуком, когда вам приходится использовать слово данных, чтобы проверить, что значение более **255**. Насыщенные операции поддерживаются модулем MMX. Если вы хотите их использовать, вам нужно просто включить поддержку насыщенности: `$saturation+`

Пример:

```
Program SaturationDemo;
{
   Пример для насыщенности, шкала данных (например, для аудио)
   От 1.5 до минус бесконечности
}
uses mmx;
var
   audio1 : tmmxword;
   I: smallint;
const
   helpdata1 : tmmxword = ($c000,$c000,$c000,$c000);
   helpdata2 : tmmxword = ($8000,$8000,$8000,$8000);
begin
   { audio1 содержит четыре 16битных аудио-сэмпла }
{$mmx+}
   { преобразовать его в $8000, определить как 0, умножить
данные на 0.75 }
   audio1:=(audio1+helpdata2)*(helpdata1);
{$saturation+}
   { избежать переполнения (все значения > $ffff будут $ffff) }
   audio1:=(audio1+helpdata2)-helpdata2;
{$saturation-}
   { теперь умножаем на 2 и преобразуем в целое }
   for i:=0 to 3 do
      audio1[i] := audio1[i] shl 1;
   audio1:=audio1-helpdata2;
{$mmx-}
end.
```

5.3. Ограничения поддержки MMX

В начале **1997** года инструкции MMX введены в процессоры Pentium, но многозадачность систем не распространялась на вновь введённые регистры MMX. Чтобы обойти эту проблему, Intel наложила регистры MMX на регистры FPU.

Следствием этого является то, что вы не можете смешивать операции MMX с операциями с плавающей точкой. После использования операций MMX и перед использованием операций с плавающей точкой, вы должны вызвать процедуру EMMS модуля MMX. Эта процедура восстанавливает регистры FPU.

ВНИМАНИЕ!

Компилятор не предупредит вас, если вы смешаете операции плавающей точки и MMX, поэтому ***будьте осторожны***.

Инструкции MMX оптимизированы для мультимедийных операций. Поэтому их нельзя использовать для всех возможных операций: некоторые операции приведут к несоответствию типов, см. раздел 5.4. Поддерживаемые операции MMX[85] Поддерживаемые операции MMX.

Важным ограничением является то, что операции MMX не выполняют проверку диапазона или переполнения, даже если вы включите эти проверки. Такова природа операций MMX.

Модуль MMX должен всегда использоваться, если выполняются операции MMX, потому что код выхода этого модуля очищает регистры MMX. Если этого не сделать, то другие программы могут быть нарушены. Следствием этого является то, что вы не можете использовать операции MMX в коде выхода вашего модуля или программы, так как они будут конфликтовать с кодом выхода модуля MMX. Компилятор не может это проверить, только вы отвечаете за это!

5.4. Поддерживаемые операции MMX

Следующие операции поддерживаются компилятором, если расширения `MMX` включены:

- Сложение: `addition (+)`.
- Вычитание: `subtraction (-)`.
- Умножение: `multiplication(*)`.
- Логическое `ИСКЛЮЧАЮЩЕЕ ИЛИ`: `logical exclusive or (xor)`.
- Логическое `И`: `logical and (and)`.
- Логическое `ИЛИ`: `logical or (or)`.
- Смена знака: `sign change (-)`.

5.5. Оптимизация поддержки MMX

Несколько рекомендаций для того, чтобы добиться максимальной производительности:

- Вызов `EMMS` занимает много времени, поэтому постарайтесь разделить операции с плавающей точкой и операции MMX.
- Используйте MMX только в процедурах низкого уровня, потому что компилятор сохраняет все используемые регистры MMX при вызове подпрограмм.
- Оператор `NOT` не поддерживается MMX, поэтому компилятору придётся создать обходной путь, что будет не эффективным.
- Простое присваивание чисел с плавающей точкой не доступно регистрам

FPU, поэтому вам нет необходимости вызывать процедуру EMMS. Только при выполнении арифметических операций вам нужно вызвать процедуру EMMS.

6. ВОПРОСЫ КОДИРОВАНИЯ

В этом разделе содержится подробная информация о коде, генерируемом компилятором Free Pascal. Эта информация может оказаться полезной при написании внешних объектных файлов, которые будут компоноваться Free Pascal при создании блоков кода.

6.1. Соглашения о регистрах

Компилятор имеет различные соглашения о регистрах, в зависимости от используемого целевого процессора. Некоторые из этих регистров имеют специфику использования при генерации кода. Следующие разделы описывают общие имена регистров на основных платформах. Они также описывают, какие регистры используются как временные регистры, а какие могут быть освобождены с помощью ассемблерных блоков.

6.1.1. Аккумулятор

Аккумулятор может быть не менее чем **32**-разрядным. Это аппаратный регистр, который предназначен для работы с целыми числами и используется для возвращения результатов функций, которые возвращают целые значения.

6.1.2. 64-разрядный аккумулятор

64-битный регистр-аккумулятор используется в **32**-разрядной среде и определяется как группа регистров, которые используются, если вызываемая функция возвращает **64**-битный результат. Это парный регистр.

6.1.3. Регистр результата с плавающей точкой

Этот регистр используется для возвращения результата с плавающей точкой из функций.

6.1.4. Регистр объектов

Регистр объектов содержит указатель на актуальный объект или класс. Регистр предоставляет доступ к данным объекта или класса, и VMT-указатель этого объекта или класса.

6.1.5. Регистр-указатель кадра

Регистр-указатель кадра используется для доступа к параметрам в подпрограмме, как будто это доступ к локальным переменным. Ссылки на

передаваемые параметры и локальные переменные создаются, используя указатель кадра. *Указатель кадра доступен **НЕ** для всех платформ.*

6.1.6. Регистр-указатель стека

Указатель стека используется для получения доступа к области стека, где хранятся локальные переменные и параметры процедур.

6.1.7. Временные регистры

Временные регистры – это регистры, которые могут быть использованы в ассемблерных блоках или внешних объектных файлах без необходимости какого-либо сохранения перед использованием.

6.1.8. Таблица регистров процессора

Здесь описано, какие регистры используются и для каких целей на каждом из поддерживаемых процессоров в `Free Pascal`. Здесь также показано, какие регистры можно использовать как временные регистры.

Версия Intel 80x86

Таблица 6.1. Регистры Intel 80x86

Общее имя регистра	Имя регистра процессора
Аккумулятор	**EAX**
64-разрядный аккумулятор, старшее/ младшее слово	**EDX:EAX**
Результат с плавающей точкой	**FP(0)**
Регистр объектов	**ESI**
Указатель кадра	**EBP**
Указатель стека	**ESP**
Временные регистры	Недоступно

Версия Motorola 680x0

Таблица 6.2. Регистры Motorola 680x0

Общее имя регистра	Имя регистра процессора	Примечание
Аккумулятор	**D0**	Для совместим

		ости с некоторыми компиляторами C, когда результат функции является указателем и объявлен в конвенции с `cdecl`, результат также сохраняется в регистре `A0`.
64-разрядный аккумулятор, старшее/младшее слово	**D0:D1**	
Результат с плавающей точкой	**FP0**	При эмуляции регистров плавающей точки результат возвращается в `D0`.
Регистр объектов	**A5**	
Указатель кадра	**A6**	
Указатель стека	**A7**	
Временные регистры	**D0, D1, A0, A1, FP0, FP1**	

6.2. Преобразование имён

В отличие от большинства компиляторов C и ассемблеров все метки, генерируемые для переменных и процедур в Pascal, имеют искажённые имена *(этого можно избежать, используя модификаторы `alias` или `cdecl`)*. Это сделано для того, чтобы компилятор мог более строго проверять типы при разборе кода. Это также позволяет выполнять перегрузку процедур и функций.

6.2.1. Преобразование имён для блоков данных

Правила преобразования имён для переменных и типизированных констант следующие:

- Все имена переменных преобразуются в верхний регистр
- Переменные в основной программе или в разделе `private` модуля будут иметь перед именем символ подчёркивания (`_`)
- Типизированные константы будут иметь `TC__` перед именем
- Переменные `private` в модуле имеют в качестве префикса имя этого модуля: `U_UNITNAME_`
- Типизированные константы в разделах `public` и `private` в модуле имеют в качестве префикса имя этого модуля: `TC__UNITNAME$$`

Примеры:

```
unit testvars;

interface

const
publictypedconst : integer = 0;
var
   publicvar : integer;

implementation
const
   privatetypedconst : integer = 1;
var
   privatevar : integer;
end.
```

В результате будет сгенерирован ассемблерный код для GNU ассемблера:

```
.file "testvars.pas"
.text
.data
# [6] publictypedconst : integer = 0;
.globl TC__TESTVARS$$_PUBLICTYPEDCONST
TC__TESTVARS$$_PUBLICTYPEDCONST:
.short 0
# [12] privatetypedconst : integer = 1;
TC__TESTVARS$$_PRIVATETYPEDCONST:
.short 1
.bss
# [8] publicvar : integer;
.comm U_TESTVARS_PUBLICVAR,2
# [14] privatevar : integer;
.lcomm _PRIVATEVAR,2
```

6.2.2. Преобразование имён для блоков кода

Для преобразования имён процедур применяются следующие правила:

- Имена процедур преобразуются в верхний регистр
- К именам процедуры в модуле будет добавлено имя этого модуля: _UNITNAME$$_
- К именам всех процедур в основной программе будет добавлено _.
- Все параметры в процедуре преобразуются с использованием типа параметра *(в верхнем регистре)* и добавлением символа $. Это делается в порядке слева направо для каждого параметра в процедуре.
- Объекты и классы используют специальное предобразование: тип класса или объекта получает преобразованное имя. Имя преобразуется следующим образом: _$$_TYPEDECL_$$ дополнительно начинается с имени модуля и заканчивается именем метода.

Следующие конструкции

```
unit testman;
interface
type
   myobject = object
      constructor init;
      procedure mymethod;
   end;

implementation
constructor myobject.init;
begin
end;

procedure myobject.mymethod;
begin
end;

function myfunc: pointer;
begin
end;

procedure myprocedure(var x: integer; y: longint; z : pchar);
begin
end;

end.
```

Будут преобразованы в следующий ассемблерный файл для целевого процессора Intel 80x86:

```
.file "testman.pas"

.text .balign 16
.globl _TESTMAN$$_$$_MYOBJECT_$$_INIT
_TESTMAN$$_$$_MYOBJECT_$$_INIT:
 pushl %ebp
 movl %esp,%ebp
 subl $4,%esp
 movl $0,%edi
 call FPC_HELP_CONSTRUCTOR
```

```
    jz .L5
    jmp .L7
    .L5:
    movl 12(%ebp),%esi
    movl $0,%edi
    call FPC_HELP_FAIL
    .L7:
    movl %esi,%eax
    testl %esi,%esi
    leave
    ret $8
    .balign 16
    .globl _TESTMAN$$_$$_MYOBJECT_$$_MYMETHOD
    _TESTMAN$$_$$_MYOBJECT_$$_MYMETHOD:
    pushl %ebp
    movl %esp,%ebp
    leave
    ret $4
    .balign 16
    _TESTMAN$$_MYFUNC:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl -4(%ebp),%eax
    leave
    ret
    .balign 16
    _TESTMAN$$_MYPROCEDURE$INTEGER$LONGINT$PCHAR:
    pushl %ebp
    movl %esp,%ebp
    leave
    ret $12
```

6.2.3. Модификация преобразованных имён

Чтобы сделать идентификаторы доступными извне, можно дать псевдонимы преобразованным именам, или изменить непосредственно преобразованное имя. Можно использовать два модификатора:

public:
: для функции, которая имеет идентификатор `public` преобразованное имя будет точно таким же, как оно объявлено.

alias:
: Модификатор `alias` можно использовать для присваивания второй ассемблерной метки для вашей функции. Эта метка будет иметь такое же имя, как объявленный вами алиас *(псевдоним)*. Это не изменяет соглашение о вызовах для функции. Иными словами, модификатор `alias` позволяет вам указать другое имя (ник) для вашей функции или процедуры.
  Прототип для процедуры или функции с псевдонимом следующий:
  ```
  Procedure AliasedProc; alias : 'AliasName';
  ```
  Процедура `AliasedProc` будет также называться как `AliasName`. Помните, что указанное имя чувствительно к регистру (как в C).

Кроме того, раздел `exports` библиотеки также используется для объявления имён, которые будут экспортироваться в общедоступной библиотеке. Имена в разделе `exports` также чувствительны к регистру *(в то время как в фактическом объявлении, как правило, не чувствительны)*. Более подробную информацию о создании общедоступных библиотек см. в разделе 12.1. Введение[156].

6.3. Механизм вызова

По умолчанию компилятор использует вызывающий механизм `register`, то есть компилятор попытается передать как можно больше параметров, сохранив их в свободных регистрах. Используются не все регистры, потому что некоторые регистры имеют специальное назначение, это зависит от типа процессора.

Результат функции возвращается в аккумуляторе (первый регистр), если его размер вписывается в регистр. Вызовы методов (из объектов или классов) имеют дополнительный невидимый параметр - `self`.

Когда процедура или функция завершается, она очищает стек.

Вызовы методов доступны для компоновки с внешними объектными файлами и библиотеками и описаны в таблице 6.3. В первом столбце приведен модификатор, которые указываются при объявлении процедуры. Во втором приведён порядок, в котором параметры помещаются в стек. В третьем столбце указывается, кто отвечает за очистку стека: вызывающий объект или вызываемая функция. Столбец *выравнивание* показывает как выравниваются параметры, отправленные в стек.

Подпрограммы будет изменять количество регистров *(изменяемые регистры)*. Перечень регистров, которые будут изменяются сильно зависит от процессора, соглашения о вызовах и ABI на целевой платформе.

Таблица 6.3. Механизм вызовов в Free Pascal

Модификатор	**Порядок помещения параметров**	**Стек очищается**	**Выравнивание**
H	Слева направо	Функцией	По умолчанию

el			
register	Слева направо	Функцией	По умолчанию
cocoll	Справа налево	Вызывающим	Выравнивание GCC
interrupt	Справа налево	Функцией	По умолчанию
pascall	Слева направо	Функцией	По умолчанию
safecall	Справа налево	Функцией	По умолчанию
stdcall	Справа налево	Функцией	Выравнивание GCC
syscall	Справа налево	Вызывающим	По умолчанию

oldfpccall			

Учтите, что соглашение о вызовах `oldfpccall` эквивалентно соглашению о вызовах по умолчанию на процессорах, отличных от **32**-разрядных Intel 386 или выше.

Более подробно об этой теме см. в разделе 7. ВОПРОСЫ КОМПОНОВКИ[100] или в ссылках. Информацию о сохранённых регистрах GCC, выравнивании стека GCC и основном выравнивании стека в операционных системах выходит за пределы данного рукводства.

C версии **2.0** (на самом деле начиная где-то с версии **1.9**) модификатор `register` является соглашением о вызовах по умолчанию, ранее по умолчанию был `oldfpccall`.

Соглашение о вызовах по умолчанию, то есть соглашение, используемое в директиве `{$calling }`, описано в разделе 1.2.7. $CALLING : Определить соглашение о вызовах[13]. Соглашение по умолчанию для текущей платформы может быть указано следующим образом:

```
{$CALLING DEFAULT}
```

ПРИМЕЧАНИЕ

Модификатор `popstack` больше не поддерживается, начиная с версии **2.0**, он был переименован в `oldfpccall`. Модификатор `saveregisters` больше не используется.

6.4. Вложенные процедуры и функции

Если процедура объявлена внутри другой процедуры или функции, то она называется вложенной. В этом случае во вложенную процедуру передаётся дополнительный невидимый параметр. Этот дополнительный параметр является указателем фрейма родительской процедуры. Это позволяет получать доступ к локальным переменным и параметрам вызывающей процедуры.

В результате фрейм стека после вхождения кода простой вложенной процедуры будет выполнено, как показано в таблице 6.4.

Таблица 6.4. Фрейм стека при вызове вложенной процедуры (**32**-битный процессор).

Смещение от указателя фрейма	Что записано
+x	Параметры
8	Указатель фрейма родительской процедуры
4	Адрес возврата
0	Сохранённый указатель фрейма

6.5. Вызовы конструктора и деструктора

Конструктор и деструкторы имеют специальные невидимые параметры, которые передаются в них. Эти невидимые параметры используются внутренне для создания экземпляров классов и объектов.

6.5.1. Объекты

В действительности невидимое объявление конструктора объекта следующее:

```
constructor init(_vmt : pointer; _self : pointer …);
```

Где `_vmt` – это указатель на таблицу виртуальных методов для этого объекта. Это значение равно `nil`, если конструктор из экземпляра объекта (такого как вызов наследуемого конструктора).

`_self` – это либо `nil`, если экземпляр должен быть создан динамически *(объект объявлен как указатель)*, или адрес экземпляра объекта, если объект объявлен как обычный объект *(сохранён в области данных)* или если экземпляр объекта уже создан.

Созданный экземпляр *(если он создан через `new`)*, возвращается в аккумулятор.

Деструктор объявляется следующим образом:

```
destructor done(_vmt : pointer; _self : pointer …);
```

Где `_vmt` – это указатель на таблицу виртуальных методов этого объекта. Это значение равно `nil`, если деструктор вызывается из экземпляра объекта *(такого как вызов наследуемого конструктора)*, или если экземпляр объекта – это переменная, а не указатель.

`_self` – это адрес экземпляра объекта.

6.5.2. Классы

В действительности невидимое объявление конструктора класса следующее:

```
constructor init(_vmt: pointer; flag : longint; ..);
```

`_vmt` – это любо `nil`, если вызывается из экземпляра класса или если

вызывающий наследованный конструктор, иначе точки для адреса таблицы виртуальных методов.

Здесь `flag` – это ноль, если конструктор вызван из экземпляра объекта или экземпляра спецификатора, иначе `flag` равен **1**.

Созданные экземпляры *(`self`)* возвращаются в аккумулятор

Объявление деструктора следующее:

```
destructor done(_self : pointer; flag : longint …);
```

`_self` – это адрес экземпляра объекта.

`flag` – это ноль, если деструктор вызван из экземпляра объекта или экземпляра спецификатора, иначе `flag` равен **1**.

6.6. Код входа/выхода

Каждая процедура в Pascal начинается и заканчивается стандартным кодом начала и завершения.

6.6.1. Стандартная процедура начала/завершения Intel 80x86

Стандартный код входа для процедур и функций в архитектуре 80x86 следующий:

```
pushl %ebp
movl %esp,%ebp
```

Сгенерированная последовательность выхода для процедур и функций выглядит следующим образом:

```
leave
ret $xx
```

Где `xx` – это общий размер переданных параметров.

Больше информации о возвращаемых функциями значениях можно найти в разделе 6.1. Соглашения о регистрах[87]. Соглашения о регистрах.

6.6.2. Стандартная процедура начала/завершения Motorola 680x0

Стандартный код входа для процедур и функций в архитектуре `680x0` следующий:

```
move.l a6,-(sp)
move.l sp,a6
```

Сгенерированная последовательность выхода для процедур и функций выглядит следующим образом *(в режиме процессора по умолчанию)*:

```
unlk a6
rtd #xx
```

Где xx – это общий размер переданных параметров.

Больше информации о возвращаемых функциями значениях можно найти в разделе 6.1. Соглашения о регистрах[87]. Соглашения о регистрах.

6.7. Передача параметра

При вызове процедуры или функции компилятор выполняет следующие действия:

1. Если имеются какие-либо параметры для передачи в процедуру, они сохраняются в известных регистрах, а если имеется больше параметров, чем свободных регистров, то они помещаются в стек слева направо.
2. Если вызывается функция, которая возвращает значение типа `String`, `Set`, `Record`, `Object` или `Array`, то возвращается адрес для хранения результата функции, аналогично выполняется передача в процедуру.
3. Если вызываемая процедура или функция – это метод объекта, то указатель на `self` передаётся в процедуру.
4. Если процедура или функция вложена в другую функцию или процедуру, то указатель фрейма родительской процедуры помещается в стек.
5. Возвращаемый адрес помещается в стек *(это делается автоматически с помощью инструкции, вызывающей подпрограмму)*.

В результате фрейм стека при выходе выглядит так, как показано в таблице 6.5.

Таблица 6.5. Фрейм стека при вызове процедуры (32-битная модель).

Смещение	Что записано	Дополнительно?
+x	Расширенные параметры	Да
+12	Результат функции	Да
+8	`self`	Да
+4	Возвращаемый адрес	Нет
+0	Указатель фрейма родительской процедуры	Да

6.7.1. Выравнивание параметров

Каждый параметр, передаваемый в процедуру, гарантированно уменьшает указатель стека на определённое минимальное число. Это поведение может меняться в зависимости от операционной системы. Например, передача байта как параметра по значению в процедуру, может уменьшить указатель стека на **1, 2, 4** или даже **8** байт, в зависимости от целевой операционной системы и процессора.

Например, на FreeBSD все параметры передаются в процедуру с минимальным уменьшением стека в **4** байта на параметр, даже если в реальности для записи параметра в стек требуется менее **4** байтов (например, при помещении в стек параметра размером **1** байт).

6.8. Ограничения стека

Некоторые процессоры имеют ограничения на размер параметров и локальных переменных в подпрограммах. Эти ограничения показаны в таблице 6.6.

Таблица 6.6. Максимальные ограничения для процессоров.

Процессор	Параметры	Локальные переменные
Intel 80x86 *(все)*	**64** КБ	Без ограничений
Motorola 68020 *(по умолчанию)*	**32** КБ	Без ограничений
Motorola 68000	**32** КБ	**32** КБ

Кроме того, компилятор m68k в режиме 68000 ограничивает размер элементов данных до **32** КБ *(массивы, записи, объекты и т.п.)*. Этого ограничения нет в режиме 68020.

7. ВОПРОСЫ КОМПОНОВКИ

Если вы используете только код на Pascal и модули Pascal, то вы не увидите большую часть того, что делает компоновщик при создании исполняемого файла (программы). Компоновщик вызывается только при компиляции программы. При компиляции модулей компоновщик не используется.

Однако имеются случаи компоновки библиотек C или внешних объектных файлов, созданных другими компиляторами, когда компоновщик может оказаться необходим. Компилятор Free Pascal может генерировать вызовы функций C, а может генерировать функции, которые могут быть вызваны из C *(экспортируемые функции)*.

7.1. Использование внешнего кода и переменных

В общем случае вы должны выполнить **3** действия, чтобы использовать функцию, которая хранится во внешней библиотеке или объектном файле:

1. Вы должны объявить в Pascal функцию или процедуру, которую хотите использовать.
2. Вы должны сделать правильное объявление в соответствии с соглашением о вызовах.
3. Вы должны указать компилятору, где расположена функция, то есть в каком объектном файле или библиотеке она находится, чтобы компилятор мог скомпоновать необходимый код.

То же справедливо и для переменных. Для доступа к переменной, которая расположена во внешнем объектном файле. Вы должны объявить её, а затем указать компилятору, где эту переменную искать. В следующих разделах мы попытаемся объяснить, как это сделать.

7.1.1. Объявление внешних функций и процедур

Первый шаг, который вы должны сделать для использования внешних блоков кода – это объявление функции, которую вы хотите использовать. Free Pascal поддерживает синтаксис Delphi, то есть вы можете использовать директиву `external`. Директива `external` по сути заменяет блок кода функции.

Директива `external` не определяет соглашение о вызовах. Она только указывает компилятору, что код процедуры или функции размещён во внешнем блоке кода. Модификатор соглашения о вызовах должен быть объявлен, если внешний блок кода не имеет не соответствует соглашениям о вызовах Free Pascal. Более подробную информацию о соглашениях о вызовах можно найти в разделе 6.3. Механизм вызова [93].Существует четыре варианта директивы `external`:

1. Простое объявление внешней подпрограммы:

```
Procedure ProcName (Args : TPRocArgs); external;
```

Директива external указывает компилятору, что функция размещена во внешнем блоке кода. Вы можете использовать её вместе с директивами {$L} или {$LinkLib} для компоновки функции или процедуры в библиотеке или внешнем объектном файле. Объектные файлы просматриваются в путях поиска объектов *(устанавливается при помощи -Fo)*, а библиотеки ищутся в путях компоновщика *(устанавливаются при помощи -Fl)*.

2. Вы можете передать в директиву external имя библиотеки как аргумент:

```
Procedure ProcName (Args : TPRocArgs); external ' Name';
```

Это укажет компилятору, что процедура размещена в библиотеке с именем «Name». Этот способ эквивалентен следующему:

```
Procedure ProcName (Args : TPRocArgs);external;{$LinkLib
'Name'}
```

3. Директива external может также использоваться с двумя аргументами:

```
Procedure ProcName (Args : TPRocArgs); external 'Name' name
'OtherProcName';
```

Это имеет такое же значение, как и в предыдущем объявлении, только компилятор будет использовать имя «OtherProcName» при компоновке библиотеки. Это можно использовать для получения различных имён для процедур и функций во внешней библиотеке. Имя процедуры чувствительно к регистру и должно точно совпадать с именем программы в объектном файле.

Этот способ эквивалентен следующему коду:

```
Procedure OtherProcName (Args : TProcArgs); external;
{$LinkLib 'Name'}
Procedure ProcName (Args : TPRocArgs);
begin
   OtherProcName (Args);
end;
```

4. Наконец, под WINDOWS и OS/2 имеется четвёртый способ определить внешнюю функцию: в файлах .DLL, где функции также имеют уникальный номер *(их индекс)*. Можно ссылаться на эти функции, используя их индекс:

```
Procedure ProcName (Args : TPRocArgs); external 'Name' Index
SomeIndex;
```

Это указывает компилятору, что процедура ProcName расположена в динамической библиотеке и имеет индекс SomeIndex.

