Free Pascal
Programmer's Guide

Programmer's Guide for Free Pascal, Version 2.6.4
Document version 2.6
February 2016

Michaël Van Canneyt

Contents

1 Compiler directives **13**

1.1 Introduction . . . 13

1.2 Local directives . . . 14

1.2.1 $A or $ALIGN : Align Data . . . 14

1.2.2 $A1, $A2,$A4 and $A8 . . . 14

1.2.3 $ASMMODE : Assembler mode (Intel 80x86 only) . . . 15

1.2.4 $B or $BOOLEVAL : Complete boolean evaluation . . . 15

1.2.5 $C or $ASSERTIONS : Assertion support . . . 15

1.2.6 $BITPACKING : Enable bitsize packing . . . 16

1.2.7 $CALLING : Specify calling convention . . . 16

1.2.8 $CHECKPOINTER : Check pointer values . . . 16

1.2.9 $CODEALIGN : Set the code alignment . . . 17

1.2.10 $COPERATORS : Allow C like operators . . . 18

1.2.11 $DEFINE or $DEFINEC : Define a symbol . . . 18

1.2.12 $ELSE : Switch conditional compilation . . . 18

1.2.13 $ELSEC : Switch conditional compilation . . . 19

1.2.14 $ELSEIF or $ELIFC : Switch conditional compilation . . . 19

1.2.15 $ENDC : End conditional compilation . . . 19

1.2.16 $ENDIF : End conditional compilation . . . 19

1.2.17 $ERROR or $ERRORC : Generate error message . . . 20

1.2.18 $ENDREGION: End of a collapsible region . . . 20

1.2.19 $EXTENDEDSYM: Ignored . . . 20

1.2.20 $EXTERNALSYM: Ignored . . . 20

1.2.21 $F : Far or near functions . . . 20

1.2.22 $FATAL : Generate fatal error message . . . 21

1.2.23 $FPUTYPE : Select coprocessor type . . . 21

1.2.24 $GOTO : Support Goto and Label . . . 22

1.2.25 $H or $LONGSTRINGS : Use AnsiStrings . . . 22

1.2.26 $HINT : Generate hint message . . . 22

1.2.27 $HINTS : Emit hints . . . 22

1.2.28 $HPPEMIT: Ignored . . . 22
1.2.29 $IF : Start conditional compilation . . . 23
1.2.30 $IFC : Start conditional compilation . . . 23
1.2.31 $IFDEF Name : Start conditional compilation . . . 23
1.2.32 $IFNDEF : Start conditional compilation . . . 23
1.2.33 $IFOPT : Start conditional compilation . . . 23
1.2.34 $IMPLICITEXCEPTIONS : Implicit finalization code generation . . . 23
1.2.35 $INFO : Generate info message . . . 24
1.2.36 $INLINE : Allow inline code. . . . 24
1.2.37 $INTERFACES : Specify Interface type. . . . 24
1.2.38 $I or $IOCHECKS : Input/Output checking . . . 24
1.2.39 $IEEEERRORS : Enable IEEE error checking for constants . . . 25
1.2.40 $I or $INCLUDE : Include file . . . 25
1.2.41 $I or $INCLUDE : Include compiler info . . . 26
1.2.42 $J or $WRITEABLECONST : Allow assignments to typed consts . . . 27
1.2.43 $L or $LINK : Link object file . . . 27
1.2.44 $LIBEXPORT : Ignored . . . 28
1.2.45 $LINKFRAMEWORK : Link to a framework . . . 28
1.2.46 $LINKLIB : Link to a library . . . 28
1.2.47 $M or $TYPEINFO : Generate type info . . . 29
1.2.48 $MACRO : Allow use of macros. . . . 29
1.2.49 $MAXFPUREGISTERS : Maximum number of FPU registers for variables . . . 29
1.2.50 $MESSAGE : Generate info message . . . 30
1.2.51 $MINENUMSIZE : Specify minimum enumeration size . . . 30
1.2.52 $MINFPCONSTPREC : Specify floating point constant precision . . . 30
1.2.53 $MMX : Intel MMX support (Intel 80x86 only) . . . 30
1.2.54 $NODEFINE : Ignored . . . 31
1.2.55 $NOTE : Generate note message . . . 31
1.2.56 $NOTES : Emit notes . . . 31
1.2.57 $OBJECTCHECKS : Check Object . . . 31
1.2.58 $OPTIMIZATION : Enable Optimizations . . . 32
1.2.59 $PACKENUM or $Z : Minimum enumeration type size . . . 33
1.2.60 $PACKRECORDS : Alignment of record elements . . . 33
1.2.61 $PACKSET : Specify set size . . . 34
1.2.62 $POP : Restore compiler settings . . . 34
1.2.63 $PUSH : Save compiler settings . . . 34
1.2.64 $Q or $OV or $OVERFLOWCHECKS: Overflow checking . . . 34
1.2.65 $R or $RANGECHECKS : Range checking . . . 35
1.2.66 $REGION : Mark start of collapsible region . . . 35
1.2.67 $R or $RESOURCE : Include resource . . . 35

1.2.68 $SATURATION : Saturation operations (Intel 80x86 only) . . . 35
1.2.69 $SAFEFPUEXCEPTIONS Wait when storing FPU values on Intel x86 . . . 36
1.2.70 $SCOPEDENUMS Control use of scoped enumeration types . . . 36
1.2.71 $SETC : Define and assign a value to a symbol . . . 36
1.2.72 $STATIC : Allow use of Static keyword. . . . 37
1.2.73 $STOP : Generate fatal error message . . . 37
1.2.74 $STRINGCHECKS : Ignored . . . 37
1.2.75 $T or $TYPEDADDRESS : Typed address operator (@) . . . 37
1.2.76 $UNDEF or $UNDEFC : Undefine a symbol . . . 38
1.2.77 $V or $VARSTRINGCHECKS : Var-string checking . . . 38
1.2.78 $W or $STACKFRAMES : Generate stackframes . . . 39
1.2.79 $WAIT : Wait for enter key press . . . 39
1.2.80 $WARN : Control emission of warnings . . . 39
1.2.81 $WARNING : Generate warning message . . . 40
1.2.82 $WARNINGS : Emit warnings . . . 40
1.2.83 $Z1, $Z2 and $Z4 . . . 40
1.3 Global directives . . . 41
1.3.1 $APPID : Specify application ID. . . . 41
1.3.2 $APPNAME : Specify application name. . . . 41
1.3.3 $APPTYPE : Specify type of application. . . . 41
1.3.4 $CALLING : Default calling convention . . . 42
1.3.5 $CODEPAGE : Set the source codepage . . . 42
1.3.6 $COPYRIGHT specify copyright info . . . 43
1.3.7 $D or $DEBUGINFO : Debugging symbols . . . 43
1.3.8 $DESCRIPTION : Application description . . . 43
1.3.9 $E : Emulation of coprocessor . . . 43
Intel 80x86 version . . . 43
Motorola 680x0 version . . . 43
1.3.10 $EXTENSION : Extension of generated binary. . . . 43
1.3.11 $FRAMEWORKPATH : Specify framework path. . . . 44
1.3.12 $G : Generate 80286 code . . . 44
1.3.13 $IMAGEBASE : Specify DLL image base location. . . . 44
1.3.14 $INCLUDEPATH : Specify include path. . . . 44
1.3.15 $L or $LOCALSYMBOLS : Local symbol information . . . 45
1.3.16 $LIBPREFIX : Specify library filename prefix. . . . 45
1.3.17 $LIBRARYPATH : Specify library path. . . . 45
1.3.18 $LIBSUFFIX : Set the library suffix . . . 45
1.3.19 $MAXSTACKSIZE : Set maximum stack size . . . 46
1.3.20 $M or $MEMORY : Memory sizes . . . 46
1.3.21 $MINSTACKSIZE : Set minimum stack size . . . 46

1.3.22 $MODE : Set compiler compatibility mode . . . 46
1.3.23 $MODESWITCH : Select mode features . . . 47
1.3.24 $N : Numeric processing . . . 48
1.3.25 $O : Level 2 Optimizations . . . 48
1.3.26 $OBJECTPATH : Specify object path. . . . 48
1.3.27 $P or $OPENSTRINGS : Use open strings . . . 49
1.3.28 $PASCALMAINNAME : Set entry point name . . . 49
1.3.29 $PIC : Generate PIC code or not . . . 49
1.3.30 $POINTERMATH : Allow use of pointer math . . . 49
1.3.31 $PROFILE : Profiling . . . 49
1.3.32 $S : Stack checking . . . 49
1.3.33 $SCREENNAME : Specify screen name . . . 50
1.3.34 $SETPEFLAGS : Specify PE Executable flags . . . 50
1.3.35 $SMARTLINK : Use smartlinking . . . 50
1.3.36 $SYSCALLS : Select system calling convention on Amiga/MorphOS . . . 51
1.3.37 $THREADNAME : Set thread name in Netware . . . 51
1.3.38 $UNITPATH : Specify unit path. . . . 51
1.3.39 $VARPROPSETTER : Enable use of var/out/const parameters for property setters. . . . 51
1.3.40 $VERSION : Specify DLL version. . . . 52
1.3.41 $WEAKPACKAGEUNIT : ignored . . . 52
1.3.42 $X or $EXTENDEDSYNTAX : Extended syntax . . . 52
1.3.43 $Y or $REFERENCEINFO : Insert Browser information . . . 53

2 Using conditionals, messages and macros 54
2.1 Conditionals . . . 54
2.1.1 Predefined symbols . . . 55
2.2 Macros . . . 55
2.3 Compile time variables . . . 57
2.4 Compile time expressions . . . 57
2.4.1 Definition . . . 57
2.4.2 Usage . . . 59
2.5 Messages . . . 62

3 Using Assembly language 64
3.1 Using assembler in the sources . . . 64
3.2 Intel 80x86 Inline assembler . . . 65
3.2.1 Intel syntax . . . 65
3.2.2 AT&T Syntax . . . 68
3.3 Motorola 680x0 Inline assembler . . . 69
3.4 Signaling changed registers . . . 70

4 Generated code **71**
4.1 Units . . . 71
4.2 Programs . . . 72

5 Intel MMX support **73**
5.1 What is it about? . . . 73
5.2 Saturation support . . . 74
5.3 Restrictions of MMX support . . . 74
5.4 Supported MMX operations . . . 75
5.5 Optimizing MMX support . . . 75

6 Code issues **76**
6.1 Register Conventions . . . 76
6.1.1 accumulator register . . . 76
6.1.2 accumulator 64-bit register . . . 76
6.1.3 float result register . . . 76
6.1.4 self register . . . 76
6.1.5 frame pointer register . . . 76
6.1.6 stack pointer register . . . 77
6.1.7 scratch registers . . . 77
6.1.8 Processor mapping of registers . . . 77
Intel 80x86 version . . . 77
Motorola 680x0 version . . . 77
6.2 Name mangling . . . 78
6.2.1 Mangled names for data blocks . . . 78
6.2.2 Mangled names for code blocks . . . 79
6.2.3 Modifying the mangled names . . . 81
6.3 Calling mechanism . . . 81
6.4 Nested procedure and functions . . . 82
6.5 Constructor and Destructor calls . . . 82
6.5.1 objects . . . 82
6.5.2 classes . . . 83
6.6 Entry and exit code . . . 83
6.6.1 Intel 80x86 standard routine prologue / epilogue . . . 83
6.6.2 Motorola 680x0 standard routine prologue / epilogue . . . 84
6.7 Parameter passing . . . 84
6.7.1 Parameter alignment . . . 84
6.8 Stack limitations . . . 85

7 Linking issues **86**
7.1 Using external code and variables . . . 86

7.1.1 Declaring external functions or procedures . . . 86
7.1.2 Declaring external variables . . . 87
7.1.3 Declaring the calling convention modifier . . . 89
7.1.4 Declaring the external object code . . . 89
Linking to an object file . . . 89
Linking to a library . . . 90
7.2 Making libraries . . . 92
7.2.1 Exporting functions . . . 92
7.2.2 Exporting variables . . . 92
7.2.3 Compiling libraries . . . 93
7.2.4 Unit searching strategy . . . 93
7.3 Using smart linking . . . 93

8 Memory issues **95**
8.1 The memory model. . . . 95
8.2 Data formats . . . 95
8.2.1 Integer types . . . 95
8.2.2 Char types . . . 96
8.2.3 Boolean types . . . 96
8.2.4 Enumeration types . . . 96
8.2.5 Floating point types . . . 96
Single . . . 96
Double . . . 97
Extended . . . 97
Comp . . . 98
Real . . . 98
8.2.6 Pointer types . . . 98
8.2.7 String types . . . 98
Ansistring types . . . 98
Shortstring types . . . 98
Widestring types . . . 99
8.2.8 Set types . . . 99
8.2.9 Static array types . . . 99
8.2.10 Dynamic array types . . . 99
8.2.11 Record types . . . 99
8.2.12 Object types . . . 99
8.2.13 Class types . . . 100
8.2.14 File types . . . 101
8.2.15 Procedural types . . . 102
8.3 Data alignment . . . 102

8.3.1 Typed constants and variable alignment . . . 102
8.3.2 Structured types alignment . . . 103
8.4 The heap . . . 103
8.4.1 Heap allocation strategy . . . 103
8.4.2 The heap grows . . . 104
8.4.3 Debugging the heap . . . 104
8.4.4 Writing your own memory manager . . . 105
8.5 Using DOS memory under the Go32 extender . . . 109
8.6 When porting Turbo Pascal code . . . 110
8.7 Memavail and Maxavail . . . 110

9 Resource strings **112**
9.1 Introduction . . . 112
9.2 The resource string file . . . 112
9.3 Updating the string tables . . . 114
9.4 GNU gettext . . . 115
9.5 Caveat . . . 116

10 Thread programming **117**
10.1 Introduction . . . 117
10.2 Programming threads . . . 117
10.3 Critical sections . . . 120
10.4 The Thread Manager . . . 121

11 Optimizations **123**
11.1 Non processor specific . . . 123
11.1.1 Constant folding . . . 123
11.1.2 Constant merging . . . 123
11.1.3 Short cut evaluation . . . 123
11.1.4 Constant set inlining . . . 123
11.1.5 Small sets . . . 124
11.1.6 Range checking . . . 124
11.1.7 And instead of modulo . . . 124
11.1.8 Shifts instead of multiply or divide . . . 124
11.1.9 Automatic alignment . . . 124
11.1.10 Smart linking . . . 124
11.1.11 Inline routines . . . 124
11.1.12 Stack frame omission . . . 124
11.1.13 Register variables . . . 125
11.2 Processor specific . . . 125
11.2.1 Intel 80x86 specific . . . 125

11.2.2 Motorola 680x0 specific . . . 127
11.3 Optimization switches . . . 127
11.4 Tips to get faster code . . . 128
11.5 Tips to get smaller code . . . 128
11.6 Whole Program Optimization . . . 129
11.6.1 Overview . . . 129
11.7 General principles . . . 129
11.7.1 How to use . . . 129
Step 1: Generate WPO feedback file . . . 129
Step 2: Use the generated WPO feedback file . . . 130
11.7.2 Available WPO optimizations . . . 130
11.7.3 format of the WPO file . . . 131

12 Programming shared libraries **132**
12.1 Introduction . . . 132
12.2 Creating a library . . . 132
12.3 Using a library in a pascal program . . . 133
12.4 Using a pascal library from a C program . . . 135
12.5 Some Windows issues . . . 136

13 Using Windows resources **137**
13.1 The resource directive `$R` . . . 137
13.2 Creating resources . . . 137
13.3 Using string tables. . . . 138
13.4 Inserting version information . . . 139
13.5 Inserting an application icon . . . 139
13.6 Using a Pascal preprocessor . . . 140

A Anatomy of a unit file **141**
A.1 Basics . . . 141
A.2 reading ppufiles . . . 141
A.3 The Header . . . 142
A.4 The sections . . . 143
A.5 Creating ppufiles . . . 144

B Compiler and RTL source tree structure **147**
B.1 The compiler source tree . . . 147
B.2 The RTL source tree . . . 147

C Compiler limits **149**

D Compiler modes **150**

D.1 FPC mode . . . 150
D.2 TP mode . . . 150
D.3 Delphi mode . . . 151
D.4 OBJFPC mode . . . 151
D.5 MACPAS mode . . . 152

E Using fpcmake **153**
E.1 Introduction . . . 153
E.2 Functionality . . . 153
E.3 Usage . . . 154
E.4 Format of the configuration file . . . 155
E.4.1 clean . . . 155
E.4.2 compiler . . . 155
E.4.3 Default . . . 156
E.4.4 Dist . . . 156
E.4.5 Install . . . 157
E.4.6 Package . . . 157
E.4.7 Prerules . . . 157
E.4.8 Requires . . . 157
E.4.9 Rules . . . 158
E.4.10 Target . . . 158
E.5 Programs needed to use the generated makefile . . . 159
E.6 Variables that affect the generated makefile . . . 159
E.6.1 Directory variables . . . 160
E.6.2 Compiler command line variables . . . 160
E.7 Variables set by fpcmake . . . 160
E.7.1 Directory variables . . . 161
E.7.2 Target variables . . . 162
E.7.3 Compiler command line variables . . . 163
E.7.4 Program names . . . 163
E.7.5 File extensions . . . 164
E.7.6 Target files . . . 164
E.8 Rules and targets created by fpcmake . . . 164
E.8.1 Pattern rules . . . 164
E.8.2 Build rules . . . 165
E.8.3 Cleaning rules . . . 165
E.8.4 Archiving rules . . . 165
E.8.5 Installation rules . . . 165
E.8.6 Informative rules . . . 166

F Compiling the compiler **167**

F.1 Introduction 167
F.2 Before starting 167
F.3 Compiling using make 168
F.4 Compiling by hand 169
F.4.1 Compiling the RTL 169
F.4.2 Compiling the compiler 170

G Compiler defines during compilation **172**

List of Tables

2.1 Predefined macros 56

6.1 Intel 80x86 Register table 77
6.2 Motorola 680x0 Register table 77
6.3 Calling mechanisms in Free Pascal 81
6.4 Stack frame when calling a nested procedure (32-bit processors) 82
6.5 Stack frame when calling a procedure (32-bit model) 84
6.6 Maximum limits for processors 85

8.1 Enumeration storage for `tp` mode 96
8.2 Processor mapping of real type 98
8.3 AnsiString memory structure (32-bit model) 98
8.4 Object memory layout (32/64-bit model) 100
8.5 Object Virtual Method Table memory layout (32/64-bit model) 100
8.6 Class memory layout (32/64-bit model) 100
8.7 Class Virtual Method Table memory layout (32/64-bit model) 101
8.8 Data alignment 103
8.9 ReturnNilIfGrowHeapFails value 104

12.1 Shared library support 132

A.1 PPU Header 142
A.2 PPU CPU Field values 142
A.3 PPU Header Flag values 143
A.4 chunk data format 143
A.5 Possible PPU Entry types 144

F.1 Possible defines when compiling FPC 171

G.1 Possible defines when compiling using FPC 173
G.2 Possible CPU defines when compiling using FPC 174
G.3 Possible FPU defines when compiling using FPC 174
G.4 Possible defines when compiling using target OS 175

About this document

This is the programmer's manual for Free Pascal.

It describes some of the peculiarities of the Free Pascal compiler, and provides a glimpse of how the compiler generates its code, and how you can change the generated code. It will not, however, provide a detailed account of the inner workings of the compiler, nor will it describe how to use the compiler (described in the User's Guide). It also will not describe the inner workings of the Run-Time Library (RTL). The best way to learn about the way the RTL is implemented is from the sources themselves.

The things described here are useful when things need to be done that require greater flexibility than the standard Pascal language constructs (described in the Reference Guide).

Since the compiler is continuously under development, this document may get out of date. Wherever possible, the information in this manual will be updated. If you find something which isn't correct, or you think something is missing, feel free to contact me[1].

[1] at michael@freepascal.org

Chapter 1

Compiler directives

1.1 Introduction

Free Pascal supports compiler directives in the source file: Basically the same directives as in Turbo Pascal, Delphi and Mac OS pascal compilers. Some are recognized for compatibility only, and have no effect.

A compiler directive is always specified as a comment using curly braces:

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

The older `(* *)` notation, or the C++ notation `//` cannot be used for directives.

The fact that a directive is specified as a comment means that it is not a pascal instruction. That in turn means that if a non-existing directive is specified, the compiler does not give an error, it just gives a warning about an invalid/unknown directive.

There is a distinction between local and global directives:

- Local directives take effect from the moment they are encountered till they are changed by another directive or the same directive with a different argument: they can be specified more than once in a source file.
- Global directives have an effect on all of the compiled code. They can, in general, only be specified once per source file. It also means that their effect ends when the current unit is compiled; the effect does not propagate to another unit.

Some directives can only take a boolean value, a '+' to switch them on, or a '-' to switch them off. These directives are also known as switches. Many switches have a long form also. If they do, then the name of the long form is given also.

For long switches, the + or - character to switch the option on or off, may be replaced by the `ON` or `OFF` keywords.

Thus `{$I+}` is equivalent to `{$IOCHECKS ON}` or `{$IOCHECKS +}` and `{$C-}` is equivalent to `{$ASSERTIONS OFF}` or `{$ASSERTIONS -}`

The long forms of the switches are the same as their Delphi counterparts.

1.2 Local directives

Local directives can occur more than once in a unit or program, If they have a command line counterpart, the command line argument is restored as the default for each compiled file. The local directives influence the compiler's behaviour from the moment they're encountered until the moment another switch annihilates their behaviour, or the end of the current unit or program is reached.

1.2.1 $A or $ALIGN : Align Data

The {$ALIGN directive can be used to select the data alignment strategy of the compiler for records. It takes a numerical argument which can be 1, 2, 4, 8, 16 or 32, specifying the alignment boundary in bytes. For these values, it has the same effect as the {$PACKRECORDS} directive (see section 1.2.60, page 33).

Thus, the following

```
{$A 8}
```

is equivalent to

```
{$PACKRECORDS 8}
```

and specifies to the compiler that all data inside a record should be aligned on 8 byte boundaries.

In MACPAS mode, additionally it can have the following values:

MAC68K Specifies alignment following the m68K ABI.

POWER Specifies alignment following the PowerPC ABI.

POWERPC Specifies alignment following the PowerPC ABI.

RESET Resets the default alignment.

ON Same as specifying 4.

OFF Same as specifying 1.

These values are not available in the {$PACKRECORDS} directive.

1.2.2 $A1, $A2,$A4 and $A8

These directives are the same as the $PACKRECORDS directive (see section 1.2.60, page 33), but they have the alignment specifier embedded in the directive. Thus the following:

```
{$A8 }
```

is equivalent to

```
{$PACKRECORDS 8}
```

Note that the special cases of $PACKRECORDS cannot be expressed this way.

1.2.3 $ASMMODE : Assembler mode (Intel 80x86 only)

The {$ASMMODE XXX} directive informs the compiler what kind of assembler it can expect in an asm block. The XXX should be replaced by one of the following:

att Indicates that asm blocks contain AT&T syntax assembler.

intel Indicates that asm blocks contain Intel syntax assembler.

direct Tells the compiler that asm blocks should be copied directly to the assembler file. It is not possible to use such assembler blocks when the internal assembler of the compiler is used.

These switches are local, and retain their value to the end of the unit that is compiled, unless they are replaced by another directive of the same type. The command line switch that corresponds to this switch is -R.

The default assembler reader is the AT&T reader.

1.2.4 $B or $BOOLEVAL : Complete boolean evaluation

By default, the compiler uses shortcut boolean evaluation, i.e., the evaluation of a boolean expression is stopped once the result of the total exression is known with certainty. The {$B } switch can be used to change this behaviour: if its argument is ON, then the compiler will always evaluate all terms in the expression. If it is OFF (the default) then the compiler will only evaluate as many terms as are necessary to determine the result of the complete expression.

So, in the following example, the function Bofu, which has a boolean result, will never get called.

```
If False and Bofu then
  ...
```

A consequence of this is that any additional actions that are done by Bofu are not executed. If compiled with {$B ON}, then BoFu will be called anyway.

1.2.5 $C or $ASSERTIONS : Assertion support

The {$ASSERTIONS} switch determines if assert statements are compiled into the binary or not. If the switch is on, the statement

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

Will be compiled in the binary. If the BooleanExpression evaluates to False, the RTL will check if the AssertErrorProc is set. If it is set, it will be called with as parameters the AssertMessage message, the name of the file, the LineNumber and the address. If it is not set, a runtime error 227 is generated.

The AssertErrorProc is defined as

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

This can be used mainly for debugging purposes. The system unit sets the AssertErrorProc to a handler that displays a message on stderr and simply exits with a run-time error 227. The sysutils unit catches the run-time error 227 and raises an EAssertionFailed exception.

1.2.6 $BITPACKING : Enable bitsize packing

The $BITPACKING directive tells the compiler whether it should use bitpacking or not when it encounters the packed keyword for a structured type. The possible values are ON and OFF. If ON, then the compiler will bitpack structures when it encounters the Packed keyword.

In the following example, the TMyRecord record will be bitpacked:

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

Note that:

- The $BITPACKING directive is ignored in macpas mode, where packed records are always bitpacked.
- The bitpacked keyword can always be used to force bitwise packing, regardless of the value of the $BITPACKING directive, and regardless of the mode.

1.2.7 $CALLING : Specify calling convention

The {$CALLING } directive tells the compiler which calling convention should be used if none is specified:

```
{$CALLING REGISTER}
```

By default it is REGISTER. The following calling conventions exist:

default

register

cdecl

pascal

safecall

stdcall

oldfpccall

For a more detailed explanation of calling conventions, see section 6.3, page 81. As a special case, DEFAULT can be used, to restore the default calling convention.

1.2.8 $CHECKPOINTER : Check pointer values

The {$CHECKPOINTER} directive turns heap pointer checking on (value ON) or off (value OFF). If heap pointer checking is on and the code is compiled with the -gh (heaptrace) option on, then a check is inserted when dereferencing a pointer. The check will verify that the pointer contains a valid value, i.e. points to a location that is reachable by the program: the stack or a location in the heap. If not, a run-time error 216 or 204 is raised.

If the code is compiled without -gh switch, then this directive has no effect. Note that this considerably slows down the code.

1.2.9 $CODEALIGN : Set the code alignment

This switch sets the code alignment. It takes an argument which is the alignment in bytes.

```
{$CODEALIGN 8}
```

There are some more arguments which can be specified, to tune the behaviour even more. The general form is

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

Where `PARAM` is the parameter to tune, and `VAR` value is a numerical value specifying an alignment. `PARAM` can be one of the following strings:

PROC Set the alignment for procedure entry points.

JUMP Set the alignment for jump destination locations.

LOOP Set alignment for loops (for, while, repeat).

CONSTMIN Minimum alignment for constants (both typed and untyped).

CONSTMAX Maximum alignment for constants (both typed and untyped).

VARMIN Minimum alignment for static and global variables.

VARMAX Maximum alignment for static and global variables.

LOCALMIN Minimum alignment for local variables.

LOCALMAX Maximum alignment for local variables.

RECORDMIN Minimum alignment for record fields.

RECORDMAX Maximum alignment for record fields.

By default the size of a data structure determines the alignment:

- A `SmallInt` will be aligned at 2 bytes.
- A `LongInt` will be aligned at 4 bytes.
- A `Int64` will be aligned at 8 bytes.

With the above switches the minimum required alignment and a maximum used alignment can be specified. The maximum allowed alignment is only meaningful if it is smaller than the natural size. i.e. setting the maximum alignment (e.g. VARMAX) to 4, the alignment is forced to be at most 4 bytes: The `Int64` will then also be aligned at 4 bytes. The `SmallInt` will still be aligned at 2 bytes.

These values can also be specified on the command line as

```
-OaPARAM=VALUE
```

1.2.10 $COPERATORS : Allow C like operators

This boolean switch determines whether C like assignment operators are allowed. By default, these assignments are not allowed. After the following statement:

```
{$COPERATORS ON}
```

The following operators are allowed:

```
Var
  I : Integer;

begin
  I:=1;
  I+=3; // Add 3 to I and assign the result to I;
  I-=2; // Substract 2 from I and assign the result to I;
  I*=2; // Multiply I with 2 and assign the result to I;
  I/=2; // Divide I with 2 and assign the result to I;
end;
```

1.2.11 $DEFINE or $DEFINEC : Define a symbol

The directive

```
{$DEFINE name}
```

defines the symbol `name`. This symbol remains defined until the end of the current module (i.e. unit or program), or until a `$UNDEF name` directive is encountered.

If `name` is already defined, this has no effect. `Name` is case insensitive.

The symbols that are defined in a unit, are not saved in the unit file, so they are also not exported from a unit.

Under Mac Pascal mode, the `$DEFINEC` directive is equivalent to the `$DEFINE` directive and is provided for Mac Pascal compatibility.

`{$DEFINE}` can also be used to define macros or compile-time constants:

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

will redefine the standard modifier `cdecl` as `stdcall`.

More information about macros and compile-time constants, can be found in section 2.2, page 55

The `{$DEFINE}` directive has a command-line equivalent, `-d`

```
-dNAME
```

will define the symbol `Name`. Using

```
-dcdecl:=stdcall
```

one can redefine the standard modifier `cdecl` as `stdcall` from the command-line.

1.2.12 $ELSE : Switch conditional compilation

The `{$ELSE}` switches between compiling and ignoring the source text delimited by the preceding `{$IFxxx}` and following `{$ENDIF}`. Any text after the `ELSE` keyword but before the brace is ignored:

```
{$ELSE some ignored text}
```

is the same as

```
{$ELSE}
```

This is useful for indication what switch is meant.

1.2.13 $ELSEC : Switch conditional compilation

In MACPAS mode, this directive can be used as an alternative to the `$ELSE` directive. It is supported for compatibility with existing Mac OS pascal compilers.

1.2.14 $ELSEIF or $ELIFC : Switch conditional compilation

This directive can be used as a shortcut for a new `{$IF }` directive inside an `{$ELSE }` clause:

```
{$IF XXX}
  // XXX Code here
{$ELSEIF YYY}
  // YYY code here
{$ELSE}
  // And default code here
{$ENDIF}
```

is equivalent to

```
{$IF XXX}
  // XXX Code here
{$ELSE }
{$IF YYY}
  // YYY code here
{$ELSE}
  // And default code here
{$ENDIF}
{$ENDIF}
```

The directive is followed by an expression like the ones recognized by the `{$IF }` directive.

The `{$ELIFC }` variant is allowed only in MACPAS mode.

1.2.15 $ENDC : End conditional compilation

In MACPAS mode, this directive can be used as an alternative to the `$ENDIF` directive. It is supported for compatibility with existing Mac OS pascal compilers.

1.2.16 $ENDIF : End conditional compilation

The `{$ENDIF}` directive ends the conditional compilation initiated by the last `{$IFxxx}` directive. Any text after the `ENDIF` keyword but before the closing brace is ignored:

```
{$ENDIF some ignored text}
```

is the same as

```
{$ENDIF}
```

This is useful for indication what switch is meant to be ended.

1.2.17 $ERROR or $ERRORC : Generate error message

The following code

```
{$ERROR This code is erroneous !}
```

will display an error message when the compiler encounters it, and increase the error count of the compiler. The compiler will continue to compile, but no code will be emitted.

The $ERRORC variant is supplied for Mac Pascal compatibility.

1.2.18 $ENDREGION: End of a collapsible region

This directive is parsed for Delphi compatibility but otherwise ignored. In Delphi, it marks the end of a collapsible region in the IDE.

1.2.19 $EXTENDEDSYM: Ignored

This directive is parsed for Delphi compatibility but otherwise ignored. A warning will be displayed when this directive is encountered.

1.2.20 $EXTERNALSYM: Ignored

This directive is parsed for Delphi compatibility but otherwise ignored.

1.2.21 $F : Far or near functions

This directive is recognized for compatibility with Turbo Pascal. Under the 32-bit and 64-bit programming models, the concept of near and far calls have no meaning, hence the directive is ignored. A warning is printed to the screen, as a reminder.

