Free Pascal 1.0.x Internal documentation
version 1.0

January 3, 2003

Carl Eric Codère

Contents

List of Figures

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 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.

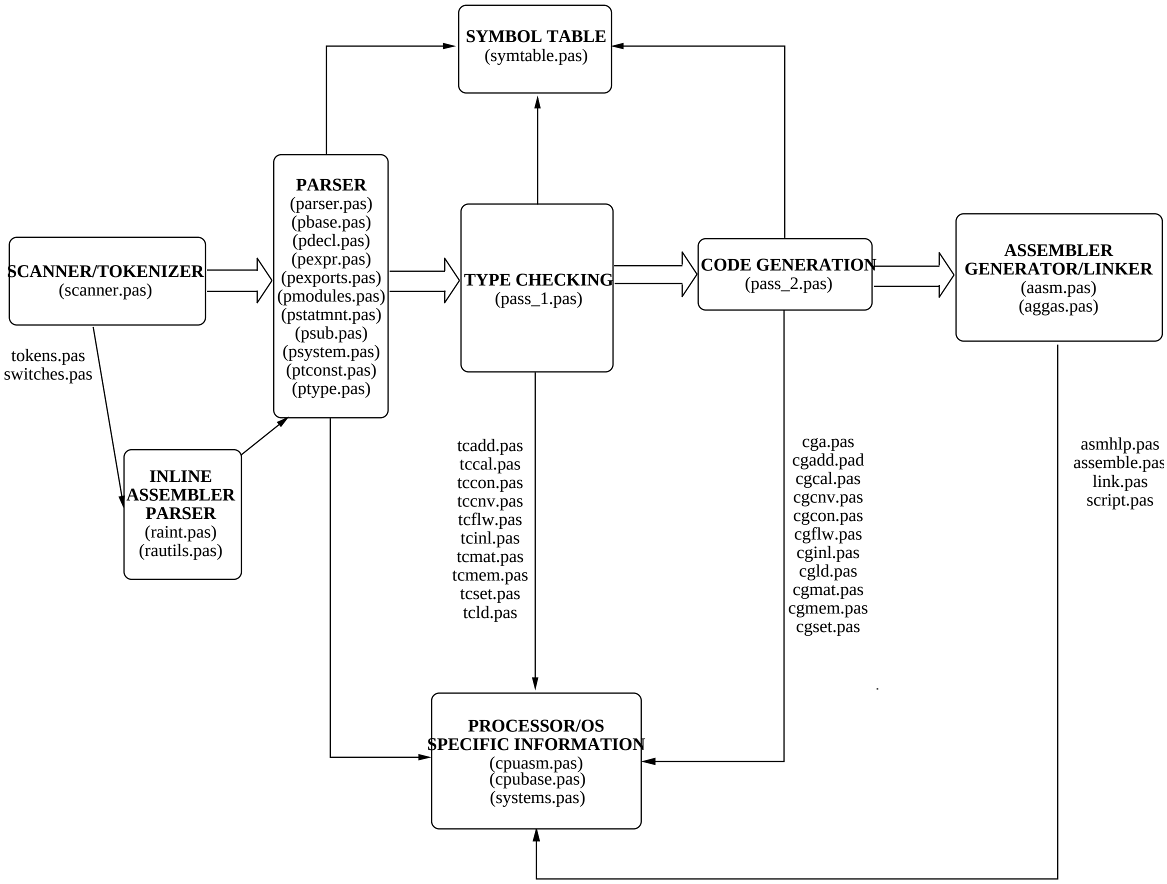

Figure 1: compiler overview

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 overriden 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**

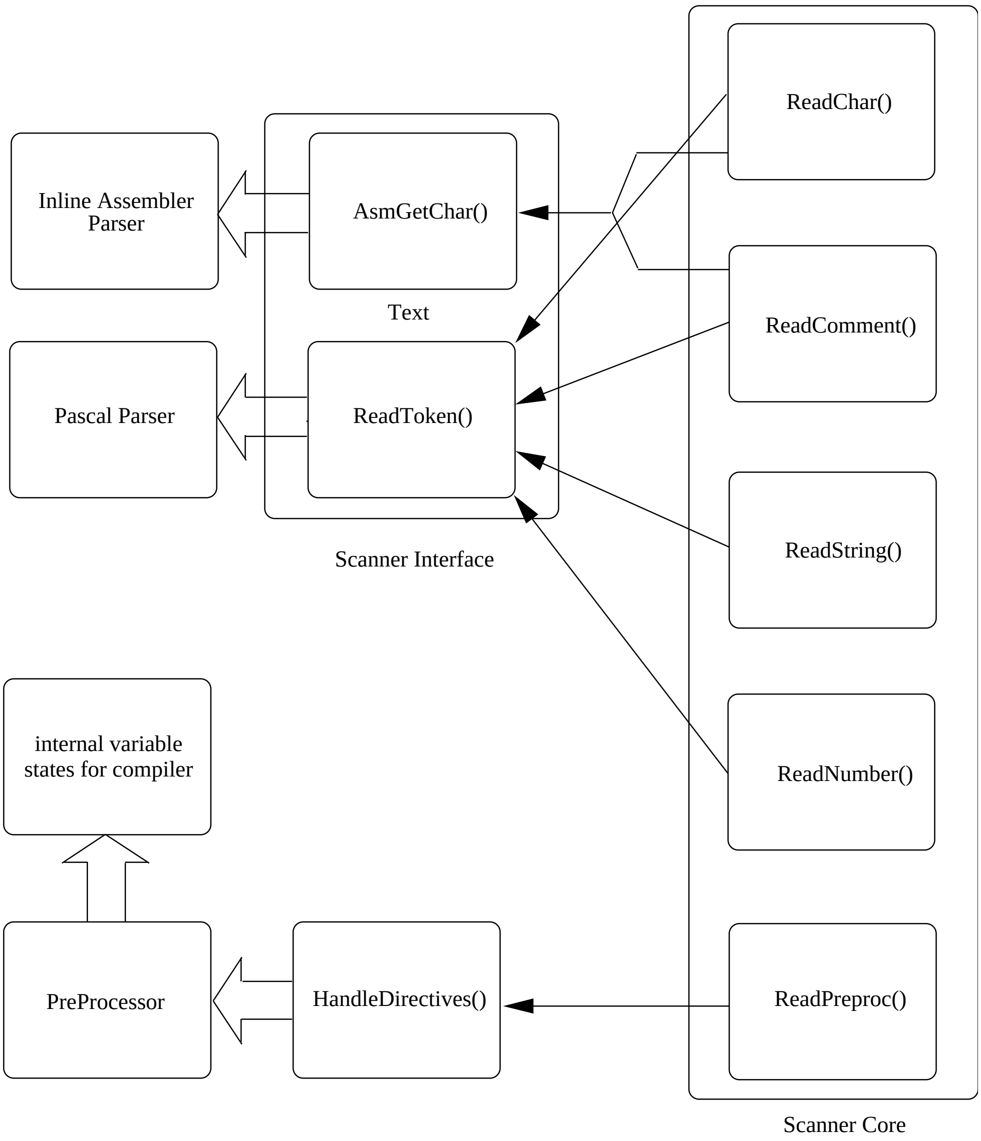

Figure 2: scanner interface overview

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 assmued 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).

