## Contents

1 Introduction .................................................. 16

2 Scanner / Tokenizer .......................................... 16
   2.1 Architecture .............................................. 17
      Input stream .............................................. 17
      Preprocessor ............................................. 19
         Conditional compilation (scandir.inc, scanner.pas) .... 19
         Compiler switches (scandir.inc, switches.pas) ....... 19
   2.2 Scanner interface ......................................... 19
      Routines ................................................. 19
      ReadToken ............................................... 19
      Variables ................................................. 20
         Token .................................................. 20
         Pattern ............................................... 20
   2.3 Assembler parser interface ............................... 20
      routines ............................................... 20
      AsmGetChar ............................................. 20

3 The tree ..................................................... 20
   3.1 Architecture ............................................. 20
   3.2 Tree types .............................................. 21
   3.3 Tree structure fields (tree.pas) ......................... 23
      Additional fields ........................................ 25
         AddN .................................................. 25
         CallParaN .......................................... 25
         AssignN ............................................. 25
         LoadN ............................................... 25
         CallN ............................................... 26
         addrn ............................................... 26
         OrdConstN .......................................... 26
         RealConstN ......................................... 26
## Symbol tables

4.1 Architecture ................................................................. 30
4.2 The Symbol table object .................................................... 30
4.3 Inserting symbols into a symbol table .................................... 32
4.4 Symbol table interface ....................................................... 32
   Routines ............................................................................ 32
   Search_a_Symtable ............................................................. 32
   GetSym ........................................................................... 32
   GlobalDef ........................................................................ 33
Variables .............................................................................. 33
   SrSym ............................................................................. 33
   SrSymTable ................................................................. 34
5 Symbol entries

5.1 Architecture .................................................. 34
5.2 Symbol entry types .......................................... 35

Base symbol type (TSym) ........................................... 35
label symbol (TLabelSym) ....................................... 36
unit symbol (TUnitSym) .......................................... 36
macro symbol (TMacroSym) ..................................... 36
error symbol (TErrorSym) ....................................... 37
procedure symbol (TProcSym) ................................ 37
type symbol (TTypeSym) ......................................... 37
variable symbol (TVarSym) ...................................... 38
property symbol (TPropertySym) ............................... 39
return value of function symbol ............................... 39
absolute declared symbol (TAbsoluteSym) .................. 39
typed constant symbol ......................................... 40
constant symbol (TConstSym) ................................ 40
enumeration symbol ............................................. 41
program symbol .................................................. 41
sys symbol ......................................................... 41

5.3 Symbol interface ............................................. 41

6 Type information

6.1 Architecture .................................................. 41
6.2 Definition types .............................................. 41

base definition (TDef) .......................................... 42
file definition (TFileDef) ....................................... 43
formal definition (TFormalDef) ................................. 44
forward definition (TForwardDef) ............................. 44
error definition (TErrorDef) ................................... 44
pointer definition (TPointerDef) ............................... 44
object definition ( TObjectDef) ................................. 44
class reference definition (TClassRefDef) .................... 46
array definition (TArrayDef) ........................................... 46
record definition (TRecordDef) ........................................ 46
ordinal definition (TOrdDef) ........................................... 46
float definition (TFloatDef) ........................................... 47
abstract procedure definition (tabstractprocdef) ................. 48
procedural variable definition (TProcVarDef) ....................... 50
procedure definition (TProcDef) ...................................... 50
string definition (TStringDef) ...................................... 51
classification definition (TEnumDef) .................................. 52
set definition (TSetDef) .............................................. 52
6.3 Definition interface ................................................. 53
routines ................................................................. 53
    TDef.Size .......................................................... 53
    TDef.Alignment .................................................... 53
7 The parser ............................................................. 53
    7.1 Module information ............................................. 54
    7.2 Parse types ....................................................... 56
        Entry ............................................................. 56
        program or library parsing .................................. 56
        unit parsing .................................................... 56
        routine parsing ............................................... 56
        label declarations ........................................... 56
        constant declarations ....................................... 56
        type declarations ............................................ 56
        variable declarations ....................................... 56
        thread variable declarations ................................ 56
        resource string declarations ................................ 56
        exports declaration ......................................... 56
        expression parsing .......................................... 56
        typed constant declarations ................................. 56
    7.3 Parser interface ................................................ 56
variables ................................................. 56
AktProcSym ........................................ 58
LexLevel .............................................. 58
Current_Module ..................................... 58
VoidDef .............................................. 58
cCharDef ............................................. 58
cWideCharDef ....................................... 59
BoolDef .............................................. 59
u8BitDef ............................................. 59
u16BitDef ............................................ 59
u32BitDef ............................................ 59
s32BitDef ............................................ 60
cu64BitDef .......................................... 60
cs64BitDef .......................................... 60
s64FloatDef .......................................... 60
s32FloatDef .......................................... 60
s80FloatDef .......................................... 61
s32FixedDef .......................................... 61
cShortStringDef ................................... 61
cLongStringDef ..................................... 61
c AnsiStringDef ...................................... 61
cWideStringDef ..................................... 62
OpenShortStringDef ................................. 62
OpenCharArrayDef ................................ 62
VoidPointerDef ..................................... 62
CharPointerDef ..................................... 62
VoidFarPointerDef ................................. 63
cFormalDef .......................................... 63
cFileDef ............................................. 63

8 The inline assembler parser 63
9 The code generator

9.1 Introduction .................................. 63

9.2 Locations (cpubase.pas) ....................... 64
   LOC_INVALID .................................. 65
   LOC_FPU ...................................... 65
      Stack based FPU .............................. 65
      Register based FPU ......................... 65
   LOC_REGISTER .................................. 65
   LOC_MEM, LOC_REFERENCE ...................... 66
   LOC_JUMP ..................................... 67
   LOC_FLAGS ................................... 67
   LOC_CREGISTER ................................ 67
   LOCATION PUBLIC INTERFACE ...................... 67
      Del_Location ................................ 67
      Clear_Location .............................. 67
      Set_Location ................................ 68
      Swap_Location ................................ 68

9.3 Registers (cpubase.pas) ....................... 68
   integer registers ............................. 68
   address registers ............................. 69
   fpu registers .................................. 69
   scratch registers .............................. 69

9.4 Special registers (cpubase.pas) ............... 69
   Stack_Pointer .................................. 69
   Frame_Pointer .................................. 69
   Self_Pointer ................................... 70
   accumulator .................................... 70
   scratch register .............................. 70

9.5 Instructions .................................. 70

9.6 Reference subsystem ............................ 70
   Architecture ................................. 70
   Code generator interface ...................... 70
9.7 The register allocator subsystem ................................. 71
   Architecture ..................................................... 71
   Code generator interface (tgen.pas) ............................ 71
      GetRegister32 .................................................. 72
      GetRegisterPair .............................................. 72
      UngetRegister32 ............................................. 72
      GetFloatRegister ........................................... 72
      IsFloatsRegister ............................................ 73
      GetAdressReg .................................................. 73
      IsAddressRegister .......................................... 73
      UngetRegister ................................................ 73
      SaveUsedRegisters ......................................... 73
      RestoreUsedRegisters ....................................... 74
      GetExplicitRegister32 ..................................... 74

9.8 Temporary memory allocator subsystem .......................... 74
   Architecture ..................................................... 74
   Temporary memory allocator interface (temp_gen.pas) .......... 75
      GetTempOfSize ............................................... 75
      GetTempOfSizeReference .................................... 75
      UnGetIfTemp .................................................. 75
      GetTempAnsiStringReference .................................. 75
      GetTempOfSizePersistant .................................... 76
      UngetPersistantTemp ........................................ 76
      ResetTempGen ................................................ 76
      SetFirstTemp ................................................ 76
      GetFirstTempSize ............................................ 76
      NormalTempToPersistant .................................... 77
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersistantTempToNormal</td>
<td>77</td>
</tr>
<tr>
<td>IsTemp</td>
<td>77</td>
</tr>
<tr>
<td>9.9 Assembler generation</td>
<td>77</td>
</tr>
<tr>
<td>Architecture</td>
<td>77</td>
</tr>
<tr>
<td>Generic instruction generation interface</td>
<td>78</td>
</tr>
<tr>
<td>Emit_Load_Loc_Reg</td>
<td>79</td>
</tr>
<tr>
<td>FloatLoad</td>
<td>79</td>
</tr>
<tr>
<td>FloatStore</td>
<td>79</td>
</tr>
<tr>
<td>emit_mov_ref_reg64</td>
<td>79</td>
</tr>
<tr>
<td>Emit_Lea_Loc_Ref</td>
<td>80</td>
</tr>
<tr>
<td>Emit_Lea_Loc_Reg</td>
<td>80</td>
</tr>
<tr>
<td>GetLabel</td>
<td>80</td>
</tr>
<tr>
<td>EmitLab</td>
<td>80</td>
</tr>
<tr>
<td>EmitLabeled</td>
<td>80</td>
</tr>
<tr>
<td>EmitCall</td>
<td>81</td>
</tr>
<tr>
<td>ConcatCopy</td>
<td>81</td>
</tr>
<tr>
<td>Emit_Flag2Reg</td>
<td>81</td>
</tr>
<tr>
<td>10 The assembler output</td>
<td>81</td>
</tr>
<tr>
<td>11 The Runtime library</td>
<td>84</td>
</tr>
<tr>
<td>11.1 Operating system hooks</td>
<td>86</td>
</tr>
<tr>
<td>System_Exit</td>
<td>86</td>
</tr>
<tr>
<td>ParamCount</td>
<td>86</td>
</tr>
<tr>
<td>Randomize</td>
<td>86</td>
</tr>
<tr>
<td>GetHeapStart</td>
<td>86</td>
</tr>
<tr>
<td>GetHeapSize</td>
<td>86</td>
</tr>
<tr>
<td>sbrk</td>
<td>87</td>
</tr>
<tr>
<td>Do_Close</td>
<td>87</td>
</tr>
<tr>
<td>Do_Erase</td>
<td>87</td>
</tr>
<tr>
<td>Do_Truncate</td>
<td>87</td>
</tr>
<tr>
<td>Do_Rename</td>
<td>88</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Do_Write</td>
<td>88</td>
</tr>
<tr>
<td>Do_Read</td>
<td>88</td>
</tr>
<tr>
<td>Do_FilePos</td>
<td>88</td>
</tr>
<tr>
<td>Do_Soke</td>
<td>89</td>
</tr>
<tr>
<td>Do_Seekend</td>
<td>89</td>
</tr>
<tr>
<td>Do_FileSize</td>
<td>89</td>
</tr>
<tr>
<td>Do_IsDevice</td>
<td>89</td>
</tr>
<tr>
<td>Do_Open</td>
<td>90</td>
</tr>
<tr>
<td>ChDir</td>
<td>90</td>
</tr>
<tr>
<td>MkDir</td>
<td>90</td>
</tr>
<tr>
<td>RmDir</td>
<td>90</td>
</tr>
<tr>
<td>11.2 CPU specific hooks</td>
<td>91</td>
</tr>
<tr>
<td>FPC_SETJMP</td>
<td>91</td>
</tr>
<tr>
<td>SetJmp</td>
<td>91</td>
</tr>
<tr>
<td>FPC_LONGJMP</td>
<td>91</td>
</tr>
<tr>
<td>function SPtr()</td>
<td>91</td>
</tr>
<tr>
<td>function Get_Caller_Frame(framebp:longint):longint;</td>
<td>91</td>
</tr>
<tr>
<td>function Get_Caller_Addr(framebp:longint):longint;</td>
<td>91</td>
</tr>
<tr>
<td>function Get_Frame:longint;</td>
<td>91</td>
</tr>
<tr>
<td>function Trunc()</td>
<td>91</td>
</tr>
<tr>
<td>11.3 String related</td>
<td>91</td>
</tr>
<tr>
<td>FPC_SHORTSTR_COPY</td>
<td>91</td>
</tr>
<tr>
<td>Int_StrCopy</td>
<td>91</td>
</tr>
<tr>
<td>FPC_SHORTSTR_COMPARE</td>
<td>92</td>
</tr>
<tr>
<td>Int_StrCmp</td>
<td>92</td>
</tr>
<tr>
<td>FPC_SHORTSTR_CONCAT</td>
<td>92</td>
</tr>
<tr>
<td>Int_StrConcat</td>
<td>92</td>
</tr>
<tr>
<td>FPC_ANSISTR_CONCAT</td>
<td>92</td>
</tr>
<tr>
<td>AnsiStr_Concat</td>
<td>92</td>
</tr>
<tr>
<td>FPC_ANSISTR_COMPARE</td>
<td>93</td>
</tr>
<tr>
<td>AnsiStr_Compare</td>
<td>93</td>
</tr>
<tr>
<td>FPC_ANSISTR_INCR_REF</td>
<td>93</td>
</tr>
</tbody>
</table>
11.4 Compiler runtime checking ................................. 96

FPC_STACKCHECK ........................................... 96

Int_StackCheck ........................................... 96

FPC_RANGEERROR ........................................... 96

Int_RangeError ........................................... 96

FPC_BOUNDCHECK ........................................... 96

Int_BoundCheck ........................................... 96

FPC_OVERFLOW .............................................. 97

Int_OverFlow .............................................. 97

FPC_CHECK_OBJECT .......................................... 97

Int_Check_Object .......................................... 97

FPC_CHECK_OBJECT_EXT ..................................... 98

Int_Check_Object_Ext ..................................... 98
11.5 Exception handling .................................................. 99
  FPC_RAISEEXCEPTION ............................................ 99
    RaiseExcept ................................................. 99
  FPC_PUSHEXCEPTADDR ........................................ 100
    PushExceptAddr ........................................... 100
  FPC_RERAISE ................................................ 100
    ReRaise .................................................. 100
  FPC_POPOBJECTSTACK .......................................... 100
    PopObjectStack ........................................... 100
  FPC_POPSECONDOBJECTSTACK ................................. 100
    PopSecondObjectStack ................................ 100
  FPC_DESTROYEXCEPTION ....................................... 101
    DestroyException .......................................... 101
  FPC_POPADDRSTACK ........................................... 101
    PopAddrStack ............................................. 101
  FPC_CATCHES ................................................ 101
    Catches .................................................. 101
  FPC_GETRESOURCESTRING ...................................... 101
    GetResourceString ........................................ 101

11.6 Runtime type information ...................................... 102
  FPC_DO_IS .................................................. 102
    Int_Do_Is ................................................ 102
  FPC_DO_AS .................................................. 102
    Int_Do_As ................................................ 102
  FPC_INITIALIZE ............................................. 102
    Initialize ................................................. 102
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPC_FINALIZE</td>
<td>103</td>
</tr>
<tr>
<td>Finalize</td>
<td>103</td>
</tr>
<tr>
<td>FPC_ADDREF</td>
<td>103</td>
</tr>
<tr>
<td>AddRef</td>
<td>103</td>
</tr>
<tr>
<td>FPC_DECREF</td>
<td>103</td>
</tr>
<tr>
<td>DecRef</td>
<td>103</td>
</tr>
<tr>
<td>11.7 Memory related</td>
<td>104</td>
</tr>
<tr>
<td>FPC_GETMEM</td>
<td>104</td>
</tr>
<tr>
<td>GetMem</td>
<td>104</td>
</tr>
<tr>
<td>FPC_FREEMEM</td>
<td>104</td>
</tr>
<tr>
<td>FreeMem</td>
<td>104</td>
</tr>
<tr>
<td>FPC_CHECKPOINTER</td>
<td>104</td>
</tr>
<tr>
<td>CheckPointer</td>
<td>104</td>
</tr>
<tr>
<td>FPC_DO_EXIT</td>
<td>104</td>
</tr>
<tr>
<td>Do.Exit</td>
<td>104</td>
</tr>
<tr>
<td>FPC_ABSTRACTERROR</td>
<td>105</td>
</tr>
<tr>
<td>AbstractError</td>
<td>105</td>
</tr>
<tr>
<td>FPC_INITIALIZEUNITS</td>
<td>105</td>
</tr>
<tr>
<td>InitializeUnits</td>
<td>105</td>
</tr>
<tr>
<td>FPC_NEW_CLASS (assembler)</td>
<td>105</td>
</tr>
<tr>
<td>int_new_class</td>
<td>105</td>
</tr>
<tr>
<td>FPC_HELP_DESTRUCTOR</td>
<td>105</td>
</tr>
<tr>
<td>Int_Help_Destructor</td>
<td>105</td>
</tr>
<tr>
<td>FPC_HELP_CONSTRUCTOR</td>
<td>106</td>
</tr>
<tr>
<td>Int_Help Constructor</td>
<td>106</td>
</tr>
<tr>
<td>FPC_HELP_FAIL_CLASS</td>
<td>106</td>
</tr>
<tr>
<td>Help_Fail_Class</td>
<td>106</td>
</tr>
<tr>
<td>FPC_HELP_FAIL</td>
<td>107</td>
</tr>
<tr>
<td>Help_Fail</td>
<td>107</td>
</tr>
<tr>
<td>11.8 Set handling</td>
<td>107</td>
</tr>
<tr>
<td>FPC_SET_COMP_SETS</td>
<td>107</td>
</tr>
<tr>
<td>Do_Comp_Sets</td>
<td>107</td>
</tr>
</tbody>
</table>
ShrInt64 ................................................................. 113
FPC_MUL_LONGINT .................................................. 113
MulLong ................................................................. 113
FPC_REM_LONGINT .................................................... 113
RemLong ................................................................. 113
FPC_DIV_LONGINT .................................................... 113
DivLong ................................................................. 113
FPC_MUL_LONGINT .................................................... 114
MulCardinal ............................................................. 114
FPC_REM_CARDINAL .................................................. 114
RemCardinal ............................................................. 114
FPC_DIV_CARDINAL .................................................. 114
DivCardinal ............................................................. 114
FPC_LONG_TO_SINGLE .............................................. 115
LongSingle ............................................................. 115

12 Optimizing your code .............................................. 115
  12.1 Simple types .................................................. 115
  12.2 constant duplicate merging ................................. 115
  12.3 inline routines ............................................... 115
  12.4 temporary memory allocation reuse ....................... 117

13 Appendix A ......................................................... 117
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>compiler overview</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>scanner interface overview</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>Example tree structure</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Interactions between symbol tables</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>Inserting into the symbol table</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>relation between symbol entry and type definition and name</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>Type symbol and definition relations</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>Parser - Scanner flow</td>
<td>57</td>
</tr>
<tr>
<td>9</td>
<td>Code generator architecture</td>
<td>64</td>
</tr>
<tr>
<td>10</td>
<td>Interaction between codegeneration and the parsing process</td>
<td>64</td>
</tr>
<tr>
<td>11</td>
<td>Assembler generation organisation</td>
<td>82</td>
</tr>
</tbody>
</table>
TODO:

- Describe in detail tsymtable, including all methods and fields
- Describe in detail procinfo (tprocinfo)
- Explain how a symbol is inserted into the symbol table (and how alignment requirements are met)
- Explain pparaitem
- Explain all symbol table fields
- Finish all internal routines definitions
- Architecture of the assembler generators + API
- Architecture of the PPU file and information
- Explain systems.pas
- routine parsing and code generation algorithm
- (MvdV) OS specific stuff (like hardcoded linker includedirs)

1 Introduction

This document describes the internal architecture of the Free Pascal Compiler version 1.0 release. This document is meant to be used as a guide for those who wish to understand how the compiler was created. Most of the architecture of the compiler described herein is based on the m68k version on the compiler, the i386 version of the compiler ressembles closely the m68k version, but there are subtle differences in the different interfaces.

The architecture, and the different passes of the compiler are shown in figure (??).

2 Scanner / Tokenizer

The scanner and tokenizer is used to construct an input stream of tokens which will be fed to the parser. It is in this stage that the preprocessing is done, that all read compiler directives change the internal state variables of the compiler, and that all illegal characters found in the input stream cause an error.
2.1 Architecture

The general architecture of the scanner is shown in figure 2.

Several types can be read from the input stream, a string, handled by `readstring()`, a numeric value, handled by `readnumeric()`, comments, compiler and preprocessor directives.

Input stream

The input data is handled via the standard way of handling all the I/O in the compiler. That is to say, that it is a hook which can be overridden in `comphook.pas (do_openinputfile)`, in case where another I/O method wants to be used.

The default hook uses a non-buffered dos stream contained in `files.pas`.

Figure 1: compiler overview
Figure 2: scanner interface overview
2.2 Scanner interface

**Preprocessor**

The scanner resolves all preprocessor directives and only gives to the parser the visible parts of the code (such as those which are included in conditional compilation). Compiler switches and directives are also saved in global variables while in the preprocessor, therefore this is part is completely independent of the parser.

**Conditional compilation (scandir.inc, scanner.pas)** The conditional compilation is handled via a preprocessor stack, where each directive is pushed on a stack, and popped when it is resolved. The actual implementation of the stack is a linked list of preprocessor directive items.

**Compiler switches (scandir.inc, switches.pas)** The compiler switches are handled via a lookup table which is linearly searched. Then another lookup table takes care of setting the appropriate bit flags and variables in the switches for this compilation process.