As an example, the following piece of code:

```
{$F+}

Procedure TestProc;

begin
 Writeln ('Hello From TestProc');
end;

begin
 testProc
end.
```

Generates the following compiler output:

```
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
```

One can see that the verbosity level was set to display warnings.

When declaring a function as `Far` (this has the same effect as setting it between `{$F+} ...{$F-}` directives), the compiler also generates a warning:

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

The same story is true for procedures declared as `Near`. The warning displayed in that case is:

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

1.2.22 $FATAL : Generate fatal error message

The following code

```
{$FATAL This code is erroneous !}
```

will display an error message when the compiler encounters it, and the compiler will immediatly stop the compilation process.

This is mainly useful in conjunction wih `{$IFDEF}` or `{$IFOPT}` statements.

1.2.23 $FPUTYPE : Select coprocessor type

This directive selects the type of coprocessor used to do floating point calculations. The directive must be followed by the type of floating point unit. The allowed values depend on the target CPU:

all `SOFT`: FPC emulates the coprocessor (not yet implemented).

i386 `X87`, `SSE`, `SSE2`: code compiled with `SSE` uses the sse to do calculations involving a float of type `Single`. This code runs only on Pentium III and above, or AthlonXP and above. Code compiled with `SSE2` uses the sse unit to do calculations with the single and double data type. This code runs only on PentiumIV and above or Athlon64 and above

x86-64 `SSE64`

powerpc `STANDARD`

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

This directive corresponds to the `-Cf` command line option.

1.2.24 $GOTO : Support Goto and Label

If {$GOTO ON} is specified, the compiler will support Goto statements and Label declarations. By default, $GOTO OFF is assumed. This directive corresponds to the -Sg command line option.

As an example, the following code can be compiled:

```
{$GOTO ON}

label Theend;

begin
  If ParamCount=0 then
    GoTo TheEnd;
  Writeln ('You specified command line options');
TheEnd:
end.
```

Remark: When compiling assembler code using the inline assembler readers, any labels used in the assembler code must be declared, and the {$GOTO ON} directive should be used.

1.2.25 $H or $LONGSTRINGS : Use AnsiStrings

If {$LONGSTRINGS ON} is specified, the keyword String (no length specifier) will be treated as AnsiString, and the compiler will treat the corresponding variable as an ansistring, and will generate corresponding code. This switch corresponds to the -Sh command line option.

By default, the use of ansistrings is off, corresponding to {$H-}. The system unit is compiled without ansistrings, all its functions accept shortstring arguments. The same is true for all RTL units, except the sysutils unit, which is compiled with ansistrings.

However, the {$MODE} statement influences the default value of {$H}: a {$MODE DELPHI} directive implies a {$H+} statement, all other modes switch it off. As a result, you should always put {$H+} after a mode directive. This behaviour has changed, in older Free Pascal versions this was not so.

1.2.26 $HINT : Generate hint message

If the generation of hints is turned on, through the -vh command line option or the {$HINTS ON} directive, then

```
{$Hint This code should be optimized }
```

will display a hint message when the compiler encounters it.

By default, no hints are generated.

1.2.27 $HINTS : Emit hints

{$HINTS ON} switches the generation of hints on. {$HINTS OFF} switches the generation of hints off. Contrary to the command line option -vh this is a local switch, this is useful for checking parts of the code.

1.2.28 $HPPEMIT: Ignored

This directive is parsed for Delphi compatibility but otherwise ignored.

1.2.29 $IF : Start conditional compilation

The directive {$IF expr} will continue the compilation if the boolean expression expr evaluates to True. If the compilation evaluates to False, then the source is skipped to the first {$ELSE} or {$ENDIF} directive.

The compiler must be able to evaluate the expression at parse time. This means that variables or constants that are defined in the source cannot be used. Macros and symbols may be used, however.

More information on this can be found in the section about conditionals, 2.4.1 on page 57.

1.2.30 $IFC : Start conditional compilation

In MACPAS mode, this directive can be used as an alternative to the $IF directive. It is supported for compatibility with existing Mac OS pascal compilers.

1.2.31 $IFDEF Name : Start conditional compilation

If the symbol Name is not defined then the {$IFDEF name} will skip the compilation of the text that follows it to the first {$ELSE} or {$ENDIF} directive. If Name is defined, then compilation continues as if the directive wasn't there.

1.2.32 $IFNDEF : Start conditional compilation

If the symbol Name is defined then the {$IFNDEF name} will skip the compilation of the text that follows it to the first {$ELSE} or {$ENDIF} directive. If it is not defined, then compilation continues as if the directive wasn't there.

1.2.33 $IFOPT : Start conditional compilation

The {$IFOPT switch} will compile the text that follows it if the switch switch is currently in the specified state. If it isn't in the specified state, then compilation continues after the corresponding {$ELSE} or {$ENDIF} directive.

As an example:

```
{$IFOPT M+}
  Writeln ('Compiled with type information');
{$ENDIF}
```

Will compile the Writeln statement only if generation of type information is on.

Remark: The {$IFOPT} directive accepts only short options, i.e. {$IFOPT TYPEINFO} will not be accepted.

1.2.34 $IMPLICITEXCEPTIONS : Implicit finalization code generation

The compiler generates an implicit try...finally frame around each procedure that needs initialization or finalization of variables, and finalizes the variables in the finally block. This slows down these procedures (up to 5-10% sometimes). With this directive, the generation of such frames can be disabled. One should be careful with this directive, because it can lead to memory leaks if an exception occurs inside the routine. Therefore, it is set to ON by default.

1.2.35 $INFO : Generate info message

If the generation of info is turned on, through the -vi command line option, then

```
{$INFO This was coded on a rainy day by Bugs Bunny}
```

will display an info message when the compiler encounters it.

This is useful in conjunction with the {$IFDEF} directive, to show information about which part of the code is being compiled.

1.2.36 $INLINE : Allow inline code.

The {$INLINE ON} directive tells the compiler that the Inline procedure modifier should be allowed. Procedures that are declared inline are copied to the places where they are called. This has the effect that there is no actual procedure call, the code of the procedure is just copied to where the procedure is needed, this results in faster execution speed if the function or procedure is used a lot.

By default, Inline procedures are not allowed. This directive must be specified to use inlined code. The directive is equivalent to the command line switch -Si. For more information on inline routines, consult the Reference Guide.

1.2.37 $INTERFACES : Specify Interface type.

The {$INTERFACES} directive tells the compiler what it should take as the parent interface of an interface declaration which does not explicitly specify a parent interface. By default the Windows COM IUnknown interface is used. Other implementations of interfaces (CORBA or Java) do not necessarily have this interface, and for such cases, this directive can be used. It accepts the following three values:

COM Interfaces will descend from IUnknown and will be reference counted.

CORBA Interfaces will not have a parent and are not reference counted (so the programmer is responsible for bookkeeping). Corba interfaces are identified by a simple string so they are assignment compatible with strings and not TGUID.

DEFAULT Currently, this is COM.

1.2.38 $I or $IOCHECKS : Input/Output checking

The {$I-} or {$IOCHECKS OFF} directive tells the compiler not to generate input/output checking code in the program. By default, the compiler generates I/O checking code. This behaviour can be controlled globally with the -Ci switch.

When compiling using the -Ci compiler switch, the Free Pascal compiler inserts input/output checking code after every input/output call in the code. If an error occurred during input or output, then a run-time error will be generated. This switch can also be used to avoid this behaviour.

If no I/O checking code is generated, to check if something went wrong, the IOResult function can be used to see if everything went without problems.

Conversely, {$I+} will turn error-checking back on, until another directive is encountered which turns it off again.

The most common use for this switch is to check if the opening of a file went without problems, as in the following piece of code:

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

See the `IOResult` function explanation in Reference Guide for a detailed description of the possible errors that can occur when using input/output checking.

1.2.39 `$IEEEERRORS` : Enable IEEE error checking for constants

This boolean directive switches IEEE (floating point) error checking for constants on or off. It is the local equivalent of the global `-C3` command-line switch.

The following turns on IEEE (floating point) error checking for constants:

```
{$IEEEERRORS ON}
```

1.2.40 `$I` or `$INCLUDE` : Include file

The `{$I filename}` or `{$INCLUDE filename}` directive tells the compiler to read further statements from the file `filename`. The statements read there will be inserted as if they occurred in the current file.

If the file with the given filename exists, it will be included. If no extension is given, the compiler will append the .pp extension to the file and try with that filename. No other extensions are tried.

The filename can be placed between single quotes, they will not be regarded as part of the file's name. Indeed, if the filename contains a space, then it must be surrounded by single quotes:

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

will try to include the file my file name or my file name.pp.

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

will try to include the file my or my.pp.

If the filename is an asterisk (*) then the compiler will use the unit or program name as filename and try to include that. The following code

```
unit testi;

interface

{$I *}

implementation

end.
```

will include the file testi or testi.pp if they exist.

```
Type
  A = Integer;
```

Care should be taken with this mechanism, because the unit name should already match the unit filename, meaning most likely the unit will include itself recursively.

Include files can be nested, but not infinitely deep. The number of files is restricted to the number of file descriptors available to the Free Pascal compiler.

Contrary to Turbo Pascal, include files can cross blocks. I.e. a block can start in one file (with a `Begin` keyword) and can end in another (with a `End` keyword). The smallest entity in an include file must be a token, i.e. an identifier, keyword or operator.

The compiler will look for the file to include in the following places:

1. It will look in the path specified in the include file name.
2. It will look in the directory where the current source file is.
3. it will look in all directories specified in the include file search path.

Directories can be added to the include file search path with the `-Fi` command line option.

1.2.41 `$I` or `$INCLUDE` : Include compiler info

In this form:

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

the `{$INCLUDE}` directive inserts a string constant in the source code.

Here `XXX` can be one of the following:

DATE Inserts the current date.

FPCTARGET Inserts the target CPU name. (deprecated, use `FPCTARGETCPU`)

FPCTARGETCPU Inserts the target CPU name.

FPCTARGETOS Inserts the target OS name.

FPCVERSION Current compiler version number.

FILE Filename in which the directive is found.

LINE Line number on which the directive is found.

LINENUM Line number on which the directive is found. In this case, the result is an integer, not a string.

TIME Current time.

If `XXX` is none of the above, then it is assumed to be the name of an environment variable. Its value will be fetched from the environment, if it exists, otherwise an empty string is inserted. As a result, this will generate a macro with the value of the `XXX` specifier, as if it were a string (or, in the case of LINENUM, an integer).

For example, the following program

```
Program InfoDemo;

Const User = {$I %USER%};

begin
  Write ('This program was compiled at ',{$I %TIME%});
  Writeln (' on ',{$I %DATE%});
  Writeln ('By ',User);
  Writeln ('Compiler version: ',{$I %FPCVERSION%});
  Writeln ('Target CPU: ',{$I %FPCTARGET%});
end.
```

Creates the following output:

```
This program was compiled at 17:40:18 on 1998/09/09
By michael
Compiler version: 0.99.7
Target CPU: i386
```

1.2.42 $J or $WRITEABLECONST : Allow assignments to typed consts

This boolean switch tells the compiler whether or not assignments to typed constants are allowed. The default is to allow assignments to typed constants.

The following statement will switch off assignments to typed constants:

```
{$WRITEABLECONST OFF}
```

After this switch, the following statement will no longer compile:

```
Const
  MyString : String = 'Some nice string';

begin
  MyString:='Some Other string';
end.
```

But an initialized variable will still compile:

```
Var
  MyString : String = 'Some nice string';
begin
  MyString:='Some Other string';
end.
```

1.2.43 $L or $LINK : Link object file

The `{$L filename}` or `{$LINK filename}` directive tells the compiler that the file filename should be linked to the program. This cannot be used for libraries, see section 1.2.46, page 28 for that.

The compiler will look for this file in the following locations:

1. In the path specified in the object file name.

2. In the directory where the current source file is.
3. In all directories specified in the object file search path.

Directories can be added to the object file search path with the `-Fo` command line option.

On LINUX systems and on operating systems with case-sensitive filesystems (such as UNIX systems), the name is case sensitive, and must be typed exactly as it appears on your system.

Remark: Take care that the object file you're linking is in a format the linker understands. Which format this is, depends on the platform you're on. Typing `ld` or `ld -help` on the command line gives a list of formats `ld` knows about.

Other files and options can be passed to the linker using the `-k` command line option. More than one of these options can be used, and they will be passed to the linker, in the order that they were specified on the command line, just before the names of the object files that must be linked.

1.2.44 $LIBEXPORT : Ignored

This directive is recognized for Darwin pascal compilers, but is otherwise ignored.

1.2.45 $LINKFRAMEWORK : Link to a framework

The `{$LINKFRAMEWORK name}` will link to a framework named `name`. This switch is available only on the Darwin platform.

1.2.46 $LINKLIB : Link to a library

The `{$LINKLIB name}` will link to a library name. This has the effect of passing `-lname` to the linker.

As an example, consider the following unit:

```
unit getlen;

interface
{$LINKLIB c}

function strlen (P : pchar) : longint;cdecl;

implementation

function strlen (P : pchar) : longint;cdecl;external;

end.
```

If one would issue the command

```
ppc386 foo.pp
```

where foo.pp has the above unit in its `uses` clause, then the compiler would link the program to the c library, by passing the linker the `-lc` option.

The same can be obtained by removing the linklib directive in the above unit, and specify `-k-lc` on the command line:

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

Note that the linker will look for the library in the linker library search path: one should never specify a complete path to the library. The linker library search path can be set with the `-Fl` command line option.

1.2.47 $M or $TYPEINFO : Generate type info

For classes that are compiled in the `{$M+}` or `{$TYPEINFO ON}` state, the compiler will generate Run-Time Type Information (RTTI). All descendent class of a class that was compiled in the `{$M+}` state will get RTTI information too. Any class that is used as a field or property in a published section will also get RTTI information.

By default, no Run-Time Type Information is generated for published sections, making them equivalent to public sections. Only when a class (or one of its parent classes) was compiled in the `{$M+}` state, the compiler will generate RTTI for the methods and properties in the published section.

The `TPersistent` object that is present in the classes unit (part of the RTL) is generated in the `{$M+}` state. The generation of RTTI allows programmers to stream objects, and to access published properties of objects, without knowing the actual class of the object.

The run-time type information is accessible through the `TypInfo` unit, which is part of the Free Pascal Run-Time Library.

Remark: The streaming system implemented by Free Pascal requires that all streamable components be descendent from `TPersistent`. It is possible to create classes with published sections that do not descend from `TPersistent`, but those classes will not be streamed correctly by the streaming system of the Classes unit.

1.2.48 $MACRO : Allow use of macros.

In the `{$MACRO ON}` state, the compiler allows the use of C-style (although not as elaborate) macros. Macros provide a means for simple text substitution. This directive is equivalent to the command line option `-Sm`. By default, macros are not allowed.

More information on using macros can be found in section 2.2, page 55.

1.2.49 $MAXFPUREGISTERS : Maximum number of FPU registers for variables

The `{$MAXFPUREGISTERS XXX}` directive tells the compiler how much floating point variables can be kept in the floating point processor registers on an Intel X86 processor. This switch is ignored unless the `-Or` (use register variables) optimization is used.

This is quite tricky because the Intel FPU stack is limited to 8 entries. The compiler uses a heuristic algorithm to determine how much variables should be put onto the stack: in leaf procedures it is limited to 3 and in non leaf procedures to 1. But in case of a deep call tree or, even worse, a recursive procedure, this can still lead to a FPU stack overflow, so the user can tell the compiler how much (floating point) variables should be kept in registers.

The directive accepts the following arguments:

N where `N` is the maximum number of FPU registers to use. Currently this can be in the range 0 to 7.

Normal restores the heuristic and standard behavior.

Default restores the heuristic and standard behaviour.

Remark: This directive is valid until the end of the current procedure.

1.2.50 $MESSAGE : Generate info message

If the generation of info is turned on, through the `-vi` command line option, then

```
{$MESSAGE This was coded on a rainy day by Bugs Bunny }
```

will display an info message when the compiler encounters it. The effect is the same as the `{$INFO}` directive.

1.2.51 $MINENUMSIZE : Specify minimum enumeration size

This directive is provided for Delphi compatibility: it has the same effect as the `$PACKENUM` directive (see section 1.2.59, page 33).

1.2.52 $MINFPCONSTPREC : Specify floating point constant precision

This switch is the equivalent of the `-CF` command line switch. Normally, the compiler will set the precision of a floating point constant to the minimally required precision to represent it exactly. This switch can be used to ensure that the compiler never lowers the precision belowed the specified value. Supported values are 32, 64 and `DEFAULT`. 80 is not supported for implementation reasons.

Note that this has nothing to do with the actual precision used by calculations: there the type of the variable will determine what precision is used. This switch determines only with what precision a constant declaration is stored:

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

The type of the above constant will be `double` even though it can be represented exactly using `single`.

Note that a value of 80 (Extended precision) is not supported.

1.2.53 $MMX : Intel MMX support (Intel 80x86 only)

Free Pascal supports optimization for the **MMX** Intel processor (see also chapter 5).

This optimizes certain code parts for the **MMX** Intel processor, thus greatly improving speed. The speed is noticed mostly when moving large amounts of data. Things that change are

- Data with a size that is a multiple of 8 bytes is moved using the `movq` assembler instruction, which moves 8 bytes at a time

Remark: MMX support is NOT emulated on non-MMX systems, i.e. if the processor doesn't have the MMX extensions, the MMX optimizations cannot be used.

When **MMX** support is on, it is not allowed to do floating point arithmetic. It is allowed to move floating point data, but no arithmetic can be done. If floating point math must be done anyway, first **MMX** support must be switched off and the FPU must be cleared using the `emms` function of the cpu unit.

The following example will make this more clear:

```
Program MMXDemo;

uses mmx;

var
   d1 : double;
   a : array[0..10000] of double;
   i : longint;

begin
   d1:=1.0;
{$mmx+}
   { floating point data is used, but we do _no_ arithmetic }
   for i:=0 to 10000 do
     a[i]:=d2;  { this is done with 64 bit moves }
{$mmx-}
   emms;   { clear fpu }
   { now we can do floating point arithmetic }
   ...
end.
```

See the chapter on MMX (5) for more information on this topic.

1.2.54 $NODEFINE : Ignored

This directive is parsed for Delphi compatibility but is otherwise ignored.

1.2.55 $NOTE : Generate note message

If the generation of notes is turned on, through the `-vn` command line option or the `{$NOTES ON}` directive, then

```
{$NOTE Ask Santa Claus to look at this code}
```

will display a note message when the compiler encounters it.

1.2.56 $NOTES : Emit notes

`{$NOTES ON}` switches the generation of notes on. `{$NOTES OFF}` switches the generation of notes off. Contrary to the command line option `-vn` this is a local switch, this is useful for checking parts of the code.

By default, `{$NOTES}` is off.

1.2.57 $OBJECTCHECKS : Check Object

This boolean switch determines whether code to test the `SELF` pointer is inserted in methods. By default it is `OFF`. For example:

```
{$OBJECTCHECKS ON}
```

If the `SELF` pointer is `NIL` a run-time error `210` (range check) will be generated.

This switch is also activated by the `-CR` command line option.

1.2.58 $OPTIMIZATION : Enable Optimizations

This switch enables optimization. It can have the following possible values:

ON Switches on optimizations, corresponding to level 2 optimizations.

OFF Switches of all kinds of optimizations.

DEFAULT Returns to default (i.e. command-line or config file) specified optimizations.

XYZ Parses the string and switches on the optimizations found in the string. The optimizations must be separated by commas.

The following strings are supported:

LEVEL1 Level 1 optimizations

LEVEL2 Level 2 optimizations

LEVEL3 Level 3 optimizations

REGVAR Use register variables.

UNCERTAIN Use uncertain optimizations.

SIZE Optimize for size.

STACKFRAME Skip stackframes.

PEEPHOLE Peephole optimizations.

ASMCSE Use common subexpression elimination at the assembler level.

LOOPUNROLL Unroll loops

TAILREC change tail recursion to regular while

ORDERFIELDS Reorder fields if this results in better alignment.

FASTMATH Fast math operations

REMOVEEMPTYPROCS remove calls to empty procedures.

CSE Use common subexpression elimination

DFA Use DFA (Data Flow Analysis).

Example:

```
{$OPTIMIZATION ON}
```

is equivalent to

```
{$OPTIMIZATION LEVEL2}
```

Multiple optimizations can be specified with:

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

This switch is also activated by the `-Ooxxx` command line switch. Note the small 'o': it is -Oo followed by the switch name.

1.2.59 $PACKENUM or $Z : Minimum enumeration type size

This directive tells the compiler the minimum number of bytes it should use when storing enumerated types. It is of the following form:

```
{$PACKENUM xxx}
{$MINENUMSIZE xxx}
```

Where the form with $MINENUMSIZE is for Delphi compatibility. xxx can be one of 1,2 or 4, or NORMAL or DEFAULT.

The default enumeration size depends on the compiler mode:

- In Delphi and TP mode, the size is 1.
- In MacPas mode, the size is 2.
- In all other modes, the default is 4.

As an alternative form one can use {$Z1}, {$Z2} {$Z4}. The {$Z} form takes a boolean argument, where ON is equivalent to {$Z4} and OFF is equivalent to {$Z1}.

So the following code

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

will use 1 byte to store a variable of type Days, whereas it nomally would use 4 bytes. The above code is equivalent to

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

or equivalent to

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

1.2.60 $PACKRECORDS : Alignment of record elements

This directive controls the byte alignment of the elements in a record, object or class type definition.

It is of the following form:

```
{$PACKRECORDS n}
```

Where n is one of 1, 2, 4, 8, 16, C, NORMAL or DEFAULT. This means that the elements of a record which have size greater than n will be aligned on n byte boundaries. Elements with size less than or equal to n will be aligned to a natural boundary, i.e. to a power of two that is equal to or larger than the element's size. The special value C is used to specify alignment as by the GNU CC compiler. It should be used only when making import units for C routines.

The default alignment (which can be selected with DEFAULT) is natural alignment, contrary to Turbo Pascal, where it is 1.

More information on this and an example program can be found in the reference guide, in the section about record types.

The following shorthands can be used for this directive:

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

1.2.61 $PACKSET : Specify set size

The $PACKSET directive takes a numeric argument of 1, 2, 4 or 8. This number determines the number of bytes used to store a set: The compiler rounds the number of bytes needed to store the set down/up to the closest multiple of the PACKSET setting, with the exception that 3-byte sets are always rounded up to 4-byte sets.

Other allowed values are FIXED, DEFAULT, or NORMAL. With these values, the compiler stores sets with less than 32 elements in 4 bytes, and sets with less than 256 elements in 32 bytes.

1.2.62 $POP : Restore compiler settings

The $POP directive restores the values of all local compiler directives with the last values that were stored on the settings stack. The settings are then deleted from the stack.

The settings can be stored on the stack with the $PUSH directive (see section 1.2.63, page 34).

Note that global settings are not restored by this directive.

1.2.63 $PUSH : Save compiler settings

The $PUSH directive saves the values of all local compiler directives that were stored on the settings stack. Up to 20 sets of settings can be stored on the stack.

The settings can be restored from the stack using the $POP directive (see section 1.2.62, page 34).

Note that global [settings (search paths etc.) are not saved by this directive.

The settings stack is preserved accross units, i.e. when the compiler starts compiling a new unit, the stack is not emptied.

1.2.64 $Q or $OV or $OVERFLOWCHECKS: Overflow checking

The {$Q+} or {$OV+} (MACPAS mode only) or {$OVERFLOWCHECKS ON} directive turns on integer overflow checking. This means that the compiler inserts code to check for overflow when doing computations with integers. When an overflow occurs, the run-time library will generate a run-time error 215: It prints a message Overflow at xxx, and exits the program with exit code 215.

Remark: Overflow checking behaviour is not the same as in Turbo Pascal since all arithmetic operations are done via 32-bit or 64-bit values. Furthermore, the Inc() and Dec standard system procedures *are* checked for overflow in Free Pascal, while in Turbo Pascal they are not.

Using the {$Q-} switch (or the {$OV-} switch in MACPAS mode) switches off the overflow checking code generation.

The generation of overflow checking code can also be controlled using the `-Co` command line compiler option (see the User's Guide).

In Delphi, overflow checking is only switchable on a procedure level. In Free Pascal, the `{$Q }` directive can be used on an expression-level.

1.2.65 $R or $RANGECHECKS : Range checking

By default, the compiler doesn't generate code to check the ranges of array indices, enumeration types, subrange types, etc. Specifying the `{$R+}` switch tells the computer to generate code to check these indices. If, at run-time, an index or enumeration type is specified that is out of the declared range of the compiler, then a run-time error is generated, and the program exits with exit code 201. This can happen when doing a typecast (implicit or explicit) on an enumeration type or subrange type.

The `{$RANGECHECKS OFF}` switch tells the compiler not to generate range checking code. This may result in faulty program behaviour, but no run-time errors will be generated.

Remark: The standard functions `val` and `Read` will also check ranges when the call is compiled in `{$R+}` mode.

In Delphi, range checking is only switchable on a procedure level. In Free Pascal, the `{$R }` directive can be used on an expression-level.

1.2.66 $REGION : Mark start of collapsible region

The `$REGION` directive is recognised for Delphi compatibility only. In Delphi it serves to mark the beginning of a collapsible region in the IDE.

1.2.67 $R or $RESOURCE : Include resource

This directive includes a resource in the binary. The argument to this directive is the resource file to include in the binary:

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

Will include the file icons.res as a resource in the binary. Up to version 2.2.N, resources are supported only for Windows (native resources are used) and for platforms using ELF binaries (linux, BSD). As of version 2.3.1, resources have been implemented for all supported platforms.

The asterix can be used as a placeholder for the current unit/program filename:

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

will include myunit.res.

1.2.68 $SATURATION : Saturation operations (Intel 80x86 only)

This works only on the intel compiler, and MMX support must be on (`{$MMX +}`) for this to have any effect. See the section on saturation support (section 5.2, page 74) for more information on the effect of this directive.

1.2.69 $SAFEFPUEXCEPTIONS Wait when storing FPU values on Intel x86

This boolean directives controls how the compiler generates code to store FPU values. If set to ON, the compiler inserts a FWAIT opcode after the store of a floating point value, so any errors are reported at once. This slows down code, but makes sure that the errors are reported on the location where the instruction was executed.

1.2.70 $SCOPEDENUMS Control use of scoped enumeration types

The boolean $SCOPEDENUMS directive controls how enumeration type values are inserted in the symbol tables. In its default state (OFF) the enumeration values are inserted directly in the symbol table of the current scope. If the directive is set to ON then the values are inserted inside a scope with the name of the enumeration type.

Practically this means that the following is the default behaviour:

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

Var
  A : TMyEnum;

begin
  A:=one;
end.
```

The value one can be referenced directly. The following will give an error:

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

However, if the SCOPEDENUMS directive is set to ON, then the assignment must be made as follows:

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

Var
  A : TMyEnum;

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

i.e. the value must be prefixed with the type name.

1.2.71 $SETC : Define and assign a value to a symbol

In MACPAS mode, this directive can be used to define compiler symbols. It is an alternative to the $DEFINE directive for macros. It is supported for compatibility with existing Mac OS Pascal compilers. It will define a symbol with a certain value (called a compiler variable expression).

The expression syntax is similar to expressions used in macros, but the expression must be evaluated at compile-time by the compiler. This means that only some basic arithmetic and logical operators can be used, and some extra possibilities such as the TRUE,FALSE and UNDEFINED operators:

```
{$SETC TARGET_CPU_PPC  := NOT UNDEFINED CPUPOWERPC}
{$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)}
```

The := assignment symbol may be replaced with the = symbol.

Note that this command works only in MACPAS mode, but independent of the -Sm command line option or {$MACRO } directive.

1.2.72 $STATIC : Allow use of Static keyword.

If you specify the {$STATIC ON} directive, then Static methods are allowed for objects. Static objects methods do not require a Self variable. They are equivalent to Class methods for classes. By default, Static methods are not allowed. Class methods are always allowed. Note that also static fields can be defined.

This directive is equivalent to the -St command line option.

1.2.73 $STOP : Generate fatal error message

The following code

```
{$STOP This code is erroneous !}
```

will display an error message when the compiler encounters it. The compiler will immediatly stop the compilation process.

It has the same effect as the {$FATAL} directive.

1.2.74 $STRINGCHECKS : Ignored

This directive is recognized for Delphi compatibility, but is currently ignored. In Delphi, it controls the generation of code that checks the sanity of string variables and arguments.

1.2.75 $T or $TYPEDADDRESS : Typed address operator (@)

In the {$T+} or {$TYPEDADDRESS ON} state, the @ operator, when applied to a variable, returns a result of type ^T, if the type of the variable is T. In the {$T-} state, the result is always an untyped pointer, which is assignment compatible with all other pointer types.

For example, the following code will not compile:

```
{$T+}

Var
  I : Integer;
  P : PChar;
```

```
begin
  P:=@I;
end.
```

The compiler will give a type mismatch error:

```
testt.pp(8,6) Error: Incompatible types: got "^SmallInt" expected "PChar"
```

By default however, the address operator returns an untyped pointer.

1.2.76 $UNDEF or $UNDEFC : Undefine a symbol

The directive

```
{$UNDEF name}
```

un-defines the symbol `name` if it was previously defined. `Name` is case insensitive.

In Mac Pascal mode, `$UNDEFC` is equivalent to `$UNDEF`, and is provided for Mac Pascal compatibility.

1.2.77 $V or $VARSTRINGCHECKS : Var-string checking

The `{$VARSTRINGCHECKS }` determines how strict the compiler is when checking string type compatibility for strings passed by reference. When in the `+` or `ON` state, the compiler checks that strings passed as parameters are of the string type as the declared parameters of the procedure.

By default, the compiler assumes that all short strings are type compatible. That is, the following code will compile:

```
Procedure MyProcedure(var Arg: String[10]);

begin
  Writeln('Arg ',Arg);
end;

Var
  S : String[12];

begin
  S:='123456789012';
  Myprocedure(S);
end.
```

The types of `Arg` and `S` are strictly speaking not compatible: The `Arg` parameter is a string of length 10, and the variable `S` is a string of length 12: The value will be silently truncated to a string of length 10.

In the `{$V+}` state, this code will trigger a compiler error:

```
testv.pp(14,16) Error: string types doesn't match, because of $V+ mode
```

Note that this is only for strings passed by reference, not for strings passed by value.

1.2.78 $W or $STACKFRAMES : Generate stackframes

The {$W} switch directive controls the generation of stackframes. In the on state, the compiler will generate a stackframe for every procedure or function.

In the off state, the compiler will omit the generation of a stackframe if the following conditions are satisfied:

- The procedure has no parameters.
- The procedure has no local variables.
- If the procedure is not an assembler procedure, it must not have a asm ...end; block.
- it is not a constructor or destructor.

If these conditions are satisfied, the stack frame will be omitted.

1.2.79 $WAIT : Wait for enter key press

If the compiler encounters a

```
{$WAIT}
```

directive, it will resume compiling only after the user has pressed the enter key. If the generation of info messages is turned on, then the compiler will display the following message:

```
Press <return> to continue
```

before waiting for a keypress.

Remark: This may interfere with automatic compilation processes. It should be used only for compiler debugging purposes.

1.2.80 $WARN : Control emission of warnings

This directive allows to selectively turn on or off the emission of warnings. It takes the following form

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

ON or + turns on emission of the warning. The OFF or - values suppress the warning. ERROR promotes the warning to an error, and the compiler will treat it as such.

The IDENTIFIER is the name of a warning message. The following names are recognized:

CONSTRUCTING_ABSTRACT Constructing an instance of a class with abstract methods.

IMPLICIT_VARIANTS Implicit use of the variants unit.

NO_RETVAL Function result is not set.

SYMBOL_DEPRECATED Deprecated symbol.

SYMBOL_EXPERIMENTAL Experimental symbol

SYMBOL_LIBRARY Not used.

SYMBOL_PLATFORM Platform-dependent symbol.

SYMBOL_UNIMPLEMENTED Unimplemented symbol.

UNIT_DEPRECATED Deprecated unit.

UNIT_EXPERIMENTAL Experimental unit.

UNIT_LIBRARY

UNIT_PLATFORM Platform dependent unit.

UNIT_UNIMPLEMENTED Unimplemented unit.

ZERO_NIL_COMPAT Converting 0 to NIL

IMPLICIT_STRING_CAST Implicit string type conversion

IMPLICIT_STRING_CAST_LOSS Implicit string typecast with potential data loss from "$1" to "$2"

EXPLICIT_STRING_CAST Explicit string type conversion

EXPLICIT_STRING_CAST_LOSS Explicit string typecast with potential data loss from "$1" to "$2"

CVT_NARROWING_STRING_LOST Unicode constant cast with potential data loss

Besides the above text identifiers, the identifier can also be a message number. The numbers of the messages are displayed when the `-vq` command-line option is used.

1.2.81 `$WARNING` : Generate warning message

If the generation of warnings is turned on, through the `-vw` command line option or the `{$WARNINGS ON}` directive, then

```
{$WARNING This is dubious code}
```

will display a warning message when the compiler encounters it.

1.2.82 `$WARNINGS` : Emit warnings

`{$WARNINGS ON}` switches the generation of warnings on. `{$WARNINGS OFF}` switches the generation of warnings off. Contrary to the command line option `-vw` this is a local switch, this is useful for checking parts of your code.

By default, no warnings are emitted.

1.2.83 `$Z1`, `$Z2` and `$Z4`

This switch is an equivalent of the `{$PACKENUM }` switch (see section 1.2.59, page 33).

1.3 Global directives

Global directives affect the whole of the compilation process. That is why they also have a command line counterpart. The command line counterpart is given for each of the directives. They must be specified *before* the `unit` or `program` clause in a source file, or they will have no effect.

1.3.1 `$APPID` : Specify application ID.

Used on the PALM os only, it can be set to specify the application name, which can be viewed on the Palm only. This directive only makes sense in a program source file, not in a unit.