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:

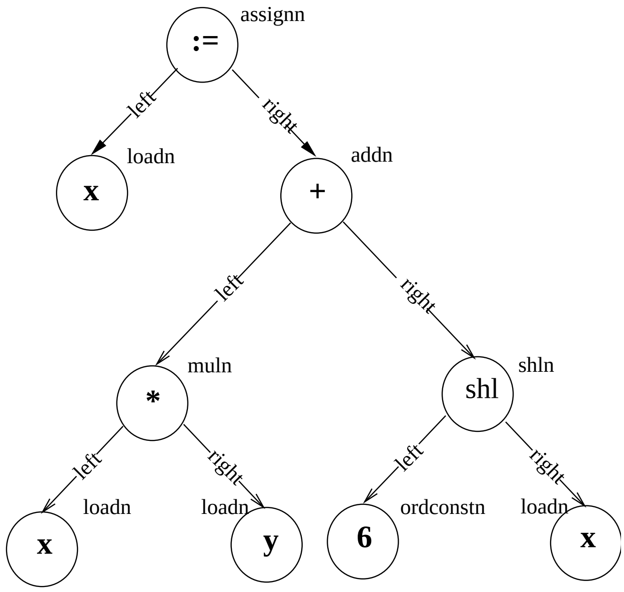

Figure 3: Example tree structure

$$x := x * y + (6\ shl\ x);$$

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

3.2 Tree types

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

Table 1: Possible node types (ttreetyp)

Tree type definition	Description
addn	Represents the + operator
muln	Represents the * operator
subn	Represents the **-** operator
divn	Represents the **div** operator
symdifn	Represents the **><** operator
modn	Represents the **mod** operator
assignn	Represents the **:=** operator (assignment)
loadn	Represents the use of a variable
rangen	Represents a numeric range (i.e 0..9)
ltn	Represents the **<** operator
lten	Represents the **<=** operator

Table 1: Possible node types (ttreetyp) - continued

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

Table 1: Possible node types (ttreetyp) - continued

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

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:

TYPE	
pTree = ˆ TTree;	
TTree = **RECORD**	
Error : boolean;	Set to TRUE if there was an error parsing this node
DisposeTyp : tdisposetyp;	
Swaped : boolean;	Set to TRUE if the left and right nodes (fields) of this node have been swaped.
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)

tlocalswitches	Switch	Description
cs_check_overflow	{$Q+}	Code generator should emit overflow checking code
cs_check_range	{$R+}	Code generator should emit range checking code
cs_check_IO	{$I+}	Code generator should emit I/O checking code
cs_check_object_ext	N/A	Code generator should emit extended object access checks
cs_omitstackframe	*N/A*	*Code generator should not emit frame_pointer setup code in entry code*
cs_do_assertion	{$C+}	Code generator supports using the assert inline routine
cs_generate_rtti	{$M+}	Code generator should emit runtime type information
cs_typed_addresses	{$T+}	Parser emits typed pointer using the @ operator
cs_ansistrings	{$H+}	Parser creates an ansistring when an unspecified String type is declared instead of the default ShortString

tlocalswitches	Switch	Description
cs_strict_var_strings	{$V+}	String types must be identical (same length) to be compatible

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

Field	Description
Use_StrConcat : Boolean;	*Currently unused (use for optimizations in future versions)*
String_Typ: TStringType;	In the case where the + operator is applied on a string, this field indicates the string type.

CallParaN

Field	Description
Is_Colon_Para : Boolean;	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)
Exact_Match_Found : Boolean;	Set to TRUE if the parameter type is exactly the same as the one expected by the routine.
ConvLevel1Found : Boolean;	Set to TRUE if the parameter type requires a level 1 type conversion to conform to the parameter expected by the routine.
ConvLevel2Found : Boolean;	Set to TRUE if the parameter type requires a level 2 type conversion to conform to the parameter expected by the routine.
HighTree : pTree;	

AssignN

Field	Description
AssignTyp : TAssignTyp;	*Currently unused (Used to be used for C-like assigns)*
Concat_String : Boolean;	*Currently unused (use for optimizations in future versions)*

LoadN

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

CallN

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

addrn

Field	Description
ProcVarLoad : Boolean;	Set to TRUE if this is a procedural variable call

OrdConstN

Field	Description
Value : Longint;	The numeric value of this constant node

RealConstN

Field	Description
Value_Real : Best_Real;	The numeric value of this constant node
Lab_Real : pAsmLabel;	The assembler label reference to this constant

FixConstN

Field	Description
Value_Fix : Longint;	The numeric value of this constant node

FuncRetN

Field	Description
FuncRetProcInfo : Pointer; (pProcInfo)	Pointer to procedure information
RetType : TType;	Indicates the return type of the function
Is_First_FuncRet : Boolean;	

SubscriptN

Field	Description
vs : pVarSym;	Symbol table entry for this variable (a field of object/class/record)

RaiseN

Field	Description
FrameTree : pTree;	Exception frame tree (code in Raise statement)

VecN

Field	Description
MemIndex : Boolean;	Set to TRUE if Mem[Seg:Ofs] directive is parsed
MemSeg : Boolean;	Set to TRUE if Mem[Seg:Ofs] directive is parsed
CallUnique: Boolean;	

StringConstN

Field	Description
Value_Str : pChar;	The constant value of the string
Length : Longint;	Length of the string in bytes (or in characters???)
Lab_Str : pAsmLabel;	The assembler label reference to this constant
StringType : TStringType;	The string type (short, long, ansi, wide)

TypeConvN

Field	Description

Field	Description
ConvType: TConvertType; Explizit : Boolean;	Indicates the conversion type to do set to TRUE if this was an explicit conversion (with explicit typecast, or calling one of the internal conversion routines)

TypeN

Field	Description
TypeNodeType : pDef; TypeNodeSym : pTypeSym;	The type definition for this node The type symbol information

InlineN

Field	Description
InlineNumber: Byte; InlineConst : Boolean;	Indicates the internal routine called (Cf. code generator) One or more of the parameters to this inline routine call contains constant values

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.