2.2 Scanner interface

The parser only receives tokens as its input, where a token is a enumeration which indicates the type of the token, either a reserved word, a special character, an operator, a numeric constant, string, or an identifier.

Resolution of the string into a token is done via lookup which searches the string table to find the equivalent token. This search is done using a binary search algorithm through the string table.

In the case of identifiers, constants (including numeric values), the value is returned in the **pattern** string variable, with the appropriate return value of the token (numeric values are also returned as non-converted strings, with any special prefix included). In the case of operators, and reserved words, only the token itself must be assumed to be preserved. The read input string is assumed to be lost.

Therefore the interface with the parser is with the **readtoken()** routine and the **pattern** variable.

**Routines**

**ReadToken**

**Declaration:** `Procedure ReadToken;`

**Description:** Sets the global variable `token` to the current token read, and sets the `pattern` variable appropriately (if required).
2.3 Assembler parser interface

Variables

Token

Description: Var Token : TToken;

Description: Contains the contain token which was last read by a call to ReadToken (19)

See also: ReadToken (19)

Pattern

Declaration: var Pattern : String;

Description: Contains the string of the last pattern read by a call to ReadToken (19)

See also: ReadToken (19)

2.3 Assembler parser interface

The inline assembler parser is completely separate from the pascal parser, therefore its scanning process is also completely independent. The scanner only takes care of the preprocessor part and comments, all the rest is passed character per character to the assembler parser via the AsmGetChar (20)() scanner routine.

routines

AsmGetChar

Declaration: Function AsmGetChar: Char;

Description: Returns the next character in the input stream.

3 The tree

3.1 Architecture

The tree is the basis of the compiler. When the compiler parses statements and blocks of code, they are converted to a tree representation. This tree representation is actually a doubly linked list. From this tree the code generation can easily be implemented.

Assuming that you have the following pascal syntax:
3.2 Tree types

The tree structure in picture 3 will be built in memory, where each circle represents an element (a node) in the tree:

$$x := x \ast y + (6 \text{ shl } x);$$

The following tree nodes are possible (of type TTreeTyp):

Table 1: Possible node types (ttreetyp)

<table>
<thead>
<tr>
<th>Tree type definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addn</td>
<td>Represents the + operator</td>
</tr>
<tr>
<td>muln</td>
<td>Represents the * operator</td>
</tr>
<tr>
<td>subn</td>
<td>Represents the - operator</td>
</tr>
<tr>
<td>divn</td>
<td>Represents the div operator</td>
</tr>
<tr>
<td>symdifn</td>
<td>Represents the &gt;= operator</td>
</tr>
<tr>
<td>modn</td>
<td>Represents the mod operator</td>
</tr>
<tr>
<td>assignn</td>
<td>Represents the := operator (assignment)</td>
</tr>
<tr>
<td>loadn</td>
<td>Represents the use of a variable</td>
</tr>
<tr>
<td>rangen</td>
<td>Represents a numeric range (i.e 0..9)</td>
</tr>
<tr>
<td>ltn</td>
<td>Represents the &lt; operator</td>
</tr>
<tr>
<td>lten</td>
<td>Represents the &lt;= operator</td>
</tr>
</tbody>
</table>
### 3.2 Tree types

Table 1: Possible node types (ttreetyp) - continued

<table>
<thead>
<tr>
<th>Tree type definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtn</td>
<td>Represents the $&gt;$ operator</td>
</tr>
<tr>
<td>gten</td>
<td>Represents the $\geq$ operator</td>
</tr>
<tr>
<td>equaln</td>
<td>Represents the $=$ operator</td>
</tr>
<tr>
<td>unequaln</td>
<td>Represents the $&lt;&gt;$ operator</td>
</tr>
<tr>
<td>inn</td>
<td>Represents the $\in$ operator</td>
</tr>
<tr>
<td>orn</td>
<td>Represents the $\or$ operator</td>
</tr>
<tr>
<td>xorn</td>
<td>Represents the $\xor$ operator</td>
</tr>
<tr>
<td>shrn</td>
<td>Represents the $\shr$ operator</td>
</tr>
<tr>
<td>shln</td>
<td>Represents the $\shl$ operator</td>
</tr>
<tr>
<td>slashn</td>
<td>Represents the $/$ operator</td>
</tr>
<tr>
<td>andn</td>
<td>Represents the $\and$ operator</td>
</tr>
<tr>
<td>subscriptn</td>
<td>Represents a field in an object or record</td>
</tr>
<tr>
<td>derefn</td>
<td>Represents a pointer reference (such as the $\hat{\text{operator}}$)</td>
</tr>
<tr>
<td>addrn</td>
<td>Represents the $@$ operator</td>
</tr>
<tr>
<td>doubleaddrn</td>
<td>Represents the $@@$ operator</td>
</tr>
<tr>
<td>ordconstn</td>
<td>Represents an ordinal constant</td>
</tr>
<tr>
<td>typeconvn</td>
<td>Represents a typecast / type conversion</td>
</tr>
<tr>
<td>calln</td>
<td>Represents a routine call</td>
</tr>
<tr>
<td>caliparan</td>
<td>Represents a parameter passed to a routine</td>
</tr>
<tr>
<td>realconstn</td>
<td>Represents a floating point constant</td>
</tr>
<tr>
<td>fixconstn</td>
<td>Represents a fixed point constant</td>
</tr>
<tr>
<td>unaryminusn</td>
<td>Represents a sign change (e.g. $: -$)</td>
</tr>
<tr>
<td>asmn</td>
<td>Represents an assembler statement node</td>
</tr>
<tr>
<td>vecn</td>
<td>Represents array indexing</td>
</tr>
<tr>
<td>pointerconstn</td>
<td>Represents a pointer constant</td>
</tr>
<tr>
<td>stringconstn</td>
<td>Represents a string constant</td>
</tr>
<tr>
<td>funcretn</td>
<td>Represents the return function result variable (not loadn)</td>
</tr>
<tr>
<td>selfn</td>
<td>Represents the self parameter</td>
</tr>
<tr>
<td>notn</td>
<td>Represents the $\not$ operator</td>
</tr>
<tr>
<td>inlinen</td>
<td>Represents one of the internal routines (writeln, ord, etc.)</td>
</tr>
<tr>
<td>niln</td>
<td>Represents the $\text{nil}$ pointer</td>
</tr>
<tr>
<td>erron</td>
<td>Represents error in parsing this node (used for error detection and correction)</td>
</tr>
<tr>
<td>typen</td>
<td>Represents a type name (i.e. typeof(obj))</td>
</tr>
<tr>
<td>hnewn</td>
<td>Represents the $\text{new}$ routine call on objects</td>
</tr>
<tr>
<td>hdisposen</td>
<td>Represents the $\text{dispose}$ routine call on objects</td>
</tr>
<tr>
<td>newn</td>
<td>Represents the $\text{new}$ routine call on non-objects</td>
</tr>
</tbody>
</table>
### Table 1: Possible node types (ttreetyp) - continued

<table>
<thead>
<tr>
<th>Tree type definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpledispose</td>
<td>Represents the <code>dispose</code> routine call on non-objects</td>
</tr>
<tr>
<td>setelementn</td>
<td>Represents set elements (i.e.: <code>[a..b]</code>, <code>[a,b,c]</code>) (non-constant)</td>
</tr>
<tr>
<td>setconstn</td>
<td>Represents set element constants i.e.: <code>[1..9]</code>, <code>[1,2,3]</code></td>
</tr>
<tr>
<td>blockn</td>
<td>Represents a block of statements</td>
</tr>
<tr>
<td>statementn</td>
<td>One statement in a block of nodes</td>
</tr>
<tr>
<td>loopn</td>
<td>Represents a loop (for, while, repeat) node</td>
</tr>
<tr>
<td>ifn</td>
<td>Represents an <code>if</code> statement</td>
</tr>
<tr>
<td>breakn</td>
<td>Represents a <code>break</code> statement</td>
</tr>
<tr>
<td>continuenn</td>
<td>Represents a <code>continue</code> statement</td>
</tr>
<tr>
<td>repeatn</td>
<td>Represents a <code>repeat</code> statement</td>
</tr>
<tr>
<td>whilen</td>
<td>Represents a <code>while</code> statement</td>
</tr>
<tr>
<td>formn</td>
<td>Represents a <code>for</code> statement</td>
</tr>
<tr>
<td>exitn</td>
<td>Represents an <code>exit</code> statement</td>
</tr>
<tr>
<td>withn</td>
<td>Represents a <code>with</code> statement</td>
</tr>
<tr>
<td>casen</td>
<td>Represents a <code>case</code> statement</td>
</tr>
<tr>
<td>labeln</td>
<td>Represents a label statement</td>
</tr>
<tr>
<td>gton</td>
<td>Represents a <code>goto</code> statement</td>
</tr>
<tr>
<td>simplenewn</td>
<td>Represents a <code>new</code> statement</td>
</tr>
<tr>
<td>tryexceptn</td>
<td>Represents a <code>try</code> statement</td>
</tr>
<tr>
<td>raisen</td>
<td>Represents a <code>raise</code> statement</td>
</tr>
<tr>
<td>switchesn</td>
<td>Unused</td>
</tr>
<tr>
<td>tryfinallyn</td>
<td>Represents a <code>try..finally</code> statement</td>
</tr>
<tr>
<td>onnn</td>
<td>Represents an <code>on..do</code> statement</td>
</tr>
<tr>
<td>isn</td>
<td>Represents the <code>is</code> operator</td>
</tr>
<tr>
<td>asn</td>
<td>Represents the <code>as</code> typecast operator</td>
</tr>
<tr>
<td>caretn</td>
<td>Represents the <code>operator</code></td>
</tr>
<tr>
<td>failn</td>
<td>Represents the <code>fail</code> statement</td>
</tr>
<tr>
<td>starstarn</td>
<td>Represents the <code>**</code> operator (exponentiation)</td>
</tr>
<tr>
<td>proclininen</td>
<td>Represents an <code>inline</code> routine</td>
</tr>
<tr>
<td>arrayconstrucn</td>
<td>Represents a <code>[..]</code> statement (array or sets)</td>
</tr>
<tr>
<td>arrayconstructrangen</td>
<td>Represents ranges in <code>[..]</code> statements (array or sets)</td>
</tr>
<tr>
<td>nothingn</td>
<td>Empty node</td>
</tr>
<tr>
<td>loadvmtn</td>
<td>Load method table register</td>
</tr>
</tbody>
</table>

### 3.3 Tree structure fields (tree.pas)

Each element in a node is a pointer to a TTree structure, which is summarily explained and defined as follows:
### 3.3 Tree structure fields (tree.pas)

```pascal
TYPE
  pTree = ^TTree;
TTree = RECORD
  Error : boolean;              // Set to TRUE if there was an error parsing this node
    DisposeTyp : tdisposeotyp;
    Swaped : boolean;            // Set to TRUE if the left and right nodes (fields) of this node have been swapped.
    VarStateSet : boolean;
    Location : tlocation;        // Location information for this information (cf. Code generator)
    Registers32 : longint;       // Minimum number of general purpose registers required to evaluate this node
    RegistersFpu : longint;      // Minimum number of floating point registers required to evaluate this node
    Left : pTree;                // LEFT leaf of this node
    Right : pTree;               // RIGHT leaf of this node
    ResultType : pDef;           // Result type of this node (cf. Type definitions)
    FileInfo : TFilePosInfo;     // Line number information for this node creation in the original source code (for error management)
    LocalSwitches : tlocalswitches; // Local compiler switches used for code generation (Cf. 2)
    IsProperty : boolean;        // TRUE if this is a property
    TreeType : ttreetyp;         // Type of this tree (cf. ??)
END;
```

#### Table 2: local compiler switches (tlocalswitches)

<table>
<thead>
<tr>
<th>tlocalswitches</th>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs_check_overflow</td>
<td>{$Q+}</td>
<td>Code generator should emit overflow checking code</td>
</tr>
<tr>
<td>cs_check_range</td>
<td>{$R+}</td>
<td>Code generator should emit range checking code</td>
</tr>
<tr>
<td>cs_check_IO</td>
<td>{$I+}</td>
<td>Code generator should emit I/O checking code</td>
</tr>
<tr>
<td>cs_check_object_ext</td>
<td>N/A</td>
<td>Code generator should emit extended object access checks</td>
</tr>
<tr>
<td>cs_omitstackframe</td>
<td>N/A</td>
<td><em>Code generator should not emit frame_pointer setup code in entry code</em></td>
</tr>
<tr>
<td>cs_do_assertion</td>
<td>{$C+}</td>
<td>Code generator supports using the assert inline routine</td>
</tr>
<tr>
<td>cs_generate_rtti</td>
<td>{$M+}</td>
<td>Code generator should emit runtime type information</td>
</tr>
<tr>
<td>cs_typed_addresses</td>
<td>{$T+}</td>
<td>Parser emits typed pointer using the @ operator</td>
</tr>
<tr>
<td>cs_ansistrings</td>
<td>{$H+}</td>
<td>Parser creates an ansiString when an unspecified String type is declared instead of the default ShortString</td>
</tr>
</tbody>
</table>
### 3.3 Tree structure fields (tree.pas)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tlocalswitches</td>
<td>Switch Description</td>
</tr>
<tr>
<td>cs_strict_var_strings</td>
<td>{$V+} String types must be identical (same length) to be compatible</td>
</tr>
</tbody>
</table>

#### Additional fields

Depending on the tree type, some additional fields may be present in the tree node. This section describes these additional fields. Before accessing these additional fields, a check on the treetype should always be done to verify if not reading invalid memory ranges.

#### AddN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use_StrConcat : Boolean;</td>
<td>Currently unused (use for optimizations in future versions)</td>
</tr>
<tr>
<td>String_Typ: TStringType;</td>
<td>In the case where the + operator is applied on a string, this field indicates the string type.</td>
</tr>
</tbody>
</table>

#### CallParaN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is_Colon_Para : Boolean;</td>
<td>Used for internal routines which can use optional format parameters (using colons). Is set to TRUE if this parameter was preceded by a colon (i.e : :1)</td>
</tr>
<tr>
<td>Exact_Match_Found : Boolean;</td>
<td>Set to TRUE if the parameter type is exactly the same as the one expected by the routine.</td>
</tr>
<tr>
<td>ConvLevel1Found : Boolean;</td>
<td>Set to TRUE if the parameter type requires a level 1 type conversion to conform to the parameter expected by the routine.</td>
</tr>
<tr>
<td>ConvLevel2Found : Boolean;</td>
<td>Set to TRUE if the parameter type requires a level 2 type conversion to conform to the parameter expected by the routine.</td>
</tr>
<tr>
<td>HighTree : pTree;</td>
<td></td>
</tr>
</tbody>
</table>

#### AssignN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssignTyp : TAssignTyp;</td>
<td>Currently unused (Used to be used for C-like assigns)</td>
</tr>
<tr>
<td>Concat_String : Boolean;</td>
<td>Currently unused (use for optimizations in future versions)</td>
</tr>
</tbody>
</table>

#### LoadN
### 3.3 Tree structure fields (tree.pas)

#### Field

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SymTableEntry : pSym;</td>
<td>Symbol table entry for this symbol</td>
</tr>
<tr>
<td>SymTable : pSymTable;</td>
<td>Symbol table in which this symbol is stored</td>
</tr>
<tr>
<td>Is_Absolute : Boolean;</td>
<td>set to TRUE if this variable is absolute</td>
</tr>
<tr>
<td>Is_First : Boolean;</td>
<td>set to TRUE if this is the first occurrence of the load for this variable (used with the varstate variable for optimizations)</td>
</tr>
</tbody>
</table>

#### CallN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SymTableProcEntry : pProcSym;</td>
<td>Symbol table entry for this routine</td>
</tr>
<tr>
<td>SymTableProc : pSymTable;</td>
<td>Symbol table associated with a call (object symbol table or routine symbol table)</td>
</tr>
<tr>
<td>ProcDefinition : pAbstractProcDef;</td>
<td>Type definition for this routine</td>
</tr>
<tr>
<td>MethodPointer : pTree;</td>
<td></td>
</tr>
<tr>
<td>No_Check : Boolean;</td>
<td>Currently unused</td>
</tr>
<tr>
<td>Unit_Specific : Boolean;</td>
<td>set to TRUE if the routine is imported in a unit specific way (for example: system.writeln())</td>
</tr>
<tr>
<td>Return_Value_Used : Boolean;</td>
<td>set to TRUE if the routine is a function and that the return value is not used (in extended syntax parsing - $X+)</td>
</tr>
<tr>
<td>Static_Call : Boolean;</td>
<td>unused</td>
</tr>
</tbody>
</table>

#### addrn

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcVarLoad : Boolean;</td>
<td>Set to TRUE if this is a procedural variable call</td>
</tr>
</tbody>
</table>

#### OrdConstN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value : Longint;</td>
<td>The numeric value of this constant node</td>
</tr>
</tbody>
</table>

#### RealConstN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value_Real : Best_Real;</td>
<td>The numeric value of this constant node</td>
</tr>
<tr>
<td>Lab_Real : pAsmLabel;</td>
<td>The assembler label reference to this constant</td>
</tr>
</tbody>
</table>

#### FixConstN
### 3.3 Tree structure fields (tree.pas)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value_Fix : Longint;</td>
<td>The numeric value of this constant node</td>
</tr>
</tbody>
</table>

#### FuncRetN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FuncRetProcInfo : Pointer; (pProcInfo)</td>
<td>Pointer to procedure information</td>
</tr>
<tr>
<td>RetType : TType;</td>
<td>Indicates the return type of the function</td>
</tr>
<tr>
<td>Is_First_FuncRet : Boolean;</td>
<td></td>
</tr>
</tbody>
</table>

#### SubscriptN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs : pVarSym;</td>
<td>Symbol table entry for this variable (a field of object/class/record)</td>
</tr>
</tbody>
</table>

#### RaiseN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FrameTree : pTree;</td>
<td>Exception frame tree (code in Raise statement)</td>
</tr>
</tbody>
</table>

#### VecN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemIndex : Boolean;</td>
<td>Set to TRUE if Mem[Seg:Ofs] directive is parsed</td>
</tr>
<tr>
<td>MemSeg : Boolean;</td>
<td>Set to TRUE if Mem[Seg:Ofs] directive is parsed</td>
</tr>
<tr>
<td>CallUnique: Boolean;</td>
<td></td>
</tr>
</tbody>
</table>

#### StringConstN

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value_Str : pChar;</td>
<td>The constant value of the string</td>
</tr>
<tr>
<td>Length : Longint;</td>
<td>Length of the string in bytes (or in characters???)</td>
</tr>
<tr>
<td>Lab_Str : pAsmLabel;</td>
<td>The assembler label reference to this constant</td>
</tr>
<tr>
<td>StringType : TStrings;</td>
<td>The string type (short, long, ansi, wide)</td>
</tr>
</tbody>
</table>

#### TypeConvN
3.3 Tree structure fields (tree.pas)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvType: TConvertType;</td>
<td>Indicates the conversion type to do</td>
</tr>
<tr>
<td>Explizit : Boolean;</td>
<td>set to TRUE if this was an explicit conversion (with explicit typecast, or calling one of the internal conversion routines)</td>
</tr>
</tbody>
</table>

**TypeN**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TypeNodeType : pDef;</td>
<td>The type definition for this node</td>
</tr>
<tr>
<td>TypeNodeSym : pTypeSym;</td>
<td>The type symbol information</td>
</tr>
</tbody>
</table>

**InlineN**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InlineNumber: Byte;</td>
<td>Indicates the internal routine called (Cf. code generator)</td>
</tr>
<tr>
<td>InlineConst : Boolean;</td>
<td>One or more of the parameters to this inline routine call contains constant values</td>
</tr>
</tbody>
</table>

**ProcInlineN**

Inline nodes are created when a routine is declared as being inline. The routine is actually inlined when the following conditions are satisfied:

- It is called within the same module
- The appropriate compiler switch to support inline is activated
- It is a non-method routine (a standard procedure or function)

Otherwise a normal call is made, ignoring the inline directive. In the case where a routine is inlined, all parameters, return values and local variables of the inlined routine are actually allocated in the stack space of the routine which called the inline routine.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InlineTree : pTree;</td>
<td>The complete tree for this inline procedure</td>
</tr>
<tr>
<td>InlineProcsym : pProcSym;</td>
<td>Symbol table entry for this procedure</td>
</tr>
<tr>
<td>RetOffset : Longint;</td>
<td>Return offset in parent routine stack space</td>
</tr>
<tr>
<td>Para_Offset : Longint;</td>
<td>Parameter start offset in parent routine stack space</td>
</tr>
<tr>
<td>Para_Size : Longint;</td>
<td>Parameter size in the parent routine stack space</td>
</tr>
</tbody>
</table>

**SetConstN**
### 3.3 Tree structure fields (tree.pas)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value_Set : pConstSet;</td>
<td>The numeric value of this constant node</td>
</tr>
<tr>
<td>Lab_Set : pAsmLabel;</td>
<td>The assembler label reference to this constant</td>
</tr>
</tbody>
</table>

**LoopN**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AsmN**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_Asm : pAasmOutput;</td>
<td>The instruction tree created by the assembler parser</td>
</tr>
<tr>
<td>Object_Preserved : Boolean;</td>
<td>set to FALSE if the Self_Register was modified in the asm</td>
</tr>
<tr>
<td></td>
<td>statement.</td>
</tr>
</tbody>
</table>

**CaseN**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes : pCaserecord;</td>
<td>Tree for each of the possible case in the case statement</td>
</tr>
<tr>
<td>ElseBlock : pTree;</td>
<td>Else statement block tree</td>
</tr>
</tbody>
</table>

**LabelN, GotoN**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LabelNr : pAsmLabel;</td>
<td>Assembler label associated with this statement</td>
</tr>
<tr>
<td>ExceptionBlock : ptree;</td>
<td>?</td>
</tr>
<tr>
<td>LabSym : pLabelSym;</td>
<td>Symbol table entry for this label</td>
</tr>
</tbody>
</table>

**WithN**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WithSymTables : pWithSymTable;</td>
<td></td>
</tr>
<tr>
<td>TableCount : Longint;</td>
<td></td>
</tr>
<tr>
<td>WithReference : pReference;</td>
<td></td>
</tr>
<tr>
<td>IsLocal : Boolean;</td>
<td></td>
</tr>
</tbody>
</table>
4 Symbol tables

4.1 Architecture

The symbol table contains all definitions for all symbols in the compiler. It also contains all type information for all symbols encountered during the parsing process. All symbols and definitions are streamable, and are used within PPU files to avoid recompiling everything to verify if all symbols are valid.