```
{$APPID MyApplication}
```

1.3.2 `$APPNAME` : Specify application name.

Used on the PALM os only, it can be set to specify the application name which can be viewed on the Palm only. This directive only makes sense in a program source file, not in a unit.

```
{$APPNAME My Application, compiled using Free Pascal.}
```

1.3.3 `$APPTYPE` : Specify type of application.

This directive is currently only supported on the following targets: Win32, Mac, OS2 and AmigaOS. On other targets, the directive is ignored.

The `{$APPTYPE XXX}` accepts one argument which specifies what kind of application is compiled. It can have the following values:

CONSOLE A console application. A terminal will be created and standard input, output and standard error file descriptors will be initialized. In Windows, a terminal window will be created. This is the default.

Note that on Mac OS such applications cannot take command line options, nor return a result code. They will run in a special terminal window, implemented as a SIOW application, see the MPW documentation for details.

On OS/2, these applications can run both full-screen and in a terminal window.

LINUX applications are always console applications. The application itself can decide to close the standard files, though.

FS Specifies a full-screen VIO application on OS/2. These applications use a special BIOS-like API to program the screen. OS/2 starts these application allways in full screen.

GUI Specifying the `{$APPTYPE GUI}` directive will mark the application as a graphical application; no console window will be opened when the application is run. No standard file descriptors will be initialized, using them (with e.g. `writeln` statements) will produce a run-time error. If run from the command line, the command prompt will be returned immediatly after the application was started.

On OS/2 and Mac OS, the GUI application type creates a GUI application, as on Windows. On OS/2, this is a real Presentation Manager application.

TOOL This is a special directive for the Mac OS. It tells the compiler to create a tool application: It initializes `Input`, `Output` and `StdErr` files, it can take parameters and return a result code. It is implemented as an MPW tool which can only be run by MPW or ToolServer.

Care should be taken when compiling GUI applications; the Input and Output files are not available in a GUI application, and attempting to read from or write to them will result in a run-time error.

It is possible to determine the application type of a WINDOWS or AMIGA application at runtime. The IsConsole constant, declared in the Win32 and Amiga system units as

```
Const
  IsConsole : Boolean;
```

contains True if the application is a console application, False if the application is a GUI application.

1.3.4 $CALLING : Default calling convention

This directive allows specifying the default calling convention used by the compiler, when no calling convention is specified for a procedure or function declaration. It can be one of the following values:

CDECL C compiler calling convention.

CPPDECL C++ compiler calling convention.

FAR16 Ignored, but parsed for Turbo Pascal compatibility reasons.

FPCCALL Older FPC (1.0.X and before) standard calling convention. If a lot of direct assembler blocks are used, this mode should be used for maximum compatibility.

INLINE Use inline code: the code for the function is inserted whenever it is called.

PASCAL Pascal calling convention.

REGISTER Register calling convention (the default).

SAFECALL Safecall calling convention (used in COM): The called procedure/function saves all registers.

STDCALL Windows library calling convention.

SOFTFLOAT For ARM processors.

This directive is equivalent to the -Cc command line option.

1.3.5 $CODEPAGE : Set the source codepage

This switch sets the codepage of the rest of the source file. The codepage is only taken into account when interpreting literal strings, the actual code must be in US-ASCII. The argument to this switch is the name of the code page to be used.

```
{$CODEPAGE UTF8}
```

The 'UTF-8' codepage can be specified as 'UTF-8' or 'UTF8'. The list of supported codepages is the list of codepages supported by the charset unit of the RTL.

1.3.6 $COPYRIGHT specify copyright info

This is intended for the NETWARE version of the compiler: it specifies the copyright information that can be viewed on a module for a Netware OS.

For example:

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

1.3.7 $D or $DEBUGINFO : Debugging symbols

When this switch is on, the compiler inserts GNU debugging information in the executable. The effect of this switch is the same as the command line switch `-g`.

By default, insertion of debugging information is off.

1.3.8 $DESCRIPTION : Application description

This switch is recognised for compatibility only, but is ignored completely by the compiler. At a later stage, this switch may be activated.

1.3.9 $E : Emulation of coprocessor

This directive controls the emulation of the coprocessor. There is no command line counterpart for this directive.

Intel 80x86 version

When this switch is enabled, all floating point instructions which are not supported by standard coprocessor emulators will give out a warning.

The compiler itself doesn't do the emulation of the coprocessor.

To use coprocessor emulation under DOS (go32v2) you must use the emu387 unit, which contains correct initialization code for the emulator.

Under LINUX and most UNIX'es, the kernel takes care of the coprocessor support, so this switch is not necessary on those platforms.

Motorola 680x0 version

When the switch is on, no floating point opcodes are emitted by the code generator. Instead, internal run-time library routines are called to do the necessary calculations. In this case all real types are mapped to the single IEEE floating point type.

Remark: By default, emulation is on for non-unix targets. For unix targets, floating point emulation (if required) is handled by the operating system, and by default it is off.

1.3.10 $EXTENSION : Extension of generated binary.

This directive can appear in a program or library source file: it sets the extension for the generated binary/library. The initial dot (sometimes associated with the extension) may not be present.

Example:

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

will set the extension to 'cgi'.

1.3.11 $FRAMEWORKPATH : Specify framework path.

This option serves to specify the framework search path on Darwin, where the compiler looks for framework files. Used as

```
{$FRAMEWORKPATH XXX}
```

it will add XXX to the framework path. The value XXX can contain one or more paths, separated by semi-colons or colons.

1.3.12 $G : Generate 80286 code

This option is recognised for Turbo Pascal compatibility, but is ignored, since the compiler works only on 32-bit and 64-bit processors.

1.3.13 $IMAGEBASE : Specify DLL image base location.

This option can be used to set the base location for a DLL on windows-based systems. The directive needs a memory location as an option (use $ to specify a hexadecimal value). It is the equivalent of the -WB command-line option.

The following sets the base location to 0x00400000

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

1.3.14 $INCLUDEPATH : Specify include path.

This option serves to specify the include path, where the compiler looks for include files. Used as

```
{$INCLUDEPATH XXX}
```

it will add XXX to the include path. The value XXX can contain one or more paths, separated by semi-colons or colons.

For example:

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

{$I strings.inc}
```

will add the directories ../inc and ../i386 to the include path of the compiler. The compiler will look for the file strings.inc in both these directories, and will include the first found file. This directive is equivalent to the -Fi command line switch.

Caution is in order when using this directive: If you distribute files, the places of the files may not be the same as on your machine; moreover, the directory structure may be different. In general it would be fair to say that you should avoid using *absolute* paths. Instead, one should use *relative* paths only, as in the example above.

1.3.15 $L or $LOCALSYMBOLS : Local symbol information

This switch (not to be confused with the local {$L file} file linking directive) is recognised for Turbo Pascal compatibility, but is ignored. Generation of symbol information is controlled by the $D switch.

1.3.16 $LIBPREFIX : Specify library filename prefix.

The LIBPREFIX directive has a similar function as the {$EXTENSION } and {$LIBSUFFIX } compiler directives: It sets the prefix of the library. This is by default 'lib' on unices, and empty on windows. The name is simply prepended to the filename.

Example:

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

begin
end.
```

will result in a filename librarytl.so on linux, or librarytl.dll on windows.

1.3.17 $LIBRARYPATH : Specify library path.

This option serves to specify the library path, where the linker looks for static or dynamic libraries. {$LIBRARYPATH XXX} will add XXX to the library path. XXX can contain one or more paths, separated by semi-colons or colons.

For example:

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

{$LINKLIB X11}
```

will add the directories /usr/X11/lib and /usr/local/lib to the linker library path. The linker will look for the library libX11.so in both these directories, and use the first found file. This directive is equivalent to the -Fl command line switch.

Caution is in order when using this directive: If you distribute files, the places of the libraries may not be the same as on your machine; moreover, the directory structure may be different. In general it would be fair to say that you should avoid using this directive. If you are not sure, it is better practice to use makefiles and makefile variables.

1.3.18 $LIBSUFFIX : Set the library suffix

The LIBSUFFIX directive has a similar function as the {$EXTENSION } and {$LIBPREFIX } compiler directives. directive: it sets the suffix of the library. This is usually used for version numbers: it is simply added to the outputfilename, before the extension.

Example:

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

begin
end.
```

will result in a filename libtl-1.2.3.so on linux, or libtl-1.2.3.dll on windows.

1.3.19 $MAXSTACKSIZE : Set maximum stack size

The {$MINSTACKSIZE} sets the maximum stack size for an executable on Windows-based systems. It needs an argument, the size (in bytes) of the stack. The maximum value is $7FFFFFFF, the minimum value is 2048 or the value of $MINSTACKSIZE if it was specified.

The following example sets the maximum stack size to$FFFFFF bytes:

```
{$MINSTACKSIZE $FFFFFF}
```

1.3.20 $M or $MEMORY : Memory sizes

This switch can be used to set the heap and stacksize. Its format is as follows:

```
{$M StackSize,HeapSize}
```

where StackSize and HeapSize should be two integer values, greater than 1024. The first number sets the size of the stack, and the second the size of the heap. The stack size setting is ignored on Unix platforms unless stack checking is enabled: in that case the stack checking code will use the size set here as maximum stack size.

On those systems, in addition to the stack size set here, the operating system or the run environment may have set other (possibly more strict) limits on stack size using the OS'es ulimit system calls.

The two numbers can be set on the command line using the -Ch and -Cs switches.

1.3.21 $MINSTACKSIZE : Set minimum stack size

The {$MINSTACKSIZE} sets the minimum stack size for an executable on Windows-based systems. It needs an argument, the size (in bytes) of the stack. This must be a number larger than 1024.

The following example sets the minimum stack size to 2048 bytes:

```
{$MINSTACKSIZE 2048}
```

1.3.22 $MODE : Set compiler compatibility mode

The {$MODE} sets the compatibility mode of the compiler. This is equivalent to setting one of the command line options -So, -Sd, -Sp or -S2. it has the following arguments:

Default Default mode. This reverts back to the mode that was set on the command line.

Delphi Delphi compatibility mode. All object-pascal extensions are enabled. This is the same as the command line option -Sd. Note that this also implies {$H ON} (i.e., in Delphi mode, ansistrings are the default).

TP Turbo pascal compatibility mode. Object pascal extensions are disabled, except ansistrings, which remain valid. This is the same as the command line option -So.

FPC FPC mode. This is the default, if no command line switch is supplied.

OBJFPC Object pascal mode. This is the same as the -S2 command line option.

MACPAS MACPAS mode. In this mode, the compiler tries to be more compatible to commonly used pascal dialects on the Mac OS, such as Think Pascal, Metrowerks Pascal, MPW Pascal.

For an exact description of each of these modes, see appendix D, on page 150.

1.3.23 $MODESWITCH : Select mode features

As of FPC 2.3.1, the {$MODESWITCH} directive selects some of the features that a {$MODE } directive selects: it can be used to use features that would otherwise not be available in the current mode. For instance, one wishes to program in TP mode, but would like to use the 'Out' parameter, an option available only in Delphi mode. The {$MODESWITCH } directive allows to activate or deactivate some individual mode features, while not changing the current compiler mode.

This switch is a global switch, and can be used wherever the {$MODE} switch can be used.

The syntax is as follows:

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

The first two will switch on feature XXX, the last one will switch it off.

The feature XXX can be one of the following:

CLASS Use object pascal classes.

OBJPAS Automatically include the ObjPas unit.

RESULT Enable the Result identifier for function results.

PCHARTOSTRING Allow automatic conversion of null-terminated strings to strings,

CVAR Allow the use of the CVAR keyword.

NESTEDCOMMENTS Allow use of nested comments.

CLASSICPROCVARS Use classical procedural variables.

MACPROCVARS Use mac-style procedural variables.

REPEATFORWARD Implementation and Forward declaration must match completely.

POINTERTOPROCVAR Allow silent conversion of pointers to procedural variables.

AUTODEREF Automatic (silent) dereferencing of typed pointers.

INITFINAL Allow use of Initialization and Finalization

ANSISTRINGS Allow use of ansistrings.

OUT Allow use of the out parameter type.

DEFAULTPARAMETERS Allow use of default parameter values.

HINTDIRECTIVE Support the hint directives (deprecated, platform etc.)

DUPLICATELOCALS Allow local variables in class methods to have the same names as properties of the class.

PROPERTIES Allow use of global properties.

ALLOWINLINE Allow inline procedures.

EXCEPTIONS Allow the use of exceptions.

ADVANCEDRECORDS allow the use of advanced records (i.e. records with methods)

Hence, the following:

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

Will switch on the support for the `out` parameter type in TP mode. It is equivalent to

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

1.3.24 $N : Numeric processing

This switch is recognised for Turbo Pascal compatibility, but is otherwise ignored, since the compiler always uses the coprocessor for floating point mathematics.

1.3.25 $O : Level 2 Optimizations

In earlier versions of FPC, this switch was recognised for Turbo Pascal compatibility, but was otherwise ignored: The concept of overlay code is not needed in 32-bit or 64-bit programs.

In newer versions of FPC (certainly as of 2.0.0), this switch became a Delphi compatible switch: it has the same meaning as the `{$OPTMIZATIONS ON/OFF}` switch, switching on or off level 2 optimizations.

See section 1.2.58 on page 32 for more explanations and more detailed optimization settings.

1.3.26 $OBJECTPATH : Specify object path.

This option serves to specify the object path, where the compiler looks for object files. `{$OBJECTPATH XXX}` will add `XXX` to the object path. `XXX` can contain one or more paths, separated by semi-colons or colons.

For example:

```
{$OBJECTPATH ../inc;../i386}

{$L strings.o}
```

will add the directories ../inc and ../i386 to the object path of the compiler. The compiler will look for the file strings.o in both these directories, and will link the first found file in the program. This directive is equivalent to the `-Fo` command line switch.

Caution is in order when using this directive: If you distribute files, the places of the files may not be the same as on your machine; moreover, the directory structure may be different. In general it would be fair to say that you should avoid using *absolute* paths, instead use *relative* paths, as in the example above. Only use this directive if you are certain of the places where the files reside. If you are not sure, it is better practice to use makefiles and makefile variables.

1.3.27 $P or $OPENSTRINGS : Use open strings

If this switch is on, all function or procedure parameters of type string are considered to be open string parameters; this parameter only has effect for short strings, not for ansistrings.

When using openstrings, the declared type of the string can be different from the type of string that is actually passed, even for strings that are passed by reference. The declared size of the string passed can be examined with the `High(P)` call.

By default, the use of openstrings is off.

1.3.28 $PASCALMAINNAME : Set entry point name

The `{$PASCALMAINNAME NNN}` directive sets the assembler symbol name of the program or library entry point to `NNN`. This directive is the equivalent of the `-XM` command line switch.

Under normal circumstances, it should not be necessary to use this switch.

1.3.29 $PIC : Generate PIC code or not

The `{$PIC }` directive takes a boolean argument and tells the compiler whether it should generate PIC (Position Indepedent Code) or not. This directive is the equivalent of the `-Cg` command line switch.

This directive is only useful on Unix platforms: Units should be compiled using PIC code if they are supposed to be in a library. For programs, using PIC code is not needed, but it doesn't hurt either (although PIC code is slower).

The following

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

tells the compiler to compile myunit using PIC code.

1.3.30 $POINTERMATH : Allow use of pointer math

This boolean directive enables or disables the use of pointer arithmatics in expressions nivolving pointers. When enabled, it allows to take the difference of 2 pointers, or to add an integer value to pointers? By default, POINTERMATH is on.

The following

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

tells the compiler to give an error whenever pointer math appears in an expression.

1.3.31 $PROFILE : Profiling

This directive turns the generation of profiling code on (or off). It is equivalent to the `-gp` command line option. Default is `OFF`. This directive only makes sense in a program source file, not in a unit.

1.3.32 $S : Stack checking

The `{$S+}` directive tells the compiler to generate stack checking code. This generates code to check if a stack overflow occurred, i.e. to see whether the stack has grown beyond its maximally

allowed size. If the stack grows beyond the maximum size, then a run-time error is generated, and the program will exit with exit code 202.

Specifying `{$S-}` will turn generation of stack-checking code off.

The command line compiler switch `-Ct` has the same effect as the `{$S+}` directive.

By default, no stack checking is performed.

Remark: Stack checking can only be used to provide help during debugging, to try and track routines that use an excessive amount of local memory. It is not intended and cannot be used to actually safely handle such errors. It does not matter whether the error handling is through exception handling or otherwise.

When a stack error occurs, this is a fatal error and the application cannot be kept running correctly, neither in a production environment, nor under debugging.

1.3.33 $SCREENNAME : Specify screen name

This directive can be used for the Novell netware targets to specify the screen name. The argument is the screen name to be used.

```
{$SCREENNAME My Nice Screen}
```

Will set the screenname of the current application to 'My Nice Screen'.

1.3.34 $SETPEFLAGS : Specify PE Executable flags

The `$SETPEFLAGS` executable sets the PE flags on Windows. These flags are written to the binary PE header file. It expects a numerical constant as an argument, this constant will be written to the header block.

Under normal circumstances, the compiler itself will determine the values of the PE flags to write to the binary file, but this directive can be used to override the compiler's behaviour.

1.3.35 $SMARTLINK : Use smartlinking

A unit that is compiled in the `{$SMARTLINK ON}` state will be compiled in such a way that it can be used for smartlinking. This means that the unit is chopped in logical pieces: each procedure is put in its own object file, and all object files are put together in a big archive. When using such a unit, only the pieces of code that you really need or call will be linked in your program, thus reducing the size of your executable substantially.

Beware: using smartlinked units slows down the compilation process, because a separate object file must be created for each procedure. If you have units with many functions and procedures, this can be a time consuming process, even more so if you use an external assembler (the assembler is called to assemble each procedure or function code block separately).

The smartlinking directive should be specified *before* the unit declaration part:

```
{$SMARTLINK ON}

Unit MyUnit;

Interface
 ...
```

This directive is equivalent to the `-CX` command line switch.

1.3.36 $SYSCALLS : Select system calling convention on Amiga/MorphOS

This directive sets the system calling convention to use on a MorphOS or Amiga system. The directive needs an argument, which should be one of the following:

LEGACY

SYSV

SYSVBASE

BASESYSV

R12BASE

The directive will generate a warning if used on another system.

1.3.37 $THREADNAME : Set thread name in Netware

This directive can be set to specify the thread name when compiling for Netware.

1.3.38 $UNITPATH : Specify unit path.

This option serves to specify the unit path, where the compiler looks for unit files. `{$UNITPATH XXX}` will add `XXX` to the unit path. `XXX` can contain one or more paths, separated by semi-colons or colons.

For example:

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

Uses strings;
```

will add the directories ../units and ../i386/units to the unit path of the compiler. The compiler will look for the file strings.ppu in both these directories, and will link the first found file in the program. This directive is equivalent to the `-Fu` command line switch.

Caution is in order when using this directive: If you distribute files, the places of the files may not be the same as on your machine; moreover, the directory structure may be different. In general it would be fair to say that you should avoid using *absolute* paths, instead use *relative* paths, as in the example above. Only use this directive if you are certain of the places where the files reside. If you are not sure, it is better practice to use makefiles and makefile variables.

Note that this switch does not propagate to other units, i.e. it's scope is limited to the current unit.

1.3.39 $VARPROPSETTER : Enable use of var/out/const parameters for property setters.

This boolean directive is meant to import COM interfaces. Sometimes COM interfaces have property setters which accept arguments that are not by value, but by reference. These setters are normally forbidden. This flag enables the use of property setters with `var`, `const`, `out` arguments. By default it is OFF. The effect is on interface declarations, but also on class definitions.

The following example only compiles in the ON state:

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

In the `OFF` state, the following error will be generated:

```
testvp.pp(7,48) Error: Illegal symbol for property access
```

1.3.40 $VERSION : Specify DLL version.

On WINDOWS, this can be used to specify a version number for a library. This version number will be used when the library is installed, and can be viewed in the Windows Explorer by opening the property sheet of the DLL and looking on the tab 'Version'. The version number consists of minimally one, maximum 3 numbers:

```
{$VERSION 1}
```

Or:

```
{$VERSION 1.1}
```

And even:

```
{$VERSION 1.1.1}
```

This cannot yet be used for executables on Windows, but may be activated in the future.

1.3.41 $WEAKPACKAGEUNIT : ignored

This switch is parsed for Delphi compatibility but is otherwise ignored. The compiler will write a warning when it is encountered.

1.3.42 $X or $EXTENDEDSYNTAX : Extended syntax

Extended syntax allows you to drop the result of a function. This means that you can use a function call as if it were a procedure. By default this feature is on. You can switch it off using the `{$X-}` or `{$EXTENDEDSYNTAX OFF}`directive.

The following, for instance, will not compile:

```
function Func (var Arg : sometype) : longint;
begin
...          { declaration of Func }
end;

...

{$X-}
Func (A);
```

The reason this construct is supported is that you may wish to call a function for certain side-effects it has, but you don't need the function result. In this case you don't need to assign the function result, saving you an extra variable.

The command line compiler switch `-Sa1` has the same effect as the `{$X+}` directive.

By default, extended syntax is assumed.

1.3.43 `$Y` or `$REFERENCEINFO` : Insert Browser information

This switch controls the generation of browser information. It is recognized for compatibility with Turbo Pascal and Delphi only, as Browser information generation is not yet fully supported.

Chapter 2

Using conditionals, messages and macros

The Free Pascal compiler supports conditionals as in normal Turbo Pascal, Delphi or Mac OS Pascal. It does, however, more than that. It allows you to make macros which can be used in your code, and it allows you to define messages or errors which will be displayed when compiling. It also has support for compile-time variables and compile-time expressions, as commonly found in Mac OS compilers.

The various conditional compilation directives (`$IF`, `$IFDEF`, `$IFOPT` are used in combination with `$DEFINE` to allow the programmer to choose at compile time which portions of the code should be compiled. This can be used for instance

- To choose an implementation for one operating system over another.
- To choose a demonstration version or a full version.
- To distinguish between a debug version and a version for shipping.

These options are then chosen when the program is compiled, including or excluding parts of the code as needed. This is opposed to using normal variables and running through selected portions of code at run time, in which case extra code is included in the executable.

2.1 Conditionals

The rules for using conditional symbols are the same as under Turbo Pascal or Delphi. Defining a symbol goes as follows:

```
{$define Symbol}
```

From this point on in your code, the compiler knows the symbol `Symbol`. Symbols are, like the Pascal language, case insensitive.

You can also define a symbol on the command line. the `-dSymbol` option defines the symbol `Symbol`. You can specify as many symbols on the command line as you want.

Undefining an existing symbol is done in a similar way:

```
{$undef Symbol}
```

If the symbol didn't exist yet, this doesn't do anything. If the symbol existed previously, the symbol will be erased, and will not be recognized any more in the code following the `{$undef ...}` statement.

You can also undefine symbols from the command line with the `-u` command line switch.

To compile code conditionally, depending on whether a symbol is defined or not, you can enclose the code in a `{$ifdef Symbol}...{$endif}` pair. For instance the following code will never be compiled:

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

Similarly, you can enclose your code in a `{$ifndef Symbol}...{$endif}` pair. Then the code between the pair will only be compiled when the used symbol doesn't exist. For example, in the following code, the call to the `DoSomething` will always be compiled:

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

You can combine the two alternatives in one structure, namely as follows

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

In this example, if `MySymbol` exists, then the call to `DoSomething` will be compiled. If it doesn't exist, the call to `DoSomethingElse` is compiled.

2.1.1 Predefined symbols

The Free Pascal compiler defines some symbols before starting to compile your program or unit. You can use these symbols to differentiate between different versions of the compiler, and between different compilers. To get all the possible defines when starting compilation, see appendix G

Remark: Symbols, even when they're defined in the interface part of a unit, are not available outside that unit.

2.2 Macros

Macros are very much like symbols or compile-time variables in their syntax, the difference is that macros have a value whereas a symbol simply is defined or is not defined. Furthermore, following the definition of a macro, any occurrence of the macro in the pascal source will be replaced with the value of the macro (much like the macro support in the C preprocessor). If macro support is required, the `-Sm` command line switch must be used to switch it on, or the directive must be inserted:

```
{$MACRO ON}
```

otherwise macros will be regarded as a symbol.

Defining a macro in a program is done in the same way as defining a symbol; in a `{$define}` preprocessor statement[1]:

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

If the compiler encounters `ident` in the rest of the source file, it will be replaced immediately by `expr`. This replacement works recursive, meaning that when the compiler expanded one macro, it will look at the resulting expression again to see if another replacement can be made. This means that care should be taken when using macros, because an infinite loop can occur in this manner.

Here are two examples which illustrate the use of macros:

```
{$define sum:=a:=a+b;}
...
sum          { will be expanded to 'a:=a+b;'
               remark the absence of the semicolon}
...
{$define b:=100}
sum          { Will be expanded recursively to a:=a+100; }
...
```

The previous example could go wrong:

```
{$define sum:=a:=a+b;}
...
sum          { will be expanded to 'a:=a+b;'
               remark the absence of the semicolon}
...
{$define b=sum} { DON'T do this !!!}
sum          { Will be infinitely recursively expanded... }
...
```

On my system, the last example results in a heap error, causing the compiler to exit with a run-time error 203.

Remark: Macros defined in the interface part of a unit are not available outside that unit! They can just be used as a notational convenience, or in conditional compiles.

By default the compiler predefines three macros, containing the version number, the release number and the patch number. They are listed in table (2.1).

Table 2.1: Predefined macros

Symbol	Contains
`FPC_FULLVERSION`	An integer version number of the compiler.
`FPC_VERSION`	The version number of the compiler.
`FPC_RELEASE`	The release number of the compiler.
`FPC_PATCH`	The patch number of the compiler.

The `FPC_FULLVERSION` macro contains a version number which always uses 2 digits for the `RELEASE` and `PATCH` version numbers. This means that version 2.3.1 will result in `FPC_FULLVERSION=20301`. This number makes it easier to determine minimum versions.

[1] In compiler versions older than 0.9.8, the assignment operator for a macros wasn't `:=` but `=`

Remark: Don't forget that macro support isn't on by default. It must be turned on with the `-Sm` command line switch or using the `{$MACRO ON}` directive.

2.3 Compile time variables

In MacPas mode, compile time variables can be defined. They are distinct from symbols in that they have a value, and they are distinct from macros, in that they cannot be used to replace portions of the source text with their value. Their behaviour are compatible with compile time variables found in popular pascal compilers for Macintosh.

A compile time variable is defined like this:

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

The expression is a so-called compile time expression, which is evaluated once, at the point where the `{$SETC }` directve is encountered in the source. The resulting value is then assigned to the compile time variable.

A second `{$SETC }` directive for the same variable overwrites the previous value.

Contrary to macros and symbols, compile time variables defined in the Interface part of a unit are exported. This means their value will be available in units which uses the unit in which the variable is defined. This requires that both units are compiled in macpas mode.

The big difference between macros and compile time variables is that the former is a pure text substitution mechanism (much like in C), where the latter resemble normal programming language variables, but they are available to the compiler only.

In mode MacPas, compile time variables are always enabled.

2.4 Compile time expressions

2.4.1 Definition

Except for the regular Turbo Pascal constructs for conditional compilation, the Free Pascal compiler also supports a stronger conditional compile mechanism: The `{$IF}` construct, which can be used to evaluate compile-time expressions.

The prototype of this construct is as follows:

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

The content of an expression is restricted to what can be evaluated at compile-time:

- Constants (strings, numbers)
- Macros
- Compile time variables (mode MacPas only)
- Pascal constant expression (mode Delphi only)

The symbols are replaced with their value. For macros recursive substitution might occur.

The following boolean operators are available:

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

The IN operator tests for presence of a compile-time variable in a set.

The following functions are also available:

TRUE Defined in MacPas mode only, it evaluates to True. In other modes, 1 can be used.

FALSE Defined in MacPas mode only, it evaluates to False. In other modes, 0 can be used.

DEFINED(sym) will evaluate to `TRUE` if a compile time symbol is defined. In MacPas mode, the parentheses are optional, i.e.

```
{$IF DEFINED(MySym)}
```

is equivalent to

```
{$IF DEFINED MySym}
```

UNDEFINED sym will evaluate to `TRUE` if a compile time symbol is *not* defined, and `FALSE` otherwise (mode MacPas only).

OPTION(opt) evaluates to `TRUE` if a compiler option is set (mode MacPas only). It is equivalent to the `{$IFOPT }` directive.

SIZEOF(passym) Evaluates to the size of a pascal type, variable or constant.

DECLARED(passym) Evaluates to `TRUE` if the pascal symbol is declared at this point in the sources, or `FALSE` if it is not yet defined.

In expressions, the following rules are used for evaluation:

- If all parts of the expression can be evaluated as booleans (with 1 and 0 representing `TRUE` and `FALSE`), the expression is evaluated using booleans.
- If all parts of the expression can be evaluated as nuumbers, then the expression is evaluated using numbers.
- In all other cases, the expression is evaluated using strings.

If the complete expression evaluates to `'0'`, then it is considered `False` and rejected. Otherwise it is considered `True` and accepted. This may have unexpected consequences:

```
{$if 0}
```

will evaluate to `False` and be rejected, while

```
{$if 00}
```

will evaluate to `True`.

2.4.2 Usage

The basic usage of compile time expressions is as follows:

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

If `expr` evaluates to `TRUE`, then `CompileTheseLines` will be included in the source.

Like in regular pascal, it is possible to use `{$ELSE }`:

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

If `expr` evaluates to `True`, `CompileTheseLines` will be compiled. Otherwise, `BetterCompileTheseLines` will be compiled.

Additionally, it is possible to use `{$ELSEIF}`

```
{$IF  expr}
  // ...
{$ELSEIF  expr}
  // ...
{$ELSEIF  expr}
  // ...
{$ELSE}
  // ...
{$ENDIF}
```

In addition to the above constructs, which are also supported by Delphi, the above is completely equivalent to the following construct in MacPas mode:

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

that is, `IFC` corresponds to `IF`, `ELIFC` corresponds to `ELSEIF`, `ELSEC` is equivalent with `ELSE` and `ENDC` is the equivalent of `ENDIF`. Additionally, `IFEND` is equivalent to `ENDIF`:

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

In MacPas mode it is possible to mix these constructs.