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

SetConstN

Field	Description
Value_Set : pConstSet; Lab_Set : pAsmLabel;	The numeric value of this constant node The assembler label reference to this constant

LoopN

Field	Description

AsmN

Field	Description
p_Asm : pAasmOutput; Object_Preserved : Boolean;	The instruction tree created by the assembler parser set to FALSE if the Self_Register was modified in the asm statement.

CaseN

Field	Description
Nodes : pCaserecord; ElseBlock : pTree;	Tree for each of the possible case in the case statement Else statement block tree

LabelN, GotoN

Field	Description
LabelNr : pAsmLabel; ExceptionBlock : ptree; LabSym : pLabelSym;	Assembler label associated with this statement ? Symbol table entry for this label

WithN

Field	Description
WithSymTables : pWithSymTable; TableCount : Longint; WithReference : pReference; IsLocal : Boolean;	

OnN

Field	Description
ExceptSymTable : pSymtable; ExceptType : pObjectdef;	

ArrayConstructorN

Field	Description
CArgs : Boolean; CArgSwap : Boolean; ForceVaria : Boolean; NoVariaAllowed : Boolean; ConstructorDef : pDef;	

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.

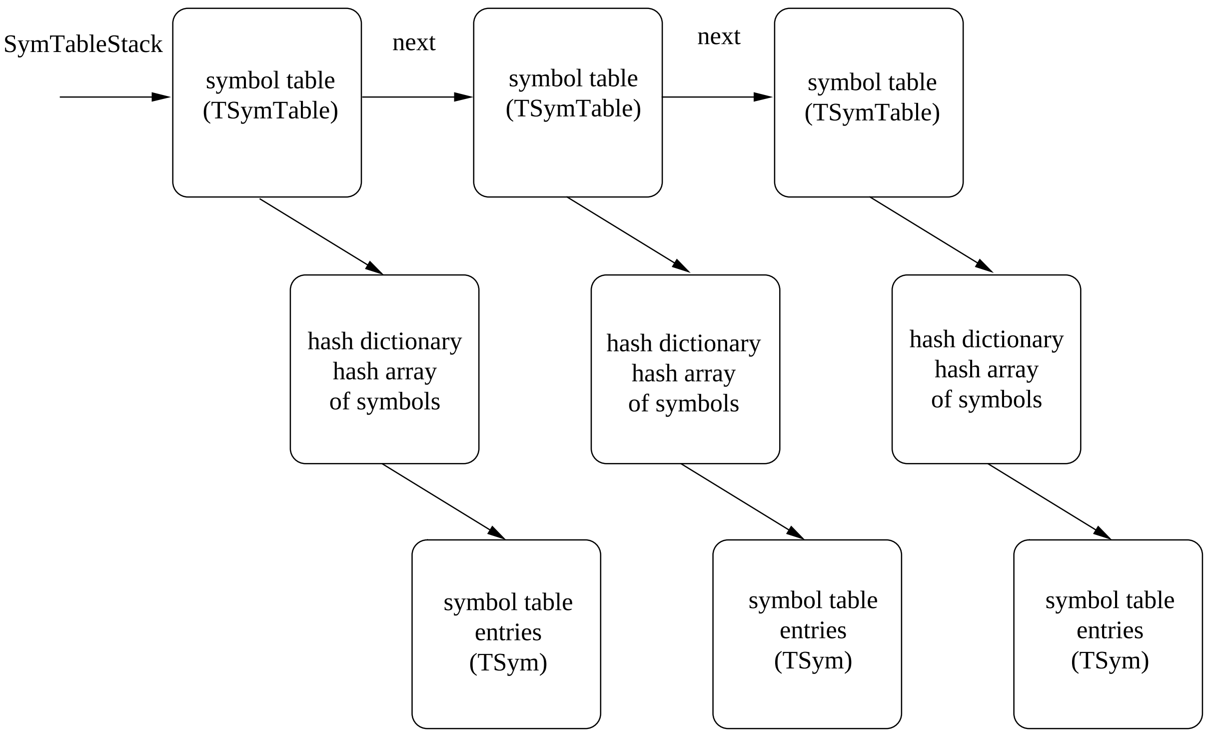

Figure 4: Interactions between symbol tables

TYPE		
pSymTable	= ˆ **TSymTable**;	
TSymTable	= **object**	
	Name : pString;	
	DataSize : Longint;	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.
	DataAlignment : Longint;	
	SymIndex : pIndexArray;	
	DefIndex : pIndexArray;	
	SymSearch : pDictionary;	
	Next : pSymtable;	Points to the next symbol table in the linked list of active symbol tables.
	DefOwner : pDef;	The owner definition (only valid in the cases of objects and records, this points to the definition of that object or record).
	Address_Fixup : Longint	
	UnitId : Word;	
	SymTableLevel : Byte;	
	SymTableType :TSymTableType;	Indicates the type of this symbol table (2).
	end;	

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

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

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.

GetSym

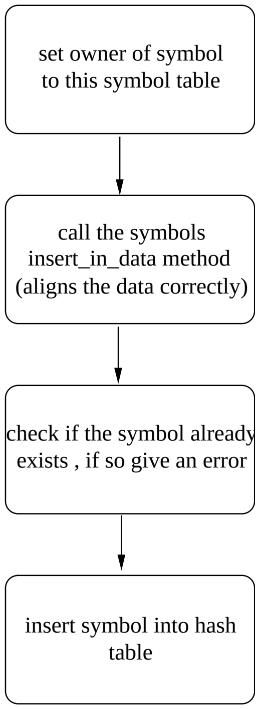

Figure 5: Inserting into the symbol table

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;`

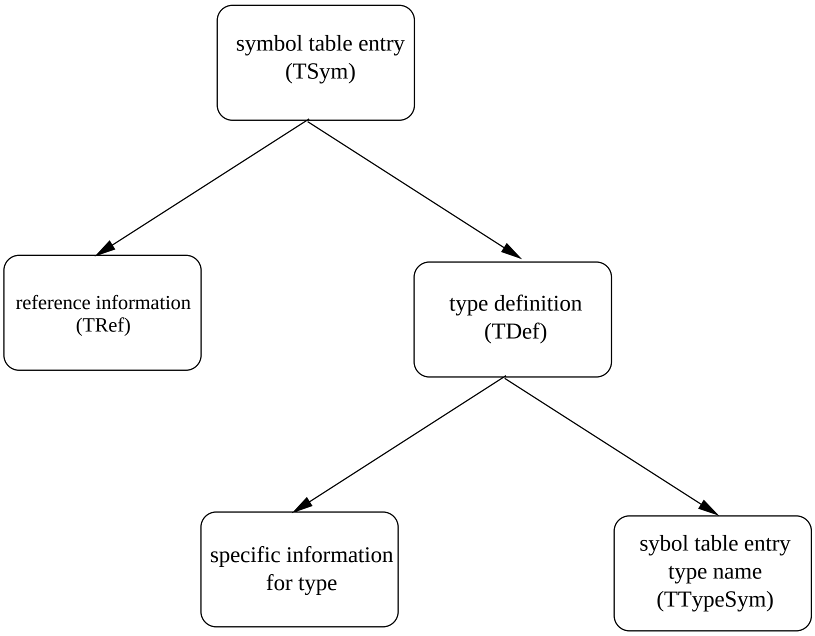

Figure 6: relation between symbol entry and type definition and name

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.