There are different types of symbol tables, all of which maybe active at one time or another depending on the context of the parser.

An architectural overview of the interaction between the symbol tables, the symbol entries and the definition entries is displayed in figure 4.1

As can be seen, the symbol table entries in the symbol table are done using the fast hashing algorithm with a hash dictionary.

4.2 The Symbol table object

All symbol tables in the compiler are from this type of object, which contains fields for the total size of the data in the symbol table, and methods to read and write the symbol table into a stream. The start of the linked list of active symbol tables is the symtablestack variable.
4.2 The Symbol table object

Figure 4: Interactions between symbol tables

```
TYPE
  pSymTable  = ^TSymTable;
  TSymTable = object
    Name : pString;
    DataSize : Longint;

    DataAlignment : Longint;
    SymIndex : plIndexArray;
    DefIndex : plIndexArray;
    SymSearch : pDictionary;
    Next : pSymtable;

    DefOwner : pDef;

    Address_Fixup : Longint
    UnitId : Word;
    SymTableLevel : Byte;
    SymTableType : TSymTableType;
  end;
```

The total size of all the data in this symbol table (after the data has been aligned). Only valid for certain types of symbol tables.

Points to the next symbol table in the linked list of active symbol tables. The owner definition (only valid in the cases of objects and records, this points to the definition of that object or record).

Indicates the type of this symbol table (2).
4.3 Inserting symbols into a symbol table

The type of possible symbol tables are shown in the following table:

<table>
<thead>
<tr>
<th>TSymTableType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvalidSymTable</td>
<td>Default value when the symbol table is created and its type is not defined. Used for debugging purposes</td>
</tr>
<tr>
<td>WithSymTable</td>
<td>All symbols accessed in a with statement</td>
</tr>
<tr>
<td>StaticSymTable</td>
<td></td>
</tr>
<tr>
<td>GlobalSymTable</td>
<td></td>
</tr>
<tr>
<td>UnitSymTable</td>
<td>Linked list of units symbol used (all or unit?). The linked list is composed of tunitsym structures.</td>
</tr>
<tr>
<td>ObjectSymTable</td>
<td></td>
</tr>
<tr>
<td>RecordSymTable</td>
<td>Contains all symbols within a record statement</td>
</tr>
<tr>
<td>MacroSymTable</td>
<td>Holds all macros currently in scope.</td>
</tr>
<tr>
<td>LocalSymTable</td>
<td>Hold symbols for all local variables of a routine</td>
</tr>
<tr>
<td>ParaSymTable</td>
<td>Holds symbols for all parameters of a routine (the actual parameter declaration symbols)</td>
</tr>
<tr>
<td>InlineParaSymTable</td>
<td>Holds all parameter symbols for the current inline routine</td>
</tr>
<tr>
<td>InlineLocalSymTable</td>
<td></td>
</tr>
<tr>
<td>Stt_ExceptSymTable</td>
<td></td>
</tr>
<tr>
<td>StaticPPUSymTable</td>
<td></td>
</tr>
</tbody>
</table>

4.3 Inserting symbols into a symbol table

To add a symbol into a specific symbol table, that’s symbol table’s Insert method is called, which in turns call the Insert_In_Data method of that symbol. Insert_In_Data, depending on the symbol type, adjusts the alignment and sizes of the data and actually creates the data entry in the correct segment.

4.4 Symbol table interface

Routines

Search_a_Symtable

Declaration: Function Search_a_Symtable(Const Symbol:String; SymTableType : TSymTableType):pSym;

Description: Search for a symbol Symbol in a specified symbol table SymTableType. Returns NIL if the symbol table is not found, and also if the symbol cannot be found in the desired symbol table.
4.4 Symbol table interface

Figure 5: Inserting into the symbol table

**Declaration:** Procedure GetSym(Const S : StringId; NotFoundError: Boolean);

**Description:** Search all the active symbol tables for the symbol s, setting the global variable SrSym to the found symbol, or to nil if the symbol was not found. NotFoundError should be set to TRUE if the routine must give out an error when the symbol is not found.

**GlobalDef**

**Declaration:** Function GlobalDef(Const S : String) : pDef;

**Description:** Returns a pointer to the definition of the fully qualified type symbol S, or NIL if not found.

**Notes:** It is fully qualified, in that the symbol system.byte, for example, will be fully resolved to a unit and byte type component. The symbol must have a global scope, and it must be a type symbol, otherwise NIL will be returned.

**Variables**

**SrSym**

**Declaration:** Var SrSym : pSym;
Description: This points to the symbol entry found, when calling getsym.

SrSymTable

Declaration: Var SrSymTable : pSymTable;

Description: This points to the symbol table of the symbol SrSym (33) when calling GetSym (32).

5 Symbol entries

5.1 Architecture

There are different possible types of symbols, each one having different fields then the others. Each symbol type has a specific signature to indicate what kind of entry it is. Each entry in the symbol table is actually one of the symbol entries described in the following sections. The relationship between a symbol entry, a type definition, and the type name symbol entry is shown in figure 5.1.
5.2 Symbol entry types

Base symbol type (TSym)

All entries in the symbol table are derived from this base object which contains information on the symbol type as well as information on the owner of this symbol entry.

```pascal
TYPE
  pSym = ^TSym;
  TSym = Object(TSymTableEntry)
      SymOptions : TSymOptions;
      FileInfo : tFilePosInfo;
      Refs : Longint;
      LastRef : pRef;
      DefRef : pRef;
      LastWritten : pRef;
      RefCount : Longint;
      Typ : tSymTyp;
      IsStabWritten : Boolean;
  end;
```

Table 30: tsymtyp

<table>
<thead>
<tr>
<th>TSymTyp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractSym</td>
<td>This is a special abstract symbol (this should never occur)</td>
</tr>
<tr>
<td>VarSym</td>
<td>This symbol is a variable declaration in the var section, or a var parameter.</td>
</tr>
<tr>
<td>TypeSym</td>
<td>This symbol is a type name</td>
</tr>
<tr>
<td>ProcSym</td>
<td>This symbol is a routine or method name</td>
</tr>
<tr>
<td>UnitSym</td>
<td>This symbol is a unit name</td>
</tr>
<tr>
<td>ProgramSym</td>
<td><em>This symbol is the main program name</em></td>
</tr>
<tr>
<td>ConstSym</td>
<td>This symbol is a constant</td>
</tr>
<tr>
<td>EnumSym</td>
<td>This symbol is an enumeration symbol (an element in an enumeration)</td>
</tr>
<tr>
<td>TypedConstSym</td>
<td>This symbol is pre-initialized variable (pascal typed constant)</td>
</tr>
<tr>
<td>ErrorSym</td>
<td>This symbol is created for error generation</td>
</tr>
<tr>
<td>SysSym</td>
<td>This symbol represents an inlined system unit routine</td>
</tr>
<tr>
<td>LabelSym</td>
<td>This symbol represents a label in a label pascal declaration</td>
</tr>
<tr>
<td>AbsoluteSym</td>
<td>This symbol represents an the symbol following an absolute variable declaration</td>
</tr>
</tbody>
</table>
### 5.2 Symbol entry types

<table>
<thead>
<tr>
<th>TSymTyp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PropertySym</td>
<td>This symbol is a property name</td>
</tr>
<tr>
<td>FuncRetSym</td>
<td>This symbol is the name of the return value for functions</td>
</tr>
<tr>
<td>MacroSym</td>
<td>This symbol is a macro symbol (just like #define in C)</td>
</tr>
</tbody>
</table>

**label symbol (TLabelSym)**

The label symbol table entry is only created when a pascal label is declared via the label keyword. The object has the following fields which are available for use publicly:

```pascal
TYPE
  pLabelSym = ^ TLabelSym;
  TLabelSym = Object(TSym)
  Used : Boolean Set to TRUE if this pascal label is used using a goto or in an assembler statement
  Defined : Boolean Set to TRUE if this label has been declared
  Lab : pAsmLabel Points to the actual assembler label structure which will be emitted by the code generator
  Code : Pointer
end;
```

**unit symbol (TUnitSym)**

The unit symbol is created and added to the symbol table each time that the uses clause is parsed and a unit name is found, it is also used when compiling a unit, with the first entry in that symbol table being the unit name being compiled. The unit symbol entry is actual part of a linked list which is used in the unit symbol table.

```pascal
TYPE
  pUnitSym = ^ TUnitSym;
  TUnitSym = Object(TSym)
  UnitSymTable : pUnitSymTable Pointer to the global symbol table for that unit, containing entries for each public symbol in that unit
  PrevSym : pUnitSym Pointer to previous entry in the linked list
end;
```

**macro symbol (TMacroSym)**

The macro symbols are used in the preprocessor for conditional compilation statements. There is one such entry created for each $define directive, it contains the value of the define (stored as a string).
5.2 Symbol entry types

**error symbol (TErrorSym)**

This symbol is actually an empty symbol table entry. When the parser encounters an error when parsing a symbol, instead of putting nothing in the symbol table, it puts this symbol entry. This avoids illegal memory accesses later in parsing.

**procedure symbol (TProcSym)**

The procedure symbol is created each time a routine is defined in the code. This can be either a forward definition or the actual implementation of the routine. After creation, the symbol is added into the appropriate symbol table stack.

**type symbol (TTypeSym)**

The type symbol is created each time a new type declaration is done, the current symbol table stack is then inserted with this symbol. Furthermore, each time the compiler compiles a module, the default base types are initialized and added into the symbol table (psystem.pas) The type symbol contains the name of a type, as well as a pointer to its type definition.
5.2 Symbol entry types

**variable symbol (TVarsym)**

Variable declarations, as well as parameters which are passed onto routines are declared as variable symbol types. Access information, as well as type information and optimization information are stored in this symbol type.

```pascal
TYPE
  pTypeSym = ^ TTypeSym;
  TTypeSym = Object(TSym)
  ResType : TType
  Contains base type information as well as the type definition
end;
```

Variable declarations, as well as parameters which are passed onto routines are declared as variable symbol types. Access information, as well as type information and optimization information are stored in this symbol type.

```pascal
TYPE
  pVarSym = ^ TVarsym;
  TVarsym = Object(TSym)
  Reg: TRegister;
  VarSpez : TVarspez;
  Address : Longint;
  If the value is a register variable, the reg field will be different then R_NO
  Indicates the variable type (parameters only) (Cf. 32).
  In the case where the variable is a routine parameter, this indicates the positive offset from the frame_pointer to access this variable. In the case of a local variable, this field indicates the negative offset from the frame_pointer to access this variable.
  LocalVarSym : pVarSym;
  VarType : TType;
  VarOptions : TVarOptions;
  VarState : TVarState
  Contains base type information as well as the type definition
  Flags for this variable (Cf. 31)
  Indicates the state of the variable, if it’s used or declared
end;
```

<table>
<thead>
<tr>
<th>TVarOptions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vo_regab</td>
<td>The variable can be put into a hardware general purpose register</td>
</tr>
<tr>
<td>vo_is_c_var</td>
<td>The variable is imported from a C module</td>
</tr>
<tr>
<td>vo_is_external</td>
<td>The variable is declared external</td>
</tr>
<tr>
<td>vo_is_DLL_var</td>
<td>The variable is a shared library variable</td>
</tr>
<tr>
<td>vo_is_thread_var</td>
<td>The variable is declared as being thread safe</td>
</tr>
</tbody>
</table>
5.2 Symbol entry types

Table 31: tvaroptions (continued)

<table>
<thead>
<tr>
<th>TVarOptions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vo_fpuregable</td>
<td>The variable can be put into a hardware floating point register</td>
</tr>
<tr>
<td>vo_is_local_copy</td>
<td>unused and useless</td>
</tr>
<tr>
<td>vo_is_const</td>
<td>The variable is declared as exported in a dynamic link library</td>
</tr>
<tr>
<td>vo_is_exported</td>
<td></td>
</tr>
</tbody>
</table>

Table 32: parameter type

<table>
<thead>
<tr>
<th>TVarSpez</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs_value</td>
<td>This is a value parameter</td>
</tr>
<tr>
<td>vs_const</td>
<td>This is a constant parameter, property or array</td>
</tr>
<tr>
<td>vs_var</td>
<td>This is a variable parameter</td>
</tr>
</tbody>
</table>

**property symbol (TPropertySym)**

```pascal
TYPE
pPropertySym = ^TPropertySym;
TPropertySym = Object(TSym)
    proptyp: tpropertyoptions;
    proptype : ttype;
    propoverriden : ppropertysym;
    indextype : ttype;
    index : longint;
    default : longint
    readaccess : psymlist
    writeaccess : psymlist
    storedaccess : psymlist
end;
```

**return value of function symbol**

**absolute declared symbol (TAbsoluteSym)**

This symbol represents a variable declared with the `absolute` keyword. The address of the TVarSym object holds the address of the variable in the case of an absolute address variable.
The possible types of absolute symbols, are from an external object reference, an absolute
address (for certain targets only), or on top of another declared variable. For the possible
types, 33.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pAbsoluteSym = ˆ TAbsoluteSym; TAbsoluteSym = Object(TVarSym)</td>
<td>Indicates the type of absolute symbol it is (Cf. 33)</td>
</tr>
<tr>
<td>abstyp : absolutetyp; absseg : boolean; ref : psym;</td>
<td>In case abstyp is tovar, this field indicates the symbol which is overlaid with this symbol. Otherwise this field is unused.</td>
</tr>
<tr>
<td>asmname : pstring;</td>
<td>In case abstyp is toasm, this field indicates label name for the variable.</td>
</tr>
</tbody>
</table>

Table 33: possible absolute variable types

<table>
<thead>
<tr>
<th>tabsolutetyp</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tovar</td>
<td>The symbol will be declared on top of another symbol (variable or typed constant)</td>
</tr>
<tr>
<td>toasm</td>
<td>The variable is imported from an external module</td>
</tr>
<tr>
<td>toaddr</td>
<td>The variable is declared as being at an absolute address</td>
</tr>
</tbody>
</table>

**typed constant symbol**

**constant symbol (TConstSym)**

This symbol type will contain all constants defined and encountered during the parsing. The values of the constants are also set in this symbol type entry.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pConstSym = ˆ TConstSym; TConstSym = Object(TSym)</td>
<td>Type information for this constant (?).</td>
</tr>
<tr>
<td>consttype : ttype; consttyp : tconsttyp</td>
<td>Indicates the type of the constant</td>
</tr>
<tr>
<td>resstrindex : longint</td>
<td>If this is a resource string constant, it indicates the index in the resource table</td>
</tr>
<tr>
<td>value : longint len : longint</td>
<td>In certain cases, contains the value of the constant</td>
</tr>
</tbody>
</table>
5.3 Symbol interface

### enumeration symbol

**program symbol**

The program symbol type (tprogramsym) is used to store the name of the program, which is declared using `program` in the pascal source. This symbol type is currently unused in FreePascal.

**sys symbol**

The tsysssym symbol type is used to load indexes into the symbol table of the internal routines which are inlined directly by the compiler. It has a single field, which is the index of the inline routine.

## 6 Type information

### 6.1 Architecture

A type declaration, which is the basis for the symbol table, since inherently everything comes down to a type after parsing is a special structure with two principal fields, which point to a symbol table entry which is the type name, and the actual definition which gives the information on other symbols in the type, the size of the type and other such information.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>TType =</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Object</strong></td>
</tr>
<tr>
<td></td>
<td>Sym : pSym;</td>
</tr>
<tr>
<td></td>
<td>Def : pDef;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
</tbody>
</table>

### 6.2 Definition types

Definitions represent the type information for all possible symbols which can be encountered by the parser. The definition types are associated with symbols in the symbol table, and are used by the parsing process (among other things) to perform type checking.

The current possible definition types are enumerated in `TDefType` and can have one of the following symbolic values:
6.2 Definition types

![Figure 7: Type symbol and definition relations](image)

<table>
<thead>
<tr>
<th>deftype of TDef object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractDef</td>
<td>array type definition</td>
</tr>
<tr>
<td>ArrayDef</td>
<td>record type definition</td>
</tr>
<tr>
<td>RecordDef</td>
<td>pointer type definition</td>
</tr>
<tr>
<td>OrdDef</td>
<td>ordinal (numeric value) type definition</td>
</tr>
<tr>
<td>StringDef</td>
<td>string type definition</td>
</tr>
<tr>
<td>EnumDef</td>
<td>enumeration type definition</td>
</tr>
<tr>
<td>ProcDef</td>
<td>procedure type definition</td>
</tr>
<tr>
<td>ObjectDef</td>
<td>object or class type definition</td>
</tr>
<tr>
<td>ErrorDef</td>
<td>error definition (empty, used for error recovery)</td>
</tr>
<tr>
<td>FileDef</td>
<td>file type definition</td>
</tr>
<tr>
<td>FormalDef</td>
<td>set type definition</td>
</tr>
<tr>
<td>SetDef</td>
<td>procedure variable type definition</td>
</tr>
<tr>
<td>FloatDef</td>
<td>floating point type definition</td>
</tr>
<tr>
<td>ClassrefDef</td>
<td></td>
</tr>
<tr>
<td>ForwardDef</td>
<td></td>
</tr>
</tbody>
</table>

**base definition (TDef)**

All type definitions are based on this object. Therefore all derived object all posess the fields in this object in addition to their own private fields.

42
6.2 Definition types

**TYPE**

\[
pDef = ^\ W TDef;\]

\[
TDef = \textbf{Object}(\text{TSymTableEntry})\]

\[
\text{TypeSym : pTypeSym; }\quad \text{Pointer to symbol table entry for this type definition}\]

\[
\text{InitTable\_Label : pAsmLabel; }\quad \text{Label to initialization information (required for some complex types)}\]

\[
\text{Rtti\_Label : pAsmLabel; }\quad \text{Label to the runtime type information.}\]

\[
\text{NextGlobal : pDef; }\quad \text{Size in bytes of the data definition}\]

\[
\text{PreviousGlobal : pDef; }\quad \text{Indicates the definition type (see table ??}).\]

\[
\text{SaveSize : Longint; }\quad \text{Can be one of the following states: (Not\_Written, written, Being\_Written)}\]

\[
\text{DefType : tDefType; }\quad \text{which indicates if the debug information for this type has been defined or not.}\]

\[
\text{Has\_InitTable : Boolean; }\quad \text{Internal stabs debug information type signature (each type definition has a numeric}}\]

\[
\text{Has\_Rtti : Boolean; }\quad \text{signature).}\]

\[
\text{Is\_Def\_Stab\_Written : TDefStabStatus}\]

\[
\text{GlobalNb : Longint; }\quad \text{end;}\]

**file definition (TFileDef)**

The file definition can occur in only some rare instances, when a file of type is parsed, a file definition of that type will be created. Furthermore, internally, a definition for a Text file type and untyped File type are created when the system unit is loaded. These types are always defined when compiling any unit or program.

