



Pascal Analyzer

Copyright © 2001-2023 Peganza

Pascal Analyzer

by Peganza

Pascal Analyzer parses Delphi or Borland Pascal source code and produces reports that help you understand your source code better.

Pascal Analyzer

Copyright © 2001-2023 Peganza

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Table of Contents

Foreword	0
Introduction	7
Known limitations	13
What's new in version 9 (April 2017, updated November 2023)?	15
What's new in version 8 (May 2016)?	31
What's new in version 7 (November 2013)?	35
What's new in version 6 (August 2011)?	38
What's new in version 5 (May 2010)?	42
What's new in version 4 (October 2006)?	45
Command-Line Options for PAL.EXE and PAL32.EXE	48
How to use PAL.EXE and PAL32.EXE	49
How to use PALCMD.EXE and PALCMD32.EXE	51
Installation folders	57
Main window	58
Reports	64
1 General Reports.....	65
Status Report	66
Strong Warnings Report	67
Warnings Report	71
Memory Report	92
Optimization Report	95
Code Reduction Report	100
Convention Compliance Report	109
Inconsistent Case Report	115
Prefix Report	116
NextGen Readiness Report	116
2 Metrics Reports.....	117

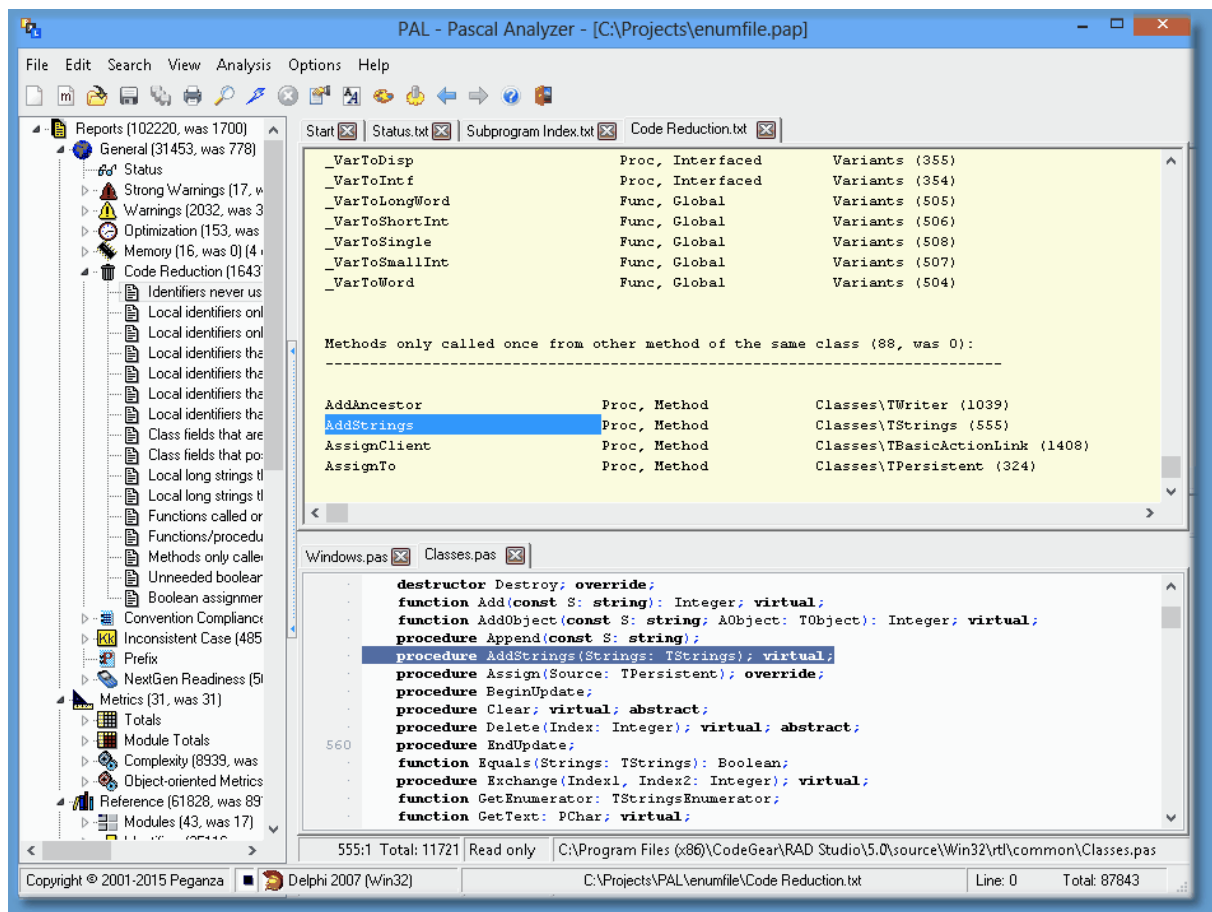
Totals Report	117
Module Totals Report	118
Complexity Report	119
Object-oriented Metrics Report	122
3 Reference Reports.....	125
Modules Report	126
Identifiers Report	127
Duplicate Identifiers Report	128
Similarity Report	128
Literal Strings/Numbers Report	129
Subprogram Index Report	130
Bindings Report	130
Third-party Dependencies Report	131
Most Called Report	132
Call Tree Report	132
Reverse Call Tree Report	133
Call Index Report	134
Exception Report	134
Brief Cross-reference Report	135
Cross-reference Report	135
Used Outside Report	137
Uses Report	137
Conditional Symbols Report	140
Directives Report	141
To-Do Report	142
Module Call Tree Report	143
Help Report	143
Searched Strings Report	144
Map File Report	145
Clone Report	146
4 Class Reports.....	149
Class Index Report	149
Class Summary Report	150
Class Hierarchy Report	150
Class Field Access Report	151
5 Control Reports.....	151
Control Index Report	151
Control Alignment Report	152
Control Size Report	152
Control Tab Order Report	153
Control Warnings Report	154
Property Value Report	155
Missing Property Report	156
Form Report	156
Events Report	157
Main menu	158
1 File menu	158
2 Edit menu	160
3 Search menu.....	161
4 View menu.....	162

5	Analysis menu	164
6	Options menu	165
	Properties - General	167
	Properties - Reports	172
	Properties - Format	176
	Properties - Parser	183
	Properties - Switches	191
	Preferences - General	193
	Preferences - Source Code	196
	Preferences - Editor	199
	Help menu	203
	Index	204

1 Introduction

Pascal Analyzer, or PAL for short, is a utility program that analyzes, documents, debugs, and helps you optimize your source code. Pascal Analyzer makes a static code analysis. It only needs the source code, unlike other similar tools that perform an analysis of the running program. We think that PAL will help you better understand your code and support you in producing code of higher quality, consistency, and reliability.

PAL quickly pays itself back in easier maintenance, less errors and improved quality, not only during development, but also throughout the entire life cycle of your code.



The main window in Pascal Analyzer

On a 64-bits computer, PAL will install both the main new 64-bits version and the 32-bits version. The 32-bits version will be installed in a separate subdirectory.

[PAL.EXE](#)

– the main 64-bits Windows program with a friendly user interface

[PALCMD.EXE](#)

– a 64-bits command-line analyzer

[PAL32.EXE](#)

– 32-bits version

[PALCMD32.EXE](#)

– 32-bits version

On a 32-bits computer, only the 32-bits versions will be installed.

The command-line analyzer produces exactly the same reports as the GUI version. You will however most often use the GUI in PAL.EXE (or PAL32.EXE). For running analyses in batch mode, or to integrate into a build process, use PALCMD.EXE (or PALCMD32.EXE).

There are also Delphi IDE plug-ins, PALWIZ.DLL (or PALWIZx.DLL, depending on Delphi version), to enable the RAD Studio IDE to load a source file when double-clicking on a report line in Pascal Analyzer. The plug-in is automatically installed and enabled. You can enable/disable the plug-in from the [Preferences](#) dialog.

The plugins work with Delphi 5 and later versions.

Pascal Analyzer functions with Pascal/Delphi Compilers from BP7 and later:

- Borland Pascal 7 (or earlier)
- Delphi 1
- Delphi 2
- Delphi 3
- Delphi 4
- Delphi 5
- Delphi 6
- Delphi 7
- Delphi 8 for .NET
- Delphi 2005 for Win32
- Delphi 2005 for .NET
- Delphi 2006 for Win32
- Delphi 2006 for .NET
- Delphi 2007 for Win32
- Delphi 2007 for .NET
- Delphi 2009 for Win32
- Delphi 2010 for Win32
- Delphi XE for Win32
- Delphi XE2 for Win32
- Delphi XE2 for Win64
- Delphi XE2 for OSX
- Delphi XE3 for Win32
- Delphi XE3 for Win64
- Delphi XE3 for OSX
- Delphi XE4 for Win32
- Delphi XE4 for Win64
- Delphi XE4 for OSX
- Delphi XE4 for iOS Device
- Delphi XE4 for iOS Simulator
- Delphi XE5 for Win32
- Delphi XE5 for Win64
- Delphi XE5 for OSX

- Delphi XE5 for iOS Device
- Delphi XE5 for iOS Simulator
- Delphi XE5 for Android

- Delphi XE6 for Win32
- Delphi XE6 for Win64
- Delphi XE6 for OSX
- Delphi XE6 for iOS Device
- Delphi XE6 for iOS Simulator
- Delphi XE6 for Android

- Delphi XE7 for Win32
- Delphi XE7 for Win64
- Delphi XE7 for OSX
- Delphi XE7 for iOS Device
- Delphi XE7 for iOS Simulator
- Delphi XE7 for Android

- Delphi XE8 for Win32
- Delphi XE8 for Win64
- Delphi XE8 for OSX
- Delphi XE8 for iOS Device 32-bits
- Delphi XE8 for iOS Device 64-bits
- Delphi XE8 for iOS Simulator
- Delphi XE8 for Android

- Delphi 10 for Win32
- Delphi 10 for Win64
- Delphi 10 for OSX
- Delphi 10 for iOS Device 32-bits
- Delphi 10 for iOS Device 64-bits
- Delphi 10 for iOS Simulator
- Delphi 10 for Android

- Delphi 10.1 for Win32
- Delphi 10.1 for Win64
- Delphi 10.1 for OSX
- Delphi 10.1 for iOS Device 32-bits
- Delphi 10.1 for iOS Device 64-bits
- Delphi 10.1 for iOS Simulator
- Delphi 10.1 for Android

- Delphi 10.2 for Win32
- Delphi 10.2 for Win64
- Delphi 10.2 for OSX
- Delphi 10.2 for iOS Device 32-bits
- Delphi 10.2 for iOS Device 64-bits
- Delphi 10.2 for iOS Simulator
- Delphi 10.2 for Android
- Delphi 10.2 for Linux 64-bits

- Delphi 10.3 for Win32

- Delphi 10.3 for Win64
 - Delphi 10.3 for OSX 32-bits
 - Delphi 10.3 for OSX 64-bits
 - Delphi 10.3 for iOS Device 32-bits
 - Delphi 10.3 for iOS Device 64-bits
 - Delphi 10.3 for iOS Simulator
 - Delphi 10.3 for Android 32-bits
 - Delphi 10.3 for Android 64-bits
 - Delphi 10.3 for Linux 64-bits
-
- Delphi 10.4 for Win32
 - Delphi 10.4 for Win64
 - Delphi 10.4 for OSX 32-bits
 - Delphi 10.4 for OSX 64-bits
 - Delphi 10.4 for iOS Device 32-bits
 - Delphi 10.4 for iOS Device 64-bits
 - Delphi 10.4 for iOS Simulator
 - Delphi 10.4 for Android 32-bits
 - Delphi 10.4 for Android 64-bits
 - Delphi 10.4 for Linux 64-bits
-
- Delphi 11 for Win32
 - Delphi 11 for Win64
 - Delphi 11 for OSX 32-bits
 - Delphi 11 for OSX 64-bits
 - Delphi 11 for iOS Device 32-bits
 - Delphi 11 for iOS Device 64-bits
 - Delphi 11 for iOS Simulator
 - Delphi 11 for Android 32-bits
 - Delphi 11 for Android 64-bits
 - Delphi 11 for Linux 64-bits
 - Delphi 11 for OSX ARM 64-bits
-
- Delphi 12 for Win32
 - Delphi 12 for Win64
 - Delphi 12 for OSX 32-bits
 - Delphi 12 for OSX 64-bits
 - Delphi 12 for iOS Device 32-bits
 - Delphi 12 for iOS Device 64-bits
 - Delphi 12 for iOS Simulator
 - Delphi 12 for Android 32-bits
 - Delphi 12 for Android 64-bits
 - Delphi 12 for Linux 64-bits
 - Delphi 12 for OSX ARM 64-bits

PAL parses your source code in the same way as the compiler. It builds large data tables in memory and when the parsing is finished, produces an assortment of reports. These reports hold plenty of useful information that can help you error-proof your applications.

Be forewarned that PAL occasionally needs a lot of memory (RAM). The amount of memory needed is proportional to the number of code lines and modules in the examined project.

In addition to common cross-reference reports, PAL produces reports that show which units are used, which identifiers are unused and so on. It also calculates industry standard metrics such as lines of code (LOC) and decision points (DP), and much more..

Projects

To analyze a particular set of source code with Pascal Analyzer, you must first create a project. Do not confuse a Pascal Analyzer project with a Delphi project, they are completely different things. The project holds the options for the analysis and lets you conveniently use separate options for different sets of source code. Projects are saved as text files with the extension "pap", like for example a file with the name MyProj.pap. The format of the files is equivalent to that of an INI file. It is possible to inspect and edit the project files in a normal text editor, although not recommended.

Multi-projects

Multi-projects are essentially collections of Pascal Analyzer projects (see above). In a way they are similar to Delphi project groups (*.bpg-files), in that they both reference other projects. When a multi-project is run, the included projects are analyzed sequentially. The reports that are generated contain mutual facts about these projects. Multi-projects are saved as text files with the extension "pam", like for example a file with the name MyMProj.pam. The format of the files is equivalent to that of an INI file.

A subset of all reports and sections are generated for a multi-project:

Warnings Report

- Interfaced identifiers that are used, but not outside of unit
- Interfaced class identifiers that are public/published, but not used outside of unit

Optimization Report

- Virtual methods (procedures/functions) that are not overridden

Reduction Report

- Identifiers never used
- Functions called only as procedures (result ignored)
- Functions/procedures (methods excluded) only called once
- Methods only called once from other method of the same class

Uses Report

- Units used by the projects
- Units used by all projects
- Unit references

See also:

[What's new in version 9?](#)

[What's new in version 8?](#)

[How to use PAL.EXE](#)

[How to use PALCMD.EXE](#)

[Known limitations](#)

[Command Line Options for PAL.EXE](#)

[Main menu](#)

[Main window](#)

Copyright © Peganza 2001-2023. All rights reserved. All product names are trademarks or registered trademarks of their respective owners.

Web site: <https://www.peganza.com>

Email: support@peganza.com

2 Known limitations

There are situations that Pascal Analyzer currently cannot handle very well. Some of these limitations, but certainly not all, are:

1. Objects that are created through a class reference cannot always be resolved. The reason for this is that the actual class used is determined at runtime.

Example:

```
1  type
2    TMyClassRef = class of TMyClass;
3
4    TMyClass = class
5    ..
6      procedure Method; virtual;
7    end;
8
9    TMyDerivedClass = class(TMyClass)
10   ..
11     procedure Method; override;
12   end;
13   ..
14   ..
15
16   procedure Proc(C : TMyClassRef);
17   begin
18     C := TMyClassRef.Create; // TMyClass or TMyDerivedClass?
19     C.Method; // TMyClass.Method or TMyDerivedClass.Method?
20     ..
21   end;
```

2. Methods that are marked as abstract in a base class and used in that class, cannot be resolved:

Example:

```
1  type
2    TBase = class
3    ..
4      procedure Main;
5      procedure Proc; virtual; abstract;
6    end;
7
8    TDesc = class(TBase)
9    ..
10     procedure Proc; overload;
11   end;
12
13   TAnotherDesc = class(TBase)
14   ..
15     procedure Proc; overload;
16   end;
17
18   procedure TBase.Main;
19   begin
20     ..
21     Proc; // which one, TDesc.Proc or TAnotherDesc.Proc?
22   end;
```

The actual usage of Proc is determined at runtime.

3. Assert calls are not excluded from the parsing process, unlike in Delphi, regardless if the \$C- setting is active or not. This means that identifiers used in the Assert procedure

call, will be registered, and appear in the reports.

Example:

```
1  procedure MyProc(P : Pointer);  
2  begin  
3      Assert(P <> nil);  
4      ..  
5  end;
```

The parameter P will be registered and appear in the reports. When compiled by Delphi, this code line will be stripped out if \$C- is defined.

See also:

[Introduction](#)

[How to use PAL.EXE](#)

[How to use PALCMD.EXE](#)

[Command Line Options for PAL.EXE](#)

[Main menu](#)

[Main window](#)

3 What's new in version 9 (April 2017, updated November 2023)?

This article describes changes and new features in Pascal Analyzer version 9, compared to the previous version 8. The new version has been enhanced with a lot of smaller and greater features. As part of our continuing quality work, there are also numerous optimizations and error fixes.

There are in this version one new report, and 22 new report sections. The total number of sections is now 237. These sections are divided over 52 different reports.

Support for Delphi 10.2 Tokyo, Delphi 10.3 Rio, Delphi 10.4 Sydney, Delphi 11 Alexandria and Delphi 12 Athens added

Pascal Analyzer now also understands code for Delphi 10.2 Tokyo (released March 2017), including the new Linux 64-bits compiler target. Also it supports Delphi 10.3.3 Rio (released November 2019), and Delphi 10.4.2 Sydney (released February 2021), and Delphi 11 (released September 2021, latest update 11.3 in February 2023), including the new OSX ARM 64-bits compiler target. Plus the current Delphi 12 Athens major version.

New report: Clone Report (CLON)

This [new report](#) detects code clones. It compares subprograms within the same module and across modules.

Use the results to refactor your code and eliminate duplicated code.

New section in Strong Warnings Report: "Index error" (STWA4)

This section reports locations in your code where index errors can occur. These are errors where an array index with an invalid value is accessed. Some examples:

```
1 procedure Proc;  
2 var  
3   X : Integer;  
4   Arr : array[0..1] of Integer;  
5 begin  
6   X := 555;  
7   ..  
8   Arr[X-2] := 0; // index error  
9 end;
```

If the code had instead been written as "Arr[553]" (an explicit value), the compiler would have halted on this line. But for a variable, it does not.

```

1  var
2    Z : Integer;
3    Arr : array[0..4] of Integer;
4    I : Integer;
5  begin
6    Z := 3;
7
8    case I of
9      0 : Z := 555; // only this option gives index error below...
10     6 : Z := 4;
11   end;
12
13   Arr[Z] := 0; // index error!
14 end;

```

These kind of errors will give an exception at runtime, and should of course be avoided as much as ever possible.

New section in Strong Warnings Report: "Possible bad pointer usage" (STWA5)

This section lists locations in your code where a pointer possibly is misused. It is for example a pointer that has been set to nil and further down in the code is dereferenced.

Example:

```

1  type
2    TMyClass = class
3      procedure MyProc;
4    end;
5
6  procedure TMyClass.MyProc;
7  begin
8    ..
9  end;
10
11 procedure Proc;
12 var
13   Obj : TMyClass;
14 begin
15   Obj := TMyClass.Create;
16   ..
17   Obj := nil;
18   ..
19   Obj.MyProc; // error, Obj is nil here
20 end;

```

```

1  type
2    TMyClass = class
3      procedure MyProc;
4    end;
5
6  procedure TMyClass.MyProc;
7  begin
8    ..
9  end;
10
11 procedure Proc;
12 var
13   Obj : TMyClass;
14 begin
15   Obj := TMyClass.Create;
16   Obj := nil;
17   Obj.MyProc; // bad pointer!
18 end;

```



```

1  type
2    TMyRec = record
3      MyField : Integer;
4    end;
5
6  procedure Proc;
7  var
8    Ptr : ^TMyRec;
9    Rec : TMyRec;
10 begin
11   Ptr := nil;
12   Ptr^.MyField := 555; // bad pointer!
13 end;

```

New section in Strong Warnings Report: "Possible bad typecast (consider using "as" for objects)" (STWA6)

This section lists locations in your code with a possibly bad typecast. These are typecasts that casts into a type other than what the variable itself has. If you use the "as" operator, an exception will instead be raised. Otherwise there may be access violations and errors in a totally different code location, which is awfully hard to track down.

```

1  type
2    TFruit = class
3      ..
4    end;
5
6    TAnimal = class
7      ..
8    end;
9
10 procedure Proc;
11 var
12   Monkey : TAnimal;
13   Banana : TFruit;
14 begin
15   ..
16   Monkey := TAnimal(Banana); // bad typecast, a fruit cannot be typecast to an animal
17 end;

```

In the example above, the last line could better be written (although still faulty!) as

```
Monkey := Banana as TAnimal;
```

This should result in an exception. But it is still preferable; instead of letting the code proceed resulting maybe in access violations later in a totally unrelated part of the code, which is not fun to debug.

```

1  procedure Proc;
2  var
3    Ptr : Pointer;
4    X : Integer;
5  begin
6    X := Integer(Ptr); // bad typecast for 64-bits! Use NativeInt instead
7  end;

```

Added in 9.2:

New section in Strong Warnings Report: "For-loop with possible bad condition" (STWA7)

This section lists locations in your code where for loop has any of these conditions:

```
1  ..
2
3  for I := 0 to SL.Count do // SL.Count-1 intended?
4  begin
5  ..
6  end;
7
8  for I := 1 to SL.Count-1 do // SL.Count intended?
9  begin
10 end;
11
12 ..
13
```

Added in 9.2:

New section in Strong Warnings Report: "Bad parameter usage (same identifier passed for different parameters)" (STWA8)

This section lists locations in your code where a call to a subprogram is made with bad parameters. The situation occurs when the called subprogram has an "out" parameter plus at least one another parameter. The identifier passed is used for both these parameters. Because an "out"-parameter is cleared in the called subprogram this will give unexpected results for reference-counted variables like strings and dynamic arrays.

```
1  ..
2
3  procedure Proc(const Value : string; out ReturnValue : string);
4  begin
5      ReturnValue := '555'+Value;
6  end;
7
8  procedure Caller;
9  var
10     S : string;
11  begin
12     S := '111';
13     Proc(S, S); // S will have '555' upon return, not '555111' which you would expect
14  end;
15
16  ..
```

Added in 9.2:

New section in Strong Warnings Report: "Generic interface has GUID" (STWA9)

This section lists generic interface types that declare a GUID:

```

1  ..
2
3  type
4    IMyInterface<t> = interface
5      ['{C9BC756B-6B30-4C44-B237-552AFBA5697C}']
6      ..
7    end;
8
9  ..
10

```

The problem with this is that all generic types created from this interface, like `IMyInterface<Integer>` and `IMyInterface<string>` will share the same GUID. This will cause type casting to malfunction.

New section in Warnings Report: "Mixing interface variables and objects" (WARN53)

This section reports locations in your code with assignments between objects and interface variables. Normally, unless you really know what you are doing, it is a bad idea to mix interfaces and objects. The reason is that the reference counting mechanism of interfaces can be disturbed, leading to access violations and/or memory leaks.

```

1  type
2    IIntf = interface
3      end;
4
5    TIntf = class(TInterfacedObject, IIntf)
6      end;
7
8  procedure Proc;
9  var
10   Obj : IIntf;
11   X : TIntf;
12  begin
13   Obj := TIntf.Create;
14   X := TIntf(Obj); // not OK, mixes objects and interfaces
15   ..
16  end;

```

New section in Warnings Report: "Set before passed as out parameter" (WARN54)

This section reports locations in your code where a variable is set and then passed as an "out" parameter to a function.

Because the "out" parameter will be set in the called function without being read first, it is at least pointless to set it before it is passed. It may also indicate some misunderstanding about the code.

Consequently it is recommended to check if it is meaningful to set the variable before passing it. If not, remove the assignment, or else modify the signature of the called function from "out" to "var".

See also our [blog article](#) about out parameters.

Example:

```
1  procedure Proc(out Param : Integer);
2  begin
3    ..
4  end;
5
6  procedure Test;
7  var
8    I : Integer;
9  begin
10   ..
11   I := 555; // this is meaningless!
12   Proc(I);
13   ..
14 end;
```

New section in Warnings Report: "Redeclares ancestor member" (WARN55)

This section lists class fields or methods that redeclare ancestor members with the same name. This may lead to confusion about which member is referenced in a given situation. The recommendation is to refrain from reusing the same name, because this will only make your code harder to understand and maintain.

Example:

```
1  type
2    TAncestor = class
3      private
4        FObj : TObject;
5        procedure Proc;
6      end;
7
8    TDescendant = class(TAncestor)
9      protected
10       FObj : TObject; // redeclares!
11       procedure Proc; // redeclares!
12     end;
```

New section in Warnings Report: "Parameter to FreeAndNil is not an object" (WARN56)

This section reports locations in your code where FreeAndNil takes a parameter which is not an object, for example an interface variable. This may lead to access violations. Unlike Free, the compiler will not complain at compile-time.

Example:

```
1  type
2    IMyInterface = interface
3      ..
4    end;
5
6    TMyClass = class(TInterfacedObject, IMyInterface)
7    end;
8
9    procedure Proc;
10   var
11     Intf : IMyInterface;
12     Obj : TMyClass;
13   begin
14     Intf := TMyClass.Create;
15     FreeAndNil(Intf); // not OK!
16
17     Obj := TMyClass.Create;
18     FreeAndNil(Obj); // OK
19   end;
```

New section in Warnings Report: "Enumerated constant missing in case structure" (WARN57)

This section lists locations in your code where a case statement does not list all possible values of an enumerated type. This is probably most often as intended, but it may also point out an error in the code.

Example:

```
1  type
2    TChessPiece = (cpPawn, cpKnight, cpBishop, cpRook,
3                  cpQueen, cpKing);
4
5  procedure Move(Piece : TChessPiece;
6                FromSquare, ToSquare : TSquare);
7  begin
8    case Piece of
9      cpPawn : ..;
10     cpKnight : ..;
11     cpBishop : ..;
12     cpRook : ..;
13     cpQueen : ..;
14   end;
15
16   ..
17 end;
```

In the code above, **cpKing** is missing from the case structure, and will trigger a warning.

If you want to suppress warnings for a case-structure where you know it is safe to exclude one or more enumerated constants, just use the PALOFF feature on the same line as the "case" keyword.

New section in Warnings Report: "Mixed operator precedence levels" (WARN58)

This section lists locations in your code where operators of different levels are mixed. Operators are in Object Pascal evaluated from left to right, unless parentheses are used to change the evaluation order. Operators of level 1 are evaluated before operators of level 2 etc.

Those are the operator precedence levels, as used in the Object Pascal language:

Level 1: @, not

Level 2: *, /, div, mod, and, shl, shr, as

Level 3: +, -, or, xor

Level 4: =, <>, <, >, <=, >=, in, is

Example:

```
1 | X := A + B * 5;
2 | // evaluated as X := A + (B * 5)
3 |
4 | B := BoolA and BoolB or C;
5 | // evaluated as B := (BoolA and BoolB) or C
```

Mixing operators is perfectly valid but you will find that your code is clearer and easier to understand if you insert parentheses. Then you do not have to think that much about operator precedence.

New section in Warnings Report: "Explicit float comparison" (WARN59)

This section lists locations in your code where floating point numbers (variables, constants, or explicit values) are directly compared. It is considered not secure to compare floating numbers directly. Instead use functions in Delphi's [System.Math](#) unit, like [IsZero](#) and [SameValue](#).

Example:

```
1 | procedure CalculateProfit;
2 | var
3 |   Yield, Income : Double;
4 | begin
5 |   Yield := GetYield;
6 |   Income := GetIncome;
7 |
8 |   if Yield = Income then // not secure!
9 |   begin
10 |    ..
11 |   end;
12 | end;
```

In the example above, use instead SameValue function from System.Math unit.

Continuous parser and report improvements

Numerous bug fixes and minor improvements have been added in this major new release. Often those fixes are for the parser and the evaluation of identifiers, like improving handling of generics and overloads. Also when it comes to performance, many parts of the program now execute even quicker than before.

Loading old PAL projects

When loading old (before version 9.x) projects, those now automatically select report

sections that are new in version 9.x. The older handling was to not automatically add those new report sections. These had to be added manually.

Delphi project files (*.dproj) can now be loaded

These files can now directly be selected and loaded as the main file for a project. You can also select a main project file (*.dpr), with the same effect as in version 8.x, which was to implicitly also load the dproj file (when the setting "Use Delphi project options" was activated).

New command line parameter for PAL.EXE

You can now use an optional second parameter /AUTO in PAL.EXE (first parameter must as before give the path to the project) to automatically analyze the loaded project and then terminate the application. This parameter makes it easier to schedule (with Windows' Task Scheduler) analyses where PAL is automatically started and stopped.

Example of a command line:

PAL.EXE C:\proj\MyProj.pap /AUTO

Another option is of course to use the command line program PALCMD.EXE for these tasks.

Integration with Lattix

For the Uses Report, an additional file Lattix.xml is created in the report directory. You can use it to integrate with [Lattix](#) products.

New Defaults button when selecting report sections

When selecting report sections, there is a new button "Defaults". It selects the default sections when clicked. This is very convenient, for example when you have temporarily removed some sections, and want to reselect only the default ones.

Modified licensing model

Starting with this major version 9, Pascal Analyzer now uses a subscription based licensing model. Each license now includes a full year of updates and new releases. After that period, additional support plan periods can be bought. See the orders page at our web site for more details.

Added in 9.1:

New section in Warnings Report: "Condition evaluates to constant value" (WARN60)

This section lists locations in your code where a condition evaluates to a constant value.

Example:

```
1  procedure MyProc;
2  var
3      X : Integer;
4  begin
5      X := 0;
6
7      ..
8
9      if X+5 = 15 then // if-condition has always the same value ...
10     begin
11     end;
12 end;
13
```

Added in 9.2:

New section in Warnings Report: Assigned to itself (WARN61)

This section lists locations in your code where a variable has been assigned to itself. Even if this assignment is harmless, it makes no sense. It may indicate other problems with the code, so you should check the surrounding code.

Added in 9.3:

New section in Warnings Report: Possible orphan event handler (WARN62)

This section lists class procedures in your code that look like event handlers. But they are not connected to any control in the corresponding DFM-file.

New section in Code Reduction Report: "Consider using interface type" (REDU18)

This list contains objects which can be declared and implemented as an interface type, instead of as the class type implementing the interface. One major advantage is that interface reference counting can be used, so you will not have to explicitly free the object yourself.

```
1  type
2      TIntf = interface
3      ..
4      end;
5
6      TIntf = class(TInterfacedObject, TIntf)
7      ..
8      end;
9
10 var
11     Obj : TIntf; // consider using TIntf!
12     ..
13
14
```

In the code example above, consider instead declaring "Obj : TIntf" instead.

New section in Code Reduction Report: "Redundant parentheses" (REDU19)

This section lists locations in your code where superfluous parentheses can be removed, simplifying the code and making it easier to read.

```
1  procedure Proc;  
2  begin  
3    if (3+4) = 7 then ..;  
4  
5    if (((True))) then ..;  
6  
7    if (True) then ..;  
8  
9    if (3+4+(5)+(6)) = 11 then ..;  
10 end;  
11
```

New section in Code Reduction Report: "Common subexpression, consider elimination" (REDU20)

This section lists locations in your code with repeated common subexpressions. Those may be candidates to put into temporary variables in order to simplify and optimize the code.

Example:

```
1  procedure Proc;  
2  var  
3    A, B, C, D, E : Integer;  
4  begin  
5    ..  
6    E := A+B+C;  
7    ..  
8    D := A+B+C; // expression is repeated  
9    ..  
10 end;
```

In the code example above, consider putting the result of the expression "A+B+C" into a local temporary integer variable. If this really gives faster execution depends on how the compiler generates the machine code. But at least your code may become easier to read and understand.

If any of the variables involved in the repeated expressions would have been modified, between the locations, there should not be any warning.

New section in Code Reduction Report: "Default parameter values that can be omitted" (REDU21)

This list contains calls to functions or procedures that use default parameters, and where the parameter can be omitted at the call site. The reason is then that the value of the parameter passed is the same as the default parameter value. Removing the unneeded parameter value will make the code shorter and easier to read.

Example:

```

1  procedure ProcWithDefaultParam(P : Pointer = nil);
2  begin
3  ..
4  end;
5
6  procedure Proc;
7  var
8      P : Pointer;
9  begin
10     P := nil;
11     ProcWithDefaultParam(P); // the parameter P is not needed here
12 end;

```

In the code example above, the call to ProcWithDefaultParam does not need to include the parameter P, because P is assured to have the value "nil".

New section in Code Reduction Report: "Inconsistent conditions" (REDU22)

This section reports locations with inconsistent conditions. These are places where a condition check is repeated, even if the outcome will be the same as in the previous location.

Example:

```

1  procedure Proc;
2  var
3      I : Integer;
4  begin
5      if I = 1 then
6      begin
7          ..
8      end
9      else
10         if I = 1 then // gives inconsistent condition warning
11         begin
12             ..
13         end;
14     end;

```

In the code example above, the expression "I = 1" has already been evaluated higher up in the code. Thus, the code is not optimal and needs work.

New section in Code Reduction Report: "Typecasts that possibly can be omitted" (REDU23)

This section reports locations with typecasts that possibly can be omitted. It is locations where the typecast casts the variable to the same type that it already has.

Example:

```

1  var
2      I, K : Integer;
3  begin
4      ..
5      K := Integer(I);
6      ..
7  end;

```

Added in 9.2:

New section in Code Reduction Report: "Local identifiers never used" (REDU24)

This section reports local identifiers that are declared but not used.

Added in 9.2:**New section in Optimization Report: "Inlined subprograms not inlined because not yet implemented" (OPTI9)**

This section lists calls to inlined subprograms, where the subprogram will not be inlined. The reason is that the subprogram has not been implemented yet. It is implemented further down in the same module. There are a number of other conditions that also must be fulfilled for the subprogram to be inlined. But this one must always be fulfilled.

```
1  ..
2
3  type
4    TMyClass = class
5      private
6        ..
7        procedure Process;
8        function GetValue : Integer; inline;
9        procedure MoreProcessing;
10       ..
11     end;
12
13   ..
14
15   procedure TMyClass.Process;
16   var
17     Value : Integer;
18   begin
19     ..
20     Value := GetValue; // this call cannot be inlined!
21     ..
22   end;
23
24   function TMyClass.GetValue : Integer;
25   begin
26     ..
27   end;
28
29   procedure TMyClass.MoreProcessing;
30   var
31     Value : Integer;
32   begin
33     ..
34     Value := GetValue; // this call can be inlined!
35     ..
36   end;
37
38   ..
```

The solution is simple: If possible move the inlined subprogram further up in the code, so the compiler reaches it before the places where it is called.

Added in 9.8:**New section in Optimization Report: "Managed local variable that can be declared inline" (OPTI10)**

This section reports local variables that can benefit from being declared inline instead of in the main var-section.

Added in 9.9:**New section in Optimization Report: "Managed local variable is inlined in loop" (OPTI11)**

This section reports local managed variables that are declared inside a loop, which can decrease performance.

New section in Convention Compliance Report: "Class fields that are not declared in the private/protected sections" (CONV22)

This is a list of all class fields that are not declared in the private/protected sections of a class.

Fields should normally not be made "public". They should instead be accessed through properties

Added in 9.10:**New section in Warnings: "Mismatch parameter value (32/64-bits)" (WARN63)**

This section reports locations where 32-bits variables are passed as 64-bits parameters (or vice versa).

Added in 9.11:**New sections in Convention Compliance Report:****"Class fields that do not start with 'F'" (CONV23)**

This section lists class fields with a name not starting with "F". It complements the CONV6 report section which only lists fields exposed as properties.

"Value parameters that do not start with selected prefix" (CONV24)

This section lists value parameters that do not start with selected prefix.

"Const parameters that do not start with selected prefix" (CONV25)

This section lists const parameters that do not start with selected prefix.

"Out parameters that do not start with selected prefix" (CONV26)

This section lists out parameters that do not start with selected prefix.

"Var parameters that do not start with selected prefix" (CONV27)

This section lists var parameters that do not start with selected prefix.

"Old-style function result" (CONV28)

This section lists functions where instead of "Result", the function name is used as the result variable.

"With statements" (CONV29)

This section lists locations where "with" is used.

"Private can be changed to strict private" (CONV30)

This section lists class members that can be strict private instead of private.

"Protected can be changed to strict protected" (CONV31)

This section lists class members that can be strict protected instead of protected.

New section in Identifiers Report:**"Inlined variables and constants" (IDEN4)**

This section lists variables and constants that are declared inline.

Added in 9.12.2:**New section in Strong Warnings: "Interface lacks GUID" (STWA10)**

This section reports interface types without a GUID.

Relative paths

Now relative paths for folders can be specified. This applies to the main file folder, output folder, searched folders, excluded report folders and excluded search folders. The relative path should be relative to the folder where the project file (PAP-file) is located.

In the dialog where paths are selected, just write the relative path manually and add it to the selected folders.

See also:

[What's new in version 8?](#)

[What's new in version 7?](#)

[What's new in version 6?](#)

[What's new in version 5?](#)

[What's new in version 4?](#)

[Introduction](#)

4 What's new in version 8 (May 2016)?

This text describes changes and new features in Pascal Analyzer version 8, compared to version 7.x. The new version has been enhanced with a lot of smaller and greater features.

Support for Delphi 10.1 Berlin

Pascal Analyzer now also supports all targets for Delphi 10.1 (released April 2016).

Support for Delphi XE8 and Delphi 10 Seattle

Pascal Analyzer now also supports all targets for Delphi XE8 (released April 2015), including the new iOS 64-bits compiler.

It also supports all targets for Delphi 10 Seattle (released August 2015)

Improved Unicode-awareness

Reports will now be written in Unicode (UTF-8). This means that if you for example use Russian letters in your source code, these will show up correctly in the reports.

New 64-bit version

Pascal Analyzer is in this version available in both 64-bits and 32-bits. The new 64-bits version is in PAL.EXE and PALCMD.EXE. The 32-bits version is in PAL32.EXE and PALCMD32.EXE.

The 64-bits version is installed (as default) under "C:\Program Files\Peganza\Pascal Analyzer 8". The 32-bits version will be located in the sub-folder "PAL32".

However, if you install on a 32-bits system, only the 32-bit version will be installed (as default in the folder "C:\Program Files (x86)\Peganza\Pascal Analyzer 8").