The following example shows some of the possibilities:

```
{$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}

{$IF NOT UNDEFINED FPC}
// Detect FPC stuff when compiling on MAC.
{$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 * Now have to follow at least 2 error messages: *}
{$info ************************************************}

{$if not(0}
{$endif}

{$if not(<}
{$endif}

end.
```

As you can see from the example, this construct isn't useful when used with normal symbols, only if you use macros, which are explained in section 2.2, page 55. They can be very useful. When trying this example, you must switch on macro support, with the `-Sm` command line switch.

The following example works only in MacPas mode:

```
{$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}
  ... new mac code
  {$ELSEC}
  ... new other code
  {$ENDC}
{$ELSEC}
... old code
{$ENDC}
```

2.5 Messages

Free Pascal lets you define normal, warning and error messages in your code. Messages can be used to display useful information, such as copyright notices, a list of symbols that your code reacts on etc.

Warnings can be used if you think some part of your code is still buggy, or if you think that a certain combination of symbols isn't useful.

Error messages can be useful if you need a certain symbol to be defined, to warn that a certain variable isn't defined, or when the compiler version isn't suitable for your code.

The compiler treats these messages as if they were generated by the compiler. This means that if you haven't turned on warning messages, the warning will not be displayed. Errors are always displayed,

and the compiler stops if 50 errors have occurred. After a fatal error, the compiler stops at once.

For messages, the syntax is as follows:

```
{$Message Message text}
```

or

```
{$Info Message text}
```

For notes:

```
{$Note Message text}
```

For warnings:

```
{$Warning Warning Message text}
```

For hints:

```
{$Hint Warning Message text}
```

For errors:

```
{$Error  Error Message text}
```

Lastly, for fatal errors:

```
{$Fatal  Error Message text}
```

or

```
{$Stop  Error Message text}
```

The difference between `$Error` and `$FatalError` or `$Stop` messages is that when the compiler encounters an error, it still continues to compile. With a fatal error, the compiler stops.

Remark: You cannot use the '`}`' character in your message, since this will be treated as the closing brace of the message.

As an example, the following piece of code will generate an error when neither of the symbols `RequiredVar1` or `RequiredVar2` are defined:

```
{$IFNDEF RequiredVar1}
{$IFNDEF RequiredVar2}
{$Error One of Requiredvar1 or Requiredvar2 must be defined}
{$ENDIF}
{$ENDIF}
```

But the compiler will continue to compile. It will not, however, generate a unit file or a program (since an error occurred).

Chapter 3

Using Assembly language

Free Pascal supports inserting assembler statements in between Pascal code. The mechanism for this is the same as under Turbo Pascal and Delphi. There are, however some substantial differences, as will be explained in the following sections.

3.1 Using assembler in the sources

There are essentially 2 ways to embed assembly code in the pascal source. The first one is the simplest, by using an asm block:

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

Everything between the `asm` and `end` block is inserted as assembler in the generated code. Depending on the assembler reader mode, the compiler performs substitution of certain names with their addresses.

The second way is implementing a complete procedure or function in assembler. This is done by adding a `assembler` modifier to the function or procedure header:

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

It's still possible to declare variables in an assembler procedure:

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

```
  movl %edi,saveedi
end;
```

The compiler will reserve space on the stack for these variables, it inserts some commands for this.

Note that the assembler name of an assembler function will still be 'mangled' by the compiler, i.e. the label for this function will not be the name of the function as declared. To change this, an `Alias` modifier can be used:

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

To make the function available in assembler code outside the current unit, the `Public` modifier can be added:

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

3.2 Intel 80x86 Inline assembler

3.2.1 Intel syntax

Free Pascal supports Intel syntax for the Intel family of Ix86 processors in its `asm` blocks.

The Intel syntax in your `asm` block is converted to AT&T syntax by the compiler, after which it is inserted in the compiled source. The supported assembler constructs are a subset of the normal assembly syntax. In what follows we specify what constructs are not supported in Free Pascal, but which exist in Turbo Pascal:

- The `TBYTE` qualifier is not supported.
- The `&` identifier override is not supported.
- The `HIGH` operator is not supported.
- The `LOW` operator is not supported.
- The `OFFSET` and `SEG` operators are not supported. Use `LEA` and the various `Lxx` instructions instead.
- Expressions with constant strings are not allowed.
- Access to record fields via parenthesis is not allowed
- Typecasts with normal pascal types are not allowed, only recognized assembler typecasts are allowed. Example:

```
mov al, byte ptr MyWord       -- allowed,
mov al, byte(MyWord)          -- allowed,
mov al, shortint(MyWord)      -- not allowed.
```

- Pascal type typecasts on constants are not allowed. Example:

  ```
  const s= 10; const t = 32767;
  ```

  in Turbo Pascal:

  ```
  mov al, byte(s)            -- useless typecast.
  mov al, byte(t)            -- syntax error!
  ```

  In this parser, either of those cases will give out a syntax error.

- Constant references expressions with constants only are not allowed (in all cases they do not work in protected mode, e.g. under LINUX i386). Examples:

  ```
  mov al,byte ptr ['c']      -- not allowed.
  mov  al,byte ptr [100h]    -- not allowed.
  ```

  (This is due to the limitation of the GNU Assembler).

- Brackets within brackets are not allowed

- Expressions with segment overrides fully in brackets are currently not supported, but they can easily be implemented in BuildReference if requested. Example:

  ```
  mov al,[ds:bx]     -- not allowed
  ```

  use instead:

  ```
  mov al,ds:[bx]
  ```

- Possible allowed indexing are as follows:

  - `Sreg:[REG+REG*SCALING+/-disp]`
  - `SReg:[REG+/-disp]`
  - `SReg:[REG]`
  - `SReg:[REG+REG+/-disp]`
  - `SReg:[REG+REG*SCALING]`

  Where `Sreg` is optional and specifies the segment override. *Notes:*

  1. The order of terms is important contrary to Turbo Pascal.
  2. The Scaling value must be a value, and not an identifier to a symbol. Examples:

     ```
     const myscale = 1;
     ...
     mov al,byte ptr [esi+ebx*myscale] -- not allowed.
     ```

     use:

     ```
     mov al, byte ptr [esi+ebx*1]
     ```

- Possible variable identifier syntax is as follows: (Id = Variable or typed constant identifier.)

  1. `ID`
  2. `[ID]`
  3. `[ID+expr]`

4. `ID[expr]`

Possible fields are as follow:

1. `ID.subfield.subfield ...`
2. `[ref].ID.subfield.subfield ...`
3. `[ref].typename.subfield ...`

- Local labels: Contrary to Turbo Pascal, local labels, must at least contain one character after the local symbol indicator. Example:

```
@:                  -- not allowed
```

use instead:

```
@1:                 -- allowed
```

- Contrary to Turbo Pascal, local references cannot be used as references, only as displacements. Example:

```
lds si,@mylabel   -- not allowed
```

- Contrary to Turbo Pascal, `SEGCS`, `SEGDS`, `SEGES` and `SEGSS` segment overrides are presently not supported. (This is a planned addition though).
- Contrary to Turbo Pascal where memory sizes specifiers can be practically anywhere, the Free Pascal Intel inline assembler requires memory size specifiers to be outside the brackets. Example:

```
mov al,[byte ptr myvar]    -- not allowed.
```

use:

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

- Base and Index registers must be 32-bit registers. (limitation of the GNU Assembler).
- `XLAT` is equivalent to `XLATB`.
- Only Single and Double FPU opcodes are supported.
- Floating point opcodes are currently not supported (except those which involve only floating point registers).

The Intel inline assembler supports the following macros:

@Result represents the function result return value.

Self represents the object method pointer in methods.

3.2.2 AT&T Syntax

In earlier versions, Free Pascal used only the GNU `as` assembler to generate its object files for the Intel x86 processors. Only after some time, an internal assembler was created, which wrote directly to an object file.

Since the GNU assembler uses AT&T assembly syntax, the code you write should use the same syntax. The differences between AT&T and Intel syntax as used in Turbo Pascal are summarized in the following:

- The opcode names include the size of the operand. In general, one can say that the AT&T opcode name is the Intel opcode name, suffixed with a '`l`', '`w`' or '`b`' for, respectively, longint (32 bit), word (16 bit) and byte (8 bit) memory or register references. As an example, the Intel construct '`mov al bl` is equivalent to the AT&T style '`movb %bl,%al`' instruction.
- AT&T immediate operands are designated with '$', while Intel syntax doesn't use a prefix for immediate operands. Thus the Intel construct '`mov ax, 2`' becomes '`movb $2, %al`' in AT&T syntax.
- AT&T register names are preceded by a '`%`' sign. They are undelimited in Intel syntax.
- AT&T indicates absolute jump/call operands with '`*`', Intel syntax doesn't delimit these addresses.
- The order of the source and destination operands are switched. AT&T syntax uses '`Source, Dest`', while Intel syntax features '`Dest, Source`'. Thus the Intel construct '`add eax, 4`' transforms to '`addl $4, %eax`' in the AT&T dialect.
- Immediate long jumps are prefixed with the '`l`' prefix. Thus the Intel '`call/jmp section:offset`' is transformed to '`lcall/ljmp $section,$offset`'. Similarly, the far return is '`lret`', instead of the Intel '`ret far`'.
- Memory references are specified differently in AT&T and Intel assembly. The Intel indirect memory reference

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

  is written in AT&T syntax as:

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

  Where `Base` and `Index` are optional 32-bit base and index registers, and `Scale` is used to multiply `Index`. It can take the values 1,2,4 and 8. The `Section` is used to specify an optional section register for the memory operand.

More information about the AT&T syntax can be found in the `as` manual, although the following differences with normal AT&T assembly must be taken into account:

- Only the following directives are presently supported:

  .byte

  .word

  .long

  .ascii

  .asciz

  .globl

- The following directives are recognized but are not supported:

.align

.lcomm

Eventually they will be supported.

- Directives are case sensitive, other identifiers are not case sensitive.
- Contrary to gas, local labels/symbols *must* start with `.L`.
- The not operator `'!'` is not supported.
- String expressions in operands are not supported.
- CBTW,CWTL,CWTD and CLTD are not supported, use the normal Intel equivalents instead.
- Constant expressions which represent memory references are not allowed, even though constant immediate value expressions are supported. Examples:

```
const myid = 10;
...
movl $myid,%eax       -- allowed
movl myid(%esi),%eax  -- not allowed.
```

- When the `.globl` directive is found, the symbol immediatly following it is made public and is immediately emitted. Therefore label names with this name will be ignored.
- Only Single and Double FPU opcodes are supported.

The AT&T inline assembler supports the following macros:

__RESULT represents the function result return value.

__SELF represents the object method pointer in methods.

__OLDEBP represents the old base pointer in recusrive routines.

3.3 Motorola 680x0 Inline assembler

The inline assembler reader for the Motorola 680x0 family of processors uses the Motorola Assembler syntax (q.v). A few differences do exist:

- Local labels start with the @ character, such as

```
@MyLabel:
```

- The `XDEF` directive in an assembler block will make the symbol available publicly with the specified name (this name is case sensitive)
- The `DB`, `DW`, `DD` directives can only be used to declare constants which will be stored in the code segment.
- The `Align` directive is not supported.
- Arithmetic operations on constant expression use the same operands as the intel version, e.g, `AND`, `XOR` ...
- Segment directives are not supported

- Only 68000 and a subset of 68020 opcodes are currently supported.

The inline assembler supports the following macros:

@Result represents the function result return value.

Self represents the object method pointer in methods.

3.4 Signaling changed registers

When the compiler uses variables, it sometimes stores them, or the result of some calculations, in the processor registers. If you insert assembler code in your program that modifies the processor registers, then this may interfere with the compiler's idea about the registers. To avoid this problem, Free Pascal allows you to tell the compiler which registers have changed in an `asm` block. The compiler will then save and reload these registers if it was using them. Telling the compiler which registers have changed is done by specifying a set of register names behind an assembly block, as follows:

```
asm
  ...
end ['R1', ... ,'Rn'];
```

Here `R1` to `Rn` are the names of the registers you modify in your assembly code.

As an example:

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

This example tells the compiler that the `EAX` register was modified.

For assembler routines, i.e., routines that are written completely in assembler, the ABI of the processor & platform must be respected, i.e. the routine itself must know what registers to save and what not, but it can tell the compiler using the same method what registers were changed or not. The compiler will save specified registers to the stack on entry and restore them on routine exit.

The only thing the compiler normally does, is create a minimal stack frame if needed (e.g. when variables are declared). All the rest is up to the programmer.

Chapter 4

Generated code

As noted in the previous chapter, older Free Pascal compilers relied on the GNU assembler to make object files. The compiler only generated an assembly language file which was then passed on to the assembler. In the following two sections, we discuss what is generated when you compile a unit or a program.

4.1 Units

When you compile a unit, the Free Pascal compiler generates 2 files:

1. A unit description file.
2. An assembly language file.

The assembly language file contains the actual source code for the statements in your unit, and the necessary memory allocations for any variables you use in your unit. This file is converted by the assembler to an object file (with extension .o) which can then be linked to other units and your program, to form an executable.

By default, the assembly file is removed after it has been compiled. Only in the case of the `-s` command line option, the assembly file will be left on disk, so the assembler can be called later. You can disable the erasing of the assembler file with the `-a` switch.

The unit file contains all the information the compiler needs to use the unit:

1. Other used units, both in interface and implementation.
2. Types and variables from the interface section of the unit.
3. Function declarations from the interface section of the unit.
4. Some debugging information, when compiled with debugging.

The detailed contents and structure of this file are described in the first appendix. You can examine a unit description file using the ppudump program, which shows the contents of the file.

If you want to distribute a unit without source code, you must provide both the unit description file and the object file.

You can also provide a C header file to go with the object file. In that case, your unit can be used by someone who wishes to write his programs in C. However, you must make this header file yourself since the Free Pascal compiler doesn't make one for you.

4.2 Programs

When you compile a program, the compiler produces again 2 files:

1. An assembly language file containing the statements of your program, and memory allocations for all used variables.
2. A linker response file. This file contains a list of object files the linker must link together.

The link response file is, by default, removed from the disk. Only when you specify the `-s` command line option or when linking fails, then the file is left on the disk. It is named link.res.

The assembly language file is converted to an object file by the assembler, and then linked together with the rest of the units and a program header, to form your final program.

The program header file is a small assembly program which provides the entry point for the program. This is where the execution of your program starts, so it depends on the operating system, because operating systems pass parameters to executables in wildly different ways.

By default, its name is prt0.o, and the source file resides in prt0.as or some variant of this name: Which file is actually used depends on the system, and on LINUX systems, whether the C library is used or not.

It usually resides where the system unit source for your system resides. Its main function is to save the environment and command line arguments and set up the stack. Then it calls the main program.

Chapter 5

Intel MMX support

5.1 What is it about?

Free Pascal supports the new MMX (Multi-Media extensions) instructions of Intel processors. The idea of MMX is to process multiple data with one instruction, for example the processor can add simultaneously 4 words. To implement this efficiently, the Pascal language needs to be extended. So Free Pascal allows to add for example two `array[0..3] of word`, if MMX support is switched on. The operation is done by the `MMX` unit and allows people without assembler knowledge to take advantage of the MMX extensions.

Here is an example:

```
uses
   MMX;   { include some predefined data types }

const
   { tmmxword = array[0..3] of word;, declared by unit 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 is exported from unit mmx }
     begin
{$mmx+}   { turn mmx on }
        w3:=w1+w2;
{$mmx-}
     end
   else
     begin
        for i:=0 to 3 do
          w3[i]:=w1[i]+w2[i];
     end;
end.
```

5.2 Saturation support

One important point of MMX is the support of saturated operations. If a operation would cause an overflow, the value stays at the highest or lowest possible value for the data type: If you use byte values you get normally 250+12=6. This is very annoying when doing color manipulations or changing audio samples, when you have to do a word add and check if the value is greater than 255. The solution is saturation: 250+12 gives 255. Saturated operations are supported by the `MMX` unit. If you want to use them, you have simple turn the switch saturation on: `$saturation+`

Here is an example:

```
Program SaturationDemo;
{
  example for saturation, scales data (for example audio)
  with 1.5 with rounding to negative infinity
}
uses mmx;

var
   audio1 : tmmxword;
   i: smallint;

const
   helpdata1 : tmmxword = ($c000,$c000,$c000,$c000);
   helpdata2 : tmmxword = ($8000,$8000,$8000,$8000);

begin
   { audio1 contains four 16 bit audio samples }
{$mmx+}
   { convert it to $8000 is defined as zero, multiply data with 0.75 }
   audio1:=(audio1+helpdata2)*(helpdata1);
{$saturation+}
   { avoid overflows (all values>$ffff becomes $ffff) }
   audio1:=(audio1+helpdata2)-helpdata2;
{$saturation-}
   { now mupltily with 2 and change to integer }
   for i:=0 to 3 do
     audio1[i] := audio1[i] shl 1;
   audio1:=audio1-helpdata2;
{$mmx-}
end.
```

5.3 Restrictions of MMX support

In the beginning of 1997 the MMX instructions were introduced in the Pentium processors, so multitasking systems wouldn't save the newly introduced MMX registers. To work around that problem, Intel mapped the MMX registers to the FPU register.

The consequence is that you can't mix MMX and floating point operations. After using MMX operations and before using floating point operations, you have to call the routine `EMMS` of the `MMX` unit. This routine restores the FPU registers.

Careful: The compiler doesn't warn if you mix floating point and MMX operations, so be careful.

The MMX instructions are optimized for multimedia operations (what else?). So it isn't possible

to perform all possible operations: some operations give a type mismatch, see section 5.4 for the supported MMX operations.

An important restriction is that MMX operations aren't range or overflow checked, even when you turn range and overflow checking on. This is due to the nature of MMX operations.

The `MMX` unit must always be used when doing MMX operations because the exit code of this unit clears the MMX unit. If it wouldn't do that, other program will crash. A consequence of this is that you can't use MMX operations in the exit code of your units or programs, since they would interfere with the exit code of the `MMX` unit. The compiler can't check this, so you are responsible for this!

5.4 Supported MMX operations

The following operations are supported in the compiler when MMX extensions are enabled:

- addition (+)
- subtraction (-)
- multiplication(*)
- logical exclusive or (`xor`)
- logical and (`and`)
- logical or (`or`)
- sign change (-)

5.5 Optimizing MMX support

Here are some helpful hints to get optimal performance:

- The `EMMS` call takes a lot of time, so try to seperate floating point and MMX operations.
- Use MMX only in low level routines because the compiler saves all used MMX registers when calling a subroutine.
- The NOT-operator isn't supported natively by MMX, so the compiler has to generate a workaround and this operation is inefficient.
- Simple assignements of floating point numbers don't access floating point registers, so you need no call to the `EMMS` procedure. Only when doing arithmetic, you need to call the `EMMS` procedure.

Chapter 6

Code issues

This chapter gives detailed information on the generated code by Free Pascal. It can be useful to write external object files which will be linked to Free Pascal created code blocks.

6.1 Register Conventions

The compiler has different register conventions, depending on the target processor used; some of the registers have specific uses during the code generation. The following section describes the generic names of the registers on a platform per platform basis. It also indicates what registers are used as scratch registers, and which can be freely used in assembler blocks.

6.1.1 accumulator register

The accumulator register is at least a 32-bit integer hardware register, and is used to return results of function calls which return integral values.

6.1.2 accumulator 64-bit register

The accumulator 64-bit register is used in 32-bit environments and is defined as the group of registers which will be used when returning 64-bit integral results in function calls. This is a register pair.

6.1.3 float result register

This register is used for returning floating point values from functions.

6.1.4 self register

The self register contains a pointer to the actual object or class. This register gives access to the data of the object or class, and the VMT pointer of that object or class.

6.1.5 frame pointer register

The frame pointer register is used to access parameters in subroutines, as well as to access local variables. References to the pushed prameters and local variables are constructed using the frame

pointer. [1].

6.1.6 stack pointer register

The stack pointer is used to give the address of the stack area, where the local variables and parameters to subroutines are stored.

6.1.7 scratch registers

Scratch registers are those which can be used in assembler blocks, or in external object files without requiring any saving before usage.

6.1.8 Processor mapping of registers

This indicates what registers are used for what purposes on each of the processors supported by Free Pascal. It also indicates which registers can be used as scratch registers.

Intel 80x86 version

Table 6.1: Intel 80x86 Register table

Generic register name	CPU Register name
accumulator	EAX
accumulator (64-bit) high / low	EDX:EAX
float result	FP(0)
self	ESI
frame pointer	EBP
stack pointer	ESP
scratch regs.	N/A

Motorola 680x0 version

Table 6.2: Motorola 680x0 Register table

Generic register name	CPU Register name
accumulator	D0[2]
accumulator (64-bit) high / low	D0:D1
float result	FP0[3]
self	A5
frame pointer	A6
stack pointer	A7
scratch regs.	D0, D1, A0, A1, FP0, FP1

[1] The frame pointer is not available on all platforms

6.2 Name mangling

Contrary to most C compilers and assemblers, all labels generated to pascal variables and routines have mangled names[4]. This is done so that the compiler can do stronger type checking when parsing the Pascal code. It also permits function and procedure overloading.

6.2.1 Mangled names for data blocks

The rules for mangled names for variables and typed constants are as follows:

- All variable names are converted to upper case
- Variables in the main program or private to a unit have an underscore (_) prepended to their names.
- Typed constants in the main program have a TC__ prepended to their names
- Public variables in a unit have their unit name prepended to them : U_UNITNAME_
- Public and private typed constants in a unit have their unit name prepended to them :TC__UNITNAME$$

Examples:

```
unit testvars;

interface

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

implementation
const
 privatetypedconst : integer = 1;
var
 privatevar : integer;

end.
```

Will result in the following assembler code for the GNU assembler :

```
.file "testvars.pas"

.text

.data
# [6] publictypedconst : integer = 0;
.globl TC__TESTVARS$$_PUBLICTYPEDCONST
TC__TESTVARS$$_PUBLICTYPEDCONST:
```

[2]For compatibility with some C compilers, when the function result is a pointer and is declared with the cdecl convention, the result is also stored in the A0 register

[3]On simulated FPU's the result is returned in D0

[4]This can be avoided by using the `alias` or `cdecl` modifiers

```
.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 Mangled names for code blocks

The rules for mangled names for routines are as follows:

- All routine names are converted to upper case.
- Routines in a unit have their unit name prepended to them : _UNITNAME$$_
- All Routines in the main program have a _ prepended to them.
- All parameters in a routine are mangled using the type of the parameter (in uppercase) prepended by the $ character. This is done in left to right order for each parameter of the routine.
- Objects and classes use special mangling : The class type or object type is given in the mangled name; The mangled name is as follows: _$$_TYPEDECL_$$ optionally preceded by mangled name of the unit and finishing with the method name.

The following constructs

```
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.
```

will result in the following assembler file for the Intel 80x86 target:

```
.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 Modifying the mangled names

To make the symbols externally accessible, it is possible to give nicknames to mangled names, or to change the mangled name directly. Two modifiers can be used:

public: For a function that has a `public` modifier, the mangled name will be the name *exactly* as it is declared.

alias: The `alias` modifier can be used to assign a second assembler label to your function. This label has the same name as the alias name you declared. This doesn't modify the calling conventions of the function. In other words, the `alias` modifier allows you to specify another name (a nickname) for your function or procedure.

The prototype for an aliased function or procedure is as follows:

```
Procedure AliasedProc; alias : 'AliasName';
```

The procedure `AliasedProc` will also be known as `AliasName`. Take care, the name you specify is case sensitive (as C is).

Furthermore, the `exports` section of a library is also used to declare the names that will be exported by the shared library. The names in the exports section are case-sensitive (while the actual declaration of the routine is not). For more information on the creating shared libraries, chapter 12, page 132.

6.3 Calling mechanism

By default, the calling mechanism the compiler uses is `register`, that is, the compiler will try to pass as much parameters as posible by storing them in a free register. Not all registers are used, because some registers have a special meaning, but this depends on the CPU.

Function results are returned in the accumulator (first register), if they fit in the register. Method calls (from either objects or clases) have an additional invisible parameter which is `self`.

When the procedure or function exits, it clears the stack.

Other calling methods are available for linking with external object files and libraries, these are summarized in table (6.3). The first column lists the modifier you specify for a procedure declaration. The second one lists the order the paramaters are pushed on the stack. The third column specifies who is responsible for cleaning the stack: the caller or the called function. The alignment column indicates the alignment of the parameters sent to the stack area.

Subroutines will modify a number of registers (the volatile registers). The list of registers that are modified is highly dependent on the processor, calling convention and ABI of the target platform.

Table 6.3: Calling mechanisms in Free Pascal

Modifier	Pushing order	Stack cleaned by	alignment
<none>	Left-to-right	Callee	default
register	Left-to-right	Callee	default
cdecl	Right-to-left	Caller	GCC alignment
interrupt	Right-to-left	Callee	default
pascal	Left-to-right	Callee	default
safecall	Right-to-left	Callee	default
stdcall	Right-to-left	Callee	GCC alignment
oldfpccall	Right-to-left	Callee	default

Note that the `oldfpccall` calling convention equals the default calling convention on processors other than 32-bit Intel 386 or higher.

More about this can be found in chapter 7, page 86 on linking. Information on GCC registers saved, GCC stack alignment and general stack alignment on an operating system basis is beyond the scope of this manual.

As of version 2.0 (actually, in 1.9.x somewhere) , the `register` modifier is the default calling convention, prior to that, it was the `oldfpccall` convention.

The default calling convention, i.e., the calling convention used when none is specified explicitly, can be set using the `{$calling }` directive, section 1.2.7, page 16. The default calling convention for the current platform can be specified with

```
{$CALLING DEFAULT}
```

Remark: The `popstack` modifier is no longer supported as of version 2.0, but has been renamed to `oldfpccall`. The `saveregisters` modifier can no longer be used.

6.4 Nested procedure and functions

When a routine is declared within the scope of a procedure or function, it is said to be nested. In this case, an additional invisible parameter is passed to the nested routine. This additional parameter is the frame pointer address of the parent routine. This permits the nested routine to access the local variables and parameters of the calling routine.

The resulting stack frame after the entry code of a simple nested procedure has been executed is shown in table (6.4).

Table 6.4: Stack frame when calling a nested procedure (32-bit processors)

Offset from frame pointer	What is stored
+x	parameters
+8	Frame pointer of parent routine
+4	Return address
+0	Saved frame pointer

6.5 Constructor and Destructor calls

Constructor and destructors have special invisible parameters which are passed to them. These invisible parameters are used internally to instantiate the objects and classes.

6.5.1 objects

The actual invisible declaration of an object constructor is as follows:

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

Where `_vmt` is a pointer to the virtual method table for this object. This value is nil if a constructor is called within the object instance (such as calling an ihnerited constructor).

_self is either nil if the instance must be allocated dynamically (object is declared as pointer), or the address of the object instance if the object is declared as a normal object (stored in the data area) or if the object instance has already been allocated.

The allocated instance (if allocated via new) (self) is returned in the accumulator.

The declaration of a destructor is as follows:

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

Where _vmt is a pointer to the virtual method table for this object. This value is nil if a destructor is called within the object instance (such as calling an ihnerited constructor), or if the object instance is a variable and not a pointer.

_self is the address of the object instance.

6.5.2 classes

The actual invisible declaration of a class constructoir is as follows:

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

_vmt is either nil if called from the instance or if calling an inherited constructor, otherwise it points to the address of the virtual method table.

Where flag is zero if the constructor is called within the object instance or with an instance qualifier otherwise this flag is set to one.

The allocated instance (self) is returned in the accumulator.

The declaration of a destructor is as follows:

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

_self is the address of the object instance.

flag is zero if the destructor is called within the object instance or with an instance qualifier otherwise this flag is set to one.

6.6 Entry and exit code

Each Pascal procedure and function begins and ends with standard epilogue and prologue code.

6.6.1 Intel 80x86 standard routine prologue / epilogue

Standard entry code for procedures and functions is as follows on the 80x86 architecture:

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

The generated exit sequence for procedure and functions looks as follows:

```
leave
ret  $xx
```

Where xx is the total size of the pushed parameters.

To have more information on function return values take a look at section 6.1, page 76.

6.6.2 Motorola 680x0 standard routine prologue / epilogue

Standard entry code for procedures and functions is as follows on the 680x0 architecture:

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

The generated exit sequence for procedure and functions looks as follows (in the default processor mode):

```
unlk    a6
rtd     #xx
```

Where `xx` is the total size of the pushed parameters.

To have more information on function return values take a look at section 6.1, page 76.

6.7 Parameter passing

When a function or procedure is called, then the following is done by the compiler:

1. If there are any parameters to be passed to the procedure, they are stored in well-known registers, and if there are more parameters than free registers, they are pushed from left to right on the stack.
2. If a function is called that returns a variable of type `String`, `Set`, `Record`, `Object` or `Array`, then an address to store the function result in, is also passed to the procedure.
3. If the called procedure or function is an object method, then the pointer to `self` is passed to the procedure.
4. If the procedure or function is nested in another function or procedure, then the frame pointer of the parent procedure is passed to the stack.
5. The return address is pushed on the stack (This is done automatically by the instruction which calls the subroutine).

The resulting stack frame upon entering looks as in table (6.5).

Table 6.5: Stack frame when calling a procedure (32-bit model)

Offset	What is stored	Optional?
+x	extra parameters	Yes
+12	function result	Yes
+8	self	Yes
+4	Return address	No
+0	Frame pointer of parent procedure	Yes

6.7.1 Parameter alignment

Each parameter passed to a routine is guaranteed to decrement the stack pointer by a certain minimum amount. This behavior varies from one operating system to another. For example, passing a byte as

a value parameter to a routine could either decrement the stack pointer by 1, 2, 4 or even 8 bytes depending on the target operating system and processor.

For example, on FREEBSD, all parameters passed to a routine guarantee a minimal stack decrease of four bytes per parameter, even if the parameter actually takes less then 4 bytes to store on the stack (such as passing a byte value parameter to the stack).

6.8 Stack limitations

Certain processors have limitations on the size of the parameters and local variables in routines. This is shown in table (6.6).

Table 6.6: Maximum limits for processors

Processor	Parameters	Local variables
Intel 80x86 (all)	64K	No limit
Motorola 68020 (default)	32K	No limit
Motorola 68000	32K	32K

Furthermore, the m68k compiler, in `68000` mode, limits the size of data elements to 32K (arrays, records, objects, etc.). This restriction does not exist in `68020` mode.

Chapter 7

Linking issues

When you only use Pascal code, and Pascal units, then you will not see much of the part that the linker plays in creating your executable. The linker is only called when you compile a program. When compiling units, the linker isn't invoked.

However, there are times that linking to C libraries, or to external object files created by other compilers, may be necessary. The Free Pascal compiler can generate calls to a C function, and can generate functions that can be called from C (exported functions).

7.1 Using external code and variables

In general, there are 3 things you must do to use a function that resides in an external library or object file:

1. You must make a pascal declaration of the function or procedure you want to use.
2. You must declare the correct calling convention to use.
3. You must tell the compiler where the function resides, i.e. in what object file or what library, so the compiler can link the necessary code in.

The same holds for variables. To access a variable that resides in an external object file, you must declare it, and tell the compiler where to find it. The following sections attempt to explain how to do this.

7.1.1 Declaring external functions or procedures

The first step in using external code blocks is declaring the function you want to use. Free Pascal supports Delphi syntax, i.e. you must use the `external` directive. The `external` directive replaces, in effect, the code block of the function.

The external directive doesn't specify a calling convention; it just tells the compiler that the code for a procedure or function resides in an external code block. A calling convention modifier should be declared if the external code blocks does not have the same calling conventions as Free Pascal. For more information on the calling conventions section 6.3, page 81.

There exist four variants of the external directive:

1. A simple external declaration:

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

The `external` directive tells the compiler that the function resides in an external block of code. You can use this together with the `{$L}` or `{$LinkLib}` directives to link to a function or procedure in a library or external object file. Object files are looked for in the object search path (set by `-Fo`) and libraries are searched for in the linker path (set by `-Fl`).