ПРИМЕЧАНИЕ

Учтите, что это доступно только под WINDOWS и OS/2.

7.1.2. Объявление внешних переменных

Некоторые библиотеки или блоки кода имею экспортируемые переменные. Вы можете получить доступ к этим переменным также, как и к внешним функциям. Для доступа к внешней переменной вы объявляете её следующим образом:

```
Var
   MyVar : MyType; external name 'varname';
```

Это имеет двойной эффект:

1. Не требуется память для размещения этой переменной
2. Имя переменной используется в ассемблерном коде как `varname`. Это имя чувствительно к регистру, вы должны это учитывать.

Переменная будет доступна по объявленному имени, в нашем случае это `MyVar` с учётом регистра.

Возможен другой вариант объявления:

```
Var
   varname : MyType; cvar; external;
```

Это объявление также имеет двойной эффект:

Модификатор `external` гарантирует, что не выделяется место для этой переменной.

Модификатор `cvar` указывает компилятору, что имя переменной, используемой в ассемблерном коде, является точно таким, как указано в объявлении. Это имя чувствительно к регистру, вы должны это учитывать.

Первый вариант позволяет изменить имя внешней переменной для внутреннего использования. В качестве примера рассмотрим следующий C-файл *(в extvar.c)*:

```
/* Объявить переменную, выделить место для хранения */
int extvar = 12;
```

и следующую программу *(в extdemo.pp)*:

```
Program ExtDemo;

{$L extvar.o}

Var { Чувствительное к регистру объявление!!! }
   extvar : longint;
   cvar; external;
   I : longint; external name 'extvar';

begin
{ Extvar можно использовать с именем, не чувствительным к
регистру!!! }
   Writeln('Переменная ''extvar'' имеет значение: ', ExtVar);
   Writeln('Переменная ''I'' имеет значение: ', i);
end.
```

Компиляция C-файла и программы на Pascal:

```
gcc -c -o extvar.o extvar.c
ppc386 -Sv extdemo
```

создаст программу, которая печатает

```
Переменная 'extvar' имеет значение: 12
Переменная 'I'      имеет значение: 12
```

на экране.

7.1.3. Объявление модификатора соглашений о вызовах

Чтобы быть уверенными в том, все параметры корректно передаются во внешнюю подпрограмму, вы должны объявлять её с модификатором соглашения о вызовах. При компоновке кодовых блоков, откомпилированных стандартными компиляторами C (такими как GCC), должен использоваться модификатор `cdecl`, чтобы показать, что внешняя подпрограмма использует соглашение о вызовах C-типа. Больше информации о поддерживаемых соглашениях о вызовах вы можете найти в разделе 6.3. Механизм вызова [93].

Как и следовало ожидать, при объявлении внешних переменных не требуется использование каких-либо модификаторов соглашений о вызовах.

7.1.4. Объявление внешнего объектного кода

Компоновка объектного файла

Объявив внешнюю функцию или переменную, которая расположена в объектном файле, вы можете использовать её, как если бы она была объявлена в вашей собственной программе или модуле. Для получения исполняемого файла вы ещё должны скомпоновать объектный файл. Это можно сделать с помощью директивы `{$L file.o}`.

Это приведёт к тому, что компоновщик будет ссылаться на объектный файл file.o. На большинстве систем это имя чувствительно к регистру. Поиск объектного файла сначала выполняется в текущей директории, а затем в директориях, указанных в командной строке директивой `-Fo`.

Вы не можете определить библиотеки таким способом, это только для объектных файлов.

Далее мы покажем пример. Будем считать, что у нас есть некоторая ассемблерная подпрограмма, которая использует соглашения о вызовах C и вычисляет n-е число Фибоначчи:

```
.text
        .align 4
```

```
.globl Fibonacci
.type Fibonacci, @function
Fibonacci:
      pushl %ebp
      movl %esp, %ebp
      movl 8(%ebp), %edx
      xorl %ecx, %ecx
      xorl %eax, %eax
      movl $1, %ebx
      incl %edx
loop:
      decl %edx
      je endloop
      movl %ecx, %eax
      addl %ebx, %eax
      movl %ebx, %ecx
      movl %eax, %ebx
      jmp loop
endloop:
      movl %ebp, %esp
      popl %ebp
      ret
```

Затем мы можем вызвать эту функцию в следующей программе на Pascal:

```
Program FibonacciDemo;
var i : longint;
Function Fibonacci (L : longint):longint; cdecl; external;
{$L fib.o}
begin
   For I:=1 to 40 do
     writeln('Fib(',i,') : ', Fibonacci(i));
end.
```

Только с двумя командами, которые могут быть выполнены в программе:

```
as -o fib.o fib.s
ppc386 fibo.pp
```

Это пример предполагает, что у вас есть ассемблерная подпрограмма в fib.s, а ваша программа на Pascal находится в файле fibo.pp.

Компоновка библиотеки

Процесс компоновки вашей программы в библиотеку зависит от того, как вы объявляете внешнюю процедуру.

В том случае, если вы используете следующий синтаксис объявления процедуры:

```
Procedure ProcName (Args : TPRocArgs); external 'Name';
```

Вам нет необходимости предпринимать какие-либо дополнительные действия для компоновки вашего файла, компилятор сделает всё что нужно вместо вас. На WINDOWS будет скомпонован файл с именем name.dll, на LINUX и других

UNIX-подобных системах ваша программа будет скомпонована в библиотеку libname, которая может быть статической или динамической библиотекой.

В случае использования

```
Procedure ProcName (Args : TPRocArgs); external;
```

Вам необходимо указать точную ссылку на библиотеку. Это можно сделать двумя путями:

1. Вы можете указать компилятору в исходном файле библиотеку для компоновки, используя директиву
   ```
   {$LinkLib 'Name'}:
   {$LinkLib 'gpm'}
   ```
   Это будет ссылка на библиотеку gpm. На UNIX системах *(таких как LINUX)*, вы не должны указывать расширение или префикс библиотеки `'lib'`. Компилятор позаботится об этом. На других системах *(таких как WINDOWS)*, вам необходимо указать полное имя.
2. Вы также можете указать компилятору ссылку на библиотеку в командной строке: для этого можно использовать опцию `-k`. Например
   ```
   ppc386 -k'-lgpm' myprog.pp
   ```
   Это эквивалентно описанному выше методу, указывает компоновщику компоновать библиотеку gpm.

В качестве примера рассмотрим следующую программу:

```
program printlength;
{$linklib c} { Чувствительно к регистру }

{ Объявление для стандартной функции C strlen }
Function strlen(P : pchar) : longint; cdecl; external;
begin
   Writeln(strlen('Программирование - это легко!'));
end.
```

Эта программа может быть скомпилирована с

```
ppc386 prlen.pp
```

Предполагая, конечно, что исходный текст программы находится в prlen.pp.

Для использования функций в C, которые имеют меняющееся количество параметров, вы должны компилировать ваш модуль или программу в режимах `objfpc` или `Delphi`, и использовать параметр `Array of const`, как это показано в следующем примере:

```
program testaocc;
{$mode objfpc}

Const
   P : Pchar
         = 'example';
   F : Pchar
         = 'This %s uses printf to print numbers (%d) and
strings.'#10;
```

```
procedure printf(fm: pchar; args: array of const); cdecl;
external 'c';
begin
   printf(F,[P, 123]);
end.
```

Результат этой программы будет подобен следующему:

```
This example uses printf to print numbers (123) and strings.
```

В качестве альтернативы программа может иметь следующую конструкцию:

```
program testaocc;

Const
  P : Pchar =
        'example';
  F : Pchar =
        'This %s uses printf to print numbers (%d) and
strings.'#10;

procedure printf(fm: pchar); cdecl; varargs; external 'c';
begin
   printf(F, P, 123);
end.
```

Модификатор `varargs` сигнализирует компилятору, что функция позволяет меняющееся количество параметров *(обозначается многоточием в C)*.

7.2. Создание библиотек

Free Pascal поддерживает создание общедоступных или статических библиотек в простой и доступной форме. Если вы хотите создать статические библиотеки для других программистов Free Pascal, вам необходимо только установить переключатель командной строки. Для создания общедоступных библиотек, читайте раздел 12.1. Введение[156]. Если вы хотите, чтобы программисты C могли использовать ваш код, вам нужно немного адаптировать свой код. Этот процесс описан вначале.

7.2.1. Экспорт функций

При экспорте функций из библиотек, вы должны учитывать две вещи:

1. Соглашение о вызовах
2. Схема именования

Соглашения о вызовах управляются модификаторами `cdecl`, `popstack`, `pascal`, `safecall`, `stdcall` и `register`. См. раздел 6.3. Механизм вызова[93], где представлена информация о различных схемах вызовов.

Соглашение об именованиях управляется двумя модификаторами в случае статических библиотек:

- `cdecl`
- `alias`

Больше информации о том, как эти различные модификаторы изменяют имя процедуры вы найдёте в разделе 6.2. Преобразование имён[89].

ПРИМЕЧАНИЕ

Если в вашем модуле вы используете функции, которые содержатся в другом модуле, то для C-программы также будет необходимо связать в объектных файла эти модули.

7.2.2. Экспорт переменных

Подобно тому, как вы экспортируете функции, вы можете экспортировать переменные. Для объявления переменной, которая должна быть использована в C-программе, нужно её объявлять с модификатором `cvar`:

```
var MyVar : MyTpe; cvar;
```

Это укажет компилятору, что ассемблерное имя переменной *(той, которая используется в C -программе)* должно быть точно таким, как указано в объявлении, то есть чувствительным к регистру.

Не допускается объявлять несколько переменных как `cvar` в одном операторе, то есть следующий код будет ошибочным:

```
var Z1,Z2 : longint; cvar;
```

7.2.3. Компиляция библиотек

Для создания библиотеки нужно использовать ключевое слово library в головном файле *(файле проекта)*. Дополнительная информация о создании библиотек в разделе 12. ПРОГРАММИРОВАНИЕ ОБЩЕДОСТУПНЫХ БИБЛИОТЕК[156].

При компиляции модуля, компилятор создаёт объектные файлы *(с расширением .o)*, они являются обычными объектными файлами, как и получаемые с помощью компилятора C. Их можно объединять с помощью инструментов ar и ranlib в статические библиотеки. Тем не менее этого ***не следует*** делать *(по разным причинам)*.

- В коде будет много ссылок на скомпилированные внутренние процедуры RTL. Этих процедур нет в библиотеке C.
- Секции инициализации не вызываются программой на C.
- Переменные потоков не будет выделены *(или инициализированы)*.
- Не будут инициализированы строки ресурсов.
- В каждой библиотеке будет сделана инициализация RTL.

Чтобы учесть эти *(и другие)* проблемы требует глубоко знать о внутренней работе компилятора и RTL, и по этому ***не следует*** использовать статические библиотеки с Free Pascal.

7.2.4. Стратегия поиска модуля

Когда вы компилируете модуль, по умолчанию компилятор всегда ищет файлы модуля.

Чтобы иметь возможность отличать модули, которые могут быть откомпилированы как статические или как динамические библиотеки, имеются два переключателя:

-XD: определяет идентификатор `FPC_LINK_DYNAMIC`
-XS: определяет идентификатор `FPC_LINK_STATIC`

Определение одного идентификатора автоматически отменяет другой.

Эти два переключателя могут быть использованы в сочетании с конфигурационным файлом fpc.cfg. Существование одного из этих идентификаторов может быть использовано для того, чтобы решить, какой установить путь поиска модуля. Например, на LINUX:

```
# Set unit paths
#IFDEF FPC_LINK_STATIC -Up/usr/lib/fpc/linuxunits/staticunits
#ENDIF
#IFDEF FPC_LINK_DYNAMIC -Up/usr/lib/fpc/linuxunits/sharedunits
#ENDIF
```

С таким конфигурационным файлом компилятор будет искать модули в различных каталогах, в зависимости от того, какой переключатель используется, `-XD` или `-XS`.

7.3. Использование умной компоновки

Вы можете компилировать ваши модули, используя умную компоновку. В этом случае компилятор создаёт серии блоков кода, которые будут настолько маленькими, насколько это возможно. То есть блок кода будет содержать только код для одной процедуры или функции.

Когда вы компилируете программу, которая использует модуль, созданный в режиме умной компоновки, то компилятор будет включать в код только те процедуры и функции, которые на самом деле необходимы, и будет пропускать весь остальной код. В результате получится небольшой двоичный файл, который будет быстрее загружаться в память, а, следовательно, быстрее выполняться.

Чтобы включить умную компоновку, нужно установить соответствующую опцию в командной строке: `-Cx`, или поместить директиву `{$SMARTLINK ON}`

в файле модуля:

```
Unit Testunit
{SMARTLINK ON}
Interface
…
```

Умная компоновка замедляет процесс компиляции, особенно для больших модулей.

В случае если модуль foo.pp использует умную компоновку, имя файла изменится на libfoo.a.

Технически говоря, компилятор делает небольшие ассемблерные файлы для каждой процедуры или функции в модуле, как и для всех глобальных переменных (находятся ли они в разделе `interface` или нет). Затем он собирает эти небольшие файлы и использует `ar` для помещения результирующих объектных файлов в один архив.

Умная компоновка и создание общедоступных *(или динамических)* библиотек являются взаимоисключающими, то есть если вы включите умную компоновку, то создание общедоступных библиотек отключится. Создание статических библиотек остаётся возможным. Причиной этого является то, что при создании динамических библиотек умная компоновка не имеет смысла. Динамическая библиотека в любом случае загружается в память целиком при помощи динамического компоновщика *(или операционной системы)*, так что вы не уменьшите размер, выполняя умную компоновку для динамической библиотеки.

8. ВОПРОСЫ ПАМЯТИ

8.1. Модель памяти

Компилятор Free Pascal генерирует **32**-разрядный или **64**-разрядный код. Это имеет несколько последствий:

- Вам нужен **32**-разрядный или **64**-разрядный процессор для запуска сгенерированного кода.
- Вам не нужно возиться с селекторами сегмента. Адресация памяти может быть организована с использованием одного **32**-разрядного *(для **32**-разрядных процессоров)* или **64**-разрядного *(для **64**-разрядных процессоров)* указателя. Объём памяти ограничен только доступным объёмом виртуальной памяти на вашей машине.
- Объявляемые вами структуры не ограничены в размерах. Массивы могут иметь такую длину, какая вам необходима. Вы можете запрашивать блоки памяти любого размера.

8.2. Форматы данных

Этот раздел содержит информацию об объёмах памяти, занимаемых различными типами данных Free Pascal. Также здесь предоставлена информация о внутреннем выравнивании.

8.2.1. Целочисленные типы

Размеры по умолчанию для хранения целочисленных типов приведены в документе Справочное руководство Free Pascal. В случае, если тип определён пользователем, размер определяется границами типа:

- Если границы находятся в диапазоне **-128..127**, то переменная сохраняется как `shortint` (***8**-разрядное число со знаком*).
- Если границы находятся в диапазоне **0..255**, то переменная сохраняется как `byte` (***8**-разрядное число без знака*).
- Если границы находятся в диапазоне **-32768..32767**, то переменная сохраняется как `smallint` (***16**-разрядное число со знаком*).
- Если границы находятся в диапазоне **0..65535**, то переменная сохраняется как `word` (***16**-разрядное число без знака*).
- Если границы находятся в диапазоне **0..4294967295**, то переменная сохраняется как `longword` (***32**-разрядное число без знака*).
- В остальных случаях переменная сохраняется как `longint` (***32**-разрядное число со знаком*).

8.2.2. Символьные типы

char, или поддиапазон типа char, сохраняется как байт. WideChar записывается как слово, то есть ***два байта***.

8.2.3. Логические типы

Тип Boolean хранится как байт и может принимать значения true или false.

Тип ByteBool хранится как байт, WordBool хранится как слово, а longbool хранится как longint.

8.2.4. Перечисляемые типы

По умолчанию все перечисления хранятся как longword (4 байта), что эквивалентно применению переключателя {$Z4}, {$PACKENUM 4} или {$PACKENUM DEFAULT}.

Поведение по умолчанию может быть изменено при помощи переключателей компилятора и установки его режима. В режиме компилятора tp, или с переключателями {$Z1} или {$PACKENUM 1}, пространство, используемое для хранения перечислений показано в таблице 8.1

Таблица 8.1. Пространство для хранения перечислений в режиме tp

Элементов в перечислении	Используемое место
0…255	byte (1 байте)
256…65535	word (2 байта)
> 65535	Longword (4 байта)

Если применяются переключатели {$Z2} или {$PACKENUM 2}, то значение записывается в **2** байта (word), если перечисление имеет количество элементов меньшее или равное **65535**. Если элементов больше, то значение перечисления записывается как **4**-байтное значение (longword).

8.2.5. Типы с плавающей точкой

Размеры типов с плавающей точкой варьируются в зависимости от типа процессора. За исключением архитектуры Intel, тип extended отображается в IEEE как двойной тип, если имеется аппаратный сопроцессор плавающей точки.

Типы плавающей точки имеют бинарный формат хранения, разделённый на три поля: мантисса, экспонента и знаковый бит, который хранит знак значения с

плавающей точкой.

Single

Тип `single` занимает **4** байта памяти, а его структура памяти такая же, как у типа IEEE-754 `single`. Только этот тип является типом, который гарантированно доступен на всех платформах *(либо эмулируется с помощью программного обеспечения, либо напрямую через аппаратное обеспечение)*.

Формат памяти для типа `single` подобен тому, который показан на рис. 8.1.

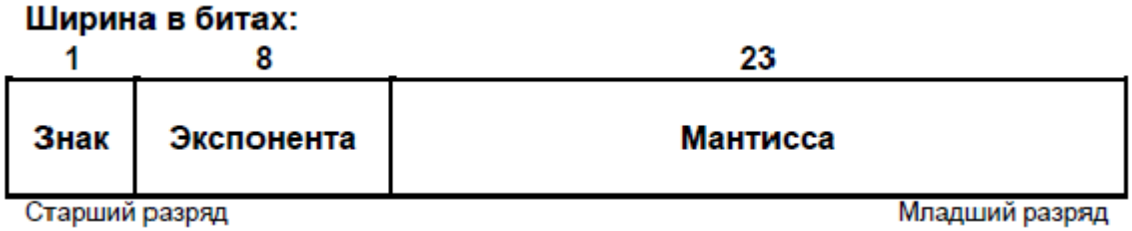

Рис. 8.1. Формат single.

Double

Тип `double` занимает **8** байтов памяти, а его структура памяти такая же, как у типа IEEE-754 `double`.

Формат памяти для типа `double` подобен тому, который показан на рис. 8.2.

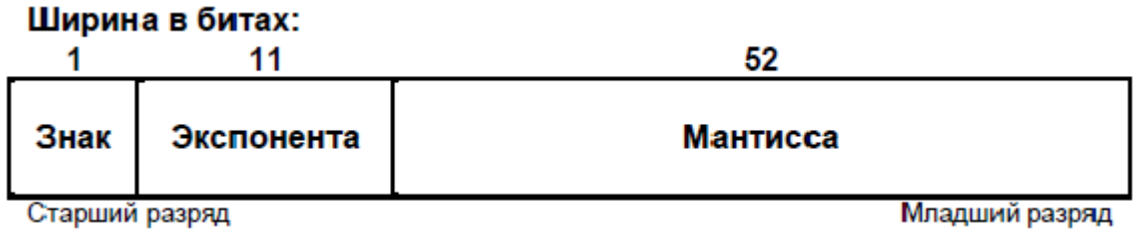

Рис. 8.2. Формат double.

На процессорах, которые не поддерживают операции сопроцессора (и которые имеют переключатель `{$E+}`), тип `double` не существует.

Extended

Для процессоров Intel 80x86, тип `extended` занимает до **10** байтов памяти. Подробности см. в руководстве программиста Intel.

Для всех других процессоров, которые поддерживают операции с плавающей точкой, тип `extended` – это псевдоним для типа, который поддерживает наибольшую точность, обычно это тип `double`. На процессорах, которые не поддерживают операции сопроцессора (и которые имеют переключатель `{$E+}`), тип `extended` обычно отображается как тип `single`.

Comp

Для процессоров Intel 80x86 тип comp содержит **63**-разрядное интегральное значение и знаковый бит *(в старшем разряде)*. Тип `comp` использует **8** байтов для хранения значения.

На других процессорах тип `comp` не поддерживается.

Real

В отличие от Turbo Pascal, где тип `real` имел специальный внутренний формат, на Free Pascal тип `real` просто преобразуется в один из других вещественных типов. Он преобразуется в `double` на процессорах, которые поддерживают операции с плавающей точкой, и преобразуется в `single` на процессорах, которые аппаратно не поддерживают операции с плавающей точкой. Подробности см. в таблице 8.2.

Таблица 8.2. Представление типа real в процессорах.

Процессор	Представление типа Real
Intel 80x86	`double`
Motorola 680x0 *(с переключателем `{$E-}`)*	`double`
Motorola 680x0 *(с переключателем `{$E+}`)*	`single`

8.2.6. Указатели

Тип `pointer` хранится как `longword` *(беззнаковое целое **32**-разрядное значение)* на **32**-разрядных процессорах и как **64**-разрядное беззнаковое целое значение на **64**-разрядных процессорах *(на самом деле это тип `qword`, который не поддерживается в Free Pascal v1.0)*.

8.2.7. Строки

Ansistring

Тип `ansistring` – это динамическая строка, которая не имеет ограничения по длине. Когда строка больше не ссылается на данные *(её счётчик ссылок равен нулю)*, то память автоматически освобождается. Если `ansistring` – это константа, то её счётчик ссылок будет равен **-1**, показывая, что она никогда не должна освобождаться. Структура памяти для `ansistring` показана в таблице

8.3.

Таблица 8.3. Структура памяти AnsiString (32-разрядная модель)

Смещение	Содержимое
-8	`Longint` с действительным размером строки
-4	`Longint` со счётчиком ссылок
0	Массив символов (`char`) с нулевым окончанием

Shortstring

Тип `shortstring` занимает столько байт, сколько требует его максимальная длина плюс ещё один. Первый байт содержит текущую динамическую длину строки. Следующие байты содержат данные *(символы типа `char`)* строки. Максимальный размер `shortstring` ограничен размером байта, то есть **255** символов.

Widestring

`widestring` размещается в куче, также как и `ansistring`. В отличие от `ansistring`, `widestring` занимает **2** байта на символ, и завершается двойным нулём.

8.2.8. Множества

Множество хранится как массив битов, где каждый бит показывает, находится ли элемент во множестве или исключён из него. Максимальное количество элементов во множестве **256**.

Если множество имеет менее **32** элементов, оно закодировано как беззнаковое **32**-разрядное значение. Иначе оно кодируется как массив из **8** беззнаковых **32**-разрядных значений *(`longwords`)*, и, следовательно, имеет размер **256** байт.

Количество элементов во множестве `E` определяется как:

```
LongwordNumber = (E div 32);
```

А количество битов в пределах 32-разрядного значения определяется как:

```
BitNumber = (E mod 32);
```

8.2.9. Статические массивы

Статический массив хранится как непрерывная последовательность переменных компонентов массива. Компоненты с младшими индексами записываются в память первыми. Между каждым элементом массива не выполняется выравнивание. В многомерном массиве первым в память

записывается крайний правый элемент.

8.2.10. Динамические массивы

Динамический массив хранится как указатель на блок памяти в куче. Память в куче является непрерывной последовательностью переменных компонентов массива, как и для статического массива. Счётчик ссылок и размер памяти записываются в память только перед действительным началом использования массива в отрицательном смещении относительно адреса указателя на массив. Они не должны использоваться.

8.2.11. Записи

Каждое поле записи хранится как непрерывная последовательность переменных, где первое поле сохраняется с младшим адресом в памяти. В случае вариантных полей в записи, каждый вариант начинается с того же адреса в памяти. Поля записи обычно выравниваются, если указана директива `packed` при объявлении типа `record`.

Подробно о выравнивании полей см. в разделе 8.3.2. Выравнивание структурированных типов[121].

8.2.12. Объекты

Объекты хранятся в памяти как обычные записи с дополнительным полем: указателем на таблицу виртуальных методов – Virtual Method Table *(VMT)*. Это поле записывается первым, а все поля объекта записываются в том порядке, в каком они объявлены (с возможным выравниванием поля адреса, если объект был объявлен как `packed`).

VMT инициализируется путём вызова метода объекта `Constructor`.Если для вызова конструктора был использован оператор `new`, то поля данных объекта будут храниться в динамической памяти *(куче)*, иначе они будут непосредственно записаны в раздел данных исполняемого файла.

Если объект не имеет виртуальных методов, указатель на VMT не устанавливается.

Объём выделяемой памяти выглядит так, как показано в таблице 8.4.

Таблица 8.4. Распределение памяти для объекта (32-разрядная модель)

Смещение	Содержимое
+0	Указатель на VMT *(не обязательно)*
+4	Данные. Все поля в порядке объявления

…	

Таблица виртуальных методов (VMT) для каждого объектного типа состоит из двух контрольных полей *(содержащих размер данных)*, указатель на VMT объекта-предка *(Nil, если предка нет)*, а затем указатели на все виртуальные методы. Макет VMT показан в таблице 8.5. VMT создаётся компилятором.

Таблица 8.5. Распределение памяти для VMT объекта *(32-разрядная модель)*

Смещение	Содержимое
+0	Размер данных объектного типа.
+4	Минус размер данных объектного типа. Позволяет определить правильность указателей VMT.
+8	Указатель на VMT предка, Nil, если нет доступных предков.
+12	Указатели на виртуальные методы.
…	

8.2.13. Классы

Подобно объектам, классы хранятся в памяти как обычные записи с дополнительным полем: указателем на таблицу виртуальных методов – Virtual Method Table *(VMT)*. Это поле записывается первым, а все поля класса записываются в том порядке, в каком они объявлены.