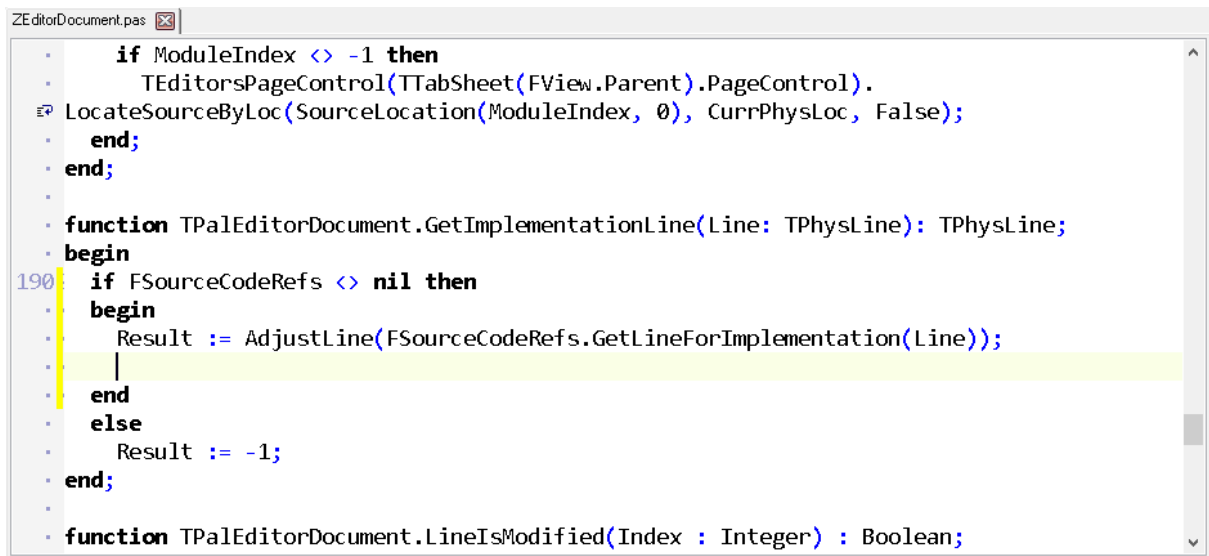
The 64-bits version is faster, but consumes more memory. Nevertheless, if you are on a 64-bits computer (which you must probably are), your first-hand choice should be to use the 64-bits version.

Built-in multi-tab-page editor

Optionally you can now edit source files within Pascal Analyzer. We have added a full-featured editor that lets you conveniently edit source files. Use this editor for small changes without leaving the Pascal Analyzer environment. Optionally you can also mimic the way backup files are copied to the "_history" folder.

To invoke the editor, just double-click on an identifier in a report. The editor window will then be displayed, the source file loaded and the cursor positioned on the relevant line.

The editor has got many of the expected features, like undo-redo capability, syntax highlighting, and so on.



```

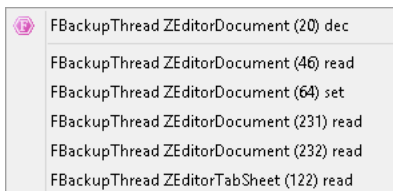
ZEditorDocument.pas
•   if ModuleIndex <> -1 then
•       TEditorsPageControl(TTabSheet(FView.Parent).PageControl).
•   LocateSourceByLoc(SourceLocation(ModuleIndex, 0), CurrPhysLoc, False);
•   end;
•   end;
•
•   function TPalEditorDocument.GetImplementationLine(Line: TPhysLine): TPhysLine;
•   begin
190  if FSourceCodeRefs <> nil then
•       begin
•           Result := AdjustLine(FSourceCodeRefs.GetLineForImplementation(Line));
•       end
•       else
•           Result := -1;
•       end;
•   end;
•   function TPalEditorDocument.LineIsModified(Index : Integer) : Boolean;

```

Editor window in Pascal Analyzer

There are also navigation features in the source code editor. You can navigate between references (reads and sets etc) for the selected object, quickly go to declaration or implementation for a selected method, etc.

For example, right-click on an identifier in the source code, and display a menu of locations where the identifier is referenced:



Local menu with references for the selected identifier

Then click a menu item to navigate to the reference location.

For advanced development work, our recommendation is that you should continue using the powerful RAD Studio IDE from Embarcadero as your main editing environment.

But the editor in PAL will come in handy when navigating your source code and making minor modifications.

Reports displayed in multi-tab-pages

Each report will now as default be displayed in a separate tab page viewer. It is then easy to switch between different reports. It is possible to revert to the pre-PAL8 action, which was to show reports in a single viewer.

Start viewing reports, as soon as they are ready

In this version, reports are available for viewing as soon as they are ready. You do not

have to wait till all reports are produced. As soon as a report is ready, its corresponding entry in the tree will be enabled. This is because reports are produced in a special thread.

Parse less code, still produce full reports

You can now choose to skip parsing implementation sections for units that are not reported. This is the default option, but you can still revert to the pre-PAL8 action, which was to also parse those parts. The results will be the same, except for the Uses Report, which will be more complete if you choose to also skip non-reported implementation sections.

Record fields also tracked

Now also individual record fields are tracked, that means they are registered whenever referenced (read or set for example). The reports will include information about these fields.

Improved performance

Performance is now increased quite a lot, both for parsing and building reports. This version is about 30-50% faster than version 7.x (your mileage may vary). Memory consumption has also been drastically reduced.

Improved quality

We have done a lot of work in this release, to avoid displaying false warnings. The internal workings of PAL have been refactored and enhanced in many ways. However, PAL can sometimes not fully resolve object references. Objects may be created through a class reference, and then it is not possible for a static code analyzer to know which objects that are in fact created at runtime.

New Description property for projects

You can now optionally associate a descriptive text or comment to a project definition. As other parts of PAL, it also supports Unicode.

New Start page

There is a new start page with online content from Peganza's blog. There is a setting in the preferences dialog, to turn on/off this feature.

New section in Warnings Report

There is a new section "Possible bad assignment". It lists assignments to bigger variables, where data loss is possible.

New section in Code Reduction Report

There is a new section "Fields only used in single method". It lists fields that probably instead could be declared as local variables.

New section in Optimization Report

"Local subprogs", displays a list of nested (local) subprograms. Normally performance is better if you avoid nesting, mainly because the compiler can do a better job using available processor registers instead of the stack.

New columns in Modules Report

The three new columns display the encoding for a source code file (like "UTF-8" etc), and also shows if it has "initialization" or "finalization" sections.

See also:

[What's new in version 9?](#)

[What's new in version 7?](#)

[What's new in version 6?](#)

[What's new in version 5?](#)

[What's new in version 4?](#)

[Introduction](#)

5 What's new in version 7 (November 2013)?

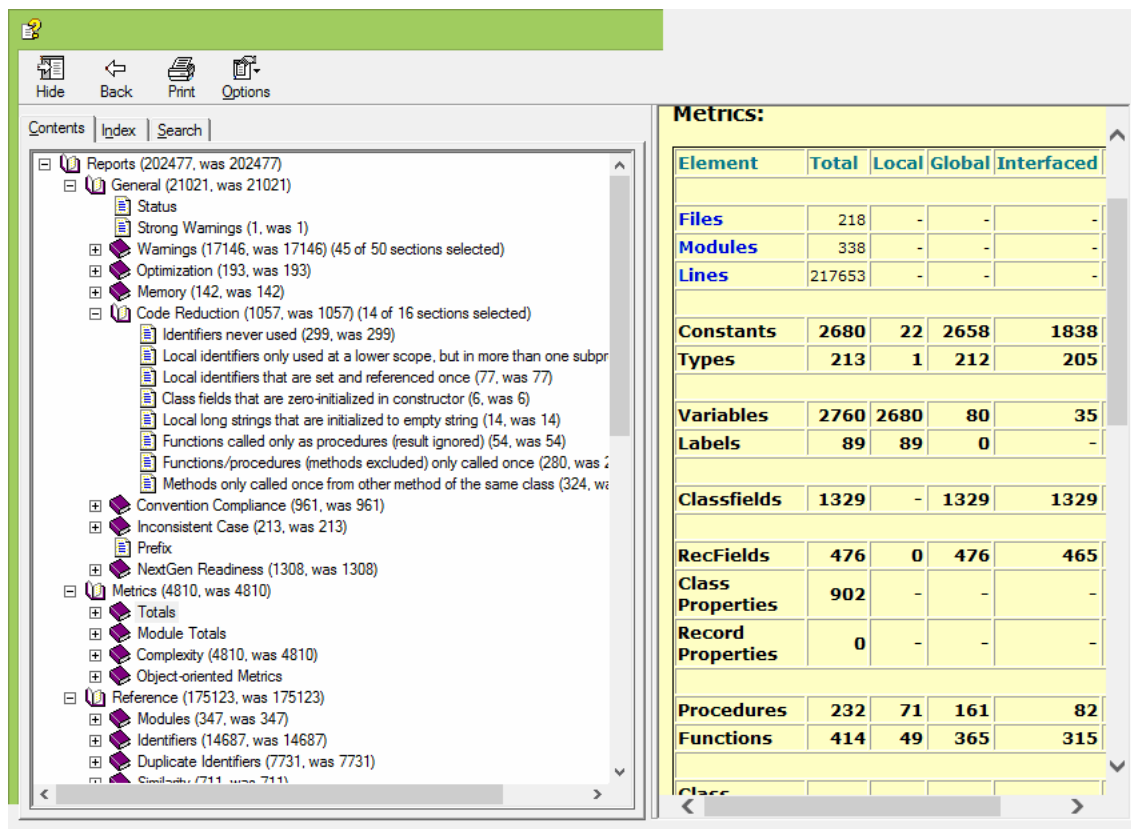
This text describes changes and new features in Pascal Analyzer version 7, compared to version 6. The new version has been enhanced with a lot of smaller and greater features.

Support for Delphi XE5, XE6 and XE7

This version adds support for Delphi XE5, XE6 and XE7 (all compiler targets including iPhone and Android mobile devices). Double-clicking on a report line in Pascal Analyzer will optionally jump to the correct location in the Delphi IDE.

Generate CHM files

It is now possible to also generate a CHM file for the collection of HTML pages that are output. The CHM file is compressed, and features full-text search.



Reports in CHM-file

Mouse wheel support

Use Ctrl+MouseWheel in reports and in the source viewer to temporarily increase/decrease font size.

Excluded files

This setting as default used to specify "Windows.pas;System.pas". But in this version it is changed to "System.pas". This means that Windows.pas will also be parsed, if found by PAL. The analysis will take a bit longer to run, but the results will be more accurate and complete.

Status Report

The [Status Report](#) now, as last section, lists all files that have been found and loaded. Use the list to check if PAL has found all the expected files.

New pane on status bar

The status bar at bottom of the main window has got a new section, telling the compiler target for the currently loaded project.

Like "Delphi XE5 (Win32)" for example. It also shows a representative glyph for the selected Delphi version.

Redeclared identifiers

There is a new section in the [Warnings Report](#) that lists redeclared identifiers from the System unit. Although allowed, this is a source for confusion when maintaining the code.

Identifier with same name as keyword/directive

This is a new section in the Warnings Report that lists identifiers that have the same name as keyword/directive.

Different string types in Totals and Module Totals Report

These two reports now also lists the number of different types of strings, like UnicodeString and WideString.

Misc. bug fixed

Some small bugs, both in the parser and report builder have been fixed.

Predefined identifiers from System unit

Identifiers from System unit are preloaded by PAL. This feature has been totally reworked, resulting in much more accurate results.

Improved handling of safe calls

Calls to functions that don't not require the passed variable to be initialized, like SizeOf(), are better detected in this version, leading to more accurate results.

New NextGen Readiness Report

This new report measures how well prepared your code is for the new NextGen compiler.

New section in Strong Warnings Report

This new section reports subprograms that unconditionally call themselves recursively.

Totals for reports

In the report list, total hit numbers (if activated) are also displayed for report groups and as a grand total for all reports. This makes it easier to spot the difference between two separate runs.

See also:

[What's new in version 9?](#)

[What's new in version 8?](#)

[What's new in version 6?](#)

[What's new in version 5?](#)

[What's new in version 4?](#)

[Introduction](#)

6 What's new in version 6 (August 2011)?

This text describes changes and new features in Pascal Analyzer version 6, compared to version 5. The new version has been enhanced with a lot of smaller and greater features. Together they make this upgrade one of the most important in the history of Pascal Analyzer. There is now a total of 51 reports, with 197 report sections. Also there are four special multi-reports.

Support for Delphi XE2, XE3 and XE4

From version 6.1, PAL also supports Delphi XE2 (Win32, Win64 and OSX targets). Support for Delphi XE3 (Win32, Win64 and OSX targets) was added in version 6.2. Delphi XE4 (Win32, Win64, OSX, iOS) support was added in version 6.3.

Relative path for main file

For the main file that is analyzed, it is now possible to use relative paths. The path is entered relative to the location of the project file itself. So, for example, if the project file is C:\PALProjects\MyProject.pap, and the main file is entered as "MyApp.dpr", PAL will look for this file in the folder C:\PALProjects. Likewise, if the main file is entered as "..\Dir\MyApp.dpr", PAL will look for the file in the C:\Dir folder.

This feature is very convenient if you share PAL projects between more than one computer. Instead of specifying a main file path as "C:\Projects\MyApp.dpr", and get problems when running on another computer that has the location "D:\Projects\MyApp.dpr", you can now specify "MyApp.dpr" which will work on both computers.

New Help Report

The new [Help Report](#) checks help topics for the analyzed project's help system. Help topics are normally defined as numbers in an external *.h-file. These values correspond to values of the HelpContext property. The first report section presents a list of all HelpContext values found in DFM files. Following two sections report topics that are either missing in the *.h-file, and HelpContext values that are missing in the DFM files.

New Searched Strings Report

The new [Searched Strings Report](#) searches for strings in the project's source code files. For each search string it reports files where the strings exists, and files where they not exist. Use the report for example to verify that your source code files include a copyright notice, or common include files.

New Map File Report

The new [Map File Report](#) displays information about modules that are linked into the binary executable (EXE, DLL). It generates this information from the MAP file that can be generated for your Delphi project when compiling the source code.

Improvements in Third-Party Report

In the [Third-party dependencies Report](#), the sort mode now has importance. The sort mode is set on the General tab page for the project options. The lists are displayed in module-identifier or identifier-module order.

Check for local variables that are referenced before being created

There is a new section in the [Memory Report](#) that reports local variables (objects) that are referenced before they are created. This is an error that in most cases will result in a runtime error.

Check for local variables that are referenced after being freed

There is a new section in the Memory Report that reports local variables (objects) that are freed, but referenced further down in the code. This is an error that in most cases will result in a runtime error.

Duplicate lines

This new section in the Warnings Report gives warnings for source code lines that are duplicated, which means that two identical lines occur right after each other. This could be a mistake in the code.

Check for local variables that are set but not read further down in the code

This is a new section in the [Warnings Report](#) that reports variables that are set, but never referenced further down in the code. It indicates some programming mistake.

Improvements in Warnings Report - Local variables that are referenced before they are set

This section in the Warnings Report has been improved. References of a variable with "SizeOf(Var)" are considered safe, and do not render an unnecessary warning.

Improvements in Warnings Report - Function result not set

This section in the Warnings Report has been improved. The detection of functions where the result has not been set, is now much more accurate.

New section in Warnings Report - Duplicate class types in except-block

This section in the Warnings Report checks if the same class type has been used more than once in an except-block, in an on-statement. It indicates some programming error.

Default report sections

Some sections in the Warnings Report, are in this version removed from the default selection of included report sections. These are the sections where the results are not certain, and where the results often are false positives. It is the sections that contain "possibly" in the name.

Improvements in Complexity Report - Complexity per module/subprogram

This section in the [Complexity Report](#) has been extended with a part listing all modules.

Furthermore, the items displayed in the report can be listed in order according to LOC, Lines, etc. This is of great value for example when finding out which modules or functions that are most complex.

Improvements in Literal Strings Report

The [Literal Strings Report](#) now supports sorting of the strings. The sorting done depends on the sort mode setting in [Options|Properties - General](#), so either the sorting will be according to module or according to the string itself.

Improvements for multi projects

All reports for multi projects have been improved. The reports now only display identifiers that are used in more than one of the analyzed projects.

New menu to check for newer version

There is a new menu command under Help, that lets PAL check if there is a newer version available. If you do not activate the option to let PAL automatically check for newer versions on startup, this is a way of doing it manually.

Close button for source viewer window

There is a new button available when the source viewer window is open. With this button you can close the window.

Customize "subprogram" string

Some of our users want to use another string than "subprogram" in the report headers. It is now possible to select another string, for example "function" or "subroutine".

New switch /D for PALCMD

This new switch lets you specify conditional defines for PALCMD, the command-line version of PAL. If present, they will override the setting in project options.

New switch /X for PALCMD

This new switch lets you specify excluded search folders for PALCMD. If present, they will override the setting in project options.

Improvements in Status Report

The [Status Report](#) now also displays total time for each report, not only parsing and report building time. It also reports user-defined environment variables, if they are found, and their translation. For example, if you have defined an environment variable "\$(THIRD)", it will be displayed together with its translation "C:\3RDPARTY".

Show implementation line numbers

A new option determines if the line number where a subprogram is declared is displayed in the reports. If set to Yes, instead implementation line number is displayed. This has meaning if you double-click on the line to jump to the source code. Either it will then

take you to the declaration or the implementation line. Often you would probably prefer to reach the implementation. In previous versions of PAL, the line number displayed was always for the declaration line.

Improved documentation

The help texts have been improved and extended. For example, report codes, like "WARN12" are now displayed for each report section. These codes can be used when running [PALCMD.EXE](#).

Also, check out the FAQ page at our web site. It will give answers to many common questions.

See also:

[What's new in version 9?](#)

[What's new in version 8?](#)

[What's new in version 7?](#)

[What's new in version 5?](#)

[What's new in version 4?](#)

[Introduction](#)

7 What's new in version 5 (May 2010)?

This section describes changes and new features in Pascal Analyzer version 5, compared to version 4.

Build reports in multiple threads

Take advantage of the new generations of multi-core processors, and generate reports quicker by using multiple threads. Of course your result will depend on your particular hardware. Greater number of processors and cores means faster reports.

There is a new option to set the number of threads that should be used when creating reports. For a dual-core processor, this means that two threads is probably a suitable choice. This is also the default value. Because your mileage may vary, you should probably experiment with this setting to find optimal number of threads for a particular project and hardware.

Suppress reporting by source line

It is now possible to let PAL skip reporting, for selected source code lines. This is convenient, to avoid the false positives that PAL sometimes outputs. Remember, that unlike the compiler, PAL does not always have access to the complete source code, making it hard to resolve all references correctly.

By adding a special comment, like for example:

```
//PALOFF
```

.. at the end of a source code line, means that PAL will *not* report any issues found on that particular line. Also, if you put the comment on a line where an identifier is declared, that identifier will not be reported. This is the most effective way to get rid of all issues for a particular identifier.

It is possible to select what string should be used as the marker. Default value for the suppression marker is "PALOFF", but you can change this to something else. Blank spaces between "/" and the suppression marker are allowed. You can also have more comment text to the right of the marker, like:

```
//PALOFF because false warning otherwise
```

New Exception Report

You could think that the 40+ reports already available cover most demands. But one area that has not been touched in previous versions is exceptions.

This new report produces two sections dealing with exceptions:

- Exception Call Tree

This section is similar to the [Reverse Call Tree Report](#), but it adds information about how exceptions are handled.

- Raised exceptions

This section lists which exceptions that are explicitly raised in the source code (with the keyword "raise").

New Similarity Report

This new report produces listings of identifiers that have similar names. It does so by using the well-known soundex algorithm. You may use this report to determine how close your identifiers are by name.

New Module Totals Report

This new report produces similar results as the [Totals Report](#), but divided for each module (unit).

New section in Duplicate Identifiers Report

This new section also lists duplicate identifiers, but only those in overlapping scope. These identifiers are most likely to cause problems, if they are confused with each other.

Command line options for PALCMD.EXE

There is a whole new bunch of command line options for [PALCMD](#), which will override settings from PAL.INI:

/F specifies the report format (text, HTML, XML)

/S specifies a semicolon-delimited list of search folders, like /S="C:\CODE1;C:\My Code"

/T specifies the number of report threads to use

/L specifies the path to a text file with limit info

This last option demands a more detailed explanation: You can now use a text file with limit information, which determine the maximum values for sections in reports.

For example, if the file contains just this line:

```
WARN1=5
```

.. "WARN1" is the code for the section "Interfaced identifiers that are used, but not outside of unit" in the [Warnings Report](#).

If then the number of warnings in a particular run of PALCMD.EXE exceeds 5, PALCMD.EXE will write out an error, and halt with an exit code of 99, and with an error message. You can use this when integrating PAL with your build process, to stop it when any unexpected results occur.

Speed improvements

There are a number of improvements in the parser, allowing for faster parsing of source code. For example, include files are now cached, so that their content does not need to be read from disk repeatedly.

The building of reports is now much faster, due to a more efficient handling of the data structures generated during parsing, and to the new ability to build reports in threads.

These speed improvements are equally available both in the GUI program PAL.EXE and the command-line program PALCMD.EXE

Check for new versions

There is now an option in the [settings dialog](#) that will let PAL inform when new versions are available at our web site.

Project options dialog

There is now a hint window displayed when hovering over reports on the [Reports](#) tab page. This will help you to get an indication what the report does. Also, the organization and overall design of this dialog has been improved

Suggest main file folder for project file

Normally, PAL will suggest saving a file in the folder that is defined as the project folder. But you can now select an option, that lets PAL suggest the same folder as the folder in which the main file for the project is located. This is desirable if you want to keep PAL's project file together with the source code it analyses.

Report tree

It is now possible to select the font and background color for the report tree.

See also:

[What's new in version 9?](#)

[What's new in version 8?](#)

[What's new in version 7?](#)

[What's new in version 6?](#)

[What's new in version 4?](#)

[Introduction](#)

8 What's new in version 4 (October 2006)?

This section describes changes and new features in Pascal Analyzer version 4, compared to version 3.

Support for Delphi 2007, Delphi 2009 (Win32), and Delphi 2010 (Win32)

PAL also supports the latest Delphi versions, 2007, 2009 and 2010.

Excluded folders and files

There are new settings for excluded folders and excluded files in the [Properties](#) dialog for a project. By using these settings you can tell PAL which files to skip when parsing your source code.

HTML help

Pascal Analyzer now uses HTML help. The help texts are loaded from the PAL.CHM file.

Folders

The folder for project files is now as default "C:\Documents and Settings\<acc>\My Documents\Pascal Analyzer\Projects".

Section and report counters

In the report tree, many report sections show the number of reported items. From this version the old (previous) numbers could also be displayed, for comparison purposes. For example, this is a caption from the Optimization Report.

"(3) Virtual methods (procedures/functions) that are not overridden (was 5)"

The first number in parenthesis (3) shows the current number of warnings for this section. The second number (5) shows the number of warnings that were generated the last time this project was analyzed. For reports that have not yet been run, the text "unknown" will be displayed, instead of a number.

It is possible to define masks for how the captions are displayed in reports and sections both when section counters are used and when they are not. In this way you can choose to display the caption in a customizable format, like

"Virtual methods (procedures/functions) that are not overridden (now 3, was 5)"

You can also customize colors for captions in the report tree. For example, an item with a growing number of warnings, can be displayed in red colors if you so like.

Separate threads for parsing and reporting

Parsing and reporting is now done by a separate working thread. This gives a more responsive user interface even during this process.

Pascal Analyzer 4 is also much quicker than its predecessor. It typically parses and

generates reports about 25% faster.

New Strong Warnings Report

The [Strong Warnings Report](#) displays severe warnings. Those warnings points to errors in the source code that may lead to runtime failure of your product. The section "Property Access in read/write methods" reports methods where properties are accessed directly, causing infinite recursion.

The section "Ambiguous unit references" lists identifiers and locations with ambiguous references. This are locations with a reference to an identifier that is exposed by two or more used units. The compiler will search the uses list backwards for a suitable unit, which makes this very unstable and prone for mistakes.

New Directives Report

The new [Directives Report](#) displays information about directives. There are five sections:

- Identifiers marked with the "deprecated" directive
- Identifiers marked with the "experimental" directive
- Identifiers marked with the "library" directive
- Identifiers marked with the "platform" directive
- Identifiers marked with the "inline" directive

New To-Do Report

The new [To-Do Report](#) displays the to-do items that are entered in the special dialog box in the Delphi IDE, or inserted directly into the source code..

New Module Call Tree Report

The new [Module Call Tree Report](#) displays a hierarchical module call tree.

New sections in the Code Reduction Report

These sections are new in the [Code Reduction Report](#):

- Unneeded boolean comparisons
- Boolean assignment can be shortened

New sections in the Convention Compliance Report

These sections are new in the [Convention Compliance Report](#):

- Hard to read identifier names
- Label usage
- Bad class visibility order
- Identifiers with numerals
- Class/member name collision

New sections in the Warnings Report

These sections are new in the [Warnings Report](#):

- Local for-loop variables read after loop

- Local for-loop variables possibly read after loop
- For-loop variables not used in loop
- Non-public constructors/destructors
- Functions called as procedures
- Mismatch property read/write specifiers

New section in the Memory Report

This section is new in the [Memory Report](#):

The new section "Unbalanced Create/Free" reports object instances that are not created and freed the same number of times.

New section in the Uses Report

This new section in the [Uses Report](#) reports "Mutual unit references", where two units reference each other.

New section in the Control Warnings Report

This new section in the [Control Warnings Report](#) reports controls with an assigned Hint property, where both ShowHint and ParentShowHint are "false".

New section in the Uses Report for multiprojects

There is a new section in the [Uses Report](#) that for each unit reports which projects that reference it.

See also:

[What's new in version 9?](#)

[What's new in version 8?](#)

[What's new in version 7?](#)

[What's new in version 6?](#)

[What's new in version 5?](#)

[Introduction](#)

9 Command-Line Options for PAL.EXE and PAL32.EXE

There are currently two types of parameters that you can use on the commandline:

Path

This is a complete path to the project you want PAL to open at start-up. Normally PAL loads the last used project, so this is a way to override that behaviour.

Always as the second parameter, use /AUTO to indicate that PAL should run the project indicated by the first parameter and then terminate itself. In this way it is possible to use for example Window's Task Scheduler to run PAL.EXE automatically.

Example:

```
PAL c:\project\MyProj.pap
```

PAL will open the project file MyProj.pap at start-up.

See also:

[How to use PAL.EXE and PAL32.EXE](#)

[How to use PALCMD.EXE and PALCMD32.EXE](#)

[Introduction](#)

[Known limitations](#)

[Main menu](#)

[Main window](#)

10 How to use PAL.EXE and PAL32.EXE

Activation

When starting PAL for the first time, your license must be activated through the Internet, unless you activated it during the installation. If you are running PAL on a computer that has not got access to the Internet, you can create an activation XML file and send to us. You will then receive a response XML file that you use to manually activate.

For the activation, use the registration key that was sent to you by mail when buying the product license. This key is good for a small number of activations. Contact us if you run out of activations, for example when reinstalling on a new computer. You are entitled to install Pascal Analyzer on up to four computers, as long as you are the Pascal Analyzer user on those computers. If more than one developer needs Pascal Analyzer, additional licenses must be bought.

If you need to move the installation to another computer, you can deactivate the license on the current computer. Then you will be able to activate the license on the new computer. Use the menu command "Deactivate License" in the [About](#)-menu for this.

Summary

PAL is an easy-to-use standalone Windows program. Just create a new project and select a source file to analyze. Either select a complete Delphi project (DPR-file), Delphi package (DPK-file) or a single source file (PAS-file), set a few options and start the analysis. PAL presents the results either as plain text in a text viewer or in a HTML browser (as HTML or XML). All reports are written to text (*.txt), HTML files (*.htm) or XML files (*.xml) for later retrieval, maybe in another tool or editor.

You do not have to alter your code in order to examine it with PAL. PAL does not either change or affect your code in any way.

Follow this simple procedure to create reports in PAL:

1. Create a new PAL project. Then select a main file to analyze, either a complete Delphi project or a single source file. You can also open an existing PAL project.
2. Make sure that the selected compiler target is suitable for the source code. Enter other options that are required, like which main file to analyze.
3. Press the [Run](#) button and wait from a few seconds till several minutes, depending on the size of the code and the number of reports selected.
4. Examine the reports with the built-in text viewer or HTML browser.

In order to make your source code possible to analyze with PAL, you should make certain that only one statement exists for every source line, like

```
procedure Proc;
```

```
var
  I : integer;
begin
  ..
  I := 5;
  CallProc( I );
  ..
end;
```

..and not like

```
procedure Proc;
var
  I : integer;
begin
  ..
  I := 5; CallProc( I );
  ..
end;
```

Avoid the latter case, because PAL cannot decide if the identifier I is set or referenced first. It is also considered better coding to only keep one statement on each line.

PAL is not a Delphi IDE plug-in (expert or wizard). It is a standalone program. You can however install the application in the Delphi IDE, allowing easy access from within the Delphi development environment. If you want to try a Delphi IDE plug-in, check out our Pascal Expert product. It contains some of the capabilities of Pascal Analyzer, available while you are developing.

To install PAL in the Delphi IDE, follow these simple steps:

1. Start Delphi and select **Tools|Configure Tools**
2. In the dialog box, press the **Add** button
3. Fill the fields, for example with these values:

Title: Pascal Analyzer
Program C:\Program Files\Peganza\Pascal Analyzer 9\PAL.exe
:
Working C:\Program Files\Peganza\Pascal Analyzer 9
dir:

See also:

[Introduction](#)

[How to use PALCMD.EXE](#)

[Known limitations](#)

[Command Line Options for PAL.EXE and PAL32.EXE](#)

[Main menu](#)

[Main window](#)

11 How to use PALCMD.EXE and PALCMD32.EXE

The standalone command-line version PALCMD.EXE (and 32-bits PALCMD32.EXE) is useful when you want to automate the process of creating reports. PALCMD.EXE uses the same engine as the GUI version PAL.EXE and produces the same output.

Run PALCMD.EXE from the command prompt using the following syntax:

```
PALCMD projectpath\ sourcepath [ options]
```

Option	Explanation
/A+	Parse both source/form files
/A-	Parse source files only
/FA	Parse all files
/F+	Parse all files
/FR	Parse main file and directly used files
/FM	Parse main file only
/F-	Parse main file only
/Q	Quiet mode
/CBP	Borland Pascal 7 (or earlier)
/CD1	Delphi 1
/CD2	Delphi 2
/CD3	Delphi 3
/CD4	Delphi 4
/CD5	Delphi 5
/CD6	Delphi 6
/CD7	Delphi 7
/CD8	Delphi 8 for .NET
/CD9W	Delphi 2005 for Win32
/CD9N	Delphi 2005 for .NET
/CD10W	Delphi 2006 for Win32 (also Turbo Delphi for Win32)
/CD10N	Delphi 2006 for .NET (also Turbo Delphi for .NET)
/CD11W	Delphi 2007 for Win32
/CD11N	Delphi 2007 for .NET
/CD12W	Delphi 2009 for Win32
/CD14W	Delphi 2010 for Win32
/CDXEW	Delphi XE for Win32
/CDXE2W32	Delphi XE2 for Win32
/CDXE2W64	Delphi XE2 for Win64
/CDXE2OSX	Delphi XE2 for OSX
/CDXE3W32	Delphi XE3 for Win32
/CDXE3W64	Delphi XE3 for Win64
/CDXE3OSX	Delphi XE3 for OSX
/CDXE4W32	Delphi XE4 for Win32
/CDXE4W64	Delphi XE4 for Win64
/CDXE4OSX	Delphi XE4 for OSX
/CDXE4IOSDEV	Delphi XE4 for iOS Device
/CDXE4IOSSIM	Delphi XE4 for iOS Simulator

```
/CDXE5W32      Delphi XE5 for Win32
/CDXE5W64      Delphi XE5 for Win64
/CDXE5OSX      Delphi XE5 for OSX
/CDXE5IOSDEV   Delphi XE5 for iOS Device
/CDXE5IOSSIM   Delphi XE5 for iOS Simulator
/CDXE5ANDROID Delphi XE5 for Android

/CDXE6W32      Delphi XE6 for Win32
/CDXE6W64      Delphi XE6 for Win64
/CDXE6OSX      Delphi XE6 for OSX
/CDXE6IOSDEV   Delphi XE6 for iOS Device
/CDXE6IOSSIM   Delphi XE6 for iOS Simulator
/CDXE6ANDROID Delphi XE6 for Android

/CDXE7W32      Delphi XE7 for Win32
/CDXE7W64      Delphi XE7 for Win64
/CDXE7OSX      Delphi XE7 for OSX
/CDXE7IOSDEV   Delphi XE7 for iOS Device
/CDXE7IOSSIM   Delphi XE7 for iOS Simulator
/CDXE7ANDROID Delphi XE7 for Android

/CDXE8W32      Delphi XE8 for Win32
/CDXE8W64      Delphi XE8 for Win64
/CDXE8OSX      Delphi XE8 for OSX
/CDXE8IOSDEV   Delphi XE8 for iOS Device 32-bits
/CDXE8IOSDEV64 Delphi XE8 for iOS Device 64-bits
4
/CDXE8IOSSIM   Delphi XE8 for iOS Simulator
/CDXE8ANDROID Delphi XE8 for Android

/CD10W32       Delphi 10 for Win32
/CD10W64       Delphi 10 for Win64
/CD10OSX       Delphi 10 for OSX
/CD10IOSDEV    Delphi 10 for iOS Device 32-bits
/CD10IOSDEV64 Delphi 10 for iOS Device 64-bits
/CD10IOSSIM    Delphi 10 for iOS Simulator
/CD10ANDROID   Delphi 10 for Android

/CD101W32      Delphi 10.1 for Win32
/CD101W64      Delphi 10.1 for Win64
/CD101OSX      Delphi 10.1 for OSX
/CD101IOSDEV   Delphi 10.1 for iOS Device 32-bits
/CD101IOSDEV64 Delphi 10.1 for iOS Device 64-bits
4
/CD101IOSSIM   Delphi 10.1 for iOS Simulator
/CD101ANDROID Delphi 10.1 for Android

/CD102W32      Delphi 10.2 for Win32
/CD102W64      Delphi 10.2 for Win64
/CD102OSX      Delphi 10.2 for OSX
/CD102IOSDEV   Delphi 10.2 for iOS Device 32-bits
/CD102IOSDEV64 Delphi 10.2 for iOS Device 64-bits
```

```
4
/CD102IOSSIM Delphi 10.2 for iOS Simulator
/CD102ANDROIDDelphi 10.2 for Android
/CD102LINUX64 Delphi 10.2 for Linux 64-bits

/CD103W32 Delphi 10.3 for Win32
/CD103W64 Delphi 10.3 for Win64
/CD103OSX Delphi 10.3 for OSX 32-bits
/CD103OSX64 Delphi 10.3 for OSX 64-bits
/CD103IOSDEV Delphi 10.3 for iOS Device 32-bits
/CD103IOSDEV6 Delphi 10.3 for iOS Device 64-bits
4
/CD103IOSSIM Delphi 10.3 for iOS Simulator
/CD103ANDROIDDelphi 10.3 for Android 32-bits
/CD103ANDROIDDelphi 10.3 for Android 64-bits
64
/CD103LINUX64 Delphi 10.3 for Linux 64-bits

/CD104W32 Delphi 10.4 for Win32
/CD104W64 Delphi 10.4 for Win64
/CD104OSX Delphi 10.4 for OSX 32-bits
/CD104OSX64 Delphi 10.4 for OSX 64-bits
/CD104IOSDEV Delphi 10.4 for iOS Device 32-bits
/CD104IOSDEV6 Delphi 10.4 for iOS Device 64-bits
4
/CD104IOSSIM Delphi 10.4 for iOS Simulator
/CD104ANDROIDDelphi 10.4 for Android 32-bits
/CD104ANDROIDDelphi 10.4 for Android 64-bits
64
/CD104LINUX64 Delphi 10.4 for Linux 64-bits

/CD11W32 Delphi 11 for Win32
/CD11W64 Delphi 11 for Win64
/CD11OSX Delphi 11 for OSX 32-bits
/CD11OSX64 Delphi 11 for OSX 64-bits
/CD11IOSDEV Delphi 11 for iOS Device 32-bits
/CD11IOSDEV64 Delphi 11 for iOS Device 64-bits
/CD11IOSSIM Delphi 11 for iOS Simulator
/CD11ANDROID Delphi 11 for Android 32-bits
/CD11ANDROID6Delphi 11 for Android 64-bits
4
/CD11LINUX64 Delphi 11 for Linux 64-bits
/CD11OSXARM6 Delphi 11 for OSX ARM 64-bits
4

/CD12W32 Delphi 12 for Win32
/CD12W64 Delphi 12 for Win64
/CD12OSX Delphi 12 for OSX 32-bits
/CD12IOSDEV Delphi 12 for iOS Device 32-bits
/CD12IOSDEV64 Delphi 12 for iOS Device 64-bits
/CD12IOSSIM Delphi 12 for iOS Simulator
/CD12ANDROID Delphi 12 for Android 32-bits
/CD12ANDROID6Delphi 12 for Android 64-bits
```

```

4
/CD12LINUX64 Delphi 12 for Linux 64-bits
/CD12OSXARM6 Delphi 12 for OSX ARM 64-bits
4

/D=x          Conditional defines (/D=MyDef1;MyDef2)
/F=format     /F=T -> text, /F=H -> HTML, /F=X -> XML
/I=path       Path to PAL.INI (/I=C:\PAL\PAL.INI)
/L=path       Path to text file with limit info (/L=C:\PAL\Limits.txt)
/P            Create/use PAP-file in source folder
/R=path       Report root folder (/R=C:\Out or /R="C:\My Out")
/S=folders    Search folders (/S="C:\CODE1;C:\My Code")
/T=n          Number of report threads 1-64
/X=x          Excluded search folders (/X=C:\DIR1<+>;C:\DIR2)

```

Options are read from the project file. Some of the settings may be overridden by options on the command-line (see above).

The command-line version can, in contrast to the GUI version, also analyze source code without first creating a project. Just supply a source code path on the command-line instead of a project path. PALCMD will then use the settings according to the template which is used for new projects. These settings are in PAL.INI which is located in C:\Documents and Settings\<acc>\Application Data\Peganza\Pascal Analyzer.

The PAL.INI file is specially handled by PALCMD. If "/I" parameter is used the PAL.INI file as pointed to by the path, will be read. Else, if a PAL.INI file exists in the same folder as the program file itself, it will read the PAL.INI file from that location, otherwise it will read it from the same folder as the GUI program does. The GUI program will read the PAL.INI file from the [special folder](#) under "C:\Documents and Settings". In this way, if you keep the PALCMD.EXE in a special folder, you can make sure that it always uses the correct default options, by copying the PAL.INI file to that folder. This PAL.INI file will then not be affected of any changes that you happen to make while running the GUI program.