TYPE	
pSym = ^ TSym;	
TSym = **Object**(TSymTableEntry)	
SymOptions : TSymOptions;	Indicate the access scope of the symbol
FileInfo : tFilePosInfo;	
Refs : Longint;	Indicates how many times this label is refered in the parsed code (is only used with variable and assembler label symbols).
LastRef : pRef;	
DefRef : pRef;	
LastWritten : pRef;	
RefCount : Longint;	Browser information indicating the reference count
Typ : tSymTyp;	Indicates the symbol type
IsStabWritten : Boolean;	Set to TRUE if the stabs debugging information has been written for this symbol.
end;	

Table 30: tsymtyp

TSymTyp	Description
AbstractSym	This is a special abstract symbol (this should never occur)
VarSym	This symbol is a variable declaration in the var section, or a var parameter.
TypeSym	This symbol is a type name
ProcSym	This symbol is a routine or method name
UnitSym	This symbol is a unit name
ProgramSym	*This symbol is the main program name*
ConstSym	This symbol is a constant
EnumSym	This symbol is an enumeration symbol (an element in an enumeration)
TypedConstSym	This symbol is pre-initialized variable (pascal typed constant)
ErrorSym	This symbol is created for error generation
SysSym	This symbol represents an inlined system unit routine
LabelSym	This symbol represents a label in a label pascal declaration
AbsoluteSym	This symbol represents an the symbol following an absolute variable declaration

TSymTyp	Description
PropertySym	This symbol is a property name
FuncRetSym	This symbol is the name of the return value for functions
MacroSym	This symbol is a macro symbol (just like #define in C)

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:

TYPE		
pLabelSym =	ˆ 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.

TYPE		
pUnitSym =	ˆ 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 synbols 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).

TYPE		
pMacroSym =	ˆ TMacroSym;	
TMacroSym =	**Object**(TSym)	
	Defined : Boolean;	TRUE if the symbol has been defined with a $define directive, or false if it has been undefined with a $undef directive
	Defined_At_Startup : Boolean;	TRUE if the symbol is a system wide define
	Is_Used: Boolean;	TRUE if the define has been used such as in a $ifdef directive.
	BufText : pChar;	The actual string text of the define
	BufLength : Longint;	The actual string length of the define
	end;	

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		
pProcSym =	ˆ TProcSym;	
TProcSym =	**Object**(TSym)	
	Is_Global : Boolean	Set if the routine is exported by the unit
	Definition : pProcDef	Procedure definition, including parameter information and return values
	end;	

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.

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

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.

TYPE		
pVarSym = ˆ TVarSym;		
TVarSym = **Object**(TSym)		
	Reg: TRegister;	If the value is a register variable, the reg field will be different then R_NO
	VarSpez : TVarSpez;	Indicates the variable type (parameters only) (Cf. 32).
	Address : Longint;	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;	Contains base type information as well as the type definition
	VarOptions : TVarOptions;	Flags for this variable (Cf. 31)
	VarState : TVarState	Indicates the state of the variable, if it's used or declared
	end;	

Table 31: tvaroptions

TVarOptions	Description
vo_regable	The variable can be put into a hardware general purpose register
vo_is_c_var	The variable is imported from a C module
vo_is_external	The variable is declared external
vo_is_Dll_var	The variable is a shared library variable
vo_is_thread_var	The variable is declared as being thread safe

Table 31: tvaroptions (continued)

TVarOptions	Description
vo_fpuregable	The variable can be put into a hardware floating point register
vo_is_local_copy	
vo_is_const	*unused and useless*
vo_is_exported	The variable is declared as exported in a dynamic link library

Table 32: parameter type

TVarSpez	Description
vs_value	This is a value parameter
vs_const	This is a constant parameter, property or array
vs_var	This is a variable parameter

property symbol (TPropertySym)

TYPE		
pPropertySym =	ˆ TPropertySym;	
TPropertySym =	**Object**(TSym)	
	propoptions: tpropertyoptions;	???
	proptype : ttype;	Indicates the type of the property
	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.

TYPE		
pAbsoluteSym =	ˆTAbsoluteSym;	
TAbsoluteSym =	**Object**(TVarSym)	
	abstyp : absolutetyp;	Indicates the type of absolute symbol it is (Cf. 33)
	absseg : boolean;	???
	ref : psym;	In case `abstyp` is `tovar`, this field indicates the symbol which is overlaid with this symbol. Otherwise this field is unused.
	asmname : pstring;	In case `abstyp` is `toasm`, this field indicates label name for the variable.

Table 33: possible absolute variable types

tabsolutetyp	Description
tovar	The symbol will be declared on top of another symbol (variable or typed constant)
toasm	The variable is imported from an external module
toaddr	The variable is declared as being at an absolute address

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.

TYPE		
pConstSym =	ˆTConstSym;	
TConstSym =	**Object**(TSym)	
	consttype : ttype;	Type information for this constant (?).
	consttyp : tconsttyp	Indicates the type of the constant
	resstrindex : longint	If this is a resource string constant, it indicates the index in the resource table
	value : longint	In certain cases, contains the value of the constant
	len : longint	

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 `tsyssym` 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.

5.3 Symbol interface

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.