В отличие от объекта, все поля класса всегда хранятся в динамической памяти.

Объём выделяемой памяти выглядит так, как показано в таблице 8.6.

Таблица 8.6. Распределение памяти для класса *(32/64-разрядная модель).*

Смещение		Содержимое
32 бит	**64 бит**	
+0	+0	Указатель на VMT
+4	+8	Данные. Все поля в порядке объявления
…		

Таблица виртуальных методов (VMT) для каждого класса состоит из нескольких полей, которые используются для информации во время выполнения. Макет VMT показан в таблице 8.7. VMT создаётся компилятором.

Таблица 8.7. Распределение памяти для VMT класса (32/64-разрядная

модель).

Смещение		Содержимое
32 бит	**64 бит**	
+0	+0	Размер данных объектного типа.
+4	+8	Минус размер данных объектного типа. Позволяет определить правильность указателей VMT.
+8	+16	Указатель на VMT предка, `Nil`, если нет доступных предков.
+12	+24	Указатель на имя класса (хранится как `shortstring`)
+16	+32	Указатель на динамическую таблицу методов *(использующую `message` с целыми числами)*.
+20	+40	Указатель на таблицу определения методов.
+24	+48	Указатель на таблицу определения полей.
+28	+56	Указатель на таблицу типа информации.
+32	+64	Указатель на таблицу инициализации экземпляра.
+36	+72	Зарезервировано.
+40	+80	Указатель на таблицу интерфейса.
+44	+88	Указатель на таблицу динамических методов *(использующую `message` со строками)*.
+48	+96	Указатель на деструктор `Destroy`.
+52	+104	Указатель на метод `NewInstance`.
+56	+112	Указатель на метод `FreeInstance`.
+60	+120	Указатель на метод `SafeCallException`.
+64	+128	Указатель на метод `DefaultHandler`.
+68	+136	Указатель на метод `AfterConstruction`.
+72	+144	Указатель на метод

		BeforeDestruction.
+76	+152	Указатель на метод DefaultHandlerStr.
+80	+160	Указатель на метод Dispatch.
+84	+168	Указатель на метод DispatchStr.
+88	+176	Указатель на метод Equals.
+92	+184	Указатель на метод GetHashCode.
+96	+192	Указатель на метод ToString.
+100	+200	Указатели на другие виртуальные методы.
…		

Посмотрите файл rtl / inc / objpash.inc для самой последней информации VMT.

8.2.14. Файлы

Файловые типы представлены как записи. Типизированные и нетипизированные файлы представлены как фиксированные записи:

```
Const
   PrivDataLength=3*SizeOf(SizeInt) + 5*SizeOf(pointer);
Type
   filerec = packed record
   handle : THandle;
   mode : longint;
   recsize: Sizeint;
   _private : array[1..PrivDataLength] of byte;
   userdata : array[1..32] of byte;
   name : array[0..filerecnamelength] of char;
End;
```

Текстовые файлы описываются, используя следующую запись:

```
TextBuf = array[0..255] of char;
textrec = packed record
   handle: THandle;
   mode: longint;
   bufsize : SizeInt;
   _private: SizeInt;
   bufpos: SizeInt;
   bufend: SizeInt;
   bufptr: ^textbuf;
   openfunc: pointer;
   inoutfunc : pointer;
   flushfunc : pointer;
   closefunc : pointer;
   userdata: array[1..32] of byte;
   name: array[0..255] of char;
```

```
    LineEnd : TLineEndStr;
    buffer: textbuf;
End;
```

handle

Поле `handle` возвращает дескриптор файла (если файл открыт), как возвращённый операционной системой.

mode

Поле mode может принимать одно из нескольких значений. Если оно равно `fmclosed`, то файл закрыт, а дескриптор файла является неправильным. Если значение равно `fminput`, то это указывает на то, что файл открыт только для чтения. `fmoutput` указывает, что файл открыт только для записи, а `fminout` указывает, что файл открыт как для записи, так и для чтения.

name

Поле `name` – это строка символов с нулевым окончанием, которая представляет имя файла.

userdata

Поле `userdata` никогда не используется процедурами обработки файлов Free Pascal, и может быть использовано для специальных задач программным обеспечением разработчика.

8.2.15. Процедурные типы

Процедурный тип хранится как обычный указатель, который содержит адрес процедуры.

Процедурный тип нормальной процедуры или функции хранится как обычный указатель, который содержит адрес точки входа процедуры.

В случае метода процедурного типа, запись состоит из двух указателей, первый содержит точку входа метода, а второй содержит указатель на `self` *(экземпляр объекта)*.

8.3. Выравнивание данных

8.3.1. Выравнивание типизированных переменных и констант

Все статические данные *(переменные и типизированные константы)*, размер которых более одного байта, обычно выровнены по границе с множителем **2**. Это выравнивание применяется только для начального адреса переменных, и не применяется для полей в пределах структур или объектов. Больше

информации о выравнивании вы можете найти в разделе 8.3.2. Выравнивание структурированных типов[121]. Выравнивание является одинаковым для всех целевых процессоров.

Таблица 8.8. Выравнивание данных.

Размер данных ***(в байтах)***	**Выравнивание** ***(наименьший размер)***	**Выравнивание** ***(быстрая компиляция)***
1	1	1
2-3	2	2
4-7	2	4
8+	2	4

В столбцах выравнивания указано выравнивание адреса переменной, то есть начальный адрес переменной будет выровнен по этой границе. Выравнивание с наименьшим размером является действительным, если выполняется оптимизация по размеру *(опция компилятора `-Og`)*, а не по скорости, иначе «быстрое» выравнивание используется для выравнивания данных *(по умолчанию)*.

8.3.2. Выравнивание структурированных типов

По умолчанию все элементы в структуре выровнены по границе **2** байта, если только директива `$PACKRECORDS` или модификатор `packed` не используются для определения другого вида выравнивания. Например, `запись` или `объект`, имеющие однобайтный элемент, будут иметь размер, округлённый до **2** байтов, так что в действительности структура будет иметь размер **2** байта.

8.4. Куча

Динамическая память *(куча)* используется для хранения всех динамических переменных и экземпляров класса. Интерфейс для кучи такой же, как и в Turbo Pascal и Delphi, хотя последствия могут быть не одинаковыми. Куча является «потокобезопасной», поэтому выделение памяти из различных потоков не является проблемой.

8.4.1. Стратегия выделения динамической памяти

Куча – это память, структура которой организована в виде стека. «Дно» кучи хранится в переменной `HeapOrg`. Изначально указатель кучи (`HeapPtr`) ссылается на дно кучи. Если переменной выделяется динамическая память в куче, то `HeapPtr` увеличивается на размер выделенного блока памяти. Это имеет эффект «складывания» динамических переменных друг на друга.

Каждый раз, когда выделяется блок памяти, его размер нормализуется, чтобы он был кратен **16** *(или **32** на **64**-разрядных системах)* байтам.

Когда вызываются `Dispose` или `FreeMem` для освобождения блока памяти, который находится не на вершине кучи, то куча становится фрагментированной. Процедуры освобождения памяти также добавляют освобождённые блоки к `freelist`, который в действительности является связанным списком свободных блоков. Кроме того, если освобождённый блок имел размер менее 8 КБ, то список свободной кэш-памяти также обновляется.

Список свободной кэш-памяти в действительности является КЭШем свободных блоков кучи, которые имеют определённую длину *(скорректированный размер блока, делённый на **16**, даёт индекс в таблице*

списка свободной кэш-памяти). Это является более быстрым доступом, чем поиск через весь список `freelist`.

Формат записи в `freelist` является следующим:

```
PFreeRecord = ^TFreeRecord;
TFreeRecord = record
   Size : longint;
   Next : PFreeRecord;
   Prev : PFreeRecord;
end;
```

Поле `Next` указывает на следующий свободный блок, в то время как поле `Prev` указывает на предыдущий свободный блок.

Алгоритм выделения памяти следующий:

1. Размер блока для выделения кратен **16** *(или 32)*
2. Список свободного КЭШа просматривается для поиска свободного блока указанного или большего размера, если такой блок найден, то память выделяется и выполняется выход из процедуры
3. `freelist` просматривается для поиска свободного блока указанного или большего размера, если такой блок найден, то память выделяется и выполняется выход из процедуры
4. Если в `freelist` свободный блок не найден, то куча увеличивается для выделения указанного размера памяти, и выполняется выход из процедуры
5. Если куча не может быть увеличена, то библиотека времени выполнения генерирует ошибку **203**

8.4.2. Увеличение кучи

Куча получает память от операционной системы по мере необходимости.Память ОС запрашивается в блоках: сначала выполняется попытка получить кусок в **64** КБ, если выделяемый размер меньше **64** КБ, или **256** КБ или **1024** КБ иначе. Если эта попытка не удаётся, то выполняется попытка увеличить кучу на столько, сколько памяти вы запросили из кучи.

Если попытка зарезервировать память у ОС не удалась, то возвращаемое значение зависит от значения глобальной переменной `ReturnNilIfGrowHeapFails`. Значения этой переменной приведены в таблице 8.9.

Таблица 8.9. Значения ReturnNilIfGrowHeapFails

Значение ReturnNilGrowHeapFails	Действие по умолчанию менеджера памяти
`FALSE` *(по умолчанию)*	Генерируется ошибка времени выполнения **203**
`TRUE`	`GetMem`, `ReallocMem` и `New`

	возвращают nil

ReturnNilIfGrowHeapFails может быть установлена для изменения поведения по умолчанию обработчика ошибок менеджера памяти.

8.4.3. Отладка кучи

Free Pascal предоставляет модуль, который позволяет вам отследить выделение и освобождение памяти в куче: heaptrc.

Если в командной строке вы укажете переключатель -gh, или подключите heaptrc как первый модуль в вашем разделе uses, то менеджер памяти будет фиксировать все выделения и освобождения памяти, и при выходе из вашей программы результаты будут отправлены на стандартный выход.

Подробнее об использовании механизма heaptrc см. в Справочник пользователя Free Pascal и в Справочное руководство Free Pascal.

8.4.4. Написание вашего собственного менеджера памяти

Free Pascal позволяет вам написать и использовать ваш собственный менеджер памяти. Стандартные функции GetMem, FreeMem, ReallocMem и т.п. специальную запись в модуле system для управления памятью. Модуль system инициализирует эту запись с собственным менеджером памяти модуля system, но вы можете прочитать и установить эту запись, используя вызов GetMemoryManager и SetMemoryManager:

```
procedure GetMemoryManager(var MemMgr: TMemoryManager);
procedure SetMemoryManager(const MemMgr: TmemoryManager);
```

Запись TMemoryManager определена следующим образом:

```
TMemoryManager = record
   NeedLock     : Boolean;
   Getmem       : Function(Size:PtrInt):Pointer;
   Freemem      : Function(var p:pointer):PtrInt;
   FreememSize  : Function(var p:pointer;Size:PtrInt):PtrInt;
   AllocMem     : Function(Size:PtrInt):Pointer;
   ReAllocMem   : Function(var p:pointer;Size:PtrInt):Pointer;
   MemSize      : function(p:pointer):PtrInt;
   InitThread        : procedure;
   DoneThread        : procedure;
   RelocateHeap      : procedure;
   GetHeapStatus     : function :THeapStatus;
   GetFPCHeapStatus  : function :TFPCHeapStatus;
end;
```

Как вы можете видеть, элементы этой записи в основном являются процедурными переменными. Модуль system не делает ничего, кроме вызова этих переменных, когда вы выделяете или освобождаете память.

Каждое из этих полей ссылается на соответствующий вызов в модуле system. Эти вызовы описаны ниже.

NeedLock

Этот флаг указывает, будет ли менеджер памяти нуждаться в блокировке: если менеджер памяти сам не является «потокобезопасным», то этот флаг можно установить в `True` и процедуры работы с памятью будут использовать блокировку для всех процедур работы с памятью. Если это поле установлено в `False`, блокировка не используется.

Getmem

Эта функция выделяет новый блок в куче. Блок должен иметь размер, указанный в `Size`. Возвращаемое значение – это указатель на вновь выделенный блок.

Freemem

Должна освободить ранее выделенный блок. Указатель `P` указывает на ранее выделенный блок. Менеджер памяти должен иметь механизм для определения размера освобождаемого блока памяти *(например, путём записи его размера в отрицательное смещение)*. Возвращаемое значение является не обязательным, и может быть использовано для возврата размера освобождённой памяти.

FreememSize

Эта функция должна освободить память, указанную при помощи `P`. Аргумент `Size` – это ожидаемый размер блока памяти, на который ссылается указатель `P`. Её не следует принимать во внимание, но можно использовать для проверки поведения программы.

AllocMem

То же, что и `getmem`, только выделенная память должна быть заполнена нулями перед возвратом.

ReAllocMem

Должна выделить блок памяти с максимальным размером `Size` байт и заполнить его содержимым блока памяти, указанным в `P`, обрезав это содержимое по новому размеру при необходимости. После этого блок памяти, указанный в `P`, может быть освобождён. Возвращаемое значение является указателем на новый блок памяти. Учтите, что `P` может быть `Nil`, в этом случае поведения эквивалентно `GetMem`.

MemSize

Должна вернуть общий объём памяти, доступной для выделения. Эта функция может возвращать ноль, если менеджер памяти не позволяет определять это значение.

InitThread

Эта процедура вызывается, когда запускается новый поток: она должна инициализировать структуру кучи для текущего потока *(если таковой имеется)*.

DoneThread
Эта процедура вызывается при закрытии потока: она должна очищать все структуры кучи для текущего потока.

RelocateHeap
Реструктурирует кучу – это только для локальных куч потоков.

GetHeapStatus
Должна возвращать запись `THeapStatus` с состоянием менеджера памяти. Эта запись должна заполняться значениями, совместимыми с Delphi.

GetHeapStatus
Должна возвращать запись `TFPCHeapStatus` с состоянием менеджера памяти. Эта запись должна заполняться значениями, совместимыми с FPC.

Чтобы реализовать ваш собственный менеджер памяти, достаточно создать такую запись и выполнить вызов `SetMemoryManager`.

Чтобы избежать конфликтов с системным менеджером памяти, настройка менеджера памяти должна произойти как можно скорее в разделе инициализации вашей программы, то есть перед любым обращением к `getmem`.

Это означает, что модуль, реализующий менеджер памяти, должен быть первым в разделе uses вашей программы или библиотеки, так как он будет инициализирован перед всеми другими модулями *(конечно, за исключением самого модуля system)*.

Это также означает, что невозможно использовать модуль heaptrc совместно с пользовательским менеджером памяти, так как модуль heaptrc использует системный менеджер памяти для работы с памятью. Поместив модуль heaptrc после модуля, реализующего ваш менеджер памяти, вы перезапишите запись вашего менеджера памяти, и наоборот.

Следующий пример реализует простой пользовательский менеджер памяти, используя менеджер памяти библиотеки C. Он распространяется в виде пакета с Free Pascal.

```
unit cmem;

interface

Const
  LibName = 'libc';

Function Malloc (Size : ptrint) : Pointer; cdecl; external
LibName name 'malloc';
Procedure Free (P : pointer); cdecl; external LibName name
'free';
function ReAlloc (P : Pointer; Size : ptrint) : pointer; cdecl;
```

```
 external LibName name 'realloc';
Function CAlloc (unitSize,UnitCount : ptrint) : pointer; cdecl;
 external LibName name 'calloc';

implementation

type
  pptrint = ^ptrint;

Function CGetMem (Size : ptrint) : Pointer;
begin
  CGetMem:=Malloc(Size+sizeof(ptrint));
  if (CGetMem <> nil) then
    begin
      pptrint(CGetMem)^ := size;
      inc(CGetMem,sizeof(ptrint));
    end;
end;

Function CFreeMem (P : pointer) : ptrint;
begin
  if (p <> nil) then
   dec(p,sizeof(ptrint));
   Free(P);
   CFreeMem:=0;
end;

Function CFreeMemSize(p:pointer;Size:ptrint):ptrint;
begin
  if size<=0 then
    begin
     if size<0 then
     runerror(204);
     exit;
    end;
  if (p <> nil) then
    begin
      if (size <> pptrint(p-sizeof(ptrint))^) then
      runerror(204);
    end;
  CFreeMemSize:=CFreeMem(P);
end;

Function CAllocMem(Size : ptrint) : Pointer;
begin
  CAllocMem:=calloc(Size+sizeof(ptrint),1);
  if (CAllocMem <> nil) then
    begin
      pptrint(CAllocMem)^ := size;
      inc(CAllocMem,sizeof(ptrint));
    end;
end;

Function CReAllocMem (var p:pointer;Size:ptrint):Pointer;
```

```
begin
  if size=0 then
  begin
    if p<>nil then
      begin
        dec(p,sizeof(ptrint));
        free(p);
        p:=nil;
      end;
  end
 else
  begin
    inc(size,sizeof(ptrint));
    if p=nil then
      p:=malloc(Size)
 else
  begin dec(p,sizeof(ptrint)); p:=realloc(p,size); end;
   if (p <> nil) then
    begin
      pptrint(p)^ := size-sizeof(ptrint);
      inc(p,sizeof(ptrint));
    end;
  end;
  CReAllocMem:=p;
end;

Function CMemSize (p:pointer): ptrint;
begin
 CMemSize:=pptrint(p-sizeof(ptrint))^;
end;

function CGetHeapStatus:THeapStatus;
var
res: THeapStatus;
begin
 fillchar(res,sizeof(res),0);
 CGetHeapStatus:=res;
end;

function CGetFPCHeapStatus:TFPCHeapStatus;
begin
   fillchar(CGetFPCHeapStatus,sizeof(CGetFPCHeapStatus),0);
end;

Const
   CMemoryManager : TMemoryManager =
  (
      NeedLock : false;
      GetMem : @CGetmem;
      FreeMem : @CFreeMem;
      FreememSize : @CFreememSize;
      AllocMem : @CAllocMem;
      ReallocMem : @CReAllocMem;
      MemSize : @CMemSize;
```

```
        InitThread : Nil;
        DoneThread : Nil;
        RelocateHeap : Nil;
        GetHeapStatus : @CGetHeapStatus;
        GetFPCHeapStatus: @CGetFPCHeapStatus;
    );

Var
    OldMemoryManager : TMemoryManager;

Initialization
    GetMemoryManager (OldMemoryManager);
    SetMemoryManager (CmemoryManager);

Finalization
    SetMemoryManager(OldMemoryManager);
end.
```

8.5. Использование памяти DOS под расширителем Go32

Поскольку Free Pascal для DOS – это **32**-битный компилятор и используется расширитель DOS, доступ к памяти DOS является нетривиальным. Далее мы попытаемся объяснить, как получить доступ к памяти и использовать DOS или память в реальном режиме *(спасибо за разъяснения Thomas Schatzl, E-mail: tom_at_work@geocities.com)*.

В ***защищённом режиме*** память назначается через ***селекторы*** и ***смещения***. Вы можете думать, что селекторы в защищённом режиме эквивалентны сегментам.

В Free Pascal указатель представляет собой смещение в селекторе DS, который указывает на данные вашей программы.

Для доступа к памяти DOS *(в реальном режиме)* вам нужен селектор. Который указывает на память DOS. Модуль go32 предоставляет вам такой селектор: переменную DosMemSelector, которая называется соответственно.

Вы также можете выделить память в пространстве памяти DOS, используя функцию global_dos_alloc модуля go32. Эта функция будет выделять память в области видимости DOS.В качестве примера здесь приведена функция, которая выделяет память в реальном режиме DOS и возвращает пару селектор:смещение (**selector:offset**).

```
procedure dosalloc(var selector : word; var segment : word;
size : longint);
var result : longint;
begin
    result := global_dos_alloc(size);
    selector := word(result);
    segment := word(result shr 16);
end;
```

(Вам необходимо освободить эту память, используя функцию `global_dos_free function`*).*

Вы можете получить доступ к любому месту в памяти, используя селектор. Вы можете получить селектор, используя функцию:

```
function allocate_ldt_descriptors(count : word) : word;
```

а затем передать этот селектор в нужную точку физической памяти, используя функцию

```
function set_segment_base_address(d : word;s : longint) :
boolean;
```

Её длина может быть установлена с помощью функции:

```
function set_segment_limit(d : word;s : longint) : boolean;
```

Вы можете управлять памятью, указанной в селекторе, используя функции модуля GO32. После использования селектора вы должны освободить его, снова используя функцию:

```
function free_ldt_descriptor(d : word) : boolean;
```

Подробности см. в описании модуля go32.

8.6. При переносе кода Турбо Паскаль

Тот факт, что 16-разрядный код больше не используется, означает, что некоторые старые конструкции и функции Turbo Pascal устарели. Ниже приведён список функций, которые больше не должны использоваться:

Seg() :
Возвращает сегмент адреса памяти. Так как сегменты больше не используются, то возвращает ноль в Free Pascal реализации `Seg`.

Ofs() :
Возвращает смещение адреса памяти. Так как сегменты больше не используются, то эта функция в Free Pascal возвращает полный адрес. Как следствие, возвращаемый тип `longint` или `int64` вместо `Word`.

Cseg(), Dseg() :
Возвращает, соответственно, сегмент кода и сегмент данных вашей программы. В Free Pascal эти функции возвращают ноль, так как и код и данные находятся в одной области памяти.

Ptr :
Принимает сегмент и смещение из адреса, и возвращает указатель на этот адрес. Она была изменена в библиотеке реального времени *(run-time library)* и сейчас просто возвращает смещение.

memw и mem :

Эти массивы дают доступ к памяти DOS. Free Pascal поддерживает их на платформе go32v2, они отображаются в пространстве памяти DOS. Для их использования вам необходим модуль go32. На других платформах они не поддерживаются.

Вы не должны использовать эти функции, так как они очень трудно переносимы, они являются специфическими для DOS и процессора 80x86. Компилятор Free Pascal разработан для совместимости с другими платформами, поэтому ваш код должен быть максимально переносимым, а не для какой-то специфической системы. Конечно, кроме случаев, когда вы пишите какие-либо модули драйверов.

8.7. Memavail и Maxavail

Старые функции Turbo Pascal `MemAvail` и `MaxAvail` больше не доступны в Free Pascal, начиная с версии **2.0**. Причина этой несовместимости описана ниже. В современных операционных системах (DOS-расширитель GO32V2 попадает под определение **«современная»**, потому что он может использовать страничную память и запускаться в многозадачной среде) идеи о **«Свободной памяти»** не являются допустимыми для приложения.

Причины заключаются в следующем:

1. В одном такте процессора после запроса приложением у ОС, сколько памяти свободно, другое приложение может занять всю память.
2. Не понятно, что означает **«свободная память»**: включая файл подкачки, кэш-память диска (на современных ОС кэш-память диска может увеличиваться и уменьшаться), включая память, выделенную для других приложений, но которая может быть освобождена и т.п.

Таким образом, программы, использующие функции `MemAvail` и `MaxAvail`, должны быть переписаны так, чтобы больше не использовать эти функции, потому что это не имеет смысла на современных операционных системах. Имеется три возможности:

1. Использовать исключения, чтобы поймать ошибки нехватки памяти.
2. Установить глобальную переменную `ReturnNilIfGrowHeapFails` в `True` и проверить после этого каждое выделение памяти, где указатель отличается от `Nil`.
3. Не «париться» по этому поводу и объявить функцию для «чайников» с именем `MaxAvail`, которая всегда будет возвращать `High(LongInt)` (или какую-то другую константу).

9. СТРОКИ РЕСУРСОВ

9.1. Введение

Строки ресурсов. Прежде всего. Существуют для того, чтобы сделать более лёгкой интернационализацию приложений, вводя языковые конструкции, которые предоставляют единый способ обработки строковых констант.

Большинство приложений общаются с пользователем посредством вывода сообщений на графический экран или в консоль. Сохранение этих сообщений в специальные константы позволяет хранить их одинаковыми способами в отдельных файлах, которые могут использоваться для перевода. А интерфейс программиста существует для манипуляции актуальными значениями строковых констант во время выполнения программы, а с компилятором Free Pascal поставляется утилита для преобразования файлов со строками ресурсов в любой формат, нужный программисту. Эти вещи обсуждаются в следующих разделах.

9.2. Файл строковых ресурсов

Если компилируется модуль, содержащий раздел `resourcestring`, то компилятор выполняет два действия:

1. Генерирует таблицу, содержащую значения строк, как они объявлены в исходных кодах
2. Генерирует ***файл строк ресурсов (resource string file)***, который содержит имена всех строк вместе с их объявленными значениями

Такой подход имеет два преимущества: во-первых, значение строки всегда присутствует в программе. Если программист не позаботился о переводе строк, то значения по умолчанию всегда присутствуют в бинарном файле. Это также позволяет избежать необходимости указывать файл, содержащий строки. Во-вторых, имея все строки вместе в сгенерированном компилятором файле, вы можете быть уверены, что все эти стоки будут в одном месте *(вы можете иметь множество разделов `resourcestring` в одном модуле или программе)*, а, имея этот файл в известном формате, программист может выбирать способ интернационализации.

Для каждого компилируемого модуля. Содержащего раздел `resourcestring`, компилятор генерирует файл, который имеет имя модуля и расширение .rst. Формат этого файла следующий:

1. Пустая строка.
2. Строка, начинающаяся со знака решётки `(#)` и хэш-значение строки, которому предшествует текст `hash value =`.
3. Третья строка, содержащая имя строки ресурсов в формате `unit-`

name.constantname *(имямодуля.имяконстанты)*, все символы в нижнем регистре, за которой следует знак равенства и значение строки в формате, эквивалентном представлению строк в Pascal. Строка может продолжаться на следующей строке, в этом случае она читается как строковое выражение в Pascal *(со знаком «Плюс» в строке)*.