If an error occurs when PALCMD is run, the application terminates with exit code 99.

Example:

```
PALCMD
```

Shows help info and stops

```
PALCMD C:\projects\MyProj.pap
```

Runs PALCMD and analyses c:\projects\MyProj.pap

```
PALCMD "C:\My Units\MyUnits.pas" /FM /CBP
```

Runs PALCMD and analyses C:\My Units\MyUnits.pas with defaults as set in PAL.INI, but specifies that only the main file should be parsed, and that the compiler target is Borland Pascal 7.

```
PALCMD "C:\My Units\MyUnits.pas" /L=C:\PALCMD\Limits.txt
```

Runs PALCMD and analyses C:\My Units\MyUnits.pas with defaults as set in PAL.INI.

Uses limits set in Limits.txt (see below under /L)

You can also, starting from PAL 8, use just a file name as parameter, like:

```
PALCMD MyProj.pap
```

.. or

```
PALCMD MyUnits.pas
```

PALCMD will then check the current working directory for this file. If the current working directory is not specified, it will use the same directory as where the PALCMD program is located.

/D specifies conditional defines

This setting overrides the setting in the PAL.INI file or the project options. Conditional defines from source code and/or found in the Delphi project file will also be used.

/F specifies the report format (text, HTML, XML)

This setting overrides the setting for report format in the PAL.INI file or the project file.

/I specifies the path to PAL.INI

This forces PALCMD to use the PAL.INI pointed to by the supplied path.

/L specifies the path to a text file with limit info, in this case C:\PALCMD\Limits.txt.

For example, if the file contains this line:

```
WARN1=5
```

.. "WARN1" is the abbreviation for the section "Interfaced identifiers that are used, but not outside of unit" in the Warnings Report.

If then the number of warnings in a particular run of PALCMD.EXE exceeds 5, PALCMD.EXE will list it in the output.

When the number of warnings is less than the expected, it will also be listed.

The listings are also written to the [Status Report](#).

You can add how many lines as you wish to the limit file. Each line will be checked against the specified section and report.

This feature is not available in PAL.EXE (the GUI program).

/P creates/uses a PAP file

When specifying only a source file and not a project file on the command-line, there will be no PAP file where to store information about new and old hits counts for reports. To easily allow for this, without having to create a PAP file, include the switch /P. A PAP-file will be created in the source folder. If it already exists, it will be used.

/R specifies the report root folder

This setting overrides the setting for report folder in the PAL.INI file or the project file.

/S specifies search folders, separated with a semicolon

This setting overrides the setting for search folders in the PAL.INI file or the project file.

/T specifies the number of report threads to use

This setting overrides the setting in the PAL.INI file

/X specifies excluded search folders, separated with a semicolon.

It is also possible to use "<+>" to specify that subdirectories also should be excluded.

This setting overrides the setting in the PAL.INI file or the project file.

See also:

[Introduction](#)

[How to use PAL.EXE and PAL32.EXE](#)

[Known limitations](#)

[Command Line Options for PAL.EXE and PAL32.EXE](#)

[Main menu](#)

[Main window](#)

12 Installation folders

Pascal Analyzer is typically installed with this folder structure (Windows XP, Windows 7, Windows 8):

C:\Program Files\Peganza\Pascal Analyzer

(or C:\Program Files (x86)\Peganza\Pascal Analyzer when running on a 32-bit computer)

This folder contains executable files, help files etc for the installation of Pascal Analyzer. On a 64-bits system, the 32-bits version will be available in the sub-folder "PAL32".

C:\Documents and Settings\<acc>\Application Data\Peganza\Pascal Analyzer

This is where the PAL.INI file is stored. The INI file contains common settings for the PAL environment.

When running under Windows Vista, this folder is instead

C:\Users\<acc>\AppData\Roaming\Peganza\Pascal Analyzer.

In this folder, you will also find error log files, if an unexpected error occurs. You should send those files to support@peganza.com together with a description about how the error occurred.

C:\Documents and Settings\<acc>\My Documents\Pascal Analyzer\Projects

This folder contains the project files (*.pap, *.pam). You can store your projects in any folder, this is just the default folder.

C:\Documents and Settings\<acc>\My Documents\Pascal Analyzer\Projects\Output

This folder is the default root folder for your report files. If you for example create reports for the project MyProj, the output files will be found in C:\Documents and Settings\<acc>\My Documents\Pascal Analyzer\Projects\Output\MyProj. Of course you are free to select any folder for the output.

See also:

[Introduction](#)

[How to use PAL.EXE and PAL32.EXE](#)

[How to use PALCMD.EXE and PALCMD32.EXE](#)

[Known limitations](#)

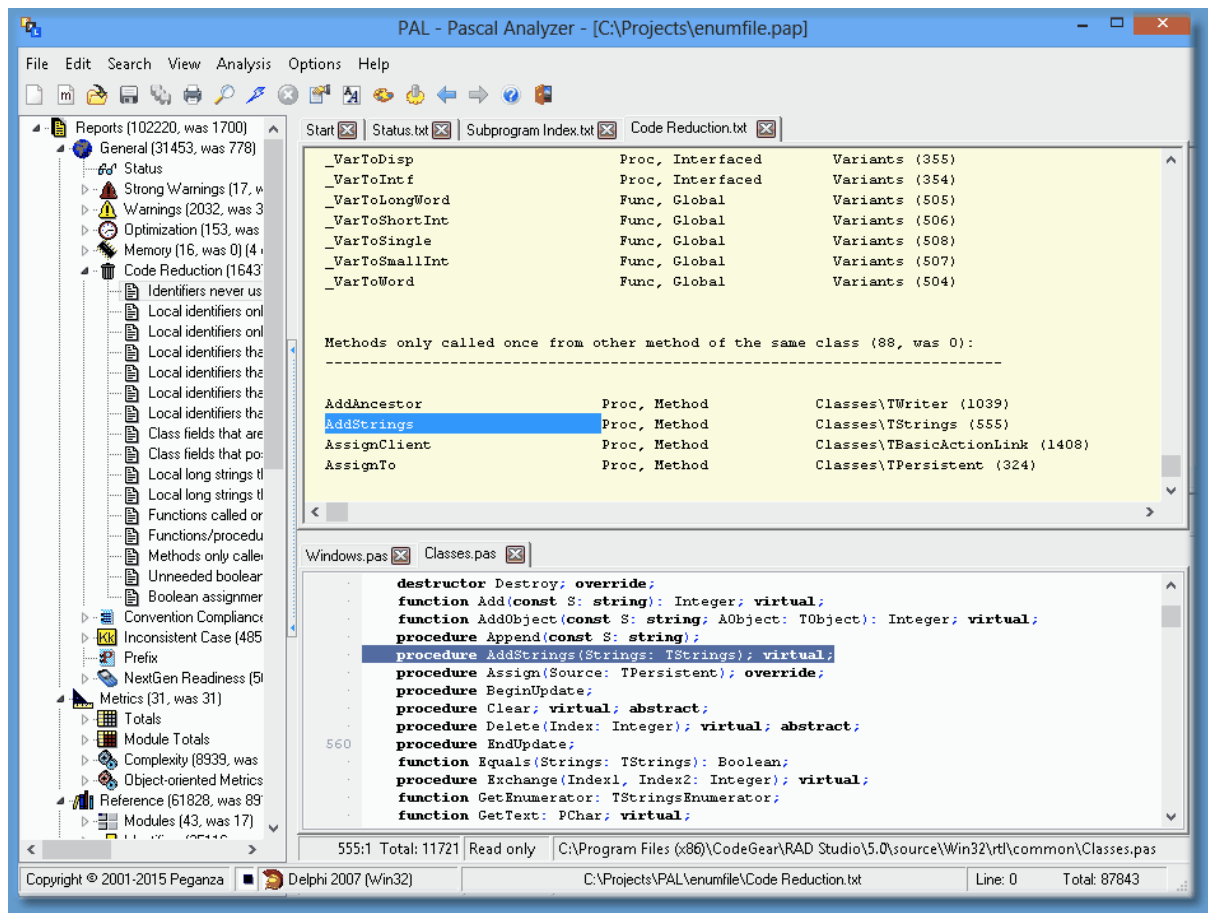
[Command Line Options for PAL.EXE and PAL32.EXE](#)

[Main menu](#)

[Main window](#)

13 Main window

The main interface of PAL consists of a menu, a toolbar, a report list window, an editor window, and a status bar. The toolbar provides speed buttons for common menu items, and the tree view enables quick report selection.



The main window in Pascal Analyzer

There are horizontal and vertical bars dividing the report list, report viewer and source viewer windows. Click on the bar with the mouse and drag it to change the sizes of the windows. If you click the hotspot button in the center of the bar (or F9), the report list window will be temporarily hidden. Click the hotspot button again (or F9) to make it visible. Click on the report window hotspot button (or F8) to toggle on/off viewing of the source window.

Select [Options|Arrange](#) in the menu to choose between possible arrangements for the report list and viewer windows.

Toolbar

The toolbar contains speed buttons for some of the most common menu selections.

Report list

The report list is displayed as a tree structure with reports. At the level below the reports, there are sections for the reports. Some reports only have one section, and in this case only the report level is shown.

If one or more report sections should be turned off (not selected), you will be notified of this by the report caption.

Click on an item in the report list window to move to a report or a report section in the viewer. The report is loaded in the report viewer window.

There is a popup menu associated with the report list that you can invoke by clicking the right mouse button. The popup menu has the following items:

- Display Root (collapse the entire list)
- Display Report Groups (collapse the list and show the report group level)
- Display Reports (collapse the list and show the report level)
- Display Report sections (expand the list fully)

You can also use the mouse-wheel to change font size in the report list.

Report Viewer

PAL presents the reports in a viewer window as read-only text blocks in one or more tab pages. The reports are also written to one or several files in the selected report folder. The format is ordinary text, HTML or XML. Reports are as default opened in multi-tab-pages

HTML format (default) is suitable for web publishing. There are also more ways to customize the layout and appearance of the reports. An advantage with text files is that they are faster to load.

Navigate in the text by scrolling or with the PgDn/PgUp keys, or click on an item in the report list.

The following commands are available in the viewer for reports in text format.

Down	Move the cursor down to the next line
Up	Move the cursor up to the previous line
Left	Move the cursor left one character
Right	Move the cursor right one character
Home	Move the cursor to the beginning of the line
End	Move the cursor to the end of the line
PgDn	Move to the next page

PgUp	Move to the previous page
Ctrl+Home	Move to the top of the text
Ctrl+End	Move to the bottom of the text
Ctrl+MouseWheel	Increase/decrease font size
Ctrl+Shift+0-9	Set the position of bookmark 0-9 to the current position
Ctrl+0-9	Move to the previously set bookmark 0-9
Ctrl+C	Copy the selected text to the clipboard
Shift+Down	Extend the selection down one line
Shift+Up	Extend the selection up one line
Shift+Left	Extend the selection to the left one character
Shift+Right	Extend the selection to the right one character
Shift+Home	Extend the selection to the beginning of the current line
Shift+End	Extend the selection to the end of the current line
Shift+PgDn	Extend the selection down one page
Shift+PgUp	Extend the selection up one page

For HTML and XML reports, PAL uses the browser component in SHDOCVW.DLL, a library included with Windows. This gives the browser in PAL capabilities similar to the browser in MS Internet Explorer.

Double-clicking in reports

If a row or an item on a row is double-clicked, three actions are possible:

- No action
- A source window is opened and the relevant source file is opened and displayed in the built-in editor (default). The cursor is positioned on the line. There are buttons to move back or forward through the list of source locations.
- The source file is opened in the editor window in the Delphi IDE and the cursor is positioned on the relevant line.

Normally, double-click the word that describes the location of the identifier.

```
*****
*                               *
*               Identifiers Report for                               *
*               C:\PROJEKT\RAMVERK\GCACHE.PAS                       *
*****
```

```
Identifiers (1286):
-----
```

_FastCompareText	Func, Interfaced	GAsmCode (7)
_FastIntToStr	Func, Interfaced	GAsmCode (8)
_FastPosIntToStr	Func, Interfaced	GAsmCode (9)
_FastSameText	Func, Interfaced	GAsmCode (10)

In the report above, double-click on "GAsmCode" to locate the corresponding source line.

```
*****
*                               Brief Cross-reference Report for                               *
*                               C: \PROJEKT\RAMVERK\GCACHE. PAS                               *
*                               2002-11-20 19:18:22                                       *
*****

Abbreviations: c=Created f=Freed i=Implemented r=Referenced s=Set u=Unknown v=Varparam

Brief crossreference:
-----

_ FastSameText                Func, Interfaced                GAsmCode (10)
_ GAsmCode                    90i
_ GTools                      782r 805r 2070r 2084r 2092r
```

To locate each reference in this report, click on "782r", "805r" and so on. Double-clicking on a line will in most reports trigger this action. One of the exception however is the [Uses Report](#).

Of course, if you edit the Delphi source code, it may happen that the code line numbers in the reports are not longer valid, for example after deleting or adding lines. Double-clicking in the report will in this case probably locate the wrong line in the source code.

To select the action taken when double-clicking, in the source viewer, go to the [Preferences Dialog](#).

There are some DLL modules that are used to interact with the Delphi IDE, PALWIZ*.DLL, for various Delphi versions.

PALWIZ.DLL	pre-Delphi 2009
PALWIZ2.DLL	Delphi 2010
PALWIZ3.DLL	Delphi 2009
PALWIZ4.DLL	Delphi XE
PALWIZ5.DLL	Delphi XE2
PALWIZ6.DLL	Delphi XE3
PALWIZ7.DLL	Delphi XE4
PALWIZ8.DLL	Delphi XE5
PALWIZ9.DLL	Delphi XE6
PALWIZ10.DLL	Delphi XE7
PALWIZ11.DLL	Delphi XE8
PALWIZ12.DLL	Delphi 10
PALWIZ13.DLL	Delphi 10.1
PALWIZ14.DLL	Delphi 10.2
PALWIZ15.DLL	Delphi 10.3
PALWIZ16.DLL	Delphi 10.4
PALWIZ17.DLL	Delphi 11
PALWIZ18.DLL	Delphi 12

Editor window

The editor window contains one or more tab pages. Each tab page is a separate editor. Use it to make modifications to your source code. To open a source code file, double-click on an identifier in a report. This is only available if you have selected to open an editor window (see [Preferences - Source Code](#)).

Many standard functions are available in the editor, like

- undo/redo
- copy/paste
- syntax highlighting
- modification markers
- line numbers
- supports Unicode

Some common commands:

Ctrl+MouseWheel Increase/decrease font size

Special navigation features:

Double-click on identifier Go to declaration for the identifier

Right-click on identifier Select from reference menu

Ctrl+Enter Open module at cursor in a new editor tab

Ctrl+Alt+C Select from a list of all classes

Ctrl+Alt+I Select from a list of all interfaces

Ctrl+Alt+S Select from a list of subprograms for the module

Ctrl+Alt+U Select from a list of used units for the module

Ctrl+Left Go to previous location in history list

Ctrl+Right Go to next location in history list

Ctrl+Alt+Left Go to previous reference location for the selected identifier (the
very first reference is the declaration for the
identifier)

Ctrl+Alt+Right Go to next reference location for the selected identifier

Ctrl+Alt+Up Go to declaration for subprogram

Ctrl+Alt+Down Go to implementation for subprogram

Alt+Left Go back to previous visited code location

Alt+Right Go forward to next visited code location

Editor files are optionally backed up to a history folder, just like in the Delphi IDE.
See [Preferences|Editor](#).

Editor files also support different formats, like ANSI and UTF-8. The editor will try and

auto detect the format while loading the file.

Status bar

The status bar presents different information, such as the currently loaded report file.

See also:

[How to use PAL.EXE and PAL32.EXE](#)

[How to use PALCMD.EXE and PALCMD32.EXE](#)

[Introduction](#)






[Known limitations](#)

[Command Line Options for PAL.EXE and PAL32.EXE](#)

[Main menu](#)

14 Reports

The purpose of Pascal Analyzer is to generate reports that give information about your source code. Currently there are 52 reports that are generated for normal Pascal Analyzer projects, and four reports that are generated for multi-projects. These reports are categorized in five report groups:

 [General](#)
 [Metrics](#)
 [Reference](#)
 [Class](#)
 [Control](#)

Please note that all reports are not always relevant for all compiler targets. Sometimes some sections are relevant and others not. Versions supported are indicated in the following reference for each report:






BP7	Borland Pascal 7 (16-bits) and earlier
D1	Delphi 1 (16-bits)
D2	Delphi 2
D3	Delphi 3
D4	Delphi 4
D5	Delphi 5
D6	Delphi 6
D7	Delphi 7
D8	Delphi 8 for .NET
D2005	Delphi 2005 for Win32
W	
D2005	Delphi 2005 for .NET
N	
D2006	Delphi 2006 for Win32
W	
D2006	Delphi 2006 for .NET
N	
D2007	Delphi 2007 for Win32
W	
D2009	Delphi 2009 for Win32
W	
D2010	Delphi 2010 for Win32
W	
DXE	Delphi XE for Win32
DXE2	Delphi XE2 for Win32/Win64/OSX
DXE3	Delphi XE3 for Win32/Win64/OSX
DXE4	Delphi XE4 for Win32/Win64/OSX/iOS
DXE5	Delphi XE5 for Win32/Win64/OSX/iOS/Android
DXE6	Delphi XE6 for Win32/Win64/OSX/iOS/Android
DXE7	Delphi XE7 for Win32/Win64/OSX/iOS/Android
DXE8	Delphi XE8 for Win32/Win64/OSX/iOS/Android
D10	Delphi 10 (Seattle) for Win32/Win64/OSX/iOS/Android
D10.1	Delphi 10.1 (Berlin) for

- Win32/Win64/OSX/iOS/Android
- D10.2 Delphi 10.2 (Tokyo) for
Win32/Win64/OSX/iOS/Android/Linux64
- D10.3 Delphi 10.3 (Rio) for
Win32/Win64/OSX/iOS/Android/Linux64
- D10.4 Delphi 10.4 (Sydney) for
Win32/Win64/OSX/iOS/Android/Linux64
- D11 Delphi 11 (Alexandria) for
Win32/Win64/OSX/iOS/Android/Linux64/OSX
ARM
- D12 Delphi 12 (Athens) for
Win32/Win64/OSX/iOS/Android/Linux64/OSX
ARM

Most reports have "Target: All" which means they are relevant for all compiler targets.











Each report group and report is associated with a small picture that is displayed in the report list.

See also:

-  [General Reports](#)
-  [Metrics Reports](#)
-  [Reference Reports](#)
-  [Class Reports](#)
-  [Control Reports](#)

14.1 General Reports

The general reports are:

-  [Status Report](#)
-  [Strong Warnings Report](#)
-  [Warnings Report](#)
-  [Optimization Report](#)
-  [Memory Report](#)
-  [Code Reduction Report](#)
-  [Convention Compliance Report](#)
-  [Inconsistent Case Report](#)
-  [Prefix Report](#)
-  [NextGen Readiness Report](#)

See also:

[Reports](#)

14.1.1 Status Report

Targets: All

PAL always generates this report, regardless of any settings. This report presents important facts about the current analysis, e.g. selected compiler directives and search paths.

The search folders list in the Status Report lists the folders that are set in the [Properties](#) dialog. But added to the list are folders for files that are found by PAL during the parsing process, for instance by following relative paths in the project file. Also added are folders from Delphi's library and browsing paths, if those options are set in the project properties dialog.

The status report also lists source files *found* by PAL, and files *not found*.

```

*****
*                               Status Report for                               *
*                               C: \PROJEKT\RAMVERK\GCACHE. PAS                     *
*****

Overview:
-----

Analyzed by:      PAL - Pascal Analyzer version 3.0.0.0
Licensed to:      NN
Parse speed:      6500 lines in 0.64 seconds (10140 lines/sec).

Main file:        C: \PROJEKT\RAMVERK\GCACHE. PAS
Compiler:         Delphi 7
Files parsed:     Both source and form files
Only main file parsed: No

Search folders:   Components

Excluded for reports: (none)
Unit aliases:     WinTypes=Windows
                  WinProcs=Windows
                  DbiTypes=BDE
                  DbiProcs=BDE
                  DbiErrs=BDE

Predefined:       VER150; MSWINDOWS; WIN32; CPU386; ConditionalExpressions

Conditional defines: (none)
Global switches:  A+; B-; C+; D+; E-; F+; G+; H+; I+; J-; K+; L+; M-; N+; O+; P+; Q-
                  R-; S+; T-; U-; W-; V+; X+; Y-; YD; Z-

```

Double-clicking on a line in the Status Report gives different actions:

If clicking on a file path, the file will open in the built-in editor.
Clicking on a folder name, will open the folder in Windows Explorer.

The Status Report, as all reports, is in UTF-8 format.

See also:

 [General Reports](#)

14.1.2 Strong Warnings Report

Targets: All except BP7 (partly)

This report generates warnings that help you identify especially severe errors. Those are errors that can cause runtime failures ("showstoppers"), or erroneous results in your application.

Property access in read/write methods (STWA1)

(Not relevant for BP7)

This section reports locations where properties are referenced in read/write methods, like for example:

```
1  ..  
2  property MyProp : Integer read GetMyProp write SetMyProp;  
3  ..  
4  
5  function TMyClass.GetMyProp : Integer;  
6  begin  
7      Result := MyProp; // error, correct is: Result := FMyProp;  
8  end;  
9  
10 procedure TMyClass.SetMyProp(Value : Integer);  
11 begin  
12     MyProp := Value; // error, correct is: FMyProp := Value;  
13 end;  
14
```

These sorts of errors can cause infinite recursion.

Ambiguous unit references (STWA2)

This sections lists identifiers with ambiguous unit references. Consider this example:

```
1  program MyProg;
2
3  uses
4      A, B;
5
6  begin
7      writeln('Value='+TheValue);
8  end.
9
10 unit A;
11
12 interface
13
14 const
15     TheValue = 'Hello';
16
17 implementation
18
19 end.
20
21 unit B;
22
23 interface
24
25 const
26     TheValue = 'Goodbye';
27
28 implementation
29
30 end.
31
```

What will be the output from the program? In this case, it will be "Goodbye", because the last unit listed in the uses clause will have precedence.

The reference to TheValue is ambiguous or unclear, so it will be listed in this report section. Consider what happens if originally only unit "A" was listed in the uses clause. Then the output would be "Hello". If then maybe another programmer without any sense of danger will add "B" to the uses clause, the output will be changed.

You should prefix the reference, like "B.TheValue", to avoid any uncertainty.

Subprogram calls itself unconditionally (STWA3)

This sections lists subprograms that call themselves unconditionally. This will lead to infinite recursion and stack failure at runtime if the subprogram is called: Consider this example:

```
1  procedure Proc;
2  begin
3      ..
4      Proc;
5      ..
6  end.
7
```

Currently overload subprograms are not examined.

Index error (STWA4)

This sections lists locations in your code with an index error.

Example:

```
1 procedure Proc;  
2 var  
3   X : Integer;  
4   Arr : array[0..1] of Integer;  
5 begin  
6   X := 555;  
7   ..  
8   Arr[X-2] := 0; // index error  
9 end;
```

If the code had been instead "Arr[553]" (an explicit value), the compiler would have halted on this line. But for a variable, it does not.

Possible bad pointer usage (STWA5)

This section lists locations in your code where a pointer possibly is misused. For example a pointer that has been set to nil and further down in the code is dereferenced.

Example:

```
1 type  
2   TMyClass = class  
3     procedure MyProc;  
4   end;  
5  
6 procedure TMyClass.MyProc;  
7 begin  
8   ..  
9 end;  
10  
11 procedure Proc;  
12 var  
13   Obj : TMyClass;  
14 begin  
15   Obj := TMyClass.Create;  
16   ..  
17   Obj := nil;  
18   ..  
19   Obj.MyProc; // error, Obj is nil here  
20 end;
```

Possible bad typecast (for objects: consider using "as") (STWA6)

This section lists locations in your code with a possibly bad typecast. If you use the "as" operator, an exception will instead be raised. Otherwise there may be access violations and errors in a totally different code location, which is not very easy to track down.

Example:

```

1  type
2    TFruit = class
3      ..
4    end;
5
6    TAnimal = class
7      ..
8    end;
9
10 procedure Proc;
11 var
12   Monkey : TAnimal;
13   Banana : TFruit;
14 begin
15   ..
16   Monkey := TAnimal(Banana); // bad typecast, a fruit cannot be typecast to an animal
17 end;

```

In the example above, the last line could better be written (although still faulty!) as

```
Monkey := Banana as TAnimal;
```

This should result in an exception. But this is still preferable; instead of letting the code proceed resulting maybe in access violations later in a totally unrelated part of the code.

Also situations where a "bigger" type is typecast to a "smaller", will trigger a warning. For example "Ch := Char(I)" where Ch is of type **Char** and I is of type **Integer**. This may of course be totally valid if you make sure that I is not too big.

For-loop with possible bad conditions (STWA7)

This section lists locations in your code where for loop has any of these conditions:

```

1  ..
2
3  for I := 0 to SL.Count do // SL.Count-1 intended?
4  begin
5    ..
6  end;
7
8  for I := 1 to SL.Count-1 do // SL.Count intended?
9  begin
10 end;
11
12 ..
13

```

Bad parameter usage (same identifier used for different parameter) (STWA8)

This section lists locations in your code where a call to a subprogram is made with bad parameters. The situation occurs when the called subprogram has an "out" parameter plus at least one another parameter. The identifier passed is used for both these parameters. Because an "out"-parameter is cleared in the called subprogram this will give unexpected results for reference-counted variables like strings and dynamic arrays.

```
1  ..
2
3  procedure Proc(const Value : string; out ReturnValue : string);
4  begin
5      ReturnValue := '555'+Value;
6  end;
7
8  procedure Caller;
9  var
10     S : string;
11  begin
12     S := '111';
13     Proc(S, S); // S will have '555' upon return, not '555111' which you would expect
14  end;
15
16  ..
```

Generic interface has GUID (STWA9)

This section lists generic interface types that declare a GUID:

```
1  ..
2
3  type
4      IMyInterface<t> = interface
5          ['{C9BC756B-6B30-4C44-B237-552AFBA5697C}']
6          ..
7      end;
8
9  ..
10
```

The problem with this is that all generic types created from this interface, like `IMyInterface<Integer>` and `IMyInterface<string>` will share the same GUID. This will cause type casting to malfunction.

Interface lacks GUID (STWA10)

This section lists interface types that lacks a GUID.

See also:

[🌐 General Reports](#)

14.1.3 Warnings Report

Targets: All

This report contains several sections that present different types of warnings. These warnings point to possible anomalies or errors in your source code. Because PAL does not require full access to all source code, some of these warnings may however turn out to be just false.

Interfaced identifiers that are used, but not outside of unit (WARN1)

This section lists all identifiers that are declared in the *interface* section of a unit, and that are used in the unit, but not outside the unit. You should declare these identifiers in the *implementation* section of the unit instead.

This section is also generated for multi-projects.

Restrictions:

Interfaced identifiers that are not used at all are not listed. These identifiers are already listed in the "Identifiers never used" section in the [Code Reduction Report](#).

Recommendation:

Declare these identifiers in the *implementation* section of the unit, to avoid unnecessary exposure.

Interfaced class identifiers that are public/published, but not used outside of unit (WARN2)

This section lists all identifiers that are members of a class, and are declared with the *public/published* directive, but not used outside of the unit.

This section is also generated for multi-projects.

Recommendation:

Declare these identifiers with the *private/protected* directive instead.

Variables that are referenced, but never set (WARN3)

This section lists all declared and referenced variables that never are set. Possibly this is an error, but the reason could also be that the variable is set in code that is not seen by the parser.

Restrictions:

Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, and are changed whenever the other variable changes.

Recommendation:

Examine why these variables are referenced, but never set. False warnings may be

generated in some cases for null-terminated strings, where the actual pointer (PChar) is not set, but when the contents of the buffer pointed to is indeed changed.

Variables that are referenced, but possibly never set (ref/set by unknown subprograms) (WARN4)

This section lists all variables that are declared and referenced but never set. They are referenced in unknown fashion, and the parser is unable to determine whether they are set or just referenced in these locations.

Restrictions:

Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, and are changed whenever the other variable changes.

Recommendation:

Examine why these variables are referenced, but never set. False warnings may be generated in some cases for null-terminated strings, where the actual pointer (PChar) is not set, but when the contents of the buffer pointed to is indeed changed.

Variables that are set, but never referenced (WARN5)

This is a list of all variables that are set but never referenced. Either these variables are unnecessary or something is missing in the code, because it is meaningless to set a variable and then never reference, or use it.

Restrictions:

Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, and are changed whenever the other variable changes.

Recommendation:

Examine why these variables are set, but never referenced.

Variables that are set, but possibly never referenced (ref/set by unknown subprograms) (WARN6)

This is a list of all variables that are set but never referenced. The variables are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. They are either unnecessary or something is missing in the code, because it is meaningless to set a variable and then never reference, or use it.

Restrictions:

Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, they are changed whenever the other variable changes.

Recommendation:

Examine why these variables are set, but never referenced. Also, try to make more source code available to PAL.

Local variables that are referenced before they are set (WARN7)

This is a list of all local variables that are referenced before they are set. Probably this is an error, because the values of these identifiers are undefined before they are set. An exception is long strings that are not examined, because they are implicitly initialized upon creation.

Restrictions:

Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, they are changed whenever the other variable changes.

Recommendation:

Examine why these variables are referenced before they are set.

Example:

```
1  procedure MyProc;
2  var
3    I : integer;
4  begin
5    if I = 55 then // !! I is undefined
6    begin
7      ..
8    end;
9
10   I := 55;
11   ..
12 end;
```

Pascal Analyzer also examines local subprograms that are called. Consider this scenario:

Example:

```
1  procedure Proc;
2  var
3    I : integer;
4
5    procedure InnerProc;
6    begin
7      if Condition then
8        if I < 5 then
9          ..
10     end;
11
12   begin
13     InnerProc;
14     ..
15     I := 55;
16 end;
```

This code triggers a warning, because the local variable I is first referenced by InnerProc.

The call to InnerProc occurs before I is set in the main body of Proc. Even if I is only referenced when Condition evaluates to True (in InnerProc), this must happen at some occasion, otherwise that check would be pointless.

A usual situation which triggers this warning is when an non-initialized variable is passed as a parameter to a function. The function signature declares the parameter as a var-parameter. Changing the parameter to an out-parameter (if possible), avoids this warning.

Local variables that may be referenced by unknown subprogram before they are set (WARN8)

This is a list of all local variables that are referenced before they are set. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Probably this is an error because the values of these identifiers are undefined before they are set. An exception is long strings that are not examined, because they are implicitly initialized to empty strings when created.

Restrictions:

Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, they are changed whenever the other variable changes.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4  begin
5    UnknownProc(I);
6
7    if I = 55 then // !! I may be undefined
8    begin
9      ..
10   end;
11
12   I := 55;
13   ..
14 end;
```

Var parameters that are used, but never set (WARN9)

This is a list of all *var* parameters that are used but never set in the subprogram they belong to. Although this is not an error, it may be an indication that something is wrong with your code. Otherwise, you may omit the *var* keyword, or change it to a *const* parameter.

Example:

```
1 | procedure MyProc(var I : Integer);
2 | begin
3 | ..
4 |   if I = 5 then // !! I is not set
5 |   begin
6 |   ..
7 |   end;
8 | ..
9 | end;
```

Restrictions:

Parameters to event handlers are not reported.

Var parameters that are used, but possibly never set (ref/set by unknown subprograms (WARN10))

This is a list of all *var* parameters that are used but never set in the subprogram they belong to. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Although this is not an error, it may be an indication that something is wrong with your code. Otherwise, you may omit the *var* keyword, or change it to a *const* parameter.

Example:

```
1 | procedure MyProc(var I : Integer);
2 | begin
3 | ..
4 |   UnknownProc(I);
5 | ..
6 |   if I = 5 then // !! I may not be set
7 |   begin
8 |   ..
9 |   end;
10 | ..
11 | end;
```

Restrictions:

Parameters to event handlers are not reported.

Value parameters that are set (WARN11)

This is a list of all value parameters that are set in the subprogram they belong to. Although this is permitted by the compiler, it may not be what you intended. If you want to really change the variable, use the *var* directive instead.

Example:

```
1 | procedure MyProc(I : Integer);
2 | begin
3 | ..
4 |   I := 5; // !! I is changed
5 | ..
6 | end;
```

Value parameters that are possibly set (ref/set by unknown subprogram) (WARN12)

This is a list of all value parameters that are set in the subprogram they belong to. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Although this is permitted by the compiler, it may not be what you intended. If your intention is to really change the variable, use the *var* directive instead.

Example:

```
1  procedure MyProc(I : Integer);  
2  begin  
3  ..  
4      UnknownProc(I); // !! I may be changed  
5  ..  
6  end;
```

Interfaces passed as parameters without "const" directive (WARN13)

This is a list of all parameters that are of interface type and passed without "const" directive.

Variables with absolute directive (WARN14)

This is a list of all variables that are declared with the *absolute* directive keyword. You should watch these variables carefully, since they may potentially overwrite memory.

Example:

```
1  procedure MyProc;  
2  var  
3      I : Byte;  
4      K : Integer absolute I;  
5  begin  
6  ..  
7      K := MaxInt; // !! I is overwritten  
8  end;
```

Recommendation:

Examine absolute variables carefully, and make sure that they do not overwrite memory.

Constructors/destructors without calls to inherited (WARN15)

This is a list of all constructors and destructors that never call their inherited constructor/destructor. This call is often required, so that the object can be correctly created or destroyed. For a class descending directly from *TObject*, the inherited call in

the constructor is not needed, since the constructor in *TObject* does not actually do anything. There is no guarantee though that the constructor will be empty in future versions. If the constructor/destructor does not call inherited itself, but calls another constructor/destructor that calls inherited, there will be no warning.

Recommendation:

Call the inherited constructor as the first statement in the constructor, and as the last statement in the destructor.

Destructors without override directive (WARN16)

This is a list of all destructors that miss the *override* directive keyword. Normally this directive must be set, or a call to the *Free* method would never be successful. This is because *Free* calls the destructor.

Limitation:

Not examined for Borland Pascal 7. For other targets, old-style objects are never reported, because in this case the *override* keyword is not allowed.

Classes with more than one destructor (WARN17)

This is a list of all classes that have more than one destructor declared. To declare more than one destructor is usually pointless and should be avoided.

Function result not set (WARN18)

This is a list of all functions where the result value is not always set. It may be set for some but not all possible code paths. Although this is acceptable for the compiler, it implies an error in the code. Maybe the function could be implemented as a procedure instead, if the result value is not needed.

Functions that return long strings are not examined. Those strings are zero-initialized by the function.

Recommendation:

Check these functions and examine if they should be implemented as procedures instead.

Recursive subprograms (WARN19)

This is a list of all subprograms (procedures and functions) that are recursive (call themselves). Recursive subprograms are difficult to implement, and should be given

extra attention.

Recommendation:

Check these subprograms and make sure that they cannot fall into infinite recursion.

Dangerous Exit-statements (WARN20)

This is a list of all locations with dangerous *Exit*-statements. These *Exit*-statements may leave a whole block of code that is never executed (dead code). Every unconditional (not within an if-statement) *Exit*-statement is considered dangerous in this respect. Exit-statements within except-blocks are considered as safe, however.

There are situations when a developer inserts Exit-commands just for testing purposes, for example to quit a function without executing a block of code. This report section catches those locations where the Exit-commands have not been removed.

Example:

```
1  procedure MyProc;  
2  begin  
3  ..  
4  Exit;  
5  ..  
6  Proc(X); // !! never executed  
7  ..  
8  end;
```

Dangerous Raise (WARN21)

This is a list of all locations with dangerous raise commands. These raise-commands may leave a whole block of code that is never executed (dead code). Every unconditional (not within an if-statement) raise-command is considered dangerous in this respect. Raise-commands within except-blocks are considered as safe, however.

There are situations when a developer inserts raise-commands just for testing purposes, for example to quit a function without executing a block of code. This report section catches those locations where raise-commands have not been removed.

Example:

```
1  procedure MyProc;  
2  begin  
3  ..  
4  raise Exception.Create('Leave here');  
5  ..  
6  Proc(X); // !! never executed  
7  ..  
8  end;
```

Dangerous Label-locations inside for-loops (WARN22)

This is a list of all locations with dangerous *goto*-labels. These labels are located inside *for*-loops. In the case of a *for*-loop, this is especially dangerous, since the loop variable will have an undefined value.

Example:

```
1  procedure MyProc;
2  label
3    MyLabel;
4
5  begin
6    ..
7    for I := 0 to NumItems-1 do
8      begin
9        ..
10       MyLabel:
11         ..
12         end;
13         ..
14         if Cond then
15           goto MyLabel; // !! dangerous
16         ..
17       end;
```

Dangerous Label-locations inside repeat/while-loops (WARN23)

This is a list of all locations with dangerous *goto*-labels. These labels are located inside *repeat/while*-loops. If the loop counter is considered, this may work just fine, but these labels should be given extra attention.

Example:

```
1  procedure MyProc;
2  label
3    MyLabel;
4
5  begin
6    ..
7    while I < NumItems-1 do
8      begin
9        ..
10       MyLabel:
11         ..
12         end;
13         ..
14         if Cond then
15           goto MyLabel; // !! dangerous
16         ..
17       end;
```

Possible bad object creation (WARN24)

This is a list of all locations in the code where an object possibly is created in a bad fashion.

Example:


```
1 procedure Proc;  
2 var  
3   Bmp : TBitmap;  
4  
5 begin  
6   ..  
7   Bmp.Create // !! Bmp := TBitmap.Create  
8   ..  
9 end;
```

This is an error!

Example:

```
1 procedure Proc2;  
2 begin  
3   TNode.Create(Parent);  
4 end;
```

PAL reports this as an error, since the reference to the new object is not assigned to a variable. It could possibly be a mistake. However, in a situation where the object is inserted into a list managed by "Parent", it is not a mistake. This is the case for the common TTreeView control.

Limitation:

Not examined for Borland Pascal 7. For other targets, old-style objects are never reported, because in this case the *override* keyword is not allowed.

Bad thread-local variables (WARN25)

This is a list of all thread-local variables (declared with the "threadvar" keyword) with bad declarations. Reference-counted variables (such as long strings, dynamic arrays, or interfaces) are not thread-safe and should not be declared with "threadvar". Also, do not create pointer- or procedural-type thread variables.

Limitation:

Not examined for Borland Pascal 7 and Delphi 1

Example:

```
1 threadvar  
2   S : string;           // !! these are all bad thread-local variables  
3   P : Pointer;  
4   R : array of Integer;  
5   X : IMyInterface;  
6   W : TProcedure;
```

Instance created of class with abstract methods (WARN26)

This is a list of all locations where instances of classes with abstract methods are created. Such classes should serve as ancestor classes only.

Example:

```
1  type
2    TAbstractClass = class
3      procedure AbstractMethod; virtual; abstract;
4      ..
5    end;
6    ..
7  var
8    Obj : TAbstractClass;
9
10 begin
11   Obj := TAbstractClass.Create; // triggers a warning
12   ..
13 end;
```

Empty code blocks and short-circuited statements (WARN27)

This is a list of all empty code blocks and short-circuited statements. Short-circuited statements are of these kinds:

Example:

```
1  if x then;
2
3  for I := 0 to 5 do;
4
5  while x do;
```

These statements may be mistakes.

Empty case labels (WARN28)

Example:

```
1  case X of
2    1 ;;
3    2 : begin
4        end;
5  end;
```

The first case-branch is empty, which may be a mistake.

Short-circuited for-statements (WARN29)

Example:

```
1  for I := 1 to 5 do;
```

Short-circuited if/case-statements (WARN30)

Example:

```
1 | if x then;
```

Also short-circuited else-branches are reported, both in if- and case-statements.

Short-circuited on-statements (WARN31)

Example:

```
1 | on Exception do;
```

Short-circuited repeat-statements (WARN32)

Example:

```
1 | repeat until X;
```

Short-circuited while-statements (WARN33)

Example:

```
1 | while X do;
```

Empty except-block (WARN34)

Example:

```
1 | try
2 |   DoSomething;
3 | except
4 | end;
```

Empty finally-block (WARN35)

Example:

```
1 | try
2 |   DoSomething;
3 | finally
4 | end;
```

Forward directive in interface (WARN36)

Even if a forward directive is allowed by at least some versions of the Pascal/Delphi compiler, they are unnecessary and should be avoided.

Empty subprogram parameter list (WARN37)

Somewhat surprisingly, this code is accepted by at least some versions of the Pascal/Delphi compiler:

Example:

```
1  procedure Proc();
2  begin
3  ..
4  end;
```

Ambiguous references in with-blocks (WARN38)

This section reports locations where a valid references to an identifier inside a with-block could be mixed up with another identifier declared in the same scope. It is not an error, but just means that you should check that the code does what you intended.

Example:

```
1  var
2    Title : string;
3
4  type
5    TMyRec = record
6      Title : string;
7    end;
8
9  var
10   Rec : TMyRec;
11
12  begin
13    ..
14    with Rec do
15      begin
16        ..
17        Title := 'Hello';
18      end;
19    end;
```

The record field referenced in the with-block could be mixed up with the global *Title*. Maybe the programmer instead intended to set the global *Title* identifier.

Classes without overrides of abstract methods (WARN39)

This section lists classes that do not override abstract methods in ancestor classes. If a

method is declared abstract in an ancestor class, it must be overridden in descendant classes. Otherwise, calling the method for the descendant class will result in a runtime error.

Local for-loop variables read after loop (WARN40)

This section lists for-loop variables that are read in code after the loop. Their values are undefined, and thus it is not recommended to use their values.

Local for-loop variables possibly read after loop (WARN41)

This section lists for-loop variables that *possibly* are read in code after the loop. Their values are undefined, and thus it is not recommended to use their values.

For-loop variables not used in loop (WARN42)

When a for-loop variable is *not* used in the loop, it may be a coding error.

Non-public constructors/destructors (WARN43)

This section lists constructors/destructors that are non-public.

Functions called as procedures (WARN44)

This section lists locations in the source code where functions are called as procedures, that is without using the result value. Maybe this is a coding error, and the function should really be called as a function instead.

Mismatch property read/write specifiers (WARN45)

This section lists property declarations with mismatch between read/write specifiers, like

```
1 | property IntValue2 : Integer read FIntValue2 write FIntValue3;
```

This is probably a coding error.

Local variables that are set but not later used (WARN46)

This section lists local variables that are set but not later used further down in the code, like

```

1  procedure MyLocal;
2  var
3    X, Z : Integer;
4  begin
5    X := 555;
6    Z := 33;
7    DoIt(X);
8
9    X := 333; // not read below this...
10
11   if Z > 888 then
12     DoIt(Z);
13 end;
..

```

Duplicate lines (WARN47)

This section lists locations in the source code where a line is duplicated, that is when two lines immediately following each other, have the same content.