TYPE		
TType =	**Object**	
	Sym : pSym;	Points to the symbol table of this type
	Def : pDef;	Points to the actual definition of this type
	end;	

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:

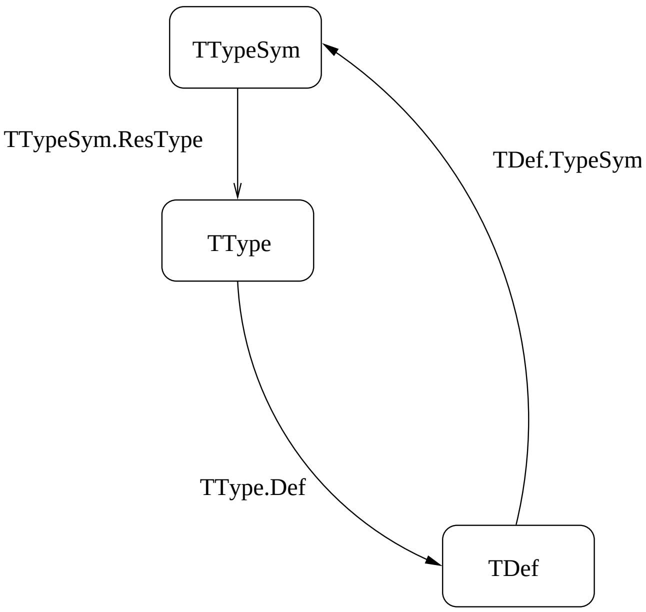

Figure 7: Type symbol and definition relations

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

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.

TYPE		
pDef = ˆ TDef;		
TDef = **Object**(TSymTableEntry)		
	TypeSym : pTypeSym;	Pointer to symbol table entry for this type definition
	InitTable_Label : pAsmLabel;	Label to initialization information (required for some complex types)
	Rtti_Label : pAsmLabel;	Label to the runtime type information.
	NextGlobal : pDef;	
	PreviousGlobal : pDef;	
	SaveSize : Longint;	Size in bytes of the data definition
	DefType : tDefType;	Indicates the definition type (see table **??**).
	Has_InitTable : Boolean;	
	Has_Rtti : Boolean;	
	Is_Def_Stab_Written : TDefStabStatus	Can be one of the following states : (Not_Written, written, Being_Written) which indicates if the debug information for this type has been defined or not.
	GlobalNb : Longint;	Internal stabs debug information type signature (each type definition has a numeric signature).
	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 =	ˆ TFileDef;	
TFileDef =	**Object**(TDef)	
	FileTyp : TFileTyp;	Indicates what type of file definition it is (text, untyped or typed).
	TypedFileType : TType;	In the case of a typed file definition, definition of the type of the file
	end;	

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 =	ˆ TForwardDef;		
TForwardDef =	**Object**(TDef)		
	toSymName : String;	The symbol name for this forward declaration (the actual real definition does not exist yet)	
	ForwardPos : TFilePosInfo;	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 =	ˆ 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.

TYPE		
pObjectDef =	ˆ 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.
	PbjectOptions : 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;	

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

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.

TYPE		
pArrayDef = ˆ TArrayDef;		
TArrayDef = **Object**(TDef)		
	IsVariant : Boolean;	
	IsConstructor : Boolean;	
	RangeNr: Longint;	Label number associated with the index values when range checking is on
	LowRange : Longint;	The lower index range of the array definition
	HighRange : Longint;	The higher index range of the array definition
	ElementType : TType;	The type information for the elements of the array
	RangeType : TType;	The type information for the index ranges of the array
	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.

TYPE		
pRecordDef = ˆ 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.

TYPE		
pOrdDef =	ˆ TOrdDef;	
TOrdDef =	**Object**(TDef)	
	Low : Longint;	The minimum value of this ordinal type
	High : Longint;	The maximum value of this ordinal type
	Typ : TBaseType;	The type of ordinal value (cf. table **??**)
	end;	

Table 36: Base types

Base ordinal type (TBaseType)	Description
uauto	user defined ordinal type definition
uvoid	Represents a void return value or node
uchar	ASCII character (1 byte)
u8bit	unsigned 8-bit value
u16bit	unsigned 16-bit value
u32bit	unsigned 32-bit value
s16bit	signed 16-bit value
s32bit	signed 32-bit value
bool8bit	boolean 8-bit value
bool16bit	boolean 16-bit value
bool32bit	boolean 32-bit value
u64bit	*unsigned 64-bit value (not fully supported/tested)*
s64bit	signed 64-bit value
uwidechar	*Currently not supported and unused*

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.

TYPE		
pFloatDef =	ˆ TFloatDef;	
TFloatDef =	**Object**(TDef)	
	Typ : TFloatType;	The type of floating point value (cf. table 37).
	end;	

Table 37: Floating point types

Base floating point type (TFloatType)	Description
s32real	IEEE Single precision floating point value
s64real	IEEE Double precision floating point value

Base floating point type (TFloatType)	Description
s80real	Extended precision floating point value (cpu-specific, usually maps to double)
s64comp	63-bit signed value, using 1 bit for sign indication
f16bit	*Unsupported*
f32bit	*Unsupported*

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.

TYPE	
pAbstractProcDef = ˆ TAbstractProcDef;	
TAbstractProcDef = **Object**(TDef)	
SymtableLevel : byte;	
Fpu_Used : Byte;	Number of floating point registers used in this routine
RetType : TType;	Type information for the return value (uvoid if it returns nothing)
ProcTypeOption : TProcTypeOption;	Indicates the type of routine it is (cf table 38).
ProcCallOptions : TProcCallOptions;	Indicates the calling convention of the routine (cf. table 39).
ProcOptions : TProcOptions;	Indicates general procedure options. (cf. table 40).
Para : pLinkedList;	This is a linked list of parameters (pparaitem list)
end;	

Table 38: Procedure type options

Procedure options (TProcTypeOption)	Description
poType_ProgInit	Routine is the program entry point (defined as 'main' in the compiler).
poType_UnitInit	Routine is the unit initialization code (defined as unitname_init in the compiler
poType_UnitFinalize	Routine is the unit exit code (defined as unitname_finalize in the compiler)
poType_Constructor	Routine is an object or class constructor

Table 38: Procedure type options (continued)

Procedure options (TProcTypeOption)	Description
poType_Destructor	Routine is an object or class destructor
poType_Operator	Procedure is an operator

Table 39: Procedure call options

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

Table 40: Procedure options

routine options (TProcOptions)	Description
po_classmethod	This is a class method
po_virtualmethod	This is a virtual method
po_abstractmethod	This is an abstract method
po_staticmethod	This is a static method
po_overridingmethod	This is an overriden method (with po_virtual flag usually)
po_methodpointer	This is a method pointer (not a normal routine pointer)
po_containsself	self is passed explicitly as a parameter to the method
po_interrupt	This routine is an interrupt handler
po_iocheck	IO checking should be done after a call to the procedure
po_assembler	The routine is in assembler
po_msgstr	method for string message handling
po_msgint	method for int message handling
po_exports	Routine has export directive
po_external	Routine is external (in other object or lib)
po_savestdregs	Routine entry should save all registers used by GCC