4. Ещё одна пустая строка.

Если модуль не содержит раздела resourcestring, то файл не создаётся.

Например, следующий модуль:

```
unit rsdemo;
{$mode delphi}
{$H+}
interface
resourcestring
First = 'First';
Second = 'A Second very long string that should cover more than
1 line';
implementation
end.
```

В результате компиляции сгенерирует файл строковых ресурсов:

```
# hash value = 5048740
rsdemo.first='First'
# hash value = 171989989
rsdemo.second='A Second very long string that should cover more
than 1 li'+ 'ne'
```

Хэш-значение вычисляется с помощью функции Hash. Она представлена в модуле objpas. Значение является таким же значением, которое использует механизм GNU gettext. Оно ни коим образом не является уникальным и может использоваться только для ускорения поиска.

Утилита rstconv, которая поставляется с компилятором Free Pascal, позволяет манипулировать этими файлами строковых ресурсов. В данный момент её можно использовать только для того, чтобы сделать файл .po, который может быть передан в программу GNU msgfmt. Если кто-то желает иметь другой формат *(приходят на ум файлы ресурсов Win32)*, то можно усовершенствовать программу rstconv таким образом, чтобы она могла генерировать другие типы файлов. Механизм GNU gettext был выбран потому, что он доступен на всех платформах и уже широко используется в Unix я является свободно распространяемым *(бесплатным)*. Так как команда Free Pascal не хочет ограничивать использование строковых ресурсов, формат .rst был выбран, чтобы обеспечить независимый метод, не ограниченный каким-либо инструментом.

Если вы используете строковые ресурсы в ваших модулях и хотите, чтобы люди могли переводить строки, вы должны предоставить файл строковых ресурсов. В настоящий момент нет какого-либо способа для извлечения их из файла модуля, хотя в принципе это возможно. Использование строковых

ресурсов не обязательно, программа может быть скомпилирована без них, но потом перевод строк будет невозможен.

9.3. Обновление таблиц строк

Для интернационализации вашей программы недостаточно выполнить её компиляцию с разделом `resourcestrings`. Во время выполнения программа должна инициализировать таблицы строк с правильными значениями для языка, который выберет пользователь. По умолчанию такая инициализация не выполняется. Все строки инициализируются с их объявленными значениями.

Модуль objpas предоставляет механизм для правильной инициализации строковых таблиц. Нет необходимости включать этот модуль в раздел `uses`, так как он загружается автоматически, если программа или модуль компилируются в режиме `Delphi` или `objfpc`. Так как для использования строковых ресурсов требуется один из этих режимов, то модуль в любом случае будет загружаться автоматически.

Строка ресурса сохраняется в таблицах, одна на модуль и одна на программу, если она также содержит раздел `resourcestring`. Каждый раздел `resourcestring` сохраняется со своим именем, хэш-значением, значением по умолчанию и текущим значением, всё как в `AnsiStrings`.Модуль objpas предоставляет методы для извлечения количества таблиц строковых ресурсов, количества строк в таблице и приведённой выше информации для каждой строки. Он также предоставляет метод для установки текущего значения строки.

Ниже приводятся объявления всех функций:

```
Function ResourceStringTableCount : Longint;
Function ResourceStringCount(TableIndex: longint): longint;
Function GetResourceStringName(TableIndex, StringIndex:
Longint): Ansistring;
Function GetResourceStringHash(TableIndex, StringIndex:
Longint): Longint;
Function GetResourceStringDefaultValue(TableIndex, StringIndex:
 Longint): AnsiString;
Function GetResourceStringCurrentValue(TableIndex, StringIndex:
 Longint): AnsiString;
Function SetResourceStringValue(TableIndex, StringIndex :
longint; Value: Ansistring): Boolean;
Procedure SetResourceStrings (SetFunction: TresourceIterator);
```

Следующие две функции существуют только для удобства:

```
Function Hash(S: AnsiString): longint;
Procedure ResetResourceTables;
```

Далее приводятся краткие описания назначения функций. Подробно они описаны в файле Справочное руководство Free Pascal.

ResourceStringTableCount возвращает количество таблиц строковых ресурсов в программе.

ResourceStringCount возвращает количество вхождений строковых ресурсов в данную таблицу *(таблицы обозначаются индексами с основанием **0**)*.

GetResourceStringName возвращает имя строкового ресурса в таблице ресурсов. Это имя модуля, точка (.) и имя строковой константы, всё в нижнем регистре. Строки обозначаются индексом с нулевым основанием.

GetResourceStringHash возвращает значение хэша строкового ресурса как вычисленное компилятором с помощью функции `Hash`.

GetResourceStringDefaultValue возвращает значение по умолчанию для строкового ресурса, то есть значение, которое появляется в объявлении строкового ресурса и которое сохранено в бинарном файле.

GetResourceStringCurrentValue возвращает текущее значение строкового ресурса, то есть значение, установленное при инициализации *(значение по умолчанию)*, или значение, установленное какой-либо процедурой инициализации.

SetResourceStringValue устанавливает текущее значение строкового ресурса. Эта функция должна быть вызвана для инициализации всех строк.

SetResourceStrings передача этой функции обратного вызова приведёт к тому, что обратный вызов будет вызывать строки одну за одной, устанавливая значение строки для возврата значения обратного вызова.

Две другие функции существуют только для удобства:

Hash может использоваться для вычисления хэш-значения строки. Значение хэша записывается в таблицы в результате выполнения этой функции и принимается как значение по умолчанию. Это значение вычисляется компилятором во время компиляции: это может ускорить трансляцию операций.

ResetResourceTables сбрасывает все строковые ресурсы в их значения по умолчанию. Она вызывается в коде инициализации модуля objpas.

Ниже приведён пример функции `Translate`, которая будет инициализировать все строковые ресурсы:

```
Var I,J : Longint;
    S : AnsiString;
begin
   For I:=0 to ResourceStringTableCount-1 do
```

```
  For J:=0 to ResourceStringCount(i)-1 do
     begin
       S:=Translate(GetResourceStringDefaultValue(I,J));
       SetResourceStringValue(I,J,S);
     end;
  end;
```

Конечно, возможны другие способы, а функция `Translate` может быть реализована различными путями.

9.4. GNU gettext

Модуль gettext предоставляет возможности интернационализации приложений при помощи утилит GNU gettext. Этот модуль поставляется с бесплатной библиотекой компонентов – Free Component Library *(FCL)*, которая может использоваться описанным ниже способом.

Для данного приложения должны быть выполнены следующие шаги:

1. Собрать все файлы строковых ресурсов и объединить их вместе.
2. Вызвать программу rstconv с файлом, полученным в результате выполнения первого пункта, то есть с одним файлом .po, содержащим строковые ресурсы программы.
3. Перевести файл .po, полученный в второго пункта на все требуемые языки.
4. Запустите программу msgfmt, форматирующую все файлы .po. В результате будет получен набор файлов .mo, которые могут распространяться с вашим приложением.
5. Вызовите метод `TranslateResourceStrings` модуля gettext, передав ему шаблон с местоположением файлов .mo, например,

   ```
   TranslateResourcestrings('intl/restest.%s.mo');
   ```

   где спецификатор `%s` будет заменён содержимым переменной среды окружения `LANG`. Это вызов должен выполняться при старте программы.

Пример программы имеется в исходниках FCL-base, в каталоге **`fcl-base/tests`**.

9.5. Предупреждения

В принципе, можно переводить все строковые ресурсы в любое время выполнения программы. Однако, эти изменения не связаны с другими строками. Изменение будет заметно только тогда, когда использована строковая константа.

Рассмотрим следующий пример:

```
Const
   help = 'With a little help of a programmer.';
Var
   A : AnsiString;
begin
   { lots of code }
   A:=Help;
   { Again some code}
   TranslateStrings;
   { More code }
```

После вызова `TranslateStrings` значение `A` останется неизменным. Это означает, что присваивание `A:=Help` должно быть выполнено снова для того, чтобы изменения стали заметны. Это важно, особенно программ GUI, которые имеют, например, меню. Для того, чтобы изменения в строковых ресурсах стали заметны, новые значения должны быть перегружены в программном коде меню…

10. ПРОГРАММИРОВАНИЕ ПОТОКОВ

10.1. Введение

Free Pascal поддерживает потоковое программирование: имеется языковая конструкция, доступная для хранения локальных потоков (`ThreadVar`), и кросс-платформенные низкоуровневые потоковые процедуры для операционных систем, которые поддерживают потоки.

Все процедуры для работы с потоками доступны в модуле system в виде менеджера потоков. Менеджер потоков должен выполнять некоторые основные процедуры, которые должна поддерживать RTL. Для Windows по умолчанию менеджер потоков встроен в модуль system. Для других платформ менеджер потоков должен подключаться непосредственно программистом. На системах, где доступны потоки posix, модуль cthreads управляет менеджером потоков, который использует библиотеку потоков C POSIX. Для таких систем не существует родной библиотеки pascal.

Хотя это не запрещено делать, не рекомендуется использовать специфические для конкретной системы потоковые процедуры: языковая поддержка для многопоточных программ не будет включена, это означает, что потоковые переменные не будут работать, менеджер кучи будет «глючить», что может привести к серьёзным программным ошибкам.

Если не будет поддержки потоков в «бинарнике», использование потоковых процедур или создание потока приведёт к ошибке времени выполнения **232**.

Для LINUX *(и других Unixes)* менеджер потоков C может быть включен вставкой модуля cthreads в блоке подключения модулей в программе. Если это не сделать, то программа с использованием потоков будет выдавать ошибку при старте. Необходимо, чтобы этот модуль был вставлен как можно раньше в списке модулей в блоке `uses`.

В будущем менеджер потоков может быть реализован без использования Libc.

В следующих разделах описано программирование потоков, защита доступа к общим данным всех используемых критических секций потоков (кросс-платформенная). В заключении более подробно будет описан менеджер потоков.

10.2. Программирование потоков

Для запуска нового потока должна использоваться функция `BeginThread`. Она имеет один обязательный параметр: функцию, которая будет выполнена в новом потоке. Результатом функции является выход результата потока. В

потоковую функцию может быть передан указатель, который может быть использован для доступа к данным инициализации: программист должен убедиться, что данные будут доступны из потока и не выходят из области видимости потока раньше, чем поток обратится к ним.

```
Type
   TThreadFunc = function(parameter : pointer) : ptrint;
function BeginThread(sa : Pointer;
                     stacksize : SizeUInt;
                     ThreadFunction : tthreadfunc;
                     p : pointer;
                     creationFlags : dword;
                     var ThreadId : TThreadID) : TThreadID;
```

Это полная форма функции, возможен также вызов функции в упрощённой форме:

```
function BeginThread(ThreadFunction : tthreadfunc) : TThreadID;
function BeginThread(ThreadFunction : tthreadfunc;p :
pointer) : TthreadID;
function BeginThread(ThreadFunction : tthreadfunc;p : pointer;
var ThreadId : TThreadID) : TthreadID;
function BeginThread(ThreadFunction : tthreadfunc;p : pointer;
var ThreadId : TThreadID;const stacksize: SizeUInt) :
TthreadID;
```

Параметры имеют следующие значения:

ThreadFunction – это функция, которая должна быть выполнена в потоке.

p – если имеется, то указатель `p` будет помещён в потоковую функцию, когда она начнёт выполняться. Если `p` не указан, то помещается `Nil`.

ThreadID – если `ThreadID` указан, то в него будет записан ID потока.

stacksize – если указан, то этот параметр определяет размер стека, используемого потоком.

sa – сигнализирует о действии. Имеет значение только для LINUX.

creationflags – это специфический для системы флаг создания. Имеет значение только для windows и OS/2.

Вновь запущенный поток будет работать до тех пор, пока не будет выполнен выход из функции `ThreadFunction`, или пока не будет явно вызвана функция `EndThread`:

```
procedure EndThread(ExitCode : DWord);
procedure EndThread;
```

`exitcode` может быть проанализирован код, который запустил поток.

Ниже приведён небольшой пример потоковой программы:

```
{$mode objfpc}
uses sysutils {$ifdef unix},cthreads{$endif} ;
```

```
const threadcount = 100;
 stringlen = 10000;

var
 finished : longint;

threadvar
 thri : ptrint;

function f(p : pointer) : ptrint;
var s : ansistring;
begin
   Writeln('thread ',longint(p),' started');
   thri:=0;
   while (thri<stringlen) do
     begin
      s:=s+'1';
      inc(thri);
     end;
   Writeln('thread ',longint(p),' finished');
   InterLockedIncrement(finished);
   f:=0;
end;
var i : longint;

begin
   finished:=0;
   for i:=1 to threadcount do
      BeginThread(@f,pointer(i));
   while finished<threadcount do ;
   Writeln(finished);
end.
```

InterLockedIncrement – это потокобезопасная версия стандартной функции Inc.

Для предоставления системно-независимой поддержки программирования потоков реализованы некоторые полезные функции манипулирования потоками. Для использования этих функций идентификатор потока должен быть получен при запуске потока, потому что большинству функций требуется идентификатор потока, с которым они должны работать:

```
function SuspendThread(threadHandle: TThreadID): dword;
function ResumeThread(threadHandle: TThreadID): dword;
function KillThread(threadHandle: TThreadID): dword;
function WaitForThreadTerminate(threadHandle: TThreadID;
TimeoutMs : longint): dword;
function ThreadSetPriority(threadHandle: TThreadID;Prio:
longint): boolean;
function ThreadGetPriority(threadHandle: TThreadID): Integer;
function GetCurrentThreadId: dword;
procedure ThreadSwitch;
```

Смысл этих функций должен быть понятен:

SuspendThread – приостанавливает выполнение потока.

ResumeThread – возобновляет выполнение приостановленного потока.

KillThread – уничтожает поток: поток удаляется из памяти.

WaitForThreadTerminate – ожидает завершения потока. Функция возвращает результат, когда выполнение потока завершено или время ожидания истекло.

ThreadSetPriority – задаёт приоритет выполнения потока. Этот вызов не всегда допустим: ваш процесс может не иметь для этого достаточных прав.

ThreadGetPriority – возвращает текущий приоритет выполнения потока.

GetCurrentThreadId – возвращает идентификатор текущего потока.

ThreadSwitch – разрешает выполнение других потоков в данный момент. Это означает, что она может переключать потоки, но это не гарантируется, так как зависит от операционной системы и количества процессоров.

10.3. Критические разделы

При программировании потоков иногда необходимо избегать одновременного доступа к некоторым ресурсам, или одновременного вызова подпрограммы из двух потоков. Это может быть сделано с использованием Critical Section *(критический раздел)*. Менеджер кучи FPC использует критические разделы, если включена многопоточность.

Тип `TRTLCriticalSection` – это непрозрачный *(Opaque)* тип. Он зависит от операционной системы, на которой выполняется код. Он должен быть инициализирован перед первым использованием, и должен быть освобождён, когда в нём больше нет необходимости.

Для защиты участка кода должен быть сделан вызов `EnterCriticalSection`: когда этот вызов возвращает результат, это гарантирует, что текущий поток – это единственный поток, выполняемый в последующем коде. Для гарантии этого данный вызов может приостановить текущий поток на некоторое время.

Когда защищённый код завершится, должна быть вызвана `LeaveCriticalSection`: она включает другие потоки и разрешает их выполнение в защищённом коде. Чтобы минимизировать время ожидания для других потоков, важно создавать и удерживать защищённый блок как можно меньше.

Объявления этих подпрограмм следующие:

```
procedure InitCriticalSection(var cs: TRTLCriticalSection);
procedure DoneCriticalSection(var cs: TRTLCriticalSection);
procedure EnterCriticalSection(var cs: TRTLCriticalSection);
procedure LeaveCriticalSection(var cs: TRTLCriticalSection);
```

Смысл этих подпрограмм также очевиден:

InitCriticalSection – инициализирует критический раздел. Это вызов нужно делать перед использованием EnterCrititicalSection или LeaveCriticalSection.

DoneCriticalSection – освобождает ресурсы, связанные с критическим разделом. После этого вызова ни EnterCrititicalSection ни LeaveCriticalSection не могут быть использованы.

EnterCriticalSection – когда этот вызов возвращает результат, вызывающий поток является единственным выполняемым потоком между вызовом EnterCriticalSection и последующим вызовом LeaveCriticalsection.

LeaveCriticalSection – сигнализирует о том, что защищённый код может выполняться другими потоками.

Учтите, что вызов LeaveCriticalsection ***должен быть*** выполнен. Если этого не сделать, то все другие потоки не смогут выполнять код в критическом разделе. Поэтому рекомендуется заключать критический раздел в блок Try..finally. Обычно код выглядит следующим образом:

```
Var
   MyCS : TRTLCriticalSection;

Procedure CriticalProc;
begin
   EnterCriticalSection(MyCS);
   Try
   // Защищённый код
   Finally
      LeaveCriticalSection(MyCS);
   end;
end;

Procedure ThreadProcedure;
begin
   // Код, выполняемый в потоке…
   CriticalProc;
   // Другой код, выполняемый в других потоках…
end;

begin
   InitCriticalSection(MyCS);
// Код запуска потоков.
   DoneCriticalSection(MyCS);
```

```
end.
```

10.4. Менеджер потоков

Подобно тому, как куча реализована с использованием менеджера кучи, а управление `widestring` выполняется менеджером `widestring`, потоки выполняются с использованием менеджера потоков. Это означает, что имеется запись, которая содержит поля процедурного типа для всех возможных используемых функций в потоковых подпрограммах. Потоковые подпрограммы фактически используют эти поля работы.

Потоковые подпрограммы устанавливают свой менеджер потоков для каждой системы. На Windows обычные подпрограммы Windows используются для выполнения функций в менеджере потоков. На Linux и подобных системах менеджер потоков ничего не делает: он будет генерировать ошибку, если используются потоковые подпрограммы. Смысл этого в том, что подпрограммы управления потоками локализованы в библиотеке C. Реализация менеджера потоков в системе привела бы к необходимости сделать RTL зависимой от библиотеки `C`, что нежелательно. Чтобы избежать этой зависимости от библиотеки C, Менеджер Потоков реализован как отдельный модуль (cthreads). Код инициализации этого модуля устанавливает менеджер потоков для управления записями потоков, которые используются подпрограммами C (pthreads).

Запись менеджера потоков может быть восстановлена и установлена также, как запись менеджера кучи. На текущий момент запись выгладит следующим образом:

```
TThreadManager = Record
   InitManager             : Function : Boolean;
   DoneManager             : Function : Boolean;
   BeginThread             : TBeginThreadHandler;
   EndThread               : TEndThreadHandler;
   SuspendThread           : TThreadHandler;
   ResumeThread            : TThreadHandler;
   KillThread              : TThreadHandler;
   ThreadSwitch            : TThreadSwitchHandler;
   WaitForThreadTerminate  : TWaitForThreadTerminateHandler;
   ThreadSetPriority       : TThreadSetPriorityHandler;
   ThreadGetPriority       : TThreadGetPriorityHandler;
   GetCurrentThreadId      : TGetCurrentThreadIdHandler;
   InitCriticalSection     : TCriticalSectionHandler;
   DoneCriticalSection     : TCriticalSectionHandler;
   EnterCriticalSection    : TCriticalSectionHandler;
   LeaveCriticalSection    : TCriticalSectionHandler;
   InitThreadVar           : TInitThreadVarHandler;
   RelocateThreadVar       : TRelocateThreadVarHandler;
   AllocateThreadVars      : TAllocateThreadVarsHandler;
   ReleaseThreadVars       : TReleaseThreadVarsHandler;
end;
```

Значение большинства этих функций должно быть понятным из описания в предыдущих разделах.

`InitManager` и `DoneManager` вызываются, когда менеджер потоков устанавливается (`InitManager`) или когда он удаляется (`DoneManager`). Они могут использоваться для инициализации менеджера потоков или для освобождения памяти, когда это необходимо. Если какая-либо из этих функций возвращает `False`, то операция заканчивается неудачей.

Имеются некоторые специальные элементы в записи, связанные с переменными менеджера потоков:

`InitThreadVar` вызывается, когда потоковая переменная должна быть инициализирована. Она имеет тип:

```
TInitThreadVarHandler = Procedure(var offset: dword;
size: dword);
```

Параметр `offset` указывает на смещение в блоке потоковых переменных: все потоковые переменные размещены в одном блоке, одна за одной. Параметр `size` указывает размер потоковой переменной. Эта функция вызывается один раз для всех потоковых переменных в программе.

`RelocateThreadVar` вызывается каждый раз при запуске потока, один раз для каждого потока. Она имеет тип:

```
TRelocateThreadVarHandler = Function(offset : dword) :
pointer;
```

Она должна возвращать новое местоположение для потоковой переменной.

`AllocateThreadVars` вызывается, когда блок должен быть выделен для всех потоковых переменных для нового потока. Это простая процедура без параметров. Общий размер потоковых переменных записывается компилятором в глобальной переменной `threadvarblocksize`. Менеджер кучи ***не может*** быть использован в этой процедуре: менеджер кучи сам использует потоковые переменные, которые ещё не были выделены.

`ReleaseThreadVars` – эта процедура *(без параметров)* вызывается при завершении потока, и вся выделенная память должна быть снова освобождена.

11. ОПТИМИЗАЦИИ

11.1. Независимо от процессора

Следующие разделы описывают общие оптимизации, выполняемые компилятором независимо от типа процессора. Некоторые из оптимизаций требуют переустановки переключателей, другие выполняются автоматически *(оптимизации, которые требуют изменения переключателей, будут отмечены)*.

11.1.1. Сложение констант

В Free Pascal, если операнд(ы) оператора являются константами, вычисление будет выполнено во время компиляции.

Пример

```
x:=1+2+3+6+5;
```

будет сгенерировано в такой код:

```
x:=17;
```

Кроме того, если индекс массива является константой, смещение будет вычислено во время компиляции. Это означает, что доступ к `MyData[5]` будет таким же эффективным, как доступ к обычной переменной.

Наконец, вызов функций `Chr`, `Hi`, `Lo`, `Ord`, `Pred` или `Succ` с константами в качестве параметров, не генерирует вызовы библиотеки времени выполнения, вместо этого значения вычисляются во время компиляции.

11.1.2. Слияние констант

При использовании одной и той же строковой константы, значения с плавающей точкой или константы-множества два и более раз генерируется только одна копия этой константы.

11.1.3. Сокращённая оценка

Оценка логических выражений останавливается сразу, как только будет известен результат, что делает выполнение кода быстрее, по сравнению с тем, когда логические операнды обрабатываются полностью.

11.1.4. Константы множеств

Использование оператора `in` всегда более эффективно, чем использование эквивалентных операторов <>, =, <=, >=, < и >. Это потому, что сравнение диапазонов может быть выполнено более легко с оператором `in`, чем с помощью обычных операторов сравнения.

11.1.5. Небольшие множества

Множества, которые содержат менее **33** элементов, могут быть напрямую закодированы с помощью **32**-битного значения, поэтому нет необходимости вызова библиотеки реального времени для работы с операндами таких множеств. Они напрямую кодируются генератором кода.

11.1.6. Проверка диапазона

Проверка диапазона выполняется во время компиляции при присваивании констант переменным, что устраняет необходимость такой проверки во время выполнения кода.

11.1.7. And вместо modulo

Если второй операнд в `mod` – это беззнаковая константа степени **2**, то вместо целого деления используется инструкция `and`. Это генерирует более эффективный код.

11.1.8. Сдвиг вместо умножения или деления

Если один из операндов в выражении умножения – это степень двойки, оно кодируется с использований инструкций арифметического сдвига, что генерирует более эффективный код.

Аналогично, если делитель в операции `div` – это степень двойки, то деление кодируется с использований инструкций арифметического сдвига.

То же самое верно, если индекс доступа к массиву является степенью двойки, адрес вычисляется с использованием арифметического сдвига вместо инструкции умножения.

11.1.9. Автоматическое выравнивание

По умолчанию все переменные с размером больше байта гарантированно выравниваются как минимум по границе слова.

Выравнивание в стеке и в разделе данных определяется процессором.

11.1.10. Умная компоновка

Эта функция удаляет весь неиспользуемый код в конечном исполняемом файле, делая его размер меньше.

Режим умной компоновки устанавливается переключателем `-Cx` в командной строке или при помощи глобальной директивы `{$SMARTLINK ON}`.

11.1.11. Встроенные подпрограммы

Следующие подпрограммы библиотеки реального времени кодируются непосредственно в конечный исполняемый файл: `Lo`, `Hi`, `High`, `Sizeof`, `TypeOf`, `Length`, `Pred`, `Succ`, `Inc`, `Dec` и `Assigned`.

11.1.12. Пропуск кадра стека

При определённых условиях кадр стека (код входа и выхода для подпрограммы, см. раздел 6.3. Механизм вызова[93]) будет пропущен и переменные будут напрямую доступны через указатель стека.

Условия для пропуска кадра стека:

- Целевой процессор x86 или ARM.
- Указан переключатель командной строки `-O2` или `-OoSTACKFRAME`.
- Не используется встроенный ассемблер.
- Не используются никакие исключения.
- Никакие подпрограммы не вызываются с выходными параметрами в стеке.
- Функция не имеет параметров.

11.1.13. Регистры переменных

Если используется переключатель `-Or`, локальные переменные или параметры, которые используются очень часто, будут перемещены в регистры для более быстрого доступа.

11.2. Конкретные процессоры

Это список оптимизаций, выполняемых на низком уровне определённым процессором.