2. You can give the `external` directive a library name as an argument:

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

This tells the compiler that the procedure resides in a library with name `'Name'`. This method is equivalent to the following:

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

3. The `external` can also be used with two arguments:

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

This has the same meaning as the previous declaration, only the compiler will use the name `'OtherProcName'` when linking to the library. This can be used to give different names to procedures and functions in an external library. The name of the routine is case-sensitive and should match exactly the name of the routine in the object file.

This method is equivalent to the following code:

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

Procedure ProcName (Args : TPRocArgs);

begin
  OtherProcName (Args);
end;
```

4. Lastly, onder WINDOWS and OS/2, there is a fourth possibility to specify an external function: In .DLL files, functions also have a unique number (their index). It is possible to refer to these fuctions using their index:

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

This tells the compiler that the procedure `ProcName` resides in a dynamic link library, with index `SomeIndex`.

Remark: Note that this is *only* available under WINDOWS and OS/2.

7.1.2 Declaring external variables

Some libaries or code blocks have variables which they export. You can access these variables much in the same way as external functions. To access an external variable, you declare it as follows:

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

The effect of this declaration is twofold:

1. No space is allocated for this variable.
2. The name of the variable used in the assembler code is `varname`. This is a case sensitive name, so you must be careful.

The variable will be accessible with its declared name, i.e. `MyVar` in this case.

A second possibility is the declaration:

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

The effect of this declaration is twofold as in the previous case:

1. The `external` modifier ensures that no space is allocated for this variable.
2. The `cvar` modifier tells the compiler that the name of the variable used in the assembler code is exactly as specified in the declaration. This is a case sensitive name, so you must be careful.

The first possibility allows you to change the name of the external variable for internal use.

As an example, let's look at the following C file (in extvar.c):

```
/*
Declare a variable, allocate storage
*/
int extvar = 12;
```

And the following program (in extdemo.pp):

```
Program ExtDemo;

{$L extvar.o}

Var { Case sensitive declaration !! }
    extvar : longint; cvar;external;
    I : longint; external name 'extvar';
begin
  { Extvar can be used case insensitive !! }
  Writeln ('Variable ''extvar'' has value: ',ExtVar);
  Writeln ('Variable ''I''      has value: ',i);
end.
```

Compiling the C file, and the pascal program:

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

Will produce a program extdemo which will print

```
Variable 'extvar' has value: 12
Variable 'I'      has value: 12
```

on your screen.

7.1.3 Declaring the calling convention modifier

To make sure that all parameters are correctly passed to the external routines, you should declare it with the correct calling convention modifier. When linking with code blocks compiled with standard C compilers (such as GCC), the `cdecl` modifier should be used so as to indicate that the external routine uses C type calling conventions. For more information on the supported calling conventions, see section 6.3, page 81.

As might be expected, external variable declarations do not require any calling convention modifier.

7.1.4 Declaring the external object code

Linking to an object file

Having declared the external function or variable that resides in an object file, you can use it as if it was defined in your own program or unit. To produce an executable, you must still link the object file in. This can be done with the `{$L file.o}` directive.

This will cause the linker to link in the object file file.o. On most systems, this filename is case sensitive. The object file is first searched in the current directory, and then the directories specified by the `-Fo` command line.

You cannot specify libraries in this way, it is for object files only.

Here we present an example. Consider that you have some assembly routine which uses the C calling convention that calculates the nth Fibonacci number:

```
.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
```

Then you can call this function with the following Pascal Program:

```
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.
```

With just two commands, this can be made into a program:

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

This example supposes that you have your assembler routine in fib.s, and your Pascal program in fibo.pp.

Linking to a library

To link your program to a library, the procedure depends on how you declared the external procedure.

In case you used the following syntax to declare your procedure:

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

You don't need to take additional steps to link your file in, the compiler will do all that is needed for you. On WINDOWS it will link to name.dll, on LINUX and most UNIX'es your program will be linked to library libname, which can be a static or dynamic library.

In case you used

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

You still need to explicity link to the library. This can be done in 2 ways:

1. You can tell the compiler in the source file what library to link to using the `{$LinkLib 'Name'}` directive:

   ```
   {$LinkLib 'gpm'}
   ```

   This will link to the gpm library. On UNIX systems (such as LINUX), you must not specify the extension or 'lib' prefix of the library. The compiler takes care of that. On other systems (such as WINDOWS), you need to specify the full name.

2. You can also tell the compiler on the command line to link in a library: The `-k` option can be used for that. For example

   ```
   ppc386 -k'-lgpm' myprog.pp
   ```

   Is equivalent to the above method, and tells the linker to link to the gpm library.

As an example, consider the following program:

```
program printlength;

{$linklib c} { Case sensitive }

{ Declaration for the standard C function strlen }
Function strlen (P : pchar) : longint; cdecl;external;

begin
  Writeln (strlen('Programming is easy !'));
end.
```

This program can be compiled with:

```
ppc386  prlen.pp
```

Supposing, of course, that the program source resides in prlen.pp.

To use functions in C that have a variable number of arguments, you must compile your unit or program in `objfpc` mode or `Delphi` mode, and use the `Array of const` argument, as in the following example:

```
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.
```

The output of this program looks like this:

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

As an alternative, the program can be constructed as follows:

```
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.
```

The `varargs` modifier signals the compiler that the function allows a variable number of arguments (the ellipsis notation in C).

7.2 Making libraries

Free Pascal supports making shared or static libraries in a straightforward and easy manner. If you want to make static libraries for other Free Pascal programmers, you just need to provide a command line switch. To make shared libraries, refer to the chapter 12, page 132. If you want C programmers to be able to use your code as well, you will need to adapt your code a little. This process is described first.

7.2.1 Exporting functions

When exporting functions from a library, there are 2 things you must take in account:

1. Calling conventions.
2. Naming scheme.

The calling conventions are controlled by the modifiers `cdecl`, `popstack`, `pascal`, `safecall`, `stdcall` and `register`. See section 6.3, page 81 for more information on the different kinds of calling scheme.

The naming conventions can be controlled by 2 modifiers in the case of static libraries:

- `cdecl`
- `alias`

For more information on how these different modifiers change the name mangling of the routine section 6.2, page 78.

Remark: If in your unit, you use functions that are in other units, or system functions, then the C program will need to link in the object files from these units too.

7.2.2 Exporting variables

Similarly as when you export functions, you can export variables. When exporting variables, one should only consider the names of the variables. To declare a variable that should be used by a C program, one declares it with the `cvar` modifier:

```
Var MyVar : MyTpe; cvar;
```

This will tell the compiler that the assembler name of the variable (the one which is used by C programs) should be exactly as specified in the declaration, i.e., case sensitive.

It is not allowed to declare multiple variables as `cvar` in one statement, i.e. the following code will produce an error:

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

7.2.3 Compiling libraries

To create shared libraries one should use the `library` keyword in the main compilation file (the project file). For more information on creating shared libraries, chapter 12, page 132.

The .o object files that the compiler writes when it compiles a unit, are regular object files as they are produced by a C compiler. They can be combined using the `ar` and `ranlib` tools into a static library. However, for various reasons, this is a bad idea.

- The code will be full of references to compiler internal routines in the RTL. These routines are not present in a C library.
- The initialization sections will not be called by a C program.
- The thread vars will not be allocated (or initialized).
- Resource strings will not be initialized.
- Each library thus made will attempt to initialize the RTL.

To remedy these (and other) problems requires intimate knowledge of the inner workings of the compiler and RTL, and it is therefor a bad idea to attempt the use of static libraries with Free Pascal.

7.2.4 Unit searching strategy

When you compile a unit, the compiler will by default always look for unit files.

To be able to differentiate between units that have been compiled as static or dynamic libraries, there are 2 switches:

-XD: This will define the symbol `FPC_LINK_DYNAMIC`

-XS: This will define the symbol `FPC_LINK_STATIC`

Definition of one symbol will automatically undefine the other.

These two switches can be used in conjunction with the configuration file fpc.cfg. The existence of one of these symbols can be used to decide which unit search path to set. For example, on 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
```

With such a configuration file, the compiler will look for its units in different directories, depending on whether `-XD` or `-XS` is used.

7.3 Using smart linking

You can compile your units using smart linking. When you use smartlinking, the compiler creates a series of code blocks that are as small as possible, i.e. a code block will contain only the code for one procedure or function.

When you compile a program that uses a smart-linked unit, the compiler will only link in the code that you actually need, and will leave out all other code. This will result in a smaller binary, which is loaded in memory faster, thus speeding up execution.

To enable smartlinking, one can give the smartlink option on the command line: `-Cx`, or one can put the `{$SMARTLINK ON}` directive in the unit file:

```
Unit Testunit

{SMARTLINK ON}
Interface
...
```

Smartlinking will slow down the compilation process, especially for large units.

When a unit foo.pp is smartlinked, the name of the codefile is changed to libfoo.a.

Technically speaking, the compiler makes small assembler files for each procedure and function in the unit, as well as for all global defined variables (whether they're in the interface section or not). It then assembles all these small files, and uses ar to collect the resulting object files in one archive.

Smartlinking and the creation of shared (or dynamic) libraries are mutually exclusive, that is, if you turn on smartlinking, then the creation of shared libraries is turned of. The creation of static libraries is still possible. The reason for this is that it has little sense in making a smartlinked dynamical library. The whole shared library is loaded into memory anyway by the dynamic linker (or the operating system), so there would be no gain in size by making it smartlinked.

Chapter 8

Memory issues

8.1 The memory model.

The Free Pascal compiler issues 32-bit or 64-bit code. This has several consequences:

- You need a 32-bit or 64-bit processor to run the generated code.
- You don't need to bother with segment selectors. Memory can be addressed using a single 32-bit (on 32-bit processors) or 64-bit (on 64-bit processors with 64-bit addressing) pointer. The amount of memory is limited only by the available amount of (virtual) memory on your machine.
- The structures you define are unlimited in size. Arrays can be as long as you want. You can request memory blocks from any size.

8.2 Data formats

This section gives information on the storage space occupied by the different possible types in Free Pascal. Information on internal alignment will also be given.

8.2.1 Integer types

The storage size of the default integer types are given in Reference Guide. In the case of user defined-types, the storage space occupied depends on the bounds of the type:

- If both bounds are within range -128..127, the variable is stored as a shortint (signed 8-bit quantity).
- If both bounds are within the range 0..255, the variable is stored as a byte (unsigned 8-bit quantity).
- If both bounds are within the range -32768..32767, the variable is stored as a smallint (signed 16-bit quantity).
- If both bounds are within the range 0..65535, the variable is stored as a word (unsigned 16-bit quantity)
- If both bounds are within the range 0..4294967295, the variable is stored as a longword (unsigned 32-bit quantity).

- Otherwise the variable is stored as a longint (signed 32-bit quantity).

8.2.2 Char types

A `char`, or a subrange of the char type, is stored as a byte. A WideChar is stored as a word, i.e. 2 bytes.

8.2.3 Boolean types

The `Boolean` type is stored as a byte and can take a value of `true` or `false`.

A `ByteBool` is stored as a byte, a `WordBool` type is stored as a word, and a `longbool` is stored as a longint.

8.2.4 Enumeration types

By default all enumerations are stored as a longword (4 bytes), which is equivalent to specifying the `{$Z4}`, `{$PACKENUM 4}` or `{$PACKENUM DEFAULT}` switches.

This default behavior can be changed by compiler switches, and by the compiler mode.

In the `tp` compiler mode, or while the `{$Z1}` or `{$PACKENUM 1}` switches are in effect, the storage space used is shown in table (8.1).

Table 8.1: Enumeration storage for `tp` mode

# Of Elements in Enum.	Storage space used
0..255	byte (1 byte)
256..65535	word (2 bytes)
> 65535	longword (4 bytes)

When the `{$Z2}` or `{$PACKENUM 2}` switches are in effect, the value is stored in 2 bytes (a word), if the enumeration has less or equal than 65535 elements. If there are more elements, the enumeration value is stored as a 4 byte value (a longword).

8.2.5 Floating point types

Floating point type sizes and mapping vary from one processor to another. Except for the Intel 80x86 architecture, the `extended` type maps to the IEEE double type if a hardware floating point coprocessor is present.

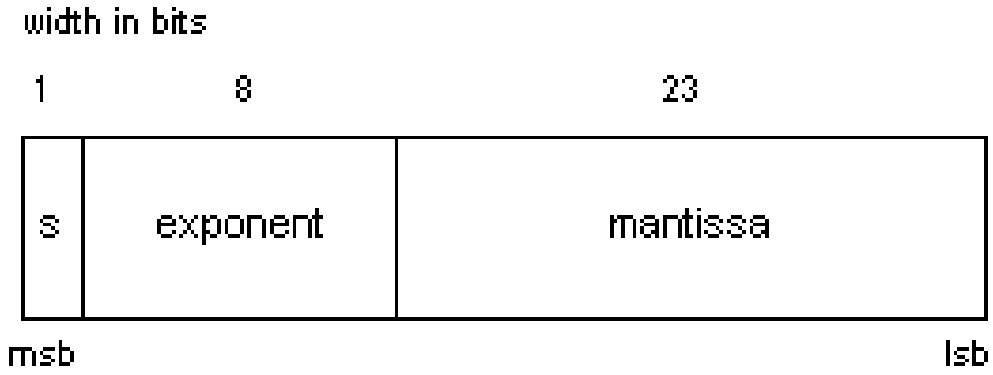
Floating point types have a storage binary format divided into three distinct fields : the mantissa, the exponent and the sign bit which stores the sign of the floating point value.

Single

The `single` type occupies 4 bytes of storage space, and its memory structure is the same as the IEEE-754 single type. This type is the only type which is guaranteed to be available on all platforms (either emulated via software or directly via hardware).

The memory format of the `single` format looks like what is shown in figure (8.1).

Figure 8.1: The single format

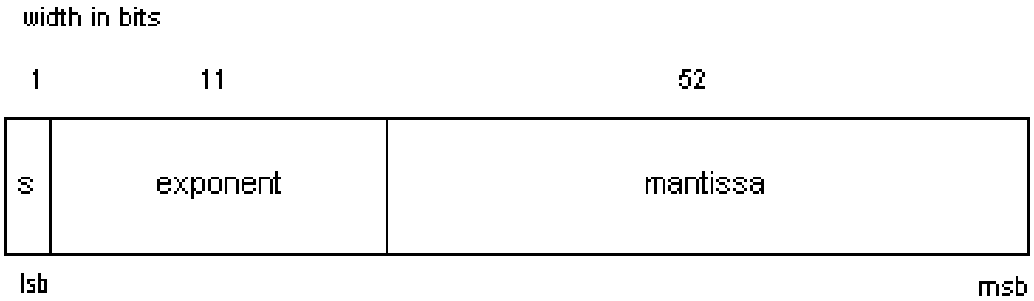


Double

The `double` type occupies 8 bytes of storage space, and its memory structure is the same as the IEEE-754 double type.

The memory format of the `double` format looks like like what is shown in figure (8.2).

Figure 8.2: The double format



On processors which do not support co-processor operations (and which have the {$E+} switch), the `double` type does not exist.

Extended

For Intel 80x86 processors, the `extended` type has takes up 10 bytes of memory space. For more information on the extended type consult the Intel Programmer's reference.

For all other processors which support floating point operations, the `extended` type is a nickname for the type which supports the most precision, this is usually the `double` type. On processors which do not support co-processor operations (and which have the {$E+} switch), the `extended` type usually maps to the `single` type.

Comp

For Intel 80x86 processors, the `comp` type contains a 63-bit integral value, and a sign bit (in the MSB position). The `comp` type uses 8 bytes of storage space.

On other processors, the `comp` type is not supported.

Real

Contrary to Turbo Pascal, where the `real` type had a special internal format, under Free Pascal the `real` type simply maps to one of the other real types. It maps to the `double` type on processors which support floating point operations, while it maps to the `single` type on processors which do not support floating point operations in hardware. See table (8.2) for more information on this.

Table 8.2: Processor mapping of real type

Processor	Real type mapping
Intel 80x86	`double`
Motorola 680x0 (with {$E-} switch)	`double`
Motorola 680x0 (with {$E+} switch)	`single`

8.2.6 Pointer types

A `pointer` type is stored as a longword (unsigned 32-bit value) on 32-bit processors, and is stored as a 64-bit unsigned value[1] on 64-bit processors.

8.2.7 String types

Ansistring types

The ansistring is a dynamically allocated string which has no length limitation. When the string is no longer being referenced (its reference count reaches zero), its memory is automatically freed.

If the ansistring is a constant, then its reference count will be equal to -1, indicating that it should never be freed. The structure in memory for an ansistring is shown in table (8.3).

Table 8.3: AnsiString memory structure (32-bit model)

Offset	Contains
-8	Longint with reference count.
-4	Longint with actual string size.
0	Actual array of `char`, null-terminated.

Shortstring types

A shortstring occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string. The following bytes contain the actual characters (of type `char`) of the string. The maximum size of a short string is the length byte followed by 255 characters.

[1] this is actually the `qword` type, which is not supported in Free Pascal v1.0

Widestring types

A widestring is allocated on the heap, much like an ansistring. Unlike the ansistring, a widestring takes 2 bytes per character, and is terminated with a double null.

8.2.8 Set types

A set is stored as an array of bits, where each bit indicates if the element is in the set or excluded from the set. The maximum number of elements in a set is 256.

If a set has less than 32 elements, it is coded as an unsigned 32-bit value. Otherwise it is coded as an array of 8 unsigned 32-bit values (longwords), and hence has a size of 256 bytes.

The longword number of a specific element `E` is given by :

```
LongwordNumber = (E div 32);
```

and the bit number within that 32-bit value is given by:

```
BitNumber = (E mod 32);
```

8.2.9 Static array types

A static array is stored as a contiguous sequence of variables of the components of the array. The components with the lowest indexes are stored first in memory. No alignment is done between each element of the array. A multi-dimensional array is stored with the rightmost dimension increasing first.

8.2.10 Dynamic array types

A dynamic array is stored as a pointer to a block of memory on the heap. The memory on the heap is a contiguous sequence of variables of the components of the array, just as for a static array. The reference count and memory size are stored in memory just before the actual start of the array, at a negative offset relative to the address the pointer refers to. It should not be used.

8.2.11 Record types

Each field of a record is stored in a continguous sequence of variables, where the first field is stored at the lowest address in memory. In case of variant fields in a record, each variant starts at the same address in memory. Fields of record are usually aligned, unless the `packed` directive is specified when declaring the record type.

For more information on field alignment, consult section 8.3.2, page 103.

8.2.12 Object types

Objects are stored in memory just as ordinary records with an extra field: a pointer to the Virtual Method Table (VMT). This field is stored first, and all fields in the object are stored in the order they are declared (with possible alignment of field addresses, unless the object was declared as being `packed`).

The VMT is initialized by the call to the object's `Constructor` method. If the `new` operator was used to call the constructor, the data fields of the object will be stored in heap memory, otherwise they will directly be stored in the data section of the final executable.

If an object doesn't have virtual methods, no pointer to a VMT is inserted.

The memory allocated looks as in table (8.4).

Table 8.4: Object memory layout (32/64-bit model)

Offset		What
32-bit	64-bit	
+0	+0	Pointer to VMT (optional).
+4	+8	Data. All fields in the order they've been declared.
...		

The Virtual Method Table (VMT) for each object type consists of 2 check fields (containing the size of the data), a pointer to the object's ancestor's VMT (`Nil` if there is no ancestor), and then the pointers to all virtual methods. The VMT layout is illustrated in table (8.5). The VMT is constructed by the compiler.

Table 8.5: Object Virtual Method Table memory layout (32/64-bit model)

Offset		What
32-bit	64-bit	
+0	+0	Size of object type data
+4	+8	Minus the size of object type data. Enables determining of valid VMT pointers.
+8	+16	Pointer to ancestor VMT, `Nil` if no ancestor available.
+12	+24	Pointers to the virtual methods.
...		

8.2.13 Class types

Just like objects, classes are stored in memory just as ordinary records with an extra field: a pointer to the Virtual Method Table (VMT). This field is stored first, and all fields in the class are stored in the order they are declared.

Contrary to objects, all data fields of a class are always stored in heap memory.

The memory allocated looks as in table (8.6).

Table 8.6: Class memory layout (32/64-bit model)

Offset		What
32-bit	64-bit	
+0	+0	Pointer to VMT.
+4	+8	Data. All fields in the order they've been declared.
...		

The Virtual Method Table (VMT) of each class consists of several fields, which are used for runtime type information. The VMT layout is illustrated in table (8.7). The VMT is constructed by the compiler.

Table 8.7: Class Virtual Method Table memory layout (32/64-bit model)

Offset 32-bit	64-bit	What
+0	+0	Size of object type data
+4	+8	Minus the size of object type data. Enables determining of valid VMT pointers.
+8	+16	Pointer to ancestor VMT, `Nil` if no ancestor available.
+12	+24	Pointer to the class name (stored as a `shortstring`).
+16	+32	Pointer to the dynamic method table (using `message` with integers).
+20	+40	Pointer to the method definition table.
+24	+48	Pointer to the field definition table.
+28	+56	Pointer to type information table.
+32	+64	Pointer to instance initialization table.
+36	+72	Pointer to the auto table.
+40	+80	Pointer to the interface table.
+44	+88	Pointer to the dynamic method table (using `message` with strings).
+48	+96	Pointer to the `Destroy` destructor.
+52	+104	Pointer to the `NewInstance` method.
+56	+112	Pointer to the `FreeInstance` method.
+60	+120	Pointer to the `SafeCallException` method.
+64	+128	Pointer to the `DefaultHandler` method.
+68	+136	Pointer to the `AfterConstruction` method.
+72	+144	Pointer to the `BeforeDestruction` method.
+76	+152	Pointer to the `DefaultHandlerStr` method.
+80	+160	Pointer to the `Dispatch` method.
+84	+168	Pointer to the `DispatchStr` method.
+88	+176	Pointer to the `Equals` method.
+92	+184	Pointer to the `GetHashCode` method.
+96	+192	Pointer to the `ToString` method.
+100	+200	Pointers to other virtual methods.
...		

See the file rtl/inc/objpash.inc for the most current VMT information.

8.2.14 File types

File types are represented as records. Typed files and untyped files are represented as a fixed record:

```
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;
```

Text files are described using the following record:

```
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 The handle field returns the file handle (if the file is opened), as returned by the operating system.

mode The mode field can take one of several values. When it is `fmclosed`, then the file is closed, and the `handle` field is invalid. When the value is equal to `fminput`, it indicates that the file is opened for read only access. `fmoutput` indicates write only access, and the `fminout` indicates read-write access to the file.

name The `name` field is a null terminated character string representing the name of the file.

userdata The `userdata` field is never used by Free Pascal file handling routines, and can be used for special purposes by software developers.

8.2.15 Procedural types

A procedural type is stored as a generic pointer, which stores the address of the routine.

A procedural type to a normal procedure or function is stored as a generic pointer, which stores the address of the entry point of the routine.

In the case of a method procedural type, the storage consists of two pointers, the first being a pointer to the entry point of the method, and the second one being a pointer to `self` (the object instance).

8.3 Data alignment

8.3.1 Typed constants and variable alignment

All static data (variables and typed constants) which are greater than a byte are usually aligned on a multiple of two boundary. This alignment applies only to the start address of the variables, and not the alignment of fields within structures or objects for example. For more information on structured alignment, section 8.3.2, page 103. The alignment is similar across the different target processors.

Table 8.8: Data alignment

Data size (bytes)	Alignment (small size)	Alignment (fast)
1	1	1
2-3	2	2
4-7	2	4
8+	2	4

The alignment columns indicates the address alignment of the variable, i.e the start address of the variable will be aligned on that boundary. The small size alignment is valid when the code generated should be optimized for size (`-Og` compiler option) and not speed, otherwise the fast alignment is used to align the data (this is the default).

8.3.2 Structured types alignment

By default all elements in a structure are aligned to a 2 byte boundary, unless the `$PACKRECORDS` directive or `packed` modifier is used to align the data in another way. For example a `record` or `object` having a 1 byte element, will have its size rounded up to 2, so the size of the structure will actually be 2 bytes.

8.4 The heap

The heap is used to store all dynamic variables, and to store class instances. The interface to the heap is the same as in Turbo Pascal and Delphi although the effects are maybe not the same. The heap is thread-safe, so allocating memory from various threads is not a problem.

8.4.1 Heap allocation strategy

The heap is a memory structure which is organized as a stack. The heap bottom is stored in the variable `HeapOrg`. Initially the heap pointer (`HeapPtr`) points to the bottom of the heap. When a variable is allocated on the heap, `HeapPtr` is incremented by the size of the allocated memory block. This has the effect of stacking dynamic variables on top of each other.

Each time a block is allocated, its size is normalized to have a granularity of 16 (or 32 on 64 bit systems) bytes.

When `Dispose` or `FreeMem` is called to dispose of a memory block which is not on the top of the heap, the heap becomes fragmented. The deallocation routines also add the freed blocks to the `freelist` which is actually a linked list of free blocks. Furthermore, if the deallocated block was less then 8K in size, the free list cache is also updated.

The free list cache is actually a cache of free heap blocks which have specific lengths (the adjusted block size divided by 16 gives the index into the free list cache table). It is faster to access then searching through the entire `freelist`.

The format of an entry in the `freelist` is as follows:

```
PFreeRecord = ^TFreeRecord;
TFreeRecord = record
  Size : longint;
  Next : PFreeRecord;
  Prev : PFreeRecord;
```

```
end;
```

The `Next` field points to the next free block, while the `Prev` field points to the previous free block.

The algorithm for allocating memory is as follows:

1. The size of the block to allocate is adjusted to a 16 (or 32) byte granularity.
2. The cached free list is searched to find a free block of the specified size or bigger size, if so it is allocated and the routine exits.
3. The `freelist` is searched to find a free block of the specified size or of bigger size, if so it is allocated and the routine exits.
4. If not found in the `freelist` the heap is grown to allocate the specified memory, and the routine exits.
5. If the heap cannot be grown internally anymore, the runtime library generates a runtime error 203.

8.4.2 The heap grows

The heap allocates memory from the operating system on an as-needed basis.

OS memory is requested in blocks: It first tries to increase memory in a 64Kb chunk if the size to allocate is less than 64Kb, or 256Kb or 1024K otherwise. If this fails, it tries to increase the heap by the amount you requested from the heap.

If the attempt to reserve OS memory fails, the value returned depends on the value of the `ReturnNilIfGrowHeapFails` global variable. This is summarized in table (8.9).

Table 8.9: ReturnNilIfGrowHeapFails value

ReturnNilIfGrowHeapFails value	Default memory manager action
FALSE	(The default) Runtime error 203 generated
TRUE	`GetMem`, `ReallocMem` and `New` returns `nil`

`ReturnNilIfGrowHeapFails` can be set to change the behavior of the default memory manager error handler.

8.4.3 Debugging the heap

Free Pascal provides a unit that allows you to trace allocation and deallocation of heap memory: heaptrc.

If you specify the `-gh` switch on the command line, or if you include `heaptrc` as the first unit in your uses clause, the memory manager will trace what is allocated and deallocated, and on exit of your program, a summary will be sent to standard output.

More information on using the `heaptrc` mechanism can be found in the User's Guide and Unit Reference.

8.4.4 Writing your own memory manager

Free Pascal allows you to write and use your own memory manager. The standard functions `GetMem`, `FreeMem`, `ReallocMem` etc. use a special record in the system unit to do the actual memory management. The system unit initializes this record with the system unit's own memory manager, but you can read and set this record using the `GetMemoryManager` and `SetMemoryManager` calls:

```
procedure GetMemoryManager(var MemMgr: TMemoryManager);
procedure SetMemoryManager(const MemMgr: TMemoryManager);
```

the `TMemoryManager` record is defined as follows:

```
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;
```

As you can see, the elements of this record are mostly procedural variables. The system unit does nothing but call these various variables when you allocate or deallocate memory.

Each of these fields corresponds to the corresponding call in the system unit. We'll describe each one of them:

NeedLock This boolean indicates whether the memory manager needs a lock: if the memory manager itself is not thread-safe, then this can be set to `True` and the Memory routines will use a lock for all memory routines. If this field is set to `False`, no lock will be used.

Getmem This function allocates a new block on the heap. The block should be `Size` bytes long. The return value is a pointer to the newly allocated block.

Freemem should release a previously allocated block. The pointer `P` points to a previously allocated block. The Memory manager should implement a mechanism to determine what the size of the memory block is. [2] The return value is optional, and can be used to return the size of the freed memory.

FreememSize This function should release the memory pointed to by `P`. The argument `Size` is the expected size of the memory block pointed to by P. This should be disregarded, but can be used to check the behaviour of the program.

AllocMem Is the same as getmem, only the allocated memory should be filled with zeroes before the call returns.

ReAllocMem Should allocate a memory block `Size` bytes large, and should fill it with the contents of the memory block pointed to by `P`, truncating this to the new size of needed. After that, the memory pointed to by P may be deallocated. The return value is a pointer to the new memory block. Note that `P` may be `Nil`, in which case the behaviour is equivalent to `GetMem`.

[2] By storing its size at a negative offset for instance.

MemSize should return the size of the memory block `P`. This function may return zero if the memory manager does not allow to determine this information.

InitThread This routine is called when a new thread is started: it should initialize the heap structures for the current thread (if any).

DoneThread This routine is called when a thread is ended: it should clean up any heap structures for the current thread.

RelocateHeap Relocates the heap - this is only for thread-local heaps.

GetHeapStatus should return a `THeapStatus` record with the status of the memory manager. This record should be filled with Delphi-compliant values.

GetHeapStatus should return a `TFPCHeapStatus` record with the status of the memory manager. This record should be filled with FPC-Compliant values.

To implement your own memory manager, it is sufficient to construct such a record and to issue a call to `SetMemoryManager`.

To avoid conflicts with the system memory manager, setting the memory manager should happen as soon as possible in the initialization of your program, i.e. before any call to `getmem` is processed.

This means in practice that the unit implementing the memory manager should be the first in the `uses` clause of your program or library, since it will then be initialized before all other units - except the system unit itself, of course.

This also means that it is not possible to use the heaptrc unit in combination with a custom memory manager, since the heaptrc unit uses the system memory manager to do all its allocation. Putting the heaptrc unit after the unit implementing the memory manager would overwrite the memory manager record installed by the custom memory manager, and vice versa.

The following unit shows a straightforward implementation of a custom memory manager using the memory manager of the `C` library. It is distributed as a package with 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 Using DOS memory under the Go32 extender

Because Free Pascal for DOS is a 32 bit compiler, and uses a DOS extender, accessing DOS memory isn't trivial. What follows is an attempt to an explanation of how to access and use DOS or real mode memory[3].

In *Proteced Mode*, memory is accessed through *Selectors* and *Offsets*. You can think of Selectors as the protected mode equivalents of segments.

In Free Pascal, a pointer is an offset into the `DS` selector, which points to the Data of your program.

To access the (real mode) DOS memory, somehow you need a selector that points to the DOS memory. The go32 unit provides you with such a selector: The `DosMemSelector` variable, as it is conveniently called.

You can also allocate memory in DOS's memory space, using the `global_dos_alloc` function of the go32 unit. This function will allocate memory in a place where DOS sees it.

As an example, here is a function that returns memory in real mode DOS and returns a selector:offset pair for it.