```

1  ..
2  begin
3    AddProc(nil);
4    AddProc(nil);
5  end;
6  ..

```

The check is done without case-sensitivity, so ...

```

1  ..
2  begin
3    AddProc(nil);
4    AddPRoc(nil);
5  end;
6  ..

```

.. are considered to be duplicate.

But for differences within string literals, the lines below..

```

1  ..
2  case AChar of
3    'ä': Result := 'ae';
4    'Ä': Result := 'Ae';
5  end;
6  ..

```

.. in the case-structure are considered NOT to be duplicate.

Duplicate class types in except-block (WARN48)

This section lists locations in the source code where an except-block contains more than one handler for the same class type.

Like in this code:

```
1  ..
2  try
3  ....
4  except
5      on E:ExceptionClass1 do
6          ExceptionHandler1;
7      on E:ExceptionClass1 do
8          ExceptionHandler2;
9  end;
10 ..
```

Redeclared identifiers from System unit (WARN49)

This section lists identifiers that use the same name as an identifier from the System.pas unit for the compiler target. Although allowed, at least it is a source for confusion when maintaining the code.

```
1  var
2      Pos : Integer; // conflicts with System function Pos()
3  ..
4  function BlockRead : boolean; // conflicts with System procedure BlockRead()
5  ..
```

Identifier with same name as keyword/directive (WARN50)

This section reports identifier with names that conflict with keywords/directives. Although allowed, is a source for confusion when maintaining the code, and sharing it with others.

Out parameter is read before set, or never set (WARN51)

This section reports parameters marked with the "out" directive that are read before set in the function/procedure, or never set. An "out" parameter is just a placeholder for a return value. The function should not assume that its initial value has any meaning.

Possible bad assignment (WARN52)

This section reports occurrences of assignments to smaller from bigger, possibly resulting in data loss. It will also report situations where for example UInt32 is assigned to Int32, where the

range of the types do not fully overlap.

Example:

```
1  var
2    B, Delta : Byte;
3    I : Integer;
4    S : string;
5    T : AnsiString;
6  begin
7    ..
8    B := I; // triggers warning
9    T := S; // triggers warning
10   ..
11  end;
```

Mixing interface variables and objects (WARN53)

This section reports locations in your code with assignments between objects and interface variables. Normally, unless you really know what you are doing, it is a bad idea to mix interfaces and objects. The reason is that the reference counting mechanism of interfaces can be disturbed, leading to access violations and/or memory leaks.

Example:

```
1  type
2    IIntf = interface
3      end;
4
5    TIntf = class(TInterfacedObject, IIntf)
6      end;
7
8  procedure Proc;
9  var
10   Obj : IIntf;
11   X : TIntf;
12  begin
13   Obj := TIntf.Create;
14   X := TIntf(Obj); // not OK, mixes objects and interfaces
15   ..
16  end;
```

Set before passed as out parameter (WARN54)

This section reports locations in your code where a variable is set and then passed as an "out" parameter to a function.

Because the "out" parameter will be set in the called function without being read first, it is at least pointless to set it before it is passed. It may also indicate some misunderstanding about the code.

It is recommended to check if it is meaningful to set the variable before passing it. If not, remove the assignment, or else modify the signature of the called function from "out" to "var".

Example:

```
1  procedure Proc(out Param : Integer);
2  begin
3    ..
4  end;
5
6  procedure Test;
7  var
8    I : Integer;
9  begin
10   ..
11   I := 555; // this is meaningless!
12   Proc(I);
13   ..
14 end;
```

See also our [blog article](#) about out parameters.

Redeclares ancestor member (WARN55)

This section lists class fields or methods that redeclare ancestor members with the same name. This may lead to confusion about which member is actually referenced.

Example:

```
1  type
2    TAncestor = class
3      private
4        FObj : TObject;
5        procedure Proc;
6      end;
7
8    TDescendant = class(TAncestor)
9      protected
10       FObj : TObject; // redeclares!
11       procedure Proc; // redeclares!
12     end;
```

Parameter to FreeAndNil is not an object (WARN56)

This section reports locations in your code where FreeAndNil takes a parameter which is not an object, for example an interface variable. This may lead to access violations. Unlike Free, the compiler will not complain.

Example:

```
1  type
2    IMyInterface = interface
3      ..
4    end;
5
6    TMyClass = class(TInterfacedObject, IMyInterface)
7    end;
8
9    procedure Proc;
10   var
11     Intf : IMyInterface;
12     Obj : TMyClass;
13   begin
14     Intf := TMyClass.Create;
15     FreeAndNil(Intf); // not OK!
16
17     Obj := TMyClass.Create;
18     FreeAndNil(Obj); // OK
19   end;
```

Enumerated constant missing in case structure (WARN57)

This section lists locations in your code where a case statement does not list all possible values of an enumerated type. This is probably most often as intended, but it may also point out an error in the code.

Example:

```
1  type
2    TChessPiece = (cpPawn, cpKnight, cpBishop, cpRook,
3                  cpQueen, cpKing);
4
5  procedure Move(Piece : TChessPiece;
6                FromSquare, ToSquare : TSquare);
7  begin
8    case Piece of
9      cpPawn : ..;
10     cpKnight : ..;
11     cpBishop : ..;
12     cpRook : ..;
13     cpQueen : ..;
14   end;
15
16   ..
17 end;
```

In the code above, **cpKing** is missing from the case structure, and will trigger a warning.

If you want to suppress warnings for a case-structure, just use PALOFF on the same line as the "case" keyword.

Mixed operator precedence levels (WARN58)

This section lists locations in your code where operators of different levels are mixed. Operators are in Object Pascal evaluated from left to right, unless parentheses are used. Operators of level 1 are evaluated before operators of level 2 etc.

Level 1: @, not

Level 2: *, /, div, mod, and, shl, shr, as

Level 3: +, -, or, xor

Level 4: =, <>, <, >, <=, >=, in, is

Example:

```
1 | X := A + B * 5;  
2 | // evaluated as X := A + (B * 5)  
3 |  
4 | B := BoolA and BoolB or C;  
5 | // evaluated as B := (BoolA and BoolB) or C
```

Mixing operators is perfectly valid but you will find that your code is clearer and easier to understand if you insert parentheses. Then you do not have to think about operator precedence.

Explicit float comparison (WARN59)

This section lists locations in your code where floating point numbers are directly compared. It is considered not secure to compare floating numbers directly. Instead use functions in Delphi's System.Math unit, like IsZero and SameValue.

Example:

```
1 | procedure CalculateProfit;  
2 | var  
3 |   Yield, Income : Double;  
4 | begin  
5 |   Yield := GetYield;  
6 |   Income := GetIncome;  
7 |  
8 |   if Yield = Income then // not secure!  
9 |   begin  
10 |    ..  
11 |   end;  
12 | end;
```

In the example above, use instead SameValue function from System.Math unit.

Condition evaluates to constant value (WARN60)

This section lists locations in your code where a condition evaluates to a constant value.

Example:

```
1  procedure MyProc;  
2  var  
3      X : Integer;  
4  begin  
5      X := 0;  
6  
7      ..  
8  
9      if X+5 = 15 then // if-condition has always the same value ...  
10     begin  
11     end;  
12 end;  
13
```

Assigned to itself (WARN61)

This section lists locations in your code where a variable has been assigned to itself. Even if this assignment is harmless, it makes no sense. It may indicate other problems with the code, so you should check the surrounding code.

Possible orphan event handler (WARN62)

This section lists class procedures in your code that look like event handlers. But they are not connected to any control in the corresponding DFM-file.

This section is analyzed only if DFM files are found.

Mismatch 32/64-bits (WARN63)

This section reports locations where 32-bits (or smaller) variables are passed as 64-bits parameters (or vice versa).

In many cases this is totally harmless, but consider the case where a 32-bits pointer is passed to a function that expects a 64-bits pointer.

Also if there is a mismatch when assigning values, it will be reported.

See also:

 [General Reports](#)

14.1.4 Memory Report

Targets: All except BP7

The Memory Report helps you find possible memory leaks in your code.

Local objects with unprotected calls to Free (MEMO1)

This section reports locations where calls to Free (and FreeAsNil or Release) are not done in try-finally blocks. Failure to wrap a try-finally block around a memory deallocation could result in a memory leak. The report does not list locations in FormDestroy and FormClose events, because these are normally called when a form is destroyed. Neither does it report calls to Free from a finalization block. Also an object that is freed in a try-except block is not reported.

Non-local objects with unprotected calls to Free (MEMO2)

Like the previous section, but for non-local objects.

Objects created in try-structure (MEMO3)

This section lists lists locations where an object is created inside a try-structure, like:

```
1  try
2      Obj := TMyObject.Create;
3      ..
4  finally
5      Obj.Free;
6  end;
```

Here, Obj should be created before the “try”, otherwise Obj.Free will be called even if the object fails to create, possibly causing a runtime error.

Unbalanced Create/Free (MEMO4)

This section reports objects that are not created and freed the same number of times. This can indicate an error, like in the following example:

```
1  procedure LocalProc;
2  var
3      Obj : TMyClass;
4  begin
5      Obj := TMyClass.Create;
6      Obj.DoSomething;
7  end;
```

Here, the locally declared object Obj is never freed, so this code will cause a memory leak.

Local objects that are created more than once without being freed in-between (MEMO5)

This section reports objects that are created more than once (in a row) without being freed in-between.

This leads to memory leakage, like in the following example:

```
1 | procedure LocalProc;  
2 | var  
3 |   Obj : TMyClass;  
4 | begin  
5 |   Obj := TMyClass.Create;  
6 |  
7 |   try  
8 |     Obj := TMyClass.Create;  
9 |  
10 |    try  
11 |      ..  
12 |      finally  
13 |        FreeAndNil(Obj);  
14 |      end;  
15 |  
16 |      ..  
17 |      finally  
18 |        FreeAndNil(Obj);  
19 |      end;  
20 | end;
```

Here, the locally declared object Obj is only freed once, which causes a memory leak.

Local objects that are referenced before being created (MEMO6)

This section reports objects that are referenced before being created.

This leads to an exception, like in the following example:

```
1 | procedure LocalProc;  
2 | var  
3 |   Obj : TMyClass;  
4 | begin  
5 |   Obj.Field := 55; // !! gives an AV  
6 | end;
```

Objects that PAL cannot determine have been created at all, are not reported, only those cases where the object has been created further down in the code. Otherwise there should be many false positives.

Local objects that are referenced after being freed (MEMO7)

This section reports objects that are freed but referenced further down in the code.

This leads to an exception, like in the following example:

```
1 procedure LocalProc;
2 var
3   Obj : TMyClass;
4 begin
5   Obj := TMyClass.Create;
6
7   try
8   ..
9   finally
10    FreeAndNil(Obj);
11  end;
12
13  Obj.Field := 55; // !! Obj is already freed
14 end;
```

See also:

[🌐 General Reports](#)

14.1.5 Optimization Report

Targets: All

This report pinpoints elements of the code that you can improve, resulting in better performance. With better performance, we here mean faster execution, not necessarily smaller code.

Missing “const” for unmodified string parameter (OPT11)

This is a list of all string parameters that you can declare with the *const* directive, resulting in better performance since the compiler can assume that the parameter will not be changed. For example, for a long string the reference count for the string does not need to be updated on entry and exit to the function. For other types of strings, like WideString, the string may have to be copied when passed to the function.

Example:

```
1 procedure MyProc(S : string);
2 begin
3   ..
4   if S = 'Foo' then // !! S is not changed
5   begin
6     ..
7   end;
8   ..
9 end;
```

In this case, the parameter S should have the *const* directive, since it is never changed in the procedure. The compiler can generate code that is more efficient.

No warning is given for methods that are marked with the "override" directive. This is because they must follow the parameter list that the overridden method has.

Missing “const” for unmodified record parameter (OPT12)

This is a list of all record parameters that you can declare with the *const* directive, resulting in better performance since the compiler can assume that the parameter will not be changed. Generally, if the parameter is larger than 4 bytes, and it doesn't need to be altered in the subroutine, *const* is more efficient. Also, it is a good idea to use *CONST* on parameters that aren't intended to be altered, so the compiler can catch those errors for you.

Example:

```

1  type
2    TRec = record
3      X : Integer;
4    end;
5
6  procedure MyProc(R : TRec);
7  begin
8    ..
9    if R.X = 5 then // !! R is not changed
10   begin
11     ..
12   end;
13   ..
14 end;
```

In this case, the parameter R should have the *const* directive, since it is never changed in the procedure. The compiler can generate code that is more efficient.

No warning is given for methods that are marked with the "override" directive. This is because they must follow the parameter list that the overridden method has.

Missing "const" for unmodified array parameter (OPTI3)

This is a list of all array parameters that you can declare with the *const* directive, resulting in better performance since the compiler can assume that the parameter will not be changed.

No warning is given for methods that are marked with the "override" directive. This is because they must follow the parameter list that the overridden method has.

Array properties that are referenced/set within methods (OPTI4)

This is a list of all array properties that are referenced or set within methods of a class. Methods that include a reference to the property are listed.

For performance reasons it is faster to directly access the private array field. However, if the Get- or Set-method performs side effects, it makes sense to access the property.

For simple non-array properties, the compiler generates the same code for both access of the property or the field. Therefore, for normal properties there is no advantage in referencing the private field.

Example:

```

1  type
2  TMyClass = class
3  private
4      FItems : array[0..10] of Integer;
5      FPlain : Integer;
6  protected
7      function GetItem(Index : Integer) : Integer;
8      procedure SetItem(Index, Value : Integer);
9  public
10     procedure MyProc;
11     property Items[Index : Integer] : Integer read GetItem write SetItem;
12     property Plain : Integer read FPlain write FPlain;
13 end;
14 ..
15 function TMyClass.GetItem(Index : Integer) : Integer;
16 begin
17     Result := FItems[Index];
18 end;
19
20 procedure TMyClass.SetItem(Index, Value : Integer);
21 begin
22     FItems[Index] := Value;
23 end;
24
25 procedure TMyClass.MyProc;
26 begin
27     if Items[0] < 5 then ///! ineffective, calls GetItem
28         Items[0] := 5;    ///! ineffective, calls SetItem
29
30     if FItems[0] < 5 then ///! this is faster
31         FItems[0] := 5;
32
33     Plain := 5; // these two lines generate the same machine code
34     FPlain := 5;
35 end;

```

Virtual methods (procedures/functions) that are not overridden (OPTI5)

This is a list of all methods that are declared as virtual, but that never are overridden. Since virtual methods have slightly worse performance than static methods, it is better to change these methods to static ones instead.

This section is also generated for multi-projects.

Recommendation:

Examine if these methods should really be overridden. If they belong to a base class, you should probably keep them virtual, so descendant classes can create their own implementations.

Local subprograms with references to outer local variables (OPTI6)

This section shows nested local procedures, with references to outer local variables. Those local variables require some special stack manipulation so that the variables of the

outer routine can be seen by the inner routine. This results in a good bit of overhead.

Subprograms with local subprograms (OPTI7)

This section lists subprograms that themselves have local subprograms. Especially when these subprograms share local variables, it can have a negative effect on performance.

Parameter is "var", can be changed to "out" (OPTI8)

This section lists parameters that are marked with the "var" directive, but that can be changed to "out".

Even if it may not improve performance, it improves the readability of the code and makes its intentions clearer.

Inlined subprograms not inlined because not yet implemented (OPTI9)

This section lists calls to inlined subprograms, where the subprogram will not be inlined. The reason is that the subprogram has not been implemented yet. It is implemented further down in the same module. There are a number of other conditions that also must be fulfilled for the subprogram to be inlined.

```
1  ..
2
3  type
4    TMyClass = class
5      private
6        ..
7        procedure Process;
8        function GetValue : Integer; inline;
9        procedure MoreProcessing;
10       ..
11     end;
12
13   ..
14
15   procedure TMyClass.Process;
16   var
17     Value : Integer;
18   begin
19     ..
20     Value := GetValue; // this call cannot be inlined!
21     ..
22   end;
23
24   function TMyClass.GetValue : Integer;
25   begin
26     ..
27   end;
28
29   procedure TMyClass.MoreProcessing;
30   var
31     Value : Integer;
32   begin
33     ..
34     Value := GetValue; // this call can be inlined!
35     ..
36   end;
37
38   ..
```

To make the subprogram inlined for this call, make sure that it is implemented higher up in the same module.

Managed local variable can be declared inline (OPTI10)

This section lists local variables that instead of being declared in the main var-section, can be declared inline.

Doing so optimizes the initialization-finalization code that is executed, and now only needs to be done when certain conditions are fulfilled.

Managed variables are strings, interfaces, dynamic arrays and records that contain managed fields.

Inline variable declarations were introduced in Delphi 10.3, so this report section is not relevant for older targets.

Managed local variable is inlined in loop (OPTI11)

This section lists local variables that instead of being declared in the main var-section, are declared inline.

They are declared inside a loop, which decreases performance. The reason is that initialization-finalization of the variable has to be done for each loop iteration.

Managed variables are strings, interfaces, dynamic arrays and records that contain managed fields.

Inline variable declarations were introduced in Delphi 10.3, so this report section is not relevant for older targets.

See also:

 [General Reports](#)

14.1.6 Code Reduction Report

Targets: All

This report pinpoints unnecessary code that could be deleted, resulting in a smaller amount of code to maintain and search for errors.

Identifiers never used (REDU1)

This is a list of all identifiers that are declared but never used. The Delphi compiler (from Delphi 2) also reports this if warnings (\$W+) have been turned on during compilation. Most often, you can remove these identifiers. If you remove any identifier, make sure your code still compiles and works properly. A wise habit is to first comment out these declarations, and remove them entirely when you have validated that the code still compiles and works as intended. Also, note that if a subprogram is not used, does not necessarily indicate that it is not needed at all. If it is part of a general unit, the subprogram could very well be used in other applications.

Identifiers (parameters, local variables etc) related to subprograms that are not used, are not reported.

Constructors/destructors are not examined by this section. Also parameters to event handlers, or methods that are referenced in form files, are not reported as unused. The reason is to avoid unnecessary warnings.

Also unused methods of a class that are implemented through interfaces are not reported. In this case, the class has no choice but to implement these methods.

Example:

```
1 procedure TMyForm.mnuOpen(Sender : TObject);  
2 begin  
3   OpenFile;  
4 end;
```

In this case, the parameter Sender is not reported as unused, since mnuOpen is an event handler.

This section is also generated for multi-projects.

Local identifiers only used at a lower scope (REDU2)

This is a list of all local identifiers that are only used at a lower scope, in nested subprograms. You can declare these identifiers in the local procedures/functions where they are actually used.

Example:

```
1  procedure Outer;  
2  var  
3    I : Integer;  
4  
5    procedure Inner;  
6    begin  
7      I := 55; // !! I is declared in Outer  
8      ..  
9    end;  
10  
11  begin  
12    ..  
13  end;
```

Local identifiers only used at a lower scope, but in more than one subprogram (REDU3)

This is a list of all local identifiers that are only used at a lower scope, in nested subprograms. You can probably declare these identifiers in the local procedures/functions where they are actually used, unless they should be shared by the nested subprograms.

Example:

```
1  procedure Outer;  
2  var  
3    I : Integer;  
4  
5    procedure Inner1;  
6    begin  
7      ..  
8      I := 55; // !! I is declared in Outer  
9      ..  
10   end;  
11  
12   procedure Inner2;  
13   begin  
14     ..  
15     I := 5;  
16     ..  
17   end;  
18  
19   begin  
20     ..  
21   end;
```

Local identifiers that are set and referenced once (REDU4)

This is a list of all local identifiers that are set and referenced just once. It may be more efficient to skip these intermediate identifiers.

Restrictions:

Identifiers that are first set as a *var* parameter in a call to a subprogram, and afterwards referenced, are not reported. Also, when the identifier is referenced in a loop, it is not reported.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4
5  begin
6    I := Func(5);
7    ..
8
9    if I = 5 then // !! if Func(5) = 5 then
10   begin
11     ..
12   end;
13 end;
```

Local identifiers that possibly are set and referenced once (REDU5)

This is a list of all local identifiers that possibly are set and referenced just once. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. It may be more efficient to skip this intermediate identifier.

Restrictions:

Identifiers that are first set as a *var* parameter in a call to a subprogram, and afterwards referenced, are not reported. Also, when the identifier is referenced in a loop, it is not reported.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4
5  begin
6    I := Func(5);
7    ..
8    UnknownProc(I); // !! UnknownProc(Func(5))
9    ..
10 end;
```

Local identifiers that are set more than once without referencing in-between (REDU6)

This is a list of all local identifiers that are set (assigned) more than once without

referencing in-between. You can probably remove all but the last assignment. It may of course also indicate a coding error.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4
5  begin
6    I := 5;
7    ..
8    I := 10; // !! I is set again
9    ..
10   if I = 10 then
11   begin
12     ..
13   end;
14 end;
```

Local identifiers that possibly are set more than once without referencing in-between (REDU7)

This is a list of all local identifiers that are set (assigned) more than once without referencing in-between. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. You can probably delete all but the last assignment.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4
5  begin
6    I := 5;
7    ..
8    UnknownProc(I);
9    ..
10   I := 10; // !! I may already be set
11   ..
12   if I = 10 then
13   begin
14     ..
15   end;
16 end;
```

Class fields that are zero-initialized in constructor (REDU8)

This is a list of all class fields that are zero-initialized in constructor. Since class fields are automatically zero-initialized when the object is created, there is usually no need to include this code.

Example:

```

1  type
2    TMyClass = class(TAncestorClass)
3  private
4    FInt : Integer;
5    FStr : string;
6    FPtr : Pointer;
7  public
8    constructor Create; override;
9    ..
10 end;
11
12 constructor TMyClass.Create;
13 begin
14   inherited Create;
15   FInt := 0; // !! unnecessary
16   FStr := '';
17   FPtr := nil;
18 end;

```

Class fields that possibly are zero-initialized in constructor (REDU9)

This is a list of all class fields that possibly are zero-initialized in constructor. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Since class fields are automatically zero-initialized when the object is created, there is usually no need to include this code.

Example:

```

1  type
2    TMyClass = class(TAncestorClass)
3  private
4    FInt : Integer;
5    FStr : string;
6    FPtr : Pointer;
7  public
8    constructor Create; override;
9    ..
10 end;
11
12 constructor TMyClass.Create;
13 begin
14   inherited Create;
15   UnknownProc1(FInt); // !! possibly unnecessary
16   UnknownProc2(FStr);
17   UnknownProc3(FPtr);
18 end;

```

Local long strings that are initialized to empty string (REDU10)

(Not relevant for BP7 and D1)

This is a list of all local long strings that are initialized to empty strings. An unnecessary action, since long strings are automatically initialized as empty strings upon creation.

Example:


```
1 procedure MyProc;  
2 var  
3   S : string;  
4  
5 begin  
6   S := ''; // !! unnecessary  
7   ..  
8 end;
```

Local long strings that possibly are initialized to empty strings (REDU11)

(Not relevant for BP7 and D1)

This is a list of all local long strings that are initialized to empty strings. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. An unnecessary action, since long strings are automatically initialized as empty strings upon creation.

Example:

```
1 procedure MyProc;  
2 var  
3   S : string;  
4  
5 begin  
6   UnknownProc(S); // !! possibly unnecessary  
7   ..  
8 end;
```

Functions called only as procedures (result ignored) (REDU12)

These functions may possibly better be implemented as procedures, because the result is never used.

This section is also generated for multi-projects.

Functions/procedures (methods excluded) only called once (REDU13)

The code in these functions/procedures could possibly be included inline instead, avoiding an unnecessary call.

This section is also generated for multi-projects.

Methods only called once from other method of the same class (REDU14)

These methods are never called from the outside. The code in these methods could possibly be included inline instead, avoiding an unnecessary call.

This section is also generated for multi-projects.

Unneeded boolean comparisons (REDU15)

This list contains locations with statements like

```
if bReady = true then
```

This could be shorter and better written as

```
if bReady then
```

Boolean assignment can be shortened (REDU16)

This list contains locations with statements like

```
1 | if X then
2 |   Bool := True
3 | else
4 |   Bool := False;
```

This could be shorter and better written as

```
1 | Bool := X;
```

Fields only used in single method (REDU17)

This list contains class or record fields that are only used in a single method. They could probably better be declared as local variables.

Consider using interface type (REDU18)

This list contains objects which can be declared and implemented as an interface type, instead of as the class type implementing the interface. The advantage is that interface reference counting can be used so you will not have to explicitly free the object.

Example:

```
1  type
2    IIntf = interface
3      ..
4    end;
5
6    TIntf = class(TInterfacedObject, IIntf)
7      ..
8    end;
9
10 var
11   Obj : TIntf; // consider using IIntf!
12   ..
13
14
```

The list will not include objects that are not created. These objects are probably just assigned to another object.
Another condition that must be met is that the object is of a class that implements exactly one interface.

Redundant parentheses (REDU19)

This section lists locations in your code where superfluous parentheses can be removed, simplifying the code.

```
1  procedure Proc;
2  begin
3    if (3+4) = 7 then ..;
4
5    if (((True))) then ..;
6
7    if (True) then ..;
8
9    if (3+4+(5)+(6)) = 11 then ..;
10 end;
11
```

Common subexpression, consider elimination (REDU20)

This section lists locations in your code with repeated common subexpressions. Those may be candidates to put into temporary variables to simplify and optimize the code.

Example:

```
1  procedure Proc;
2  var
3    A, B, C, D, E : Integer;
4  begin
5    ..
6    E := A+B+C;
7    ..
8    D := A+B+C; // expression is repeated
9    ..
10 end;
```

If any of the variables involved in the repeated expressions would have been modified,

between the locations, there should not be any warning.

Default parameter values that can be omitted (REDU21)

This list contains calls to functions or procedures that use default parameters, and where the parameter can be omitted at the call site. The reason is then that the value of the parameter passed is the same as the default parameter value.

Example:

```
1  procedure ProcWithDefaultParam(P : Pointer = nil);
2  begin
3      ..
4  end;
5
6  procedure Proc;
7  var
8      P : Pointer;
9  begin
10     P := nil;
11     ProcWithDefaultParam(P); // the parameter P is not needed here
12 end;
```

Inconsistent conditions (REDU22)

This section reports locations with inconsistent conditions. These are places where a condition check is repeated, even if the outcome will be the same as in the previous location.

Example:

```
1  procedure Proc;
2  var
3      I : Integer;
4  begin
5      if I = 1 then
6      begin
7          ..
8      end
9      else
10         if I = 1 then // gives inconsistent condition warning
11         begin
12             ..
13         end;
14     end;
```

Typecasts that possibly can be omitted (REDU23)

This section reports locations with typecasts that possibly can be omitted. It is locations where the typecast casts the variable to the same type that it already has.

Example:

```
1  var
2    I, K : Integer;
3  begin
4    ..
5    K := Integer(I);
6    ..
7  end;
```

Local identifiers never used (REDU24)

This is a list of all local identifiers that are declared but never used. It is actually a subset of the REDU1 report section, which reports all identifiers, not only local.

See also:

 [General Reports](#)

14.1.7 Convention Compliance Report

Targets: All

This report contains several lists with identifiers that do not comply with conventions for naming of identifiers.

The choice of names for identifiers has a considerable influence on the ease of understanding and maintenance costs of your source code. Developers familiar with the coding standards can understand the code more easily if it follows general conventions.

Ordinary types that do not start with "T" (CONV1)

This is a list of all ordinary types that do not start with the letter "T". Exception, pointer and interface types are not included. As a convention, user-defined type names start with the letter "T". A class that is a CoClass is an exception and is not reported. PAL assumes a CoClass when the name of the class starts with the letters "Co". Furthermore, the class must have a class function with the name "Create".

Also custom attributes inheriting from TCustomAttribute are not reported.

Exception types that do not start with "E" (CONV2)

This is a list of all exception types that do not start with the letter "E". As a convention, user-defined exception type names start with the letter "E".

Pointer types that do not start with "P" (CONV3)

This is a list of all pointer types that do not start with the letter "P". As a convention, user-defined pointer type names start with the letter "P".

Interface types that do not start with "I" (CONV4)

This is a list of all interface types that do not start with the letter "I". As a convention, user-defined interface type names start with the letter "I".

Class fields that are not declared in the private section (CONV5)

This is a list of all class fields that are not declared in the private section of a class.

Class fields that are exposed by properties (read/write) but do not start with "F" (CONV6)

This is a list of all class fields that are exposed by properties but do not start with the letter "F". As a convention, private class field names start with the letter "F".

This section is similar to CONV23, but that section reports all fields, not only those exposed as properties.

Properties to method pointers that do not start with "On/Before/After" (CONV7)

This is a list of all properties to method pointers that do not start with "On/Before/After"

Functions that are exposed by properties (read) but do not start with "Get" (CONV8)

This is a list of all functions that are exposed by properties read methods, but do not start with "Get". As a convention, these functions (methods) should start with the letters "Get" (e.g. GetIndex, GetBitmap).

Procedures that are exposed by properties (write) but do not start with "Set" (CONV9)

This is a list of all functions that are exposed by properties write methods, but do not start with "Set". As a convention, these procedures (methods) should start with the letters "Set" (e.g. SetIndex, SetBitmap).

Classes that have visible constructors with bad names (CONV10)

This is a list of all classes that have constructors with bad names. As a convention, constructor names start with the letters "Create". For old-style objects (BP7), the constructor names start with the letters "Init".

Classes that have visible destructors with bad names (CONV11)

This is a list of all classes that have destructors with bad names. As a convention, destructor names start with the letters "Destroy". For old-style objects (BP7), the destructor names start with the letters "Done".

Identifiers that have unsuitable names (CONV12)

This is a list of all identifiers with names that are the same as directives, e.g. "pascal", "dynamic", "index" and others. Even if the compiler allows this, it may lead to misunderstandings. For Delphi 1 and higher, the list also includes identifiers with identical names as identifiers from the System unit (like "Copy", "AllocMem").

Multiple with-variables (CONV13)

This is a list of all locations in the source where multiple with-variables ("with A, B do") are used. It is often considered a bad coding habit to use multiple with-variables, since they make the source more difficult to understand.

Property access methods that are not private/protected (CONV14)

This is a list of all property access methods that are not declared as private/protected. Property access methods are used with properties, e. g:

```
property MyProp : integer read GetMyProp write SetMyProp
```

where GetMyProp and SetMyProp are property access methods.

Those methods should not be directly callable from the outside, because all access should go through the associated property.

Hard to read identifier names (CONV15)

This is a list of all identifiers with hard to read names. A name is considered hard to read if it contains both the letter "O" and the number "0", or both the letter "l" and the number "1".

Label usage (CONV16)

This list contains all labels that are used in the source code. Labels define jump-locations for a goto statement. Usage of labels and goto-statements is considered as a bad thing, which is most often not needed in modern object-oriented programming. There are situations though, when a label may be the right choice.

Bad class visibility order (CONV17)

This list contains all class types with bad class visibility order in the declaration. Bad order is defined as when **private** sections appear after **public/protected** sections or when **protected** sections appear after **public** sections. The code is probably easier to understand and maintain if a good visibility order is used.

Classes that PAL thinks are derived from TForm are not reported. This is because these type of classes depend on a special order, starting with published identifiers.

Identifiers with numerals (CONV18)

This list contains all identifiers with names that contain numerals.

Local identifiers that "shadow" outer scope identifiers (CONV19)

This list contains local identifiers that have the same name as outer scope identifiers in the same unit.

Example:

```
1  var
2    I : Integer;
3
4  procedure Proc;
5    var
6      I : Integer; // I shadows outer global I!
7  begin
8    ..
9  end;
```

Although this is allowed, it may lead to confusion and misunderstandings when maintaining the code.

Local identifiers that "shadow" class members (CONV20)

This list contains local identifiers in methods that have the same name as a class member.

Example:


```
1 type
2   TMyClass = class
3   private
4     Field : Integer;
5     procedure M;
6   end;
7   ..
8   procedure TMyClass.M;
9   var
10    Field : Integer;
11  begin
12    ..
13  end;
```

Although this is allowed, it may lead to confusion and misunderstandings when maintaining the code.

Class/member name collision (CONV21)

This section reports situations where class and member names collide.

Class fields that are not declared in the private/protected sections (CONV22)

This is a list of all class fields that are not declared in the private/protected sections of a class.

Fields should normally not be made "public". They should instead be accessed through properties

Class fields that do not start with "F" (CONV23)

This is a list of all class fields that not start with the letter "F". As a convention, private class field names start with the letter "F".

Component fields in the DFM-file are not reported.

This section is similar to CONV6, but that section only reports field exposed as properties.

Value parameters that do not start with selected prefix (CONV24)

This is a list of all value parameters that do not start with the selected prefix. Set the prefix in the [Reports tab page](#) for the project. Double-click on the Convention Report, select report section CONV24 and press the Prefix button to enter the prefix.

Const parameters that do not start with selected prefix (CONV25)

This is a list of all const parameters that do not start with the selected prefix. Set the prefix in the [Reports tab page](#) for the project. Double-click on the Convention Report, select report section CONV25 and press the Prefix button to enter the prefix.

Out parameters that do not start with selected prefix (CONV26)

This is a list of all out parameters that do not start with the selected prefix. Set the prefix in the [Reports tab page](#) for the project. Double-click on the Convention Report, select report section CONV26 and press the Prefix button to enter the prefix.

Var parameters that do not start with selected prefix (CONV27)

This is a list of all var parameters that do not start with the selected prefix. Set the prefix in the [Reports tab page](#) for the project. Double-click on the Convention Report, select report section CONV27 and press the Prefix button to enter the prefix.

Old-style function result (CONV28)

This is a list of functions where instead of "Result", the function name is used as the result variable.

With statements (CONV29)

This is a list of locations where "with" is used.

Private can be changed to strict private (CONV30)

This is a list of class members that are private but can be changed to strict private.

Protected can be changed to strict protected (CONV31)

This is a list of class members that are protected but can be changed to strict protected.

See also:

[🌐 General Reports](#)

14.1.8 Inconsistent Case Report

Targets: All

Inconsistent case for same identifier (INCA1)

This is a list of the locations where the identifier is written with a different case compared with the declaration.

Example:

```
1 | procedure MyProc;  
2 | var  
3 |   MyStr : string;  
4 |  
5 | begin  
6 |   Mystr := 'Hi'; // !! this will trigger a listing  
7 |   ..  
8 |   MyStr := 'Again'; // this will not
```

Inconsistent case for different identifiers with same name (INCA2)

This is a list of all identifiers with the same name, but that are declared with different case.

Example:

```
1 | var  
2 |   MyInt : Integer;  
3 |  
4 | procedure Proc;  
5 |   var  
6 |     Myint : string; // same name as global MyInt above, but different case  
7 |   begin  
8 |     ..  
9 |   end;
```

Mismatch unit name/file name (INCA3)

This is a list of all units with mismatch between the unit name and file name.

See also:

[General Reports](#)

14.1.9 Prefix Report

Targets: All except BP7

This report contains a list with variables that have a different prefix than the defined ones. For instance:

```
var
  Label1 : TLabel;
```

Assume that one defined prefix is "lbl" for variables of type TLabel. The declared variable Label1 will then be listed in the report, since it does not comply with this rule. Set rules for prefixes in the [Options](#) dialog.