11.2.1. Intel 80x86

Здесь приводится список методов оптимизаций, используемых в компиляторе:

1. При оптимизации для указанного процессора *(`-Op1`, `-Op2`, `-Op3`)*

выполняется следующее:

- В операторе `case` выполняется проверка, является ли оптимальным для выполнения переход по таблице или последовательность условных переходов.
- Определяется ряд стратегий, когда выполняется визуальная оптимизация, например, `movzbl (%ebp), %eax` будет заменено на `xorl %eax,%eax; movb (%ebp),%al` для Pentium и PentiumMMX.

2. При оптимизации по скорости (`-OG`, по умолчанию) или размеру (`-Og`) будет сделан выбор между использованием наиболее коротких инструкций *(по размеру)*, таких как `enter $4`, или длинных инструкций `subl $4,%esp` для оптимизации по скорости. Если требуется наименьший размер, данные выравниваются по минимальной границе. Если требуется скорость, данные выравниваются по наиболее эффективной границе, насколько это возможно.
3. Быстрые оптимизации *(`-O1`)*: активируют визуальный оптимизатор.
4. Медленные оптимизации *(`-O2`)*: также активируют общие выражения ликвидации *(раньше называлось **«перезагрузка оптимизатора»**)*.
5. Неопределённые оптимизации *(`-OoUNCERTAIN`)*: с этим переключателем алгоритм общих выражений ликвидации может быть форсирован для создания неопределённых оптимизаций.

Хотя вы можете включить неопределённые оптимизации для людей, которые не понимают следующие технические разъяснения, это может быть безопасным.

ПРИМЕЧАНИЕ

Если неопределённые оптимизации включены, CSE алгоритм предполагает, что

- Если что-то записано в локальный/глобальный регистр или параметр процедуры/функции, это значение не перезапишет значение, для которого имеется указатель.
- Если что-то записано в память, на которую ссылается переменная-указатель, то это значение не перезаписывается локальной/глобальной переменной или параметром процедуры/функции.

Практическим следствием этого является то, что вы не можете использовать неопределённые оптимизации, если вы читаете/записываете переменные как напрямую, так и использованием указателей *(включая как параметры `Var`, так и указатели)*.

Следующий пример создаёт плохой код, если вы включите неопределённые оптимизации:

```
Var
   temp: Longint;

Procedure Foo(Var Bar: Longint);
Begin
   If (Bar = temp) Then
```

```
        Begin
           Inc(Bar);
           If (Bar <> temp) then Writeln('bug!')
        End
End;

Begin
   Foo(Temp);
End.
```

В этом примере создаётся плохой код, потому что вы работаете с переменной `Temp` как через её имя. Так и через указатель, используя в этом случае параметр `Bar`, который является указателем на `Temp` в вышеописанном коде.

С другой стороны, вы можете использовать неопределённые оптимизации, если работаете с глобальными/локальными переменными или параметрами через указатели, и ***только*** через этот указатель *(вы можете использовать несколько указателей на одну переменную – это не имеет значения)*.

Пример

```
Type TMyRec = Record
     a, b: Longint;
End;
PMyRec = ^TMyRec;

TMyRecArray = Array [1..100000] of TMyRec;
PMyRecArray = ^TMyRecArray;

Var MyRecArrayPtr: PMyRecArray;
    MyRecPtr: PMyRec;
    Counter: Longint;

Begin
  New(MyRecArrayPtr);
  For Counter := 1 to 100000 Do
    Begin
      MyRecPtr := @MyRecArrayPtr^[Counter];
      MyRecPtr^.a := Counter;
      MyRecPtr^.b := Counter div 2;
    End;
End.
```

Это пример создаёт правильный код, потому что глобальная переменная `MyRecArrayPtr` не доступна непосредственно, а только через указатель *(`MyRecPtr` в данном примере)*.

В заключение можно сказать, что вы можете использовать неопределённые оптимизации только тогда, когда вы знаете, что делаете.

11.2.2. Motorola 680x0

Использование переключателя `-O2` (по умолчанию) делает несколько

оптимизаций в генерируемом коде, наиболее заметные из них перечислены ниже:

- Расширение из байта в длинное целое будет использовать `EXTB`.
- Возвращение из функции будет использовать `RTD`.
- Проверка диапазона не будет генерироваться во время вызовов.
- Умножение будет использовать длинную инструкцию `MULS`, вызов библиотеки реального времени не будет сгенерирован.
- Деление будет использовать длинную инструкцию `DIVS`, вызов библиотеки реального времени не будет сгенерирован.

11.3. Переключатели оптимизации

Здесь описаны различные переключатели оптимизаций и их действие на генерируемый код, материал сгруппирован по переключателям.

`-On`: где ***n = 1..3***: эти переключатели активируют оптимизатор. Более высокие уровни автоматически включают в себя более низкие.

- **Уровень 1** (`-O1`) активирует визуальный оптимизатор *(общая последовательность инструкций заменяется более быстрыми эквивалентами)*.
- **Уровень 2** (`-O2`) включает анализатор потока данных ассемблера, который позволяет очистить общие подвыражения процедур для удаления ненужных перезагрузок регистров значениями, которые уже содержатся в регистрах.
- **Уровень 3** (`-O3`) то же, что уровень 2, плюс некоторые трудоёмкие операции.

`-OG`: заставляет генератор кода *(и оптимизатор, если он включен)* работать быстрее, но с наибольшим количество **«умного»** кода, последовательности инструкций *(такие как `"subl $4,%esp"`)* вместо медленных, инструкции меньшего размера по возможности *(`"enter $4"`)*. Установлен по умолчанию.

`-Og`: противоположность переключателя `-OG`. Эти переключателя являются взаимоисключающими: включение одного будет выключать другой.

`-Or`: этот параметр заставляет генератор кода проверять, какие переменные используются больше, и сохранять их в регистры.

`-Opn`: где ***n = 1..3***: указывает целевому процессору не активировать оптимизатор. Он влияет только на генератор кода, если активирован оптимизатор.

- Во время генерации кода процесса, эта установка используется для определения, будет ли переход по таблице или последовательность условных переходов в операторе `case`.
- Визуальный оптимизатор принимает ряд решений на основе этого параметра, например, он переводит некоторые сложные инструкции, такие

как

```
movzbl (mem), %eax|
```

в комбинацию простых инструкций

```
xorl %eax, %eax
movb (mem), %al
```

для Pentium.

-Ou: включает неопределённые оптимизации. Однако вы не всегда можете его использовать. В предыдущем разделе описано, как и когда они могут быть использованы, а когда не могут.

11.4. Советы по генерации наиболее быстрого кода

Здесь представлены некоторые общие советы для получения наилучшего кода. В основном они касаются стиля кодирования.

- Найти наиболее эффективный алгоритм. Неважно, как вы и компилятор настроите код, алгоритм быстрой сортировки всегда превосходит **«метод пузырька»**.
- Использовать переменные, размер которых является **«родным»** для процессора, для которого вы пишите программу. В настоящее время для Free Pascal это **32**-разрядные или **64**-разрядные переменные, поэтому используйте типы `longword` и `longint`.
- Включить оптимизатор.
- Пишите ваши операторы `if/then/else` таким образом, чтобы код в части "`then`" выполнялся большую часть времени *(улучшает скорость, если велика вероятность успешного перехода)*.
- Не используйте `ansistrings`, `widestrings` и поддержку исключений, так как это требует много перегружаемого кода.
- Профилируйте ваш код *(см. переключатель `-pg`)* для поиска **«узких»** мест. Если хотите, можете переписать эти части на ассемблере. Вы можете взять код, сгенерированный компилятором в качестве отправной точки. Если в командной строке передан переключатель `-a`, то компилятор не будет стирать ассемблерный файл в конце процесса ассемблирования, так что вы можете изучить ассемблерный файл.

11.5. Советы по генерации наименьшего кода

Здесь представлены некоторые общие советы для получения кода минимально возможного размера.

- Найти лучший алгоритм.
- Использовать переключатель `-Og`.
- Перегруппировать глобальные статические переменные в том же модуле,

который имеет одинаковый размер вместе с минимизацией количества директив выравнивания *(которые увеличивают необязательные разделы* `.bss` *и* `.data` *)*. Внутренне это связано с тем, что фактически все статические данные записываются в ассемблерный файл в том порядке, в котором они объявлены в исходном коде на Pascal.

- Не использовать модификатор `cdecl`, так как это генерирует около одной дополнительной инструкции после каждого вызова подпрограммы.
- Использовать опцию **«умной компоновки»** для ваших модулей *(включая модуль system)*.
- Не используйте `ansistrings`, `widestrings` и поддержку исключений, так как это требует много перегружаемого кода.
- Выключить проверку диапазона и проверку стека.
- Выключить генерацию информации времени выполнения.

11.6. Оптимизация программы в целом

11.6.1. Общие сведения

Традиционно компиляторы оптимизируют программу процедура за процедурой, или в лучшем случае модуль за модулем. Оптимизация программы в целом (WPO) означает, что компилятор думает, что компиляция всех модулей создаёт программу или библиотеку и оптимизирует их, используя комбинации информации о том, как они используются вместе в данном конкретном случае.

Как правило WPO работает следующим образом:

- Обычно программа компилируется с опцией, которая указывает компилятору, что он должен записать различные биты информации в файл обратной связи.
- Программа перекомпилируется второй раз (и дополнительно все используемые модули) с включением WPO, предоставляя сгенерированный на первом этапе файл обратной связи как дополнительный вход компилятору.

Этой схеме следует Free Pascal.

Реализация этой схемы сильно зависит от компилятора. Другой реализацией может быть то, что компилятор генерирует некий промежуточный код *(например, байт-код)* и компоновщик выполняет оптимизацию программы в целом вместе с трансляцией в код для целевой машины.

11.7. Основные принципы

Несколько общих принципов, которые соблюдаются когда разработка FPC

выполняет WPO:

- Вся информация, необходимая для генерации файла обратной связи WPO для программы всегда записывается в файлы ppu. Это означает, что можно использовать обычную RTL для WPO *(или, в общем случае, любой откомпилированный модуль)*. То есть сама RTL не будет оптимизирована, а скомпилированный код программы и её модулей могут быть корректно оптимизированы, потому что компилятор знает всё. что он должен знать о модулях RTL.
- Сгенерированный файл обратной связи WPO – это простой текст. Идея заключается в том, что он должен легко проверяться вручную, а при необходимости можно добавить в него информацию внешними инструментами, при желании *(например, профильную информацию)*.
- Выполнение подсистемы WPO в компиляторе является весьма модульным, поэтому должно быть лёгким для включения в дополнительного поставщика информации WPO, или для выбора во время выполнения между различными поставщиками для одного вида информации. В то же время, взаимодействие с остальными подсистемами компилятора будет сведено к минимуму, чтобы улучшить ремонтопригодность.
- Можно создать файл обратной связи WPO и в то же время использовать файл как вход. В некоторых случаях использования этого второго файла обратной связи в качестве входных данных во время компиляции может в дальнейшем улучшить результаты.

11.7.1. Как использовать

Шаг 1: Генерация файла обратной связи WPO

Первый шаг в WPO – это компиляция программы (или библиотеки) и всех её модулей. Это можно сделать обычным способом, но с указанием двух опций в командной строке. Как это показано ниже:

```
-FW/path/to/feedbackfile.wpo -OW<selected_wpo_options>
```

Первая опция указывает компилятору, где должен быть записан файл обратной связи WPO, вторая опция говорит компилятору, что нужно включить оптимизации WPO.

Затем, после компоновки программы или библиотеки, компилятор будет собирать всю информацию, необходимую для выполнения WPO и записывать её в указанный файл.

Шаг 2: Использование сгенерированных файлов обратной связи WPO

Для фактического применения опций WPO программа (или библиотека) и все или некоторые из используемых модулей должны быть перекомпилированы с применением опции

```
-Fw/path/to/feedbackfile.wpo -Ow<selected_wpo_options>
```

(Обратите внимание на нижний регистр символа `w`*)*. Это говорит компилятору, что нужно использовать файл обратной связи, сгенерированный на предыдущем шаге. Затем компилятор будет читать информацию о программе, собранную во время предыдущего запуска компилятора и использовать её во время текущей компиляции модулей и/или программы/ библиотеки.

Модули, не компилируемые во время второго прохода, очевидно не могут быть оптимизированы, но они будут работать правильно при использовании вместе с оптимизированными модулями и программой/библиотекой.

ПРИМЕЧАНИЕ

Обратите внимание, что опции должны быть всегда указаны в командной строке: нет директивы для включения WPO в исходном коде, так как это имело бы смысл только при компиляции целой программы *(без модулей)*.

11.7.2. Доступные оптимизации WPO

Опции командной строки `-OW` и `-Ow` требуют список оптимизаций программы в целом, разделённый запятыми. Это строки, где каждая строка обозначает опцию. Список доступных опций приведён ниже:

`all` – включает все доступные оптимизации.

`devirtcalls` – заменяет вызов виртуального метода вызовом обычного *(статического)* метода, если компилятор может определить, что вызов виртуального метода всегда будет таким же, как статический метод. Это делает код меньше и быстрее. В общем, это включает оптимизацию других оптимизаций, потому что это делает программу лёгкой для анализа в связи с тем, что снижает косвенный контроль потока.

Имеются два ограничения этой опции:

1. В текущей реализации она не чувствительна к содержимому. Это значит, что компилятор только просматривает программу в целом и определяет для каждого класса тип, методы которого могут быть девиртуализированы. Но компилятор не просматривает каждый вызов, чтобы определить, действительно ли этот вызов может быть девиртуализирован.
2. В текущей реализации ещё не девиртуализуются вызовы методов интерфейса. Ни при вызове их через интерфейс, ни при вызове их через экземпляр класса.

`optvmts` – эта оптимизация смотрит, какие типы класса могут быть реализованы и какие виртуальные методы могут быть вызваны в программе. На основании этой информации она заменяет записи виртуальной таблицы методов (VMT), которые могут быть никогда не вызваны на ссылку `FPC_ABSTRACTERROR`. Это означает, что такие

методы, если они не вызываются напрямую через унаследованный вызов из дочернего класса/объекта, могут быть удалены при компоновке. Это не имеет практически никакого влияния на скорость, но может помочь при уменьшении размера кода.

Имеются два ограничения этой опции:

1. Общедоступные методы или `getters/setters` общедоступных свойств никогда не будут оптимизированы этим способом, потому что они могут в любой момент быть связаны и вызваны через RTTI *(который компилятор не может определить)*.
2. Такая оптимизация пока не реализована для виртуальных методов класса.

`wsymbolliveness` – этот параметр не выполняет каких-либо оптимизаций сам по себе. Он просто указывает компилятору записи каких функций/процедур и где были сохранены компоновщиком в конечной программе. Во время следующего прохода WPO компилятор может игнорировать удалённые функции/процедуры настолько, насколько WPO сочтёт нужным *(например, если конкретный тип класса только создан в одной неиспользуемой процедуре, то игнорирование этой процедуры может повысить эффективность двух предыдущих оптимизаций)*.

Имеются два ограничения этой опции:

1. Эта оптимизация требует, чтобы утилита nm была установлена в системе. Для двоичных файлов Linux также будет работать objdump. В будущем эта информация также может быть получена из встроенного компоновщика для платформ, которые это поддерживают.
2. Сбор информации для этой оптимизации *(используя `-OWsymbolliveness`)* требует, чтобы умная компоновка была включена (`-XX`), а символьный анализ отключен (`-Xs-`). Если использовать предыдущий способ сбора информации, то эти ограничения не действуют.

11.7.3. Формат файла WPO

Эта информация может оказаться интересной, если есть необходимость добавить внешние данные в файл обратной связи WPO, например, с помощью профилирующего инструмента. Для регулярного использования функции WPO следующая информация не нужна и может быть игнорирована.

Файл состоит из комментариев и некоторого количества разделов. Комментариями являются строки, которые начинаются с символа #. Каждый раздел начинается с `"%"`, после чего следует имя раздела *(например, `% contextinsensitive_devirtualization`)*.

После этого, пока не достигнут конец файла или до начала следующего раздела *(то есть до строки, которая начинается с `"%"`)*, сначала идёт понятное для

человека описание формата раздела *(в комментариях)*, а затем содержимое самого раздела.

Не существует правил для того, как должно выглядеть содержимое раздела, за исключением того, что строки, которые начинаются с символа #, являются зарезервированными для комментариев, а строки, начинающиеся с %, зарезервированы для метки начала раздела.

12. ПРОГРАММИРОВАНИЕ ОБЩЕДОСТУПНЫХ БИБЛИОТЕК

12.1. Введение

Free Pascal поддерживает создание общедоступных библиотек на нескольких операционных системах. В таблице 12.1 показано, для каких операционных систем поддерживается создание библиотек.

Таблица 12.1. Поддержка общедоступных библиотек.

Операционная система	Расширение библиотеки	Префикс библиотеки
linux	.so	**lib**
windows	.dll	**Нет**
BeOS	.so	**lib**
FreeBSD	.so	**lib**
NetBSD	.so	**lib**

В столбце **«префикс библиотеки»** показано, какие имена библиотек разрешены и будут созданы. Например, для LINUX имя библиотеки всегда будет иметь префикс `lib`, когда она будет создана. Так что если вы создаёте библиотеку с именем mylib, то для LINUX в результате создания это будет библиотека libmylib.so. Кроме того, при импорте процедур из общедоступной библиотеки нет необходимости указывать префикс или расширение библиотеки.

В следующих разделах мы узнаем, как создать библиотеку и использовать её в программе.

12.2. Создание библиотеки

Создание библиотек поддерживается в любом режиме компилятора Free Pascal, но может случиться так, что значения аргументов или возвращаемых результатов будут отличаться, если библиотека скомпилирована в двух разных режимах. Например, если ваша функция принимает целочисленный аргумент *(типа `Integer`)*, то библиотека будет принимать целые числа разных размеров, если вы компилируете её в режимах `Delphi` или `TP`.

Библиотека создаётся как программа, только используется ключевое слово `library`, и имеется раздел `exports`. В следующем листинге приведён пример

простой библиотеки:

Листинг: progex/subs.pp

```
{ Пример библиотеки }

library subs;

function SubStr (CString : PChar; FromPos, ToPos : Longint) :
PChar; cdecl;
var
   Length : Integer;
begin
   Length : = StrLen(CString);
   SubStr : = CString + Length;
   if (FromPos > 0) and (ToPos >= FromPos) then
     begin
       if Length >= FromPos then
       SubStr : = CString + FromPos - 1;
       if Length > ToPos then CString[ToPos] : = #0;
     end;
end;

exports
 SubStr;
end.
```

Функция `SubStr` не должна быть объявлена в файле самой библиотеки. Она может быть объявлена в разделе интерфейса *(interface)* модуля, который используется библиотекой.

В результате компиляции этого исходного кода будет создана библиотека с именем libsubs.so для UNIX-систем, или subs.dll для WINDOWS или OS/2. Компилятор сам позаботится о создании всех дополнительных связей, которые требуются для создания общедоступной библиотеки.

Библиотека экспортирует одну функцию: `SubStr`. Это важный момент. В том случае, когда функция имеется в разделе `exports`, она будет экспортироваться.

Если вы хотите, чтобы ваша функция вызывалась из программы, созданной с помощью другого компилятора, важно соблюдать правильность соглашения о вызовах для экспортируемых функций. Так как программы, созданные с помощью других компиляторов, не знают о соглашениях о вызовах Free Pascal, ваша функция будет вызвана неправильно, в результате чего будет повреждён стек.

В WINDOWS большинство библиотек используют соглашение о вызовах `stdcall`, так что может быть лучше использовать именно его, если ваша библиотека будет работать в системах WINDOWS. В большинстве систем UNIX используется соглашения о вызовах C, поэтому следует использовать модификатор `cdecl`.

12.3. Использование библиотеки в программе на Паскале

Чтобы использовать функцию из библиотеки, достаточно объявить, что эта функция в библиотеке существует как внешняя *(external)*, с правильными аргументами и типом возвращаемого значения. Используемое функцией соглашение о вызовах должно быть также правильно объявлено. Затем компилятор скомпонует библиотеку, как это указано в операторе `external` для вашей программы (если вы опустите имя библиотеки в модификаторе `external`, то вы всё равно можете указать компилятору ссылку на эту библиотеку, используя директиву `{$Linklib}`).

Например, для использования описанной выше библиотеки из программы на Pascal, вы можете использовать следующий способ:

Листинг: progex/psubs.pp

```
program testsubs;
function SubStr(const CString : PChar; FromPos, ToPos :
longint) : PChar;
             cdecl; external 'subs';
var
   s : PChar;
   FromPos, ToPos : Integer;

begin
   s : = 'Test';
   FromPos : = 2;
   ToPos : = 3;
   WriteLn(SubStr(s, FromPos, ToPos));
end.
```

Как показано в примере, вы должны объявить функцию как внешнюю (`external`). Здесь также необходимо определить правильное соглашение о вызовах *(оно должно всегда соответствовать соглашению, используемому функцией в библиотеке)* и использовать правильный формат вашего объявления. Также обратите внимание, что в имени импортируемой библиотеки не указывается расширение, и префикс `lib` также не добавляется.

Эта программа может быть откомпилирована без каких-либо дополнительных командных переключателей, и должна выполняться только в том случае, если предоставленная библиотека размещена в таком месте, где система может её найти. Например, для LINUX это **`/usr/lib`** или любая из директорий, перечисленных в файле /etc/ld.so.conf. Для WINDOWS это может быть каталог программы или любой каталог, упомянутый в команде `PATH`.

При подобном методе связи библиотеки с вашей программой библиотека используется во время компиляции. Это означает, что

1. Библиотека должна быть представлена в системе, где программа компилируется.
2. Библиотека должна быть представлена в системе, где программа

выполняется.

3. Обе библиотеки должны быть одинаковыми.

А ещё может быть, что вы не знаете имя вызываемой функции, вы только знаете аргументы, которые она принимает.

Поэтому также возможно загрузить библиотеку во время выполнения, записать адрес функции в процедурную переменную и использовать эту процедурную переменную для доступа к функции в библиотеке.

Следующий пример демонстрирует эту технологию:

Листинг: progex/plsubs.pp

```
program testsubs;

Type
   TSubStrFunc =
     function(const CString : PChar; FromPos, ToPos :
longint) : PChar; cdecl;

Function dlopen(name : pchar; mode : longint) : pointer; cdecl;
 external 'd l';
Function dlsym(lib : pointer; name : pchar) : pointer; cdecl;
external 'dl';
Function dlclose(lib : pointer) : longint; cdecl; external
'dl';

Var
  S : PChar;
  FromPos, ToPos : Integer;
  lib : pointer;
  SubStr : TSubStrFunc;

begin
   s : = 'Test';
   FromPos : = 2;
   ToPos : = 3;
   lib := dlopen('libsubs.so', 1);
   Pointer(Substr) : = dlsym(lib, 'SubStr');
   WriteLn(SubStr(s, FromPos, ToPos));
   dlclose(lib);
end.
```

Как и в случае связи во время компиляции, главное в этом листинге – это объявление типа `TSubStrFunc`. Он должен соответствовать объявлению функции, которую вы пытаетесь использовать. Неправильное объявление приведёт к повреждению стека или, что ещё хуже, ваша программа может вызвать крах системы.

12.4. Использование библиотек Паскаль с программами на C

ПРИМЕЧАНИЕ

В примерах этого раздела предполагается использование системы LINUX. Однако аналогичные команды существуют и для других систем.

Вы также можете вызвать библиотеку, сгенерированную Free Pascal из программы, написанной на C:

Листинг: progex/ctest.c

```
#include <string.h>

extern char* SubStr(const char*, int, int);

int main()
{
  Char* s;
  int FromPos, ToPos;
  s = strdup("Test");
  FromPos = 2;
  ToPos = 3;
  printf("Result from SubStr : '%s '\n", SubStr(s, FromPos,
ToPos));
  return 0;
}
```

Для компиляции этого примера может быть использована следующая команда:

```
gcc -o ctest ctest.c -lsubs
```

при условии, что код находится в файле ctest.c.

Библиотека также может быть загружена динамически из программы C, как показано ниже в примере:

Листинг: progex/ctest2.c

```
#include <dlfcn.h>
#include <string.h>

int main()
{
  void *lib;
  char *s;
  int FromPos, ToPos;
  char*(*SubStr)(const char*, int, int);
  lib = dlopen("./libsubs.so", RTLD_LAZY);
  SubStr = dlsym(lib, "SUBSTR");
  s = strdup(" Test ");
  FromPos = 2;
  ToPos = 3;
  printf("Result from SubStr : '%s ' \n", (*SubStr)(s, FromPos,
```

```
   ToPos));
    dlclose(lib);
    return 0;
  }
```

Для компиляции этого примера может быть использована следующая команда:

```
gcc -o ctest2 ctest2.c -ldl
```

Параметр `-ldl` говорит gcc, что программе необходима библиотека libdl.so для загрузки динамических библиотек.

12.5. Некоторые вопросы Windows

По умолчанию Free Pascal *(на самом деле компоновщик, используемый Free Pascal)* создаёт библиотеки, которые не являются перемещаемыми. Это значит, что они должны загружаться по фиксированному адресу памяти: этот адрес называется ImageBase. Если в программу загружаются две библиотеки, сгенерированные с помощью Free Pascal, то будет конфликт, потому что первая библиотека уже заняла место в памяти, куда должна загружаться вторая библиотека.

В Free Pascal есть два переключателя, которые управляют генерацией общедоступных библиотек для WINDOWS:

-WR – генерирует перемещаемую библиотеку. Эта библиотека может быть перемещена в другое место в памяти, если адрес ImageBase уже используется.