```
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;
```

(You need to free this memory using the `global_dos_free` function.)

You can access any place in memory using a selector. You can get a selector using the function:

[3]Thanks for the explanation to Thomas Schatzl (E-mail: `tom_at_work@geocities.com`)

```
function allocate_ldt_descriptors(count : word) : word;
```

and then let this selector point to the physical memory you want using the function

```
function set_segment_base_address(d : word;s : longint) : boolean;
```

Its length can be set using the function:

```
function set_segment_limit(d : word;s : longint) : boolean;
```

You can manipulate the memory pointed to by the selector using the functions of the GO32 unit. For instance with the `seg_fillchar` function. After using the selector, you must free it again using the function:

```
function free_ldt_descriptor(d : word) : boolean;
```

More information on all this can be found in the Unit Reference, the chapter on the go32 unit.

8.6 When porting Turbo Pascal code

The fact that 16-bit code is no longer used, means that some of the older Turbo Pascal constructs and functions are obsolete. The following is a list of functions which shouldn't be used anymore:

Seg() : Returned the segment of a memory address. Since segments have no more meaning, zero is returned in the Free Pascal run-time library implementation of `Seg`.

Ofs() : Returned the offset of a memory address. Since segments have no more meaning, the complete address is returned in the Free Pascal implementation of this function. This has as a consequence that the return type is `longint` or `int64` instead of `Word`.

Cseg(), Dseg() : Returned, respectively, the code and data segments of your program. This returns zero in the Free Pascal implementation of the system unit, since both code and data are in the same memory space.

Ptr : Accepted a segment and offset from an address, and would return a pointer to this address. This has been changed in the run-time library, it now simply returns the offset.

memw and mem : These arrays gave access to the DOS memory. Free Pascal supports them on the go32v2 platform, they are mapped into DOS memory space. You need the go32 unit for this. On other platforms, they are *not* supported

You shouldn't use these functions, since they are very non-portable, they're specific to DOS and the 80x86 processor. The Free Pascal compiler is designed to be portable to other platforms, so you should keep your code as portable as possible, and not system specific. That is, unless you're writing some driver units, of course.

8.7 Memavail and Maxavail

The old Turbo Pascal functions `MemAvail` and `MaxAvail` functions are no longer available in Free Pascal as of version 2.0. The reason for this incompatibility is below:

On modern operating systems, [4] the idea of "Available Free Memory" is not valid for an application. The reasons are:

[4] The DOS extender GO32V2 falls under this definition of "modern" because it can use paged memory and run in multi-tasked environments

1. One processor cycle after an application asked the OS how much memory is free, another application may have allocated everything.
2. It is not clear what "free memory" means: does it include swap memory, does it include disk cache memory (the disk cache can grow and shrink on modern OS'es), does it include memory allocated to other applications but which can be swapped out, etc.

Therefore, programs using `MemAvail` and `MaxAvail` functions should be rewritten so they no longer use these functions, because it does not make sense anymore on modern OS'es. There are 3 possibilities:

1. Use exceptions to catch out-of-memory errors.
2. Set the global variable "ReturnNilIfGrowHeapFails" to `True` and check after each allocation whether the pointer is different from `Nil`.
3. Don't care and declare a dummy function called `MaxAvail` which always returns `High(LongInt)` (or some other constant).

Chapter 9

Resource strings

9.1 Introduction

Resource strings primarily exist to make internationalization of applications easier, by introducing a language construct that provides a uniform way of handling constant strings.

Most applications communicate with the user through some messages on the graphical screen or console. Storing these messages in special constants allows storing them in a uniform way in separate files, which can be used for translation. A programmers interface exists to manipulate the actual values of the constant strings at runtime, and a utility tool comes with the Free Pascal compiler to convert the resource string files to whatever format is wanted by the programmer. Both these things are discussed in the following sections.

9.2 The resource string file

When a unit is compiled that contains a `resourcestring` section, the compiler does 2 things:

1. It generates a table that contains the value of the strings as it is declared in the sources.
2. It generates a *resource string file* that contains the names of all strings, together with their declared values.

This approach has 2 advantages: first of all, the value of the string is always present in the program. If the programmer doesn't care to translate the strings, the default values are always present in the binary. This also avoids having to provide a file containing the strings. Secondly, having all strings together in a compiler generated file ensures that all strings are together (you can have multiple resourcestring sections in 1 unit or program) and having this file in a fixed format, allows the programmer to choose his way of internationalization.

For each unit that is compiled and that contains a resourcestring section, the compiler generates a file that has the name of the unit, and an extension .rst. The format of this file is as follows:

1. An empty line.
2. A line starting with a hash sign (`#`) and the hash value of the string, preceded by the text `hash value =`.
3. A third line, containing the name of the resource string in the format `unitname.constantname`, all lowercase, followed by an equal sign, and the string value, in a format equal to the pascal

representation of this string. The line may be continued on the next line, in that case it reads as a pascal string expression with a plus sign in it.

4. Another empty line.

If the unit contains no `resourcestring` section, no file is generated.

For example, the following unit:

```
unit rsdemo;

{$mode delphi}
{$H+}

interface

resourcestring

  First = 'First';
  Second = 'A Second very long string that should cover more than 1 line';


implementation

end.
```

Will result in the following resource string file:

```
# hash value = 5048740
rsdemo.first='First'


# hash value = 171989989
rsdemo.second='A Second very long string that should cover more than 1 li'+
'ne'
```

The hash value is calculated with the function `Hash`. It is present in the objpas unit. The value is the same value that the GNU gettext mechanism uses. It is in no way unique, and can only be used to speed up searches.

The rstconv utility that comes with the Free Pascal compiler allows manipulation of these resource string files. At the moment, it can only be used to make a .po file that can be fed to the GNU msgfmt program. If someone wishes to have another format (Win32 resource files spring to mind), one can enhance the rstconv program so it can generate other types of files as well. GNU gettext was chosen because it is available on all platforms, and is already widely used in the `Unix` and free software community. Since the Free Pascal team doesn't want to restrict the use of resource strings, the .rst format was chosen to provide a neutral method, not restricted to any tool.

If you use resource strings in your units, and you want people to be able to translate the strings, you must provide the resource string file. Currently, there is no way to extract them from the unit file, though this is in principle possible. It is not required to do this, the program can be compiled without it, but then the translation of the strings isn't possible.

9.3 Updating the string tables

Having compiled a program with resourcestrings is not enough to internationalize your program. At run-time, the program must initialize the string tables with the correct values for the language that the user selected. By default no such initialization is performed. All strings are initialized with their declared values.

The objpas unit provides the mechanism to correctly initialize the string tables. There is no need to include this unit in a `uses` clause, since it is automatically loaded when a program or unit is compiled in `Delphi` or `objfpc` mode. Since one of these mode is required to use resource strings, the unit is always loaded when needed anyway.

The resource strings are stored in tables, one per unit, and one for the program, if it contains a `resourcestring` section as well. Each resourcestring is stored with its name, hash value, default value, and the current value, all as `AnsiStrings`.

The objpas unit offers methods to retrieve the number of resourcestring tables, the number of strings per table, and the above information for each string. It also offers a method to set the current value of the strings.

Here are the declarations of all the functions:

```
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);
```

Two other function exist, for convenience only:

```
Function Hash(S: AnsiString): longint;
Procedure ResetResourceTables;
```

Here is a short explanation of what each function does. A more detailed explanation of the functions can be found in the Reference Guide.

ResourceStringTableCount returns the number of resource string tables in the program.

ResourceStringCount returns the number of resource string entries in a given table (tables are denoted by a zero-based index).

GetResourceStringName returns the name of a resource string in a resource table. This is the name of the unit, a dot (.) and the name of the string constant, all in lowercase. The strings are denoted by index, also zero-based.

GetResourceStringHash returns the hash value of a resource string, as calculated by the compiler with the `Hash` function.

GetResourceStringDefaultValue returns the default value of a resource string, i.e. the value that appears in the resource string declaration, and that is stored in the binary.

GetResourceStringCurrentValue returns the current value of a resource string, i.e. the value set by the initialization (the default value), or the value set by some previous internationalization routine.

SetResourceStringValue sets the current value of a resource string. This function must be called to initialize all strings.

SetResourceStrings giving this function a callback will cause the calback to be called for all resource strings, one by one, and set the value of the string to the return value of the callback.

Two other functions exist, for convenience only:

Hash can be used to calculate the hash value of a string. The hash value stored in the tables is the result of this function, applied on the default value. That value is calculated at compile time by the compiler: having the value available can speed up translation operations.

ResetResourceTables will reset all the resource strings to their default values. It is called by the initialization code of the objpas unit.

Given some `Translate` function, the following code would initialize all resource strings:

```
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;
```

Other methods are of course possible, and the `Translate` function can be implemented in a variety of ways.

9.4 GNU gettext

The unit gettext provides a way to internationalize an application with the GNU gettext utilities. This unit is supplied with the Free Component Library (FCL). it can be used as follows:

for a given application, the following steps must be followed:

1. Collect all resource string files and concatenate them together.
2. Invoke the rstconv program with the file resulting out of step 1, resulting in a single .po file containing all resource strings of the program.
3. Translate the .po file of step 2 in all required languages.
4. Run the msgfmt formatting program on all the .po files, resulting in a set of .mo files, which can be distributed with your application.
5. Call the gettext unit's `TranslateResourceStrings` method, giving it a template for the location of the .mo files, e.g. as in

```
TranslateResourcestrings('intl/restest.%s.mo');
```

the `%s` specifier will be replaced by the contents of the `LANG` environment variable. This call should happen at program startup.

An example program exists in the FCL-base sources, in the fcl-base/tests directory.

9.5 Caveat

In principle it is possible to translate all resource strings at any time in a running program. However, this change is not communicated to other strings; its change is noticed only when a constant string is being used.

Consider the following example:

```
Const
  help = 'With a little help of a programmer.';

Var
  A : AnsiString;


begin

  { lots of code }

  A:=Help;

  { Again some code}

  TranslateStrings;

  { More code }
```

After the call to `TranslateStrings`, the value of `A` will remain unchanged. This means that the assignment `A:=Help` must be executed again in order for the change to become visible. This is important, especially for GUI programs which have e.g. a menu. In order for the change in resource strings to become visible, the new values must be reloaded by program code into the menus ...

Chapter 10

Thread programming

10.1 Introduction

Free Pascal supports thread programming: There is a language construct available for thread-local storage (`ThreadVar`), and cross-platform low-level thread routines are available for those operating systems that support threads.

All routines for threading are available in the system unit, under the form of a thread manager. A thread manager must implement some basic routines which the RTL needs to be able to support threading. For Windows, a default threading manager is integrated in the system unit. For other platforms, a thread manager must be included explicitly by the programmer. On systems where posix threads are available, the cthreads unit implements a thread manager which uses the C POSIX thread library. No native pascal thread library exists for such systems.

Although it is not forbidden to do so, it is not recommended to use system-specific threading routines: The language support for multithreaded programs will not be enabled, meaning that threadvars will not work, the heap manager will be confused which may lead to severe program errors.

If no threading support is present in the binary, the use of thread routines or the creation of a thread will result in an exception or a run-time error 232.

For LINUX (and other Unixes), the C thread manager can be enabled by inserting the `cthreads` unit in the program's unit clause. Without this, threading programs will give an error when started. It is imperative that the unit be inserted as early in the uses clause as possible.

At a later time, a system thread manager may be implemented which implements threads without Libc support.

The following sections show how to program threads, and how to protect access to data common to all threads using (cross-platform) critical sections. Finally, the thread manager is explained in more detail.

10.2 Programming threads

To start a new thread, the `BeginThread` function should be used. It has one mandatory argument: the function which will be executed in the new thread. The result of the function is the exit result of the thread. The thread function can be passed a pointer, which can be used to acces initialization data: The programmer must make sure that the data is accessible from the thread and does not go out of scope before the thread has accessed it.

```
Type
```

```
  TThreadFunc = function(parameter : pointer) : ptrint;

function BeginThread(sa : Pointer;
                     stacksize : SizeUInt;
                     ThreadFunction : tthreadfunc;
                     p : pointer;
                     creationFlags : dword;
                     var ThreadId : TThreadID) : TThreadID;
```

This rather complicated full form of the function also comes in more simplified forms:

```
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;
```

The parameters have the following meaning:

ThreadFunction is the function that should be executed in the thread.

p If present, the pointer `p` will be passed to the thread function when it is started. If `p` is not specified, `Nil` is passed.

ThreadID If `ThreadID` is present, the ID of the thread will be stored in it.

stacksize if present, this parameter specifies the stack size used for the thread.

sa signal action. Important for LINUX only.

creationflags these are system-specific creation flags. Important for Windows and OS/2 only.

The newly started thread will run until the `ThreadFunction` exits, or until it explicitly calls the `EndThread` function:

```
procedure EndThread(ExitCode : DWord);
procedure EndThread;
```

The exitcode can be examined by the code which started the thread.

The following is a small example of how to program a thread:

```
{$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.
```

The `InterLockedIncrement` is a thread-safe version of the standard `Inc` function.

To provide system-independent support for thread programming, some utility functions are implemented to manipulate threads. To use these functions the thread ID must have been retrieved when the thread was started, because most functions require the ID to identify the thread on which they should act:

```
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;
```

The meaning of these functions should be clear:

SuspendThread Suspends the execution of the thread.

ResumeThread Resumes execution of a suspended thread.

KillThread Kills the thread: the thread is removed from memory.

WaitForThreadTerminate Waits for the thread to terminate. The function returns when the thread has finished executing, or when the timeout expired.

ThreadSetPriority Sets the execution priority of the thread. This call is not always allowed: your process may not have the necessary permissions to do this.

ThreadGetPriority Returns the current execution priority of the thread.

GetCurrentThreadId Returns the ID of the current thread.

ThreadSwitch Allows other threads to execute at this point. This means that it can cause a thread switch, but this is not guaranteed, it depends on the OS and the number of processors.

10.3 Critical sections

When programming threads, it is sometimes necessary to avoid concurrent access to certain resources, or to avoid having a certain routine executed by two threads. This can be done using a Critical Section. The FPC heap manager uses critical sections when multithreading is enabled.

The `TRTLCriticalSection` type is an Opaque type; it depends on the OS on which the code is executed. It should be initialized before it is first used, and should be disposed of when it is no longer necessary.

To protect a piece of code, a call to `EnterCriticalSection` should be made: When this call returns, it is guaranteed that the current thread is the only thread executing the subsequent code. The call may have suspended the current thread for an indefinite time to ensure this.

When the protected code is finished, `LeaveCriticalSection` must be called: this will enable other threads to start executing the protected code. To minimize waiting time for the threads, it is important to keep the protected block as small as possible.

The definition of these calls is as follows:

```
procedure InitCriticalSection(var cs: TRTLCriticalSection);
procedure DoneCriticalSection(var cs: TRTLCriticalSection);
procedure EnterCriticalSection(var cs: TRTLCriticalSection);
procedure LeaveCriticalSection(var cs: TRTLCriticalSection);
```

The meaning of these calls is again almost obvious:

InitCriticalSection Initializes a critical section. This call must be made before either `EnterCritititicalSection` or `LeaveCriticalSection` is used.

DoneCriticalSection Frees the resources associated with a critical section. After this call neither `EnterCritititicalSection` nor `LeaveCriticalSection` may be used.

EnterCriticalSection When this call returns, the calling thread is the only thread running the code between the `EnterCriticalSection` call and the following `LeaveCriticalsection` call.

LeaveCriticalSection Signals that the protected code can be executed by other threads.

Note that the `LeaveCriticalsection` call *must* be executed. Failing to do so will prevent all other threads from executing the code in the critical section. It is therefore good practice to enclose the critical section in a `Try..finally` block. Typically, the code will look as follows:

```
Var
  MyCS : TRTLCriticalSection;

Procedure CriticalProc;

begin
  EnterCriticalSection(MyCS);
  Try
    // Protected Code
  Finally
    LeaveCriticalSection(MyCS);
  end;
end;

Procedure ThreadProcedure;

begin
  // Code executed in threads...
 CriticalProc;
  // More Code executed in threads...
end;

begin
  InitCriticalSection(MyCS);
  // Code to start threads.
  DoneCriticalSection(MyCS);
end.
```

10.4 The Thread Manager

Just like the heap is implemented using a heap manager, and widestring management is left to a widestring manager, the threads have been implemented using a thread manager. This means that there is a record which has fields of procedural type for all possible functions used in the thread routines. The thread routines use these fields to do the actual work.

The thread routines install a system thread manager specific for each system. On Windows, the normal Windows routines are used to implement the functions in the thread manager. On Linux and other unices, the system thread manager does nothing: it will generate an error when thread routines are used. The rationale is that the routines for thread management are located in the C library. Implementing the system thread manager would make the RTL dependent on the C library, which is not desirable. To avoid dependency on the C library, the Thread Manager is implemented in a separate unit (cthreads). The initialization code of this unit sets the thread manager to a thread manager record which uses the C (pthreads) routines.

The thread manager record can be retrieved and set just as the record for the heap manager. The record looks (currently) as follows:

```
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;
```

The meaning of most of these functions should be obvious from the descriptions in previous sections.

The `InitManager` and `DoneManager` are called when the threadmanager is set (`InitManager`), or when it is unset (`DoneManager`). They can be used to initialize the thread manager or to clean up when it is done. If either of them returns `False`, the operation fails.

There are some special entries in the record, linked to thread variable management:

InitThreadVar is called when a thread variable must be initialized. It is of type

```
TInitThreadVarHandler = Procedure(var offset: dword;
                                  size: dword);
```

The `offset` parameter indicates the offset in the thread variable block: All thread variables are located in a single block, one after the other. The size parameter indicates the size of the thread variable. This function will be called once for all thread variables in the program.

RelocateThreadVar is called each time when a thread is started, and once for the main thread. It is of type:

```
TRelocateThreadVarHandler = Function(offset : dword) : pointer;
```

It should return the new location for the thread-local variable.

AllocateThreadVars is called when room must be allocated for all threadvars for a new thread. It's a simple procedure, without parameters. The total size of the threadvars is stored by the compiler in the `threadvarblocksize` global variable. The heap manager may *not* be used in this procedure: the heap manager itself uses threadvars, which have not yet been allocated.

ReleaseThreadVars This procedure (without parameters) is called when a thread terminates, and all memory allocated must be released again.

Chapter 11

Optimizations

11.1 Non processor specific

The following sections describe the general optimizations done by the compiler, they are not processor specific. Some of these require some compiler switch override while others are done automatically (those which require a switch will be noted as such).

11.1.1 Constant folding

In Free Pascal, if the operand(s) of an operator are constants, they will be evaluated at compile time.

Example

```
x:=1+2+3+6+5;
```

will generate the same code as

```
x:=17;
```

Furthermore, if an array index is a constant, the offset will be evaluated at compile time. This means that accessing MyData[5] is as efficient as accessing a normal variable.

Finally, calling `Chr`, `Hi`, `Lo`, `Ord`, `Pred`, or `Succ` functions with constant parameters generates no run-time library calls, instead, the values are evaluated at compile time.

11.1.2 Constant merging

Using the same constant string, floating point value or constant set two or more times generates only one copy of that constant.

11.1.3 Short cut evaluation

Evaluation of boolean expression stops as soon as the result is known, which makes code execute faster then if all boolean operands were evaluated.

11.1.4 Constant set inlining

Using the `in` operator is always more efficient then using the equivalent <>, =, <=, >=, < and > operators. This is because range comparisons can be done more easily with the `in` operator than

with normal comparison operators.

11.1.5 Small sets

Sets which contain less than 33 elements can be directly encoded using a 32-bit value, therefore no run-time library calls to evaluate operands on these sets are required; they are directly encoded by the code generator.

11.1.6 Range checking

Assignments of constants to variables are range checked at compile time, which removes the need of the generation of runtime range checking code.

11.1.7 And instead of modulo

When the second operand of a `mod` on an unsigned value is a constant power of 2, an `and` instruction is used instead of an integer division. This generates more efficient code.

11.1.8 Shifts instead of multiply or divide

When one of the operands in a multiplication is a power of two, they are encoded using arithmetic shift instructions, which generates more efficient code.

Similarly, if the divisor in a `div` operation is a power of two, it is encoded using arithmetic shift instructions.

The same is true when accessing array indexes which are powers of two, the address is calculated using arithmetic shifts instead of the multiply instruction.

11.1.9 Automatic alignment

By default all variables larger then a byte are guaranteed to be aligned at least on a word boundary.

Alignment on the stack and in the data section is processor dependent.

11.1.10 Smart linking

This feature removes all unreferenced code in the final executable file, making the executable file much smaller.

Smart linking is switched on with the `-Cx` command line switch, or using the `{$SMARTLINK ON}` global directive.

11.1.11 Inline routines

The following runtime library routines are coded directly into the final executable: `Lo`, `Hi`, `High`, `Sizeof`, `TypeOf`, `Length`, `Pred`, `Succ`, `Inc`, `Dec` and `Assigned`.

11.1.12 Stack frame omission

Under specific conditions, the stack frame (entry and exit code for the routine, see section section 6.3, page 81) will be omitted, and the variable will directly be accessed via the stack pointer.

Conditions for omission of the stack frame:

- The target CPU is x86 or ARM.
- The -O2 or -OoSTACKFRAME command line switch must be specified.
- No inline assembler is used.
- No exceptions are used.
- No routines are called with outgoing parameters on the stack.
- The function has no parameters.

11.1.13 Register variables

When using the `-Or` switch, local variables or parameters which are used very often will be moved to registers for faster access.

11.2 Processor specific

This lists the low-level optimizations performed, on a processor per processor basis.

11.2.1 Intel 80x86 specific

Here follows a listing of the optimizing techniques used in the compiler:

1. When optimizing for a specific Processor (`-Op1, -Op2, -Op3`, the following is done:
   - In `case` statements, a check is done whether a jump table or a sequence of conditional jumps should be used for optimal performance.
   - Determines a number of strategies when doing peephole optimization, e.g.: `movzbl (%ebp), %eax` will be changed into `xorl %eax,%eax; movb (%ebp),%al` for Pentium and PentiumMMX.
2. When optimizing for speed (`-OG`, the default) or size (`-Og`), a choice is made between using shorter instructions (for size) such as `enter $4`, or longer instructions `subl $4,%esp` for speed. When smaller size is requested, data is aligned to minimal boundaries. When speed is requested, data is aligned on most efficient boundaries as much as possible.
3. Fast optimizations (`-O1`): activate the peephole optimizer
4. Slower optimizations (`-O2`): also activate the common subexpression elimination (formerly called the "reloading optimizer")
5. Uncertain optimizations (`-OoUNCERTAIN`): With this switch, the common subexpression elimination algorithm can be forced into making uncertain optimizations.

   Although you can enable uncertain optimizations in most cases, for people who do not understand the following technical explanation, it might be the safest to leave them off.

Remark: If uncertain optimizations are enabled, the CSE algortihm assumes that

- If something is written to a local/global register or a procedure/function parameter, this value doesn't overwrite the value to which a pointer points.

- If something is written to memory pointed to by a pointer variable, this value doesn't overwrite the value of a local/global variable or a procedure/function parameter.

The practical upshot of this is that you cannot use the uncertain optimizations if you both write and read local or global variables directly and through pointers (this includes `Var` parameters, as those are pointers too).

The following example will produce bad code when you switch on uncertain optimizations:

```
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.
```

The reason it produces bad code is because you access the global variable `Temp` both through its name `Temp` and through a pointer, in this case using the `Bar` variable parameter, which is nothing but a pointer to `Temp` in the above code.

On the other hand, you can use the uncertain optimizations if you access global/local variables or parameters through pointers, and *only* access them through this pointer[1].

For example:

```
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.
```

[1] You can use multiple pointers to point to the same variable as well, that doesn't matter.

Will produce correct code, because the global variable `MyRecArrayPtr` is not accessed directly, but only through a pointer (`MyRecPtr` in this case).

In conclusion, one could say that you can use uncertain optimizations *only* when you know what you're doing.

11.2.2 Motorola 680x0 specific

Using the `-O2` (the default) switch does several optimizations in the code produced, the most notable being:

- Sign extension from byte to long will use `EXTB`.
- Returning of functions will use `RTD`.
- Range checking will generate no run-time calls.
- Multiplication will use the long `MULS` instruction, no runtime library call will be generated.
- Division will use the long `DIVS` instruction, no runtime library call will be generated.

11.3 Optimization switches

This is where the various optimizing switches and their actions are described, grouped per switch.

-On: with n = 1..3: these switches activate the optimizer. A higher level automatically includes all lower levels.

- Level 1 (`-O1`) activates the peephole optimizer (common instruction sequences are replaced by faster equivalents).
- Level 2 (`-O2`) enables the assembler data flow analyzer, which allows the common subexpression elimination procedure to remove unnecessary reloads of registers with values they already contain.
- Level 3 (`-O3`) equals level 2 optimizations plus some time-intensive optimizations.

-OG: This causes the code generator (and optimizer, IF activated), to favor faster, but code-wise larger, instruction sequences (such as "`subl $4,%esp`") instead of slower, smaller instructions ("`enter $4`"). This is the default setting.

-Og: This one is exactly the reverse of -OG, and as such these switches are mutually exclusive: enabling one will disable the other.

-Or: This setting causes the code generator to check which variables are used most, so it can keep those in a register.

-Opn: with n = 1..3: Setting the target processor does NOT activate the optimizer. It merely influences the code generator and, if activated, the optimizer:

- During the code generation process, this setting is used to decide whether a jump table or a sequence of successive jumps provides the best performance in a case statement.
- The peephole optimizer takes a number of decisions based on this setting, for example it translates certain complex instructions, such as

  ```
  movzbl (mem), %eax|
  ```

  to a combination of simpler instructions

```
xorl %eax, %eax
movb (mem), %al
```

for the Pentium.

-Ou: This enables uncertain optimizations. You cannot use these always, however. The previous section explains when they can be used, and when they cannot be used.

11.4 Tips to get faster code

Here, some general tips for getting better code are presented. They mainly concern coding style.

- Find a better algorithm. No matter how much you and the compiler tweak the code, a quicksort will (almost) always outperform a bubble sort, for example.
- Use variables of the native size of the processor you're writing for. This is currently 32-bit or 64-bit for Free Pascal, so you are best to use longword and longint variables.
- Turn on the optimizer.
- Write your if/then/else statements so that the code in the "then"-part gets executed most of the time (improves the rate of successful jump prediction).
- Do not use ansistrings, widestrings and exception support, as these require a lot of code overhead.
- Profile your code (see the -pg switch) to find out where the bottlenecks are. If you want, you can rewrite those parts in assembler. You can take the code generated by the compiler as a starting point. When given the `-a` command line switch, the compiler will not erase the assembler file at the end of the assembly process, so you can study the assembler file.

11.5 Tips to get smaller code

Here are some tips given to get the smallest code possible.

- Find a better algorithm.
- Use the `-Og` compiler switch.
- Regroup global static variables in the same module which have the same size together to minimize the number of alignment directives (which increases the `.bss` and `.data` sections unecessarily). Internally this is due to the fact that all static data is written to in the assembler file, in the order they are declared in the pascal source code.
- Do not use the `cdecl` modifier, as this generates about 1 additional instruction after each subroutine call.
- Use the smartlinking options for all your units (including the `system` unit).
- Do not use ansistrings, widestrings and exception support, as these require a lot of code overhead.
- Turn off range checking and stack-checking.
- Turn off runtime type information generation.

11.6 Whole Program Optimization

11.6.1 Overview

Traditionally, compilers optimise a program procedure by procedure, or at best compilation unit per compilation unit. Whole program optimisation (WPO) means that the compiler considers all compilation units that make up a program or library and optimises them using the combined knowledge of how they are used together in this particular case.

The way WPO generally works is as follows:

- The program is compiled normally, with an option to tell the compiler that it should store various bits of information into a feedback file.
- The program is recompiled a second time (and optionally all units that it uses) with WPO enabled, providing the feedback file generated in the first step as extra input to the compiler.

This is the scheme followed by Free Pascal.

The implementation of this scheme is highly compiler dependent. Another implementation could be that the compiler generates some kind of intermediary code (e.g., byte code) and the linker performs all wpo along with the translation to the target machine code

11.7 General principles

A few general principles have been followed when designing the FPC implementation of WPO:

- All information necessary to generate a WPO feedback file for a program is always stored in the ppu files. This means that it is possible to use a generic RTL for WPO (or, in general, any compiled unit). It does mean that the RTL itself will then not be optimised, the compiled program code and its units can be correctly optimised because the compiler knows everything it has to know about all RTL units.
- The generated WPO feedback file is plain text. The idea is that it should be easy to inspect this file by hand, and to add information to it produced by external tools if desired (e.g., profile information).
- The implementation of the WPO subsystem in the compiler is very modular, so it should be easy to plug in additional WPO information providers, or to choose at run time between different information providers for the same kind of information. At the same time, the interaction with the rest of the compiler is kept to a bare minimum to improve maintainability.
- It is possible to generate a WPO feedback file while at the same time using another one as input. In some cases, using this second feedback file as input during a third compilation can further improve the results.

11.7.1 How to use

Step 1: Generate WPO feedback file

The first step in WPO is to compile the program (or library) and all of its units as it would be done normally, but specifying in addition the 2 following options on the command-line:

```
-FW/path/to/feedbackfile.wpo -OW<selected_wpo_options>
```

The first option tells the compiler where the WPO feedback file should be written, the second option tells the compiler to switch on WPO optimalizations.

The compiler will then, right after the program or library has been linked, collect all necessary information to perform the requested WPO options during a subsequent compilation, and will store this information in the indicated file.

Step 2: Use the generated WPO feedback file

To actually apply the WPO options, the program (or library) and all or some of the units that it uses, must be recompiled using the option

```
-Fw/path/to/feedbackfile.wpo -Ow<selected_wpo_options>
```

(Note the small caps in the `w`). This will tell the compiler to use the feedback file generated in the previous step. The compiler will then read the information collected about the program during the previous compiler run, and use it during the current compilation of units and/or program/library.

Units not recompiled during the second pass will obviously not be optimised, but they will still work correctly when used together with the optimised units and program/library.

Remark: Note that the options must always be specified on the command-line: there is no source directive to turn on WPO, as it makes only sense to use WPO when compiling a complete program.

11.7.2 Available WPO optimizations

The `-OW` and `-Ow` command-line options require a comma-separated list of whole-program-optimization options. These are strings, each string denotes an option. The following is a list of available options:

all This enables all available whole program optimisations.

devirtcalls Changes virtual method calls into normal (static) method calls when the compiler can determine that a virtual method call will always go to the same static method. This makes such code both smaller and faster. In general, it is mainly an enabling optimisation for other optimisations, because it makes the program easier to analyse due to the fact that it reduces indirect control flow.

There are 2 limitations to this option:

1. The current implementation is context-insensitive. This means that the compiler only looks at the program as a whole and determines for each class type which methods can be devirtualised, rather than that it looks at each call statement and the surrounding code to determine whether or not this call can be devirtualised;
2. The current implementation does not yet devirtualise interface method calls. Not when calling them via an interface instance, nor when calling them via a class instance.

optvmts This optimisation looks at which class types can be instantiated and which virtual methods can be called in a program, and based on this information it replaces virtual method table (VMT) entries that can never be called with references to FPC_ABSTRACTERROR. This means that such methods, unless they are called directly via an inherited call from a child class/object, can be removed by the linker. It has little or no effect on speed, but can help reducing code size.

This option has 2 limitations:

1. Methods that are published, or getters/setters of published properties, can never be optimised in this way, because they can always be referred to and called via the RTTI (which the compiler cannot detect).

2. Such optimisations are not yet done for virtual class methods.

wsymbolliveness This parameter does not perform any optimisation by itself. It simply tells the compiler to record which functions/procedures were kept by the linker in the final program. During a subsequent wpo pass, the compiler can then ignore the removed functions/procedures as far as WPO is concerned (e.g., if a particular class type is only constructed in one unused procedure, then ignoring this procedure can improve the effectiveness of the previous two optimisations).

Again, there are some limitations:

1. This optimisation requires that the nm utility is installed on the system. For Linux binaries, objdump will also work. In the future, this information could also be extracted from the internal linker for the platforms that it supports.
2. Collecting information for this optimisation (using -OWsymbolliveness) requires that smart linking is enabled (-XX) and that symbol stripping is disabled (-Xs-). When only using such previously collected information, these limitations do not apply.

11.7.3 format of the WPO file

This information is mainly interesting if external data must be added to the WPO feedback file, e.g. from a profiling tool. For regular use of the WPO feature, the following information is not needed and can be ignored.

The file consists of comments and a number of sections. Comments are lines that start with a #. Each section starts with "% " followed by the name of the section (e.g.,% contextinsensitive_devirtualization).

After that, until either the end of the file or until the next line starting with with "% ", first a human readable description follows of the format of this section (in comments), and then the contents of the section itself.

There are no rules for how the contents of a section should look, except that lines starting with # are reserved for comments and lines starting with % are reserved for section markers.

Chapter 12

Programming shared libraries

12.1 Introduction

Free Pascal supports the creation of shared libraries on several operating systems. The following table (table (12.1)) indicates which operating systems support the creation of shared libraries.

Table 12.1: Shared library support

Operating systems	Library extension	Library prefix
linux	.so	lib
windows	.dll	<none>
BeOS	.so	lib
FreeBSD	.so	lib
NetBSD	.so	lib

The library prefix column indicates how the names of the libraries are resolved and created. For example, under LINUX, the library name will alwaus have the `lib` prefix when it is created. So if you create a library called mylib, under LINUX, this will result in the libmylib.so. Furthermore, when importing routines from shared libraries, it is not necessary to give the library prefix or the filename extension.

In the following sections we discuss how to create a library, and how to use these libraries in programs.

12.2 Creating a library

Creation of libraries is supported in any mode of the Free Pascal compiler, but it may be that the arguments or return values differ if the library is compiled in 2 different modes. E.g. if your function expects an `Integer` argument, then the library will expect different integer sizes if you compile it in Delphi mode or in TP mode.

A library can be created just as a program, only it uses the `library` keyword, and it has an `exports` section. The following listing demonstrates a simple library:

Listing: progex/subs.pp

```
{
```

```
  Example library
}
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.
```

The function `SubStr` does not have to be declared in the library file itself. It can also be declared in the interface section of a unit that is used by the library.

Compilation of this source will result in the creation of a library called libsubs.so on UNIX systems, or subs.dll on WINDOWS or OS/2. The compiler will take care of any additional linking that is required to create a shared library.

The library exports one function: `SubStr`. The case is important. The case as it appears in the `exports` clause is used to export the function.

If you want your library to be called from programs compiled with other compilers, it is important to specify the correct calling convention for the exported functions. Since the generated programs by other compilers do not know about the Free Pascal calling conventions, your functions would be called incorrectly, resulting in a corrupted stack.

On WINDOWS, most libraries use the `stdcall` convention, so it may be better to use that one if your library is to be used on WINDOWS systems. On most UNIX systems, the C calling convention is used, therefore the `cdecl` modifier should be used in that case.

12.3 Using a library in a pascal program

In order to use a function that resides in a library, it is sufficient to declare the function as it exists in the library as an `external` function, with correct arguments and return type. The calling convention used by the function should be declared correctly as well. The compiler will then link the library as specified in the `external` statement to your program[1].

For example, to use the library as defined above from a pascal program, you can use the following pascal program:

Listing: progex/psubs.pp

[1] If you omit the library name in the `external` modifier, then you can still tell the compiler to link to that library using the `{$Linklib}` directive.