**TYPE**

\[
pFileDef = ^\ W TFileDef;\]

\[
TFileDef = \textbf{Object}(TDef)\]

\[
\text{FileTyp : TFileTyp; }\quad \text{Indicates what type of file definition it is (text, untyped or typed).}\]

\[
\text{TypedFileType : TType; }\quad \text{In the case of a typed file definition, definition of the type of the file}\]

\[
\text{end;}\]
6.2 Definition types

formal definition (TFormalDef)

forward definition (TForwardDef)

The forward definition is created, when a type is declared before an actual definition exists. This is the case, when, for example `type pmyobject = tmyobject`, while `tmyobject` has yet to be defined.

```
TYPE
  pForwardDef = ^ TForwardDef;
  TForwardDef = Object(TDef)
    toSymName : String;  The symbol name for this forward declaration (the actual real definition does not exist yet)
    ForwardPos : TFileInfo;  Indicates file position where this forward definition was declared.
  end;
```

error definition (TErrorDef)

This definition is actually an empty definition entry. When the parser encounters an error when parsing a definition instead of putting nothing in the type for a symbol, it puts this entry. This avoids illegal memory accesses later in parsing.

pointer definition (TPointerDef)

The pointer definition is used for distinguishing between different types of pointers in the compiler, and are created at each `typename` parsing construct found.

```
TYPE
  pPointerDef = ^ TPointerDef;
  TPointerDef = Object(TDef)
    Is_Far : Boolean;  Used to indicate if this is a far pointer or not (this flag is cpu-specific)
    PointerType : TType;  This indicates to what type definition this pointer points to.
  end;
```

object definition (TObjectDef)

The object definition is created each time an object declaration is found in the type declaration section.
6.2 Definition types

TYPE
pObjectDef = ^ TObjectDef;
TObjectDef = Object(TDef)
  ChildOf : pObjectDef;
  This is a pointer to the parent object definition. It is set to nil, if this object definition has no parent.
  ObjName : pString;
  This is the object name
  SymTable : pSymTable;
  This is a pointer to the symbol table entries within this object.
  PobjectOptions : TObjectOptions;
  The options for this object, see the following table for the possible options for the object.
  VMT_Offset : Longint;
  This is the offset from the start of the object image in memory where the virtual method table is located.
Writing_Class_Record_Stab : Boolean;
end;

<table>
<thead>
<tr>
<th>Object Options(TObjectOptions)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oo_is_class</td>
<td>This is a delphi styled class declaration, and not a Turbo Pascal object.</td>
</tr>
<tr>
<td>oo_is_forward</td>
<td>This flag is set to indicate that the object has been declared in a type section, but there is no implementation yet.</td>
</tr>
<tr>
<td>oo_has_virtual</td>
<td>This object / class contains virtual methods</td>
</tr>
<tr>
<td>oo_has_private</td>
<td>This object / class contains private fields or methods</td>
</tr>
<tr>
<td>oo_has_protected</td>
<td>This object / class contains protected fields or methods</td>
</tr>
<tr>
<td>oo_has_constructor</td>
<td>This object / class has a constructor method</td>
</tr>
<tr>
<td>oo_has_destructor</td>
<td>This object / class has a destructor method</td>
</tr>
<tr>
<td>oo_has_vmt</td>
<td>This object / class has a virtual method table</td>
</tr>
<tr>
<td>oo_has_msgstr</td>
<td>This object / class contains one or more message handlers</td>
</tr>
<tr>
<td>oo_has_msgint</td>
<td>This object / class contains one or more message handlers</td>
</tr>
<tr>
<td>oo_has_abstract</td>
<td>This object / class contains one or more abstract methods</td>
</tr>
<tr>
<td>oo_can_have_published</td>
<td>the class has runtime type information, i.e. you can publish properties</td>
</tr>
<tr>
<td>oo_cpp_class</td>
<td>the object/class uses an C++ compatible class layout</td>
</tr>
<tr>
<td>oo_interface</td>
<td>this class is a delphi styled interface</td>
</tr>
</tbody>
</table>
class reference definition (TClassRefDef)

array definition (TArrayDef)

This definition is created when an array type declaration is parsed. It contains all the information necessary for array type checking and code generation.

```pascal
TYPE
  pArrayDef = ^ TArrayDef;
  TArrayDef = Object(TDef)
    IsVariant : Boolean;
    IsConstructor : Boolean;
    RangeNr: Longint;
    LowRange : Longint;
    HighRange : Longint;
    ElementType : TType;
    RangeType : TType;
    IsArrayofConst : Boolean;
end;
```

record definition (TRecordDef)

The record definition entry is created each time a record type declaration is parsed. It contains the symbol table to the elements in the record.

```pascal
TYPE
  pRecordDef = ^ TRecordDef;
  TRecordDef = Object(TDef)
    SymTable : PSymTable;
    This is a pointer to the symbol table entries within this record.
end;
```

ordinal definition (TOrdDef)

This type definition is the one used for all ordinal values such as char, bytes and other numeric integer type values. Some of the predefined type definitions are automatically created and loaded when the compiler starts. Others are created at compile time, when declared.
6.2 Definition types

**Base ordinal type (TBaseType)**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uauto</td>
</tr>
<tr>
<td>uvoid</td>
</tr>
<tr>
<td>uchar</td>
</tr>
<tr>
<td>u8bit</td>
</tr>
<tr>
<td>u16bit</td>
</tr>
<tr>
<td>u32bit</td>
</tr>
<tr>
<td>s16bit</td>
</tr>
<tr>
<td>s32bit</td>
</tr>
<tr>
<td>bool8bit</td>
</tr>
<tr>
<td>bool16bit</td>
</tr>
<tr>
<td>bool32bit</td>
</tr>
<tr>
<td>u64bit</td>
</tr>
<tr>
<td>s64bit</td>
</tr>
<tr>
<td>uwidechar</td>
</tr>
</tbody>
</table>

**Base floating point type (TFloatType)**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s32real</td>
</tr>
<tr>
<td>s64real</td>
</tr>
</tbody>
</table>

**float definition (TFloatDef)**

This type definition is the one used for all floating point values such as SINGLE, DOUBLE. Some of the predefined type definitions are automatically created and loaded when the compiler starts.
6.2 Definition types

<table>
<thead>
<tr>
<th>Base floating point type (TFloatType)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s80real</td>
<td>Extended precision floating point value (cpu-specific, usually maps to double)</td>
</tr>
<tr>
<td>s64comp</td>
<td>63-bit signed value, using 1 bit for sign indication</td>
</tr>
<tr>
<td>f16bit</td>
<td>Unsupported</td>
</tr>
<tr>
<td>f32bit</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>

**abstract procedure definition (Tabstractprocdef)**

This is the base of all routine type definitions. This object is abstract, and is not directly used in a useful way. The derived object of this object are used for the actual parsing process.

```pascal
TYPE
  pAbstractProcDef = ^ TAbstractProcDef;
  TAbstractProcDef = Object(TDef)
    SymtableLevel : byte;
    Fpu_Used : Byte;
    RetType : TType;
    ProcTypeOption : TProcTypeOption;
    ProcCallOptions : TProcCallOptions;
    ProcOptions : TProcOptions;
    Para : pLinkedList;
end;
```

**Table 38: Procedure type options**

<table>
<thead>
<tr>
<th>Procedure options (TProcTypeOption)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>poType_ProgInit</td>
<td>Routine is the program entry point (defined as ‘main’ in the compiler).</td>
</tr>
<tr>
<td>poType_UnitInit</td>
<td>Routine is the unit initialization code</td>
</tr>
<tr>
<td></td>
<td>(defined as unitname_init in the compiler)</td>
</tr>
<tr>
<td>poType_UnitFinalize</td>
<td>Routine is the unit exit code</td>
</tr>
<tr>
<td></td>
<td>(defined as unitname_finalize in the compiler)</td>
</tr>
<tr>
<td>poType_Constructor</td>
<td>Routine is an object or class constructor</td>
</tr>
</tbody>
</table>

48
### 6.2 Definition types

#### Table 38: Procedure type options (continued)

<table>
<thead>
<tr>
<th>Procedure options (TProcTypeOption)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>poTypeDestructor</td>
<td>Routine is an object or class destructor</td>
</tr>
<tr>
<td>poTypeOperator</td>
<td>Procedure is an operator</td>
</tr>
</tbody>
</table>

#### Table 39: Procedure call options

<table>
<thead>
<tr>
<th>Call options (TProcCallOptions)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pocall_clearstack</td>
<td>The routine caller clears the stack upon return</td>
</tr>
<tr>
<td>pocall_leftright</td>
<td>Send parameters to routine from left to right</td>
</tr>
<tr>
<td>pocall_cdecl</td>
<td>Passing parameters is done using the GCC alignment scheme, passing parameter values is directly copied into the stack space</td>
</tr>
<tr>
<td>pocall_register</td>
<td>unused (Send parameters via registers)</td>
</tr>
<tr>
<td>pocall_stdcall</td>
<td>Passing parameters is done using GCC alignment scheme, standard GCC registers are saved</td>
</tr>
<tr>
<td>pocall_safeSCALL</td>
<td>Standard GCC registers are saved</td>
</tr>
<tr>
<td>pocall_palmsysSCALL</td>
<td>This is a special syscall macro for embedded system</td>
</tr>
<tr>
<td>pocall_system</td>
<td>unused</td>
</tr>
<tr>
<td>pocall_inline</td>
<td>Routine is an inline assembler macro (not a true call)</td>
</tr>
<tr>
<td>pocall_internproc</td>
<td>System unit code generator helper routine</td>
</tr>
<tr>
<td>pocall_internconst</td>
<td>System unit code generator helper macro routine</td>
</tr>
</tbody>
</table>

#### Table 40: Procedure options

<table>
<thead>
<tr>
<th>Routine options (TProcOptions)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>po_classmethod</td>
<td>This is a class method</td>
</tr>
<tr>
<td>po_virtualmethod</td>
<td>This is a virtual method</td>
</tr>
<tr>
<td>po_abstractmethod</td>
<td>This is an abstract method</td>
</tr>
<tr>
<td>po_staticmethod</td>
<td>This is a static method</td>
</tr>
<tr>
<td>po_overridingmethod</td>
<td>This is an overridden method (with po_virtual flag usually)</td>
</tr>
<tr>
<td>po_methodpointer</td>
<td>This is a method pointer (not a normal routine pointer)</td>
</tr>
<tr>
<td>po_containself</td>
<td>self is passed explicitly as a parameter to the method</td>
</tr>
<tr>
<td>po_interrupt</td>
<td>This routine is an interrupt handler</td>
</tr>
<tr>
<td>po_iocheck</td>
<td>IO checking should be done after a call to the procedure</td>
</tr>
<tr>
<td>po_assembler</td>
<td>The routine is in assembler</td>
</tr>
<tr>
<td>po_assembler</td>
<td>method for string message handling</td>
</tr>
<tr>
<td>po_exports</td>
<td>Routine has export directive</td>
</tr>
<tr>
<td>po_extra</td>
<td>Routine is external (in other object or lib)</td>
</tr>
<tr>
<td>po_savestdregs</td>
<td>Routine entry should save all registers used by GCC</td>
</tr>
</tbody>
</table>
6.2 Definition types

Table 40: Procedure options (continued)

<table>
<thead>
<tr>
<th>Routine options (TProcOptions)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>po_saveregisters</code></td>
<td>Routine entry should save all registers</td>
</tr>
<tr>
<td><code>po_overload</code></td>
<td>Routine is declared as being overloaded</td>
</tr>
</tbody>
</table>

**procedural variable definition (TProcVarDef)**

This definition is created when a procedure variable type is declared. It gives information on the type of a procedure, and is used when assigning and directly calling a routine through a pointer.

```pascal
TYPE
  pProcVarDef = ^TProcVarDef;
  TProcVarDef = Object(TAbstractProcDef)
  end;
```

**procedure definition (TProcDef)**

When a procedure head is parsed, the definition of the routine is created. Thereafter, other fields containing information on the definition of the routine are populated as required.
### 6.2 Definition types

**pProcDef = TProcDef**

**TProcDef =** `Object(TAbstractProcDef)`

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ForwardDef : Boolean;</code></td>
<td>TRUE if this is a forward definition</td>
</tr>
<tr>
<td><code>InterfaceDef : Boolean;</code></td>
<td></td>
</tr>
<tr>
<td><code>ExtNumber : Longint;</code></td>
<td></td>
</tr>
<tr>
<td><code>MessageInf : TMessageInf;</code></td>
<td></td>
</tr>
<tr>
<td><code>NextOverloaded : pProcDef;</code></td>
<td></td>
</tr>
<tr>
<td><code>FileInfo : TFilePosInfo;</code></td>
<td>Position in source code for the declaration of this routine. Used for error management.</td>
</tr>
<tr>
<td><code>Localst : pSymTable;</code></td>
<td>The local variables symbol table</td>
</tr>
<tr>
<td><code>Parast : pSymTable;</code></td>
<td>The parameter symbol table</td>
</tr>
<tr>
<td><code>ProcSym : pProcSym;</code></td>
<td>Points to owner of this definition</td>
</tr>
<tr>
<td><code>LastRef : pRef;</code></td>
<td></td>
</tr>
<tr>
<td><code>DefRef : pRef;</code></td>
<td></td>
</tr>
<tr>
<td><code>CrossRef : pRef;</code></td>
<td></td>
</tr>
<tr>
<td><code>LastWritten : pRef;</code></td>
<td></td>
</tr>
<tr>
<td><code>RefCount : Longint;</code></td>
<td></td>
</tr>
<tr>
<td><code>__Class : TObjectDef;</code></td>
<td></td>
</tr>
<tr>
<td><code>Code : Pointer;</code></td>
<td>The actual code for the routine (only for in-lined routines)</td>
</tr>
<tr>
<td><code>UsedRegisters : TRegisterSet;</code></td>
<td>The set of registers used in this routine</td>
</tr>
<tr>
<td><code>HasForward : Boolean;</code></td>
<td></td>
</tr>
<tr>
<td><code>Count : Boolean;</code></td>
<td></td>
</tr>
<tr>
<td><code>Is_Used : Boolean;</code></td>
<td></td>
</tr>
<tr>
<td><code>end;</code></td>
<td></td>
</tr>
</tbody>
</table>

**string definition (TStringDef)**

This definition represents all string types as well as derived types. Some of the default string type definitions are loaded when the compiler starts up. Others are created at compile time as they are declared with a specific length type.

**pStringDef = ^TStringDef;**

**TStringDef =** `Object(TDef)`

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String_Typ : TStringType;</code></td>
<td>Indicates the string type definition (cf. 41)</td>
</tr>
<tr>
<td><code>Len : Longint;</code></td>
<td>This is the maximum length which can have the string</td>
</tr>
<tr>
<td><code>end;</code></td>
<td></td>
</tr>
</tbody>
</table>
6.2 Definition types

Table 41: string types

<table>
<thead>
<tr>
<th>String type (TStringType)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_default</td>
<td>Depends on current compiler switches, can either be a st_ShortString or st_AnsiString</td>
</tr>
<tr>
<td>st_shortstring</td>
<td>short string (length byte followed by actual ASCII characters (1 byte/char))</td>
</tr>
<tr>
<td>st_longstring</td>
<td>long string (length longint followed by actual ASCII characters (1 byte/char))</td>
</tr>
<tr>
<td>st_ansistring</td>
<td>long string garbage collected (pointer to a length, reference count followed by actual ASCII characters (1 byte/char))</td>
</tr>
<tr>
<td>st_widestring</td>
<td>long string garbage collected (pointer to a length, reference count followed by actual unicode characters (1 word/char (utf16)))</td>
</tr>
</tbody>
</table>

enumeration definition (TEnumDef)

An enumeration definition is created each time an enumeration is declared and parsed. Each element in the enumeration will be added to the linked list of symbols associated with this enumeration, and this symbol table will then be attached to the enumeration definition.

```pascal
TYPE
  pEnumDef = ^TEnumDef;
  TEnumDef = object(TDef)
  Has_Jumps : Boolean; { Currently unused }
  MinVal : Longint;
  MaxVal : Longint;
  FirstEnum : pEnumSym;
  BaseDef : pEnumDef;
  end;
```

set definition (TSetDef)

This definition is created when a set type construct is parsed (set of declaration).
6.3 Definition interface

routines

TDef.Size

Declaration: Function TDef.Size : Longint;

Description: This method returns the true size of the memory space required in bytes for this type definition (after alignment considerations).

TDef.Alignment

Declaration: Function TDef.Alignment : Longint;

Description: This method returns the alignment of the data for complex types such as records and objects, otherwise returns 0 or 1 (no alignment).

7 The parser

The task of the parser is to read the token fed by the scanner, and make sure that the pascal syntax is respected. It also populates the symbol table, and creates the intermediate nodes (the tree) which will be used by the code generator.
7.1 Module information

Each module being compiled, be it a library, unit or main program has some information which is required. This is stored in the tmodule object in memory. To avoid recompilation of already compiled module, the dependencies of the modules is stored in a PPU file, which makes it easier to determine which modules to recompile.

```pascal
TYPE
  pModule = ^TModule;
  TModule = Object(TLinkedList_Item)
  PPUFile : pPPUFile;
  Crc : Longint;
  Interface_CRC : Longint;
  Flags: Longint;
  Compiled: Boolean;
  Do_Reload : Boolean;
  Do_Assemble : Boolean;
  Sources_Avail : Boolean;
  Sources_Checked : Boolean;
  Is_Unit: Boolean;
  In_Compile: Boolean;
  In_Second_Compile: Boolean;
  In_Second_Load: Boolean;
  In_Implementation : Boolean;
  In_Global : Boolean;
  Recompile_Reason : TRecompile_Reason;

  Pointer to PPU file object (unit file)
  CRC-32 bit of the whole PPU file
  CRC-32 bit of the interface part of
  the PPU file
  Unit file flags
  TRUE if module is already compiled
  TRUE if the PPU file must be reloaded
  Only assemble, don’t recompile unit
  TRUE if all sources of module are available
  TRUE if the sources has already been checked
  TRUE if this is a unit (otherwise a library or a main program)
  module is currently being recompiled
  module is being compiled for second time
  module is being reloaded a second time
  currently compiling implementation part (units only)
  currently compiling implementation part (units only)
  Reason why module should be recompiled
```
### Module information

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Islibrary : Boolean;</td>
<td>TRUE if this module is a shared library</td>
</tr>
<tr>
<td>Map : pUnitMap;</td>
<td>Map of all used units for this unit</td>
</tr>
<tr>
<td>Unitcount : Word;</td>
<td>Internal identifier of unit (for GDB support)</td>
</tr>
<tr>
<td>Unit_index : Eord;</td>
<td></td>
</tr>
<tr>
<td>Globalsymtable : Pointer;</td>
<td>Symbol table for this module of externally visible symbols</td>
</tr>
<tr>
<td>Localsymtable : Pointer;</td>
<td>Symbol table for this module of locally visible symbols</td>
</tr>
<tr>
<td>Scanner : Pointer;</td>
<td>Scanner object pointer</td>
</tr>
<tr>
<td>Loaded_From : pModule;</td>
<td>Module which referred to this module</td>
</tr>
<tr>
<td>Uses_Imports : Boolean;</td>
<td>TRUE if this module imports symbols from a shared library</td>
</tr>
<tr>
<td>Imports : pLinkedList</td>
<td>Linked list of imported symbols</td>
</tr>
<tr>
<td>_Exports : pLinkedList;</td>
<td>Linked list of exported symbols (libraries only)</td>
</tr>
<tr>
<td>SourceFiles : pFileManager;</td>
<td>List of all source files for this module</td>
</tr>
<tr>
<td>ResourceFiles : TStringContainer;</td>
<td>List of all resource files for this module</td>
</tr>
<tr>
<td>Used_Units : TLinkedList;</td>
<td>Information on units used by this module (pused_unit)</td>
</tr>
<tr>
<td>Dependent_Units : TLinkedList;</td>
<td></td>
</tr>
<tr>
<td>LocalUnitSearchPath,</td>
<td>Search path for obtaining module source code</td>
</tr>
<tr>
<td>LocalObjectSearchPath,</td>
<td></td>
</tr>
<tr>
<td>LocalIncludeSearchPath,</td>
<td></td>
</tr>
<tr>
<td>LocalLibrarySearchPath:TSearchPathList;</td>
<td>Path were module is located or created</td>
</tr>
<tr>
<td>Path : pString;</td>
<td></td>
</tr>
<tr>
<td>OutputPath : pString;</td>
<td>Path where object files (unit), executable (program) or shared library (library) is created</td>
</tr>
<tr>
<td>ModuleName : pString;</td>
<td>Name of the module in uppercase</td>
</tr>
<tr>
<td>ObjFileName : pString;</td>
<td>Full name of object file or executable file</td>
</tr>
<tr>
<td>AsmFileName : pString;</td>
<td></td>
</tr>
<tr>
<td>PPUFileName : pString;</td>
<td>Full name of the assembler file</td>
</tr>
</tbody>
</table>
### 7.2 Parse types

#### Entry

- program or library parsing
- unit parsing
- routine parsing
- label declarations
- constant declarations
- type declarations
- variable declarations
- thread variable declarations
- resource string declarations
- exports declaration
- expression parsing
- typed constant declarations

#### 7.3 Parser interface

- variables
7.3 Parser interface

Figure 8: Parser - Scanner flow
AktProcSym

Declaration: Var AktProcSym : pProcSym;

Description: Pointer to the symbol information for the routine currently being parsed.

LexLevel

Declaration: var LexLevel : longint;

Description: Level of code currently being parsed and compiled

0 = for main program
1 = for subroutine
2 = for local / nested subroutines.

Current_Module

Declaration: Var Current_Module : pModule;

Description: Information on the current module (program, library or unit) being compiled.

