Free Pascal :
Compiler documentation

Compiler documentation for Free Pascal, version 1.0.6
1.0
February 2003

Michaël Van Canneyt
Florian Klämpfl

Contents

Chapter 1

Introduction

1.1 About this document

This document tries to make the internal workings of Free Pascal more clear. It is assumed that the reader has some knowledge about compiler building.

This document describes the compiler as it is/functions at the time of writing. Since the compiler is under continuous development, some of the things described here may be outdated. In case of doubt, consult the README files distributed with the compiler. The README files are, in case of conflict with this manual, authoritative.

I hope, my poor english is quite understandable. Feel free to correct spelling mistakes.

1.2 About the compiler

1.3 Getting more information.

The ultimate source for information about compiler internals is the compiler source, though it isn't very well documented. If you need more information you should join the developers mailing list or you can contact the developers.

Chapter 2

Overview

2.1 History

2.2 The compiler passes

It isn't easy to divide the compilation process of Free Pascal into passes how it is described by many thesis about compiler building, but I would say Free Pascal does the compilation in five passes:

1. Scanning and Parsing. The compiler reads the input file, does preprocessing (i. e. reading include files, expanding macros ...) (2.2) and the parser (3.1) creates a parse tree (3.1). While this pass the compiler builds also the symbol tables (3).
2. Semantic analysis. This pass checks if semantic of the code is correct, i.e. if the types of expressions matches to the operators (3.1). This pass determines also how many registers are needed to evalute an expression, this information is used by the code generator later.
3. Code generation
4. Optimizing of the assembler
5. Assembler writing

Chapter 3

The symbol tables

The symbol table is used to store information about all symbols, declarations and definitions in a program. In an abstract view, a symbol table is a data base with a string field as index. Free Pascal implements the symbol table mainly as a binary tree, but for big symbol tables some hash technics are used. The implementation can be found in symtable.pas, object tsymtable.

The symbol table module can't be associated with a stage of the compiler, each stage accesses it. The scanner uses a symbol table to handle preprocessor symbols, the parser inserts declaration and the code generator uses the collected information about symbols and types to generate the code.

3.1 Definitions

Definitions are one of the most important data structures in Free Pascal. They are used to describe types, for example the type of a variable symbol is given by a definition and the result type of a expression is given as a definition. They have nothing to do with the definition of a procedure. Definitions are implemented as an object (in file symtable.pas, `tdef` and its descendents). There are a lot of different definitions, for example to describe ordinal types, arrays, pointers, procedures, ...

To make it more clear let's have a look at the fields of tdef:

Chapter 4

The assembler writers

Free Pascal doesn't generate machine language, it generates assembler which must be assembled and linked.

The assembler output is configurable, Free Pascal can create assembler for the GNU AS, the NASM (Netwide assembler) and the assemblers of Borland and Microsoft. The default assembler is the GNU AS, because it is fast and and available on many platforms. Why don't we use the NASM? It is 2-4 times slower than the GNU AS and it is created for hand-written assembler, while the GNU AS is designed as back end for a compiler.

Chapter 5

The register allocation

The register allocation is very hairy, so it gets an own chapter in this manual. Please be careful when changing things regarding the register allocation and test such changes intensive.

Future versions will may implement another kind of register allocation to make this part of the compiler more robust, see 5.6. But the current system is less or more working and changing it would be a lot of work, so we have to live with it.

The current register allocation mechanism was implemented 5 years ago and I didn't think that the compiler would become so popular, so not much time was spent in the design of it.

5.1 Basics

The register allocation is done in the first and the second pass of the compiler. The first pass of a node has to calculate how much registers are necessary to generate code for the node, but it also has to take care of child nodes i.e. how much registers they need.

The register allocation is done via `getregister`

Registers can be released via `ungetregister`. All registers of a reference (i.e. base and index) can be released by `del_reference`. These procedures take care of the register type, i.e. stack/base registers and registers allocated by register variables aren't added to the set of unused registers.

If there is a problem in the register allocation an `internalerror(10)` occurs.

5.2 A simple example

The first pass

This is a part of the first pass for a pointer dereferencation (`p^ `), the type determination and some other stuff are left out.

```
procedure firstderef(var p : ptree);

  begin
     // .....
     // first pass of the child node
     firstpass(p^.left);

     // .....
```

```
    // to dereference a pointer we need one one register
    // but if the child node needs more registers, we
    // have to pass this to our parent node
    p^.registers32:=max(p^.left^.registers32,1);

    // a pointer dereferencation doesn't need
    // fpu or mmx registers
    p^.registersfpu:=p^.left^.registersfpu;
    p^.registersmmx:=p^.left^.registersmmx;

    // .....
  end;
```