Table 40: Procedure options (continued)

routine options (TProcOptions)	Description
po_saveregisters	Routine entry should save all registers
po_overload	Routine is declared as being overloaded

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.

```
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.

TYPE		
pProcDef = ˆ TProcDef;		
TProcDef =**Object**(TAbstractProcDef)		
	ForwardDef : Boolean;	TRUE if this is a forward definition
	InterfaceDef: Boolean;	
	ExtNumber : Longint;	
	MessageInf : TMessageInf;	
	NextOverloaded : pProcDef;	
	FileInfo : TFilePosInfo;	Position in source code for the declaration of this routine. Used for error management.
	Localst : pSymTable;	The local variables symbol table
	Parast: pSymTable;	The parameter symbol table
	ProcSym : pProcSym;	Points to owner of this definition
	LastRef : pRef;	
	DefRef: pRef;	
	CrossRef : pRef;	
	LastWritten : pRef;	
	RefCount : Longint;	
	_Class : ProbjectDef;	
	Code : Pointer;	The actual code for the routine (only for inlined routines)
	UsedRegisters : TRegisterSet;	The set of registers used in this routine
	HasForward : Boolean;	
	Count: Boolean;	
	Is_Used : Boolean;	
	end;	

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.

TYPE		
pStringDef =	ˆ TStringDef;	
TStringDef =	**Object**(TDef)	
	String_Typ : TStringType;	Indicates the string type definition (cf. 41)
	Len : Longint;	This is the maximum length which can have the string
	end;	

Table 41: string types

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

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.

TYPE		
pEnumDef	= ˆ **TEnumDef**;	
TEnumDef	= **object**(TDef)	
	Has_Jumps : Boolean;	*Currently unused*
	MinVal : Longint;	Value of the first element in the enumeration
	MaxVal : Longint;	Value of the last element in the enumeration
	FirstEnum : pEnumSym;	Pointer to a linked list of elements in the enumeration, each with its name and value.
	BaseDef : pEnumDef;	In the case where the enumeration is a subrange of another enumeration, this gives information on the base range of the elements
	end;	

set definition (TSetDef)

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

TYPE		
pSetDef	= ^ **TSetDef**;	
TSetDef	= **object**(TDef)	
	SetType : TSetType;	Indicates the storage type of the set (Cf. table 42).
	ElementType : TType;	Points the type definition and symbol table to the elements in the set.
	end;	

Table 42: set types

set type (TSetType)	Description
NormSet	Normal set of up to 256 elements (32 byte storage space required)
SmallSet	Small set of up to 32 elements (4 byte storage space)
VarSet	*Variable number of element set (storage size is dependent on number of elements) (currently unused and unsupported)*

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.

An overview of the parsing process, as well as its relationship with the tree the type checker and the code generator is shown in the following diagram:

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.

TYPE
pModule = ˆ TModule;
TModule =**Object**(TLinkedList_Item)

PPUFile : pPPUFile;	Pointer to PPU file object (unit file)
Crc : Longint;	CRC-32 bit of the whole PPU file
Interface_CRC : Longint;	CRC-32 bit of the interface part of the PPU file
Flags: Longint;	Unit file flags
Compiled: Boolean;	TRUE if module is already compiled
Do_Reload : Boolean;	TRUE if the PPU file must be reloaded
Do_Assemble : Boolean;	Only assemble, don't recompile unit
Sources_Avail : Boolean;	TRUE if all sources of module are available
Sources_Checked : Boolean;	TRUE if the sources has already been checked
Is_Unit: Boolean;	TRUE if this is a unit (otherwise a library or a main program)
In_Compile: Boolean;	module is currently being recompiled
In_Second_Compile: Boolean;	module is being compiled for second time
In_Second_Load: Boolean;	module is being reloaded a second time
In_Implementation : Boolean;	currently compiling implementation part (units only)