The following variables are default type definitions which are created each time compilation begins (default system-unit definitions), these definitions should always be valid:

VoidDef

Declaration: Var VoidDef : pOrdDef;

Description: Pointer to nothing type

Notes: This is loaded as a default supported type for the compiler

cCharDef

Declaration: Var cCharDef : pOrdDef;

Description: Type definition for a character (char)

Notes: This is loaded as a default supported type for the compiler
cWideCharDef

Declaration: Var cWideCharDef : pOrdDef;

Description: Type definition for a unicode character (widechar)

Notes: This is loaded as a default supported type for the compiler

BoolDef

Declaration: Var BoolDef : pOrdDef;

Description: Type definition for a boolean value (boolean)

Notes: This is loaded as a default supported type for the compiler

u8BitDef

Declaration: Var u8BitDef : pOrdDef;

Description: Type definition for an 8-bit unsigned value (byte)

Notes: This is loaded as a default supported type for the compiler

u16BitDef

Declaration: Var u16BitDef : pOrdDef;

Description: Type definition for an unsigned 16-bit value (word)

Notes: This is loaded as a default supported type for the compiler

u32BitDef

Declaration: Var u32BitDef : pOrdDef;

Description: Type definition for an unsigned 32-bit value (cardinal)

Notes: This is loaded as a default supported type for the compiler
s32BitDef
Declaration: Var s32BitDef : pOrdDef;
Description: Type definition for a signed 32-bit value (longint)
    Notes: This is loaded as a default supported type for the compiler

cu64BitDef
Declaration: Var cu64BitDef : pOrdDef;
Description: Type definition for an unsigned 64-bit value (qword)
    Notes: This is loaded as a default supported type for the compiler

cs64BitDef
Declaration: Var cs64BitDef : pOrdDef;
Description: Type definition for a signed 64-bit value (int64)
    Notes: This is loaded as a default supported type for the compiler

    The following variables are default type definitions which are created each time compilation begins (default system-unit definitions), these definitions should always be valid:

s64FloatDef
Declaration: Var s64FloatDef : pFloatDef;
Description: Type definition for a 64-bit IEEE floating point type (double)
    Notes: This is loaded as a default supported type for the compiler. This might not actually really point to the double type if the cpu does not support it.

s32FloatDef
Declaration: Var s32FloatDef : pFloatDef;
Description: Type definition for a 32-bit IEEE floating point type (single)
    Notes: This is loaded as a default supported type for the compiler. This might not actually really point to the single type if the cpu does not support it.
7.3  Parser interface

s80FloatDef

Declaration:  Var s80FloatDef : pFloatDef;

Description:  Type definition for an extended floating point type (extended)

Notes:  This is loaded as a default supported type for the compiler. This might not actually really point to the extended type if the cpu does not support it.

s32FixedDef

Declaration:  Var s32FixedDef : pFloatDef;

Description:  Type definition for a fixed point 32-bit value (fixed)

Notes:  This is loaded as a default supported type for the compiler. This is not supported officially in FPC 1.0

The following variables are default type definitions which are created each time compilation begins (default system-unit definitions), these definitions should always be valid:

cShortStringDef

Declaration:  Var cShortStringDef : pStringDef;

Description:  Type definition for a short string type (shortstring)

Notes:  This is loaded as a default supported type for the compiler.

cLongStringDef

Declaration:  Var cLongStringDef : pStringDef;

Description:  Type definition for a long string type (longstring)

Notes:  This is loaded as a default supported type for the compiler.

cAnsiStringDef

Declaration:  Var cAnsiStringDef : pStringDef;

Description:  Type definition for an ansistring type (ansistring)

Notes:  This is loaded as a default supported type for the compiler.
cWideStringDef

Declaration: Var cWideStringDef : pStringDef;

Description: Type definition for an wide string type (widestring)

Notes: This is loaded as a default supported type for the compiler.

OpenShortStringDef

Declaration: Var OpenShortStringDef : pStringDef;

Description: Type definition for an open string type (openstring)

Notes: This is loaded as a default supported type for the compiler.

OpenCharArrayDef

Declaration: Var OpenCharArrayDef : pArrayDef;

Description: Type definition for an open char array type(openchararray)

Notes: This is loaded as a default supported type for the compiler.

The following variables are default type definitions which are created each time compilation begins (default system-unit definitions), these definitions should always be valid:

VoidPointerDef

Declaration: Var VoidPointerDef : pPointerDef;

Description: Type definition for a pointer which can point to anything (pointer)

Notes: This is loaded as a default supported type for the compiler

CharPointerDef

Declaration: Var CharPointerDef : pPointerDef;

Description: Type definition for a pointer which can point to characters (pchar)

Notes: This is loaded as a default supported type for the compiler
VoidFarPointerDef

Declaration: Var VoidFarPointerDef : pPointerDef;

Description: Type definition for a pointer which can point to anything (intra-segment) (far pointer)

Notes: This is loaded as a default supported type for the compiler

cFormalDef

Declaration: Var cFormalDef : pFormalDef;

Notes: This is loaded as a default supported type for the compiler

cFileDef

Declaration: Var cFileDef : pFileDef;

Description: This is the default file type (file)

Notes: This is loaded as a default supported type for the compiler

8 The inline assembler parser

To be written.

9 The code generator

9.1 Introduction

The code generator is responsible for creating the assembler output in form of a linked list, taking as input the node created in the parser and the 1st pass. Picture figure (9.1) shows an overview of the code generator architecture:

The code generation is only done when a procedure body is parsed; the interaction, between the 1st pass (type checking phase), the code generation and the parsing process is show in the following diagram:

The secondpass() is actually a simple dispatcher. Each possible tree type node (Cf. Tree types) is associated with a second pass routine which is called using a dispatch table.
9.2 Locations (cpubase.pas)

The code generator uses the tree location component to indicate the location where the current node operands are located. This is then used by the code generator to generate the appropriate instruction, all depending on the location of the operand. The possible operand locations:

<table>
<thead>
<tr>
<th>Location define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC_INVALID</td>
<td>Invalid location (should never occur)</td>
</tr>
<tr>
<td>LOC_FPU</td>
<td>Floating point registers</td>
</tr>
<tr>
<td>LOC_REGISTER</td>
<td>Integer registers</td>
</tr>
<tr>
<td>LOC_MEM</td>
<td>Memory Location</td>
</tr>
<tr>
<td>LOC_REFERENCE</td>
<td>Constant node with constant value</td>
</tr>
<tr>
<td>LOC_JUMP</td>
<td>Label operand</td>
</tr>
<tr>
<td>LOC_FLAGS</td>
<td>Flags operand</td>
</tr>
</tbody>
</table>

Figure 9: Code generator architecture

Figure 10: Interaction between code generation and the parsing process
9.2 Locations (cpubase.pas)

<table>
<thead>
<tr>
<th>Location define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC_CREGISTER</td>
<td>Constant integer register (when operand is in this location, it should be considered as read-only)</td>
</tr>
</tbody>
</table>

Depending on the location type, a variable structure is defined indicating more information on the operand. This is used by the code generator to generate the exact instructions.

**LOC_INVALID**

This location does not contain any related information, when this location occurs, it indicates that the operand location was not initially allocated correctly. This indicates a problem in the compiler.

**LOC_FPU**

This indicates a location in the coprocessor; this is platform dependant.

**Stack based FPU**  Only one CPU uses a stack based FPU architecture, this is the intel 80x86 family of processors. When the operand is on the top of the stack, the operand is of type LOC_FPU.

**Register based FPU**  When the floating point co-processor is register based, the following field(s) are defined in the structure to indicate the current location of the operand:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FpuRegister : TRegister;</td>
<td>Indicates in what register the operand is located (a general purpose register in emulation mode, and a floating point register when floating point hardware is present)</td>
</tr>
<tr>
<td>FpuRegisterHigh, FpuRegisterLow : TRegister;</td>
<td>Indicates in what registers the operand are located (for emulation support - these are general purpose registers)</td>
</tr>
</tbody>
</table>

**LOC_REGISTER**

This fields indicates that the operand is located in a CPU register. It is possible to allocate more then one register, if trying to access 64-bit values on 32-bit wide register machines.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register : TRegister</td>
<td>Indicates in what register the operand is located.</td>
</tr>
</tbody>
</table>
9.2 Locations (cpubase.pas)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegisterHigh : TRegister;</td>
<td>High 32-bit of 64-bit virtual register (on 32-bit machines)</td>
</tr>
<tr>
<td>RegisterLow : TRegister;</td>
<td>Low 32-bit of 64-bit virtual register (on 32-bit machines)</td>
</tr>
</tbody>
</table>

**LOC_MEM, LOC_REFERENCE**

This either indicates an operand in memory, or a constant integer numeric value. The fields for this type of operand is as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference : TReference;</td>
<td>Information on the location in memory</td>
</tr>
</tbody>
</table>

References are the basic building blocks of the code generator, every load and store in memory is done via a reference. A reference type can either point to a symbolic name, an assembler expression (base register + index register + offset)*scale factor, as well as simply giving information on a numeric value.

The reference consists of the following:

```pascal
TYPE
  pReference = *TReference;
  TReference = packed Record
    Is_Immediate : Boolean;
    Segment : TRegister;               (cpu-specific)
    Base : TRegister;                  Base address register for assembler expression
    Index : TRegister;                 Index register for assembler expression
    ScaleFactor : Byte;                Multiplication factor for assembler expression (this field is cpu-specific)
    Offset : Longint;                 Either an offset from base assembler address expression to add (if Is_Constant = FALSE) otherwise the numeric value of the operand
    Symbol : pAsmSymbol;
    OffsetFixup : Longint;
    Options : TRefOptions;
  END;
```

Indicates that this location points to a memory location, but to a constant value (TRUE), which is located in the offset field.
LOC_JUMP

There are no fields associated with this location, it simply indicates that it is a boolean comparison which must be done to verify the succeeding operations. (i.e the processor zero flag is valid and gives information on the result of the last operation).

LOC_FLAGS

The operand is in the flags register. From this operand, the conditional jumps can be done. This is processor dependant, but normally the flags for all different comparisons should be present.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResFlags : TResFlags;</td>
<td>This indicates the flag which must be verified for the actual jump operation.</td>
</tr>
<tr>
<td></td>
<td>tresflags is an enumeration of all possible conditional flags which can be set by the processor.</td>
</tr>
</tbody>
</table>

LOC_CREGISTER

This is a read-only register allocated somewhere else in the code generator. It is used mainly for optimization purposes. It has the same fields as LOC_REGISTER, except that the registers associated with this location can only be read from, and should never be modified directly.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register : TRegister</td>
<td>Indicates in what register the operand is located.</td>
</tr>
<tr>
<td>RegisterHigh : TRegister;</td>
<td>High 32-bit of 64-bit virtual register (on 32-bit machines)</td>
</tr>
<tr>
<td>RegisterLow : TRegister;</td>
<td>Low 32-bit of 64-bit virtual register (on 32-bit machines)</td>
</tr>
</tbody>
</table>

LOCATION PUBLIC INTERFACE

Del_Location

Declaration: procedur Del_Location(const L : TLocation);

Description: If the location points to a LOC_REGISTER or LOC_CREGISTER, it frees up the allocated register(s) associated with this location. If the location points to LOC_REFERENCE or LOC_MEM, it frees up the the allocated base and index registers associated with this node.

Clear_Location

Declaration: procedure Clear_location(var Loc : TLocation);

Description: Sets the location to point to a LOC_INVALID type.
9.3 Registers (cpubase.pas)

Set_Location

Declaration: procedure Set_Location(var Destloc, Sourceloc : TLocation);

Description: The destination location now points to the destination location (now copy is made, a simple pointer assignment)

Swap_Location

Declaration: Procedure Swap_Location(var Destloc, Sourceloc : TLocation);

Description: Swap both location pointers.

9.3 Registers (cpubase.pas)

The code generator defines several types of registers which are categorized by classes. All (except for the scratch register class) of these register classes are allocated / freed on the fly, when the code is generated in the code generator: The registers are defined in a special enumeration called tregister. This enumeration contains all possible register defines for the target architecture, and a possible definition could be as follows:

```pascal
tregister = ( { general purpose registers }
    R_NO, R_D0, R_D1, R_D2, R_D3, R_D4, R_D5, R_D6, R_D7,
    { address registers }
    R_A0, R_A1, R_A2, R_A3, R_A4, R_A5, R_A6, R_SP,
    { PUSH/PULL- quick and dirty hack }
    R_SPPUSH, R_SPPULL,
    { misc. and floating point registers }
    R_CCR, R_FP0, R_FP1, R_FP2, R_FP3, R_FP4, R_FP5, R_FP6,
    R_FP7, R_FPCR, R_SR, R_SSP, R_DFC, R_SFC, R_VBR, R_FPSR,
    { other - not used }
    R_DEFAULT_SEG
);```

integer registers

intregs: array[1..maxintregs] of tregister;

General purpose registers which can contain any data, usually integer values. These can also be used, when no floating point coprocessor is present, to hold values for floating point operations.
9.4 Special registers (cpubase.pas)

address registers

addrregs: array[1..maxaddrregs] of tregister;

Registers which are used to construct assembler address expressions, usually the address registers are used as the base registers in these assembler expressions.

fpu registers

FpuRegs: array[1..MaxFpuRegs] of TRegister;

Hardware floating point registers. These registers must at least be able to load and store IEEE DOUBLE floating point values, otherwise they cannot be considered as FPU registers. Not available on systems with no floating point coprocessor.

scratch registers

Scratch_Regs: array[1..MaxScratchRegs] of TRegister;

These registers are used as scratch, and can be used in assembler statement in the pascal code, without being saved. They will always be valid across routine calls. These registers are sometimes temporarily allocated inside code generator nodes, and then immediately freed (always inside the same routine).

9.4 Special registers (cpubase.pas)

The code generator has special uses for certain types of registers. These special registers are of course CPU dependant, but as an indication, the following sections explains the uses of these special registers and their defines.

Stack_Pointer

Const Stack_Pointer = R_A7

This represents the stack pointer, an address register pointing to the allocated stack area.

Frame_Pointer

Const Frame_Pointer = R_A6

This represents the frame register which is used to access values in the stack. This is usually also an address register.
9.5 Instructions

9.6 Reference subsystem

Architecture

As described before in the locations section, one of the possible locations for an operand is a memory location, which is described in a special structure `reference` (described earlier). This subsection describes the interface available by the code generator for allocation and freeing reference locations.

Code generator interface

DisposeReference

Declaration: `Procedure DisposeReference(Var R : pReference);`

Description: Disposes of the reference \( R \) and sets \( r \) to NIL

Notes: Does not verify if \( R \) is assigned first.
NewReference

Declaration: Function NewReference(Const R : TReference) : pReference;
Description: Allocates in the heap a copy of the reference \( r \) and returns that allocated pointer.

Del_Reference

Declaration: Procedure Del_Reference(Const Ref : tReference);
Description: Free up all address registers allocated in this reference for the index and base (if required).
Notes: Does not free the reference symbol if it exists.

New_Reference

Declaration: Function New_Reference(Base : TRegister; Offset : Longint) : PReference;
Description: Allocates a reference pointer, clears all the fields to zero, and sets the offset to the offset field
and the base to the base fields of the newly allocated reference. Returns this newly allocated reference.

Reset_Reference

Declaration: Procedure Reset_Reference(Var Ref : TReference);
Description: Clears all fields of the reference.

9.7 The register allocator subsystem

Architecture

This system allocates and deallocates registers, from a pool of free registers. Each time the code generator requires a register for generating assembler instructions, it either calls the register allocator subsystem to get a free register or directly uses the scratch registers (which are never allocated in a pool except in the optimization phases of the compiler).

The code generator when no longer referencing the register should deallocate it so it can be used once again.

Code generator interface (tgen.pas)

The following interface routines are used by the code generator to allocate and deallocate registers from the different register pools available to code generator.
9.7 The register allocator subsystem

**GetRegister32**

**Declaration:** Function GetRegister32 : TRegister;

**Description:** Allocates and returns a general purpose (integer) register which can be used in the code generator. The register, when no longer used should be deallocated with ungetregister32() or ungetregister().

**Notes:** On non-32-bit machines, this routine should return the normal register for this machine (eg: 64-bit machines will allocate and return a 64-bit register).

**GetRegisterPair**

**Declaration:** Procedure GetRegisterPair(Var Low, High : TRegister);

**Description:** Returns a register pair to be used by the code generator when accessing 64-bit values on 32-bit wide register machines.

**Notes:** On machines which support 64-bit registers naturally, this routine should never be used, it is intended for 32-bit machines only. Some machines support 64-bit integer operations using register 32-bit pairs in hardware, but the allocated registers must be specific, this routine is here to support these architectures.

**UngetRegister32**

**Declaration:** Procedure UnGetRegister32(R : TRegister);

**Description:**Deallocates a general purpose register which was previously allocated with GetRegister32().

**GetFloatRegister**

**Declaration:** Function GetFloatRegister : TRegister;

**Description:** Allocates and returns a floating point register which can be used in the code generator. The register, when no longer used should be deallocated with ungetregister(). The register returned is a true floating point register (if supported).

**Notes:** This routine should only be used when floating point hardware is present in the system. For emulation of floating point, the general purpose register allocator/deallocator routines should be used instead.
IsFloatsRegister

Declaration: Function IsFloatsRegister(R : TRegister): Boolean;

Description: Returns TRUE if the register r is actually a floating point register, otherwise returns FALSE. This is used when the location is LOC_FPU on machines which do not support true floating point registers.

GetAdressReg

Declaration: Function GetAddressReg: TRegister;

Description: Allocates and returns an address register which can be used for address related opcodes in the code generator. The register, when no longer used should be deallocated with ungetregister()

Notes: If there is no distinction between address registers, and general purpose register in the architecture, this routine may simply call and return the getregister32() result.

IsAddressRegister

Declaration: Function IsAddressRegister(r : TRegister): Boolean;

Description: Returns TRUE if the register r is actually an address register, otherwise returns FALSE.

Notes: If there is no distinction between address registers, and general purpose register in the architecture, this routine may simply verify if this is a general purpose register and return TRUE in that case.

UngetRegister

Declaration: Procedure UngetRegister(r : TRegister);

Description: Deallocates any register which was previously allocated with any of the allocation register routines.

SaveUsedRegisters

Declaration: Procedure SaveUsedRegisters(Var Saved: TSaved; ToSave: TRegisterset);

Description: Saves in a temporary location all specified registers. On stack based machines the registers are saved on the stack, otherwise they are saved in a temporary memory location. The registers which were saved are stored in the saved variable. The constant ALL_REGISTERS passed to the tosave parameter indicates to save all used registers.
9.8 Temporary memory allocator subsystem

**RestoreUsedRegisters**

**Declaration:** procedure restoreusedregisters(Saved : TSaved);

**Description:** Restores all saved registers from the stack (or a temporary memory location). Free any temporary memory space allocated, if necessary.

**GetExplicitRegister32**

**Declaration:** Function GetExplicitRegister32(R : TRegister): TRegister;

**Description:** This routine allocates specifically the specified register \( r \) and returns that register. The register to allocate can only be one of the scratch registers.

**Notes:** This routine is used for debugging purposes only. It should be used in conjunctions with UnGetRegister32() to explicitly allocate and deallocate a scratch register.

### 9.8 Temporary memory allocator subsystem

**Architecture**

Sometimes it is necessary to reserve temporary memory locations on the stack to store intermediate results of statements. This is done by the temporary management module.

Since entry and exit code for routines are added after the code for the statements in the routine have been generated, temporary memory allocation can be used ‘on the fly’ in the case where temporary memory values are required in the code generation phase of the routines being compiled. After usage, the temporary memory space should be freed, so it can be reused if necessary.

The temporary memory allocation is a linked list of entries containing information where to access the data via a negative offset from the Frame_Pointer register. The linked list is only valid when compiling and generating the code for the procedure bodies; it is reset and cleared each time a new routine is compiled. There are currently three different types of memory spaces in use: volatile (tt_Normal) which can be allocated and freed any time in the procedure body, ansistring, which is currently the same as volatile, except it only stored references to ansistring’s, and persistent (tt_Persistent) which are memory blocks which are reserved throughout the routine duration; persistent allocated space can never be reused in a procedure body, unless explicitly released.

The temporary memory allocator guarantees to allocate memory space on the stack at least on a 16-bit alignment boundary. The exact alignment depends on the operating system required alignment.
9.8 Temporary memory allocator subsystem

Temporary memory allocator interface (temp_gen.pas)

GetTempOfSize

Declaration: Function GetTempOfSize(Size : Longint) : Longint;

Description: Allocates at least \texttt{size} bytes of temporary volatile memory on the stack. The return value is the negative offset from the frame pointer where this memory was allocated.

Notes: The return offset always has the required alignment for the target system, and can be used as an offset from the Frame_Pointer to access the temporary space.

GetTempOfSizeReference

Declaration: Procedure GetTempOfSizeReference(L : Longint; Var Ref : TReference);

Description: This routine is used to assign and allocate extra temporary volatile memory space on the stack from a reference. \texttt{L} is the size of the persistent memory space to allocate, while \texttt{Ref} is a reference entry which will be set to the correct offset from the Frame_Pointer register base. The \texttt{Offset} and \texttt{Base} fields of \texttt{Ref} will be set appropriately in this routine, and can be considered valid on exit of this routine.

Notes: The return offset always has the required alignment for the target system.

UnGetIfTemp

Declaration: Procedure UnGetIfTemp(Const Ref : TReference);

Description: Frees a reference \texttt{Ref} which was allocated in the volatile temporary memory space.

Notes: The freed space can later be reallocated and reused.

GetTempAnsiStringReference

Declaration: Procedure GetTempAnsiStringReference(Var Ref : TReference);

Description: Allocates \texttt{Ref} on the volatile memory space and sets the \texttt{Base} to the Frame_Pointer register and \texttt{Offset} to the correct offset to access this allocated memory space.

Notes: The return offset always has the required alignment for the target system.
9.8 Temporary memory allocator subsystem

**GetTempOfSizePersistant**

**Declaration:** Function GetTempOfSizePersistant(Size : Longint) : Longint;

**Description:** Allocates persistent storage space on the stack. return value is the negative offset from the frame pointer where this memory was allocated.

**Notes:** The return offset always has the required alignment for the target system.

**UngetPersistantTemp**

**Declaration:** Procedure UnGetPersistantTemp(Pos : Longint);

**Description:** Frees space allocated as being persistent. This persistent space can then later be used and reallocated. Pos is the offset relative to the Frame_Pointer of the persistent memory block to free.

**ResetTempGen**

**Declaration:** Procedure ResetTempGen;

**Description:** Clear and free the complete linked list of temporary memory locations. The list is set to nil.

**Notes:** This routine is called each time a routine has been fully compiled.

**SetFirstTemp**

**Declaration:** Procedure SetFirstTemp(L : Longint);

**Description:** This routine sets the start of the temporary local area (this value is a negative offset from the Frame_Pointer, which is located after the local variables). Usually the start offset is the size of the local variables, modified by any alignment requirements.

**Notes:** This routine is called once before compiling a routine, it indicates the start address where to allocate temporary memory space.

**GetFirstTempSize**

**Declaration:** Function GetFirstTempSize : Longint;

**Description:** Returns the total number of bytes allocated for local and temporary allocated stack space. This value is aligned according to the target system alignment requirements, even if the actual size is not aligned.

**Notes:** This routine is used by the code generator to get the total number of bytes to allocate locally (i.e the stackframe size) in the entry and exit code of the routine being compiled.
9.9 Assembler generation

**NormalTempToPersistant**

**Declaration:** Procedure NormalTempToPersistant(Pos : Longint);

**Description:** Searches the list of currently temporary memory allocated for the one with the offset Pos, and if found converts this temporary memory space as persistent (can never be freed and reallocated).

**PersistantTempToNormal**

**Declaration:** Procedure PersistantTempToNormal(Pos : Longint);

**Description:** Searches the list of currently allocated persistent memory space as the specified address Pos, and if found converts this memory space to normal volatile memory space which can be freed and reused.

**IsTemp**

**Declaration:** Function IsTemp(const Ref : TReference): Boolean;

**Description:** Returns TRUE if the reference ref is allocated in temporary volatile memory space, otherwise returns FALSE.

9.9 Assembler generation

**Architecture**

The different architectures on the market today only support certain types of operands as assembler instructions. The typical format of an assembler instruction has the following format:

```
OPCODE [opr1,opr2[,opr3][...]]
```

The opcode field is a mnemonic for a specific assembler instruction, such as MOV on the 80x86, or ADDX on the 680x0. Furthermore, in most cases, this mnemonic is followed by zero to three operands which can be of the following types:

**Possible Operand Types**

- a LABEL or SYMBOL (to code or data)
- a REGISTER (one of the predefined hardware registers)
- a CONSTANT (an immediate value)
• a MEMORY EXPRESSION (indirect addressing through offsets, symbols, and address registers)

In the compiler, this concept of different operand types has been directly defined for easier generation of assembler output. All opcodes generated by the code generator are stored in a linked list of opcodes which contain information on the operand types. The opcode and the size (which is important to determine on what size the operand must be operated on) are stored in that linked list.

The possible operand sizes for the code generator are as follows (a enumeration of type top-size):

<table>
<thead>
<tr>
<th>Operand size enum (topsize)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_B</td>
<td>8-bit integer operand</td>
</tr>
<tr>
<td>S_W</td>
<td>16-bit integer operand</td>
</tr>
<tr>
<td>S_L</td>
<td>32-bit integer operand</td>
</tr>
<tr>
<td>S_Q</td>
<td>64-bit integer operand</td>
</tr>
<tr>
<td>S_FS</td>
<td>32-bit IEEE 754 Single floating point operand</td>
</tr>
<tr>
<td>S_FL</td>
<td>64-bit IEEE 754 Double floating point operand</td>
</tr>
<tr>
<td>S_FX</td>
<td>Extended point floating point operand (cpu-specific)</td>
</tr>
<tr>
<td>S_CPU</td>
<td>A constant equal to one of the previous sizes (natural size of operands)</td>
</tr>
</tbody>
</table>

The possible operand types for the code generator are as follows (other might be added as required by the target architecture):

<table>
<thead>
<tr>
<th>Operand type (TOpType)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_None</td>
<td>No operand</td>
</tr>
<tr>
<td>top_Reg</td>
<td>Operand is a register</td>
</tr>
<tr>
<td>top_Ref</td>
<td>Operand is a reference (treference type)</td>
</tr>
<tr>
<td>top_Symbol</td>
<td>Operand is a symbol (reference or label)</td>
</tr>
</tbody>
</table>

The architecture specific opcodes are done in an enumeration of type tasmop. An example of an enumeration for some of the opcodes of the PowerPC 32-bit architecture is as follows:

```c
Type TAsmOp = (a_Add, a_Add_, a_Addo, a_Addo_, a_Addc, a_Addc_, a_Addco, a_Addco_, a_Adde, a_Adde_, a_Addeo, a_Addeo_, a_Addi, a_Addic, a_Addic_, a_Addis) |
```

Generic instruction generation interface

To independently generate code for different architectures, wrappers for the most used instructions in the code generator have been created which are totally independent of the target
9.9 Assembler generation

system.

**Emit_Load_Loc_Reg**

**Declaration:** Procedure Emit_Load_Loc_Reg(Src:TLocation;Srcdef:pDef; DstDef : pDef; Dst : TRegister);

**Description:** Loads an operand from the source location in Src into the destination register Dst taking into account the source definition and destination definition (sign-extension, zero extension depending on the sign and size of the operands).

**Notes:** The source location can only be in LOC_REGISTER, LOC_CREGISTER, LOC_MEM or LOC_REFERENCE otherwise an internal error will occur. This generic opcode does not work on floating point values, only integer values.

**FloatLoad**

**Declaration:** Procedure FloatLoad(t : tFloatType;Ref : TReference; Var Location:TLocation);

**Description:** This routine is to be called each time a location must be set to LOC_FPU and a value loaded into a FPU register.

**Notes:** The routine sets up the register field of LOC_FPU correctly. The source location can only be LOC_MEM or LOC_REFERENCE. The destination location is set to LOC_FPU.

**FloatStore**

**Declaration:** Procedure FloatStore(t : TFloatType;Var Location:TLocation; Ref:TReference);

**Description:** This routine is to be called when a value located in LOC_FPU must be stored into memory.

**Notes:** The destination must be LOC_REFERENCE or LOC_MEM. This routine frees the LOC_FPU location.

**emit_mov_ref_reg64**

**Declaration:** Procedure Emit_Mov_Ref_Reg64(r : TReference;rl,rh : TRegister);

**Description:** This routine moves a 64-bit integer value stored in memory location r into the low 32-bit register rl and the high 32-bit register rh.
9.9 Assembler generation

**Emit_Lea_Loc_Ref**

Declaration: Procedure Emit_Lea_Loc_Ref(Const t:TLocation;Const Ref:TReference;FreeTemp:Boolean);

Description: Loads the address of the location loc and stores the result into Ref

Notes: The store address ref should point to an allocated area at least sizeof(pointer) bytes, otherwise unexpected code might be generated.

**Emit_Lea_Loc_Reg**

Declaration: Procedure Emit_Lea_Loc_Reg(const t:TLocation;Reg:TRegister;Freetemp:Boolean);

Description: Loads the address of the location loc and stores the result into the target register reg

**GetLabel**

Declaration: Procedure GetLabel(Var l : pAsmLabel);

Description: Returns a label associated with code. This label can then be used with the instructions output by the code generator using the instruction generation templates which require labels as parameters. The label itself can be emitted to the assembler source by calling the EmitLab routine.

**EmitLab**

Declaration: Procedure EmitLab(Var l : pAsmLabel);

Description: Output the label l to the assembler instruction stream.

Notes: The label should have been previously allocated with GetLabel. The output label will be of the form label: in the instruction stream. This label is usually a jump target.

**EmitLabeled**

Declaration: Procedure EmitLabeled(op : TAsmOp; Var l : pAsmLabel);

Description: Output the opcode op with the operand l which is a previously allocated label.

Notes: This routine is used to output jump instructions such as : jmp label, jne label. The label should have been previously allocated with a call to GetLabel.
**EmitCall**

**Declaration:** Procedure EmitCall(Const Routine:String);

**Description:** Emit a call instruction to an internal routine

**Parameters:** Routine = The name of the routine to call.

**ConcatCopy**

**Declaration:** procedure ConcatCopy(Source,Dest : TReference;Size : Longint;DelSource : Boolean; loadref:boolean);

**Description:** This routine copies Size data from the Source reference to the destination Dest reference.

**Parameters:** Source = Source reference to copy from

Dest = Depending on the value of loadref, either indicates a location where a pointer to the data to copy is Stored, or this reference directly the address to copy to.

Size = Number of bytes to copy

DelSource = TRUE if the source reference should be freed in this routine

LoadRef = TRUE if the source reference contains a pointer to the address we wish to copy to, otherwise the reference itself is the destination location to copy to.

**Emit_Flag2Reg**

**Declaration:** Procedure Emit_Flag2Reg(Flag:TResflags;HRegister:TRegister);

**Description:** Sets the value of the register to 1 if the condition code flag in Flag is TRUE, otherwise sets the register to zero.

**Notes:** The operand should be zero extended to the natural register size for the target architecture.

**10 The assembler output**

All code is generated via special linked lists of instructions. The base of this is a special object, an abstract assembler which implements all directives which are usually implemented in the different assemblers available on the market. When the code generator and parser generates the final output, it is generated as a linked list for each of the sections available for the output assembler. Each entry in the linked list is either an instruction, or one of the abstract directives for the assembler.
Figure 11: Assembler generation organisation
The different possible sections which are output are as follows:

<table>
<thead>
<tr>
<th>Internal section name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExparAsmList</td>
<td>temporary list</td>
</tr>
<tr>
<td>DataSegment</td>
<td>initialized variables</td>
</tr>
<tr>
<td>CodeSegment</td>
<td>instructions and general code directives</td>
</tr>
<tr>
<td>DebugList</td>
<td>debugging information</td>
</tr>
<tr>
<td>WithDebugList</td>
<td>?????????????????</td>
</tr>
<tr>
<td>Consts</td>
<td>read only constants</td>
</tr>
<tr>
<td>ImportSection</td>
<td>imported symbols</td>
</tr>
<tr>
<td>ExportSection</td>
<td>exported symbols</td>
</tr>
<tr>
<td>ResourceSection</td>
<td>Resource data</td>
</tr>
<tr>
<td>RttiList</td>
<td>Resource data</td>
</tr>
<tr>
<td>ResourceStringList</td>
<td>runtime type information data</td>
</tr>
</tbody>
</table>

The following directives for the abstract assembler currently exist:

Abstract assembler node types:

<table>
<thead>
<tr>
<th>Node entry Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ait_None</td>
<td>This entry in the linked list is invalid (this should normally never occur)</td>
</tr>
<tr>
<td>Ait_Direct</td>
<td>Direct output to the resulting assembler file (as string)</td>
</tr>
<tr>
<td>Ait_String</td>
<td>Shortstring with a predefined length</td>
</tr>
<tr>
<td>Ait_Label</td>
<td>Numbered assembler label used for jumps</td>
</tr>
<tr>
<td>Ait_Comment</td>
<td>Assembler output comment</td>
</tr>
<tr>
<td>Ait_Instruction</td>
<td>Processor specific instruction</td>
</tr>
<tr>
<td>Ait_DataBlock</td>
<td>Initialized data block (BSS)</td>
</tr>
<tr>
<td>Ait_Symbol</td>
<td>Entry represents a symbol (exported, imported, or other public symbol type)</td>
</tr>
<tr>
<td></td>
<td>Possible symbol types : NONE, EXTERNAL, LOCAL and GLOBAL</td>
</tr>
<tr>
<td>Ait_Symbol_End</td>
<td>Symbol end (for example the end of a routine)</td>
</tr>
<tr>
<td>Ait_Const_32bit</td>
<td>Initialized 32-bit constant (without a symbol)</td>
</tr>
<tr>
<td>Ait_Const_16bit</td>
<td>Initialized 16-bit constant (without a symbol)</td>
</tr>
<tr>
<td>Ait_Const_8bit</td>
<td>Initialized 8-bit constant (without a symbol)</td>
</tr>
<tr>
<td>Ait_Const_symbol</td>
<td>?????????????</td>
</tr>
<tr>
<td>Ait_Real_80bit (x86)</td>
<td>Initialized 80-bit floating point constant (without symbol)</td>
</tr>
</tbody>
</table>
### Node entry Type

<table>
<thead>
<tr>
<th>Node entry Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ait_Real_64bit</td>
<td>Initialized Double IEEE floating point constant (without symbol)</td>
</tr>
<tr>
<td>Ait_Real_32bit</td>
<td>Initialized Single IEEE floating point constant (without symbol)</td>
</tr>
<tr>
<td>Ait_Comp_64bit (x86)</td>
<td>Initialized 64-bit floating point integer (without symbol)</td>
</tr>
<tr>
<td>Ait_Align</td>
<td>Alignment directive</td>
</tr>
<tr>
<td>Ait_Section</td>
<td>Section directive</td>
</tr>
<tr>
<td>Ait_const_rva (Win32)</td>
<td>stabs debugging information (numerical value)</td>
</tr>
<tr>
<td>Ait_Stabn</td>
<td>stabs debugging information (string)</td>
</tr>
<tr>
<td>Ait_Stabs</td>
<td>stabs debugging line information</td>
</tr>
<tr>
<td>Ait_Force_Line</td>
<td>stabs debug information routine name</td>
</tr>
<tr>
<td>Ait_Stab_Function_Name</td>
<td>Cut in the assembler files (used for smartlinking)</td>
</tr>
<tr>
<td>Ait_Cut</td>
<td>Debugging information for the register allocator</td>
</tr>
<tr>
<td>Ait_RegAlloc</td>
<td>????????????</td>
</tr>
<tr>
<td>Ait_Marker</td>
<td>Unused - should never appear</td>
</tr>
</tbody>
</table>

### Define Name

<table>
<thead>
<tr>
<th>Define Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386</td>
<td>Intel 80x86 family of processors (and compatibles)</td>
</tr>
<tr>
<td>m68k</td>
<td>Motorola 680x0 family of processors (excludes coldfire)</td>
</tr>
<tr>
<td>alpha</td>
<td>Alpha 21x64 family of processors</td>
</tr>
<tr>
<td>powerpc</td>
<td>Motorola / IBM 32-bit family of processors</td>
</tr>
<tr>
<td>sparc</td>
<td>SPARC v7 compatible processors</td>
</tr>
</tbody>
</table>

### Define name

<table>
<thead>
<tr>
<th>Define name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTLLITE</td>
<td>Removes some extraneous routine from compilation (system unit is minimal). Mvdv: Afaik the status of this is unknown</td>
</tr>
<tr>
<td>DEFAULT_EXTENDED</td>
<td>The runtime library routines dealing with fixed point values have the <strong>extended</strong> type instead of the <strong>real</strong> type.</td>
</tr>
</tbody>
</table>

---

### 11 The Runtime library

This section describes the requirements of the internal routines which MUST be implemented for all relevant platforms to port the system unit to a new architecture or operating system.

The following defines are available when compiling the runtime library:
The compiler supports the single floating point precision type.
The compiler supports the double floating point precision type.
The compiler supports the extended floating point precision type.
The compiler supports the fixed floating point precision type.
The compiler supported the widechar character type.
The compiler supports 64-bit integer operations.
Text I/O uses Mac styled line breaks (#13) instead of #13#10.
Text I/O uses UNIX styled line breaks (#10) instead of #13#10.
A Ctrl-Z character in a text file is an EOF marker (UNIX mostly).

<table>
<thead>
<tr>
<th>Define name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUPPORT_SINGLE</td>
<td>The compiler supports the single floating point precision type</td>
</tr>
<tr>
<td>SUPPORT_DOUBLE</td>
<td>The compiler supports the double floating point precision type</td>
</tr>
<tr>
<td>SUPPORT_EXTENDED</td>
<td>The compiler supports the extended floating point precision type</td>
</tr>
<tr>
<td>SUPPORT_FIXED</td>
<td>The compiler supports the fixed floating point precision type</td>
</tr>
<tr>
<td>HASWIDECHAR</td>
<td>The compiler supported the widechar character type</td>
</tr>
<tr>
<td>INT64</td>
<td>The compiler supports 64-bit integer operations</td>
</tr>
<tr>
<td>MAC_LINEBREAK</td>
<td>Text I/O uses Mac styled line break (#13) instead of #13#10</td>
</tr>
<tr>
<td>SHORT_LINEBREAK</td>
<td>Text I/O uses UNIX styled line breaks (#10) instead of #13#10</td>
</tr>
<tr>
<td>EOF_CTRLZ</td>
<td>A Ctrl-Z character in a text file is an EOF marker (UNIX mostly)</td>
</tr>
</tbody>
</table>

The following defines are used for fexpand definitions:

<table>
<thead>
<tr>
<th>Define name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPC_EXPAND_DRIVES</td>
<td>Different devices with different names (as drives) are supported</td>
</tr>
<tr>
<td></td>
<td>(like DOS, Netware, etc...)</td>
</tr>
<tr>
<td>FPC_EXPAND_UNC</td>
<td>Universal Naming convention support i.e</td>
</tr>
<tr>
<td></td>
<td>\ \ &lt;server-name&gt;&lt;share-name&gt;&lt;directory/filename&gt;</td>
</tr>
<tr>
<td>UNIX</td>
<td>Unix style file names</td>
</tr>
<tr>
<td>FPC_EXPAND_VOLUMES</td>
<td>Volume names (i.e. drive descriptions longer than 1 character) are supported.</td>
</tr>
<tr>
<td>FPC_EXPAND_TILDE</td>
<td>Replaces the ~ character, with the ‘HOME’ directory (mostly on UNIX platforms)</td>
</tr>
</tbody>
</table>

The following defines some debugging routines for the runtime library:

<table>
<thead>
<tr>
<th>Define Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE NAME</td>
<td>Description</td>
</tr>
<tr>
<td>ANSISTRDEBUG</td>
<td>Add Debug routines for ansi string support</td>
</tr>
<tr>
<td>EXCDEBUG</td>
<td>Add Debug routines for exception debugging</td>
</tr>
<tr>
<td>LOGGING</td>
<td>Log the operations to a file</td>
</tr>
</tbody>
</table>

85
11.1 Operating system hooks

This section contains information on all routines which should be hooked and implemented to be able to compile and use the system unit for a new operating system:

System_Exit

Declaration: Procedure System_Exit;

Description: This routine is internally called by the system unit when the application exits.

Notes: This routine should actually exit the application. It should exit with the error code specified in the ExitCode variable.

Algorithm: Exit application with ExitCode value.

ParamCount

Declaration: Function ParamCount : Longint;

Description: This routine is described in the Free Pascal reference manual.

Randomize

Declaration: Procedure Randomize;

Description: This routine should initialize the built-in random generator with a random value.

Notes: This routine is used by random

Algorithm: Randseed := pseudo random 32-bit value

GetHeapStart

Declaration: Function GetHeapStart : Pointer;

Description: This routine should return a pointer to the start of the heap area.

Algorithm: GetHeapStart := address of start of heap.

GetHeapSize

Declaration: Function GetHeapSize : Longint;

Description: This routine should return the total heap size in bytes

Algorithm: GetHeapSize := total size of the initial heap area.
11.1 Operating system hooks

sbrk

Declaration: Function Sbrk(Size : Longint): Longint;

Description: This routine should grow the heap by the number of bytes specified. If the heap cannot be
grown it should return -1, otherwise it should return a pointer to the newly allocated area.

Parameters: size = Number of bytes to allocate

Do_Close

Declaration: Procedure Do_Close(Handle : Longint);

Description: This closes the file specified of the specified handle number.

Parameters: handle = file handle of file to close

Notes: This routine should close the specified file.

Notes: This routine should set InoutRes in case of error.

Do_Erase

Declaration: Procedure Do_Erase(p: pChar);

Description: This erases the file specified by p.

Parameters: p = name of the file to erase

Notes: This routine should set InoutRes in case of error.

Do_Truncate

Declaration: Procedure Do_Truncate(Handle, FPos : Longint);

Description: This truncates the file at the specified position.

Parameters: handle = file handle of file to truncate fpos = file position where the truncate should occur

Notes: This routine should set InoutRes in case of error.
11.1 Operating system hooks

Do_Rename

Declaration: Procedure Do_Rename(p1, p2 : pchar);

Description: This renames the file specified.

Parameters: p1 = old file name p2 = new file name

Notes: This routine should set InoutRes in case of error.

Do_Write

Declaration: Function Do_Write(Handle, Addr, Len: Longint): longint;

Description: This writes to the specified file. Returns the number of bytes actually written.

Parameters: handle = file handle of file to write to addr = address of buffer containing the data to write len = number of bytes to write

Notes: This routine should set InoutRes in case of error.

Do_Read

Declaration: Function Do_Read(Handle, Addr, Len: Longint): Longint;

Description: Reads from a file. Returns the number of bytes read.

Parameters: handle = file handle of file to read from addr = address of buffer containing the data to read len = number of bytes to read

Notes: This routine should set InoutRes in case of error.

Do_FilePos

Declaration: function Do_FilePos(Handle: Longint): longint;

Description: Returns the file pointer position

Parameters: handle = file handle of file to get file position on

Notes: This routine should set InoutRes in case of error.
11.1 Operating system hooks

Do_Seek

Declaration: Procedure Do_Seek(Handle, Pos: Longint);

Description: Set file pointer of file to a new position

Parameters: handle = file handle of file to seek in pos = new position of file pointer (from start of file)

Notes: This routine should set InoutRes in case of error.

Do_Seekend

Declaration: Function Do_SeekEnd(Handle: Longint): Longint;

Description: Seeks to the end of the file. Returns the new file pointer position.

Parameters: handle = file handle of file to seek to end of file

Notes: This routine should set InoutRes in case of error.

Do_FileSize

Declaration: Function Do_FileSize(Handle: Longint): Longint;

Description: Returns the filesize in bytes.

Parameters: handle = file handle of file to get the file size

Notes: This routine should set InoutRes in case of error.

Do_IsDevice

Declaration: Function Do_IsDevice(Handle: Longint): boolean;

Description: Returns TRUE if the file handle points to a device instead of a file.

Parameters: handle = file handle to get status on

Notes: This routine should set InoutRes in case of error.
11.1 Operating system hooks

Do_Open

Declaration: Procedure Do_Open(var f;p:pchar;flags:longint);

Description: Opens a file in the specified mode, and sets the mode and handle fields of the f structure parameter.

Parameters: f = pointer to textrec or filerec structure p = name and path of file to open flags = access mode to open the file with

Notes: This routine should set InoutRes in case of error.

ChDir

Declaration: Procedure ChDir(Const s: String);[IOCheck];

Description: Changes to the specified directory. . and .. should also be supported by this call.

Parameters: s = new directory to change to

Notes: This routine should set InoutRes in case of error.

MkDir

Declaration: Procedure MkDir(Const s: String);[IOCheck];

Description: Creates the specified directory.

Parameters: s = name of directory to create

Notes: This routine should set InoutRes in case of error.

RmDir

Declaration: Procedure RmDir(Const s: String);[IOCheck];

Description: Removes the specified directory.

Parameters: s = name of directory to remove

Notes: This routine should set InoutRes in case of error.

The following variables should also be defined for each new operating system, they are used by external units:

argc : The number of command line arguments of the program
argv : A pointer to each of the command line arguments (an array of pchar pointers)
11.2 CPU specific hooks

The following routines must absolutely be implemented for each processor, as they are dependent on the processor:

**FPC_SETJMP**

*SetJmp*

**Declaration:** Function SetJmp (Var S : Jmp_Buf) : Longint;

**Description:** A call to SetJmp(), saves the calling environment in its $s$ argument for later use by longjmp(). Called by the code generator in exception handling code. The return value should be zero.

**Notes:** This routine should save / restore all used registers (except the accumulator which should be cleared).

**FPC_LONGJMP**

*function SPtr()*

*function Get_Caller_Frame(framebp:longint):longint;*

*function Get_Caller_Addr(framebp:longint):longint;*

*function Get_Frame:longint;*

*function Trunc()*

11.3 String related

**FPC_SHORTSTR_COPY**

*Int_StrCopy*

**Declaration:** Procedure Int_StrCopy(Len:Longint;SStr,DStr:pointer);

**Description:** This routine copies the string pointed to by the address in sstr, to the string pointed in the destination. The old string is overwritten, and the source string will be truncated to make it fit in destination if the length of the source is greater then destination string len (the len parameter).

**Parameters:** Len = maximum length to copy (the destination string length)
SStr = pointer to source shortstring
DStr = point to destination shortstring
Notes: Called by code generator when a string is assigned to another string.

**FPC_SHORTSTR_COMPARE**

**Int_StrCmp**

**Declaration:** Function Int_StrCmp(DStr, SStr: Pointer) : Longint;

**Description:** The routine compares two shortstrings, and returns 0 if both are equal, 1 if DStr is greater then SSrc, otherwise it returns –1.

**Notes:** Both pointers must point to shortstrings. Length checking must be performed in the routine.

**FPC_SHORTSTR_CONCAT**

**Int_StrConcat**

**Declaration:** Procedure Int_StrConcat(Src, Dest: Pointer);

**Description:** This routine appends the string pointed to by Src to the end of the string pointed to by Dest.

**Parameters:**
- Src = pointer to shortstring to append to dest
- Dest = pointer to shortstring to receive appended string

**Notes:** Both pointers must point to shortstrings. In the case where the src string length does not fit in dest, it is truncated.

**Algorithm:**