-WB – указывает адрес `ImageBase` для создаваемой библиотеки. Стандартный адрес ImageBase, используемый Free Pascal – это **0x10000000**. Этот переключатель позволяет изменить его путём указания другого адреса, например `-WB11000000`.

Первый вариант является более предпочтительным, так как программа может загрузить несколько библиотек, имеющихся в системе, и они уже будут использовать адрес ImageBase. Вторая опция быстрее, так как нет необходимости перемещать библиотеку, если адрес ImageBase не используется.

13. ИСПОЛЬЗОВАНИЕ РЕСУРСОВ WINDOWS

13.1. Директива ресурса $R

Под WINDOWS и LINUX или другой платформой, использующей бинарные файлы ELF *(в разрабатываемой версии **2.3.1** все поддерживаемые FPC платформы имеют доступ к ресурсам)*, вы можете включать ресурсы в ваш исполняемый файл или библиотеку, используя директиву `{$R filename}`. Эти ресурсы могут быть доступны через стандартные вызовы WINDOWS API: эти вызовы могут иметься и в других платформах.

Когда компилятор обнаруживает директиву ресурса, он просто создаёт запись в файле модуля .ppu. Он не создаёт ссылку на ресурс. Только когда создаётся исполняемый файл или библиотека, он просматривает все файлы ресурсов, которые перечислены в директиве, и пытается объединить их.

По умолчанию расширением для файлов ресурсов является .res. Если имя файла имеет в качестве первого символа звёздочку `(*)`, компилятор заменит звёздочку именем текущего модуля, библиотеки или программы.

ПРИМЕЧАНИЕ

Это значит, что звёздочка может использоваться только после подставляющей части модуля, библиотеки или программы *(`unit`, `library` или `program`)*.

13.2. Создание ресурсов

Компилятор Free Pascal сам не создаёт каких-либо файлов ресурсов. Он только компилирует их в исполняемый файл. Для создания файлов ресурсов вы можете использовать некоторые инструменты GUI, такие как Borland resource workshop. Но можно также использовать компилятор ресурсов WINDOWS, например, GNU windres, который поставляется с GNU binutils. Дистрибутив Free Pascal также содержит версии, которые вы можете использовать.

Использовать windres просто. Он читает входной файл описания ресурсов и генерирует выходной файл ресурсов.

Типичный вызов windres:

```
windres -i mystrings.rc -o mystrings.res
```

он будет читать файл mystrings.rc и создаст выходной файл ресурса mystrings.res.

Полное описание windres выходит за рамки этого документа, но ниже приведены некоторые вещи, которые вы можете использовать:

stringtables *(таблицы строк)*, которые содержат списки строк.
bitmaps *(рисунки)*, которые читаются из внешнего файла.
icons *(значки)*, которые также читаются из внешнего файла.
Version information *(информация о версии)*, которая может быть просмотрена в проводнике WINDOWS.
Menus *(меню)* может быть разработано как ресурсы и использовано в GUI ваших приложений.
Arbitrary data *(произвольные данные)* могут быть включены как ресурсы и прочитаны с помощью вызовов API.

Некоторые из них будут описаны ниже.

13.3. Использование строковых таблиц

Таблицы строк могут использоваться для записи и извлечения больших коллекций строк в вашем приложении.

Таблица строка выглядит следующим образом:

```
STRINGTABLE { 1, "hello World !"
              2, "hello world again !"
              3, "last hello world !" }
```

Вы можете откомпилировать её *(мы предполагаем, что файл называется tests.rc)* следующим образом:

```
windres -i tests.rc -o tests.res
```

Способ получения строки из вашей программы будет следующим:

```
program tests;
{$mode objfpc}

Uses Windows;
{$R *.res}

Function LoadResourceString (Index : longint): Shortstring;
 begin
   SetLength(Result,LoadString(FindResource(0,Nil,RT_STRING),
                               Index,
                               @Result[1],
                               SizeOf(Result)));
 end;

Var I: longint;

begin
   For i:=1 to 3 do
    Writeln(Loadresourcestring(I));
end.
```

Вызов FindResource ищет таблицу строк в скомпилированных ресурсах. Функция LoadString затем читает строку от индекса i до конца таблицы, и помещает её в буфер, который может быть использован. Оба вызова имеются в модуле windows.

13.4. Вставка информации о версии

win32 API позволяет хранить информацию о версии программы в вашем бинарном файле. Эту информацию можно просмотреть с помощью проводника WINDOWS, щёлкнув правой кнопкой на программе или библиотеке и выбрав меню СВОЙСТВА. Во вкладке ВЕРСИЯ будет отображаться информация о версии программы.

Здесь показано, как вставить информацию о версии в ваш бинарный файл:

```
1 VERSIONINFO
FILEVERSION 4, 0, 3, 17
PRODUCTVERSION 3, 0, 0, 0
FILEFLAGSMASK 0
FILEOS 0x40000
FILETYPE 1
{
   BLOCK "StringFileInfo"
   {
       BLOCK "040904E4"
       {
           VALUE "CompanyName", "Free Pascal"
           VALUE "FileDescription", "Free Pascal version in-
formation extractor"
           VALUE "FileVersion", "1.0"
           VALUE "InternalName", "Showver"
           VALUE "LegalCopyright", "GNU Public License"
           VALUE "OriginalFilename", "showver.pp"
           VALUE "ProductName", "Free Pascal"
           VALUE "ProductVersion", "1.0"
        }
    }
}
```

Как вы можете видеть, можно вставлять различные виды информации о версии информационный блок. Ключевое слово VERSIONINFO отмечает начало ресурсного блока информации о версии. Ключевые слова FILEVERSION, PRODUCTVERSION устанавливают актуальную версию файла, в то время как блок StringFileInfo устанавливает другую информацию, которая отображается в проводнике.

Бесплатная библиотека компонентов (Free Component Library) поставляется с

модулем (fileinfo), который позволяет извлекать и просматривать информацию о версии в простой и доступной форме. Демонстрационная программа, которая поставляется с ним (showver) показывает информацию о версии для любого исполняемого файла или DLL.

13.5. Вставка значка приложения

Если WINDOWS отображает файл в проводнике, то она ищет значок в исполняемом файле, чтобы отобразить его перед именем файла. Этот значок является значком приложения.

Вставить значок приложения очень просто. Это можно сделать следующим образом:

```
AppIcon ICON "filename.ico"
```

Здесь читается файл filename.ico и значок вставляется в файл ресурсов.

13.6. Использование препроцессора Pascal

Иногда вы хотите использовать символьные имена в вашем ресурсном файле и использовать те же имена в вашей программе для доступа к ресурсам. Чтобы достичь этого, существует препроцессор для windres, который понимает синтаксис Pascal: fprcp. Этот препроцессор поставляется с дистрибутивом Free Pascal.

Идея заключается в том, что препроцессор читает модуль на Pascal, который имеет некоторые символьные константы, определённые в этом модуле, и заменяет символьные имена в файле ресурсов значениями констант в модуле.В качестве примера рассмотрим следующий модуль:

```
unit myunit;

interface

Const
  First = 1;
  Second = 2:
  Third = 3;
Implementation
end.
```

И следующий файл ресурсов:

```
#include "myunit.pp"
STRINGTABLE { 1, "hello World !"
              2, "hello world again !"
              3, "last hello world !" }
```

Если вы вызовите windres с опцией препроцессора:

```
windres --preprocessor fprcp -i myunit.rc -o myunit.res
```

то препроцессор заменит символьные имена `first`, `second` и `third` на их фактические значения.

В вашей программе вы можете ссылаться на строки с помощью их символьных имён *(констант)* вместо использования числовых индексов.

ПРИЛОЖЕНИЕ A: АНАТОМИЯ ФАЙЛА МОДУЛЯ

A.1. Основы

Как описано в главе 4. СГЕНЕРИРОВАННЫЙ КОД[81], файл описания модуля *(далее для краткости **«файлы PPU»**)* используется для определения необходимости перекомпиляции модуля. Иными словами, файлы PPU работают как `mini-makefiles`, которые используются для проверки зависимостей в коде различных модулей, а также для проверки даты модулей. Кроме того, они содержат определения общедоступных идентификаторов для модуля.

Общий формат файла ppu показан на рис A.1. в разделе A.5. Создание ppu-файлов[172]

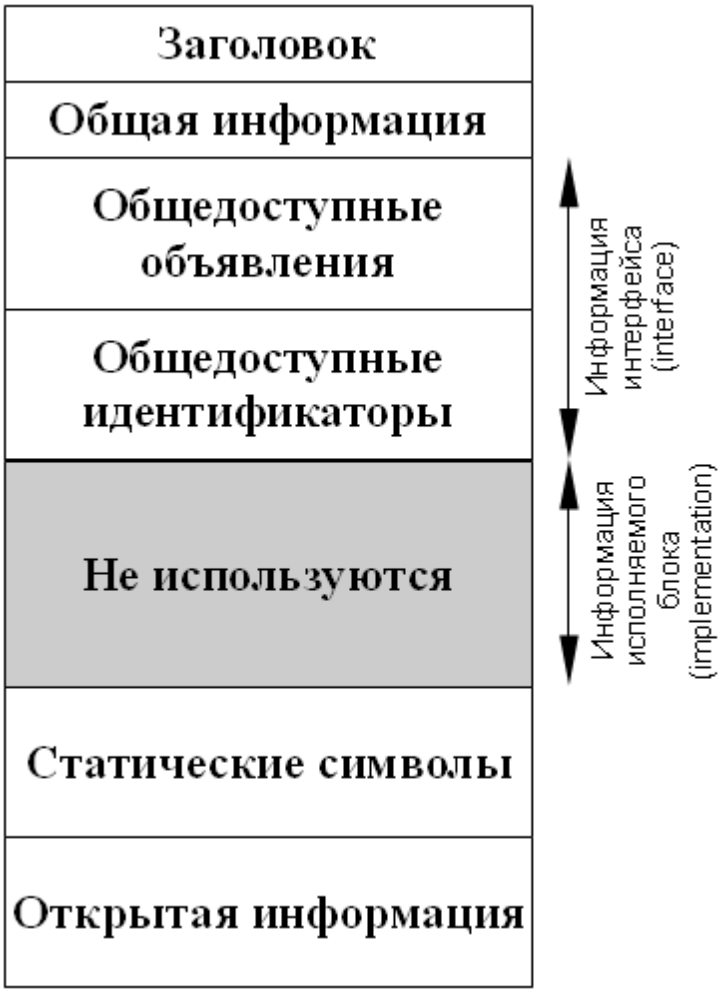

Рис A.1: Формат PPU файла

Для чтения или записи `ppu`-файла, может использоваться ppu-модуль ppu.pas, который имеет объект с именем tppufile, содержащий все подпрограммы для обработки ppu-файла. В описании макета ppu-файла также представлены методы, которые могут быть использованы.

Файл модуля состоит в основном из **5** или **6** частей:

1. Заголовок модуля.
2. Блок общей информации *(в коде ошибочно назван разделом интерфейса)*.
3. Блок объявлений. Содержит все объявления типов и процедур.
4. Блок идентификаторов. Содержит символьные имена и представления всех объявлений.
5. Блок связей. Содержит все ссылки из этого модуля на другие модули и внутри этого модуля. Доступно только когда установлен флаг `uf_has_browser` в модуле флагов.
6. Исполняемый блок (в данное время не используется).

A.2. Чтение ppu-файлов

Сначала мы создаём объект ppufile, который будет использоваться ниже. Мы открываем модуль test.ppu в качестве примера.

```
var
   ppufile : pppufile;
begin
{ Initialize object }
   ppufile:=new(pppufile,init('test.ppu');
{ open the unit and read the header, returns false when it
fails }
if not ppufile.openfile then
   error('error opening unit test.ppu');
{ here we can read the unit }
{ close unit }
   ppufile.closefile;
{ release object }
   dispose(ppufile,done);
end;
```

ПРИМЕЧАНИЕ

Если функция завершилась ошибкой *(например, в записи осталось недостаточно байт)*, она устанавливает переменную `ppufile.error`.

A.3. Заголовок

Заголовок состоит из записи (`tppuheader`), содержащей несколько частей информации для перекомпиляции. Это показано в таблице A.1. Заголовок всегда записывается в формате `little-endian`.

Таблица A.1. Заголовок `PPU`.

Смещение	Размер (в байтах)	Описание
00h	3	Надпись «PPU» в коде ASCII

03h	3	Версия формата PPU-файла (например, «021» в ASCII)
06h	2	Версия компилятора, используемого для компиляции этого модуля *(старшая, младшая)*
08h	2	Код модуля целевого процессора
0Ah	2	Код модуля целевой операционной системы
0Ch	4	Флаги для PPU-файла
10h	4	Размер PPU-файла *(без заголовка)*
14h	4	CRC-32 для всего PPU-файла
18h	4	CRC-32 для части PPU-файла (в основном, общедоступных данных)
1Ch	8	Зарезервировано

Заголовок уже прочитан командой `ppufile.openfile`. Вы можете получить доступ ко всем полям, используя `ppufile.header`, которая содержит текущую запись заголовка.

Таблица А.2. Значения PPU CPU Field**.**

Значение	Описание
0	Неизвестный
1	Intel 80x86 или совместимый
2	Motorola 680x0 или совместимый
3	Alpha AXP или совместимый
4	PowerPC или совместимый

Некоторые возможные флаги в заголовке описаны в таблице А.3. Не все флаги описаны. Для получения дополнительной информации читайте исходный код ppu.pas.

Таблица А.3. Значения PPU Header Flag**.**

Имя бита символьного флага	Описание
uf_init	Модуль имеет раздел инициализации (стиль Delphi или TP)
uf_finalize	Модуль имеет раздел завершения
uf_big_endian	Все данные записаны частями в формате big-endian

uf_has_browser	Модуль содержит информацию символьного обозревателя
uf_smart_linked	Код модуля создан с использованием **«умной»** компоновки
uf_static_linked	Код скомпонован статически
uf_has_resources	Модуль имеет раздел ресурсов

A.4. Разделы

Помимо заголовка раздела, все данные файла PPU разделены на блоки данных, которые позволяют легко добавлять блоки данных без потери обратной совместимости. Они похожи на два куска форматов Electronic Arts IFF и Microsoft RIFF.Каждый «кусок» (`tppuentry`) имеет следующий формат, и куски могут быть вложенными.

Таблица A.4. Формат данных «куска».

Смещение	Размер (в байтах)	Описание
00h	**1**	Тип блока *(**1** – основной, **2** – вложенный)*.
01h	**1**	Идентификатор блока.
02h	**4**	Размер этого блока данных.
06h+	<переменный>	Данные для этого блока.

Каждый основной раздел куска должен заканчиваться концом куска. Вложенные куски используются для записей, классов или полей объекта.

Чтобы прочитать запись, вы можете просто вызвать `ppufile.readentry: byte`, который возвращает поле `tppuentry.nr`, которое содержит тип записи. В общем случае это работает так *(пример для идентификаторов)*:

```
repeat
   b:=ppufile.readentry;
   case b of
   ib<etc> : begin end;
ibendsyms : break;
   end;
until false;
```

Возможные типы записей находятся в ppu.pas, но краткое описание наиболее распространённых из них находится в таблице A.5.

Таблица A.5. Возможные типы PPU Entry.

Символьное имя	Размещение	Описание

ibmodulename	General	Имя этого модуля
ibsourcefiles	General	Имя исходного файла
ibusedmacros	General	Имя и состояние используемых макросов
ibloadunit	General	Модули, используемые этим модулем
inlinkunitofiles	General	Объектные файлы, связанные с этим модулем
iblinkunitstaticlibs	General	Статические библиотеки, связанные с этим модулем
iblinkunitsharedlibs	General	Общедоступные библиотеки, связанные с этим модулем
ibendinterface	General	Конец раздела общей информации (General)
ibstartdefs	Interface	Начало объявлений
ibenddefs	Interface	Конец объявлений
ibstartsyms	Interface	Начало символьных данных
ibendsyms	Interface	Конец символьных данных
ibendimplementation	Implementation	Конец данных исполняемого блока
ibendbrowser	Browser	Конец раздела обозревателя
ibend	General	Конец файла модуля

Затем вы можете обработать каждый тип записи по своему усмотрению. О пропуске непрочитанных байтов в записи и правильном чтении следующей записи позаботится `ppufile.readentry`. Специальная функция `skipuntilentry(untilb:byte):boolean;` будет читать файл ppufile до тех пор, пока не найдёт вход `untilb` в основной записи.

Разбор записи можно сделать с помощью функций `ppufile.getxxx`. Доступные функции:

```
procedure ppufile.getdata(var b;len:longint);
function getbyte:byte; function getword:word;
function getlongint:longint;
function getreal:ppureal;
function getstring:string;
```

Чтобы проверить, достигли ли вы конца записи, вы можете использовать функцию:`function EndOfEntry:boolean;`

ПРИМЕЧАНИЯ

1. `ppureal` – это лучше, чем `real`, потому что создаётся для процессора, для которого создан модуль. В настоящее время это `extended` для i386 и `single` для m68k.
2. `ibobjectdef` и `ibrecorddef` имеют сохранённые объявления и раздел идентификаторов для самих себя. Так что вы должны делать рекурсивный вызов.
   См. правильное выполнение в ppudump.pp.

Полный список записей и содержимое их полей можно найти в ppudump.pp.

A.5. Создание ppu-файлов

Создание нового ppu-файла происходит почти также как его чтение. Сначала вам нужно инициализировать объект и создать его:

```
ppufile:=new(pppufile,init('output.ppu'));
ppufile.createfile;
```

После этого вы можете просто записать все необходимые данные. Вам придётся позаботиться о том, что вы пишите, по крайней мере, создать основные записи для разделов:

```
ibendinterface
ibenddefs
ibendsyms
ibendbrowser (только если вы установили uf_has_browser!)
ibendimplementation
ibend
```

Создание записей немного отличается от их чтения. Вам нужно сначала поместить все записи в `ppufile.putxxx`:

```
procedure putdata(var b;len:longint);
procedure putbyte(b:byte);
procedure putword(w:word);
procedure putlongint(l:longint);
procedure putreal(d:ppureal);
procedure putstring(s:string);
```

После помещения всех данных в запись, вам необходимо вызвать `ppufile.writeentry(ibnr:byte)`, где `ibnr` – это номер записи, которую вы создаёте.

В конце файла вам необходимо вызвать `ppufile.writeheader` для записи нового заголовка в файл. Это автоматически заботится о новом размере ppu-файла. Когда это сделано, вы можете вызвать `ppufile.closefile` и освободить объект.

Дополнительные функции/переменные, доступные для записи:

```
ppufile.NewHeader;
ppufile.NewEntry;
```

Они позволяют вам создать чистый заголовок или запись. Обычно они

вызываются автоматически в `ppufile.writeentry`, так что нет необходимости вызывать эти методы.

Вы можете вызвать

```
ppufile.flush;
```

чтобы очистить текущий буферы диска, и вы можете установить

```
ppufile.do_crc:boolean;
```

в `False`, если вы не хотите, чтобы контрольные суммы были обновлены при записи на диск. Это необходимо, если вы, например, пишите обозреватель данных.

ПРИЛОЖЕНИЕ B: СТРУКТУРА ДЕРЕВА ИСХОДНОГО КОДА КОМПИЛЯТОРА И RTL

B.1. Дерево исходного кода компилятора

Все файлы исходного кода компилятора находятся в нескольких каталогах, как правило, общие для всех процессоров части находятся в source/compiler. Подкаталоги имеются для каждого из поддерживаемых процессоров и целевых операционных систем.

Для получения дополнительной информации о структуре компилятора см.Руководство программиста Free Pascal, которое содержит некоторые данные о внутреннем устройстве компилятора.

Каталог compiler также содержит подкаталог `utils`, который содержит основные утилиты для создания и сохранения файлов сообщений.

B.2. Дерево исходного кода RTL

Дерево исходных кодов RTL делится на множество подкаталогов, но оно хорошо структурированное и лёгкое для понимания. В основном, оно состоит из трёх частей:

1. ОС-зависимый каталог. Содержит файлы, которые отличаются для каждой операционной системы. При компиляции RTL вы должны делать это здесь. Следующие каталоги существуют:
   - amiga для AMIGA.
   - atari для ATARI.
   - beos для BEOS. Имеет один подкаталог для каждого поддерживаемого процессора.
   - bsd – общие файлы для различных BSD-платформ.
   - darwin для `unix`-совместимости на Mac OS.
   - embedded – шаблон для встраивания в целей.
   - emx – OS/2, использующая расширитель EMX.
   - freebsd для платформы FREEBSD.
   - gba Game Boy Advanced.
   - go32v2 для DOS, использующей расширитель GO32v2.
   - linux для платформ LINUX. Имеет один подкаталог для каждого поддерживаемого процессора.
   - macos для платформы OS.
   - morphos для платформы MorphOS.
   - nds для платформы Nintendo DS.

- netbsd для платформ NETBSD. Имеет один подкаталог для каждого поддерживаемого процессора.
- netware для платформ Novell netware.
- netwlibc для платформ Novell netware, использующих библиотеку C.
- openbsd для платформ OpenBSD.
- os2 для OS/2.
- palmos для платформ PALMOS, базирующихся на процессоре Dragonball.
- posix для интерфейсов posix *(используемых для облегчения портирования)*.
- solaris для платформ SOLARIS. Имеет один подкаталог для каждого поддерживаемого процессора.
- symbian для ОС мобильных телефонов symbian.
- qnx для QNX REALTIME PLATFORM.
- unix для общих интерфейсов unix *(используемых для облегчения портирования)*.
- win32 для 32-разрядный платформ Windows.
- win64 для 64-разрядный платформ Windows.
- wince для платформ Windows CE *(arm CPU)*.
- posix для интерфейсов posix interfaces (используемых для облегчения портирования).

2. Зависимый от процессора каталог. Содержит файлы, которые не зависят от операционной системы, но зависят от типа процессора. Каталог содержит, в основном, оптимизированные процедуры для конкретных процессоров. Следующие каталоги существуют:
   - arm для процессоров серии ARM.
   - i386 для процессоров серии Intel 80x86.
   - m68k для процессоров серии Motorola 680x0.
   - powerpc для процессора PowerPC.
   - powerpc64 для процессора PowerPC 64-bit.
   - sparc для процессора SUN SPARC.
   - x86_64 для Intel-совместимых **64**-разрядных процессоров, таких как AMD64.
3. Независимый от ОС и процессора каталог: inc. Содержит полные модули и подключаемые файлы, содержащие интерфейсные части модулей, а также общие версии процедур для конкретного процессора.
4. Расширения Object Pascal *(в основном совместимые с Delphi модули)* находятся в каталоге **objpas**. Модули sysutils и classes находятся в отдельных подкаталогах каталога **objpas**.

ПРИЛОЖЕНИЕ C: ОГРАНИЧЕНИЯ КОМПИЛЯТОРА

Имеются некоторые ограничения, присущие компилятору:

1. Объявления процедур и функций могут быть вложенными до **32**-уровня. Это значение может быть изменено с помощью константы `maxnesting`.
2. В программе может быть использовано не более **1024** модулей. Вы можете изменить это значение, переопределив константу `maxunits` в исходном коде компилятора.
3. Максимальный уровень вложенности макросов препроцессора – это **16**. Можно изменить, поменяв значение `max_macro_nesting`.
4. Размер массивов ограничен 2 ГБ по умолчанию в режиме **32**-разрядного процессора.

Чтобы узнать об ограничениях конкретного процессора, см. раздел 6.8. Ограничения стека[99].

ПРИЛОЖЕНИЕ D: РЕЖИМЫ КОМПИЛЯТОРА

Здесь приведён точный список отличий в различных режимах работы компилятора. Режим можно установить переключателем `$Mode`, или переключателем командной строки.

D.1. Режим FPC

Этот режим выбирается переключателем `$MODE FPC`. В командной строке это означает, что вы не используете какой-либо другой переключатель совместимости режимов. Этот режим используется по умолчанию компилятором (`-Mfpc`). Это значит, что:

1. Вы должны использовать адрес оператора, чтобы назначить процедурные переменные.
2. Раннее объявление должно в точности повторять реализацию функции/процедуры. В частности, вы не можете пропустить параметры, когда реализуете функцию или процедуру.
3. Перегрузка функций не допускается.
4. Вложенные комментарии не допускаются.
5. Модуль Objpas ***НЕ*** загружен.
6. Вы можете использовать тип `cvar`.
7. `PChar` автоматически преобразуется в `string`.
8. Строки являются короткими строками `(shortstrings)` по умолчанию.

D.2. Режим TP

Этот режим выбирается переключателем `$MODE TP`. Он пытается эмулировать, насколько это возможно, поведение компилятора `Turbo Pascal 7`. В командной строке этот режим выбирается переключателем `-Mtp`.

1. Размер перечисления по умолчанию равен один байт, если перечисление содержит менее **257** элементов.
2. Вы не можете использовать адрес оператора, чтобы назначить процедурные переменные.
3. Раннее объявление не должно в точности повторять реализацию функции/процедуры. В частности, вы можете пропустить параметры, когда реализуете функцию или процедуру.
4. Перегрузка функций не допускается.
5. Модуль Objpas ***НЕ*** загружен.
6. Вложенные комментарии не допускаются.
7. Вы не можете использовать тип `cvar`.
8. Строки являются короткими строками `(shortstrings)` по умолчанию.

D.3. Режим Delphi

Этот режим выбирается переключателем `$MODE DELPHI`. Он пытается

эмулировать, насколько это возможно, поведение Delphi 4 и выше. В командной строке этот режим выбирается переключателем -Mdelphi.