In_Global : Boolean;	currently compiling implementation part (units only)
Recompile_Reason : TRecompile_Reason;	Reason why module should be recompiled

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

StaticLibFilename : pString;	Full name of the static library name (used when smart linking is used)
SharedLibFilename : pString;	Filename of the output shared library (in the case of a library)
ExeFileName : pString;	Filename of the output executable (in the case of a program)
AsmPrefix : pString;	Filename prefix of output assembler files when using smartlinking
MainSource : pString;	Name of the main source file
end;	

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

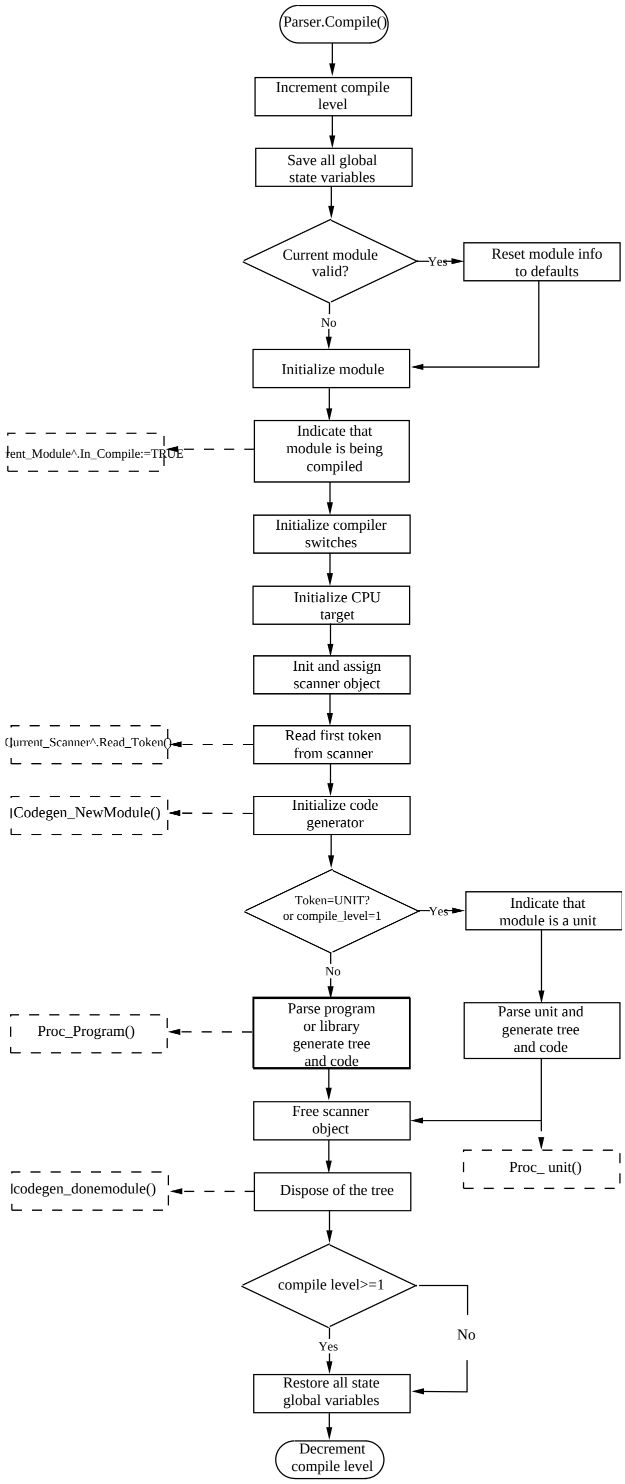

Figure 8: Parser - Scanner flow

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-nit 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.

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(opencharↄarray)

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

cfFileDef

Declaration: `Var cfFileDef : 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 1^{st} 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 1^{st} 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.

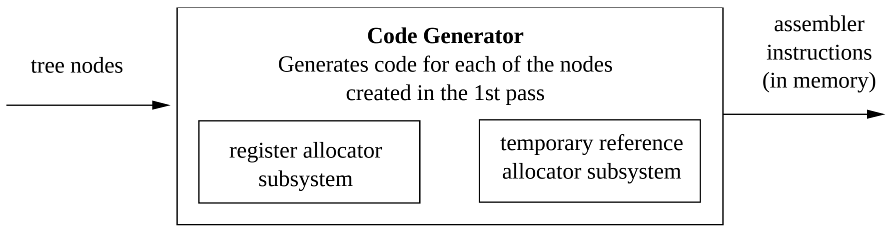

Figure 9: Code generator architecture

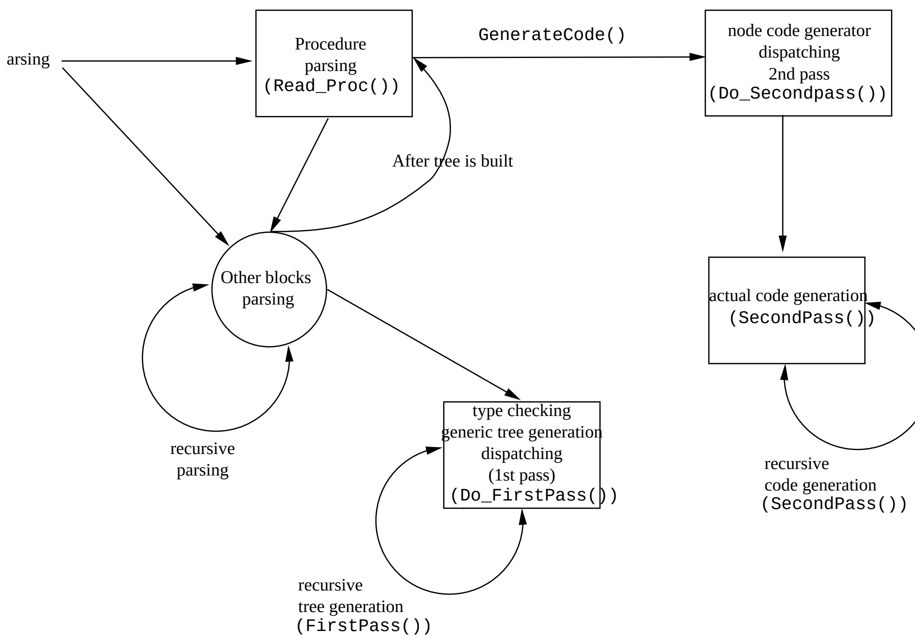

Figure 10: Interaction between codegeneration and the parsing process

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:

Location define	Description
LOC_INVALID	Invalid location (should never occur)
LOC_FPU	Floating point registers
LOC_REGISTER	Integer registers
LOC_MEM	Memory Location
LOC_REFERENCE	Constant node with constant value
LOC_JUMP	Label operand
LOC_FLAGS	Flags operand

Location define	Description
LOC_CREGISTER	Constant integer register (when operand is in this location, it should be considered as read-only)

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:

Field	Description
FpuRegister : TRegister;	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)
FpuRegisterHigh, FpuRegisterLow : TRegister;	Indicates in what registers the operand are located (for emulation support - these are general purpose registers)

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.

Field	Description
Register : TRegister	Indicates in what register the operand is located.

Field	Description
RegisterHigh : TRegister;	High 32-bit of 64-bit virtual register (on 32-bit machines)
RegisterLow : TRegister;	Low 32-bit of 64-bit virtual register (on 32-bit machines)

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:

Field	Description
Reference : TReference;	Information on the location in memory

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 treference consists of the following:

TYPE		
pReference	= ˆ **TReference**;	
TReference	= **packed Record**	
	Is_Immediate : Boolean;	Indicates that this location points to a memory location, but to a constant value (TRUE), which is located in the offset field.
	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;	Pointer to the symbol name string of the reference in case where it is a symbolic reference
	OffsetFixup : Longint;	
	Options : TRefOptions;	
	END;	

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.

Field	Description
ResFlags : TResFlags;	This indicates the flag which must be verified for the actual jump operation. tresflags is an enumeration of all possible conditional flags which can be set by the processor.

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.

Field	Description
Register : TRegister	Indicates in what register the operand is located.
RegisterHigh : TRegister;	High 32-bit of 64-bit virtual register (on 32-bit machines)
RegisterLow : TRegister;	Low 32-bit of 64-bit virtual register (on 32-bit machines)

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.

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 :