```
*****
*                               Prefix Report for                               *
*                               C: \PROJEKT\WBT\WBTSTORE. DPR                       *
*****
```

These variables do not have the right prefix (17):

CmdClose : TBitBtn	ClassField	WbtList\TListForm (17)
CmdDelete : TBitBtn	ClassField	WbtList\TListForm (16)
CmdEdit : TBitBtn	ClassField	WbtList\TListForm (15)
CmdHelp : TBitBtn	ClassField	GRtfMod\TModalRTFViewerForm (23)
CmdNew : TBitBtn	ClassField	WbtList\TListForm (14)
CmdPrint : TBitBtn	ClassField	GRtfMod\TModalRTFViewerForm (22)
CmdSaveAs : TBitBtn	ClassField	GRtfMod\TModalRTFViewerForm (21)
FButton : TOvcEdButton	ClassField	OvcBCalc\TOvcBorderEdPopup (61)
FButton : TOvcEdButton	ClassField	OvcEdPop\TOvcEdPopup (60)
FEdit : TOvcCustomEdit	ClassField	OvcBCalc\TOvcBorderEdPopup (60)
FEdit : TOvcCustomEdit	ClassField	OvcBordr\TOvcBorderParent (130)
FOvcEdit : TOvcNumberEditEx	ClassField	OvcBCalc\TOvcBorderedNumberEdit (139)
FOvcEdit : TOvcDateEditEx	ClassField	OvcBCldr\TOvcBorderedDateEdit (52)
FOvcEdit : TOvcEditEx	ClassField	OvcBEdit\TOvcBorderedEdit (49)
GroupBox1 : TGroupBox	ClassField	WbtQrCon\TQRFilterDialog (15)
GroupBox2 : TGroupBox	ClassField	WbtQrCon\TQRFilterDialog (20)
GroupBox3 : TGroupBox	ClassField	WbtQrCon\TQRFilterDialog (24)

See also:

[General Reports](#)

14.1.10 NextGen Readiness Report

Targets: All

This report measures how well prepared your code is for the NextGen compiler. It gives you hints to help you rework your code so that it is compatible with the NextGen compiler.

N.B. From Delphi 10.4 this report is not so relevant. This is because the NextGen compilers have been retired. All compilers now share the same old-fashioned Delphi memory model that we all know.

There are currently three sections in this report:

Unsupported types (NEXT1)

This is a list of identifiers with types that are not supported by the NextGen compiler.

Except-block without call to non-inlined subprograms (NEXT2)

This is a list of except-blocks that do not call non-inlined subprograms. This is a requirement for exceptions to work well with the NextGen compiler.

Assembler code in these subprograms (NEXT3)

This is a list of subprograms containing assembler code. Assembler code is not supported by the NextGen compiler.

See also:

 [General Reports](#)

14.2 Metrics Reports

The metrics reports are:

-  [Totals Report](#)
-  [Module Totals Report](#)
-  [Complexity Report](#)
-  [Object-oriented metrics Report](#)

See also:

[Reports](#)

14.2.1 Totals Report

Targets: All

This report writes a table that summarizes the number of files, procedures, functions, types etc, in the analyzed source code. It also shows how many of each category that are global, interfaced and unused. Here is a small excerpt from such a table.

```

*                               Totals Report for                               *
*                               C: \PROJEKT\PSEARCH\PSEARCH.DPR                     *
*****
Metrics:
-----
Element                Total      Local      Global Interfaced      Unused

Files                   3         -         -         -         -
Modules                 23         -         -         -         -
Lines                  715         -         -         -         -

Constants               1         1         0         0         0
  Typed                 0         0         0         0         0
  Resourcestring        0         0         0         0         0

Types                   1         0         1         1         0
  Simple/unknown        0         0         0         0         0
  Array                 0         0         0         0         0
  Class                  1         -         1         1         0
  Class ref              0         -         0         0         0
  Enumerated            0         0         0         0         0

```

Restrictions:

Only identifiers that are declared in code that is found and parsed are included in the totals. It will include all identifiers, regardless if the folder where the code is found is excluded for reporting.

To see the total number of lines for different source code directories, see the [Complexity Report](#).

You can set the option for “Details” in the [Options](#) dialog, to let PAL generate detailed information.

See also:

 [Metrics Reports](#)

14.2.2 Module Totals Report

Targets: All

This report writes the same type of tables as the [Totals Report](#), but for each reported unit in the analyzed project.

Restrictions:

Only identifiers that are declared in code that is found and parsed are included in the totals.

You can set the option for “Details” in the [Options](#) dialog, to let PAL generate detailed information.

See also:

 [Metrics Reports](#)

14.2.3 Complexity Report

Targets: All

Complexity per module/subprogram

This section lists a number of important metrics:

Total Lines
Lines of Code (LOC)
Comment Lines
Comments/Total Lines
Comments/Lines of Code (Comments/LOC)
Decision Points (DP)
Decision Points/Lines of Code (DP/LOC)
Characters/Lines of Code (Chrs/LOC)
Comment Characters/Characters

These metrics will help you evaluate source code in terms of understandability, complexity, and reusability. Complex code is harder to maintain and is more error prone. You can also locate complex subprograms that would be better split up.

Minimizing complexity is a major key to writing high quality code. Measure the complexity regularly during development so complex code can be identified early and rewritten to improve understanding and to reduce testing and future maintenance effort.

Results are also calculated for code from different source folders. Each folder generates one set of data. In this way, you can compare for instance code from different third-party toolkits that are located in different folders.

For these metrics, only source code files are counted, so for instance DFM files are not included.

Total Lines are all lines, including blank lines and comment lines.

Lines of Code are lines, excluding blank lines and full comment lines that contain actual code. For a subprogram (procedure/function) this is the code part including lines with *begin/end*.

Comment Lines are comment lines or lines that are partly commented.

Decision points is equivalent to McCabe's metric, or cyclometric complexity. These are all well-known metrics in the software industry that are used to estimate code complexity. Decision points are positive integers where increasing values mean a higher degree of complexity. It is calculated by PAL in this way:

1. Start with a value of 1 for the normal flow through a subprogram.
2. Add 1 for each of these keywords: IF, WHILE, REPEAT, FOR, AND, OR, XOR, GOTO. Note that PAL always counts AND, OR, XOR, even if they are used as arithmetic operators in the actual code.
3. Add 1 for each case label in a CASE statement. When the CASE does not have an

ELSE, add 1 more.

A normal interpretation of decision points is that you should try to break routines with a value of 10 or more into smaller pieces.

Decision Points/Lines of Code is just an average of the DP over the lines of code.

Characters/Lines of Code is an average of characters over lines of code.

Comment Characters/Characters is an average of comment characters over total characters. All comment characters including leading and trailing white space (beginning and end lines) are accounted for, plus the comment tokens themselves.

Example:

```
{ this is a comment }
```

This comment has 21 characters

```
// another one
```

This comment has 14 characters

```
{this
comment is on
multiple
lines}
```

This comment has a total of 32 characters

This is an excerpt from a complexity report:

```
*****
*                               Complexity Report for                               *
*                               C:\PROJEKT\PSEARCH\PSEARCH.DPR                       *
*****
```

Abbreviations: C=Constructor D=Destructor F=Function
 FM=Function (method) MB=Program main block
 P=Procedure PM=Procedure (method)
 UF=Unit finalization UI=Unit initialization

Module/Subprogram	Total Lines	LOC	Cmt Lines	Cmts/ Total Lines	DP	DP/ LOC	Chrs/ LOC
Overall	715	436	132	0,18	46	0,11	1
psearch	14	10	1	0,07	1	0,10	20
psearch (MB)	5	5	0	0,00	1	0,20	19
psform	701	426	131	0,19	45	0,11	26
TPSMainForm.chkAllFilesClick (PM)	5	4	0	0,00	2	0,50	12
TPSMainForm.CmdQuitClick (PM)	4	3	0	0,00	1	0,33	5
TPSMainForm.CmdSaveClick (PM)	13	11	0	0,00	2	0,18	14
TPSMainForm.CmdSearchClick (PM)	55	42	0	0,00	9	0,21	19
TPSMainForm.CmdSelStartDirClick (PM)	8	6	0	0,00	2	0,33	28
TPSMainForm.CmdSpecial1Click (PM)	104	53	29	0,28	3	0,06	30
TPSMainForm.CmdSpecial2Click (PM)	130	96	2	0,02	13	0,14	21

Even if it may seem simple, lines of code is quite a good measure of how complex a program is. Decision points (DP) also provide a good indication.

Be forewarned that the Complexity Report may take relatively long time to generate, compared with the other reports.

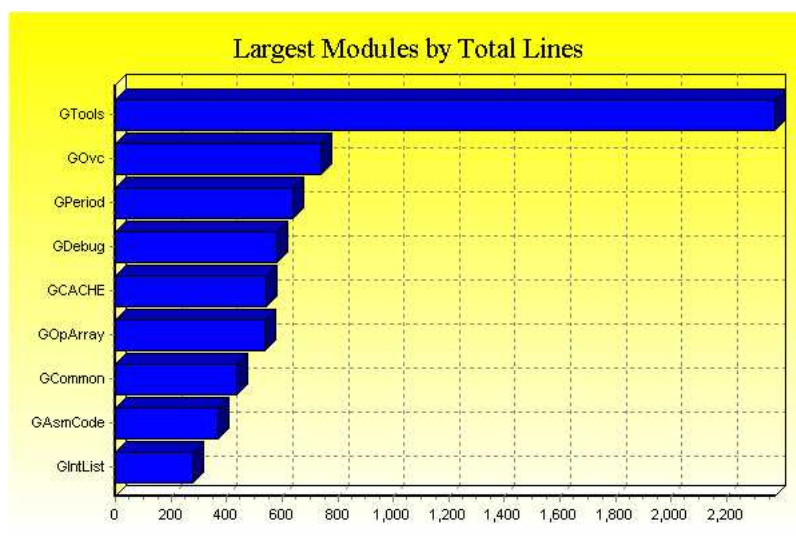
It is possible to sort this section according to any of the keys, like Total Lines, LOC, DP etc.

Long identifier names

PAL also creates a list of all identifiers with names that are longer or equal to 15 (default) characters in length. You can select the identifier length, for which PAL will start reporting. E.g. if you select 20, PAL will report all identifiers having 20 or more identifiers.

The remaining sections in the Complexity Report show different lists. These list rate modules, subprograms and classes by varying criteria, like most lines of code (LOC). Optionally, if HTML reports are created, a chart can be generated for each section. The charts are saved as image files (JPG) in the report folder.

This is a sample of a chart illustrating *Largest Modules by Total Lines*:



These sections show statistics:

Largest Modules by Total Lines

Largest Modules by LOC

Most Complex Modules by DP

Most Complex Modules by DP/LOC

Most Complex Modules by Characters per LOC

Most Commented Modules by Comment Lines

Most Commented Modules by Comment Lines per Total Lines

Most Commented Modules by Comment Lines per LOC

Most Commented Modules by Comment Characters per Total Characters

Largest Subprograms by Total Lines

Largest Subprograms by LOC

Most Complex Subprograms by DP

Most Complex Subprograms by DP/LOC

Most Complex Subprograms by Characters per LOC

Most Commented Subprograms by Comment Lines

Most Commented Subprograms by Comment Lines per Total Lines

Most Commented Subprograms by Comment Lines per LOC

Most Commented Subprograms by Comment Characters per Total Characters

Most Methods per Class

Most Fields per Class

Most Properties per Class

Most Parameters per Subprogram

See also:

 [Metrics Reports](#)

14.2.4 Object-oriented Metrics Report

Targets: All

This report has six different sections, each calculating and displaying an important metrics:

Weighted methods per class (WMC)

Depth of inheritance tree (DIT)

Number of children (NOC)

Coupling between object classes (CBO)

Response for a class (RFC)

Lack of cohesion in methods (LCOM)

For more detailed reading about these metrics, their theory and implementation, see <http://www.pitt.edu/~ckemerer/clnieee.pdf>.

Weighted methods per class (WMC) (OOME1)

The number of methods in a class predicts the time and effort needed to develop and maintain the class. In the PAL implementation, the mean decision-point (DP) is calculated for the class. This is presented together with the DP for each method in the class.

Weighted Methods per Class (WMC):

```
-----
BAbout.TAboutBox (WMC=2,0)..... BAbout (14)
  CmdLicenseClick (PM) (DP=1)..... BAbout\TAboutBox (34)
  Create (C) (DP=1)..... BAbout\TAboutBox (48)
  FillControls (PM) (DP=4)..... BAbout\TAboutBox (45)

BCat.TPegCategory (WMC=0,7)..... BCat (23)
  Create (C) (DP=0)..... BCat\TPegCategory (80)
  ReadStream (PM) (DP=1)..... BCat\TPegCategory (82)
```

```
WriteStream (PM) (DP=1)..... BCat\TPegCategory (83)
```

Depth of inheritance tree (DIT) (OOME2)

The deeper a class hierarchy is, the more complex it becomes. Deep trees indicate greater complexity, but also promote code reuse because of inheritance.

DIT is calculated as the number of classes traversed from the actual class to the root class. Because every class inherits from TObject, this number is at least 1. For a class that inherits from an unknown class (a class for which code has not been found) has a number of 2. This is because the unknown class has at least DIT=1.

```
Depth of Inheritance Tree (DIT):
```

```
-----
BAbout.TAboutBox (DIT=2)..... BAbout (14)
BCat.TPegCategory (DIT=1)..... BCat (23)
BCat.TPegCategoryHandler (DIT=1)..... BCat (25)
BCat.TPegCategoryList (DIT=1)..... BCat (22)
BFile.EIncludeBroken (DIT=2)..... BFile (12)
BLicFrm.TSettingsDialog (DIT=2)..... BLicFrm (14)
BRes.TPegResource (DIT=1)..... BRes (23)
BRes.TPegResourceHandler (DIT=1)..... BRes (25)
BRes.TPegResourceList (DIT=1)..... BRes (22)
```

Number of children (NOC) (OOME3)

This number is calculated as the immediate number of children for a given class. A high number indicates high code reuse. A large number of children could also mean a bad abstraction of the parent class.

```
Number of Children (NOC):
```

```
-----
BAbout.TAboutBox (NOC=0)..... BAbout (14)
BCat.TPegCategory (NOC=0)..... BCat (23)
BCat.TPegCategoryHandler (NOC=0)..... BCat (25)
GFiles.TAbstractFile (NOC=2)..... GFiles (43)
```

Coupling between object classes (CBO) (OOME4)

Two classes are coupled when methods in one class use methods, properties or fields in another class. This is counted both ways, it does not matter which class that calls the other.

A high number for CBO is not desirable. It means that the class is less independent or less loosely coupled. A low number for CBO on the other hand, the easier it is to reuse it in another project.

```
Coupling between Object Classes (CBO):
```

```
-----
BAbout.TAboutBox (CBO=2)..... BAbout (14)
  OvcRLbl.TOvcRotatedLabel..... OvcRLbl (126)
  OvcURL.TOvcURL..... OvcURL (54)
BCat.TPegCategory (CBO=5)..... BCat (23)
```

```

TdPrnWiz. TPrintWizardDialog..... TdPrnWiz ( 19)
TdMain. TTodoMainForm..... TdMain ( 19)
TdEdCat. TEditCategoryDialog..... TdEdCat ( 14)
BCat. TPegCategoryHandler..... BCat ( 25)
BCat. TPegCategoryList..... BCat ( 22)

```

Response for a class (RFC) (OOME5)

The response number for a class is the number of methods or procedures that can be potentially executed if a message (for example a function call) is received by the class.

It is defined as

$$\text{RFC} = \text{M} + \text{R}$$

where

M = number of methods in the class

R = number of remote methods directly called by methods of the class

RFC counts only the first level of calls outside the class. It is calculated by following all calls from methods in the class, if these calls lead to methods in other classes or to procedures or functions. Every given method/subprogram is only counted once.

A large number for RFC indicates that the class is complex and hard to understand. It means that more time must be given for testing and debugging.

Response for a Class (RFC):

```

-----
BAbout. TAboutBox ( RFC=3)..... BAbout ( 14)
BCat. TPegCategory ( RFC=3)..... BCat ( 23)
BCat. TPegCategoryHandler ( RFC=14)..... BCat ( 25)
BCat. TPegCategoryList ( RFC=6)..... BCat ( 22)
BFile. EIncludeBroken ( RFC=0)..... BFile ( 12)
BLicFrm. TSettingsDialog ( RFC=4)..... BLicFrm ( 14)
BRes. TPegResource ( RFC=2)..... BRes ( 23)
BRes. TPegResourceHandler ( RFC=13)..... BRes ( 25)
BRes. TPegResourceList ( RFC=6)..... BRes ( 22)

```

Lack of cohesion of methods (LCOM) (OOME6)

A class that is cohesive performs one and only one function. Lack of cohesion means the class performs more than one function.

$$\text{LCOM} = \text{P} - \text{Q}, \text{ if } \text{P} > \text{Q}$$

else

$$\text{LCOM} = 0$$

The number is calculated by taking each possible pair of methods in the class. If they access the same field, increase Q by one. If they don't, increase P by one.

LCOM = 0 means a cohesive class.

If LCOM > 0, the class could be split into two or more classes. A high LCOM indicates that the class is error-prone.

Lack of Cohesion in Methods (LCOM):

```

BAbout.TAboutBox (LCOM=0)..... BAbout (14)
BCat.TPegCategory (LCOM=0)..... BCat (23)
GEnhLb.TEnhancedListBox (LCOM=13)..... GEnhLb (16)
GFiles.TBinaryFile (LCOM=26)..... GFiles (51)
GFiles.TTextFile (LCOM=0)..... GFiles (118)
GFrView.TFrPreviewDialog (LCOM=0)..... GFrView (16)
GGlyphLb.TGlyphListBox (LCOM=0)..... GGlyphLb (14)
GNag.TNagDialog (LCOM=0)..... GNag (15)
GSapi.TSapiVoice (LCOM=0)..... GSapi (13)
TdApp.TToDoAppHandler (LCOM=0)..... TdApp (16)
TdArch.TTdViewArchiveDialog (LCOM=1)..... TdArch (14)
TdCommon.TCommonModule (LCOM=0)..... TdCommon (16)
TdEdCat.TTdEditCategoryDialog (LCOM=4)..... TdEdCat (14)

```

See also:

 [Metrics Reports](#)

14.3 Reference Reports

The reference reports are:

-  [Modules Report](#)
-  [Identifiers Report](#)
-  [Duplicate Identifiers Report](#)
-  [Similarity Report](#)
-  [Literal Strings Report](#)
-  [Subprogram Index Report](#)
-  [Bindings Report](#)
-  [Third-party dependencies Report](#)
-  [Most Called Report](#)
-  [Call Tree Report](#)
-  [Reverse Call Tree Report](#)
-  [Call Index Report](#)
-  [Exception Report](#)
-  [Brief Cross-reference Report](#)
-  [Cross-reference Report](#)
-  [Used Outside Report](#)
-  [Uses Report](#)
-  [Conditional Symbols Report](#)
-  [Directives Report](#)
-  [To-Do Report](#)
-  [Module Call Tree Report](#)
-  [Help Report](#)
-  [Searched Strings Report](#)
-  [Map File Report](#)
-  [Clone Report](#)

See also:

[Reports](#)

14.3.1 Modules Report

Targets: All

The Modules Report consists of these sections:

Module information
Last modified file
Unique files
Missing files
Binary DFM/XFM files

Module information

This section lists all modules *that were found* by PAL, reporting the following data:

- Lines
- Size (bytes)
- Date
- Time
- Path
- Encoding
- Initialization (if the module has an initialization section)
- Finalization (if the module has a finalization section)

Last modified file

This tells which source file that was last modified. If there are more than one file that share the same modification time, PAL will just report the first one it encounters.

Unique files

This is a list of all unique files referenced. You may use this information to create a batch backup utility.

Missing files

This is a list of all missing files. Those are files that are referenced, but that PAL was unable to find.

Binary DFM/XFM files

This is a list of all DFM/XFM files that are saved in binary format. It is advisable to choose text format instead.

See also:

 [Reference Reports](#)

14.3.2 Identifiers Report

Targets: All

Identifiers (IDEN1)

This section writes a table listing all the identifiers in the program indicating their type and location. Use this as an index for your source code, and to locate a specific identifier.

Global Variables (IDEN2)

This section lists variables that are globally declared in unit interface parts.

Module Global variables (IDEN3)

This section lists variables that are globally declared in the unit implementation parts.

Inlined variables and constants (IDEN4)

This section lists variables and constants that are declared inline.

```
*****
*                               Identifiers Report for                               *
*                               C: \PROJEKT\PSEARCH\PSEARCH. DPR                       *
*****

AllowClickEvents : Boolean      ClassField      psform\TPSMainForm ( 61)
Analyzing : Boolean            ClassField      psform\TPSMainForm ( 62)
Attrs : integer                Var, Local      psform\TPSMainForm\CmdSpecial2Click
( 422)

Base : TStringList             Var, Local      psform\TPSMainForm\CmdSpecial1Click
( 315)
Bevell : TBevel                ClassField      psform\TPSMainForm ( 35)

CanClose : Boolean             VarParam        psform\TPSMainForm\FormCloseQuery
( 56)
chkAllFiles : TCheckBox         ClassField      psform\TPSMainForm ( 28)
chkAllFilesClick               Proc, Method    psform\TPSMainForm ( 53)
chkCaseSensitive : TCheckBox    ClassField      psform\TPSMainForm ( 43)
chkIncludeSubFolders : TCheckBox ClassField      psform\TPSMainForm ( 25)
chkListAllSearched : TCheckBox ClassField      psform\TPSMainForm ( 37)
chkOnlyDelphi : TCheckBox       ClassField      psform\TPSMainForm ( 33)
chkOnlyInc : TCheckBox          ClassField      psform\TPSMainForm ( 30)
chkOnlySql : TCheckBox          ClassField      psform\TPSMainForm ( 31)
```

chkOnlyTxt : TCheckBox	ClassField	psform\TPSMainForm (34)
chkOnlyVB : TCheckBox	ClassField	psform\TPSMainForm (32)
chkOnlyWeb : TCheckBox	ClassField	psform\TPSMainForm (29)
chkReadOnly : TCheckBox	ClassField	psform\TPSMainForm (45)

See also:

 [Reference Reports](#)

14.3.3 Duplicate Identifiers Report

Targets: All

The Duplicate Identifiers Report consists of two sections:

Duplicate Identifiers (DUP1)

This section produces a list of all identifiers that share the same name. Even if it is legal to duplicate identifier names, it may sometimes be confusing, and lead to bugs that are hard to find.

```

*****
*                                     Duplicate Identifiers Report for          *
*                                     C: \PROJEKT\RAMVERK\GCACHE. PAS             *
*****

Duplicate Identifiers ( 572):
-----

AFreeOnDelete : Boolean      ValParam      GCACHE\TPointerCache\Create ( 81)
AFreeOnDelete : Boolean      ValParam      GCACHE\TObjectCache\Create ( 65)

AMaxEntries : Integer        ValParam      GCACHE\TPointerCache\Create ( 80)
AMaxEntries : Integer        ValParam      GCACHE\TObjectCache\Create ( 64)
AMaxEntries : Integer        ValParam      GCACHE\TStringCache\Create ( 50)
AMaxEntries : Integer        ValParam      GCACHE\TAbstractCache\Create ( 26)

AParam : Long String         ConstParam    GCACHE\TStringCache\AddString ( 53)
AParam : Long String         ConstParam    GCACHE\TAbstractCache\AddToCache ( 29)

```

Duplicate Identifiers in overlapping scope (DUP12)

This section produces a list of all identifiers that share the same name. But it only lists those that are in overlapping scope. Those duplicate identifiers are most likely to lead to confusion.

See also:

 [Reference Reports](#)

14.3.4 Similarity Report

Targets: All

The Similarity Report has currently just one section:

Similarity by Soundex

This section produces a list of all identifiers that have the same Soundex value, and that are in the same unit (prior to version 8.1.5, identifiers that were in overlapping scope were included). Soundex is a phonetic algorithm for indexing names by sound, as pronounced in English.

See also:

 [Reference Reports](#)

14.3.5 Literal Strings/Numbers Report

Targets: All

This report creates a list of all literal strings/numbers, both those declared as constants and those immersed in the code. Consider using a constant or a resourcestring instead of a hard-coded literal string. Using literal strings often makes your code harder to read and maintain. You may use this report to locate and document all strings, for instance when translating your program to another language. This report consists of five sections:

Resourcestrings (from Delphi 3) (LSTR1)

Literal strings declared as constants, more than one character (LSTR2)

Literal strings in code, more than one character (LSTR3)

Literal strings that could be replaced with constants/resourcestrings, more than one character (LSTR4)

Literal numbers in code (LSTR5)

The last section only reports numbers that are $< > 0$.

The sorting done depends on the sort mode setting in [Options|Properties - General](#), so either the sorting will be according to module or according to the string itself.

```
*****
*                               Literal Strings Report for                               *
*                               C:\PROJEKT\RAMVERK\GCACHE.PAS                             *
*****

Literal strings declared as constants, more than one character (136):
-----

' --'                               GPeriod ( 25)
'- not used -'                       GCommon ( 229)
'- not selected -'                   GCommon ( 228)
'.avi'                               GCommon ( 174)
'.bin'                               GCommon ( 121)
'.bmp'                               GCommon ( 142)
'.db'                                GCommon ( 139)
'.dbf'                               GCommon ( 127)
```

See also:

[Reference Reports](#)

14.3.6 Subprogram Index Report

Targets: All

This report produces an index of all subprograms. Procedures, functions, constructors, destructors, and methods are reported. The indentation at the beginning of each line indicates which routines are nested within one another. This report helps you quickly locate any subprogram in your source code. Here is an example:

```
*****
*                               Subprogram Index Report for                               *
*                               C:\PROJEKT\PSEARCH\PSEARCH.DPR                             *
*****

Abbreviations: C=Constructor  D=Destructor  F=Function
               FM=Function (method)  MB=Program main block
               P=Procedure  PM=Procedure (method)
               UF=Unit finalization  UI=Unit initialization

psearch (MB).....psearch (10)
TPSMainForm.chkAllFilesClick (PM).....psform\TPSMainForm (53)
TPSMainForm.CmdQuitClick (PM).....psform\TPSMainForm (57)
TPSMainForm.CmdSaveClick (PM).....psform\TPSMainForm (47)
TPSMainForm.CmdSearchClick (PM).....psform\TPSMainForm (46)
TPSMainForm.CmdSelStartDirClick (PM).....psform\TPSMainForm (48)
TPSMainForm.CmdSpecial1Click (PM).....psform\TPSMainForm (58)
TPSMainForm.CmdSpecial2Click (PM).....psform\TPSMainForm (59)
TPSMainForm.CmdStopClick (PM).....psform\TPSMainForm (55)
TPSMainForm.FormCloseQuery (PM).....psform\TPSMainForm (56)
TPSMainForm.FormCreate (PM).....psform\TPSMainForm (49)
TPSMainForm.FormDestroy (PM).....psform\TPSMainForm (50)
TPSMainForm.FormShow (PM).....psform\TPSMainForm (54)
TPSMainForm.GeneralSearch (PM).....psform\TPSMainForm (67)
TPSMainForm.SpecialSearch (PM).....psform\TPSMainForm (68)
TPSMainForm.txtPatternChange (PM).....psform\TPSMainForm (52)
TPSMainForm.txtRootDirChange (PM).....psform\TPSMainForm (51)
TPSMainForm.UpdateStatus (PM).....psform\TPSMainForm (69)
```

See also:

[Reference Reports](#)

14.3.7 Bindings Report

Targets: All

This report produces a list of all subprograms (functions/procedures) in your source code. For each subprogram, all outside subprograms, variables etc that are referenced (used) by the subprogram are listed. This information measures how dependent a subprogram is of other code.

Subprograms with few bindings are more generic and standalone. For instance, use this report when you change a subprogram, to locate which other parts of the code that you need to test.

```
*****
*                               Bindings Report for                               *
*                               C:\PROJEKT\RAMVERK\GCACHE.PAS                             *
*****

Bindings:
-----
```

Abbreviations: C=Constructor D=Destructor F=Function
 FM=Function (method) MB=Program main block
 P=Procedure PM=Procedure (method)
 UF=Unit finalization UI=Unit initialization

```
GDebug (UF)..... GDebug (425)
  ref AllocList : TList          Var, Global      GDebug (68)

GDebug (UI)..... GDebug (413)
  ref AllocList : TList          Var, Global      GDebug (68)
  set AllocList : TList          Var, Global      GDebug (68)
  set gDebugLevel : Integer      Var, Interfaced  GDebug (17)
  set MarkMemCount : Integer     Var, Global      GDebug (72)
  set MarkMemSize : Integer      Var, Global      GDebug (71)

AssignPeriodFormats (P)..... GPeriod (38)
  unk fpFormats : Array (static) Typed const, Interfaced GPeriod (102)
```

See also:

 [Reference Reports](#)

14.3.8 Third-party Dependencies Report

Targets: All

This report has five sections:

- Third-party folders directly referenced (THPA1)**
- Third-party units directly referenced (THPA2)**
- Third-party identifiers directly referenced (THPA3)**
- Third-party identifiers directly referenced, with locations (THPA4)**
- Third-party class and interface types directly referenced (THPA5)**

All code is considered to be third-party (external), if it is located in a folder that is different than the folder for the project main file. Possibly this will also include code that is not really third-party (external), but your own.

The report will give you an overview of how your code depends on third-party code. You may for example discover that a particular set of third-party code is only used briefly. In that case you may decide to replace it to get rid of the dependency.

The report sections showing identifiers include ALL found identifiers, even those that are located in folders that are excluded for reporting. The reason for this is that you otherwise would have to include these folders to get this information, leading to much bigger lists for other reports.

Also note, that this report gives the best results if you select to parse all code (option "All files" in project options, [General](#) tab page).
 As explained above, you can use the option "Exclude identifiers from other folders in reports.." to avoid big lists in other reports.

There is an option "Also subfolders". If this option is checked, code that is located in subfolders below the main folder, is NOT considered to be third-party code.

```

*****
*                               Third-party dependencies Report for          *
*                               C: \PROJEKT\PIM\EPA.DPR                      *
*****

```

(9) Third-party folders directly referenced:

```

-----
C: \Program Files\FastReports\FastReport\Source
D: \3rdParty\TurboPower\Orpheus\source
D: \3rdParty\TurboPower\ShellShock\source
D: \3rdParty\TurboPower\SysTools\source
D: \3rdParty\VirtualTreeview

```

(36) Third-party modules directly referenced:

```

-----
BAbout
BCat
BFile
BGlobals
BGuard
BIcon
BRes
BToDo
BUtils

```

See also:

 [Reference Reports](#)

14.3.9 Most Called Report

Targets: All

This report lists, in descending order, the subprograms that are most often called. Please note that this number just reflects the number of locations in the code where the particular subprogram is called. It cannot predict how many times a subprogram is called when the program is run. Instead use a profiler tool to obtain this information.

```

*****
*                               Most Called Report for                      *
*                               C: \PROJEKT\PSEARCH\PSEARCH.DPR             *
*****

```

These subprograms are called from two or more locations (4):

5	UpdateStatus	Proc, Method	psform\TPSMainForm (69)
3	GetElapsedTimeString	Func, Method	GTimer\TTimeMeasurer (21)
2	Create	Constructor	PSEngine\TPatternSearcher (62)
2	Search	Proc, Method	PSEngine\TPatternSearcher (68)

See also:

 [Reference Reports](#)

14.3.10 Call Tree Report

Targets: All

Source code consists of calls from subprograms to other subprograms. These

subprograms in turn call other subprograms. All these calls form hierarchical call trees. Note that PAL often will create several separate trees that form the hierarchy. Use this tree to get an understanding of the calls between subprograms in your source code.

Subprograms from units in [excluded folders](#) are not reported.

Each branch in the call tree is assigned a number. The string “.” followed by a branch number indicates that the particular section has already been drawn in the hierarchy. For HTML reports, a link is created, allowing you to easily find the section.

There is an option “All” for the Call Tree Report. If you select this option, PAL also reports branches that have already been drawn. This can result in a very long call listing. The default value is False.

Recursive calls are marked with “(rec)”. Here is an example of a Call Tree Report:

```
psform.TPSMainForm.chkAllFilesClick  
psform.TPSMainForm.UpdateStatus
```

```
psform.TPSMainForm.CmdSearchClick  
psform.TPSMainForm.GeneralSearch  
psform.TPSMainForm.SpecialSearch  
psform.TPSMainForm.UpdateStatus
```

```
psform.TPSMainForm.FormShow  
psform.TPSMainForm.UpdateStatus
```

```
psform.TPSMainForm.txtPatternChange  
psform.TPSMainForm.UpdateStatus
```

See also:

 [Reference Reports](#)

14.3.11 Reverse Call Tree Report

Targets: All

This report is like the [Call Tree Report](#), but reverse. It produces an outline of the call structure of your program, like the Call Tree Report. But it does so in reverse order. This means it starts with the subprograms that are only called, and that not call any other subprograms. This report can help you get a better understanding of how your low-level routines are used.

See also:

 [Reference Reports](#)

14.3.12 Call Index Report

Targets: All

This report is similar to the [Call Tree Report](#). For every subprogram, it lists which other subprograms that are called. In addition, it lists all calls from other subprograms to the particular subprogram.

Subprograms from units in [excluded folders](#) are not reported.