1. Вы не можете использовать адрес оператора, чтобы назначить процедурные переменные.
2. Раннее объявление не должно в точности повторять реализацию функции/процедуры. В частности, вы можете пропустить параметры, когда реализуете функцию или процедуру.
3. Ansistring установлен по умолчанию, это значит, что $MODE DELPHI неявно подразумевает {$H ON}.
4. Перегрузка функций не допускается.
5. Вложенные комментарии не допускаются.
6. Модуль Objpas загружен сразу после модуля system. Одним из следствий этого является то, что тип Integer переопределяется как Longint.
7. Параметры в методах класса могут иметь такие же имена, как свойства класса *(хотя это плохая практика программирования)*.

D.4. Режим OBJFPC

Этот режим выбирается переключателем $MODE OBJFPC. В командной строке этот режим выбирается переключателем -Mobjfpc.

1. Вы должны использовать адрес оператора, чтобы назначить процедурные переменные.
2. Раннее объявление должно в точности повторять реализацию функции/процедуры. В частности, вы не можете пропустить параметры, когда реализуете функцию или процедуру.
3. Перегрузка функций допускается.
4. Вложенные комментарии допускаются.
5. Модуль Objpas загружен сразу после модуля system. Одним из следствий этого является то, что тип Integer переопределяется как Longint.
6. Вы можете использовать тип cvar.
7. PChar автоматически преобразуется в string.
8. Параметры в методах класса не могут иметь такие же имена, как свойства класса.
9. Строки являются короткими строками (shortstrings) по умолчанию. Вы можете использовать переключатель командной строки -Sh или переключатель {$H+} для изменения этого поведения.

D.5. Режим MACPAS

Этот режим выбирается переключателем $MODE MACPAS. В командной строке этот режим выбирается переключателем -Mmacpas. В основном это включает ряд дополнительных функций:

1. Поддержка директивы $SETC.
2. Поддержка директив $IFC, $ELSEC и $ENDC.

3. Поддержка конструкции `UNDEFINED` в макросах.
4. Поддержка `TRUE` и `FALSE` как значений в макровыражениях.
5. Макросу могут присваиваться шестнадцатеричные числа, такие как **$2345**.
6. Ключевое слово `Implementation` может быть пропущено в соответствующем разделе, если он пустой.
7. Модификатор `cdecl` может быть сокращён в C.
8. Модификатор `UNIV` для типов в списке параметров принимается, но в противном случае игнорируется.
9. … *(многоточие)* допускается в объявлениях процедур, функционально эквивалентно ключевому слову `varargs`.

ПРИМЕЧАНИЕ

Макрос называется «Compiler Variables» в диалекте Mac OS.

В настоящее время следующие расширения Pascal для Mac OS ещё не поддерживаются в режиме `MACPAS`:

- Вложенная процедура не может быть фактическим параметров для процедуры.
- Не анонимные процедурные типы в формальных параметрах.
- Объявленные в интерфейсе внешние процедуры должны иметь директиву `External`.
- `Continue` вместо `Cycle`.
- `Break` вместо `Leave`.
- `Exit` не должен иметь для выхода имени процедуры в качестве параметра. Вместо этого, возвращаемое функцией значение может быть поставлено в качестве параметра.
- Нет распространения `uses`.
- Директивы компилятора, определённые в интерфейсе, не экспортируются.

ПРИЛОЖЕНИЕ E: ИСПОЛЬЗОВАНИЕ fpcmake

E.1. Введение

Free Pascal поставляется со специальным инструментом создания файлов fpcmake, который можно использовать для создания Makefile для использования с GNU make. Все исходные файлы команды Free Pascal скомпилированы с помощью этой системы. fpcmake использует файл Makefile.fpc и создаёт из него файл Makefile, основанный на настройках в Makefile.fpc.

В следующих разделах описано, какие настройки могут быть установлены в Makefile.fpc, какие переменные устанавливаются с помощью fpcmake, какие переменные эта программа ожидает и какие задачи определяет. Затем разъясняются некоторые параметры, которые получаются в результате в Makefile.

E.2. Функциональность

fpcmake генерирует makefile, подходящий для GNU make, который можно использовать для:

1. Компиляции модулей и программ, пригодных для тестирования или окончательного распространения.
2. Компиляции примеров модулей и программ отдельно.
3. Установки скомпилированных модулей и программ в стандартных местах.
4. Создания архивов для распространения сгенерированных программ и модулей.
5. Очистки после компиляции и тестирования.

fpcmake знает, как работает компилятор Free Pascal, какие опции командной строки он использует, как он ищет файлы и т.п., и использует эти знания для конструирования продуманных командных строк.

В частности, он создаёт следующие объекты в окончательном файле makefile:

`all` – создаёт все модули и программы.
`debug` – создаёт все модули и программы с включением отладочной информации.
`smart` – создаёт все модули и программы в версии умной компоновки.
`examples` – создаёт все примеры модулей и программ.
`shared` – создаёт все модули и программы в версии общедоступной библиотеки *(в настоящее время отключено)*.
`install` – устанавливает все модули и программы.
`sourceinstall` – устанавливает все исходные файлы в дерево исходных

файлов Free Pascal.

exampleinstall – устанавливает любые примеры программ и модулей.

distinstall – устанавливает все модули и программы, а также примеры модулей и программ.

zipinstall – создаёт архив программ и модулей, который может быть использован для их установки в другом месте, то есть создаёт архив, который можно использовать для распространения модулей и программ.

zipsourceinstall – создаёт архив исходных кодов модулей и программ, который можно использовать для распространения исходных кодов.

zipexampleinstall – создаёт архив примеров программ и модулей, которые можно использовать для распространения примеров программ и модулей.

zipdistinstall – создаёт архив как обычных программ. Так и примеров программ и модулей. Этот архив можно использовать для установки этих модулей и программ в другом месте, то есть создаёт архив, который можно использовать для распространения.

clean – удаляет все файлы, которые были созданы при компиляции.

distclean – удаляет все файлы, которые были созданы при компиляции, а также любые архивы, примеры или файлы, оставленные примерами.

cleanall – то же, что и `clean`.

info – выводит информацию на экран об используемых программах, размещении файлов и каталогов, где всё это происходит во время установки и т. д.

Каждый из этих объектов может быть точно сконфигурирован или даже полностью перезаписан с помощью файла конфигурации Makefile.fpc.

E.3. Использование

fpcmake читает файл Makefile.fpc и преобразует его в Makefile, пригодный для чтения GNU make для компиляции ваших проектов. Он похож по функциональности на GNU configure или Imake для создания X-проектов.

fpcmake принимает имена файлов из описания файлов в makefile как аргументы командной строки. Для каждого из этих файлов он будет создавать Makefile в той же директории, где размещён файл, перезаписывая любой существующий файл с таким именем.

Если опции не указаны, то он просто пытается прочитать файл Makefile.fpc в текущем каталоге, и пытается из него создать Makefile, если указана опция `-m`. Любые ранее созданные и существующие файлы Makefile будут удалены.

Если передана опция `-p`, то вместо Makefile генерируется Package.fpc. Файл Package.fpc описывает пакеты и их зависимости от других пакетов.

Кроме того, распознаются следующие опции командной строки:

-p – будет создан файл Package.fpc.
-w – генерируется файл Makefile.
-T targets – Поддержка только указанных операционных систем. `Targets` – это список разделённых запятыми целевых операционных систем. Правила будут записаны только для указанных ОС.
-v – будет больше подробностей.
-q – «тихий» режим.
-h – выводит небольшие справочные сообщения на экран.

E.4. Формат файла конфигурации

Этот раздел описывает правила, которые могут находиться в файле, обработанным при помощи fpcmake. Файл Makefile.fpc – это простой текстовый ASCII-файл, содержащий ряд заранее определённых разделов как в WINDOWS .ini-файле или конфигурационном файле Samba.

Он выглядит примерно так:

```
[package]
name=mysql
version=1.0.5

[target]
units=mysql_com mysql_version mysql
examples=testdb

[require]
libc=y

[install]
fpcpackage=y

[default]
fpcdir=../..
```

Ниже описаны разделы *(в алфавитном порядке)*.

E.4.1. clean

Определяет правила для очистки каталога модулей и программ. Следующие записи распознаются:

units – имена всех модулей, которые должны быть удалены при очистке. Не указывайте расширения, makefile будет их добавлять сам.
files – имена дополнительных файлов *(не файлы модулей)*, которые должны быть удалены. Указывайте полные имена. Файлы таблиц строковых ресурсов *(файлы .rst)* будут удалены, если они указаны в разделе `files`.

E.4.2. compiler

В этом разделе можно указать различные опции компилятора, такие как расположение нескольких каталогов и пути поиска.Следующие основные ключевые слова распознаются:

options – значение этого ключа будет передано в компилятор *(дословно)* в качестве параметра командной строки.

version – если конкретная или минимальная версия компилятора необходима для компиляции модулей или программ, то эта версия должна быть указана здесь.

Следующие ключи могут использоваться для управления местоположением различных каталогов, используемых компилятором:

unitdir – список каталогов, разделённых двоеточием, которые должны быть добавлены в путь поиска модулей компилятора *(используется опция `-Fu`)*.

librarydir – список каталогов, разделённых двоеточием, которые должны быть добавлены в путь поиска библиотек компилятора *(используется опция `-Fl`)*.

objectdir – список каталогов, разделённых двоеточием, которые должны быть добавлены в путь поиска объектных файлов *(используется опция `-Fo`)*.

targetdir – определяет каталог, где должны быть скомпилированные программы *(используется опция `-FE`)*.

sourcedir – список каталогов, разделённых пробелами, где могут находиться исходные коды. Будет использован параметр `vpath` в GNU make.

unittargetdir – определяет каталог, где должны быть размещены скомпилированные модули *(используется опция `-FU`)*.

includedir – список каталогов, разделённых двоеточием, которые должны быть добавлены в путь поиска подключаемых файлов компилятора *(используется опция `-Fi`)*.

E.4.3. Default

Раздел `default` содержит некоторые настройки по умолчанию. Распознаются следующие ключи:

cpu – определяет целевой процессор по умолчанию, для которого Makefile должен компилировать модули и программы. По умолчанию это определяется из информации компилятора.

dir – определяет все подкаталоги, которые подойти для указанных объектов.

fpcdir – определяет каталог, где находится всё дерево исходных кодов Free Pascal. Ниже этого каталог Makefile ожидает найти rtl дерево каталогов пакетов.

rule – определяет правило по умолчанию для исполняемого файла. fpcmake будет проверять, что это правило выполняется, если сделать

исполняемый файл без аргументов, то есть без явной цели.

target – определяет целевую операционную систему по умолчанию для каждого Makefile, для которого должны быть скомпилированы модуль или программа. По умолчанию параметр устанавливается из целевого компилятора по умолчанию.

E.4.4. Dist

Раздел `Dist` управляет генерацией пакета дистрибутива. Пакет дистрибутива – это набор архивных файлов *(zip-файлы или tar-файлы на unix -системах)*, который может быть использован для распространения пакета.

Следующие ключи могут быть в этом разделе:

destdir – определяет каталог, в котором должны быть размещены zip-файлы.

zipname – имя архивного файла, который должен быть создан. Если `zipname` не указан, то по умолчанию это будет имя пакета.

ziptarget – это задачи, которые должны быть выполнены перед созданием архивного файла. По умолчанию это `install`.

E.4.5. Install

Содержит инструкции для установки скомпилированных модулей и программ. Следующие ключевые слова распознаются:

basedir – каталог, который используется как основной каталог для установки модулей. По умолчанию это `prefix`, добавляемый к **`/lib/fpc/FPC_VERSION`** для LINUX или просто каталог **`prefix`** на других платформах.

datadir – каталог, где будут установлены файлы данных, то есть каталог, указанный с ключевым словом `Files`.

fpcpackage – логический ключ. Если этот ключ указан и равен `y`, то файлы будут установлены как пакет fpc в каталог модулей Free Pascal, то есть в отдельный каталог. Каталог будет иметь имя, указанное в разделе `package`.

files – дополнительные файлы данных для установки в каталог, указанный в ключе `datadir`.

prefix – каталог, в который выполняются все установки. Он соответствует аргументу `prefix` в GNU configure. Используется для установки программ и модулей. По умолчанию это **`/usr`** в LINUX и **`/pp`** на всех других платформах.

units – дополнительные модули, которые должны быть установлены и которые не являются частью целевых модулей. Целевые модули будут установлены автоматически.

Модули будут установлены в подкаталоге **`units/$(OS_TARGET)`** записи `dirbase`.

E.4.6. Package

Если компилируется пакет *(то есть коллекция модулей, которые работают вместе)*, то этот раздел используется для хранения информации пакета. Следующая информация может быть записана:

name – имя пакета. При установке в каталог пакета это имя будет использовано для создания директории *(если это не будет переопределено одной из опций установки)*.

version – версия пакета.

main – если пакет является частью другого пакета, то этот ключ определяем, частью какого пакета является данный пакет.

E.4.7. Prerules

Всё. Что находится в этом разделе, будет вставлено «как есть» в makefile ***ПЕРЕД*** целевыми правилами, которые сгенерированы fpcmake. Это значит, что любые переменные, которые обычно определяют правила fpcmake, не должны использоваться в этом разделе.

E.4.8. Requires

Этот раздел используется для указания зависимостей от внешних пакетов (например, модулей) или инструментов. Следующие ключевые слова могут использоваться:

fpcmake – минимальная версия fpcmake, которая необходима makefile.fpc.

packages – другие пакеты, которые должны быть скомпилированы перед этим пакетом и могут быть скомпилированы. Учтите, что это также добавит все пакеты к этим пакетам в зависимость от этого пакета. По умолчанию в этот список добавлена Free Pascal Run-Time Library.

libc – логическое значение, которое указывает, нужна ли этому пакету библиотека C.

nortl – логическое значение, которое предотвращает добавление Free Pascal Run-Time Library в требуемые пакеты.

unitdir – эти каталоги будут добавлены в путь поиска модулей компилятора.

packagedir – список каталогов пакета. Пакеты в этих каталогах будут созданы прежде, чем будет создан текущий пакет.

tools – список исполняемых файлов, дополнительные инструменты, которые необходимы. Полный путь к этим инструментам будет определён в makefile как переменная с тем же именем, что и имя инструмента, только в верхнем регистре. Например, следующее определение:

```
tools=upx
```

приведёт к определению переменной с именем UPX, которая будет содержать полный путь к исполняемому файлу upx.

E.4.9. Rules

В этом разделе могут находиться правила зависимостей для модулей и любых других целей. Это раздел будет включён в конец сгенерированного файла makefile. Здесь могут быть вставлены «правила по умолчанию» (default rules), которые определены с помощью fpcmake. Если они не представлены, то fpcmake сгенерирует правило, которое будет вызывать общую версию fpc. Список стандартных целей, которые будут определены с помощью fpcmake см. в разделе E.2. Функциональность[180].

Например, можно определить цель `all`:. Если она не определена, то fpcmake сгенерирует один простой вызов `fpc_all`:

```
all: fpc_all
```

Правило `fpc_all` создаст все цели так, как это определено в разделе `Target`.

E.4.10. Target

Это очень важный раздел файла makefile.fpc. Здесь определены файлы, которые должны компилироваться, если выполняется цель `all`.

Здесь можно использовать следующие ключевые слова:

`dirs` – список каталогов, разделённых пробелом, где make также должна быть запущена.

`exampledirs` – список каталогов, разделённых пробелом, где находятся примеры программ. Объекты `examples` также будут связаны с этим списком каталогов.

`examples` – список примеров программ, разделённых пробелом, которые необходимо компилировать, если пользователь запросит компиляцию примеров. Не указывайте расширение, расширение будет добавлено.

`loaders` – список ассемблерных файлов, разделённых пробелом, которые должны быть ассемблированы. Не указывайте расширение, расширение будет добавлено.

`programs` – список имён программ, разделённых пробелом, которые необходимо компилировать. Не указывайте расширение, расширение будет добавлено.

`rsts` – список файлов rst, которые необходимо преобразовать в файлы .po для использования с GNU gettext и подпрограмм интернационализации. Эти файлы будут установлены вместе с файлами модулей.

`units` – список имён модулей, разделённых пробелом, которые необходимо компилировать. Не указывайте расширение, достаточно только имя модуля как оно представлено в разделе `uses`.

E.5. Программы, необходимые для работы с созданным makefile

По крайней мере, необходимы следующие программы, чтобы сгенерированный Makefile работал правильно:

cp – копирование программы.
date – программа, которая печатает дату.
install – программа для установки файлов.
make – программа `make`, очевидно.
pwd – программа, которая печатает текущую рабочую директорию.
rm – программа для удаления файлов.
zip – программа-архиватор zip. *(на системах Dos / Windows / OS/2)*.
tar – программа-архиватор tar *(только на системах Unix)*.

Это стандартные программы на системах LINUX, за исключением make. Для DOS, WINDOWS NT или OS/2 / eComStation, они поставляются как часть пакета Free Pascal.

Следующие программы могут оказаться необходимы, если вы используете некоторые специальные задачи/цели. Какие из этих программ необходимы, определяется настройками раздела `tools`.

cmp – сравнивает файлы в DOS и WINDOWS NT.
diff – сравнивает файлы.
ppdep – определяет зависимости. Поставляется с Free Pascal.
ppumove – перемещает модули Free Pascal.
upx – исполняемый файл упаковщика UPX.

Всё это можно найти на сайте Free Pascal FTP для DOS и WINDOWS NT. Программы ppdep и ppumove поставляются с компилятором Free Pascal.

E.6. Переменные, которые влияют на генерируемый makefile

Makefile, сгенерированный с помощью fpcmake, содержит большое количество переменных. Некоторые из них устанавливаются непосредственно в makefile, другие могут быть установлены и взяты во время установки.Эти переменные можно разделить на две группы:

- Переменные каталогов
- Переменные командной строки компилятора

Каждая группа будет обсуждаться отдельно.

E.6.1. Переменные каталогов

Первый набор переменных управляет каталогами, которые распознаны в

makefile. Они не должны устанавливаться в Makefile.fpc, но могут быть указаны в командной строке.

INCDIR – это список директорий, разделённых пробелами. Эти директории будут добавлены как подключаемые каталоги для командной строки компилятора. Каждый каталог в списке добавляет `-Fi` к опциям командной строки компилятора.

UNITDIR – это список директорий, разделённых пробелами. Эти директории будут добавляться как директории для поиска модулей в командной строке компилятора. Каждый каталог в списке добавляет `-Fu` к опциям командной строки компилятора

LIBDIR – это список директорий, разделённых пробелами. Это директории для поиска библиотек. Каждый каталог в списке добавляет `-Fl` к опциям командной строки компилятора

OBJDIR – это список директорий, разделённых пробелами. Это директории объектных файлов. Каждый каталог в списке добавляется с помощью опции `-Fo`.

E.6.2. Переменные командной строки компилятора

Следующие переменные могут быть установлены в командной строке make, они будут распознаны и интегрированы в опции командной строки:

CREATESMART – если эта переменная определена, то она говорит компилятору, что модули созданы с использованием **«умной компоновки»**. Добавляет `-CX` к опциям командной строки.

DEBUG – если определена, то компилятор будет включать отладочную информацию в генерируемый модуль или программу. Добавляет `-gl` к опциям командной строки и будет определять объявление `DEBUG`.

LINKSMART – определение этой переменной говорит компилятору, что нужно использовать **«умную компоновку»**. Добавляет `-XX` к опциям командной строки.

OPT – любые опции, которые вы хотите передать в компилятор. Содержимое `OPT` просто добавляется в командную строку компилятора.

OPTDEF – необязательное дополнительное определение опции, добавляемое в командную строку компилятора. Для него добавлен `-d`.

OPTIMIZE – если эта переменная определена, то будет добавлено `-OG2p3` в командную строку компилятора.

RELEASE – если эта переменная определена, то будут добавлены опции `-Xs -OG2p3 -n` в командную строку компилятора, а также будет

определена RELEASE.

STRIP – если эта переменная определена, то будет добавлено -Xs в командную строку компилятора.

VERBOSE – если эта переменная определена, то будет добавлено -vnwi в командную строку компилятора.

E.7. Переменные, установленные с помощью fpcmake

Makefile, сгенерированный с помощью fpcmake, содержит множество переменных makefile. fpcmake будет записывать все ключи в makefile.fpc как переменные makefile в формате SECTION_KEYNAME. Это значит, что следующий раздел:

```
[package]
name=mysql
version=1.0.5
```

в результате работы определит следующие переменные:

```
override PACKAGE_NAME=mysql
override PACKAGE_VERSION=1.0.5
```

Большинство объектов и правил создано с использованием этих переменных. Они будут перечислены ниже, вместе с другими переменными, которые определены с помощью fpcmake.

Определены следующие группы переменных:

- Переменные каталогов.
- Имена программ.
- Расширения файлов.
- Целевые файлы.

Каждая из этих групп обсуждается далее.

E.7.1. Переменные каталогов

Следующие каталоги компилятора определяются с помощью makefile:

BASEDIR – устанавливает текущий каталог, если команда pwd доступна. Если нет, то устанавливается в '.'.

COMPILER_INCDIR – разделённый пробелами список путей к подключаемым файлам. Каждый каталог в списке добавляется с -Fi и добавляется к опциям компилятора. Устанавливается с помощью ключевого слова incdir в разделе Compiler.

COMPILER_LIBDIR – разделённый пробелами список путей к библиотекам. Каждый каталог в списке добавляется с -Fl и добавляется к опциям

компилятора. Устанавливается с помощью ключевого слова `libdir` в разделе `Compiler`.

COMPILER_OBJDIR – разделённый пробелами список путей к каталогам объектных файлов. Каждый каталог в списке добавляется с `-Fo` и добавляется к опциям компилятора. Устанавливается с помощью ключевого слова `objdir` в разделе `Compiler`.

COMPILER_TARGETDIR – этот каталог добавляется как выходной каталог компилятора, где сохраняются все модули и исполняемые файлы, то есть добавляется к `-FE`. Устанавливается с помощью ключевого слова `targetdir` в разделе `Compiler`.

COMPILER_TARGETUNITDIR – если установлен, то каталог добавляется как выходной каталог компилятора, где сохраняются все модули и исполняемые файлы, то есть добавляется к `-FU`. Устанавливается с помощью ключевого слова `targetdir` в разделе `Dirs`.

COMPILER_UNITDIR – разделённый пробелами список каталогов модулей. Каждый каталог в списке добавляется с `-Fu` и добавляется к опциям компилятора. Устанавливается с помощью ключевого слова `unitdir` в разделе `Compiler`.

GCCLIBDIR - (только `LINUX`) – устанавливает каталог, где находится `libgcc.a`. Если `needgcclib` установлена в `True` в разделе `Libs`, то этот каталог добавляется к командной строке компилятора с `-Fl`.

OTHERLIBDIR – разделённый пробелами список путей к библиотекам. Каждый каталог в списке добавляется с `-Fl` и добавляется к опциям компилятора. Если не определён на `linux`, то добавляется содержимое файла `/etc/ld.so.conf`.Следующие директории используются для инсталляции:

INSTALL_BASEDIR – является базовым для всех каталогов, где установлены модули. По умолчанию на `LINUX` установлен в `$(INSTALL_PREFIX)/lib/fpc/$(RELEASEVER)`. На других системах он установлен в `$(PREFIXINSTALLDIR)`. Вы можете также установить его с помощью переменной `basedir` в разделе `Install`.

INSTALL_BINDIR – установлен в `$(INSTALL_BASEDIR)/bin` на LINUX и в `$(INSTALL_BASEDIR)/bin/$(OS_TARGET)` на других системах. Это место, где устанавливаются бинарные файлы.

INSTALL_DATADIR – каталог, где устанавливаются файлы данных. Устанавливается ключом `Data` в разделе `Install`.

INSTALL_LIBDIR – устанавливается в `$(INSTALL_PREFIX)/lib` на LINUX и в `$(INSTALL_UNITDIR)` на других системах.

INSTALL_PREFIX – устанавливается в `/usr/local` на LINUX, `/pp` на DOS или WINDOWS NT. Устанавливается ключом `prefix` в разделе `Install`.

INSTALL_UNITDIR – определяет, где будут установлены модули. Устанавливается в `$(INSTALL_BASEDIR)/units/$(OS_TARGET)`. Если модули скомпилированы как пакет, то `$(PACKAGE_NAME)` добавляется к каталогу.

E.7.2. Целевые переменные

Второй набор переменных управляет целями/задачами, которые создаются с помощью makefile. Они создаются с помощью fpcmake, поэтому вы можете использовать их в ваших правилах, но вы не должны самостоятельно присваивать им значения.

TARGET_DIRS – это список каталогов, которые будут пропущены при компиляции. Устанавливается ключом `Dirs` в разделе `Target`.

TARGET_EXAMPLES – список примеров программ, которые должны быть откомпилированы. Устанавливается ключом `examples` в разделе `Target`.

TARGET_EXAMPLEDIRS – это список каталогов, которые будут пропущены при компиляции примеров. Устанавливается ключом `exampledirs` в разделе `Target`.

TARGET_LOADERS – список разделённых пробелами имён, которые идентифицируют загрузчики для компиляции. В основном, это используется в исходных кодах компилятора RTL. Устанавливается ключом `loaders` в разделе `Target`.

TARGET_PROGRAMS – это список имён исполняемых файлов, которые будут компилироваться. `Makefile` добавляет `$(EXEEXT)` для этих имён. Устанавливается ключом `programs` в разделе `Target`.

TARGET_UNITS – это список имён модулей, которые будут компилироваться. Makefile добавляет `$(PPUEXT)` для каждого из этих имён, чтобы сформировать имя файла модуля. Имя исходного кода формируется путём добавления `$(PASEXT)`. Устанавливается ключом `units` в разделе `Target`.