```
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.

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.

Self_Pointer

Const Self_Pointer = R_A5

This represents the self register, which represents a pointer to the current instance of a class or object.

accumulator

Const Accumulator = R_D0

The accumulator is used (except in the i386) as a scratch register, and also for return value in functions (in the case where they are 32-bit or less). In the case it is a 64-bit value (and the target processor only supports 32-bit registers) , the result of the routine is stored in the accumulator for the low 32-bit value, and in the scratch register (scratch_register) for the high 32-bit value.

scratch register

const scratch_reg = R_D1

This register is used in special circumstances by the code generator. It is simply a define to one of the registers in the scratch_regs array.

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 treference (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) : PRefere`

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.

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 alloate 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.par 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 (72)().

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.

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.

Temporary memory allocator interface (temp_gen.pas)

GetTempOfSize

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

Description: Allocates at least 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. l is the size of the persistent memory space to allocate, while Ref is a reference entry which will be set to the correct offset from the Frame_Pointer register base. The Offset and Base fields of 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 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 Ref on the volatile memory space and sets the Base to the Frame_Pointer register and Offset to the correct offset to access this allocated memory space.

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

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.

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 topsize):

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

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

Operand type (TOpType)	Description
top_None	No operand
top_Reg	Operand is a register
top_Ref	Operand is a reference (treference type)
top_Symbol	Operand is a symbol (reference or label)

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:

```
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 \ldots
```

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 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:TL`

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:TRefe`

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.

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:Bo`

Description: Loads the address of the location loc and stores the result into ther 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 (80) 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;DelSour : 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.

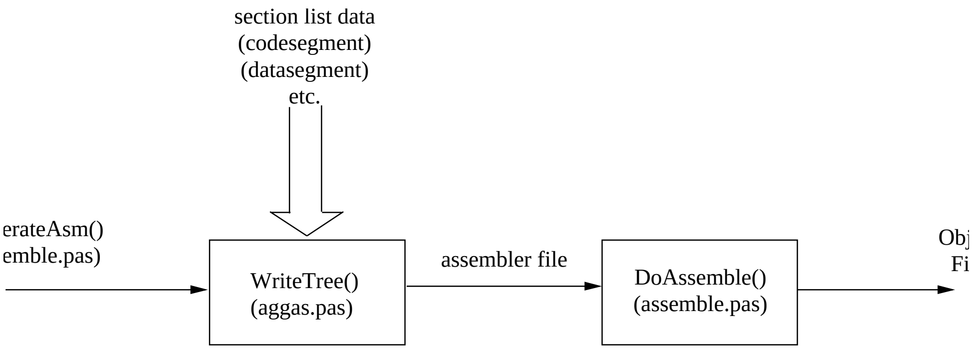

Figure 11: Assembler generation organisation

The different possible sections which are output are as follows:

Section lists for the assembler output

Internal section name	Description
ExparAsmList	temporary list
DataSegment	initialized variables
CodeSegment	instructions and general code directives
DebugList	debugging information
WithDebugList	???????????????
Consts	read only constants
ImportSection	imported symbols
ExportSection	exported symbols
ResourceSection	Resource data
RttiList	runtime type information data
ResourceStringList	resource string data

The following directives for the abstract assembler currently exist:

Abstract assembler node types:

Node entry Type	Description
Ait_None	This entry in the linked list is invalid (this should normally never occur)
Ait_Direct	Direct output to the resulting assembler file (as string)
Ait_String	Shortstring with a predefined length
Ait_Label	Numbered assembler label used for jumps
Ait_Comment	Assembler output comment
Ait_Instruction	Processor specific instruction
Ait_DataBlock	Unitialized data block (BSS)
Ait_Symbol	Entry represents a symbol (exported, imported, or other public symbol type) Possible symbol types : NONE, EXTERNAL, LOCAL and GLOBAL eg : A symbol followed by an Ait_const_32bit
Ait_Symbol_End	Symbol end (for example the end of a routine)
Ait_Const_32bit	Initialized 32-bit constant (without a symbol)
Ait_Const_16bit	Initialized 16-bit constant (without a symbol)
Ait_Const_8bit	Initialized 8-bit constant (without a symbol)
Ait_Const_symbol	????????????
Ait_Real_80bit (x86)	Initialized 80-bit floating point constant (without symbol)

Node entry Type	Description
Ait_Real_64bit	Initialized Double IEEE floating point constant (without symbol)
Ait_Real_32bit	Initialized Single IEEE floating point constant (without symbol)
Ait_Comp_64bit (x86)	Initialized 64-bit floating point integer (without symbol)
Ait_Align	Alignment directive
Ait_Section	Section directive
Ait_const_rva (Win32)	
Ait_Stabn	stabs debugging information (numerical value)
Ait_Stabs	stabs debugging information (string)
Ait_Force_Line	stabs debugging line information
Ait_Stab_Function_Name	stabs debug information routine name
Ait_Cut	Cut in the assembler files (used for smartlinking)
Ait_RegAlloc	Debugging information for the register allocator
Ait_Marker	????????????
Ait_Frame (Alpha)	
Ait_Ent (Alpha)	
Ait_Labeled_Instruction (m68k)	
Ait_Dummy	Unused - should never appear

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:

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

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

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

The following defines are used for fexpand definitions:

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

The following defines some debugging routines for the runtime library:

Define Name	Description
DEFINE NAME	Description
ANSISTRDEBUG	Add Debug routines for ansi string support
EXCDEBUG	Add Debug routines for exception debugging
LOGGING	Log the operations to a file

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.

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 specifed 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.

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.

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 gtet status on

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

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 ansistrings.

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 then `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 ansistring; 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).

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:

```
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.

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:

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

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 an 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:

```
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.

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:Long`

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

11.5 Exception handling

FPC_RAISEEXCEPTION

RaiseExcept

Declaration: `function RaiseExcept (Obj : Tobject; AnAddr,AFrame : Pointer) : Tobject;`

Description: Called by the code generator in the raise statement to raise an exception.

Parameters: Obj = Instance of class exception handler

AnAddr = Address of exception

Aframe = Exception frame address

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???????????

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

typeinfo = 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

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_Help_Destructor

Declaration:
```
Procedure Int_Help_Destructor(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 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:

```
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: `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 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.

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.

11.8 Set handling

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.

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
```

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

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) posess 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	Definition and notes
function pi : extended;	
function abs(d : extended) : extended;	
function sqr(d : extended) : extended;	
function sqrt(d : extended) : extended;	
function arctan(d : extended) : extended;	
function ln(d : extended) : extended;	
function sin(d : extended) : extended;	
function cos(d : extended) : extended;	
function ord(X): longint;	Changes node type to be type compatible
function lo(X) : byte or word;	Generates 2-3 instruction sequence inline
function hi(X) : byte or word;	Generates 2-3 instruction sequence inline
function chr(b : byte) : Char;	Changes node type to be type compatible
function Length(s : string) : byte;	Generate 2-3 instruction sequence
function Length(c : char) : byte;	Generates 1 instruction sequence (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 writen;	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);	
function seg(X): longint;	
function High(X)	Generates a TREE NODE of type ordconstn
function Low(X)	Generates a TREE NODE of type ordconstn
function pred(x)	Generates 2-3 instruction sequence inline
function succ(X)	Generates 2-3 instruction sequence inline
procedure inc(var X [; N: longint]);	Generate 2-3 instruction sequence inline
procedure dec(var X [; N:longint]);	Generate 2-3 instruction sequence inline
procedure include(var s: set of T; I: T);	
procedure exclude(var S : set of T; I: T);	
procedure assert(expr : Boolean);	Calls routine FPC_ASSERT if the assert fails.
function addr(X): pointer;	Generates a TREE NODE of type addrn
function typeInfo(typeIdent): pointer;	Generates 1 instruction sequence inline

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