```
if src = nil or dest = nil then
  exit routine;
if (src string length + dest string length) > 255 then
  number of bytes to copy = 255 -- dest string length
else
  number of bytes to copy = src string length;
  copy the string data (except the length byte)
dest string length = dest string length + number of bytes to copied
```

**FPC_ANSISTR_CONCAT**

**AnsiStr_Concat**

**Declaration:** Procedure AnsiStr_Concat(s1, s2: Pointer; var s3: Pointer);

**Description:** This routine appends s1+s2 and stores the result at the address pointed to by s3.

**Notes:** All pointers must point to anstrings.
11.3 String related

FPC_ANSISTR_COMPARE

AnsiStr_Compare

Declaration: Function AnsiStr_Compare(s1, s2 : Pointer): Longint;

Description: The routine compares two ansistrings, and returns 0 if both are equal, 1 if s1 is greater than s2, otherwise it returns -1.

Parameters: Both pointers must point to ansistrings.

FPC_ANSISTR_INCR_REF

AnsiStr_Incr_Ref

Declaration: procedure AnsiStr_Incr_Ref (var s : Pointer);

Description: This routine simply increments the ANSI string reference count, which is used for garbage collection of ANSI strings.

Parameters: s = pointer to the ansi string (including the header structure)

FPC_ANSISTR_DECR_REF

AnsiStr_Decr_Ref

Declaration: procedure AnsiStr_Decr_Ref (Var S : Pointer);

Parameters: s = pointer to the ansi string (including the header structure)

Algorithm: Decreases the internal reference count of this non constant ansi string; If the reference count is zero, the string is deallocated from the heap.

FPC_ANSISTR_ASSIGN

AnsiStr_Assign

Declaration: Procedure AnsiStr_Assign (var s1 : Pointer; s2 : Pointer);

Parameters: s1 = address of ANSI string to be assigned to
            s2 = address of ANSI string which will be assigned

Algorithm: Assigns S2 to S1 (S1:=S2), also by the time decreasing the reference count to S1 (it is no longer used by this variable).
11.3 String related

**FPC_PCHAR_TO_SHORTSTR**

**StrPas**

**Declaration:** Function StrPas(p:pChar):ShortString;

**Description:** Copies and converts a null-terminated string (pchar) to a shortstring with length checking.

**Parameters:** p = pointer to null terminated string to copy

**Notes:** Length checking is performed. Verifies also p=nil, and if so sets the shortstring length to zero. Called by the type conversion generated code of code generator.

**Algorithm:**

if p=nil then
  string length = 0
else
  string length = string length(p)
if string length>255 then
  string length = 255
if string length>0 then
  Copy all characters of pchar array to string (except length byte)

**FPC_SHORTSTR_TO_ANSISTR**

**FPC_ShortStr_To_AnsiStr**

**Notes:** Called by the type conversion generated code of code generator.

**FPC_STR_TO_CHARARRAY**

**Str_To_CharArray**

**Declaration:** procedure Str_To_CharArray(StrTyp, ArraySize: Longint; src,dest: pChar);

**Description:** Converts a string to a character array (currently supports both shortstring and AnsiString types). Length checking is performed, and copies up to arraysize elements to dest.

**Parameters:** strtyp = Indicates the conversion type to do (0 = shortstring, 1 = AnsiString, 2 = longstring, 3 = widestring)
arraysize = size of the destination array
src = pointer to source string
dest = pointer to character array
Notes: Called by the type conversion generated code of code generator when converting a string to an array of char. If the size of the string is less then the size of the array, the rest of the array is filled with zeros.

**FPC_CHARARRAY_TO_SHORTSTR**

**StrCharArray**

**Declaration:** Function StrCharArray(p: pChar; l: Longint): ShortString;

**Description:** Copies a character array to a shortstring with length checking (upto 255 characters are copied)

**Parameters:**
- p = Character array pointer
- l = size of the array

**Notes:** Called by the type conversion generated code of code generator when converting an array of char to a shortstring.

**Algorithm:**

```pascal
if size of array >= 256 then
  length of string = 255
else
  if size of array < 0 then
    length of string = 0
  else
    length of string = size of array
Copy all characters from array to shortstring
```

**FPC_CHARARRAY_TO_ANSISTR**

**Fpc_Chararray_To_AnsiStr**

**Notes:** Called by the type conversion generated code of code generator when converting an array of char to an ansistring.

**FPC_CHAR_TO_ANSISTR**

**Fpc_Char_To_AnsiStr**

**Notes:** Called by the type conversion generated code of code generator when converting a char to an ansistring.
11.4 Compiler runtime checking

**FPC_PCHAR_TO_ANSISTR**

Fpc_pChar_To_AnsiStr

Notes: Called by the type conversion generated code of code generator when converting a pchar to an ansistring.

11.4 Compiler runtime checking

**FPC_STACKCHECK**

Int_StackCheck

Declaration: procedure int_stackcheck (stack_size:longint);

Description: This routine is used to check if there will be a stack overflow when trying to allocate stack space from the operating system. The routine must preserve all registers. In the case the stack limit is reached, the routine calls the appropriate error handler.

Parameters: stack_size = The amount of stack we wish to allocate

Notes: Inserted in the entry code of a routine in the {$S+} state by the code generator

Algorithm:

```
if ((StackPointer - stack_size) < System.StackLimit) then
    Throw a Runtime error with error code 202 (stack overflow)
```

**FPC_RANGEERROR**

Int_RangeError

Declaration: procedure Int_RangeError;

Description: This routine is called when a range check error is detected when executing the compiled code. This usually simply calls the default error handler, with the correct runtime error code to produce.

Parameters: Inserted in code generator when a Runtime error 201 {$R+} should be generated

**FPC_BOUNDCHECK**

Int_BoundCheck

Declaration: procedure Int_BoundCheck(l : Longint; Range : Pointer);
Description: This routine is called at runtime in $R+$ mode to check if accessing indexes in a string or array is out of bounds. In this case, the default error handler is called, with the correct runtime error code to produce.

Parameters: \( l = \) Index we need to check

- range = pointer to a structure containing the minimum and maximum allowed indexes (points to two 32-bit signed values which are the limits of the array to verify).

Notes: Inserted in the generated code after assignments, and array indexing to verify if the result of operands is within range (in the \{$R+$} state)

**FPC_OVERFLOW**

**Int_OverFlow**

Declaration: `procedure Int_OverFlow;`

Description: This routine is called when an overflow is detected when executing the compiled code. This usually simply calls the default error handler, with the correct runtime error code to produce.

Parameters: Inserted in code generator when a Runtime error 215 \{$Q+$} should be generated.

**FPC_CHECK_OBJECT**

**Int_Check_Object**

Declaration: `procedure Int_Check_Object(vmt : Pointer);`

Description: This routine is called at runtime in the $R+$ state each time a virtual method is called. It verifies that the object constructor has been called first to build the VMT of the object, otherwise it throws an Runtime error 210.

Parameters: \( vmt = \) Current value of the SELF register

Notes: Call inserted by the code generator before calling the virtual method. This routine should save / restore all used registers.

Algorithm:

```plaintext
if vmt = nil or size of method table = 0 then
  Throw a Runtime error with error code 210 (object not initialized)
```

97
11.4 Compiler runtime checking

**FPC_CHECK_OBJECT_EXT**

**Int_Check_Object_Ext**

**Declaration:** procedure Int_Check_Object_Ext(vmt, expvmt : pointer);

**Description:** This routine is called at runtime when extended object checking is enabled (on the command line) and a virtual method is called. It verifies that the object constructor has been called first to build the VMT of the object, otherwise it throws a Runtime error 210, and furthermore it check that the object is actually a descendant of the parent object, otherwise it returns a Runtime error 219.

**Parameters:**
- vmt = Current value of the SELF register
- expvmt = Pointer to TRUE object definition

**Notes:** Call inserted by the code generator before calling the virtual method.

This routine should save / restore all used registers.

**Algorithm:**

```plaintext
if vmt = nil or size of method table =0 then
  Throw a Runtime error with error code 210 (object not initialized)
Repeat
  If SELF (VMT) <> VMT Address (expvmt) Then
    Get Parent VMT Address
  Else
    Exit;
  until no more ent;
Throw a Runtime error with error code 220 (Incorrect object reference)
```

**FPC_IO_CHECK**

**Int_IOCheck**

**Declaration:** procedure Int_IOCheck(addr : longint);

**Description:** This routine is called after an I/O operation to verify the success of the operation when the code is compiled in the $I+$ state.

**Parameters:**
- addr = currently unused

**Algorithm:** Check last I/O was successful, if not call error handler.
11.5 Exception handling

**FPC_HANDLEERROR**

**HandleError**

**Declaration:** procedure HandleError (Errno : longint);

**Description:** This routine should be called to generate a runtime error either from one of the system unit routines or the code generator.

**Parameters:** Errno = Runtime error to generate

**Notes:** This routine calls the appropriate existing error handler with the specified error code.

**Algorithm:**

**FPC_ASSERT**

**Int_Assert**

**Declaration:** procedure Int_Assert(Const Msg,FName:Shortstring;LineNo,ErrorAddr:Longint);

**Description:** This routine is called by the code generator in an assert statement. When the assertion fails, this routine is called.

**Parameters:** msg = string to print
- Fname = Current filename of source
- LineNo = Current line number of source
- ErrorAddr = Address of assertion failure

**Notes:** REGISTERS NOT SAVED???????????
**FPC_PUSHEXCEPTADDR**

**PushExceptAddr**

**Declaration:** function PushExceptAddr (Ft: Longint): PJmp_buf ;

**Description:** This routine should be called to save the current caller context to be used for exception handling, usually called in the context where ANSI strings are used (they can raise exceptions), or in a try..finally or on statements to save the current context.

**Parameters:** Ft = Indicates the frame type on the stack (1= Exception frame or 2=Finalize frame)

**Algorithm:** Adds this item to the linked list of stack frame context information saved. Allocates a buffer for the jump statement and returns it.

**FPC_RERAISE**

**ReRaise**

**Declaration:** procedure ReRaise;

**Notes:** REGISTERS NOT SAVED???????????

**FPC_POPOBJECTSTACK**

**PopObjectStack**

**Declaration:** function PopObjectStack : TObject;

**Description:** This is called by the code generator when an exception occurs, it is used to retrieve the exception handler object from the context information.

**Notes:** REGISTERS NOT SAVED???????????

**FPC_POPSECONDOBJECTSTACK**

**PopSecondObjectStack**

**Declaration:** function PopSecondObjectStack : TObject;

**Description:** This is called by the code generator when a double exception occurs, it is used to retrieve the second exception handler object from the context information.

**Notes:** REGISTERS NOT SAVED????????????
11.5 Exception handling

FPC_DESTROYEXCEPTION

DestroyException

Declaration: Procedure DestroyException(o : TObject);

Description: This routine is called by the code generator after the exception handling code is complete to destroy the exception object.

Parameters: o = Exception handler object reference

Notes: REGISTERS NOT SAVED

FPC_POPADDRSTACK

PopAddrStack

Declaration: procedure PopAddrStack;

Description: Called by the code generator in the finally part of a try statement to restore the stackframe and dispose of all the saved context information.

Notes: REGISTERS NOT SAVED

FPC_CATCHES

Catches

Declaration: function Catches(Objtype : TExceptObjectClass) : TObject;

Description: This routine is called by the code generator to get the exception handler object.

Parameters: ObjType = The exception type class

Notes: REGISTERS NOT SAVED

FPC_GETRESOURCESTRING

GetResourceString

Declaration: function GetResourceString(Const TheTable: TResourceStringTable; Index : longint) : AnsiString;

Description: Called by code generator when a reference to a resource string is made. This routine loads the correct string from the resource string section and returns the found string (or ”” if not found).
Parameters: TheTable = pointer to the resource string table
Index = Index in the resource string table.

11.6 Runtime type information

FPC_DO_IS

Int_Do_Is

Declaration: Function Int_Do_Is(AClass: TClass; AObject: TObject) : Boolean;

Description: If aclass is of type aobject, returns TRUE otherwise returns FALSE.

Parameters: aclass = class type reference
aobject = Object instance to compare against

Notes: This is called by the code generator when the is operator is used.

Algorithm:

FPC_DO_AS

Int_Do_As

Declaration: Procedure Int_Do_As(AClass: TClass; AObject: TObject);

Description: Typecasts aclass as aobject, with dynamic type checking. If the object is not from the correct type class, a runtime error 219 is generated. Called by the code generator for the as statement.

Parameters: aclass = Class to typecast to
aobject = Object to typecast

FPC_INITIALIZE

Initialize

Declaration: Procedure Initialize(Data, TypeInfo: Pointer);

Description:

Parameters: data = pointer to the data to initialize
typeinfo = pointer to the type information for this data
Notes: This routine should save / restore all used registers.

Algorithm: Initializes the class data for runtime typed values

**FPC_FINALIZE**

`Finalize`

**Declaration:** procedure Finalize (Data, TypeInfo: Pointer);

**Description:** Called by code generator if and only if the reference to finalize <> nil.

**Parameters:**
- data = point to the data to finalize
- typename = Pointer to the type information of this data

**Notes:** This routine should save / restore all used registers. Finalizes and frees the heap class data for runtime typed values (decrements the reference count)

**FPC_ADDREF**

`AddRef`

**Declaration:** procedure AddRef (Data, TypeInfo: Pointer);

**Description:** Called by the code generator for class parameters (property support) of type const or value in parameters, to increment the reference count of ANSI strings.

**Notes:** This routine should save / restore all used registers. This routine can be called recursively with a very deep nesting level, an assembler implementation in suggested.

**FPC_DECREF**

`DecRef`

**Declaration:** procedure DecRef (Data, TypeInfo: Pointer);

**Description:** Called by the code generator for class parameters (property support) of type const or value parameters, to decrement the reference count of ANSI strings.

**Parameters:**

**Notes:** This routine should save / restore all used registers. This routine can be called recursively with a very deep nesting level, an assembler implementation in suggested.
11.7 Memory related

FPC_GETMEM

GetMem

Declaration: procedure GetMem(Var p:Pointer; Size: Longint);

FPC_FREEMEM

FreeMem

Declaration: Procedure FreeMem(Var P: Pointer; Size: Longint);

FPC_CHECKPOINTER

CheckPointer

Declaration: Procedure CheckPointer(p : Pointer);

Description: Called by the code generator when a pointer is referenced in heap debug mode. Verifies that the pointer actually points in the heap area.

Parameters: p = pointer to check

Notes: This routine should save /restore all used registers.

FPC_DO_EXIT

Do.Exit

Declaration: procedure Do.Exit;

Description: Called by code generator at the end of the program entry point.

Notes: Called to terminate the program

Algorithm: Call all unit exit handlers.

- Finalize all units which have a finalization section
- Print runtime error in case of error
- Call OS-dependant system_exit routine
11.7 Memory related

**FPC_ABSTRACTERROR**

**AbstractError**

*Declaration:* procedure AbstractError;

*Description:* The code generator allocates a VMT entry equal to this routine address when a method of a class is declared as being abstract. This routine simply calls the default error handler.

*Algorithm:* Throw a Runtime error with error code 211 (Abstract call)

**FPC_INITIALIZEUNITS**

**InitializeUnits**

*Declaration:*

*Description:* Called by the code generator in the main program, this is only available if an initialization section exists in one of the units used by the program.

**FPC_NEW_CLASS (assembler)**

**int_new_class**

*Description:* This routine will call the TObject.InitInstance() routine to instantiate a class (Delphi-styled class) and allocate the memory for all fields of the class.

On entry the self_register should be valid, and should point either to nil, for a non-initialized class, or to the current instance of the class. The first parameter on the top of the stack should be a pointer to the VMT table for this class(????).

**FPC_HELP_DESTRUCTOR**

Could be implemented in ASM directly with register parameter passing.

**Int_HelpDestructor**

*Declaration:* Procedure Int_HelpDestructor(Var _Self : Pointer; Vmt : Pointer; Vmt_Pos : Cardinal);

*Description:* Frees the memory allocated for the object fields, and if the object had a VMT field, sets it to nil.
### Parameters

- `self` = pointer to the object field image in memory
- `vmt` = pointer to the actual vmt table (used to get the size of the object)
- `vmt_pos` = offset in the object field image to the vmt pointer field

### Notes

This routine should / save restore all used registers.

### Algorithm