The second pass

The following code contains the complete second pass for a pointer dereferencing node as it is used by current compiler versions:

```
procedure secondderef(var p : ptree);

  var
     hr : tregister;

  begin
     // second pass of the child node, this generates also
     // the code of the child node
     secondpass(p^.left);
     // setup the reference (this sets all values to nil, zero or
     // R_NO)
     clear_reference(p^.location.reference);

     // now we have to distinguish the different locations where
     // the child node could be stored
     case p^.left^.location.loc of

        LOC_REGISTER:
          // LOC_REGISTER allows us to use simply the
          // result register of the left node
          p^.location.reference.base:=p^.left^.location.register;

        LOC_CREGISTER:
          begin
             // we shouldn't destroy the result register of the
             // result node, because it is a register variable
             // so we allocate a register
             hr:=getregister32;

             // generate the loading instruction
             emit_reg_reg(A_MOV,S_L,p^.left^.location.register,hr);

             // setup the result location of the current node
             p^.location.reference.base:=hr;
```

```
          end;

        LOC_MEM,LOC_REFERENCE:
          begin
             // first, we have to release the registers of
             // the reference, before we can allocate
             // register, del_reference release only the
             // registers used by the reference,
             // the contents of the registers isn't destroyed
             del_reference(p^.left^.location.reference);

             // now there should be at least one register free, so
             // we can allocate one for the base of the result
             hr:=getregister32;

             // generate dereferencing instruction
             exprasmlist^.concat(new(pai386,op_ref_reg(
               A_MOV,S_L,newreference(p^.left^.location.reference),
               hr)));

             // setup the location of the new created reference
             p^.location.reference.base:=hr;
          end;
       end;
  end;
```

5.3 Binary nodes

The whole thing becomes a little bit more hairy if you have to generate code for a binary+ node (a node with two or more childs). If a node calls second pass for a child node, it has to ensure that enough registers are free to evaluate the child node (`usableregs>=childnode^.registers32`). If this condition isn't met, the current node has to store and restore all registers which the node owns to release registers. This should be done using the procedures `maybe_push` and `restore`. If still `usableregs<childnode^.registers32`, the child nodes have to solve the problem. The point is: if `usableregs<childnode^.registers32`, the current node has to release all registers which it owns before the second pass is called. An example for generating code of a binary node is `cg386add.secondadd`.

5.4 FPU registers

The number of required FPU registers also has to be calculated, but there's one difference: you don't have to save registers. If not enough FPU registers are free, an error message is generated, as the user has to take care of this situation since this is a consequence of the stack structure of the FPU.

5.5 Testing register allocation

To test new stuff, you should compile a procedure which contains some local longint variables with -Or, to limit the number of registers:

```
procedure test;
```

```
var
   l,i,j,k : longint;

begin
   l:=i;  // this forces the compiler to assign as much as
   j:=k;  // possible variables to registers
   // here you should insert your code
end;
```

5.6 Future plans

Appendix A

Coding style guide

This chapter describes what you should consider if you modify the compiler sources.

A.1 The formatting of the sources

Rules how to format the sources.

- All compiler files should be saved in UNIX format i.e. only a line feed (#10), no carrige return (#13).
- Don’t use tabs

A.2 Some hints how to write the code

- Assigned should be used instead of checking for nil directly, as it can help solving pointer problems when in real mode.

Appendix B

Compiler Defines

The compiler can be configured using command line defines, the basic set is decribed here, switches which change rapidly or which are only used temporarly are described in the header of PP.PAS.

B.1 Target processor

The target processor must be set always and it can be:

I386 for Intel 32 bit processors of the i386 class

M68K for Motorola processors of the 68000 class

B.2 Include compiler Parts

General

GDB include GDB stab debugging (-g) support

UseBrowser include Browser (-b) support

B.3 Leave Out specific Parts

Leaving out parts of the compiler can be useful if you want to create a compiler which should also run on systems with less memory requirements (for example a real mode version compiled with Turbo Pascal).

General

NoOpt will leave out the optimizer

I386 specific

The following defines apply only to the i386 version of the compiler.

NoAg386Int No Intel styled assembler (for MASM/TASM) writer

`NoAg386Nsm` No NASM assembler writer

`NoAg386Att` No AT&T assembler (for the GNU AS) writer

`NoRA386Int` No Intel assembler parser

`NoRA386Dir` No direct assembler parser

`NoRA386Att` No AT&T assembler parser

M68k specific

The following defines apply only to the M68k version of the compiler.

`NoAg68kGas` No gas asm writer

`NoAg68kMit` No mit asm writer

`NoAg68kMot` No mot asm writer

`NoRA68kMot` No Motorola assembler parser

Appendix C

Location of the code generator functions

This appendix describes where to find the functions of the code generator. The file names are given for the i386, for the m68k rename the 386 to 68k

cg386con Constant generation

- **secondordconst**
- **secondrealconst**
- **secondstringconst**
- **secondfixconst**
- **secondsetconst**
- **secondniln**

cg386mat Mathematic functions

- **secondmoddiv**
- **secondshlshr**
- **secondumminus**
- **secondnot**

cg386cnv Type conversion functions

- **secondtypeconv**
- **secondis**
- **secondas**

cg386add Add/concat functions

- **secondadd**

cg386mem Memory functions

- **secondvecn**
- **secondaddr**
- **seconddoubleaddr**

secondsimplenewdispose
secondhnewn
secondhdisposen
secondselfn
secondwith
secondloadvmt
secondsubscriptn
secondderef

cg386flw Flow functions

secondifn
second_while_repeatn
secondfor
secondcontinuen
secondbreakn
secondexitn
secondlabel
secondgoto
secondtryfinally
secondtryexcept
secondraise
secondfail

cg386ld Load/Store functions

secondload
secondassignment
secondfuncret

cg386set Set functions

secondcase
secondin

cg386cal Call/inline functions

secondparacall
secondcall
secondprocinline
secondinline

cgi386 Main secondpass handling

secondnothing
seconderror
secondasm
secondblockn
secondstatement