ZIPNAME – имя архива, который будет создан с помощью `makefile`. Устанавливается ключом `zipname` в разделе `Zip`.

ZIPTARGET – объект, который создаётся перед созданием архива. Этот объект собирается первым. Если сборка прошла успешно, то будет создан zip-архив. Устанавливается ключом `ziptarget` в разделе `Zip`.

E.7.3. Переменные командной строки компилятора

Следующие переменные управляют командной строкой компилятора:

CPU_SOURCE – исходный тип процессора, добавленный как определение для командной строки компилятора. Определяется при помощи самого Makefile.

CPU_TARGET – тип целевого процессора, добавленный как определение для командной строки компилятора. Определяется при помощи самого Makefile.

OS_SOURCE – какая платформа используется в makefile. Определяется автоматически.

OS_TARGET – какая платформа будет для компилирования. Добавляется к опциям командной строки компилятора `-T`.

E.7.4. Имена программ

В объектах makefile используются следующие переменные для имён программ:

AS – ассемблер. По умолчанию установлено значение `as`.

COPY – программа копирования файлов. По умолчанию установлено значение `cp -fp`.

COPYTREE – программа копирования дерева каталогов. По умолчанию установлено значение `cp -frp`.

CMP – программа сравнения файлов. По умолчанию установлено значение `cmp`.

DEL – программа удаления файла. По умолчанию установлено значение `rm -f`.

DELTREE – программа удаления каталога. По умолчанию установлено значение `rm -rf`.

DATE – программа отображения даты.

DIFF – программа создания файлов `diff`.

ECHO – программа `echo`.

FPC – исполняемый файл компилятора `Free Pascal`. По умолчанию ppc386.exe.

INSTALL – программа для установки файлов. По умолчанию `install -m 644` на LINUX.

INSTALLEXE – программа для установки исполняемых файлов. По умолчанию `install -m 755` на LINUX.

LD – компоновщик. По умолчанию `ld`.

LDCONFIG - *(только LINUX)* – программа, используемая для обновления загрузчика КЭШа.

MKDIR – программа для создания каталогов, если они ещё не существуют. По умолчанию `install -m 755 -d`.

MOVE – программа перемещения файла. По умолчанию `mv -f`.

PP – исполняемый файл компилятора Free Pascal. По умолчанию ppc386.exe.

PPAS – имя сценария оболочки, созданного с помощью компилятора, если была указана опция `-s`. Эта команда будет выполняться после компиляции, если опция `-s` была обнаружена среди опций.

PPUMOVE – программа для перемещения модулей в одну большую библиотеку модулей.

PWD – программа `pwd`.

SED – редактор программы в потоке. По умолчанию `sed`.

UPX – исполняемый файл упаковщика для сжатия ваших исполняемых файлов в самораскрывающийся сжатый исполняемый файл.

ZIPPROG – программа zip для сжатия файлов. Zip-объекты создаются с помощью этой программы.

E.7.5. Расширения файлов

Следующие переменные обозначают расширения файлов. Эти переменные включают в себя **.** ***(точку)*** расширения. Они добавляются к именам объектов.

ASMEXT – расширение ассемблерных файлов, создаваемых компилятором.

LOADEREXT – расширение ассемблерных файлов, которые входят в код запуска исполняемого файла.

OEXT – расширение объектных файлов, которые создаются компилятором.

PACKAGESUFFIX – суффикс, который добавляется к именам пакетов в zip-объектах. Такие службы, как пакеты, могут быть созданы для разных операционных систем.

PPLEXT – расширение файлов модулей общедоступных библиотек.

PPUEXT – расширение по умолчанию для модулей.

RSTEXT – расширение .rst файлов строковых ресурсов.

SHAREDLIBEXT – расширение общедоступных библиотек.

SMARTEXT – расширение файлов ассемблерных модулей, созданных в режиме **«умной компоновки»**.

STATICLIBEXT – расширение статических библиотек.

E.7.6. Целевые файлы

Следующие переменные определены для облегчения создания целей и правил:

COMPILER – это полная командная строка компилятора со всеми добавленными опциями, после всех переменных Makefile, которые были проанализированы.

DATESTR – содержит дату.

UNITPPUFILES – список файлов модулей, которые будут созданы. Это только список объектов модулей с правильными расширениями модулей.

E.8. Правила и цели, созданные с помощью fpcmake

makefile.fpc определяет серию целей, которые могут быть вызваны вашими собственными целями. Они имеют имена, которые напоминают имена по умолчанию *(такие как `'all', 'clean'`)*, только с приставкой `fpc_`.

E.8.1. Шаблонные правила

makefile создаёт следующие шаблонные правила:

units – как сделать модуль Pascal из исходного файла Pascal.

executables – как сделать исполняемый файл Pascal из исходного файла Pascal.

object file – как сделать объектный файл из ассемблерного файла.

E.8.2. Правила сборки

Следующие цели сборки определены:

fpc_all – сборка всех модулей и исполняемых файлов, а также загрузчиков. Если определена `DEFAULTUNITS`, то исполняемые файлы исключаются из цели.

fpc_debug – то же, что и `fpc_all`, только с включением отладочной информации.

fpc_exes – создавать все исполняемые файлы в EXEOBJECTS.

fpc_loaders – создавать все файлы в LOADEROBJECTS.

fpc_packages – создавать все необходимые пакеты для создания файлов.

fpc_shared – создавать все модули как динамические библиотеки.

fpc_smart – создавать все модули в режиме **«умной компоновки»**.

fpc_units – создавать все модули в UNITOBJECTS.

E.8.3. Правила очистки

Следующие цели очистки определены:

fpc_clean – удаляет все файлы, которые были созданы в результате работы fpc_all.

fpc_distclean – то же, что и предыдущая команда, но только удаляет все объекты, модули и ассемблерные файлы, которые имеются.

E.8.4. Правила архивации

Следующие цели архивации определены:

fpc_zipdistinstall – создавать дистрибутив пакета.

fpc_zipinstall – создавать инсталлятор zip скомпилированных модулей пакета.

fpc_zipexampleinstall – создавать zip файлов примеров.

fpc_zipsourceinstall – создавать zip исходных файлов.

zip создаётся с помощью программы ZIPEXE. В LINUX будет создаваться файл .tar.gz.

E.8.5. Правила инсталляции

fpc_distinstall – цель, которая вызывает цели install и exampleinstall.

fpc_install – установка модулей.

fpc_sourceinstall – установка исходных файлов, если создаётся дистрибутив.

fpc_exampleinstall – установка примеров, если создаётся дистрибутив.

E.8.6. Информативные правила

Есть только одна цель, которая создаёт информацию об используемых переменных, правилах и целях: `fpc_info`.

Следующая информация о makefile представлена:

- Общая информация о конфигурации: размещение makefile, версия компилятора, целевая ОС, процессор.
- Каталоги, используемые компилятором.
- Все каталоги, где будут устанавливаться файлы.
- Все объекты, которые будут созданы.
- Все определённые инструменты.

ПРИЛОЖЕНИЕ F: КОМПИЛЯЦИЯ КОМПИЛЯТОРА

F.1. Введение

Команда Free Pascal периодически выпускает полностью готовые пакеты, содержащие компилятор и модули, готовые для использования, так называемые «релизы». После выпуска релиза работа над компилятором продолжается, устраняются ошибки и добавляются новые функции. Команда Free Pascal не создаёт новые релизы каждый раз, когда они меняют что-то в компиляторе. Вместо этого они предлагают для общего доступа и компилирования исходные коды. Автоматический процесс, который создаёт скомпилированные версии RTL и компилятора также выполняется ежедневно и выкладывается в Интернете *(если сборка завершена)*. Zip-файлы с исходными кодами также создаются ежедневно.

Однако существуют обстоятельства, когда компилятор должен быть откомпилирован вручную. Если были сделаны изменения в коде компилятора или если компилятор загружен через `Subversion`.

Имеется два пути перекомпиляции компилятора: вручную, или с помощью makefiles. Каждый из этих методов будет описан.

F.2. Перед стартом

Для лёгкой компиляции компилятора лучше соблюдать следующую структура каталогов *(базовый каталог **`/pp/src`** поддерживается, но может отличаться)*:

```
/pp/src/Makefile
       /makefile.fpc
       /rtl/linux
           /inc
           /i386
           /…
       /compiler
```

Если используется makefiles, то должно использоваться описанное выше дерево каталогов.

Исходные коды компилятора и rtl сжаты таким образом, что когда они распаковываются в тот же каталог *(**`/pp/src`** в примере выше)*, то в результате получается описанное выше дерево каталогов.

Есть два способа приступить к компиляции компилятора и RTL. Оба способа должны использоваться в зависимости от ситуации. Обычно RTL должна

компилироваться первой, перед компиляцией компилятора, после чего компилятор компилируется с использованием текущего компилятора. В некоторых особых случаях первым должен компилироваться компилятор, с заранее скомпилированной RTL.

Как решить, что следует компилировать в первую очередь? В общем случае ответ заключается в том, что сначала должна быть скомпилирована RTL. Из этого правила существует два исключения:

1. Первый случай – это когда некоторые внутренние подпрограммы в RTL изменились, или если новые подпрограммы добавились. Поскольку старый компилятор не знает об этих изменениях, он будет вызывать функции, которые основаны на старой RTL, и, следовательно, это не будет работать корректно. В результате компоновка не будет выполнена или бинарный файл приведёт к ошибке.
2. Второй случай – это когда что-то добавлено к RTL, о чём компилятору необходимо знать: например, новый механизм ассемблера по умолчанию.

Как узнать, что один из этих случаев произошёл? Нет других путей, кроме почтовой рассылки команды Free Pascal. Если компилятор не может быть перекомпилирован, когда компиляция RTL выполняется первой, то попробуйте другой способ.

F.3. Компиляция с использованием make

При компиляции с make необходимо иметь описанную выше структуру каталогов. Компиляция компилятора достигается с помощью цели `cycle`.

При нормальных обстоятельствах перекомпиляция компилятора сводится к следующим инструкциям *(предполагаем, что вы начинаете с каталога **/pp/src**)*:

```
cd compiler
make cycle
```

Это будет работать только в том случае, если makefile правильно настроен и если имеются необходимые инструменты в `PATH`. Какие инструменты должны быть установлены, можно найти в приложении E.1. Введение[180].

Приведённые выше инструкции будут делать следующее:

1. Используя текущий компилятор, RTL компилируется в правильный каталог, который определяется операционной системой, например, на LINUX RTL компилируется в каталог **`rtl/linux`**.
2. Компилятор компилируется, используя вновь откомпилированную RTL. Если эта операция прошла успешно, то вновь скомпилированный исполняемый файл компилятора копируется во временный исполняемый файл.
3. Используя временный исполняемый файл, созданный на предыдущем шаге, RTL перекомпилируется.

4. Используя временный исполняемый файл и вновь скомпилированную RTL, созданную на предыдущем шаге, компилятор снова компилируется.

Последние два действия повторяются по три раза, пока три прохода не будут сделаны или пока сгенерированный бинарный файл компилятора не будет эквивалентен бинарному файлу, с помощью которого он был сгенерирован. Этот процесс гарантирует, что бинарный файл компилятора является правильным.

Компиляция для другой целевой ОС: Если выполняется компиляция компилятора для другой целевой ОС, то необходимо определить переменную `OS_TARGET` в makefile. Она может быть установлена в следующие значения: win32, go32v2, os2 и linux. Например, выбор кросс-компиляции для целевой системы go32v2 из win32:

```
cd compiler
make cycle OS_TARGET=go32v2
```

это будет компилировать go32v2 RTL и go32v2 компилятор.

При компиляции нового компилятора компилятор должен быть скомпилирован с использованием существующей скомпилированной RTL, все цели должны использоваться, и другой каталог RTL, отличающийся от каталога по умолчанию (**`../rtl/$(OS_TARGET)`**) должен быть указан.

Например, если предположить что скомпилированные модули RTL находятся в **`/pp/rtl/units/i386-linux`**, набрав

```
cd compiler
make clean
make all UNITDIR=/pp/rtl/units/i386-linux
```

следует использовать RTL из каталога **`/pp/rtl/units/i386-linux`**.

Данный пример компилирует компилятор, используя модули RTL в **`/pp/rtl/units/i386-linux`**. После того, как это сделано, можно использовать '`make cycle`', запустив этот компилятор:

```
make cycle PP=./ppc386
```

это будет `make cycle` из примера выше, но запустит компилятор, который был сгенерирован с помощью инструкции `make all`.

В любом случае многие параметры могут быть переданы в make, чтобы повлиять на процесс компиляции. В общем, makefiles добавляют любые необходимые опции в командную строку, чтобы RTL и компилятор могли быть скомпилированы. Дополнительные опции *(например, опции оптимизации)* можно указать, передав их в `OPT`.

F.4. Компиляция вручную

Компиляция вручную – это трудно и нудно. Но если очень хочется. То можно

сделать. Компиляция RTL и компилятора будут рассмотрены отдельно.

F.4.1. Компиляция RTL

Для перекомпиляции RTL, так чтобы новый компилятор был собран, как минимум должны быть собраны следующие модули в указанном порядке:

loaders
Программа заглушки , которая является стартовым кодом для каждой программы на `Pascal`. Эти файлы имеют расширение .as, потому что они написаны на ассемблере. Они должны быть ассемблированы ассемблером GNU as. Эти заглушки находятся в каталоге, который зависит от ОС, за исключением LINUX, где они находятся в каталоге LINUX в подкаталоге, который зависит от процессора *(i386 или m68k)*.

System
Системный модуль. Этот модуль находится в зависимости от ОС в подкаталогах RTL.

Strings
Модуль strings. Этот модуль находится в RTL в подкаталоге inc.

dos
Модуль dos. Этот модуль находится в зависимости от ОС в подкаталоге RTL. Возможно, другие модули будут откомпилированы как вследствие попытки компиляции этого модуля *(например, на LINUX модуль linux будет скомпилирован, на go32, будет скомпилирован модуль go32)*.

objects
Модуль objects. Этот модуль находится в RTL в подкаталоге inc.

Для компиляции этих модулей на `i386` будут созданы следующие объявления:

```
ppc386 -Tlinux -b- -Fi../inc -Fi../i386 -FE. -di386 -Us -Sg
system.pp
ppc386 -Tlinux -b- -Fi../inc -Fi../i386 -FE. -di386 ../inc/
strings.pp
ppc386 -Tlinux -b- -Fi../inc -Fi../i386 -FE. -di386 dos.pp
ppc386 -Tlinux -b- -Fi../inc -Fi../i386 -FE. -di386 ../inc/ob-
jects.pp
```

Это минимальный набор опций командной строки, необходимых для компиляции RTL.

Для другого процессора i386 должно быть заменено на соответствующий процессор. Для другой целевой ОС настройка целевой системы (`-T`) должна быть сделана соответствующим образом.

В зависимости от целевой ОС есть и другие модули, которые могут быть

откомпилированы, но которые не являются строго необходимыми для перекомпиляции компилятора. Следующие модули доступны для всех платформ:

objpas
Необходим для режима Delphi. Требуется опция `-Mobjpas`. Находится в подкаталоге objpas.

sysutils
Многие полезные функции, как и в Delphi. Находится в каталоге objpas и нужно использовать опцию `-MObjpas` для компиляции.

typinfo
Функции для доступа к информации RTTI, подобно Delphi. Находится в каталоге objpas.

math
Математические функции как в Delphi. Находится в каталоге objpas.

mmx
Расширения MMX для класса процессоров Intel. Находится в каталоге i386.

getopts
GNU-совместимый модуль getopts. Находится в каталоге inc.

heaptrc
Для отладки кучи. Находится в каталоге inc.

F.4.2. Компиляция компилятора

Компиляцию компилятора можно выполнить одним объявлением. Это всегда лучше, чем сначала удалять все модули из каталога компилятора и выглядит примерно так:

```
rm *.ppu *.o
```

на LINUX, а для DOS:

```
del *.ppu
del *.o
```

После этого компилятор можно откомпилировать с помощью следующей командной стоки:

```
ppc386 -Tlinux -Fu../rtl/units/i386-linux -di386 -dGDB pp.pas
```

То есть, минимальным набором опций являются:

1. Целевая **ОС**. Может быть пропущена, если компиляция выполняется для той же целевой платформы, какая используется компилятором.
2. Путь к RTL. Может быть пропущен, если в системе имеется правильная конфигурации fpc.cfg. Если компилятор должен быть скомпилирован с RTL, которая была откомпилирована первой, это должен быть **`../rtl/os`**

*(замените **OS** на соответствующую директорию операционной системы в RTL)*.

3. Определение процессора, для которого компилируется компилятор. Обязательно.
4. `-dGDB`. Обязательно.
5. `-Sg` необходима, некоторые части компилятора используют операторы `goto` (если конкретно, то сканер).
6. Каталог с дополнительными модулями и включаемыми *(include)* файлами для компиляции.
7. Каталог с системными определениями.

Так что абсолютно минимальная командная строка выглядит так:

```
ppc386 -di386 -dGDB -Sg pp.pas
```

Некоторые другие опции командной строки могут быть использованы, но это минимум. Список распознаваемых опций можно найти в таблице F.1.

Таблица F.1. Возможные определения при компиляции FPC.

Определение	Что делает
GDB	Поддерживает отладчик GNU Debugger (требуется переключатель).
I386	Генерирует компилятор для семейства процессоров Intel i386+.
M68K	Генерирует компилятор для семейства процессоров M680x0.
X86_64	Генерирует компилятор для семейства процессоров AMD64.
POWERPC	Генерирует компилятор для семейства процессоров PowerPC.
POWERPC64	Генерирует компилятор для семейства **64**-разрядных процессоров PowerPC.
ARM	Генерирует компилятор для семейства процессоров Intel ARM.
SPARC	Генерирует компилятор для семейства процессоров SPARC.
EXTDEBUG	Выполняет некоторый дополнительный код отладки.
MEMDEBUG	Отображает некоторую полезную информацию о памяти.
SUPPORT_MMX	Только i386: Включает переключатель компилятора `MMX`, который позволяет компилятору генерировать инструкции MMX.

EXTERN_MSG	Не компилируйте msgfiles в компилятор, всегда используйте внешние messagefiles.
NOOPT	Не включать оптимизацию в компиляторе.
CMEM	Использовать менеджер памяти C.

Этот список может быть изменён, исходный файл pp.pas всегда содержит последнюю актуальную версию списка.

ПРИЛОЖЕНИЕ G: ОПРЕДЕЛЕНИЯ КОМПИЛЯТОРА ВО ВРЕМЯ КОМПИЛЯЦИИ

В этом приложении описываются возможные определения при компиляции программ, использующих Free Pascal. Приведены краткие описания определений, а затем также описано его использование.

Таблица G.1. Возможные определения при компиляции с использованием FPC.

Определение	Описание
FPC_LINK_DYNAMIC	Определено, если выход будет связан динамически. Определяется с помощью переключателя компилятора `-XD`.
FPC_LINK_STATIC	Определено, если выход будет скомпонован статически. Это значение по умолчанию.
FPC_LINK_SMART	Определено, если выход с «умной компоновкой». Определено, если использован переключатель компилятора `-XX`.
FPC_PROFILE	Определено, если профилирующий код добавлен в программу. Определено, если использован переключатель компилятора `-pg`.
FPC_CROSSCOMPILING	Определено, если целевые ОС/процессор отличаются от исходных ОС/процессора.
FPC	Всегда определено для Free Pascal.
VER2	Всегда определено для Free Pascal версии **2.x.x**.
VER2_0	Всегда определено для Free Pascal версии **2.0.x**.
VER2_2	Всегда определено для Free Pascal версии **2.2.x**.
FPC_VERSION	Содержит сташий номер версии FPC.
FPC_RELEASE	Содержит младший номер версии FPC.
FPC_PATCH	Содержит третью часть номера версии FPC.
FPC_FULLVERSION	Содержит весь номер версии FPC *(как единый номер)*, который может

	использоваться для сравнения. Для FPC **2.2.4** он будет **20204**.
ENDIAN_LITTLE	Определено, если целевой процессор Free Pascal имеет прямой порядок байтов *(80x86, Alpha, ARM)*.
ENDIAN_BIG	Определено, если целевой процессор Free Pascal имеет обратный порядок байтов *(680x0, PowerPC, SPARC, MIPS)*.
FPC_DELPHI	Free Pascal в режиме Delphi, либо используется переключатель компилятора `-MDelphi` или директива `$MODE DELPHI`.
FPC_OBJFPC	Free Pascal в режиме `OBJFPC`, либо используется переключатель компилятора `-Mobjfpc` или директива `$MODE OBJFPC`.
FPC_TP	Free Pascal в режиме Turbo Pascal, либо используется переключатель компилятора `-Mtp` или директива `$MODE TP`.
FPC_GPC	Free Pascal в режиме GNU Pascal, либо используется переключатель компилятора `-SP` или директива `$MODE GPC`.

ПРИМЕЧАНИЕ

Определения `ENDIAN_LITTLE` и `ENDIAN_BIG` добавлены, начиная с версии Free Pascal **1.0.5**.

Таблица G.2. Возможные определения процессора (CPU) при компиляции с использованием FPC.

Определение	Когда определено
CPU86	Free Pascal целевой процессор Intel 80x86 или совместимый.
CPU87	Free Pascal целевой процессор Intel 80x86 или совместимый.
CPU386	Free Pascal целевой процессор Intel 80386 или старше.
CPUI386	Free Pascal целевой процессор Intel 80386 или старше.
CPU68K	Free Pascal целевой процессор Motorola 680x0 или совместимый.

CPUM68K	Free Pascal целевой процессор Motorola 680x0 или совместимый.
CPUM68020	Free Pascal целевой процессор Motorola 68020 или старше.
CPU68	Free Pascal целевой процессор Motorola 680x0 или совместимый.
CPUSPARC32	Free Pascal целевой процессор SPARC v7 или совместимый.
CPUSPARC	Free Pascal целевой процессор SPARC v7 или совместимый.
CPUALPHA	Free Pascal целевой процессор Alpha AXP или совместимый.
CPUPOWERPC	Free Pascal целевой процессор 32-битный или 64-битный PowerPC или совместимый.
CPUPOWERPC32	Free Pascal целевой процессор 32-битный PowerPC или совместимый.
CPUPOWERPC64	Free Pascal целевой процессор 64-битный PowerPC или совместимый.
CPUX86_64	Free Pascal целевой процессор AMD64 или Intel 64-битный.
CPUAMD64	Free Pascal целевой процессор AMD64 или Intel 64-битный.
CPUX64	Free Pascal целевой процессор AMD64 или Intel 64-битный.
CPUIA64	Free Pascal целевой процессор Intel itanium 64-битный.
CPUARM	Free Pascal целевой процессор ARM **32**-битный.
CPUAVR	Free Pascal целевой процессор AVR **16**-битный.
CPU16	Free Pascal целевой процессор **16**-битный.
CPU32	Free Pascal целевой процессор **32**-битный.
CPU64	Free Pascal целевой процессор **64**-битный.

CPU8086	указывает на **16**-битную платформу на процессоре **x86** (**i8086**)
CPUI8086	указывает на **16**-битную платформу на процессоре **x86** (**i8086**)

Таблица G.3. Возможные определения плавающей точки (FPU) при компиляции с использованием FPC.

Определение	Когда определено
FPUSOFT	Программная эмуляция FPU (все типы).
FPUSSE64	SSE64 FPU на Intel I386 и выше, AMD64.
FPUSSE	SSE инструкции на Intel I386 и выше.
FPUSSE2	SSE 2 инструкции на Intel I386 и выше.
FPUSSE3	SSE 3 инструкции на Intel I386 и выше, AMD64.
FPULIBGCC	GCC библиотека эмуляции FPU на ARM и M68K.
FPU68881	68881 на M68K.
FPUFPA	FPA на ARM.
FPUFPA10	FPA 10 на ARM.
FPUFPA11	FPA 11 на ARM.
FPUVFP	VFP на ARM.
FPUX87	X87 FPU на Intel I386 и выше.
FPUITANIUM	На Intel Itanium.
FPUSTANDARD	На PowerPC (**32**/**64** bit).
FPUHARD	На Sparc.

Таблица G.4. Возможные определения компиляции с целевой ОС.

Целевая ОС	Определение
linux	LINUX, UNIX
freebsd	FREEBSD, BSD, UNIX
netbsd	NETBSD, BSD, UNIX
sunos	SUNOS, SOLARIS, UNIX
go32v2	GO32V2, DPMI
реальный режим 16 бит	*MSDOS (2.7.1 и выше)*

MS-DOS	
os2	OS2
emx	OS2, EMX
Windows (all)	WINDOWS
Windows 32-bit	WIN32, MSWINDOWS
Windows 64-bit	WIN64, MSWINDOWS
Windows (winCE)	WINCE, UNDER_CE, UNICODE
Classic Amiga	AMIGA
Atari TOS	ATARI
Classic Macintosh	MACOS
PalmOS	PALMOS
BeOS	BEOS, UNIX
QNX RTP	QNX, UNIX
Mac OS X	BSD, DARWIN, UNIX

В **16** битной модели памяти MS-DOS, для компиляции программы, используется одно из `FPC_MM_TINY`, `FPC_MM_SMALL`,`FPC_MM_MEDIUM`, `FPC_MM_COMPACT`,`FPC_MM_LARGE`, `FPC_MM_HUGE`.

ПРИМЕЧАНИЕ

Определение UNIX было добавлено, начиная с версии Free Pascal **1.0.5**. Операционные системы BSD больше не определяют LINUX, начиная с версии **1.0.7**.

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

- Z -