```pascal
if self = nil then
  exit

set VMT field in object field image, if present, to nil
Free the allocated heap memory for the field objects
set Self = nil
```

---

**FPC_HELP_CONSTRUCTOR**

Could be implemented in ASM directly with register parameter passing.

### Int_Help_Constructor

**Declaration:**

```pascal
function Int_Help_Constructor(Var _self : Pointer; Var VMT : Pointer;
  Vmt_Pos : Cardinal):Pointer;
```

**Description:**

Allocates the memory for an object’s field, and fills the object fields with zeros. Returns the newly allocated `self_pointer`

**Parameters**

- `self` = pointer to the object field image in memory
- `vmt` = pointer to the actual vmt table (used to get the size of the object)
- `vmt_pos` = offset in the object field image to the vmt pointer field

**Notes**

The `self_pointer` register should be set appropriately by the code generator to the allocated memory (self parameter)

**Algorithm**

- Self = Allocate Memory block for object fields
- Fill the object field image with zeros
- Set the VMT field in allocated object to VMT pointer

---

**FPC_HELP_FAIL_CLASS**

**Help_Fail_Class**

**Description:**

Inserted by code generator after constructor call. If the constructor failed to allocate the memory for its fields, this routine will be called.
11.8 Set handling

FPC_HELP_FAIL

Help_Fail

Description: Inserted by code generator after constructor call. If the constructor failed to allocate the memory for its fields, this routine will be called.

FPC_SET_COMP_SETS

Do_Comp_Sets

Declaration: function Do_Comp_Sets(Set1, Set2 : Pointer): Boolean;

Description: This routine compares if set1 and set2 are exactly equal and returns 1 if so, otherwise it returns false.

Parameters: set1 = Pointer to 32 byte set to compare
set2 = Pointer to 32 byte set to compare

Notes: Both pointers must point to normal sets.

FPC_SET_CONTAINS_SET

Do_Contains_Sets

Declaration: Procedure Do_Contains_Sets(Set1, Set2 : Pointer): Boolean;

Description: Returns 1 if set2 contains set1 (That is all elements of set2 are in set1).

Parameters: set1 = Pointer to 32 byte set to verify
set2 = Pointer to 32 byte set to verify

Notes: Both pointers must point to normal sets.

FPC_SET_CREATE_ELEMENT

Do_Create_Element

Declaration: procedure Do_Create_Element(p : Pointer; b : Byte);

Description: Create a new normal set in the area pointed to by p and add the element value b in that set.
Parameters: \( p = \) pointer to area where the 32 byte set will be created
\( b = \) bit value within that set which must be set

Notes: This works on normal sets only.

Algorithm: Zero the area pointed to by \( p \)
Set the bit number \( b \) to 1

**FPC_SET_SET_RANGE**

**Do_Set_Range**

Declaration: Procedure Do_Set_Range(\( P : \) Pointer; \( l, h : \) Byte);

Description: Sets the bit values within the \( l \) and \( h \) bit ranges in the normal set pointed to by \( p \)

Parameters: \( p = \) pointer to area where the 32 bytes of the set will be updated
\( l = \) low bit number value to set
\( h = \) high bit number value to set

Notes: This works on normal sets only.

Algorithm: Set all bit numbers from \( l \) to \( h \) in set \( p \)

**FPC_SET_SET_BYTE**

**Do_Set_Byte**

Declaration: procedure Do_Set_Byte(\( P : \) Pointer; \( B : \) byte);

Description: Add the element \( b \) in the normal set pointed to by \( p \)

Parameters: \( p = \) pointer to 32 byte set
\( b = \) bit number to set

Notes: This works on normal sets only. The intel 80386 version of the compiler does not save the used registers, therefore, in that case, it must be done in the routine itself.

Algorithm: Set bit number \( b \) in \( p \)
FPC_SET_SUB_SETS

Do_Sub_Sets

Declaration: Procedure Do_Sub_Sets(Set1, Set2, Dest: Pointer);

Description: Calculate the difference between set1 and set2, setting the result in dest.

Parameters: set1 = pointer to 32 byte set
             set2 = pointer to 32 byte set
             dest = pointer to 32 byte set which will receive the result

Notes: This works on normal sets only.

Algorithm:
For each bit in the set do
  dest bit = set1 bit AND NOT set2 bit

FPC_SET_MUL_SETS

Do_Mul_Sets

Declaration: procedure Do_Mul_Sets(Set1, Set2, Dest: Pointer);

Description: Calculate the multiplication between set1 and set2, setting the result in dest.

Parameters: set1 = pointer to 32 byte set
             set2 = pointer to 32 byte set
             dest = pointer to 32 byte set which will receive the result

Notes: This works on normal sets only.

Algorithm:
For each bit in the set do
  dest bit = set1 bit AND set2 bit

FPC_SET_SYMDIF_SETS

Do_Symdif_Sets

Declaration: Procedure Do_Symdif_Sets(Set1, Set2, Dest: Pointer);

Description: Calculate the symmetric between set1 and set2, setting the result in dest.
11.8  Set handling

Parameters: set1 = pointer to 32 byte set
set2 = pointer to 32 byte set
dest = pointer to 32 byte set which will receive the result

Notes: This works on normal sets only.
Algorithm:

For each bit in the set do
dest bit = set1 bit XOR set2 bit

FPC_SET_ADD_SETS

Do_Add_Sets

Declaration: procedure Do_Add_Sets(Set1, Set2, Dest: Pointer);

Description: Calculate the addition between set1 and set2, setting the result in dest.

Parameters: set1 = pointer to 32 byte set
set2 = pointer to 32 byte set
dest = pointer to 32 byte set which will receive the result

Notes: This works on normal sets only.
Algorithm:

For each bit in the set do
dest bit = set1 bit OR set2 bit

FPC_SET_LOAD_SMALL

Do_Load_Small

Declaration: Procedure Do_Load_Small(P: Pointer; L: Longint);

Description: Load a small set into a 32-byte normal set.

Parameters: p = pointer to 32 byte set
l = value of the small set

Notes: Called by code generator (type conversion) from small set to large set. Apart from the first 32 bits of the 32 byte set, other bits are not modified.
Algorithm:

For n = bit 0 to bit 31 of l do
p bit n = l bit n
11.9 Optional internal routines

**FPC_SET_UNSET_BYTE**

**Do_Unset_Byte**

*Declaration:* Procedure Do_Unset_Byte(P : Pointer; B : Byte);

*Description:* Called by code generator to exclude element b from a big 32-byte set pointed to by p.

*Parameters:* p = pointer to 32 byte set  
  b = element number to exclude

*Notes:* The intel 80386 version of the compiler does not save the used registers, therefore, in that case, it must be done in the routine itself.

*Algorithm:* Clear bit number b in p

**FPC_SET_IN_BYTE**

**Do_In_Byte**

*Declaration:* Function Do_In_Byte(P : Pointer; B : Byte):boolean;

*Description:* Called by code generator to verify the existence of an element in a set. Returns TRUE if b is in the set pointed to by p, otherwise returns FALSE.

*Parameters:* p = pointer to 32 byte set  
  b = element number to verify

*Notes:* This routine should save / restore all used registers.

*Algorithm:* Clear bit number b in p

11.9 Optional internal routines

These routines are dependant on the target architecture. They are present in software if the hardware does not support these features.

They could be implemented in assembler directly with register parameter passing.

**FPC_MUL_INT64**

**MulInt64**

*Declaration:* function MulInt64(f1,f2 : Int64;CheckOverflow : LongBool) : Int64;
Description: Called by the code generator to multiply two int64 values, when the hardware does not support this type of operation. The value returned is the result of the multiplication.

Parameters:
- f1 = first operand
- f2 = second operand
- checkoverflow = TRUE if overflow checking should be done

**FPC_DIV_INT64**

**DivInt64**

**Declaration:**
```
function DivInt64(n,z : Int64) : Int64;
```

Description: Called by the code generator to get the division two int64 values, when the hardware does not support this type of operation. The value returned is the result of the division.

Parameters:
- n = numerator
- z = denominator

**FPC_MOD_INT64**

**ModInt64**

**Declaration:**
```
function ModInt64(n,z : Int64) : Int64;
```

Description: Called by the code generator to get the modulo two int64 values, when the architecture does not support this type of operation. The value returned is the result of the modulo.

Parameters:
- n = numerator
- z = denominator

**FPC_SHL_INT64**

**ShlInt64**

**Declaration:**
```
function ShlInt64(Cnt : Longint; Low, High: Longint): Int64;
```

Description: Called by the code generator to shift left a 64-bit integer by the specified amount cnt, when this is not directly supported by the hardware. Returns the shifted value.

Parameters:
- low,high = value to shift (low / high 32-bit value)
- cnt = shift count
11.9  Optional internal routines

FPC_SHR_INT64

ShrInt64

Declaration: function ShrInt64(Cnt : Longint; Low, High: Longint): Int64;

Description: Called by the code generator to shift left a 64-bit integer by the specified amount cnt, when this is not directly supported by the hardware. Returns the shifted value.

Parameters: low,high = value to shift (low/high 32-bit values)
cnt = shift count

FPC_MUL_LONGINT

MulLong

Declaration: Function MulLong: Longint;

Description: Called by the code generator to multiply two longint values, when the hardware does not support this type of operation. The value returned is the result of the multiplication.

Parameters: Parameters are passed in registers.

Notes: This routine should save / restore all used registers.

FPC_REM_LONGINT

RemLong

Declaration: Function RemLong: Longint;

Description: Called by the code generator to get the modulo two longint values, when the hardware does not support this type of operation. The value returned is the result of the modulo.

Parameters: Parameters are passed in registers.

Notes: This routine should save / restore all used registers.

FPC_DIV_LONGINT

DivLong

Declaration: Function DivLong: Longint;
Description: Called by the code generator to get the division two longint values, when the hardware does not support this type of operation. The value returned is the result of the division.

Parameters: Parameters are passed in registers.

Notes: This routine should save / restore all used registers.

FPC_MUL_LONGINT

MulCardinal

Declaration: Function MulCardinal: Cardinal;

Description: Called by the code generator to multiply two cardinal values, when the hardware does not support this type of operation. The value returned is the result of the multiplication.

Parameters: Parameters are passed in registers.

Notes: This routine should save / restore all used registers.

FPC_REM_CARDINAL

RemCardinal

Declaration: Function RemCardinal: Cardinal;

Description: Called by the code generator to get the modulo two cardinal values, when the hardware does not support this type of operation. The value returned is the result of the modulo.

Parameters: Parameters are passed in registers.

Notes: This routine should save / restore all used registers.

FPC_DIV_CARDINAL

DivCardinal

Declaration: Function DivCardinal: Cardinal;

Description: Called by the code generator to get the division two cardinal values, when the hardware does not support this type of operation. The value returned is the result of the division.

Parameters: Parameters are passed in registers.

Notes: This routine should save / restore all used registers.
**FPC_LONG_TO_SINGLE**

**LongSingle**

**Declaration:** Function LongSingle: Single;

**Description:** Called by the code generator to convert a longint to a single IEEE floating point value.

**Parameters:** Parameters are passed in registers

**Notes:** This routine should save / restore all used registers.

FPC_ADD_SINGLE
FPC_SUB_SINGLE
FPC_MUL_SINGLE
FPC_REM_SINGLE
FPC_DIV_SINGLE
FPC_CMP_SINGLE
FPC_SINGLE_TO_LONGINT

**12 Optimizing your code**

**12.1 Simple types**

Use the most simple types, when defining and declaring variables, they require less overhead. Classes, and complex string types (ansi strings and wide strings) possess runtime type information, as well as more overhead for operating on them then simple types such as shortstring and simple ordinal types.

**12.2 constant duplicate merging**

When duplicates of constant strings, sets or floating point values are found in the code, they are replaced by only once instance of the same string, set or floating point constant which reduces the size of the final executable.

**12.3 inline routines**

The following routines of the system unit are directly inlined by the compiler, and generate more efficient code:
### Prototype

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition and notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>function pi : extended;</td>
<td></td>
</tr>
<tr>
<td>function abs(d : extended) : extended;</td>
<td></td>
</tr>
<tr>
<td>function sqr(d : extended) : extended;</td>
<td></td>
</tr>
<tr>
<td>function sqrt(d : extended) : extended;</td>
<td></td>
</tr>
<tr>
<td>function arctan(d : extended) : extended;</td>
<td></td>
</tr>
<tr>
<td>function ln(d : extended) : extended;</td>
<td></td>
</tr>
<tr>
<td>function sin(d : extended) : extended;</td>
<td></td>
</tr>
<tr>
<td>function cos(d : extended) : extended;</td>
<td></td>
</tr>
<tr>
<td>function ord(X): longint;</td>
<td>Changes node type to be type compatible</td>
</tr>
<tr>
<td>function lo(X) : byte or word;</td>
<td>Generates 2-3 instruction sequence inline</td>
</tr>
<tr>
<td>function hi(X) : byte or word;</td>
<td>Generates 2-3 instruction sequence inline</td>
</tr>
<tr>
<td>function chr(b : byte) : Char;</td>
<td>Changes node type to be type compatible</td>
</tr>
</tbody>
</table>
| function Length(s : string) : byte; | Generate 2-3 instruction sequence (
| function Length(c : char) : byte; | appx.) |
| procedure Reset(var f : TypedFile); | Calls FPC_RESET_TYPED |
| procedure rewrite(var f : TypedFile); | Calls FPC_REWRITE_TYPED |
| procedure settextbuf(var F : Text; var Buf); | Calls SetTextBuf of runtime library |
| procedure writeln; | Calls FPC_WRITE_XXXX routines |
| procedure writeln; | Calls FPC_WRITE_XXXX routines |
| procedure read; | Calls FPC_READ_XXXX routines |
| procedure readln; | Calls FPC_READ_XXXX routines |
| procedure concat; | Generates a TREE NODES of type addn |
| function assigned(var p): boolean; | Generates 1-2 instruction sequence inline |
| procedure str(X :[Width [:Decimals]]; var S); | |
| function sizeof(X): longint; | Generates 2-3 instruction sequence inline |
| function typeof(X): pointer; | Generates 2-3 instruction sequence inline |
| procedure val(S;var V; var Code: integer); | Generates a TREE NODE of type ordconstn |
| function seg(X): longint; | Generates 2-3 instruction sequence inline |
| function High(X) | Generates 2-3 instruction sequence inline |
| function Low(X) | Generates 2-3 instruction sequence inline |
| function pred(x) | Generates 2-3 instruction sequence inline |
| function succ(X) | Generates 2-3 instruction sequence inline |
| procedure inc(var X [ ; N: longint]); | Calls routine FPC_ASSERT if the assert fails. |
| procedure dec(var X [: ; N:longint]); | Generates a TREE NODE of type addm |
| procedure include(var s: set of T; I: T); | Generates 1 instruction sequence inline |
| procedure exclude(var s : set of T; I: T); | |
| procedure assert(expr : Boolean); | |
| function addr(X): pointer; | |
| function typelInfo(typeldent): pointer; | |
12.4 temporary memory allocation reuse

When routines are very complex, they may require temporary allocated space on the stack to store intermediate results. The temporary memory space can be reused for several different operations if other space is required on the stack.

13 Appendix A

This appendix describes the temporary defines when compiling software under the compiler:
The following defines are defined in FreePascal for v1.0.x, but they will be removed in future versions, they are used for debugging purposes only:

- INT64
- HASRESOURCESTRINGS
- NEWVMTOFFSET
- HASINTERNMATH
- SYSTEMVARREC
- INCLUDEOK
- NEWMM
- HASWIDECHAR
- INT64FUNCRESOK
- CORRECTFLDCW
- ENHANCEDRAISE
- PACKENUMFIXED

NOTE: Currently, the only possible stack alignment are either 2 or 4 if the target operating system pushes parameters on the stack directly in assembler (because for example if pushing a long value on the stack while the required stack alignment is 8 will give out wrong access to data in the actual routine – the offset will be wrong).
Index

AbstractError, 105
AddRef, 103
AktProcSym, 56
AnsiStr_Assign, 93
AnsiStr_Compare, 93
AnsiStr_Concat, 92
AnsiStr_Decr_Ref, 93
AnsiStr_Incr_Ref, 93
AsmGetGetChar, 20

BoolDef, 59
cAnsiStringDef, 61
Catches, 101
cCharDef, 58
cFileDef, 63
cFormalDef, 63
CharPointerDef, 62
ChDir, 90
CheckPointer, 104
Clear_Location, 67
cLongStringDef, 61
ConcatCopy, 81
cs64BitDef, 60
cShortStringDef, 61
cu64BitDef, 60
Current_Module, 58
cWideCharDef, 59
cWideStringDef, 62

DecRef, 103
Del_Location, 67
Del_Reference, 71
DestroyException, 101
DisposeReference, 70
DivCardinal, 114
DivInt64, 112
DivLong, 113
Do_Add_Sets, 110
Do_Close, 87
Do_Comp_Sets, 107
Do_Contains_Sets, 107
Do_Create_Element, 107
Do_Erase, 87
Do_Exit, 104
Do_FilesPos, 88
Do_FilesSize, 89
Do_In_Byte, 111
Do_IsDevice, 89
Do_Load_Small, 110
Do_Mul_Sets, 109
Do_Open, 90
Do_Read, 88
Do_Rename, 88
Do_Seek, 89
Do_SeeKend, 89
Do_Set_Byte, 108
Do_Set_Range, 108
Do_Sub_Sets, 109
Do_Symdif_Sets, 109
Do_Truncate, 87
Do_Unset_Byte, 111
Do_Write, 88

Emit_Load_Loc_Reg, 79
EmitCall, 81
EmitLab, 80
EmitLabeled, 80
Emit_Flag2Reg, 81
Emit_Lea_Loc_Ref, 80
Emit_Lea_Loc_Reg, 80
emit_mov_ref_reg64, 79

Finalize, 103
FloatLoad, 79
FloatStore, 79
Fpc_Chararray_To_AnsiStr, 95
Fpc_Char_To_AnsiStr, 95
Fpc_pChar_To_AnsiStr, 96
FPC_ShortStr_To_AnsiStr, 94
FreeMem, 104
GetAdressReg, 73
GetExplicitRegister32, 74
GetFirstTempSize, 76
GetFloatRegister, 72
GetHeapSize, 86
GetHeapStart, 86
GetLabel, 80
GetMem, 104
GetRegister32, 72
GetRegisterPair, 72
GetResourceString, 101
GetSym, 32
GetTempAnsiStringReference, 75
GetTempOfSize, 75
GetTempOfSizePersistant, 76
GetTempOfSizeReference, 75
GlobalDef, 33
HandleError, 99
Help_Fail, 107
Help_Fail_Class, 106

Initialize, 102
InitializeUnits, 105
Int_Assert, 99
Int_BoundCheck, 96
Int_Check_Object, 97
Int_Check_Object_Ext, 98
Int_Do_As, 102
Int_Do_Is, 102
Int_Help_Constructor, 106
Int_HelpDestructor, 105
Int_IOCheck, 98
int_new_class, 105
Int_OverFlow, 97
Int_RangeError, 96
Int_ShrackCheck, 96
Int_StrCmp, 92
Int_StrConcat, 92
Int_StrCopy, 91
IsAddressRegister, 73
IsFloatsRegister, 73
IsTemp, 77

LexLevel, 58
LongSingle, 115

MkDir, 90
ModInt64, 112
MulCardinal, 114
MulInt64, 111
MulLong, 113

NewReference, 71
New_Reference, 71
NormalTempToPersistant, 77

OpenCharArrayDef, 62
OpenShortStringDef, 62

ParamCount, 86
Pattern, 20
PersistantTempToNormal, 77
PopAddrStack, 101
PopObjectStack, 100
PopSecondObjectStack, 100
PushExceptAddr, 100

RaiseExcept, 99
Randomize, 86
ReadToken, 19
RemCardinal, 114
RemLong, 113
ReRaise, 100
ResetTempGen, 76
Reset_Reference, 71
RestoreUsedRegisters, 74
RmDir, 90

s32BitDef, 60
s32FixedDef, 61
s32FloatDef, 60
s64FloatDef, 60
s80FloatDef, 61
SaveUsedRegisters, 73
sbrk, 87
Search_a_Symtable, 32
SetFirstTemp, 76
SetJmp, 91
Set_Location, 68
ShlInt64, 112
ShrInt64, 113
SrSym, 33
SrSymTable, 34
StrCharArray, 95
StrPas, 94
Str_To_CharArray, 94
Swap_Location, 68
System_Exit, 86
TDef.Alignment, 53
TDef.Size, 53
Token, 20
u16BitDef, 59
u32BitDef, 59
u8BitDef, 59
UnGetIfTemp, 75
UngetPersistantTemp, 76
UngetRegister, 73
UngetRegister32, 72
VoidDef, 58
VoidFarPointerDef, 63
VoidPointerDef, 62