```
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.
```

As is shown in the example, you must declare the function as `external`. Here also, it is necessary to specify the correct calling convention (it should always match the convention as used by the function in the library), and to use the correct casing for your declaration. Also notice, that the library importing did not specify the filename extension, nor was the `lib` prefix added.

This program can be compiled without any additional command-switches, and should run just like that, provided the library is placed where the system can find it. For example, on LINUX, this is /usr/lib or any directory listed in the /etc/ld.so.conf file. On WINDOWS, this can be the program directory, the WINDOWS system directory, or any directoy mentioned in the `PATH`.

Using the library in this way links the library to your program at compile time. This means that

1. The library must be present on the system where the program is compiled.
2. The library must be present on the system where the program is executed.
3. Both libraries must be exactly the same.

Or it may simply be that you don't know the name of the function to be called, you just know the arguments it expects.

It is therefore also possible to load the library at run-time, store the function address in a procedural variable, and use this procedural variable to access the function in the library.

The following example demonstrates this technique:

Listing: 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 'dl';
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.
```

As in the case of compile-time linking, the crucial thing in this listing is the declaration of the `TSubStrFunc` type. It should match the declaration of the function you're trying to use. Failure to specify a correct definition will result in a faulty stack or, worse still, may cause your program to crash with an access violation.

12.4 Using a pascal library from a C program

Remark: The examples in this section assume a LINUX system; similar commands as the ones below exist for other operating systems, though.

You can also call a Free Pascal generated library from a C program:

Listing: 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;
}
```

To compile this example, the following command can be used:

```
gcc -o ctest ctest.c -lsubs
```

provided the code is in ctest.c.

The library can also be loaded dynamically from C, as shown in the following example:

Listing: 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;
}
```

This can be compiled using the following command:

```
gcc -o ctest2 ctest2.c -ldl
```

The `-ldl` tells gcc that the program needs the libdl.so library to load dynamical libraries.

12.5 Some Windows issues

By default, Free Pascal (actually, the linker used by Free Pascal) creates libraries that are not relocatable. This means that they must be loaded at a fixed address in memory: this address is called the ImageBase address. If two Free Pascal generated libraries are loaded by a program, there will be a conflict, because the first librarie already occupies the memory location where the second library should be loaded.

There are 2 switches in Free Pascal which control the generation of shared libraries under WINDOWS:

-WR Generate a relocatable library. This library can be moved to another location in memory if the ImageBase address it wants is already in use.

-WB Specify the ImageBase address for the generated library. The standard ImageBase used by Free Pascal is `0x10000000`. This switch allows changing that by specifying another address, for instance `-WB11000000`.

The first option is preferred, as a program may load many libraries present on the system, and they could already be using the ImageBase address. The second option is faster, as no relocation needs to be done if the ImageBase address is not yet in use.

Chapter 13

Using Windows resources

13.1 The resource directive $R

Under WINDOWS and LINUX (or any platform using ELF binaries) [1], you can include resources in your executable or library using the `{$R filename}` directive. These resources can then be accessed through the standard WINDOWS API calls: these calls have been made available in the other platforms as well.

When the compiler encounters a resource directive, it just creates an entry in the unit .ppu file; it doesn't link the resource. Only when it creates a library or executable, it looks for all the resource files for which it encountered a directive, and tries to link them in.

The default extension for resource files is .res. When the filename has as the first character an asterix (*), the compiler will replace the asterix with the name of the current unit, library or program.

Remark: This means that the asterix may only be used after a `unit`, `library` or `program` clause.

13.2 Creating resources

The Free Pascal compiler itself doesn't create any resource files; it just compiles them into the executable. To create resource files, you can use some GUI tools as the Borland resource workshop; but it is also possible to use a WINDOWS resource compiler like GNU windres. windres comes with the GNU binutils, but the Free Pascal distribution also contains a version which you can use.

The usage of windres is straightforward; it reads an input file describing the resources to create and outputs a resource file.

A typical invocation of windres would be

```
windres -i mystrings.rc -o mystrings.res
```

this will read the mystrings.rc file and output a mystrings.res resource file.

A complete overview of the windres tools is outside the scope of this document, but here are some things you can use it for:

stringtables that contain lists of strings.

bitmaps which are read from an external file.

[1] As of development version 2.3.1, all FPC supported platforms now have resources available.

icons which are also read from an external file.

Version information which can be viewed with the WINDOWS explorer.

Menus Can be designed as resources and used in your GUI applications.

Arbitrary data Can be included as resources and read with the windows API calls.

Some of these will be described below.

13.3 Using string tables.

String tables can be used to store and retrieve large collections of strings in your application.
A string table looks as follows:

```
STRINGTABLE { 1, "hello World !"
              2, "hello world again !"
              3, "last hello world !" }
```

You can compile this (we assume the file is called tests.rc) as follows:

```
windres -i tests.rc -o tests.res
```

And this is the way to retrieve the strings from your program:

```
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.
```

The call to `FindResource` searches for the stringtable in the compiled-in resources. The `LoadString` function then reads the string with index `i` out of the table, and puts it in a buffer, which can then be used. Both calls are in the windows unit.

13.4 Inserting version information

The win32 API allows the storing of version information in your binaries. This information can be made visible with the WINDOWS Explorer, by right-clicking on the executable or library, and selecting the 'Properties' menu. In the tab 'Version' the version information will be displayed.

Here is how to insert version information in your binary:

```
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 information 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"
  }
 }
}
```

As you can see, you can insert various kinds of information in the version info block. The keyword `VERSIONINFO` marks the beginning of the version information resource block. The keywords `FILEVERSION`, `PRODUCTVERSION` give the actual file version, while the block `StringFileInfo` gives other information that is displayed in the explorer.

The Free Component Library comes with a unit (fileinfo) that allows to extract and view version information in a straightforward and easy manner; the demo program that comes with it (showver) shows version information for an arbitrary executable or DLL.

13.5 Inserting an application icon

When WINDOWS shows an executable in the Explorer, it looks for an icon in the executable to show in front of the filename, the application icon.

Inserting an application icon is very easy and can be done as follows

```
AppIcon ICON "filename.ico"
```

This will read the file filename.ico and insert it in the resource file.

13.6 Using a Pascal preprocessor

Sometimes you want to use symbolic names in your resource file, and use the same names in your program to access the resources. To accomplish this, there exists a preprocessor for windres that understands pascal syntax: fprcp. This preprocessor is shipped with the Free Pascal distribution.

The idea is that the preprocessor reads a pascal unit that has some symbolic constants defined in it, and replaces symbolic names in the resource file by the values of the constants in the unit:

As an example: consider the following unit:

```
unit myunit;

interface

Const
  First  = 1;
  Second = 2:
  Third  = 3;

Implementation
end.
```

And the following resource file:

```
#include "myunit.pp"

STRINGTABLE { First, "hello World !"
              Second, "hello world again !"
              Third, "last hello world !" }
```

If you invoke windres with the `preprocessor` option:

```
windres --preprocessor fprcp -i myunit.rc -o myunit.res
```

then the preprocessor will replace the symbolic names 'first', 'second' and 'third' with their actual values.

In your program, you can then refer to the strings by their symbolic names (the constants) instead of using a numeric index.

Appendix A

Anatomy of a unit file

A.1 Basics

As described in chapter 4, page 71, unit description files (hereafter called PPU files for short), are used to determine if the unit code must be recompiled or not. In other words, the PPU files act as mini-makefiles, which is used to check dependencies of the different code modules, as well as verify if the modules are up to date or not. Furthermore, it contains all public symbols defined for a module.

The general format of the `ppu` file format is shown in figure (A.1).

To read or write the ppufile, the ppu unit ppu.pas can be used, which has an object called tppufile which holds all routines that deal with ppufile handling. While describing the layout of a ppufile, the methods which can be used for it are presented as well.

A unit file consists of basically five or six parts:

1. A unit header.
2. A general information part (wrongly named interface section in the code)
3. A definition part. Contains all type and procedure definitions.
4. A symbol part. Contains all symbol names and references to their definitions.
5. A browser part. Contains all references from this unit to other units and inside this unit. Only available when the `uf_has_browser` flag is set in the unit flags
6. A file implementation part (currently unused).

A.2 reading ppufiles

We will first create an object ppufile which will be used below. We are opening unit test.ppu as an example.

```
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;
```

Note: When a function fails (for example not enough bytes left in an entry) it sets the `ppufile.error` variable.

A.3 The Header

The header consists of a record (`tppuheader`) containing several pieces of information for recompilation. This is shown in table (A.1). The header is always stored in little-endian format.

Table A.1: PPU Header

offset	size (bytes)	description
00h	3	Magic : 'PPU' in ASCII
03h	3	PPU File format version (e.g : '021' in ASCII)
06h	2	Compiler version used to compile this module (major,minor)
08h	2	Code module target processor
0Ah	2	Code module target operating system
0Ch	4	Flags for PPU file
10h	4	Size of PPU file (without header)
14h	4	CRC-32 of the entire PPU file
18h	4	CRC-32 of partial data of PPU file (public data mostly)
1Ch	8	Reserved

The header is already read by the `ppufile.openfile` command. You can access all fields using `ppufile.header` which holds the current header record.

Table A.2: PPU CPU Field values

value	description
0	unknown
1	Intel 80x86 or compatible
2	Motorola 680x0 or compatible
3	Alpha AXP or compatible
4	PowerPC or compatible

Some of the possible flags in the header are described in table (A.3). Not all the flags are described, for more information, read the source code of ppu.pas.

Table A.3: PPU Header Flag values

Symbolic bit flag name	Description
uf_init	Module has an initialization (either Delphi or TP style) section.
uf_finalize	Module has a finalization section.
uf_big_endian	All the data stored in the chunks is in big-endian format.
uf_has_browser	Unit contains symbol browser information.
uf_smart_linked	The code module has been smartlinked.
uf_static_linked	The code is statically linked.
uf_has_resources	Unit has resource section.

A.4 The sections

Apart from the header section, all the data in the PPU file is separated into data blocks, which permit easily adding additional data blocks, without compromising backward compatibility. This is similar to both Electronic Arts IFF chunk format and Microsoft's RIFF chunk format.

Each 'chunk' (`tppuentry`) has the following format, and can be nested:

Table A.4: chunk data format

offset	size (bytes)	description
00h	1	Block type (nested (2) or main (1))
01h	1	Block identifier
02h	4	Size of this data block
06h+	<variable>	Data for this block

Each main section chunk must end with an end chunk. Nested chunks are used for record, class or object fields.

To read an entry you can simply call `ppufile.readentry:byte`, it returns the `tppuentry.nr` field, which holds the type of the entry. A common way how this works is (example is for the symbols):

```
  repeat
    b:=ppufile.readentry;
    case b of
   ib<etc> : begin
             end;
 ibendsyms : break;
    end;
  until false;
```

The possible entry types are found in ppu.pas, but a short description of the most common ones are shown in table (A.5).

Table A.5: Possible PPU Entry types

Symbolic name	Location	Description
ibmodulename	General	Name of this unit.
ibsourcefiles	General	Name of source files.
ibusedmacros	General	Name and state of macros used.
ibloadunit	General	Modules used by this units.
inlinkunitofiles	General	Object files associated with this unit.
iblinkunitstaticlibs	General	Static libraries associated with this unit.
iblinkunitsharedlibs	General	Shared libraries associated with this unit.
ibendinterface	General	End of General information section.
ibstartdefs	Interface	Start of definitions.
ibenddefs	Interface	End of definitions.
ibstartsyms	Interface	Start of symbol data.
ibendsyms	Interface	End of symbol data.
ibendimplementation	Implementation	End of implementation data.
ibendbrowser	Browser	End of browser section.
ibend	General	End of Unit file.

Then you can parse each entry type yourself. `ppufile.readentry` will take care of skipping unread bytes in the entry and reads the next entry correctly! A special function is `skipuntilentry(untilb:byte):boole` which will read the ppufile until it finds entry `untilb` in the main entries.

Parsing an entry can be done with `ppufile.getxxx` functions. The available functions are:

```
procedure ppufile.getdata(var b;len:longint);
function  getbyte:byte;
function  getword:word;
function  getlongint:longint;
function  getreal:ppureal;
function  getstring:string;
```

To check if you're at the end of an entry you can use the following function:

```
function  EndOfEntry:boolean;
```

notes:

1. `ppureal` is the best real that exists for the cpu where the unit is created for. Currently it is `extended` for i386 and `single` for m68k.
2. the `ibobjectdef` and `ibrecorddef` have stored a definition and symbol section for themselves. So you'll need a recursive call. See ppudump.pp for a correct implementation.

A complete list of entries and what their fields contain can be found in ppudump.pp.

A.5 Creating ppufiles

Creating a new ppufile works almost the same as reading one. First you need to init the object and call create:

```
ppufile:=new(pppufile,init('output.ppu'));
ppufile.createfile;
```

After that you can simply write all needed entries. You'll have to take care that you write at least the basic entries for the sections:

```
ibendinterface
ibenddefs
ibendsyms
ibendbrowser (only when you've set uf_has_browser!)
ibendimplementation
ibend
```

Writing an entry is a little different than reading it. You need to first put everything in the entry with 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);
```

After putting all the things in the entry you need to call `ppufile.writeentry(ibnr:byte)` where `ibnr` is the entry number you're writing.

At the end of the file you need to call `ppufile.writeheader` to write the new header to the file. This takes automatically care of the new size of the ppufile. When that is also done you can call `ppufile.closefile` and dispose the object.

Extra functions/variables available for writing are:

```
ppufile.NewHeader;
ppufile.NewEntry;
```

This will give you a clean header or entry. Normally this is called automatically in `ppufile.writeentry`, so there should be no need to call these methods. You can call

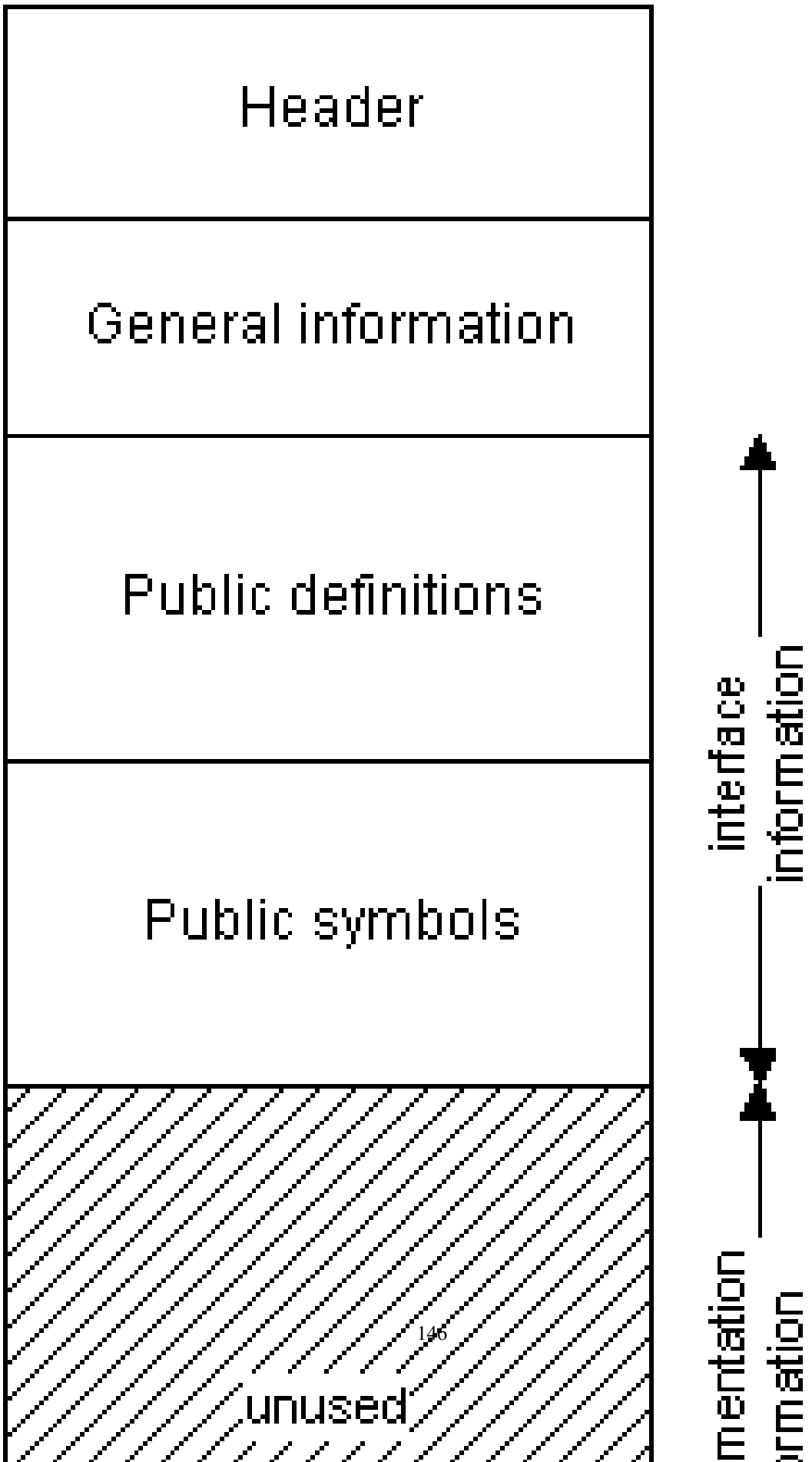
```
ppufile.flush;
```

to flush the current buffers to the disk, and you can set

```
ppufile.do_crc:boolean;
```

to `False` if you don't want the crc to be updated when writing to disk. This is necessary if you write for example the browser data.

Figure A.1: The PPU file format



Appendix B

Compiler and RTL source tree structure

B.1 The compiler source tree

All compiler source files are in several directories, normally the non-processor specific parts are in source/compiler. Subdirectories are present for each of the supported processors and target operating systems.

For more informations about the structure of the compiler have a look at the Compiler Manual which contains also some informations about compiler internals.

The compiler directory also contains a subdirectory `utils`, which contains mainly the utilities for creation and maintainance of the message files.

B.2 The RTL source tree

The RTL source tree is divided in many subdirectories, but is very structured and easy to understand. It mainly consists of three parts:

1. A OS-dependent directory. This contains the files that are different for each operating system. When compiling the RTL, you should do it here. The following directories exist:
   - amiga for the AMIGA.
   - atari for the ATARI.
   - beos for BEOS. It has one subdirectory for each of the supported processors.
   - bsd Common files for the various BSD platforms.
   - darwin for the unix-compatibility layer on Mac OS.
   - embedded A template for embedded targets.
   - emx OS/2 using the EMX extender.
   - freebsd for the FREEBSD platform.
   - gba Game Boy Advanced.
   - go32v2 For DOS, using the GO32v2 extender.
   - linux for LINUX platforms. It has one subdirectory for each of the supported processors.
   - macos for the Mac OS platform.

- morphos for the MorphOS platform.
- nds for the Nintendo DS platform.
- netbsd for NETBSD platforms. It has one subdirectory for each of the supported processors.
- netware for the Novell netware platform.
- netwlibc for the Novell netware platform using the C library.
- openbsd for the OpenBSD platform.
- os2 for OS/2.
- palmos for the PALMOS Dragonball processor based platform.
- posix for posix interfaces (used for easier porting).
- solaris for the SOLARIS platform. It has one subdirectory for each of the supported processors.
- symbian for the symbian mobile phone OS.
- qnx for the QNX REALTIME PLATFORM.
- unix for unix common interfaces (used for easier porting).
- win32 for Windows 32-bit platforms.
- win64 for Windows 64-bit platforms.
- wince for the Windows CE embedded platform (arm CPU).
- posix for posix interfaces (used for easier porting).

2. A processor dependent directory. This contains files that are system independent, but processor dependent. It contains mostly optimized routines for a specific processor. The following directories exist:
   - arm for the ARM series of processors.
   - i386 for the Intel 80x86 series of processors.
   - m68k for the Motorola 680x0 series of processors.
   - powerpc for the PowerPC processor.
   - powerpc64 for the PowerPC 64-bit processor.
   - sparc for the SUN SPARC processor.
   - x86_64 for Intel compatible 64-bit processors such as the AMD64.
3. An OS-independent and Processor independent directory: inc. This contains complete units, and include files containing interface parts of units as well as generic versions of processor specific routines.
4. The Object Pascal extensions (mainly Delphi compatibility units) are in the objpas directory. The sysutils and classes units are in separate subdirectories of the objpas directory.

Appendix C

Compiler limits

There are certain compiler limits inherent to the compiler:

1. Procedure or Function definitions can be nested to a level of 32. This can be changed by changing the `maxnesting` constant.
2. Maximally 1024 units can be used in a program when using the compiler. You can change this by redefining the `maxunits` constant in the compiler source file.
3. The maximum nesting level of pre-processor macros is 16. This can be changed by changing the value of `max_macro_nesting`.
4. Arrays are limited to 2 GBytes in size in the default (32-bit) processor mode.

For processor specific compiler limitations refer to the Processor Limitations section in this guide (6.8).

Appendix D

Compiler modes

Here we list the exact effect of the different compiler modes. They can be set with the `$Mode` switch, or by command line switches.

D.1 FPC mode

This mode is selected by the `$MODE FPC` switch. On the command line, this means that you use none of the other compatibility mode switches. It is the default mode of the compiler (`-Mfpc`). This means essentially:

1. You must use the address operator to assign procedural variables.
2. A forward declaration must be repeated exactly the same by the implementation of a function/procedure. In particular, you cannot omit the parameters when implementing the function or procedure.
3. Overloading of functions is allowed.
4. Nested comments are allowed.
5. The Objpas unit is NOT loaded.
6. You can use the cvar type.
7. PChars are converted to strings automatically.
8. Strings are shortstrings by default.

D.2 TP mode

This mode is selected by the `$MODE TP` switch. It tries to emulate, as closely as possible, the behavior of Turbo Pascal 7. On the command line, this mode is selected by the `-Mtp` switch.

1. Enumeration sizes default to a storage size of 1 byte if there are less than 257 elements.
2. You cannot use the address operator to assign procedural variables.
3. A forward declaration does not have to be repeated exactly the same by the implementation of a function/procedure. In particular, you can omit the parameters when implementing the function or procedure.

4. Overloading of functions is not allowed.
5. The Objpas unit is NOT loaded.
6. Nested comments are not allowed.
7. You cannot use the cvar type.
8. Strings are shortstrings by default.

D.3 Delphi mode

This mode is selected by the `$MODE DELPHI` switch. It tries to emulate, as closely as possible, the behavior of Delphi 4 or higher. On the command line, this mode is selected by the `-Mdelpih` switch.

1. You cannot use the address operator to assign procedural variables.
2. A forward declaration does not have to be repeated exactly the same by the implementation of a function/procedure. In particular, you can omit the parameters when implementing the function or procedure.
3. Ansistrings are default, this means that `$MODE DELPHI` implies an implicit `{$H ON}`.
4. Overloading of functions is not allowed.
5. Nested comments are not allowed.
6. The Objpas unit is loaded right after the system unit. One of the consequences of this is that the type `Integer` is redefined as `Longint`.
7. Parameters in class methods can have the same names as class properties (although it is bad programming practice).

D.4 OBJFPC mode

This mode is selected by the `$MODE OBJFPC` switch. On the command line, this mode is selected by the `-Mobjfpc` switch.

1. You must use the address operator to assign procedural variables.
2. A forward declaration must be repeated exactly the same by the implementation of a function/procedure. In particular, you cannot omit the parameters when implementing the function or procedure, and the calling convention must be repeated as well.
3. Overloading of functions is allowed.
4. Nested comments are allowed.
5. The Objpas unit is loaded right after the system unit. One of the consequences of this is that the type `Integer` is redefined as `Longint`.
6. You can use the cvar type.
7. PChars are converted to strings automatically.
8. Parameters in class methods cannot have the same names as class properties.
9. Strings are shortstrings by default. You can use the `-Sh` command line switch or the `{$H+}` switch to change this.

D.5 MACPAS mode

This mode is selected by the `$MODE MACPAS` switch. On the command line, this mode is selected by the `-Mmacpas` switch. It mainly switches on some extra features:

1. Support for the `$SETC` directive.
2. Support for the `$IFC`, `$ELSEC` and `$ENDC` directives.
3. Support for the `UNDEFINED` construct in macros.
4. Support for `TRUE` and `FALSE` as values in macro expressions.
5. Macros may be assigned hexadecimal numbers, like `$2345`.
6. The `Implementation` keyword can be omitted if the implementation section is empty.
7. The `cdecl` modifier keyword can be abbreviated to `C`.
8. `UNIV` modifer for types in parameter lists is accepted, but is otherwise ignored.
9. `...` (ellipsis) is allowed in procedure declarations, is functionally equal to the `varargs` keyword.

(Note: Macros are called 'Compiler Variables' in Mac OS dialects.)

Currently, the following Mac OS pascal extensions are not yet supported in `MACPAS` mode:

- A nested procedure cannot be an actual parameter to a procedure.
- No anonymous procedure types in formal parameters.
- External procedures declared in the interface must have the directive `External`.
- `Continue` instead of `Cycle`.
- `Break` instead of `Leave`
- `Exit` should not have the name of the procedure to exit as parameter. Instead, for a function the value to return can be supplied as parameter.
- No propagating `uses`.
- Compiler directives defined in interface sections are not exported.

Appendix E

Using fpcmake

E.1 Introduction

Free Pascal comes with a special makefile tool, fpcmake, which can be used to construct a Makefile for use with GNU make. All sources from the Free Pascal team are compiled with this system.

fpcmake uses a file Makefile.fpc and constructs a file Makefile from it, based on the settings in Makefile.fpc.

The following sections explain what settings can be set in Makefile.fpc, what variables are set by `fpcmake`, what variables it expects to be set, and what targets it defines. After that, some settings in the resulting Makefile are explained.

E.2 Functionality

fpcmake generates a makefile, suitable for GNU make, which can be used to

1. Compile units and programs, fit for testing or for final distribution.
2. Compile example units and programs separately.
3. Install compiled units and programs in standard locations.
4. Make archives for distribution of the generated programs and units.
5. Clean up after compilation and tests.

fpcmake knows how the Free Pascal compiler operates, which command line options it uses, how it searches for files and so on; It uses this knowledge to construct sensible command lines.

Specifically, it constructs the following targets in the final makefile:

all Makes all units and programs.

debug Makes all units and programs with debug info included.

smart Makes all units and programs in smartlinked version.

examples Makes all example units and programs.

shared Makes all units and programs in shared library version (currently disabled).

install Installs all units and programs.

sourceinstall Installs the sources to the Free Pascal source tree.

exampleinstall Installs any example programs and units.

distinstall Installs all units and programs, as well as example units and programs.

zipinstall Makes an archive of the programs and units which can be used to install them on another location, i.e. it makes an archive that can be used to distribute the units and programs.

zipsourceinstall Makes an archive of the program and unit sources which can be used to distribute the sources.

zipexampleinstall Makes an archive of the example programs and units which can be used to install them on another location, i.e. it makes an archive that can be used to distribute the example units and programs.

zipdistinstall Makes an archive of both the normal as well as the example programs and units. This archive can be used to install them on another location, i.e. it makes an archive that can be used to distribute.

clean Cleans all files that are produced by a compilation.

distclean Cleans all files that are produced by a compilation, as well as any archives, examples or files left by examples.

cleanall Same as clean.

info Produces some information on the screen about used programs, file and directory locations, where things will go when installing and so on.

Each of these targets can be highly configured, or even totally overridden by the configuration file Makefile.fpc.

E.3 Usage

fpcmake reads a Makefile.fpc and converts it to a Makefile suitable for reading by GNU make to compile your projects. It is similar in functionality to GNU configure or Imake for making X projects.

fpcmake accepts filenames of makefile description files as its command line arguments. For each of these files it will create a Makefile in the same directory where the file is located, overwriting any existing file with that name.

If no options are given, it just attempts to read the file Makefile.fpc in the current directory and tries to construct a Makefile from it if the `-m` option is given. Any previously existing Makefile will be erased.

if the `-p` option is given, instead of a Makefile, a `Package.fpc` is generated. A Package.fpc file describes the package and its dependencies on other packages.

Additionally, the following command line options are recognized:

-p A Package.fpc file is generated.

-w A Makefile is generated.

-T targets Support only specified target systems. `Targets` is a comma-separated list of targets. Only rules for the specified targets will be written.

-v Be more verbose.

-q be quiet.

-h Writes a small help message to the screen.

E.4 Format of the configuration file

This section describes the rules that can be present in the file that is processed by fpcmake.

The file Makefile.fpc is a plain ASCII file that contains a number of pre-defined sections as in a WINDOWS .ini-file, or a Samba configuration file.

They look more or less as follows:

```
[package]
name=mysql
version=1.0.5

[target]
units=mysql_com mysql_version mysql
examples=testdb

[require]
libc=y

[install]
fpcpackage=y

[default]
fpcdir=../..
```

The following sections are recognized (in alphabetical order):

E.4.1 clean

Specifies rules for cleaning the directory of units and programs. The following entries are recognized:

units names of all units that should be removed when cleaning. Don't specify extensions, the makefile will append these by itself.

files names of additional (not unit files) files that should be removed. Specify full filenames. The resource string table files (.rst files) are cleaned if they are specified in the `files` section.

E.4.2 compiler

In this section values for various compiler options can be specified, such as the location of several directories and search paths.

The following general keywords are recognised:

options The value of this key will be passed on to the compiler (verbatim) as command line options.

version If a specific or minimum compiler version is needed to compile the units or programs, then this version should be specified here.

The following keys can be used to control the location of the various directories used by the compiler:

unitdir A colon-separated list of directories that must be added to the unit search path of the compiler (using the `-Fu` option).

librarydir A colon-separated list of directories that must be added to the library search path of the compiler (using the `-Fl` option).

objectdir A colon-separated list of directories that must be added to the object file search path of the compiler (using the `-Fo` option).

targetdir Specifies the directory where the compiled programs should go (using the `-FE` option).

sourcedir A space separated list of directories where sources can reside. This will be used for the `vpath` setting of GNU make.

unittargetdir Specifies the directory where the compiled units should go (using the `-FU` option).

includedir A colon-separated list of directories that must be added to the include file search path of the compiler (using the `-Fi` option).

E.4.3 Default

The `default` section contains some default settings. The following keywords are recognized:

cpu Specifies the default target processor for which the Makefile should compile the units and programs. By default this is determined from the compiler info.

dir Specifies any subdirectories that make should also descend in and make the specified target there as well.

fpcdir Specifies the directory where all the Free Pascal source trees reside. Below this directory the Makefile expects to find the rtl and packages directory trees.

rule Specifies the default rule to execute. fpcmake will make sure that this rule is executed if make is executed without arguments, i.e., without an explicit target.

target Specifies the default operating system target for which the Makefile should compile the units and programs. By default this is determined from the default compiler target.

E.4.4 Dist

The `Dist` section controls the generation of a distribution package. A distribution package is a set of archive files (zip files or tar files on unix systems) that can be used to distribute the package.

The following keys can be placed in this section:

destdir Specifies the directory where the generated zip files should be placed.

zipname Name of the archive file to be created. If no zipname is specified, this defaults to the package name.

ziptarget This is the target that should be executed before the archive file is made. This defaults to `install`.

E.4.5 Install

Contains instructions for installation of the compiled units and programs. The following keywords are recognized:

basedir The directory that is used as the base directory for the installation of units. Default this is `prefix` appended with `/lib/fpc/FPC_VERSION` for LINUX or simply the `prefix` directory on other platforms.

datadir Directory where data files will be installed, i.e. the files specified with the `Files` keyword.

fpcpackage A boolean key. If this key is specified and equals `y`, the files will be installed as a fpc package under the Free Pascal units directory, i.e. under a separate directory. The directory will be named with the name specified in the `package` section.

files extra data files to be installed in the directory specified with the `datadir` key.

prefix is the directory below which all installs are done. This corresponds to the `prefix` argument to GNU configure. It is used for the installation of programs and units. By default, this is /usr on LINUX, and /pp on all other platforms.

units extra units that should be installed, and which are not part of the unit targets. The units in the units target will be installed automatically.

Units will be installed in the subdirectory units/$(OS_TARGET) of the `dirbase` entry.

E.4.6 Package

If a package (i.e. a collection of units that work together) is being compiled, then this section is used to keep package information. The following information can be stored:

name The name of the package. When installing it under the package directory, this name will be used to create a directory (unless it is overridden by one of the installation options).

version The version of this package.

main If the package is part of another package, this key can be specified to indicate which package it is part of.

E.4.7 Prerules

Anything that is in this section will be inserted as-is in the makefile *before* the makefile target rules that are generated by fpcmake. This means that any variables that are normally defined by fpcmake rules should not be used in this section.

E.4.8 Requires

This section is used to indicate dependency on external packages (i.e units) or tools. The following keywords can be used:

fpcmake Minimal version of fpcmake that this makefile.fpc needs.

packages Other packages that should be compiled before this package can be compiled. Note that this will also add all packages these packages depend on to the dependencies of this package. By default, the Free Pascal Run-Time Library is added to this list.

libc A boolean value that indicates whether this package needs the C library.

nortl A boolean that prevents the addition of the Free Pascal Run-Time Library to the required packages.

unitdir These directories will be added to the units search path of the compiler.

packagedir List of package directories. The packages in these directories will be made as well before making the current package.

tools A list of executables of extra tools that are required. The full path to these tools will be defined in the makefile as a variable with the same name as the tool name, only in uppercase. For example, the following definition:

```
tools=upx
```

will lead to the definition of a variable with the name `UPX` which will contain the full path to the upx executable.

E.4.9 Rules

In this section dependency rules for the units and any other needed targets can be inserted. It will be included at the end of the generated makefile. Targets or 'default rules' that are defined by fpcmake can be inserted here; if they are not present, then fpcmake will generate a rule that will call the generic `fpc_` version. For a list of standard targets that will be defined by fpcmake, see section E.2, page 153.

For example, it is possible to define a target `all:`. If it is not defined, then fpcmake will generate one which simply calls `fpc_all`:

```
all: fpc_all
```

The `fpc_all` rule will make all targets as defined in the `Target` section.

E.4.10 Target

This is the most important section of the makefile.fpc file. Here the files are defined which should be compiled when the 'all' target is executed.

The following keywords can be used there:

dirs A space separated list of directories where make should also be run.

exampledirs A space separated list of directories with example programs. The examples target will descend in this list of directories as well.

examples A space separated list of example programs that need to be compiled when the user asks to compile the examples. Do not specify an extension, the extension will be appended.

loaders A space separated list of names of assembler files that must be assembled. Don't specify the extension, the extension will be appended.

programs A space separated list of program names that need to be compiled. Do not specify an extension, the extension will be appended.

rsts a list of rst files that needs to be converted to .po files for use with GNU gettext and internationalization routines. These files will be installed together with the unit files.

units A space separated list of unit names that need to be compiled. Do not specify an extension, just the name of the unit as it would appear un a `uses` clause is sufficient.

E.5 Programs needed to use the generated makefile

At least the following programs are needed by the generated Makefile to function correctly:

cp A copy program.

date A program that prints the date.

install A program to install files.

make The make program, obviously.

pwd A program that prints the current working directory.

rm A program to delete files.

zip The zip archiver program. (on Dos / Windows / OS/2 systems only)

tar The tar archiver program (on Unix systems only).

These are standard programs on LINUX systems, with the possible exception of make. For DOS, WINDOWS NT or OS/2 / eComStation, they are distributed as part of Free Pascal releases.

The following programs are optionally needed if you use some special targets. Which ones you need are controlled by the settings in the `tools` section.

cmp A DOS and WINDOWS NT file comparer.

diff A file comparer.

ppdep The ppdep dependency lister. Distributed with Free Pascal.

ppumove The Free Pascal unit mover.

upx The UPX executable packer.

All of these can also be found on the Free Pascal FTP site for DOS and WINDOWS NT. ppdep and ppumove are distributed with the Free Pascal compiler.

E.6 Variables that affect the generated makefile

The makefile generated by fpcmake contains a lot of variables. Some of them are set in the makefile itself, others can be set and are taken into account when set.

These variables can be split in two groups:

- Directory variables.
- Compiler command line variables.

Each group will be discussed separately.

E.6.1 Directory variables

The first set of variables controls the directories that are recognised in the makefile. They should not be set in the Makefile.fpc file, but can be specified on the command line.

INCDIR This is a list of directories, separated by spaces, that will be added as include directories to the compiler command line. Each directory in the list is prepended with `-Fi` and added to the compiler options.

UNITDIR This is a list of directories, separated by spaces, that will be added as unit search directories to the compiler command line. Each directory in the list is prepended with `-Fu` and added to the compiler options.

LIBDIR Is a list of library paths, separated by spaces. Each directory in the list is prepended with `-Fl` and added to the compiler options.

OBJDIR Is a list of object file directories, separated by spaces, that is added to the object files path, i.e. Each directory in the list is prepended with `-Fo`.

E.6.2 Compiler command line variables

The following variables can be set on the make command line, they will be recognised and integrated in the compiler command line options.:

CREATESMART If this variable is defined, it tells the compiler to create smartlinked units. Adds `-CX` to the command line options.

DEBUG If defined, this will cause the compiler to include debug information in the generated units and programs. It adds `-gl` to the compiler command line, and will define the `DEBUG` define.

LINKSMART Defining this variable tells the compiler to use smartlinking. It adds `-XX` to the compiler command line options.

OPT Any options that you want to pass to the compiler. The contents of `OPT` is simply added to the compiler command line.

OPTDEF Are optional defines, added to the command line of the compiler. They get `-d` prepended to them.

OPTIMIZE If this variable is defined, this will add `-OG2p3` to the command line options.

RELEASE If this variable is defined, this will add the `-Xs -OG2p3 -n` options to the command line options, and will define the `RELEASE` define.

STRIP If this variable is defined, this will add the `-Xs` option to the command line options.

VERBOSE If this variable is defined, then `-vnwi` will be added to the command line options.

E.7 Variables set by fpcmake

The makefile generated by fpcmake contains a lot of makefile variables. fpcmake will write all of the keys in the makefile.fpc as makefile variables in the form `SECTION_KEYNAME`. This means that the following section:

```
[package]
name=mysql
version=1.0.5
```

will result in the following variable definitions:

```
override PACKAGE_NAME=mysql
override PACKAGE_VERSION=1.0.5
```

Most targets and rules are constructed using these variables. They will be listed below, together with other variables that are defined by fpcmake.

The following sets of variables are defined:

- Directory variables.
- Program names.
- File extensions.
- Target files.

Each of these sets is discussed in the subsequent:

E.7.1 Directory variables

The following compiler directories are defined by the makefile:

BASEDIR Is set to the current directory if the pwd command is available. If not, it is set to '.'.

COMPILER_INCDIR Is a space-separated list of include file paths. Each directory in the list is prepended with `-Fi` and added to the compiler options. Set by the `incdir` keyword in the `Compiler` section.

COMPILER_LIBDIR Is a space-separated list of library paths. Each directory in the list is prepended with `-Fl` and added to the compiler options. Set by the `libdir` keyword in the `Compiler` section.

COMPILER_OBJDIR Is a list of object file directories, separated by spaces. Each directory in the list is prepended with `-Fo` and added to the compiler options. Set by the `objdir` keyword in the `Compiler` section.

COMPILER_TARGETDIR This directory is added as the output directory of the compiler, where all units and executables are written, i.e. it gets `-FE` prepended. It is set by the `targetdir` keyword in the `Compiler` section.

COMPILER_TARGETUNITDIR If set, this directory is added as the output directory of the compiler, where all units and executables are written, i.e. it gets `-FU` prepended.It is set by the `targetdir` keyword in the `Dirs` section.

COMPILER_UNITDIR Is a list of unit directories, separated by spaces. Each directory in the list is prepended with `-Fu` and is added to the compiler options. Set by the `unitdir` keyword in the `Compiler` section.

GCCLIBDIR (LINUX only) Is set to the directory where libgcc.a is. If `needgcclib` is set to `True` in the `Libs` section, then this directory is added to the compiler command line with `-Fl`.

OTHERLIBDIR Is a space-separated list of library paths. Each directory in the list is prepended with `-Fl` and added to the compiler options. If it is not defined on linux, then the contents of the /etc/ld.so.conf file is added.

The following directories are used for installs:

INSTALL_BASEDIR Is the base for all directories where units are installed. By default, On LINUX, this is set to `$(INSTALL_PREFIX)/lib/fpc/$(RELEASEVER)`.
On other systems, it is set to `$(PREFIXINSTALLDIR)`. You can also set it with the `basedir` variable in the `Install` section.

INSTALL_BINDIR Is set to `$(INSTALL_BASEDIR)/bin` on LINUX, and `$(INSTALL_BASEDIR)/bin/$(OS_TARGET)` on other systems. This is the place where binaries are installed.

INSTALL_DATADIR The directory where data files are installed. Set by the `Data` key in the `Install` section.

INSTALL_LIBDIR Is set to `$(INSTALL_PREFIX)`/lib on LINUX,
and `$(INSTALL_UNITDIR)` on other systems.

INSTALL_PREFIX Is set to /usr/local on LINUX, /pp on DOS or WINDOWS NT. Set by the `prefix` keyword in the `Install` section.

INSTALL_UNITDIR Is where units will be installed. This is set to `$(INSTALL_BASEDIR)`/units/`$(OS_TARGET)`. If the units are compiled as a package, `$(PACKAGE_NAME)` is added to the directory.

E.7.2 Target variables

The second set of variables controls the targets that are constructed by the makefile. They are created by fpcmake, so you can use them in your rules, but you shouldn't assign values to them yourself.

TARGET_DIRS This is the list of directories that make will descend into when compiling. Set by the `Dirs` key in the `Target` section?

TARGET_EXAMPLES The list of examples programs that must be compiled. Set by the `examples` key in the `Target` section.

TARGET_EXAMPLEDIRS The list of directories that make will descend into when compiling examples. Set by the `exampledirs` key in the `Target` section.

TARGET_LOADERS Is a list of space-separated names that identify loaders to be compiled. This is mainly used in the compiler's RTL sources. It is set by the `loaders` keyword in the `Targets` section.

TARGET_PROGRAMS This is a list of executable names that will be compiled. the makefile appends `$(EXEEXT)` to these names. It is set by the `programs` keyword in the `Target` section.

TARGET_UNITS This is a list of unit names that will be compiled. The makefile appends `$(PPUEXT)` to each of these names to form the unit file name. The sourcename is formed by adding `$(PASEXT)`. It is set by the `units` keyword in the `Target` section.

ZIPNAME Is the name of the archive that will be created by the makefile. It is set by the `zipname` keyword in the `Zip` section.

ZIPTARGET Is the target that is built before the archive is made. This target is built first. If successful, the zip archive will be made. It is set by the `ziptarget` keyword in the `Zip` section.

E.7.3 Compiler command line variables

The following variables control the compiler command line:

CPU_SOURCE The source CPU type is added as a define to the compiler command line. This is determined by the Makefile itself.

CPU_TARGET The target CPU type is added as a define to the compiler command line. This is determined by the Makefile itself.

OS_SOURCE What platform the makefile is used on. Detected automatically.

OS_TARGET What platform will be compiled for. Added to the compiler command line with a `-T` prepended.

E.7.4 Program names

The following variables are program names, used in makefile targets.

AS The assembler. Default set to as.

COPY A file copy program. Default set to cp -fp.

COPYTREE A directory tree copy program. Default set to cp -frp.

CMP A program to compare files. Default set to `cmp`.

DEL A file removal program. Default set to rm -f.

DELTREE A directory removal program. Default set to rm -rf.

DATE A program to display the date.

DIFF A program to produce diff files.

ECHO An echo program.

FPC The Free Pascal compiler executable. Default set to `ppc386.exe`

INSTALL A program to install files. Default set to install -m 644 on LINUX.

INSTALLEXE A program to install executable files. Default set to install -m 755 on LINUX.

LD The linker. Default set to ld.

LDCONFIG (LINUX only) The program used to update the loader cache.

MKDIR A program to create directories if they don't exist yet. Default set to install -m 755 -d

MOVE A file move program. Default set to mv -f

PP The Free Pascal compiler executable. Default set to `ppc386.exe`

PPAS The name of the shell script created by the compiler if the `-s` option is specified. This command will be executed after compilation, if the `-s` option was detected among the options.

PPUMOVE The program to move units into one big unit library.

PWD The pwd program.

SED A stream-line editor program. Default set to sed.

UPX An executable packer to compress your executables into self-extracting compressed executables.

ZIPPROG A zip program to compress files. Zip targets are made with this program.

E.7.5 File extensions

The following variables denote extensions of files. These variables include the . (dot) of the extension. They are appended to object names.

ASMEXT Is the extension of assembler files produced by the compiler.

LOADEREXT Is the extension of the assembler files that make up the executable startup code.

OEXT Is the extension of the object files that the compiler creates.

PACKAGESUFFIX Is a suffix that is appended to package names in zip targets. This serves so packages can be made for different OSes.

PPLEXT Is the extension of shared library unit files.

PPUEXT Is the extension of default units.

RSTEXT Is the extension of the .rst resource string files.

SHAREDLIBEXT Is the extension of shared libraries.

SMARTEXT Is the extension of smartlinked unit assembler files.

STATICLIBEXT Is the extension of static libraries.

E.7.6 Target files

The following variables are defined to make targets and rules easier:

COMPILER Is the complete compiler command line, with all options added, after all Makefile variables have been examined.

DATESTR Contains the date.

UNITPPUFILES A list of unit files that will be made. This is just the list of unit objects, with the correct unit extension appended.

E.8 Rules and targets created by fpcmake

The `makefile.fpc` defines a series of targets, which can be called by your own targets. They have names that resemble default names (such as 'all', 'clean'), only they have `fpc_` prepended.

E.8.1 Pattern rules

The makefile makes the following pattern rules:

units How to make a pascal unit form a pascal source file.

executables How to make an executable from a pascal source file.

object file How to make an object file from an assembler file.

E.8.2 Build rules

The following build targets are defined:

fpc_all Builds all units and executables as well as loaders. If `DEFAULTUNITS` is defined, executables are excluded from the targets.

fpc_debug The same as `fpc_all`, only with debug information included.

fpc_exes Make all executables in `EXEOBJECTS`.

fpc_loaders Make all files in `LOADEROBJECTS`.

fpc_packages Make all packages that are needed to make the files.

fpc_shared Make all units as dynamic libraries.

fpc_smart Make all units as smartlinked units.

fpc_units Make all units in `UNITOBJECTS`.

E.8.3 Cleaning rules

The following cleaning targets are defined:

fpc_clean Cleans all files that result when `fpc_all` was made.

fpc_distclean Is the same as both previous target commands, but also deletes all object, unit and assembler files that are present.

E.8.4 Archiving rules

The following archiving targets are defined:

fpc_zipdistinstall Make a distribution install of the package.

fpc_zipinstall Make an install zip of the compiled units of the package.

fpc_zipexampleinstall Make a zip of the example files.

fpc_zipsourceinstall Make a zip of the source files.

The zip is made uzing the `ZIPEXE` program. Under LINUX, a .tar.gz file is created.

E.8.5 Installation rules

fpc_distinstall Target which calls the `install` and `exampleinstall` targets.

fpc_install Install the units.

fpc_sourceinstall Install the sources, in case a distribution is made.

fpc_exampleinstall Install the examples, in case a distribution is made.

E.8.6 Informative rules

There is only one target which produces information about the used variables, rules and targets: `fpc_info`.

The following information about the makefile is presented:

- general Configuration information: the location of the makefile, the compiler version, target OS, CPU.
- The directories, used by the compiler.
- All directories where files will be installed.
- All objects that will be made.
- All defined tools.

Appendix F

Compiling the compiler

F.1 Introduction

The Free Pascal team releases at intervals a completely prepared package, with compiler and units all ready to use, the so-called releases. After a release, work on the compiler continues, bugs are fixed and features are added. The Free Pascal team doesn't make a new release whenever they change something in the compiler, instead the sources are available for anyone to use and compile. There is an automated process that creates compiled versions of RTL and compiler are also made daily, and put on the web (if the build succeeds). Zip files with the sources are also created daily.

There are, nevertheless, circumstances when the compiler must be recompiled manually. When changes are made to compiler code, or when the compiler is downloaded through Subversion.

There are essentially 2 ways of recompiling the compiler: by hand, or using the makefiles. Each of these methods will be discussed.

F.2 Before starting

To compile the compiler easily, it is best to keep the following directory structure (a base directory of /pp/src is supposed, but that may be different):

```
/pp/src/Makefile
       /makefile.fpc
       /rtl/linux
           /inc
           /i386
           /...
       /compiler
```

When the makefiles should be used, the above directory tree must be used.

The compiler and rtl source are zipped in such a way that when both are unzipped in the same directory (/pp/src in the above) the above directory tree results.

There are 2 ways to start compiling the compiler and RTL. Both ways must be used, depending on the situation. Usually, the RTL must be compiled first, before compiling the compiler, after which the compiler is compiled using the current compiler. In some special cases the compiler must be compiled first, with a previously compiled RTL.

How to decide which should be compiled first? In general, the answer is that the RTL should be compiled first. There are 2 exceptions to this rule:

1. The first case is when some of the internal routines in the RTL have changed, or if new internal routines appeared. Since the OLD compiler doesn't know about these changed internal routines, it will emit function calls that are based on the old compiled RTL, and hence are not correct. Either the result will not link, or the binary will give errors.
2. The second case is when something is added to the RTL that the compiler needs to know about: a new default assembler mechanism, for example.

How to know if one of these things has occurred? There is no way to know, except by mailing the Free Pascal team. When the compiler cannot be recompiled when first compiling the RTL, then try the other way.

F.3 Compiling using make

When compiling with `make` it is necessary to have the above directory structure. Compiling the compiler is achieved with the target `cycle`.

Under normal circumstances, recompiling the compiler is limited to the following instructions (assuming you start in directory /pp/src):

```
cd compiler
make cycle
```

This will work only if the makefile is installed correctly and if the needed tools are present in the `PATH`. Which tools must be installed can be found in appendix E.

The above instructions will do the following:

1. Using the current compiler, the RTL is compiled in the correct directory, which is determined by the OS. e.g. under LINUX, the RTL is compiled in directory rtl/linux.
2. The compiler is compiled using the newly compiled RTL. If successful, the newly compiled compiler executable is copied to a temporary executable.
3. Using the temporary executable from the previous step, the RTL is re-compiled.
4. Using the temporary executable and the newly compiled RTL from the last step, the compiler is compiled again.

The last two steps are repeated 3 times, until three passes have been made or until the generated compiler binary is equal to the binary it was compiled with. This process ensures that the compiler binary is correct.

Compiling for another target: When compiling the compiler for another target, it is necessary to specify the `OS_TARGET` makefile variable. It can be set to the following values: `win32`, `go32v2`, `os2` and `linux`. As an example, cross-compilation for the go32v2 target from the win32 target is chosen:

```
cd compiler
make cycle OS_TARGET=go32v2
```

This will compile the go32v2 RTL, and compile a `go32v2` compiler.

When compiling a new compiler and the compiler should be compiled using an existing compiled RTL, the `all` target must be used, and another RTL directory than the default (which is the ../rtl/$(OS_TARGET) directory) must be indicated. For instance, assuming that the compiled RTL units are in `/pp/rtl/units/i386-linux`, typing

```
cd compiler
make clean
make all UNITDIR=/pp/rtl/units/i386-linux
```

should use the RTL from the `/pp/rtl/units/i386-linux` directory.

This will then compile the compiler using the RTL units in /pp/rtl/units/i386-linux. After this has been done, the 'make cycle' can be used, starting with this compiler:

```
make cycle PP=./ppc386
```

This will do the `make cycle` from above, but will start with the compiler that was generated by the `make all` instruction.

In all cases, many options can be passed to `make` to influence the compile process. In general, the makefiles add any needed compiler options to the command line, so that the RTL and compiler can be compiled. Additional options (e.g. optimization options) can be specified by passing them in `OPT`.

F.4 Compiling by hand

Compiling by hand is difficult and tedious, but can be done. The compilation of RTL and compiler will be treated separately.

F.4.1 Compiling the RTL

To recompile the RTL, so a new compiler can be built, at least the following units must be built, in the order specified:

loaders The program stubs, that are the startup code for each pascal program. These files have the .as extension, because they are written in assembler. They must be assembled with the GNU as assembler. These stubs are in the OS-dependent directory, except for LINUX, where they are in a processor dependent subdirectory of the LINUX directory (i386 or m68k).

system The system unit. This unit resides in the OS-dependent subdirectories of the RTL.

strings The strings unit. This unit resides in the inc subdirectory of the RTL.

dos The dos unit. It resides in the OS-dependent subdirectory of the RTL. Possibly other units will be compiled as a consequence of trying to compile this unit (e.g. on LINUX, the linux unit will be compiled, on go32, the go32 unit will be compiled).

objects The objects unit. It resides in the inc subdirectory of the RTL.

To compile these units on a i386, the following statements will do:

```
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/objects.pp
```

These are the minimum command line options, needed to compile the RTL.

For another processor, the `i386` should be changed into the appropriate processor. For another target OS, the target OS setting (`-T`) must be set accordingly.

Depending on the target OS there are other units that can be compiled, but which are not strictly needed to recompile the compiler. The following units are available for all plaforms:

objpas Needed for Delphi mode. Needs `-Mobjfpc` as an option. Resides in the objpas subdirectory.

sysutils Many utility functions, like in Delphi. Resides in the objpas directory, and needs `-MObjfpc` to compile.

typinfo Functions to access RTTI information, like Delphi. Resides in the objpas directory.

math Math functions like in Delphi. Resides in the objpas directory.

mmx Extensions for MMX class Intel processors. Resides in in the i386 directory.

getopts A GNU compatible getopts unit. Resides in the inc directory.

heaptrc To debug the heap. Resides in the inc directory.

F.4.2 Compiling the compiler

Compiling the compiler can be done with one statement. It's always best to remove all units from the compiler directory first, so something like

```
rm *.ppu *.o
```

on LINUX, and on DOS

```
del *.ppu
del *.o
```

After this, the compiler can be compiled with the following command line:

```
ppc386 -Tlinux  -Fusystems -Fii386/ -Fui386 \
   -Fu../rtl/units/i386-linux -di386 -dGDB pp.pas
```

All the options must be given on a single line, the backslash is needed if you want to specify the options on multiple lines.

So, the minimum options are:

1. The target OS. Can be skipped when compiling for the same target as the compiler which is being used.
2. A path to an RTL. Can be skipped if a correct fpc.cfg configuration is on the system. If the compiler should be compiled with the RTL that was compiled first, this should be ../rtl/OS (replace the OS with the appropriate operating system subdirectory of the RTL).
3. A define with the processor for which the compiler is compiled for. Required.
4. `-dGDB`. Required.
5. `-Sg` is needed, some parts of the compiler use `goto` statements (to be specific: the scanner).
6. A directory with some units and include files for the processor you're compiling for.
7. A directory with the system definitions.

So the absolute minimal command line is

```
ppc386 -Fusystems -Fii386 -Fui386 -di386 -dGDB -Sg pp.pas
```

Some other command line options can be used, but the above are the minimum. A list of recognised options can be found in table (F.1).

Table F.1: Possible defines when compiling FPC

Define	does what
GDB	Support of the GNU Debugger (required switch).
I386	Generate a compiler for the Intel i386+ processor family.
M68K	Generate a compiler for the M680x0 processor family.
X86_64	Generate a compiler for the AMD64 processor family.
POWERPC	Generate a compiler for the PowerPC processor family.
POWERPC64	Generate a compiler for the 64-bit PowerPC processor family.
ARM	Generate a compiler for the Intel ARM processor family.
SPARC	Generate a compiler for the SPARC processor family.
EXTDEBUG	Some extra debug code is executed.
MEMDEBUG	Some memory usage information is displayed.
SUPPORT_MMX	only i386: enables the compiler switch `MMX` which allows the compiler to generate MMX instructions.
EXTERN_MSG	Don't compile the msgfiles in the compiler, always use external messagefiles.
NOOPT	Do not include the optimizer in the compiler.
CMEM	Use the C memory manager.

This list may be subject to change, the source file pp.pas always contains an up-to-date list.

Appendix G

Compiler defines during compilation

This appendix describes the possible defines when compiling programs using Free Pascal. A brief explanation of the define, and when it is used is also given.

Table G.1: Possible defines when compiling using FPC

Define	description
FPC_LINK_DYNAMIC	Defined when the output will be linked dynamically. This is defined when using the -XD compiler switch.
FPC_LINK_STATIC	Defined when the output will be linked statically. This is the default mode.
FPC_LINK_SMART	Defined when the output will be smartlinked. This is defined when using the -XX compiler switch.
FPC_PROFILE	Defined when profiling code is added to program. This is defined when using the -pg compiler switch.
FPC_CROSSCOMPILING	Defined when the target OS/CPU is different from the source OS/CPU.
FPC	Always defined for Free Pascal.
VER2	Always defined for Free Pascal version 2.x.x.
VER2_0	Always defined for Free Pascal version 2.0.x.
VER2_2	Always defined for Free Pascal version 2.2.x.
FPC_VERSION	Contains the major version number from FPC.
FPC_RELEASE	Contains the minor version number from FPC.
FPC_PATCH	Contains the third part of the version number from FPC.
FPC_FULLVERSION	Contains the entire version number from FPC as a single number which can be used for comparing. For FPC 2.2.4 it will contain 20204.
ENDIAN_LITTLE	Defined when the Free Pascal target is a little-endian processor (80x86, Alpha, ARM).
ENDIAN_BIG	Defined when the Free Pascal target is a big-endian processor (680x0, PowerPC, SPARC, MIPS).
FPC_DELPHI	Free Pascal is in Delphi mode, either using compiler switch -MDelphi or using the `$MODE DELPHI` directive.
FPC_OBJFPC	Free Pascal is in OBJFPC mode, either using compiler switch -Mobjfpc or using the `$MODE OBJFPC` directive.
FPC_TP	Free Pascal is in Turbo Pascal mode, either using compiler switch -Mtp or using the `$MODE TP` directive.
FPC_GPC	Free Pascal is in GNU Pascal mode, either using compiler switch -SP or using the `$MODE GPC` directive.

Remark: The `ENDIAN_LITTLE` and `ENDIAN_BIG` defines were added starting from Free Pascal version 1.0.5.

Table G.2: Possible CPU defines when compiling using FPC

Define	When defined?
CPU86	Free Pascal target is an Intel 80x86 or compatible.
CPU87	Free Pascal target is an Intel 80x86 or compatible.
CPU386	Free Pascal target is an Intel 80386 or later.
CPUI386	Free Pascal target is an Intel 80386 or later.
CPU68K	Free Pascal target is a Motorola 680x0 or compatible.
CPUM68K	Free Pascal target is a Motorola 680x0 or compatible.
CPUM68020	Free Pascal target is a Motorola 68020 or later.
CPU68	Free Pascal target is a Motorola 680x0 or compatible.
CPUSPARC32	Free Pascal target is a SPARC v7 or compatible.
CPUSPARC	Free Pascal target is a SPARC v7 or compatible.
CPUALPHA	Free Pascal target is an Alpha AXP or compatible.
CPUPOWERPC	Free Pascal target is a 32-bit or 64-bit PowerPC or compatible.
CPUPOWERPC32	Free Pascal target is a 32-bit PowerPC or compatible.
CPUPOWERPC64	Free Pascal target is a 64-bit PowerPC or compatible.
CPUX86_64	Free Pascal target is a AMD64 or Intel 64-bit processor.
CPUAMD64	Free Pascal target is a AMD64 or Intel 64-bit processor.
CPUIA64	Free Pascal target is a Intel itanium 64-bit processor.
CPUARM	Free Pascal target is an ARM 32-bit processor.
CPUAVR	Free Pascal target is an AVR 16-bit processor.
CPU16	Free Pascal target is a 16-bit CPU.
CPU32	Free Pascal target is a 32-bit CPU.
CPU64	Free Pascal target is a 64-bit CPU.

Table G.3: Possible FPU defines when compiling using FPC

Define	When defined?
FPUSOFT	Software emulation of FPU (all types).
FPUSSE64	SSE64 FPU on Intel I386 and higher, AMD64.
FPUSSE	SSE instructions on Intel I386 and higher.
FPUSSE2	SSE 2 instructions on Intel I386 and higher.
FPUSSE3	SSE 3 instructions on Intel I386 and higher, AMD64.
FPULIBGCC	GCC library FPU emulation on ARM and M68K.
FPU68881	68881 on M68K.
FPUFPA	FPA on ARM.
FPUFPA10	FPA 10 on ARM.
FPUFPA11	FPA 11 on ARM.
FPUVFP	VFP on ARM.
FPUX87	X87 FPU on Intel I386 and higher.
FPUITANIUM	On Intel Itanium.
FPUSTANDARD	On PowerPC (32/64 bit).
FPUHARD	On Sparc.

Table G.4: Possible defines when compiling using target OS

Target operating system	Defines
linux	LINUX, UNIX
freebsd	FREEBSD, BSD, UNIX
netbsd	NETBSD, BSD, UNIX
sunos	SUNOS, SOLARIS, UNIX
go32v2	GO32V2, DPMI
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

Remark: The UNIX define was added starting from Free Pascal version 1.0.5. The BSD operating systems no longer define LINUX starting with version 1.0.7.