```
*****
*                               Call Index Report for                               *
*                               C: \PROJEKT\RAMVERK\GCACHE. PAS                       *
*****

Call Index:
-----

GAsmCode._FastSameText, GAsmCode (10) called by (5):

  GTools.CheckStringIsValid GTools (221)
  GTools.IsFullPathToDir    GTools (75)
  GTools.IsPartStr          GTools (74)
  GTools.SameDirectory      GTools (224)
  GTools.SameExtension      GTools (225)

GCACHE.TAbstractCache.AddToCache, GCACHE.TAbstractCache (29) calls (1):

  GIntList.TIntList.AddInteger          GIntList.TIntList (18)
```

See also:

 [Reference Reports](#)

14.3.13 Exception Report

Targets: All except BP7

The Exception Report currently consists of two sections:

Exception Call Tree (EXCP1)

This section is like the [Reverse Call Tree Report](#), but it also describes how exceptions are handled.

Use this section to identify parts of your code that are not protected, and that will allow exceptions to bubble up to the highest level.

Raised exceptions (EXCP2)

This section produces a list of all raised exception types.

See also:

 [Reference Reports](#)

14.3.14 Brief Cross-reference Report

Targets: All

This report lists for every identifier, all locations where it is referenced and set (but only locations in modules that are not excluded for reporting). The same results are produced as in the [Cross-reference Report](#), but with another formatting. For objects it also reports locations where the object is created and freed. Identifiers that are only declared, but never used, are not included. Here is an excerpt from such a table:

```
*****
*                               Brief Cross-reference Report for          *
*                               C:\PROJEKT\RAMVERK\GCACHE.PAS              *
*****

Abbreviations: c=Created f=Freed i=Implemented r=Referenced s=Set u=Unknown v=Varparam

Brief crossreference:
-----

_FastSameText          Func, Interfaced          GAsmCode (10)
  GAsmCode              90i
  GTools                782r 805r 2070r 2084r 2092r

AChar : Char           ValParam                  GTools\CharExistsPas (239)
  GTools                2160s

AddInteger              Func, Method              GIntList\TIntList (18)
  GCACHE                125r 126r 127r 375r
  GIntList               46i 57r
```

See also:

 [Reference Reports](#)

14.3.15 Cross-reference Report

Targets: All

This report lists for every identifier, all locations where it is referenced and set (but only locations in modules that are not excluded for reporting). For objects it also reports locations where the object is created and freed. Identifiers that are only declared, but never used, are not included. The layout is more spacious than in the [Brief Cross-reference Report](#), but the content is equivalent. Here is an excerpt from such a table:

```
*****
*                               Cross-reference Report for                *
*                               C:\PROJEKT\RAMVERK\GCACHE.PAS              *
*****

Abbreviations: cre=Created fre=Freed imp=Implemented ref=Referenced
                set=Set unk=Unknown var=Varparam

Cross-reference:
-----

_FastSameText          Func, Interfaced
  dec  GAsmCode (10)
  imp  GAsmCode (90)
  ref  GTools\CheckStringIsValid (2070)
  ref  GTools\IsFullPathToDir (805)
  ref  GTools\IsPartStr (782)
  ref  GTools\SameDirectory (2084)
  ref  GTools\SameExtension (2092)
```

```

AChar : Char                               ValParam
dec   GTools\CharExistsPas (239)
unk   GTools\CharExistsPas (2160)

```

Please note a special case. Consider this rather meaningless code:

```

1  program Prop;
2
3  type
4    TMyClass = class
5      private
6        FMyField : Integer;
7      public
8        property MyField : Integer read FMyField write FMyField;
9      end;
10
11 var
12   Obj : TMyClass;
13 begin
14   Obj := TMyClass.Create;
15
16   try
17     if Obj.MyField = 5 then
18       Obj.MyField := 555;
19   finally
20     Obj.Free;
21   end;
22 end.

```

It will produce this report:

Cross-reference:

```

-----
FMyField : Integer                ClassField
dec   Prop\TMyClass (5)
ref   Prop (7)
ref   Prop\ (16)
set   Prop\ (17)

MyField : Integer                Property
dec   Prop\TMyClass (7)
ref   Prop\ (16)
set   Prop\ (17)

Obj : TMyClass                   Var, Global
cre   Prop\ (14)
dec   Prop (12)
fre   Prop\ (19)
ref   Prop\ (16)
ref   Prop\ (17)

TMyClass = Class                 Type, Global
dec   Prop (3)
ref   Prop (12)
ref   Prop\ (14)

```

Note that the property declaration on line 8 just renders a "ref" for FMyField. This is because FMyField is really just formally referenced (in read/write declarations) on this line. However, on line 17-18 the property MyField is referenced and set, so this also gives an implicit "ref" and "set" for FMyField,

See also:

 [Reference Reports](#)

14.3.16 Used Outside Report

Targets: All

For each unit, every identifier is listed that is referenced from other units. This gives a measurement of how general the unit is. This information gives an idea of how general and standalone a specific unit is. Here is an example of such a report:

```
*****
*                               Used Outside Unit Report for          *
*                               C: \PROJEKT\PSEARCH\PSEARCH. DPR        *
*****
```

These interfaced identifiers are used outside their units

PsForm:

PSMainForm : TPSMainForm	Var, Interfaced	psform (74)
TPSMainForm = Class	Type, Interfaced	psform (13)

See also:

 [Reference Reports](#)

14.3.17 Uses Report

Targets: All

For a normal PAL project, this report consists of eight sections:

Unit usage (USES1)
Modules not needed in the Delphi project file (USES2)
Modules not added to the Delphi project file (USES3)
Unit references (USES4)
Optimal uses list (USES5)
Runtime initialization order (USES6)
Mutual unit references (USES7)
All modules (USES8)

For multi-projects there are two sections:

Units used by the projects (MUSE1)
Units used by all projects (MUSE2)

Please note that the Uses Report reports units from all folders, even from those that are set as excluded in the project options.

An additional file Lattix.xml is always created in the report directory. It is a file that you can use to integrate with Lattix products, if you use them.

Unit usage

This report section lists all units that PAL is able to find. For each unit, the *uses*

statements are analyzed. The units specified in these clauses are listed, indicating if they are actually used. Units that have initialization sections are specially marked. It is often desirable to keep those units in the *uses* lists, even if they are not formally needed, in order to execute the initialization code.

Removing unused uses references has multiple benefits:

- cleaner code to maintain, no need to bother about code that is not used
- code from initialization and finalization sections in unused units is not linked in and run, reducing the size of the EXE
- resources (icons, bitmaps etc) used by referenced units are not linked in
- compilation runs smoother and quicker

Units specified in the **interface** *uses* list are categorized as:

Unnecessary
Used in unknown way
Used in interface
Used in implementation
Used by inherited form

You can probably remove units categorized as "unnecessary" from the *uses* list. If they are categorized as "used in implementation", move them to the *uses* list in the implementation section.

Units used by base forms (in form inheritance), are also listed by the Delphi IDE for child forms even if they are not needed for the compilation. Trying to remove such a reference will only result in the IDE reinserting the declaration when the unit is saved. These situations are detected and the unit is listed as "used by inherited form" instead of "unnecessary".

Units specified in the **implementation** *uses* list are categorized as:

Unnecessary
Used in unknown way
Used in implementation

You can probably remove units categorized as "unnecessary" from the *uses* list.

A special case is Delphi project files (DPR-files). The units listed in the *uses* list in the DPR-file are often not referenced in that file, and PAL consequently categorizes them as *unnecessary*. Nevertheless, you probably want to keep them in that *uses* list, so they are available to Delphi's Project Manager.

When you drop a VCL component on a Delphi form the Delphi IDE automatically adds the unit or units required by the component to the interface section *uses* statement. This is done to ensure that the form file (DFM/XFM) can locate the code needed to stream the form and components. Even if you later remove the component, the units are not deleted from the *uses* statement. This causes the need to sometimes clean the *uses* statement of unused units. In addition, units that you add manually may be left behind in the *uses* statements if you later delete the code sections that need these units.

Even if the compiler does not include code from units that are never used, the

initialization section (or finalization section) is included in the EXE file regardless of whether any methods from the unit are used or not. Also resources may be linked into the final EXE. One tip here is to generate and examine the MAP-file. The initialization section is also always executed when the application starts. This is a good reason why you should remove superfluous units from the *uses* lists. It will also make compilations more efficient. The fewer units in the *uses* clause, the quicker the compiler can do its job.

Sometimes, PAL determines that a unit is not needed, but Delphi reinserts it anyway into the uses-list when the unit is saved. This happens when working in the Delphi IDE.

If a project uses conditional compilation, maybe a unit is sometimes needed and sometimes not; such a unit must remain within a project. You could wrap conditional defines around such a unit declaration in the *uses* clause, like:

```
1 | uses
2 | Common, (*$IFDEF Special*) Spec, (*$ENDIF*), Utils;
```

There is an option "Only warnings" for the Uses Report in the Properties dialog. Select this checkbox to only report those units that can be removed from the uses list or moved from the interface to implementation uses list. The default value is FALSE.

Modules that are referenced in the Delphi project file, but not used

This section lists all units that are included in the Delphi project file, but are not used. These references can probably be removed from the project file. One reason to include them in the project is to make them accessible from the *Project Manager* in the Delphi IDE.

Modules that are used but not added to the Delphi project file

This is a list of units that are used, but not added to the Delphi project file. One reason to include them in the project is to make them accessible from the *Project Manager* in the Delphi IDE.

References

This section lists for every unit, which other units that are referenced. It also lists all units that reference this unit.

Optimal uses list

This is a list of the units, in the order that they are parsed by PAL.

It is a good idea to keep general units placed before less general units in the uses lists, for the compiler to run as smoothly and quickly as possible. You can use the Optimal Uses list to achieve this aim.

Runtime initialization order

This section lists the order in which the initialization sections are executed at runtime.

Units used by the projects

This section, which is available only for multi-projects, lists all units used by the projects.

Units used by all projects

This section, which is available only for multi-projects, lists all common units used by all analyzed projects.

Mutual unit references

This section lists units that reference each other. This leads to a strong coupling between the units, which is usually to be avoided.

All modules

This section lists all referenced units in the project, regardless if the source is found or not. Use the section to get an overview of the used units.

See also:

 [Reference Reports](#)

14.3.18 Conditional Symbols Report

Targets: All

This report has different sections presenting information about conditional symbols.

Conditional symbols that were defined (\$DEFINE) in code (COND1)

This is a list of all conditional symbols that were defined in the parsed code. Use this list for a sanity-check that your code defines the correct symbols. This section also reports symbols that are defined in code that is excluded for reporting.

Conditional symbols that were undefined (\$UNDEF) in code (COND2)

This is a list of all conditional symbols that were undefined in the parsed code. Use this list for a sanity-check that your code defines the correct symbols. This section also reports symbols that are defined in code that is excluded for reporting.

Conditional symbols/expressions that were evaluated as true (\$IF/\$IFDEF/\$ELSEIF) (COND3)

This is a list of all conditional symbols or expressions that at some point in the code were

evaluated as true.

Conditional symbols/expressions that were evaluated as false (\$IF/\$IFDEF/\$ELSEIF) (COND4)

This is a list of all conditional symbols or expressions that at some point in the code were evaluated as false.

Unnecessary \$DEFINE/\$UNDEF (COND5)

This section lists locations in the source code where a \$DEFINE or \$UNDEF directive is repeated unnecessarily. The directive can normally be removed from the location where it is repeated, giving code that is easier to comprehend and maintain.

Example

```
1  ..  
2  (*$DEFINE Final*)  
3  var  
4    X : Integer;  
5    Y : Integer;  
6  
7  (*$DEFINE Final*) // unnecessary, Final is already defined  
8  const  
9    sAppName = 'AppName';  
10 (*$ENDIF*)  
11 ..  
12 (*$ENDIF*)
```

Unnecessary \$IFDEF/\$IFNDEF (COND6)

This section lists locations in the source code where a \$IFDEF or \$IFNDEF directive is repeated unnecessarily. The directive can normally be removed from the location where it is repeated, giving code that is easier to comprehend and maintain.

Example

```
1  (*$IFDEF Special*)  
2  var  
3    X : Integer;  
4    Y : Integer;  
5  
6  (*$IFDEF Special*) // this directive is unnecessary  
7  S : string;  
8  (*$ENDIF*)  
9  (*$ENDIF*)
```

\$DEFINE/IFDEF used but no matching \$IFDEF/\$IFNDEF in code

This section lists conditional symbols that were defined (\$DEFINE/IFDEF) either in code, or by options, but are not referenced (\$IFDEF/\$IFNDEF) in the code. Those conditional symbols are probably unnecessary and can be removed.

See also:

 [Reference Reports](#)

14.3.19 Directives Report

Targets: From Delphi 6

This report displays information about various directives.

Identifiers marked with the “deprecated” directive (DIRE1)

The “deprecated” directive indicates that an item is obsolete or supported only for backward compatibility.

Identifiers marked with the “experimental” directive (DIRE2)

The “experimental” directive indicates that the identifier is created for experimental purposes.

Identifiers marked with the “library” directive (DIRE3)

The “library” directive indicates that the identifier may not exist or the implementation may vary considerably on different library architectures.

Identifiers marked with the “platform” directive (DIRE4)

The “platform” directive indicates that the identifier may not exist or that the implementation may vary considerably on different platforms.

Subprograms marked with the “inline” directive (DIRE5)

This section lists subprograms that are marked with the “inline” directive.

See also:

 [Reference Reports](#)

14.3.20 To-Do Report

Targets: Delphi 5 and later

This report displays information about To-Do items that are entered in the source code or by the IDE. Results are sorted after module, priority and category.

Open To-Do items (TODO1)

This section lists open To-Do items. For each item, the text, module, priority, owner and category is displayed.

Closed To-Do items (TODO2)

This section lists closed To-Do items.

See also:

 [Reference Reports](#)

14.3.21 Module Call Tree Report

Targets: All

This report shows a call tree for modules. It shows how modules “use” each other by inclusion in the uses lists in the interface and implementation sections.

Example:

Module Call Tree:

```
-----  
psearch  
  PsForm  
    StBrowsr  
      SsBase  
        SsConst  
          SsConst  
            RzPanel  
              RzCommon  
                RzButton  
                  RzCommon  
RzStatus  
  RzCommon  
    RzPrgres  
      RzCommon  
        RzPanel...  
          RzSysRes  
PsEngine  
  StStrL  
    StBase  
      StConst  
        StBase...  
          SsBase...  
            GTimer  
              GCommon  
                StStrL...
```

Branches are not repeated. The sequence “...” marks a branch that already has been written.

See also:

 [Reference Reports](#)

14.3.22 Help Report

Targets: All except BP7

The Help Report gives you information about the linkings between help context values in your Delphi project and the help include file.

All help context values (HELP1)

This section lists all help context numbers that are used in the source code (set in DFM files).

Help context values that are not set in DFM files (HELP2)

This section lists help context numbers that exist in the help include file, but are not used in the source code.

Help topics missing (HELP3)

This section lists all help topics that are missing from the help include file. The help context numbers are used in the source code, but are not listed in the include file.

The help include file, should consist of lines with this format:

TopicName = TopicNumber

See also:

 [Reference Reports](#)

14.3.23 Searched Strings Report

Targets: All

This report searches for strings in the source code. It reports the presence or the lack of strings. It does not matter if the strings are found in code that may not be compiled, because of compilation directives.

You may use this report for example to verify that all your files contain your copyright notice, or that they contain specific include files etc.

Found strings (SEAR1)

This section lists files where the strings are found.

Not found strings (SEAR2)

This section lists files where the strings are **not** found.

See also:

 [Reference Reports](#)

14.3.24 Map File Report

Targets: All

The Map File Report is based on a MAP file, and reports which modules (units) that are included in the executable. You can use it to verify that there are not any modules linked in that should not be.

Linked modules (MAPF1)

This section lists modules that are included in the executable according to the MAP file. Size in bytes is also displayed.

Example:

MAPF1-Linked modules:

- BAbsRpt (22076)
- BContRpt (2904)
- BGlobals (160)
- BSelFile (2640)
- BThread (172)
- BUtils (164)
- GBrowEve (496)
- GBrowUti (1556)
- GControls (2316)
- GDelphiBitmaps (708)
- GDyn2Arr (784)
- GDynArr (4320)
- GFont (120)
- GGlobals (172)
- ..

Linked modules by size (MAPF2)

This section lists modules (sorted by size) that are included in the executable according to the MAP file.

Example:

MAPF2-Linked modules by size:

- PFuncs (730940)
- PPreXE8 (653652)
- PPre10 (650076)
- PPre101 (640536)
- PPreXE7 (551948)
- PPreXE6 (538068)
- PPreXE5 (538040)

```
JclStrings (526988)
PPreXE4 (446668)
System.Classes (288604)
PPreXE3 (262884)
..
```

See also:

 [Reference Reports](#)

14.3.25 Clone Report

Targets: All

The Clone Report tries to identify similar sections of code (code clones).

Expect this report to develop over time, as we add improvements and optimizations.

Similar subprograms internally in module (CLON1)

This section lists subprograms within the same module, with similar code. Only subprograms with at least five lines of code are considered.

Also subprograms that are overloaded will not be examined.

A percentage is calculated to indicate the similarity between the two compared subprograms. A value of 0% meaning a perfect match.

The display shows for each compared pair of functions, a top line with the percent value and the names of the two subprograms and their line numbers.

Example:

```
0,0% STReducer.TReducer.ByNumLastWeeks {132-149} STReducer.TReducer.ByRowLastWeeks {226-243}
...
132   Prev1Week := FindPrevWeek(FBet.Week, 1);
133
134   if NumWeeks > 1 then
135     Prev2Week := FindPrevWeek(FBet.Week, 2)
136   else
137     Prev2Week := nil;
138
139   if NumWeeks > 2 then
140     Prev3Week := FindPrevWeek(FBet.Week, 3)
141   else
142     Prev3Week := nil;
143
144   NumReduced := 0;
145   ReduceWeek(Prev3Week);
146   ReduceWeek(Prev2Week);
147   ReduceWeek(Prev1Week);
148
149   FBet.RedLastWeeksMinMax := NumReduced;
...
...
226   Prev1Week := FindPrevWeek(FBet.Week, 1);
227
228   if NumWeeks > 1 then
229     Prev2Week := FindPrevWeek(FBet.Week, 2)
230   else
231     Prev2Week := nil;
232
233   if NumWeeks > 2 then
234     Prev3Week := FindPrevWeek(FBet.Week, 3)
235   else
236     Prev3Week := nil;
237
238   NumReduced := 0;
239   ReduceWeek(Prev3Week);
240   ReduceWeek(Prev2Week);
241   ReduceWeek(Prev1Week);
242
243   FBet.RedLastWeeksMaxRow := NumReduced;
...
```

Code snippets are displayed for the two similar sections. Double-click on the first line (first or second subprogram name) to jump to the relevant code section in the source (or in the Delphi IDE depending to your settings).

Similar subprograms over all modules (CLON2)

This section lists subprograms over all modules, with similar code. Only subprograms with at least five lines of code are considered.

Also subprograms that are overloaded will not be examined.

A percentage is calculated to indicate the similarity between the two compared subprograms. A value of 0% meaning a perfect match.

The display shows for each compared pair of functions, a top line with the percent value and the names of the two subprograms and their line numbers.

Example:

```

0,0% STWeek.TWeek.GetNumFavWins (345-354) STUtils.GetNumFavWinsForRow (1719-1728)
...
345 Result := 0;
346 RowIndex := RankOdds(FRowOdds);
347
348 for I := 1 to 5 do
349 begin
350 Match := RowIndex[I];
351 Sign := LowestOddsAsSign(FRowOdds[Match]);
352
353 if FCorrectRow.GetSign(Match) = Sign then
354 Inc(Result);
...
...
1719 Result := 0;
1720 RowIndex := RankOdds(RowOdds);
1721
1722 for I := 1 to 5 do
1723 begin
1724 Match := RowIndex[I];
1725 Sign := LowestOddsAsSign(RowOdds[Match]);
1726
1727 if Row.GetSign(Match) = Sign then
1728 Inc(Result);
...

```

Code snippets are displayed for the two similar sections. Double-click on the first line (first or second subprogram name) to jump to the relevant code section in the source (or in the Delphi IDE depending to your settings).

Here is an example of a not perfect match, but where the code snippets are similar:

```

6,1% STUtils.HighestOddsAsSign (756-766) STUtils.LowestOddsAsSign (784-794)
...
756 Result := _1;
757 CurrHigh := Odds.For1;
758
759 if Odds.ForX > CurrHigh then
760 begin
761 Result := _X;
762 CurrHigh := Odds.ForX;
763 end;
764
765 if Odds.For2 > CurrHigh then
766 Result := _2;
...
...
784 Result := _1;
785 CurrLow := Odds.For1;
786
787 if Odds.ForX < CurrLow then
788 begin
789 Result := _X;
790 CurrLow := Odds.ForX;
791 end;
792
793 if Odds.For2 < CurrLow then
794 Result := _2;
...

```

See also:

 [Reference Reports](#)

14.4 Class Reports

The class reports are:

-  [Class Index Report](#)
-  [Class Summary Report](#)
-  [Class Hierarchy Report](#)
-  [Class Field Access Report](#)

See also:

[Reports](#)

14.4.1 Class Index Report

Targets: All

This report presents important facts about every class in the class hierarchy, like type of class member and scope. More specific, it presents for every method the scope and inheritance information.

```
*****
*                               Class Index Report for                *
*                               C: \PROJEKT\RAMVERK\GCACHE. PAS          *
*****
```

TAbstractCache in GCACHE (13) inherits from TObject

Name	Type	Scope	Directive	Current/Introduced
AddToCache	Procedure	Public	Static	TAbstractCache -
Count	Property	Public	Static	TAbstractCache -
Create	Constructor	Public	Virtual	TAbstractCache TObject
CreateDates	Field	Private	Static	TAbstractCache -
CreateTimes	Field	Private	Static	TAbstractCache -
DeleteAll	Procedure	Public	Static	TAbstractCache -
DeleteEntry	Procedure	Public	Virtual	TAbstractCache -
DeleteExpiredEntries	Procedure	Public	Static	TAbstractCache -
DeleteUnRequested	Procedure	Public	Static	TAbstractCache -
Destroy	Destructor	Public	Override	TAbstractCache TObject
Expired	Function	Public	Static	TAbstractCache -
Get	Procedure	Public	Static	TAbstractCache -
GetNumCacheEntries	Function	Protected	Static	TAbstractCache -
GetParam	Function	Public	Static	TAbstractCache -
GetParamsDoc	Procedure	Public	Static	TAbstractCache -
Hits	Field	Private	Static	TAbstractCache -
IndexInCache	Function	Public	Static	TAbstractCache -
MaxEntries	Field	Private	Static	TAbstractCache -
Params	Field	Private	Static	TAbstractCache -
TimeLimitMins	Field	Private	Static	TAbstractCache -

TObjectCache in GCACHE (59) inherits from TAbstractCache in GCACHE (13)

Name	Type	Scope	Directive	Current/Introduced
AddObjectToCache	Procedure	Public	Static	TObjectCache -
Create	Constructor	Public	Virtual	TObjectCache TAbstractCache
DeleteEntry	Procedure	Public	Override	TObjectCache TAbstractCache
Destroy	Destructor	Public	Override	TObjectCache TAbstractCache
FreeMemory	Procedure	Public	Virtual	TObjectCache -
FreeOnDelete	Field	Private	Static	TObjectCache -
GetObjectFromCache	Function	Public	Static	TObjectCache -
Pointers	Field	Private	Static	TObjectCache -

See also:

 [Class Reports](#)

14.4.2 Class Summary Report

Targets: All

This report presents for every class a table of the methods, how many are static, inherited etc.

```
*****
*                               Class Summary Report for                *
*                               C: \PROJEKT\RAMVERK\GCACHE. PAS           *
*****
```

TAbstractCache in GCACHE (13) inherits from TObject

	Inherited	New	Total
	-----	---	-----
Constructors	1	1	2
Destructors	1	1	2
Procedures	7	7	14
Functions	16	4	20

TObjectCache in GCACHE (59) inherits from TAbstractCache in GCACHE (13)

	Inherited	New	Total
	-----	---	-----
Constructors	2	1	3
Destructors	2	1	3
Procedures	14	3	17
Functions	20	1	21

See also:

 [Class Reports](#)

14.4.3 Class Hierarchy Report

Targets: All

This report creates a hierarchical list of all classes. Use this list to get an understanding of the class hierarchy.

```
*****
*                               Class Hierarchy Report for              *
*                               C: \PROJEKT\RAMVERK\GCACHE. PAS           *
*****
```

```
TAbstractCache in GCACHE (13)
  TObjectCache in GCACHE (59)
    TPointerCache in GCACHE (75)
    TStringCache in GCACHE (46)

TIntList in GIntList (16)
  TDoubleIntList in GIntList (26)
```

See also:

 [Class Reports](#)

14.4.4 Class Field Access Report

Targets: All

This is a list of all class fields that are accessed directly from the outside. This means that they are not accessed by means of a property. OOP purists consider this bad practice, and you should avoid this.

```
*****
*                               Class Field Access Report for          *
*                               C: \PROJEKT\NEWPAL\CHECK.DPR              *
*****

F : Untyped File                               ClassField              GFiles\TBinaryFile ( 57)

Referenced from these locations (1):

FixMissingCharInFile                          Func, Interfaced              GFiles (191)
```

See also:

 [Class Reports](#)

14.5 Control Reports

The control reports are:

 [Control Index Report](#)
 [Control Alignment Report](#)
 [Control Size Report](#)
 [Control Tab Order Report](#)
 [Control Warnings Report](#)
 [Property Value Report](#)
 [Missing Property Report](#)
 [Form Report](#)
 [Events Report](#)

The control reports deal with Delphi form files (DFM/NFM/XFM-files). These reports are not available when the compiler target is Borland Pascal 7.

See also:

[Reports](#)

14.5.1 Control Index Report

Targets: All except BP7

This is a list of all controls that are included in form files (DFM-files). Use this report to get a general view of the controls on your forms. Because it only lists controls that are included in the DFM-file, controls from inherited forms will not show up in descendant forms. This is of course unless they change some property, so that they are represented in the DFM-file.

```
*****
```

```

*                               Control Index Report for                               *
*                               C: \PROJEKT\PSEARCH\PSEARCH. DPR                         *
*****
List of all controls (33):

Module: psform

  CmdSave : TBitBtn
  CmdSearch : TBitBtn
  GroupBox1 : TGroupBox
  Label3 : TLabel
  Memo : TMemo
    chkCaseSensitive : TCheckBox
    chkIncludeSubFolders : TCheckBox
    chkListAllSearched : TCheckBox
    chkReadOnly : TCheckBox
    CmdSelStartDir : TButton
    Label1 : TLabel
    Label2 : TLabel
    txtPattern : TEdit
    txtRootDir : TEdit

```

See also:

 [Control Reports](#)

14.5.2 Control Alignment Report

Targets: All except BP7

This report gives a list of the alignment of the controls. You may use this report to detect cases of bad alignment between controls. For instance, if two buttons have a Top-property value of 20 and 26, they are supposed to be aligned, and a horizontal alignment warning is triggered.

See also:

 [Control Reports](#)

14.5.3 Control Size Report

Targets: All except BP7

This report gives a list of the controls and their sizes. You may use this report to detect unwanted size differences. For instance, for aligned buttons on a form, you probably want them to be of the same size.

```

*****
*                               Control Size Report for                               *
*                               C: \PROJEKT\PSEARCH\PSEARCH. DPR                         *
*****

Horizontal size:

    ( none)

Vertical size:

```


Module: psform Parent: GroupBox2

```
chkOnlyWeb (Width = 77) different size relative chkAllFiles (Width = 157)
chkOnlySql (Width = 69) different size relative chkAllFiles (Width = 157)
chkOnlyVB (Width = 117) different size relative chkOnlyDelphi (Width = 125)
chkOnlyTxt (Width = 81) different size relative chkOnlyDelphi (Width = 125)
```

Module: psform Parent: GroupBox1

```
chkCaseSensitive (Width = 101) different size relative chkIncludeSubFolders (Width = 109)
chkListAllSearched (Width = 125) different size relative chkIncludeSubFolders (Width = 109)
chkReadOnly (Width = 125) different size relative chkIncludeSubFolders (Width = 109)
txtPattern (Width = 197) different size relative txtRootDir (Width = 325)
```

See also:

 [Control Reports](#)

14.5.4 Control Tab Order Report

Targets: All except BP7

This report lists all controls that possibly have a bad tab order. Normal tab order is considered as left to right, up down, as given by the TabOrder property.

It is assumed that "TabStop = true" is activated for all controls, unless "TabStop = false" is explicitly set in the DFM/XFM-file. Depending on the default value of the TabStop property, this assumption is sometimes incorrect. A bad tab order as listed in this report, is consequently not always a cause of alarm.

```
*****
*                               Control Tab Order Report for          *
*                               C:\PROJEKT\PSEARCH\PSEARCH.DPR          *
*                               *****                               *
```

TabOrder probably wrong:

Module: psform Parent: PSMainForm

```
Memo (TabOrder=0) tabs upwards to GroupBox1 (TabOrder=1)
```

TabOrder possibly wrong:

Module: psform Parent: PSMainForm

```
RzStatusBar (TabOrder=3) tabs upwards and right to CmdSearch (TabOrder=4)
```

Module: psform Parent: GroupBox1

```
txtPattern (TabOrder=1) tabs upwards and right to chkIncludeSubFolders (TabOrder=2)
chkListAllSearched (TabOrder=4) tabs upwards and right to CmdSelStartDir (TabOrder=5)
```

Module: psform Parent: GroupBox2

```
chkOnlySql (TabOrder=3) tabs upwards and right to chkOnlyDelphi (TabOrder=4)
```

See also:

 [Control Reports](#)

14.5.5 Control Warnings Report

Targets: All except BP7

This report presents several lists, helping you spot possible errors in your form files.

Controls that overlap visually (COWA1)

This is a list of controls that overlap each other visually, possibly hiding each other.

Labels with Caption-property that does not end in ":" (COWA2)

Labels (TLabels) above or to the left of other controls usually end their caption with the char ":". This lists all labels that not confirm to this. Of course, this does not apply to labels that are standalone and just used for display purposes. PAL cannot know the purpose of a label and reports all labels with missing ":".

A false warning is generated for captions that are so long that they span over more than one line in the DFM file.

Conflicting accelerators (COWA3)

This is a list of all controls with conflicting accelerators in the Caption property. Some types of controls are not reported even if they share the same accelerators, because they do not conflict. Those are menu items on different sub menus, and controls that reside on different TTabSheet pages of a TPageControl control.

Labels (or static texts) that have accelerators but FocusControl is not set (COWA4)

Labels and static texts cannot receive focus. When an accelerator key is pressed, focus is given to the control specified by the FocusControl property. It is an error to omit the FocusControl property in this case.

Conflicting shortcuts (COWA5)

This is a list of all menu items with conflicting shortcuts (key combination) (property ShortCut).

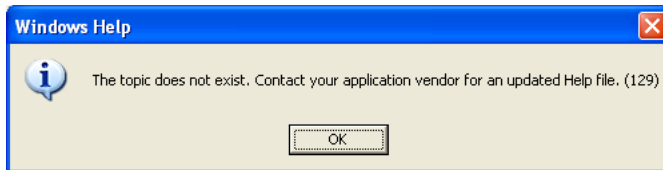
Buttons/menu items with OnClick-event that is unassigned (COWA6)

This is a list of all buttons and menu items with an unassigned OnClick-event. Normally

there should be an action on OnClick for these controls, so it indicates an error in the code. Warnings are not created when the property Action is set. Buttons with ModalResult set will be excluded from the list.

Menu items that have HelpContext=0 (COWA7)

This is a list of all menu items with a HelpContext property value of 0. Probably the menu item has not been assigned a topic in the help file. Failing to assign a topic could trigger this messagebox when the help system is invoked:



You can also use the Missing Property Report to generate this information. Add a "TMenuItem;HelpContext" item to the check list.

Hint is not activated (COWA8)

This is a list of all controls where the Hint property is set, and both ShowHint and ParentShowHint properties are "false".

See also:

 [Control Reports](#)

14.5.6 Property Value Report

Targets: All except BP7

This report shows values for selected properties in your Delphi forms. For instance, it can show every value of the Caption property for all controls of type TButton. Or, show all properties for controls of the type TLabel, or values for all properties with the name SQL, regardless of the class type. Select which properties to monitor in the [Options](#) dialog. Note that the displayed properties and values are those stored in the DFM file. This means that a property possessing a default value is not shown.

```
*****
*                               Property Value Report for                *
*                               C: \PROJEKT\WBT\WBTSTORE. DPR              *
*****
```

GDbLogin

```
DBLoginDialog.BorderStyle = bsDialog
lblUserName.Caption = 'lblUserName'
```

GRTfMod

```
ModalRTFViewerForm.BorderStyle = bsDialog
```

GSelList

```
SelectListDialog.BorderStyle = bsDialog
```

See also:

 [Control Reports](#)

14.5.7 Missing Property Report

Targets: All except BP7

This report is similar to the [Property Value Report](#). You specify one or more types and properties that you want to monitor with this report, for example the class type TCustomLabel and its Caption property. This report will then present a list of all occurrences in DFM-files, where a TCustomLabel (or a descendant like TLabel) is used, but where Caption has **not** been set.

```
*****
*                               Missing Property Value Report for          *
*                               C: \PROJECT\MISSPROP.DPR                    *
*****
(2) Missing Property Value:
-----

MissPropMain

lblWithoutCaption.Caption ( TLabel)
UpDownWithoutAssociate.Associate ( TUpDown)
```

Use this report to for example check that all your TLabel controls have their Caption property set, or that all TUpDown controls have set their Associate property.

For TButton and TBitBtn object with missing OnClick properties, warning will not be given, if a ModalResult property value is set.
Another exception is for TBitBtn object with missing OnClick property, the warning will not be given, if a Kind property value is set.

See also:

 [Control Reports](#)

14.5.8 Form Report

Targets: All except BP7

This report shows important properties for your Delphi forms. For instance, it shows the value of the Position property. If you are like us, you want this property to always have the value poScreenCenter. In this report, you can easily verify that all forms have the correct value.

The Form Report does not report forms of type TDataModule or TWebModule (or their descendants, because these forms are non-visual).

```
*****
*                               Form Report for                            *
*****
```

```

*                                     C: \PROJECT\PSEARCH\PSEARCH. DPR                                     *
*****
Form Module BorderIcons      BorderStyle  FormStyle   Position
-----
psearch      Sys+Min+Max      bsSingle   fsNormal    poDefault   MS Sans Serif
ZAbout       Sys              bsDialog   fsStayOnTop poScreenCenter MS Sans Serif
ZLicFrm       Sys              bsDialog   fsStayOnTop poScreenCenter MS Sans Serif

```

See also:

 [Control Reports](#)

14.5.9 Events Report

Targets: All except BP7

This report shows how events are linked to controls in your Delphi forms.
There are two section:

Controls and linked events (EVEN1) Events and linked controls (EVEN2)

```

*****
*                                     Events Report for                                     *
*                                     C: \PROJEKT\PIM\DETO. DPR                                     *
*****

( 254) Controls and linked events:
-----

Module: BAbout

CmdLicense.OnClick = CmdLicenseClick


Module: GFrView

btnFirst.OnClick = btnFirstClick
btnLast.OnClick = btnLastClick
btnNext.OnClick = btnNextClick

```

See also:

 [Control Reports](#)

15 Main menu

The main menu has seven submenus.

[File menu](#)

[Edit menu](#)

[Search menu](#)

[View menu](#)

[Analysis menu](#)

[Options menu](#)

[Help menu](#)

See also:

[How to use PAL.EXE](#)

[How to use PALCMD.EXE](#)

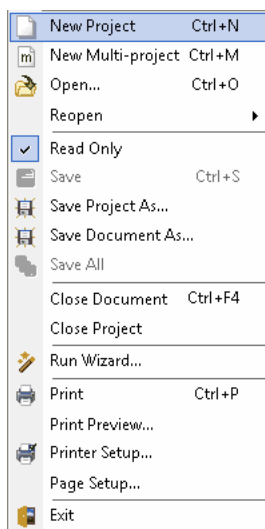
[Introduction](#)

[Known limitations](#)

[Command Line Options for PAL.EXE](#)

[Main window](#)

15.1 File menu



File|New Project (Ctrl+N)

This command creates a new blank project. Settings from the template for projects will be used to initialize the new project.

File|New Multi-project (Ctrl+M)

This command creates a new blank multi-project. Settings from the template for

multi-projects will be used to initialize the new multi-project.

File|Open (Ctrl+O)

Open an existing Pascal Analyzer project or multi-project. The currently opened project is active. There can only be one active project at any single time.

File|Reopen

Under this menu item you can reopen one of the latest opened projects.

File|Read Only

If an editor source file is open, use this menu command to toggle the read-only status.

File|Save (Ctrl+S)

This command saves the currently active editor source file. It will always also save the currently active project.

File|Save Project As

With this menu command, you can save the current project under a different name.

File|Save Document As

If there is an editor window active, you can save the source file under a different name.

File|Save All

This command saves all open documents, and the current active project.

File|Close (Ctrl+F4)

This command closes the current document or report.

File|Close Project

This command closes the current active project.

File|Run Wizard

Run the wizard. The wizard assists you in selecting source code to analyze, and helps you choose other settings that are important for the analysis. If you want complete control over your analysis, use the [Options](#) dialog instead. Some of the settings provided by the Options dialog cannot be set by the wizard.

File|Print (Ctrl+P)

Select between printing the entire viewer text, the selected report, or just the selected text block. Please observe that the printed file may be **very** large.

File|Print Preview

This command shows a preview of the current editor document.

File|Printer Setup

Show Windows standard printer setup dialog box.

File|Page Setup

Show a page setup dialog for an editor document.

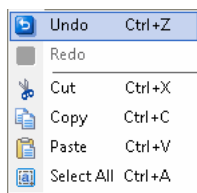
File|Exit

Quit the application. Your settings are saved to the file PAL.INI in your settings folder.

See also:

[Main menu](#)

15.2 Edit menu



Edit|Undo (Ctrl+Z)

Undo the latest action in the editor.

Edit|Redo

Redo the latest action in the editor.

Edit|Cut (Ctrl+X)

Cut the selected editor text to the clipboard.

Edit|Copy (Ctrl+C)

Copy the selected text to the clipboard.

Edit|Paste (Ctrl+V)

Copy the clipboard text to the active editor document

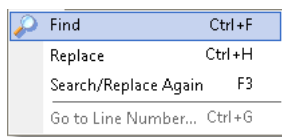
Edit|Select all (Ctrl+A)

Select all text in the currently selected report or editor document.

See also:

[Main menu](#)

15.3 Search menu



Search|Find (Ctrl+F)

Search for a string in the viewer or editor document. This function is not available for reports in HTML format.

Search|Replace (Ctrl+H)

Search and replace a string in the current editor document.

Search|Search/Replace Again (F3)

Repeat the last search/replace action. This function is not available for reports in HTML format.

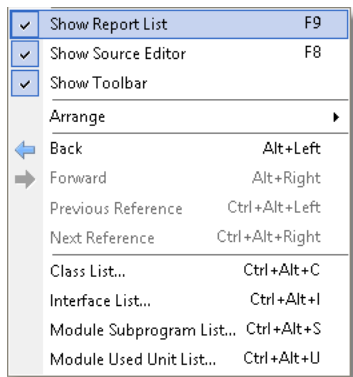
Search|Go to Line Number (Ctrl+G)

Select a line number, and move to that line. This function is not available for reports in HTML format.

See also:

[Main menu](#)

15.4 View menu



View|Show Report List (F9)

Show/hide the report list. You may also click the hotspot button in the center of the bar.

View|Show Source Editor (F8)

Show/hide the source viewer window. You can also click the hotspot button in the center of the bar.

View|Show Toolbar

Mark this menu item if you want the toolbar to appear.
Default = Yes

View|Arrange

Select how to arrange the report list window and the viewer window. Choose between the

following arrangements:

Reports Left, Viewer Right (default)
Reports Top, Viewer Bottom
Viewer Left, Reports Right
Viewer Top, Reports bottom

View|Back (Alt+Left)

Go to the previous editor location.

View|Forward (Alt+Right)

Go to the next editor location.

View|Previous Reference (Ctrl+Alt+Left)

For a selected identifier in the editor, go to the previous reference. The first reference for an identifier is its declaration.

View|Next Reference (Ctrl+Alt+Right)

For a selected identifier in the editor, go to the next reference.

View|Class List (Ctrl+Alt+C)

Displays a list of all reported classes in the project. Select a class to go to its declaration in the source code editor.

View|Module Subprogram List (Ctrl+Alt+S)

Displays a list of all subprograms in the module (unit). Select a subprogram to locate its declaration in the source code editor.

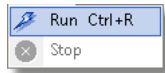
View|Module Used Units List (Ctrl+Alt+U)

Displays a list of all units used by the module (unit). Select a unit to open its source code file in the editor.

See also:

[Main menu](#)

15.5 Analysis menu



Analysis|Run (Ctrl+R)

Start the analysis. PAL scans the selected source file. Any units specified in the *uses* statement of a unit, will optionally be read, if they are found. This is a recursive process, and potentially a lot of source code can be involved. For Delphi source code, if a DFM/NFM/XFM-file is found for a PAS-file, the DFM/NFM/XFM-file will be parsed and examined as well.

If PAL finds any include directives (like `{ $I MYSOURCE.INC }`) in the source code, these files will also be read and parsed.

Even if PAL can detect many syntax errors, it is required that the code is possible to compile. Otherwise, the results may be incorrect and possibly misleading. So, as a rule, always make sure that the code compiles, before examining it with PAL.

Also bear in mind, that the memory consumption and the time needed to parse and produce reports is proportional to the amount of source code and the number of reports selected. For large projects, with lots of source code, it is often wise to start generating just one or a few reports.

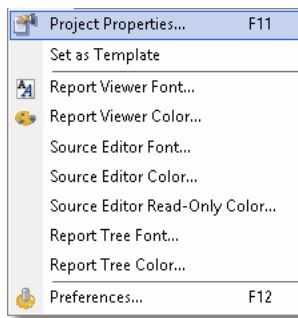
Analysis|Stop

This command stops (cancels) the analysis. Please observe that it may take a few seconds before the analysis stops. Just the [Status Report](#) is created in this case.

See also:

[Main menu](#)

15.6 Options menu



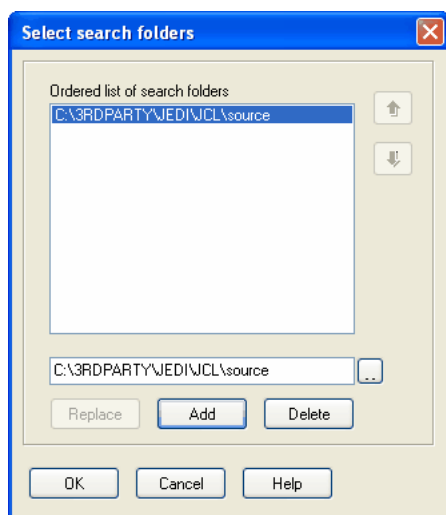
Options|Project Properties... (F11)

Select properties for the currently active project or multi-project. The properties are divided over a number of tab pages in the dialog window. The dialog window has somewhat different content depending on whether a project or a multi-project is active.

[Properties - General](#)
[Properties - Reports](#)
[Properties - Source](#)
[Properties - Format](#)
[Properties - Parser](#)
[Properties - Switches](#)

Click **OK** to confirm your selections and return to the main window.

When selecting paths, conditional defines and other parameters, a special selection dialog is shown:



In this dialog box, select items and add them to a list. For instance, to add a search folder, enter the path in the input field below the list box. Then press the Add-button to

add the folder to the list. Click on the ellipsis button to select a folder by browsing. When selecting excluded folders, it is possible to mark a checkbox, meaning that also subfolders are excluded. Those folders will show up in the main dialog box with a suffix "<+>" attached.

Options|Set as Template

Select this command to save the currently active project or multi-project as a template. The template will be used to initialize new projects. The settings for the templates are stored in the PAL.INI file.

Options|Report Viewer Font

Select a non-proportional font for the report viewer window.
Default = Courier New 10 pt, black color

Options|Report Viewer Color

Select a background color for the report viewer window.
Default = White

Options|Source Editor Font

Select a font for the source editor window. Default = Courier New, 8 pt, black color

Options|Source Editor Color

Select a background color for the source editor window. Default = White

Options|Source Editor Read-Only Color

Select a background color for the source editor window, when the source file is read-only.
Default = White

Options|Report Tree Font

Select a font for the report tree. Default = MS Sans Serif, 8 pt

Options|Report Tree Color

Select a background color for the report tree. Default = White

Options|Preferences... (F12)

Select this command to set options for the entire application. This dialog window has three tab pages.

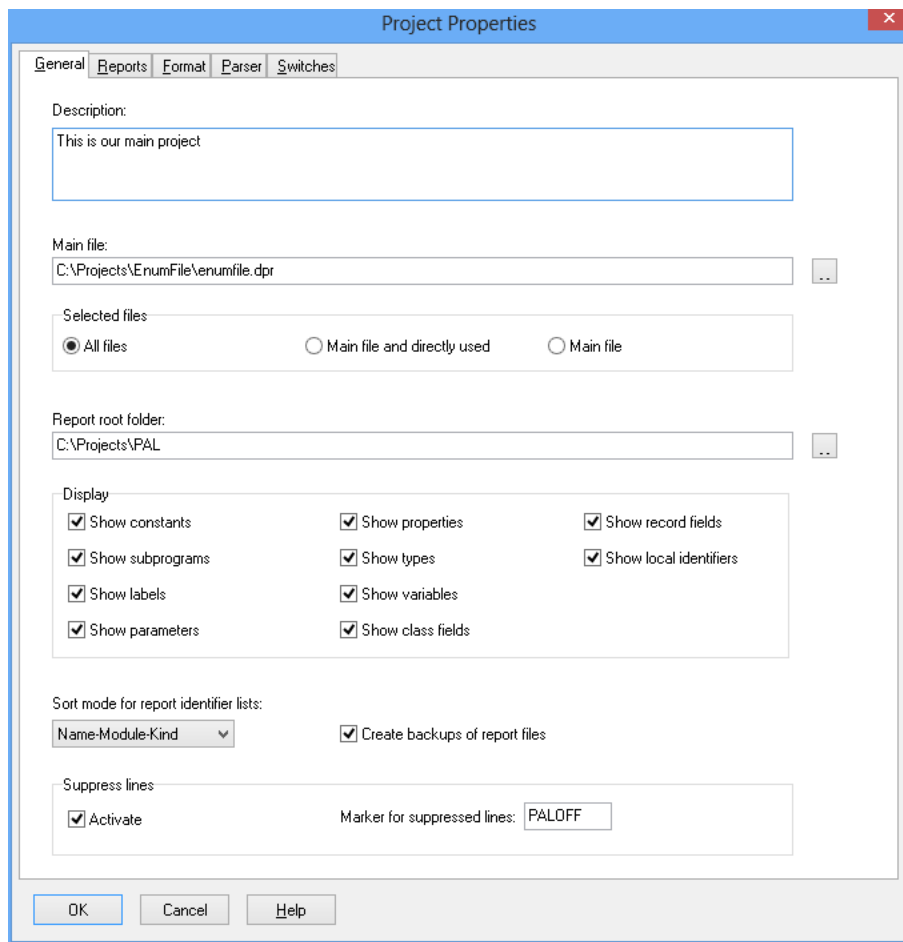
[General](#)
[Source code](#)
[Editor](#)

See also:

[Main menu](#)

15.6.1 Properties - General

For an ordinary Pascal Analyzer project, the General tab looks like this:



Description

Optionally enter a descriptive text for the project.

Main file

Select either a single source file for analysis, or an entire Delphi project (DPR-file), or an entire Borland Pascal project (PAS-file). You can also select a Delphi package (DPK file). PAL will automatically find and use the DPROJ file, if there is any. You can use environment variables here, like for example, "\$ (MYPROJECTS)\Current\MyProj.dpr".

Preferable is to select a complete project (DPR- or DPROJ-file). Pascal Analyzer will then have most chances of resolving identifiers and references.

You can also use a relative path. The path is then relative to the folder where the project file (PAL-file) is located.

Example:

The project file is saved at C:\Projects\MyProj.pap. The main file is C:\Projects\Mine\MyProj.dpr.

You could use the relative path "..\Mine\MyProj.dpr" for the main file.

Selected files

There are three possible settings:

All files

PAL will parse and analyze all found files. This is the recommended selection. The results will be better if PAL has access to as much source code as possible. If you want to limit the output in reports, use the option to exclude folders from reporting on the [Parser](#) tab page.

Main files and directly used (default)

The main files and those files (units) that are listed in the uses lists of the main file, will be parsed and analyzed.

Main file

Only the main file will be parsed and analyzed

Report root folder

Specify the folder where PAL writes the report files upon completion of the analysis. For example, if a project called "MyProject" is analyzed, the report files will be written to a folder "\\MyProject" just below the report root folder. The folder will be created if it does not exist.

The default output root folder is in a folder below "My Documents", like "C:\Documents and Settings\<account>\My Documents\Pascal Analyzer\Projects\Output".

The files are named MYPROG.TXT (or MYPROG.HTM for HTML reports) when MYPROG is examined. If the backup option is selected and a file with the same name exists, this file is saved under the name MYPROG.~XT (or MYPROG.~TM for HTML reports).

Environment variables can also be used here, if you wish.

You can also use a relative path. The path is then relative to the folder where the project file (PAP-file) is located.

Example:

The project file is saved at C:\Projects\MyProj.pap. If you want the report root folder in the same folder, just enter "." as the report root folder.

Create backup of report files

Default = Yes

Mark this checkbox if PAL should create backup files when reports are written. Backup files are given the extension ~XT (or ~TM for HTML reports or ~ML for XML reports). Whenever a new report text file is written, any existing report file will be copied to a backup file.

Show constants

Default = Yes

If you select this option, constants are reported.

Show types

Default = Yes

If you select this option, types are reported.

Show subprograms

Default = Yes

If you select this option, subprograms (procedures/functions) are reported.

Show variables

Default = Yes

If you select this option, variables are reported.

Show labels

Default = Yes

If you select this option, labels are reported.

Show class fields

Default = Yes

If you select this option, class fields are reported.

Show parameters

Default = Yes

If you select this option, parameters are reported.

Show record fields

Default = Yes

If you select this option, record fields are reported.

Show properties

Default = Yes

If you select this option, properties are reported.

Show local identifiers

Default = Yes

If you select this option, local identifiers are reported.

Sort mode for report identifier lists

Default is sorting by Name-Module-Kind

Select the type of sorting you want for identifier lists in the reports. This option affects all reports that show identifier in lists.

Sort modes:

Name-Module-Kind (sort first by Name, then by Module, then by kind of identifier)

Name-Kind-Module

Module-Name-Kind

Module-Kind-Name

Kind-Name-Module

Kind-Module-Name

Suppressed lines-Activate

Default = False

Check this option if you want to suppress lines that are marked with the suppressed lines maker (see the following text block).

Suppressed lines-Marker for suppressed lines

Default = PALOFF

By adding a comment:

```
// PALOFF
```

.. as a comment to a source code line, means that PAL will not report any issues encountered on that line. If you place the comment on a line where an identifier is declared, that identifier will not be reported. This is the most effective way to get rid of all issues for an identifier.

Note also that you can use curly brackets, like "{PALOFF}". Like for "/*" blanks are allowed, for example "{ PALOFF }".

Example

```
type
```

```
TGlobalDLLData = record
  Path : string[255];
  LineNr : integer; //PALOFF (1-based line number in source module)
end;
```

In the code example above, LineNr will not be reported. Observe that the marker must be first in the comment string on the line, and that it is allowed to add comment text to the right of the marker.

In some report sections you will find that even if you place the comment on the line which you believes make PAL report the identifier, it will not have any effect. Then try placing the comment on the line where the identifier is declared.

It is possible to select what string should be used as the marker. Default value for the suppression marker is "PALOFF", but you can change this to something else. Blank spaces between "/" and the suppression marker are allowed. You can also have more text to the right of the marker, like:

```
//PALOFF because false warning otherwise
```

To select only some report sections that will be excluded use this syntax:

Examples:

```
//PALOFF WARN8 (report section WARN8 will not be reported)
```

```
//PALOFF WARN2;OPTI8;OPTI2 (report sections WARN2, OPTI8, OPTI2 will not be reported)
```

```
//PALOFF OPTI (all report sections for the Optimization Report will be excluded)
```

```
//PALOFF STWA2;WARN;OPTI4 (STWA2 and OPTI4 will be excluded, plus all sections in the Warnings Report)
```

When a multi-project is currently active, the General tab holds somewhat different content:

Multi-projects

Press the ellipsis button to select the Pascal Analyzer projects that make up the multi-project. A multi-project must have at least two connected projects.

Remove

Press this button to remove a project from the list.

Exclude identifiers from these folders

Press the ellipsis button to select folders that will be excluded from reports.

The rest of the fields on this tab page have the same meaning as for ordinary projects.

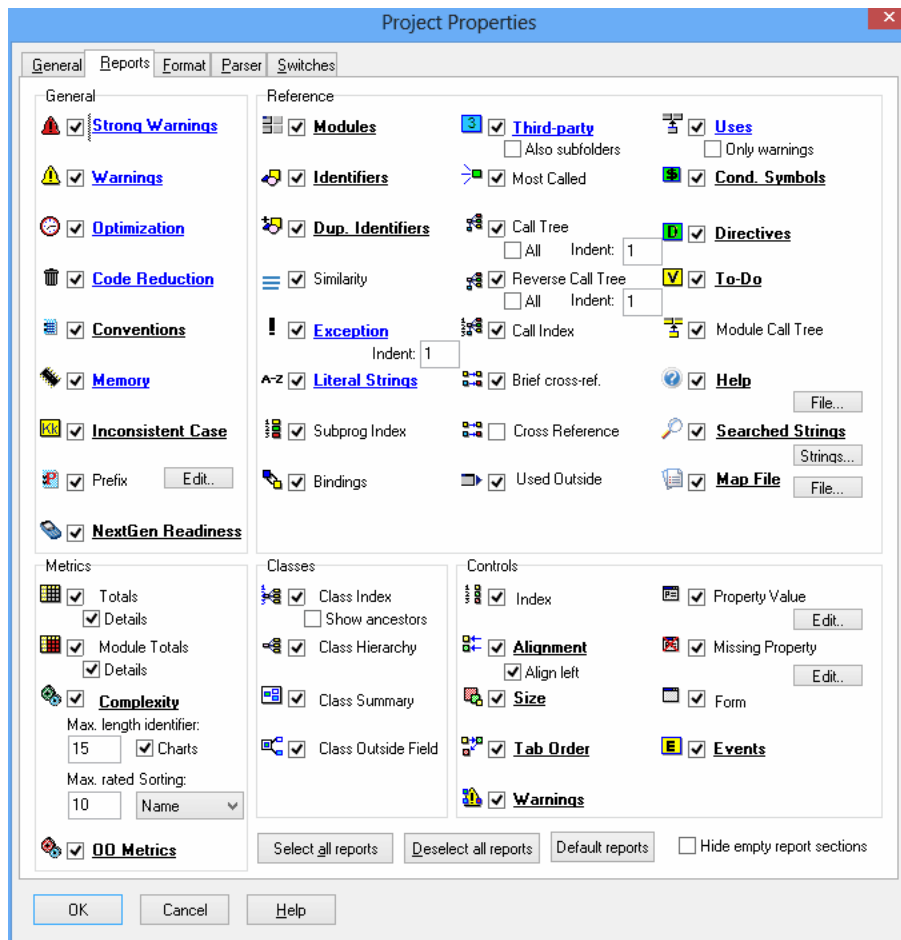
See also:

[Options menu](#)
[Properties - Format](#)

[Properties - Parser](#)
[Properties - Reports](#)
[Properties - Source](#)
[Properties - Switches](#)

15.6.2 Properties - Reports

The **Reports** tab is the same for normal projects and multi-projects. However, when a multi-project is active, only those reports and settings that are relevant for a multi-project, are enabled.



On this tab select the reports you want PAL to create for the current project.

Reports with two or more sections are underlined. Click on the underlined text to open up a new dialog window. In this dialog, you can select the sections that you want to generate.

Also for the [Convention Report](#), use the Prefix button to select prefixes for report sections CONV24, CONV25, CONV26 and CONV27.

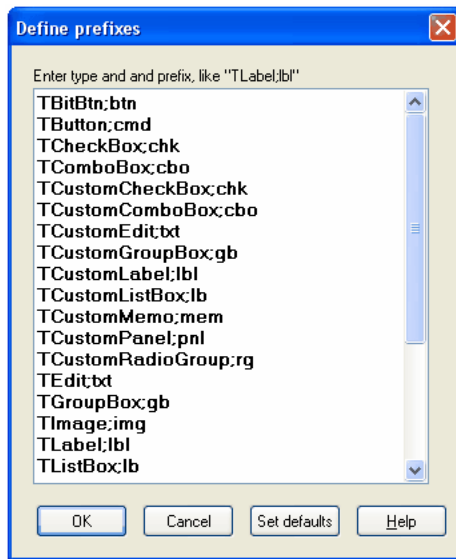
As default, all sections are created.

If the report caption is displayed in red color, this means that the report is selected, but there are no sections selected for it. If the caption is displayed in blue color, it indicates that some sections, but not all, are selected.

To select all reports, press the special button (or **Alt+A**). To deselect all reports, press the special button (or **Alt+D**).

For some of the reports, you may set further options by pressing the **Edit**-button. These reports are the [Prefix Report](#), [Property Value Report](#) and the [Missing Property Report](#).

For the [Prefix Report](#), this button will open a dialog, where you enter definitions for the prefixes that you want checked:



Enter a definition on the form *typename;prefix* like

TCustomLabel;lbl

In this case, PAL will check that every TCustomLabel object (or descendant of a TCustomLabel, like TLabel) that is found, begins with the (case-sensitive) letters "lbl".

If you press the button **Set Defaults** the following prefix definitions are inserted:

TBitBtn;btn
TButton;cmd

TCheckBox;chk
TComboBox;cbo
TCustomCheckBox;chk
TCustomComboBox;cbo
TCustomEdit;txt
TCustomGroupBox;gb
TCustomLabel;lbl
TCustomListBox;lb
TCustomMemo;mem
TCustomPanel;pnl

TCustomRadioGroup;rg

TEdit;txt

TGroupBox;gb

TImage;img

TLabel;lbl

TListBox;lb

TMainMenu;mnu

TMemo;mem

TMenu;mnu

TPanel;pnl

TPopupMenu;mnu

TRadioButton;rb

TRadioGroup;rg

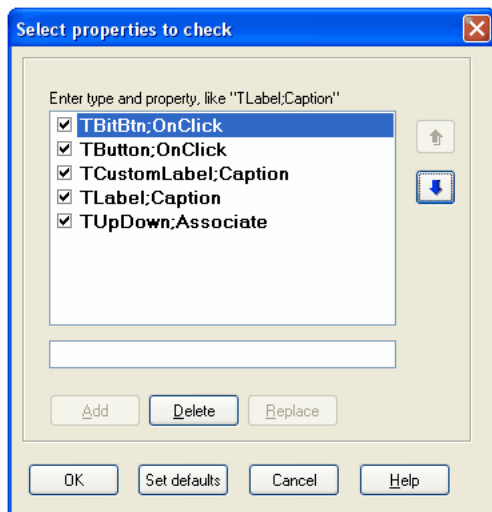
TSpeedButton;btn

These prefixes are just suggestions; you may want to use your own prefix conventions.

The ordering of the items is irrelevant; however you may want to keep them in alphabetical order for clarity.

Also note that prefixes are evaluated for ancestors. For example, if you have entered a single rule "TCustomEdit;txt" and a TEdit-control exists with the name "Edit1", it will be marked as having a bad prefix. There is no rule entered for TEdit, so the rule for the ancestor TCustomEdit will be applied. This requires that the source code for TEdit is parsed, so that PAL is able to determine that the ancestor is TCustomEdit.

For the [Property Value Report](#) and [Missing Property Report](#), this button will open a dialog, where you enter definitions for the properties that you want documented:



Enter a definition on the form *typename;property* like

TCustomLabel;Caption

In this case, PAL will document the value of every TCustomLabel Caption property (or descendant of a TCustomLabel, like TLabel) that is found in a DFM file. There are some extensions for the Property Value Report **only**, for example if you enter:

;SQL

every property with the name "SQL" will be documented regardless of its type.

If you enter

TButton;

every property of TButton will be documented.

For the Complexity Report, there are two additional numeric input fields:

Max. length identifier (default = 15)

Max. rated items (default = 10)

Those apply to the section in the [Complexity Report](#) listing long identifier names, and the sections presenting rating lists.

The checkbox "Details" for the [Totals Report](#) should be checked if you want PAL to generate detail info for this report.

There is also a checkbox for charts. Mark this checkbox if you want PAL to generate charts (image files) for the Complexity Report. This only applies for HTML reports and the default value is TRUE.

There is a checkbox "Also subfolders" for the [Third-party dependencies Report](#). Mark this checkbox if you want identifiers in subfolders to the main folder to be considered as "native" code, and not as third-party code. In PAL 5 where this option did not exist,

identifiers in subfolders were always treated as third-party code. Most often, third-party code is not located below the main folder, and this option may be of value in those situations.

There is a checkbox "Only warnings" for the [Uses Report](#). Mark this checkbox if you want PAL to only report those units that are either unnecessary or could be moved to the implementation uses list. Default is FALSE, which means that all units are written.

For the [Call Tree Report](#) and the [Reverse Call Tree Report](#), there is an option "All". Mark this checkbox if you want PAL to generate a complete tree, even including repeating branches when possible. The default value is FALSE. (This option was displayed as "Complete" in previous versions.)

For the Call Tree Report, Reverse Call Tree Report, and Exception Report, it is also possible to select the level of indentation. (Default 1).

In the [Control Alignment Report](#), there is an option to select whether left- or right alignment should be examined. (Default LEFT).

Hide empty report sections

Default = False

Check this option if you do not want to show empty report sections.

See also:

[Options menu](#)

[Properties - General](#)

[Properties - Format](#)

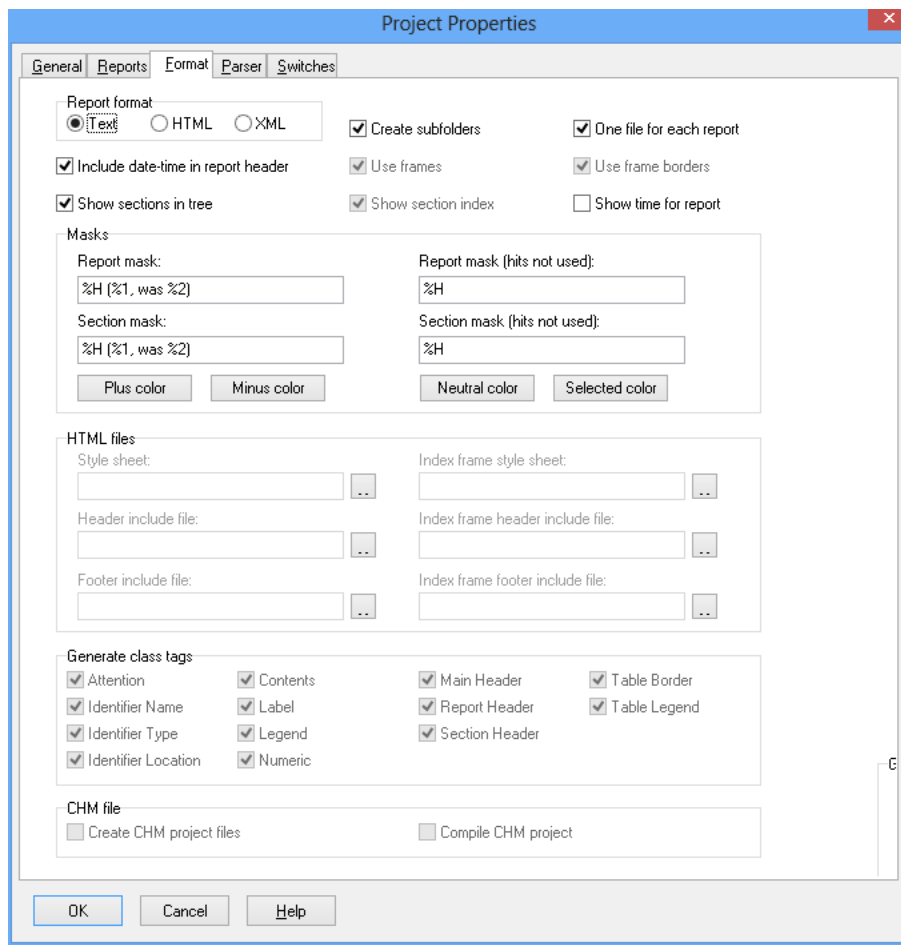
[Properties - Source](#)

[Properties - Parser](#)

[Properties - Switches](#)

15.6.3 Properties - Format

On this tab, you can select options for the report format.



Report format

Default = HTML

Select between reports in text, HTML or XML. The HTML format (default) is suitable for web publishing (Internet). There are also more ways to customize the layout and appearance of the reports. On the other hand, an advantage with text files is that they are somewhat faster to load. XML is the preferred format for transferring PAL data to for instance a database. However, the XML format is rather spacious and those files can become very large and the time to load them for viewing is often long.

You can view the HTML and XML reports in PAL's main window, or in your browser (e.g. MS Internet Explorer, Netscape). PAL uses Microsoft's Shell and Doc Extensions API. Thus, when you view the HTML reports in PAL, you are actually viewing them in the same way as if you view them in MS Internet Explorer. Even the context sensitive menus are available (via mouse right-click).

To load the HTML reports in your browser, select **MyProj_Index.htm** if frames are used, where MyProj is the name of the parsed main file. If frames are not used, load **Contents.htm** when several report files are created, or **MyProj.htm** if all reports belong to a single file.

Create subfolders

Default = Yes

If you select this option, PAL will create a separate subfolder beneath the report folder for each project that is analyzed. E.g. if a Delphi project Gcache.dpr is analyzed and the report folder is set to D:\Data\PALReports a folder D:\Data\PALReports\GCache is created. The generated report files are stored in that folder.

One file for each report

Default = Yes

Select this option if you want a single report file (text or HTML) for each report. The reports are named Status.htm, Warnings.htm etc.

Include date-time in report header

Default = Yes

Mark this checkbox if you want a text string with the current date and time included at the top of each report.

Use frames

Default = Yes

Check this option to generate HTML frame pages. This option is only available when HTML reports are selected. The option "One file for each report" (see above) must also be selected. If frames are used, a left frame page will keep an index to the different reports.

Use frame borders

Default = Yes

Check this option to generate frame borders. This option is only available when HTML reports and frames are selected.

Show sections in tree

Default = Yes

Check this option to include report sections in the report tree in the main window. This will allow you to jump directly to a report section by clicking on the appropriate line in the report tree.

Show section index

Default = Yes

For reports that are divided in several sections, like the Warnings Report, you may select this option to let PAL generate an index at the top of each report. This option is only available when HTML reports are selected.

Show time for report

Default = No

Check this option to include information about the time needed to generate a report. This information is written at the bottom of each report. Time usage for each section in the report is also written.

Report mask

Specify the format mask to use when displaying a report caption. The mask will only be used when section counters are displayed, otherwise only the plain caption is displayed.

For example, the mask:

%C-%H (%1, was %2)

will for the Optimization Report display:

"OPTI-Optimization Report (3, was 5)"

Those placeholders can be used:

%C report code, like "WARN1" etc, see report topics

%H report caption

%1 counter (reported items)

%2 old counter

The default mask is:

%H (%1, was %2)

Report mask (no hits)

Specify the format mask to use when displaying a report caption. The mask will only be used when section counters are not displayed.

For example, the mask:

%C-%H

will for the Call Tree Report display:

"CTRE-Call Tree"

Those placeholders can be used:

%C report code

%H report caption

The default mask is:

%C-%H

Report section mask

Specify the format mask to use when displaying a report section caption. The mask will be used when section counters are displayed.

For example, the mask:

%C-%H (%1, was %2)

will for one section in the Optimization Report, display:

"OPTI4-Virtual methods (procedures/functions) that are not overridden (3, was 5)"

Those placeholders can be used:

%C report code, like "WARN1" etc, see report topics
%H report caption
%1 counter (reported items)
%2 old counter

The default mask is:

%H (%1, was %2)

Report section mask (no hits)

Specify the format mask to use when displaying a report section caption. The mask will be used when section counters are not displayed.

For example, the mask:

%C-%H

will for one section in the Object-oriented Metrics Report, display:

"OOME1-Weighted methods per class (WMC)"

Those placeholders can be used:

%C report code
%H report caption

The default mask is:

%H

Plus color

Press this button to select the color used for "plus" captions, e.g. when the number of "new" warnings is greater than the number of "old" warnings, like in this caption:

"(5) Identifiers never used (was 3)"

Minus color

Press this button to select the color used for "minus" captions, e.g. when the number of "new" warnings is greater than the number of "old" warnings, like in this caption:

"(3) Identifiers never used (was 5)"

Neutral color

Press this button to select the color used for "plus" captions, e.g. when the number of "new" warnings is the same as number of "old" warnings, like in this caption:

"(5) Identifiers never used (was 5)"

Selected color

Press this button to select the color used for the background of the selected item in the report tree. You should make sure that this color fits together with the colors selected for plus, minus and neutral (see above),

Style sheet

Default = (blank)

Pascal Analyzer includes a style sheet block with default settings in every HTML report. If you want to use your own customized style sheet, enter the absolute or relative link to it in

this input field. E.g. if the HTML reports are created in the D:\Data\PALReports\GCache folder, and the style sheet file is in D:\Data\PALReports\PALstyles.css, you should enter "../PALstyles.css".

If this field is blank, PAL will include the style sheet block, otherwise it will just include a HTML link (<LINK REL) to the external style sheet file.

This is the style sheet block that is inserted at the top of each HTML file. You may use it as a template when creating your own style sheet.

```
<STYLE TYPE="text/css">
<!--
BODY{BACKGROUND: #FFFFC4; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt}
H1{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 14pt}
TD{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt}
.Attention{COLOR: #FF0000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt}
.IdN{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt; FONT-WEIGHT: BOLD}
.IdT{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt; FONT-STYLE: ITALIC}
.IdL{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt}
.Contents{COLOR: #0000FF; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt; FONT-WEIGHT: BOLD}
.Label{COLOR: #0000FF; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt; FONT-WEIGHT: BOLD}
.Legend{COLOR: #0000FF; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt}
.Numeric{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 8pt}
.MainHeader{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt; FONT-WEIGHT: BOLD}
.ReportHeader{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 14pt; FONT-WEIGHT:
BOLD}
.SectionHeader{COLOR: #000000; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 12pt; FONT-WEIGHT:
BOLD}
.TableBorder{BORDER-BOTTOM: #000000 1px solid; BORDER-LEFT: #000000 1px solid; BORDER-RIGHT:
#000000 1px solid; BORDER-TOP: #000000 1px solid}
.TableLegend{COLOR: #008080; FONT-FAMILY: Verdana,Arial; FONT-SIZE: 10pt; FONT-WEIGHT: BOLD}
-->
</STYLE>
```

The style sheet includes styles for the following classes:

Attention	Text that points out anomalies, e.g. unused unit in Uses Report
IdN	Identifier names, e.g. DrawCanvas
IdT	Identifier type, e.g. Integer
IdL	Identifier location, e.g. CodUtil(321)
Contents	Text in contents list
Label	Descriptive text like "Analyzed by" in the Status Report
Legend	Descriptive text for an entire table, like "Abbreviations in Complexity
Report	
Numeric	Numeric, e.g. "350", like in Totals Report
ReportHeader	Report header
SectionHeader	Section header

MainHeader Header in sections

TableBorder Table borders

TableLegend Legends in tables

Header include file

Default = (blank)

If you enter an URL in this field, PAL will include the file at the top of each HTML report. Use this to include common header information on each page, like company name, logo etc.

Footer include file

Default = (blank)

If you enter an URL in this field, PAL will include the file at the bottom of each HTML report. Use this for common footer information.

Index frame style sheet

Default = (blank)

Pascal Analyzer includes a style sheet block with default settings in the index frame HTML page. If you want to use your own style sheet, enter the absolute or relative link to it in this input field. E.g. if the HTML reports are created in the D:\Data\PALReports\GCache folder, and the style sheet file is in D:\Data\PALReports\PALstyles.css, you should enter "../PALstyles.css".

If this field is blank, PAL will include the style sheet block, otherwise it will just include a HTML link (<LINK REL) to the external style sheet file.

Index frame header include file

Default = (blank)

If you enter an URL in this field, PAL will include the file at the top of the index frame page for HTML reports. This option is only available when HTML frames are selected. Use this to include common header information on the index frame page, like company name, logo etc.

Index frame footer include file

Default = (blank)

If you enter an URL in this field, PAL will include the file at the bottom of the index frame page for HTML reports. This option is only available when HTML frames are selected. Use this for common footer information.

Generate class tags

For every checkbox that you select, PAL will generate extra HTML code to select the corresponding font class, as defined in the style sheet (see above). This will increase the size of the HTML files considerably, so just select those font classes that you want to customize. All options are by default selected.

For instance, to activate a special font for all identifier names, check "Identifier Name".

Then edit the font class definition ("IdN") in your style sheet file. Make sure the style sheet is referenced in the "Style sheet URL" field.

Create CHM project files

Mark this checkbox if you want Pascal Analyzer to create CHM project files, in addition to the generated HTML pages. This is especially handy if you have generated HTML files and want full-text search. This option is only enabled if you have selected HTML as the report format on the top of the tab page.

The CHM project files including the resulting CHM file will be written to the output directory. You can either work with the project files in HTML Help Workshop, or compile them directly from within Pascal Analyzer.

Default = No

Compile CHM project

Mark this checkbox if you want Pascal Analyzer to compile the CHM project, and produce a compressed CHM file. You must first download and install "HTML Help Workshop version 1.3" from Microsoft.

This option is only enabled if you have selected HTML as the report format on the top of the tab page, and if you have marked the "Create CHM project files" option (see above).

When compiling, a log file will also be created in the output directory. It will have the name format **<ProjectName>.log**. Check this file if errors occur.

Please note that the compilation for a help file can take a while. You can always create a CHM project file, and then later decide to compile it manually with the help compiler.

Default = No

See also:

[Options menu](#)

[Properties - General](#)

[Properties - Parser](#)

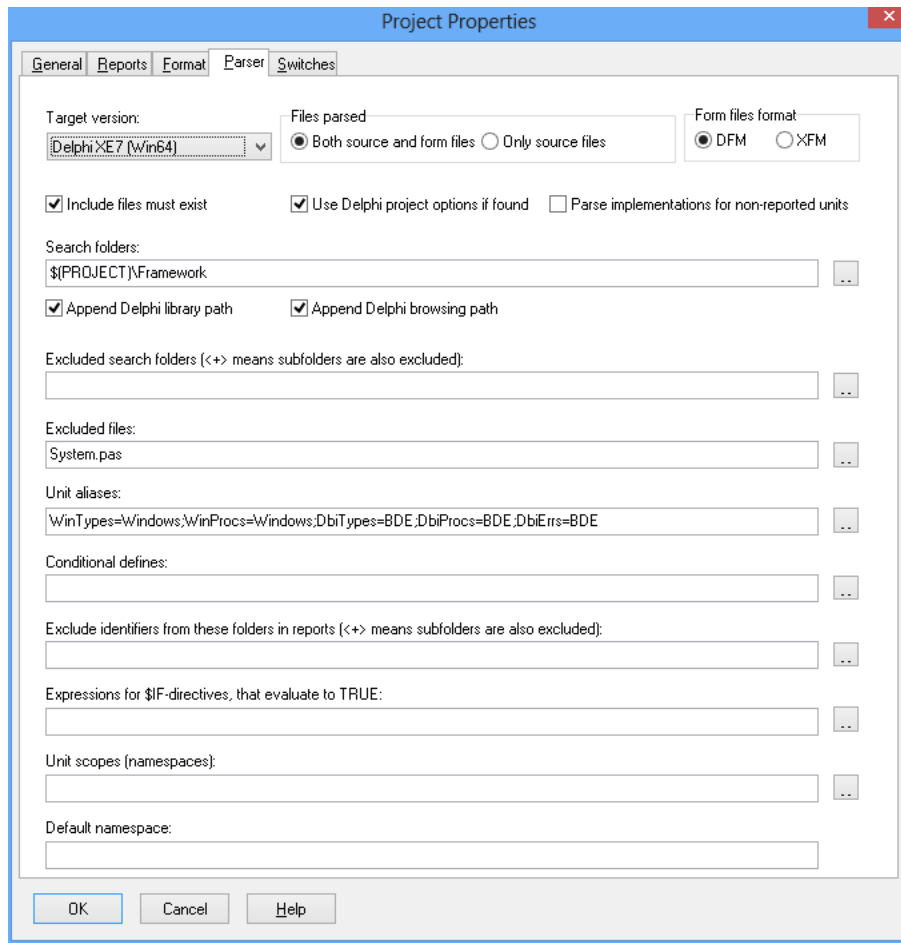
[Properties - Reports](#)

[Properties - Source](#)

[Properties - Switches](#)

15.6.4 Properties - Parser

The Parser tab page is not available for a multi-project.



Target version

Select the compiler version targeted for the current analysis. Note that, although PAL scans the source files in the same way as the compiler, it does not detect every syntax error. Make sure that the source code compiles correctly for the specified target, otherwise the results output by PAL may be incorrect.

PAL supports code written for these compilers/versions:

- Borland Pascal 7 (or earlier)
- Delphi 1
- Delphi 2
- Delphi 3
- Delphi 4
- Delphi 5
- Delphi 6
- Delphi 7
- Delphi 8 for .NET
- Delphi 2005 for Win32
- Delphi 2005 for .NET

- Delphi 2006 for Win32 (also Turbo Delphi for Win32)
- Delphi 2006 for .NET (also Turbo Delphi for .NET)
- Delphi 2007 for Win32
- Delphi 2007 for .NET
- Delphi 2009 for Win32
- Delphi 2010 for Win32

- Delphi XE for Win32

- Delphi XE2 for Win32
- Delphi XE2 for Win64
- Delphi XE2 for OSX

- Delphi XE3 for Win32
- Delphi XE3 for Win64
- Delphi XE3 for OSX

- Delphi XE4 for Win32
- Delphi XE4 for Win64
- Delphi XE4 for OSX
- Delphi XE4 for iOS Device
- Delphi XE4 for iOS Simulator

- Delphi XE5 for Win32
- Delphi XE5 for Win64
- Delphi XE5 for OSX
- Delphi XE5 for iOS Device
- Delphi XE5 for iOS Simulator
- Delphi XE5 for Android

- Delphi XE6 for Win32
- Delphi XE6 for Win64
- Delphi XE6 for OSX
- Delphi XE6 for iOS Device
- Delphi XE6 for iOS Simulator
- Delphi XE6 for Android

- Delphi XE7 for Win32
- Delphi XE7 for Win64
- Delphi XE7 for OSX
- Delphi XE7 for iOS Device
- Delphi XE7 for iOS Simulator
- Delphi XE7 for Android

- Delphi XE8 for Win32
- Delphi XE8 for Win64
- Delphi XE8 for OSX
- Delphi XE8 for iOS Device 32-bits
- Delphi XE8 for iOS Device 64-bits
- Delphi XE8 for iOS Simulator
- Delphi XE8 for Android

- Delphi 10 for Win32

- Delphi 10 for Win64
- Delphi 10 for OSX
- Delphi 10 for iOS Device 32-bits
- Delphi 10 for iOS Device 64-bits
- Delphi 10 for iOS Simulator
- Delphi 10 for Android

- Delphi 10.1 for Win32
- Delphi 10.1 for Win64
- Delphi 10.1 for OSX
- Delphi 10.1 for iOS Device 32-bits
- Delphi 10.1 for iOS Device 64-bits
- Delphi 10.1 for iOS Simulator
- Delphi 10.1 for Android

- Delphi 10.2 for Win32
- Delphi 10.2 for Win64
- Delphi 10.2 for OSX
- Delphi 10.2 for iOS Device 32-bits
- Delphi 10.2 for iOS Device 64-bits
- Delphi 10.2 for iOS Simulator
- Delphi 10.2 for Android
- Delphi 10.2 for Linux 64-bits

- Delphi 10.3 for Win32
- Delphi 10.3 for Win64
- Delphi 10.3 for OSX 32-bits
- Delphi 10.3 for OSX 64-bits
- Delphi 10.3 for iOS Device 32-bits
- Delphi 10.3 for iOS Device 64-bits
- Delphi 10.3 for iOS Simulator
- Delphi 10.3 for Android 32-bits
- Delphi 10.3 for Android 64-bits
- Delphi 10.3 for Linux 64-bits

- Delphi 10.4 for Win32
- Delphi 10.4 for Win64
- Delphi 10.4 for OSX 32-bits
- Delphi 10.4 for OSX 64-bits
- Delphi 10.4 for iOS Device 32-bits
- Delphi 10.4 for iOS Device 64-bits
- Delphi 10.4 for iOS Simulator
- Delphi 10.4 for Android 32-bits
- Delphi 10.4 for Android 64-bits
- Delphi 10.4 for Linux 64-bits

- Delphi 11 for Win32
- Delphi 11 for Win64
- Delphi 11 for OSX 32-bits
- Delphi 11 for OSX 64-bits
- Delphi 11 for iOS Device 32-bits
- Delphi 11 for iOS Device 64-bits
- Delphi 11 for iOS Simulator

- Delphi 11 for Android 32-bits
- Delphi 11 for Android 64-bits
- Delphi 11 for Linux 64-bits
- Delphi 11 for OSX ARM 64-bits

- Delphi 12 for Win32
- Delphi 12 for Win64
- Delphi 12 for OSX 32-bits
- Delphi 12 for OSX 64-bits
- Delphi 12 for iOS Device 32-bits
- Delphi 12 for iOS Device 64-bits
- Delphi 12 for iOS Simulator
- Delphi 12 for Android 32-bits
- Delphi 12 for Android 64-bits
- Delphi 12 for Linux 64-bits
- Delphi 12 for OSX ARM 64-bits

Default = Delphi 12 for Win64

PAL may also work with earlier versions of Turbo Pascal for DOS and Windows (prior to Borland Pascal 7), but this has not been validated and consequently is not guaranteed. In this case, select Borland Pascal 7 for best results.

For targets Delphi 1 and upward, PAL is able to load identifiers from the *System* unit in Delphi's runtime library. This makes PAL aware of subprograms like *Inc* and *FreeMem* and the TObject root class, resulting in reports that are more accurate and complete.

Files parsed

Default = Both source and form files (DFM/NFM/XFM-files)

Select an option:

Both source and form files (DFM/FMX/NFM/XFM-files)

This is the default option. If PAL finds a form file, it will be examined together with the corresponding PAS-file.

Only source files

No form files will be examined.

It is recommended to let PAL also find and parse form files, so normally keep this option selected.

Include files must exist

Default = Yes

Mark this checkbox if a missing include file should trigger an error and stop the analysis. Keep this option selected if possible, since a vital missing include file could generate incorrect results.

Use Delphi project options if found

Default = Yes

When a Delphi compiler and a DPR file is selected, PAL tries to load the corresponding project options file. If successful, these options are used. For search paths, unit aliases and defines, the options are merged with the options you select. This makes it possible to instance, in PAL to provide the path to the VCL source files.

If a Delphi project file (DPR file) is parsed, search paths following the *in* keyword are automatically followed.

The [Status Report](#) shows which search paths that are used for the particular analysis.

Parse implementation for non-reported units

Default = No

It is not necessary to parse implementation parts of units that are not reported.

Search folders

Select the drives and folders where PAL will search for source files. The folder containing the primary source file is automatically searched, and there is no need to include this folder.

Unlike the compiler, PAL does not require that all source code is available. However, it is often best to make as much source code visible to PAL as possible.

Enter search folders, in priority order, separated with a semicolon e. g.:

```
c:\source\myunits;c:\source\generic;c:\source\proj1
```

Alternatively, press the ellipsis button to select the folders in a selection dialog.

You may also enter relative paths, like "..\..\generic".

If a Delphi project or package file (DPR, or DPK file) is parsed, search paths following the *in* keyword are automatically followed.

Because the parser looks in the directories according to the order specified, it is wise to put more frequently used directories first in the list.

User-defined environmental variables, like "\$(UTILS)", set in the Delphi IDE or in the System settings in the Control Panel, may also be used.

You can also use a relative path. The path is then relative to the folder where the project file (PAP-file) is located.

Append Delphi Library Path

Default = Yes

Mark this checkbox if you want PAL to look for modules also in these folders. The Delphi Library Path is set in the Delphi IDE under Tools|Environment Options and the Library tab page.

Append Delphi browsing path

Default = Yes

Mark this checkbox if you want PAL to look for modules also in these folders. The Delphi browsing path is set in the Delphi IDE under Tools|Environment Options and the Library tab page.

Excluded search folders

Source code from these folders will NOT be parsed. Enter excluded folders separated with a semicolon e. g.:

C:\source\notused;c:\source\3rdparty

Alternatively, press the ellipsis button to select the folders in a dialog box.

It is possible to select that an exclude folder should also apply to its subfolders. In this way it is possible to exclude "C:\Program Files\Borland\Delphi7\Source" and all the subfolders. When subfolders are excluded, the folder name is suffixed with "<+>".

User-defined environmental variables, like "\$\{UTILS}", set in the Delphi IDE or in the System settings in the Control Panel, may also be used.

You can also use a relative path. The path is then relative to the folder where the project file (PAP-file) is located.

Excluded files

Source code from these files will NOT be parsed. Enter excluded files separated with a semicolon, e. g.:

myfile.pas;obsolete.pas

Alternatively, press the ellipsis button to select the files in a dialog box.

Unit aliases

Default for Win32 versions:

`WinTypes=Windows;WinProcs=Windows;DbiTypes=BDE;DbiProcs=BDE;DbiErrs=BDE`

Default for .NET versions:

`WinTypes=Borland.Vcl.Windows;WinProcs=Borland.Vcl.Windows;DbiTypes=BDE;DbiProcs=BDE;DbiErrs=BDE`

See the Delphi documentation for an explanation of unit aliases.

Conditional defines

Specify conditional compilation directives that are valid for the current analysis. PAL parses source code just like the compiler, and must treat conditional directives, meaning that it will ignore sections of not activated code. You may also include defines within the source code. Defines set in the Delphi project file will also be used, if found.

Enter conditional defines, separated with a semicolon e. g.:

Final;Special

Alternatively, press the ellipsis button to enter the defines in a dialog box.

PAL even initializes predefined defines just as the compiler does (e g 'WIN32'). It also keeps track of the state of all compiler directives so that directives like {\$IFOPT} will function correctly.

The conditional compilation directive `_PEGANZA_` is always defined.

Exclude identifiers from these folders in reports (but always report main file):

In addition, it is possible to select folders that should be considered in the generated reports. Identifiers declared in source code from these folders will not appear in the reports. The [Totals Report](#) and [Third-party dependencies Report](#) will however include even these identifiers.

Enter excluded folders separated with a semicolon e. g.:

c:\source\myunits;c:\source\generic

Alternatively, press the ellipsis button to select the folders in a selection dialog.

It is possible to select that an exclude folder should also apply to its subfolders. In this way it is possible to exclude "C:\Program Files\Borland\Delphi7\Source" and all the subfolders. When subfolders are excluded, the folder name is suffixed with "<+>".

User-defined environmental variables set in the Delphi IDE, may also be used.

You can also use a relative path. The path is then relative to the folder where the project file (PAP-file) is located.

Expressions for \$IF-directives, that evaluate to TRUE

In Delphi 6, the new \$IF-directive was introduced. The \$IF-directive is followed by an expression, that evaluates to TRUE or FALSE. If you use \$IF-directives, you must supply all expressions that evaluate to TRUE, because PAL cannot always determine the value of an expression. Enter the expressions separated with semicolons, like:

RTLVersion > 14;Declared(Windows)

Please observe that you do not need to include Defined-directives like "Defined(MSWINDOWS)", because PAL manages to evaluate those directives.

When PAL's parser finds a \$IF-directive in code, it will try to evaluate it. If it is an

expression that you have supplied, it will be evaluated to TRUE, otherwise it will be evaluated as FALSE. Directives that are evaluated as FALSE, imply that the corresponding code is not activated.

Unit scopes (namespaces)

This field is only enabled (and relevant) if the compiler is set to Delphi 8 or higher compilers. Press the ellipsis button to select the namespaces.

Default namespace

This field is only enabled (and relevant) if the compiler is set to Delphi 8 or higher compilers.

See also:

[Options menu](#)

[Properties - General](#)

[Properties - Format](#)

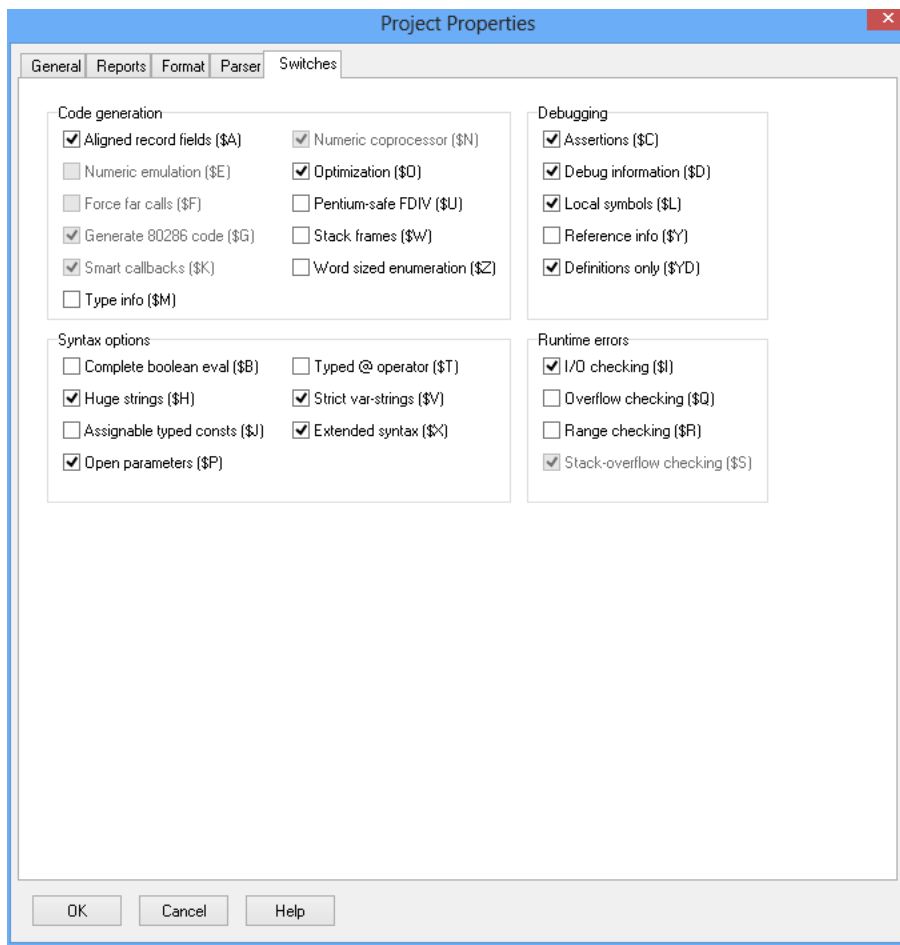
[Properties - Reports](#)

[Properties - Source](#)

[Properties - Switches](#)

15.6.5 Properties - Switches

The Switches tab is not available for a multi-project.



On this tab, you can select which compiler switches that are active for the current analysis. This is important for how PAL evaluates \$IFOPT directives. For instance, if \$R+ (range check) is activated, a directive {\$IFOPT R+} will evaluate to TRUE, and the following code is activated.

Not all compiler switches are included on this page. Only those that can be used with \$IFOPT are listed.

A better method than setting these switches is to maintain a **common include file** used by all your modules. The purpose of the file is to set compiler directives and conditional defines that dictate the compilation (and parsing by PAL). The include file directive \$I is used to embed the contents of the include file into the source file.

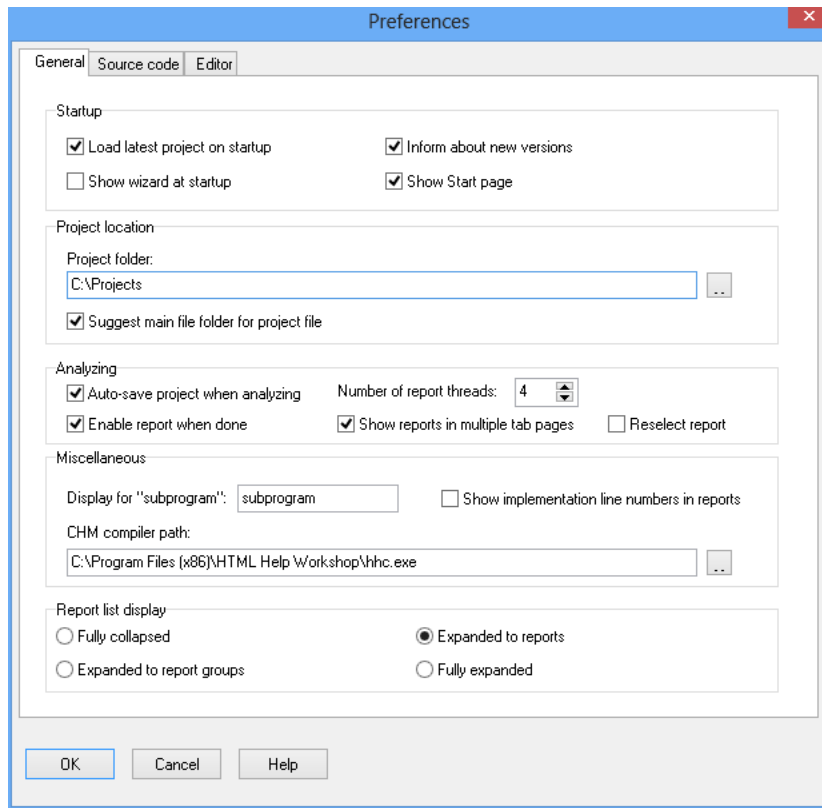
See also:

[Options menu](#)
[Properties - General](#)
[Properties - Format](#)
[Properties - Parser](#)
[Properties - Reports](#)

[Properties - Source](#)

15.6.6 Preferences - General

On the first tab page (General) there are some general settings:



Load latest project at startup

Default = Yes

Mark this checkbox if you want PAL to load the latest project at startup. Please observe that this option and the "Show wizard at startup" option cannot both be selected.

Inform about new versions

Default = Yes

Mark this checkbox if you want to be notified when new versions of PAL are available. To protect your privacy, no identifying information is transferred from your computer when PAL checks for new versions. It is just a simple HTTP request to the Peganza web site to retrieve the latest product version number. If you do not want to check automatically by activating this option, select the "Check for newer version" menu item in the [Help](#) menu to display this information.

Show wizard at startup

Default = Yes

Mark this checkbox if you want to activate the wizard when PAL is started. Please observe that this option and the "Load project at startup" option cannot both be selected.

Show Start page

Default = Yes

Mark this checkbox if you want to show the start page with current news from Peganza.

Project folder

This is the folder that PAL suggests for new project files. Default is "C:\Documents and Settings\<acc>\My Documents\Pascal Analyzer\Projects".

Suggest main file folder for project file

Default = No

Mark this checkbox if you want PAL to suggest the same folder as where the main file is located, when saving a new project file. This overrides the setting for "Project folder" (see above).

Auto-save project when analyzing

Default = Yes

Mark this checkbox if you want to automatically save the project file when the analysis is started.

Number of report threads

Default = 2

This is the number of report threads that will be run in parallel, when reports are built. For example, if you have a dual-core processor, the suitable value is probably two. For a single-core processor, set it to one instead.

Enable report when done

Default = Yes

Mark this checkbox if you want reports to be accessible immediately when they have been generated, without waiting for all reports to be generated. Otherwise, the behaviour will be as in pre-PAL8, that reports are not available until all work is done.

Show reports in multiple tab pages

Default = Yes

Mark this checkbox if you want reports to open in new tab pages. Otherwise, the

behaviour will be as in pre-PAL8, that only one report is visible at any time.

Reselect report

Default = Yes

Mark this checkbox if you want the active report to be reselected automatically when analysis is done.

Display for "subprogram"

Default = subprogram

If you want to use another string than "subprogram" when displaying report and section texts, you can enter the string here, for example "function". Enter it as the singular term with small letters.

Show implementation line numbers in reports

Default = No

This option determines if the line number where a subprogram is declared is displayed in the reports. If set to Yes, instead implementation line number is displayed. It has meaning if you double-click on the line to jump to the source code. Either it will then take you to the declaration or the implementation line. Often you would probably prefer to reach the implementation. If so, then set this option to Yes.

CHM compiler path

Edit the path to the hhc.exe help compiler (Microsoft HTML Help Workshop) if needed.

Default = <PROGRAMFILES>\HTML Help Workshop\hhc.exe

Report list display

Default = Expanded to reports

Select how the report list will be displayed initially:

- Fully collapsed
- Expanded to report groups
- Expanded to reports
- Fully expanded

See also:

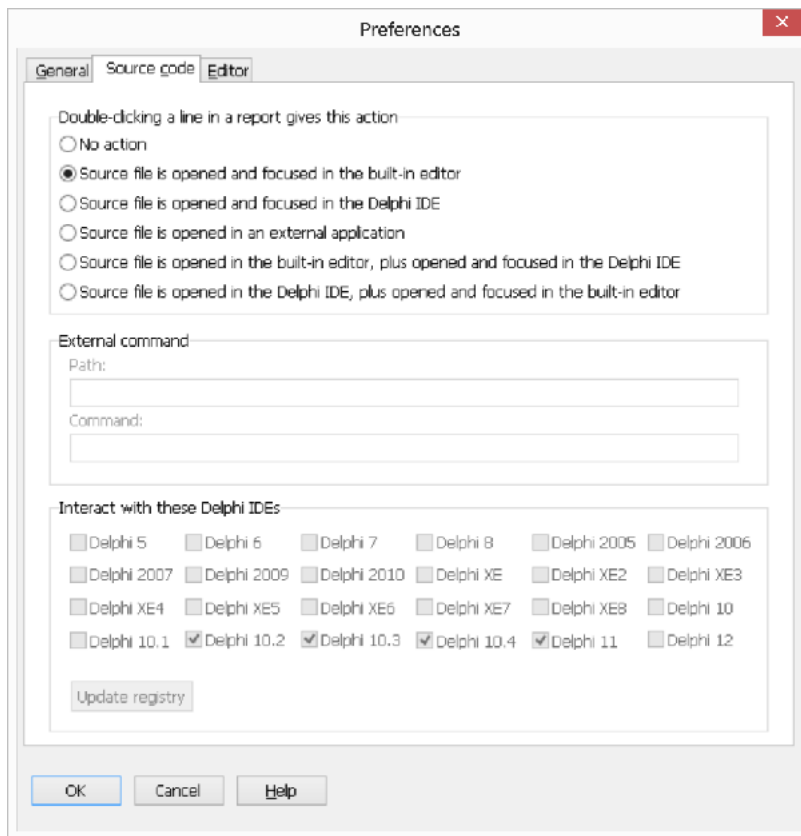
[Options menu](#)

[Preferences - Source Code](#)

[Preferences - Editor](#)

15.6.7 Preferences - Source Code

On the Source code tab page, select the action taken when double-clicking on a line in the report viewer.



Double-clicking a line in a report gives this action

These options are available:

No action

Nothing happens when double-clicking.

Source file is opened and focused in the built-in editor (default)

If you want to quickly view the source code line, this is a convenient option. The file is opened in the built-in editor. plus a button that closes the window.

Source file is opened and focused in the Delphi IDE

If Delphi is running, control is transferred to the source file in the code editor. This option works with Delphi from version 5.

Source file is opened in an external application

If this option is selected, you must enter a complete path to the application, and a command-line.

Source file is opened in the built-in editor, plus opened and focused in the Delphi IDE

This is a combination of the options to open in the built-in editor and in the Delphi IDE.

The Delphi IDE will be focused

Source file is opened in the Delphi IDE, plus opened and focused in the built-in editor.

This is a combination of the options to open in the Delphi IDE and in the built-in editor. The editor will be focused

Path

Enter a complete path to the application, e.g "C:\WINDOWS\notepad". This option is only relevant when "Source file is opened in an external application" has been selected.

Command

Enter a command-line that is supplied as a parameter. In the command-line, use "%1" for the source file path and "%2" for the row number. These strings will be substituted for the real values when the call is done.

This option is only relevant when "Source file is opened in an external application" has been selected.

For example, when configuring Notepad to open the source file, enter "C:\WINDOWS\notepad" as path, and "%1" for command.

Interact with Delphi IDEs

The plugins PALWIZ*.DLL enable Delphi to show the relevant code module when double-clicking on a report line in Pascal Analyzer.

For all installed versions of Delphi 5, 6 and 7, the wizard will be registered under the following registry key:

HKEY_CURRENT_USER\Software\Borland\Delphi\x.0\Experts
(exchange "x" for the version number).

For other versions:

Delphi 8

HKEY_CURRENT_USER\Software\Borland\BDS\2.0\Experts

Delphi 2005

HKEY_CURRENT_USER\Software\Borland\BDS\3.0\Experts

Delphi 2006

HKEY_CURRENT_USER\Software\Borland\BDS\4.0\Experts

Delphi 2007

HKEY_CURRENT_USER\Software\Borland\BDS\5.0\Experts

Delphi 2009

HKEY_CURRENT_USER\Software\CodeGear\BDS\6.0\Experts

Delphi 2010

HKEY_CURRENT_USER\Software\CodeGear\BDS\7.0\Experts

Delphi XE

HKEY_CURRENT_USER\Software\Embarcadero\BDS\8.0\Experts

Delphi XE2

HKEY_CURRENT_USER\Software\Embarcadero\BDS\9.0\Experts

Delphi XE3

HKEY_CURRENT_USER\Software\Embarcadero\BDS\10.0\Experts

Delphi XE4

HKEY_CURRENT_USER\Software\Embarcadero\BDS\11.0\Experts

Delphi XE5

HKEY_CURRENT_USER\Software\Embarcadero\BDS\12.0\Experts

Delphi XE6

HKEY_CURRENT_USER\Software\Embarcadero\BDS\14.0\Experts

Delphi XE7

HKEY_CURRENT_USER\Software\Embarcadero\BDS\15.0\Experts

Delphi XE8

HKEY_CURRENT_USER\Software\Embarcadero\BDS\16.0\Experts

Delphi 10

HKEY_CURRENT_USER\Software\Embarcadero\BDS\17.0\Experts

Delphi 10.1

HKEY_CURRENT_USER\Software\Embarcadero\BDS\18.0\Experts

Delphi 10.2

HKEY_CURRENT_USER\Software\Embarcadero\BDS\19.0\Experts

Delphi 10.3

HKEY_CURRENT_USER\Software\Embarcadero\BDS\20.0\Experts

Delphi 10.4

HKEY_CURRENT_USER\Software\Embarcadero\BDS\21.0\Experts

Delphi 11

HKEY_CURRENT_USER\Software\Embarcadero\BDS\22.0\Experts

Delphi 12

HKEY_CURRENT_USER\Software\Embarcadero\BDS\23.0\Experts

If you install any Delphi version after Pascal Analyzer has been installed, the registry settings have to be updated. In this case, enter the [Preferences dialog](#), mark check boxes and click the button **Update registry**.

After clicking **Update registry**, the changes will take place the next time you start the Delphi IDE. Of course, because the registry is modified, you must run under an account that is allowed to do this, when clicking this button.

When updating the registry, the changes are immediately applied, regardless if you leave the dialog by pressing the **OK** or **Cancel** button.

Buffered viewer

Default = Yes

Select this checkbox to load source files in a more efficient manner. The drawback is that the source code file is locked by the viewer, so it cannot be edited by another application. For example, if the file is loaded in the viewer you cannot at the same time edit and save it in the Delphi IDE. If you want to be able to do so, you must deselect this option. This may give longer loading times especially for large source code files.

Show line numbers

Default = Yes

Select this checkbox if you want line numbers to appear in the left margin of the source viewer window.

Expand tab characters

Default = 8

Select the number of characters that a TAB character should be expanded to, when viewing source code.

See also:

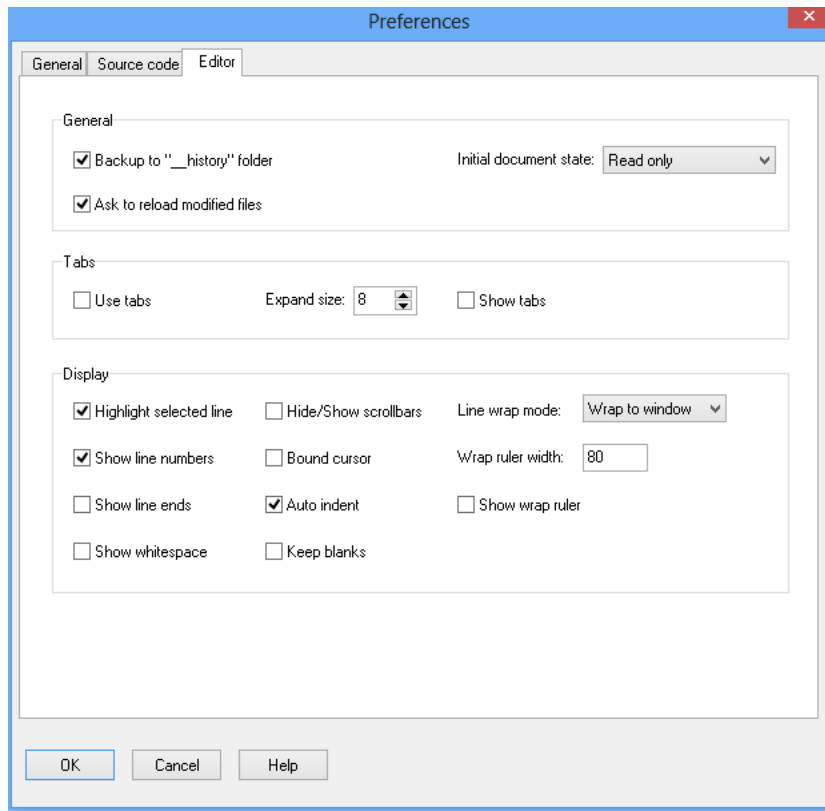
[Options menu](#)

[Preferences - General](#)

[Preferences - Editor](#)

15.6.8 Preferences - Editor

On the Editor tab page, select settings for the source code editor tab pages.



Backup to "__history" folder

Default = Yes

If you select this option, any changed source files will be backuped to a "__history" folder below the current folder. This works in a similar way like in the RAD Studio IDE.

Initial document state

Default = Read only

Choose between different initial states for the document (source code file):

- Read only. source file will initially be read only
- Read only for files in foreign folder
- Writable, source file will be writable

It is at any moment possible to toggle between read-only and writeable state (see [File|Read Only](#)), regardless of the initial document state.

Ask to reload

Default = Yes

If "Yes", a message will appear when PAL detects that a file has been changed outside the application. You can then select to load the changed file into the editor window.

Use Tabs

Default = No

If "No", Tab characters will be converted to spaces, according to expand size (see next option).

Expand size

Default = 8

Choose the width of a Tab character.

Show tabs

Default = No

Highlight selected line

Default = Yes

Hide/Show scrollbars

Default = No

Line wrap mode

Default = Wrap to window

Choose between:

- no wrap, text will not be wrapped if it does not fit into window and horizontal scrollbars will be shown.
- wrap to ruler, line will be wrapped if it is longer than the right ruler
- wrap to window, line will be wrapped if it does not fit in the editor window

Show line numbers

Default = Yes

Selecting this option, lets line numbers appear in the left gutter.

Bound cursor

Default = No

Wrap ruler width

Default = 80

Show line ends

Default = No

Auto indent

Default = Yes

Show wrap ruler

Default = No

If turned on, shows a vertical ruler line for the right margin.

Show whitespace

Default = No

Keep blanks

Default = No

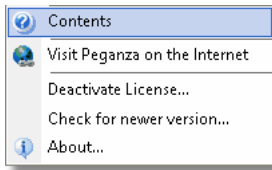
See also:

[Options menu](#)

[Preferences - General](#)

[Preferences - Source Code](#)

16 Help menu



Help|Contents

This command displays the help system.

Help|Visit Peganza on the Internet

This command opens Peganza's home page in your web browser. Read the latest information about Pascal Analyzer and our other products. You will also find information about updates and new versions.

Help|Deactivate License

If you need to move the installation to another computer, you can deactivate the license on the current computer. Then you will be able to activate the license on the new computer. The installation on the old computer will not longer work.

If you need additional activations, or help deactivating licenses on unused computers, just contact our support mail and we will help you.

See the license.txt file in the program directory for more information.

Help|Check for newer version

This command checks if there is a newer version of PAL available. This is the same kind of check that can be done automatically when starting the program, see [Preferences|General](#).

Help|About

Displays the "About" dialog box. It will show you among other things, the version number for the application.

Click "Refresh" to update the information about your support plan, and when it expires.

See also:

[Main menu](#)

Index

- / -

/A- 51
 /A+ 51
 /AUTO 48
 /CBP 51
 /CD1 51
 /CD101ANDROID 51
 /CD101IOSDEV 51
 /CD101IOSDEV64 51
 /CD101IOSSIM 51
 /CD101OSX 51
 /CD101W32 51
 /CD101W64 51
 /CD102ANDROID 51
 /CD102IOSDEV 51
 /CD102IOSDEV64 51
 /CD102IOSSIM 51
 /CD102LINUX64 51
 /CD102OSX 51
 /CD102W32 51
 /CD102W64 51
 /CD10ANDROID 51
 /CD10IOSDEV 51
 /CD10IOSDEV64 51
 /CD10IOSSIM 51
 /CD10N 51
 /CD10OSX 51
 /CD10W 51
 /CD10W32 51
 /CD10W64 51
 /CD11N 51
 /CD11W 51
 /CD12W 51
 /CD14W 51
 /CD2 51
 /CD3 51
 /CD4 51
 /CD5 51
 /CD6 51
 /CD7 51
 /CD8 51
 /CD9N 51
 /CD9W 51

/CDXE2OSX 51
 /CDXE2W32 51
 /CDXE2W64 51
 /CDXE3OSX 51
 /CDXE3W32 51
 /CDXE3W64 51
 /CDXE4IOSDEV 51
 /CDXE4IOSSIM 51
 /CDXE4OSX 51
 /CDXE4W32 51
 /CDXE4W64 51
 /CDXE5AND 51
 /CDXE5IOSDEV 51
 /CDXE5IOSSIM 51
 /CDXE5OSX 51
 /CDXE5W32 51
 /CDXE5W64 51
 /CDXE6AND 51
 /CDXE6IOSDEV 51
 /CDXE6IOSSIM 51
 /CDXE6OSX 51
 /CDXE6W32 51
 /CDXE6W64 51
 /CDXE7AND 51
 /CDXE7IOSDEV 51
 /CDXE7IOSSIM 51
 /CDXE7OSX 51
 /CDXE7W32 51
 /CDXE7W64 51
 /CDXE8AND 51
 /CDXE8IOSDEV 51
 /CDXE8IOSDEV64 51
 /CDXE8IOSSIM 51
 /CDXE8OSX 51
 /CDXE8W32 51
 /CDXE8W64 51
 /CDXEW 51
 /D 51
 /FA 51
 /FM 51
 /FR 51
 /I 51
 /L 51
 /P 51
 /Q 51
 /R 51
 /S 51
 /T 51

/X 51

- _ -

PEGANZA 183

- 6 -

64-bits 7, 31

- A -

about the product 203
 abstract methods 13
 activation 49
 Alt+Left 162
 Alt+Right 162
 ambiguous unit reference 67
 Analysis menu 164
 ANSI 58
 arrange 165
 arrange windows 162
 array properties
 set/referenced within methods 95
 assert 13
 auto indent 199
 auto-save 193

- B -

back 162
 backup 167
 backup to _history folder 199
 bad assignments 38
 bad pointer usage 67
 bad typecast 67
 binary files 126
 Bindings Report 130
 bound cursor 199
 Brief Cross-reference Report 135
 browser 58
 browsing path 183
 buffered viewer 196

- C -

Call Index Report 134
 Call Tree Report 132
 check for newer version 203
 CHM compiler path 193
 class
 constructors with bad names 109
 destructors with bad names 109
 multiple destructors 71
 Class Field Access Report 151
 class fields
 exposed by properties but do not start with "F" 109
 not declared in private section 109
 not declared in private/protected sections 109
 zero-initialized in constructor 100
 Class Hierarchy Report 150
 Class Index Report 149
 class list 162
 Class Summary Report 150
 class tags 176
 clone 146
 close 158
 Code Reduction Report 100
 code-blocks
 empty 71
 Command Line Options 48
 command-line 48
 command-line analyzer 51
 compile CHM project 176
 Complexity Report 119
 condition
 evaluates to constant value 71
 conditional defines 183
 Conditional Symbols Report 140
 constructors
 bad name 109
 no call to inherited 71
 Control Alignment Report 152
 Control Index Report 151
 Control Size Report 152
 Control Tab Order Report 153
 Control Warnings Report 154
 Convention Compliance Report 109
 copy 160
 create CHM project files 176

Cross-reference Report 135
 Ctrl+A 160
 Ctrl+Alt+C 58, 162
 Ctrl+Alt+Down 58
 Ctrl+Alt+I 58
 Ctrl+Alt+Left 58, 162
 Ctrl+Alt+Right 58, 162
 Ctrl+Alt+S 58, 162
 Ctrl+Alt+U 58
 Ctrl+Alt+Up 58
 Ctrl+C 160
 Ctrl+Enter 58
 Ctrl+F 161
 Ctrl+F4 158
 Ctrl+G 161
 Ctrl+H 161
 Ctrl+Left 58
 Ctrl+M 158
 Ctrl+MouseWheel 58
 Ctrl+N 158
 Ctrl+O 158
 Ctrl+P 158
 Ctrl+Right 58
 Ctrl+S 158
 Ctrl+V 160
 Ctrl+X 160
 Ctrl+Z 160
 customize "subprogram" 38
 cut 160

- D -

deactivation 203
 Delphi 10.2 Tokyo 15
 Delphi 10.3 Rio 15
 Delphi 10.4 Sydney 15
 Delphi 11 Alexandria 15
 Delphi IDE 49, 196
 deprecated directive 141
 destructors
 bad name 109
 more than one 71
 no call to inherited 71
 no override directive 71
 directives
 deprecated 141
 experimental 141
 inline 141

library 141
 platform 141
 Directives Report 141
 dual monitors 58
 Duplicate Identifiers Report 128

- E -

Edit menu 160
 editor 58
 editor options 199
 enable report when done 193
 encoding 126
 environment variables 183
 Events Report 157
 Exception Report 134
 Exit-statement
 dangerous 71
 expand size 199
 experimental directive 141

- F -

F11 165
 F12 165
 F3 161
 F8 162
 F9 162
 file date/time 126
 File menu 158
 file size 126
 finalization section 126
 find 161
 folders 57
 Form Report 156
 forward 162
 forward directive
 interface section 71
 frameborders 176
 frames 176
 function
 called only as procedures 100
 exposed by properties but do not start with "Get"
 109
 result not set 71

- G -

generics 13
go to line 161

- H -

Help menu 203
help report 143
hide empty report sections 172
hide/show scrollbars 199
highlight selected line 199
how to use 49
How to use PALCMD.EXE and PALCMD32.EXE 51

- I -

identifiers
 local only used at lower scope 100
 local possibly set and referenced once 100
 local possibly set more than once without referencing in-between 100
 local set and referenced once 100
 local set more than once without referencing in-between 100
 not used 100
 unsuitable name 109
 zero-initialized in constructor 100
Identifiers Report 127
include footer 176
include header 176
Inconsistent Case Report 115
index error 67
index frame
 footer include file 176
 header include file 176
 stylesheet 176
initial state 199
initialization section 126
inline directive 141
install in Delphi IDE 49
installation folders 57
interfaced class identifiers
 not used outside of unit 71
interfaced identifiers

 not used outside of unit 71
Introduction 7

- K -

keep blanks 199
Known limitations 13

- L -

labels
 inside for-loops 71
 inside repeat/while-loops 71
Lattix 137
library directive 141
library path 183
licensing model 15
limitations 13
line numbers 196
line wrap mode 199
Linux 15
Literal Strings/Numbers Report 129
long strings
 local initialized to empty strings 100
 local possibly initialized to empty strings 100

- M -

Main Window 58
map file report 145
marker for suppressed lines 167
Memory Report 92
methods
 called once from method in same class 100
 virtual but not overridden 95
missing files 126
Missing Property Report 156
Module Call Tree Report 143
module subprogram list 162
Module Totals Report 118
modules
 not added to DPR 137
 not needed in DPR 137
Modules Report 126
Most Called Report 132
multi-projects 7

- N -

namespaces 183
 navigation features 58
 new multi-project 158
 new project 158
 new versions 193
 next reference 162
 NextGen Readiness Report 116

- O -

object
 bad creation 71
 Object-oriented Metrics Report 122
 one file for each report 176
 open 158
 optimal uses list 137
 Optimization Report 95
 Options - General 167
 Options - Parser 183
 Options - Reports 172
 Options - Switches 191
 Options menu 165
 overloaded methods 13

- P -

PAL.EXE 49
 PAL.INI 57
 PAL32.EXE 49
 PALCMD.EXE 51
 PALCMD32.EXE 51
 PALOFF 167
 PALWIZ*.DLL 58, 196
 parameters
 out parameter read before set 71
 parse all 165
 paste 160
 platform directive 141
 pointer, bad usage 67
 Preferences - General 193
 prefix 116
 Prefix Report 116
 previous reference 162
 print 158

printer setup 158
 procedures
 exposed by properties but do not start with "Set"
 109
 project folder 193
 Properties - Format 176
 Properties - Source 196
 property access 67
 property value 155
 Property Value Report 155

- R -

record parameters
 unmodified and missing "const" 95
 redo 160
 relative path 38
 replace 161
 report list 58, 193
 report tree color 165
 report tree font 165
 reports 64
 reports in multiple tab pages 193
 reselect report 193
 Reverse Call Tree Report 133
 run 164

- S -

save 158
 save as 158
 search 161
 search folders 183
 Search menu 161
 searched strings report 144
 select all 160
 shadowed identifiers 109
 SHDOCVW.DLL 58
 show counter in section header 176
 show line ends 199
 show line numbers 199
 show report list 162
 show section index 176
 show source editor 162
 show tabs 199
 show toolbar 162
 show whitespace 199

- show wrap ruler 199
- Similarity Report 128
- soundex 128
- statements
 - empty 71
 - short-circuited 71
- status bar 58
- Status Report 66
- stay on top 165
- stop 164
- string parameters
 - unmodified and missing "const" 95
- Strong Warnings Report 67
- stylesheet 176
- subfolders 176
- subprogram 38
- subprogram calls itself 67
- Subprogram Index Report 130
- subprograms
 - empty parameter list 71
 - only called once 100
 - recursive 71
- support plan 203
- suppressed lines 167
- switches 191
- syntax anomalies 71

- T -

- tab characters 196
- target 183
- Third-party dependencies Report 131
- threads, number of 193
- to-do 142
- To-Do Report 142
- toolbar 58, 165
- Totals Report 117
- typecast, bad 67
- types
 - exception types that do not start with "E" 109
 - interface types that do not start with "I" 109
 - ordinary that do not start with "T" 109
 - pointer types that do not start with "P" 109

- U -

- undo 160

- Unicode 31, 58
- unique files 126
- unit aliases 183
- unit scopes 183
- unit usage 137
- unit, ambiguous reference 67
- unsupported types 116
- use tabs 199
- Used Outside Report 137
- Uses Report 137
- UTF-8 66

- V -

- value parameters
 - possibly set 71
 - set 71
- var parameters
 - never set 71
 - possibly never set 71
- variables
 - absolute 71
 - bad thread-local 71
 - bigger assigned to smaller 71
 - local possibly referenced before set 71
 - local referenced before set 71
 - local set not later used 71
 - never referenced 71
 - never set 71
 - possibly never referenced 71
 - possibly never set 71

- View menu 162
- viewer 58
- viewer color 165
- viewer font 165
- virtual methods
 - not overridden 95

- W -

- Warnings Report 71
- What's new in version 4? 45
- What's new in version 5? 42
- What's new in version 6? 38
- What's new in version 7? 35
- What's new in version 8? 31
- What's new in version 9? 15

wizard 158, 193

wrap ruler width 199