NitroPascal Compiler DESIGN

VERSION: 1.0
DATE: 2025-10-10
STATUS: Definitive Architecture Guide

TABLE OF CONTENTS

---

1. VISION & PHILOSOPHY

1.1 What is NitroPascal?

NitroPascal is a **real compiler** that compiles Delphi/Object Pascal source code to machine code, but instead of emitting assembly or machine code directly, it emits **C++20 code as an intermediate representation (IR)**.

Key Points:

- C++20 is used as "portable assembly"
- The C++ compiler (zig/c++) handles final machine code generation
- Users write Delphi, get native cross-platform binaries

1.2 Why This Approach?

Traditional Compiler:

```text
Delphi Source → Parser → Semantic Analysis → x86/ARM Assembly → Machine Code
```

NitroPascal Compiler:

```text
Delphi Source → Parser → Semantic Analysis → C++20 Code → zig/c++ → Machine Code
```

Benefits:

1. **Cross-Platform for Free** - C++ compiles everywhere (Windows, Linux, macOS, embedded, WASM)
2. **Leverage Existing Optimizers** - clang are world-class optimizers
3. **No Assembly Required** - Don't need to write x86/ARM/etc backends
4. **Portable** - C is truly portable assembly
5. **Interop** - Easy to call C/C++ libraries, and be called from C/C++

Precedent: This strategy is used by:

- Early C++ (cfront → C)
- Nim compiler (Nim → C)
- Vala compiler (Vala → C)
- Many other successful compilers

1.3 The Problem We're Solving

Before (Complex Architecture):

- 11+ units for code generation
- Complex logic mapping Delphi constructs to C++
- Hard to maintain
- Easy to introduce bugs
- File corruption from context exhaustion

After (Simple Architecture):

- Delphi semantics are **wrapped in C++ RTL**
- Code generator is **trivial syntax translation**
- Just map AST nodes to RTL function calls
- Maintainable, simple, correct

---

2. ARCHITECTURE OVERVIEW

2.1 Full Pipeline

```text
┌─────────────────────────────┐
│  Delphi Source Code         │
│  (test.pas)                 │
└─────────────┬───────────────┘
              │
              ▼
     ┌────────────┐
     │ DelphiAST  │  (existing parser - handles full Object Pascal)
     └─────┬──────┘
           │
           ▼
     ┌────────────┐
     │    JSON    │  (AST representation)
     └─────┬──────┘
           │
           ▼
┌─────────────────────┐
│  NitroPascal        │
│  Code Generator     │  (THIS IS WHAT WE BUILD)
│  (reads JSON,       │
│   emits C++20)      │
└─────────┬───────────┘
          │
          ▼
     ┌────────────┐
     │ C++20 Code │  (generated .cpp/.h files)
     └─────┬──────┘
           │
           ▼
     ┌───────────────────────────────────────────────────┐
     │   Links                                           │
     │   Against                                         │
     │  RTL (.lib) │ (NitroPascal Runtime Library in C++)│
     └─────┬─────────────────────────────────────────────┘
           │
           ▼
     ┌─────────────────────────┐
     │ C++ Compiler │ (zig/c++)│
     └─────┬───────────────────┘
           │
           ▼
     ┌──────────────────────────────────┐
     │ Machine Code│ (native executable)│
     └──────────────────────────────────┘
```

2.2 Component Responsibilities

DelphiAST (Existing):

- ✅ Tokenization/Lexing
- ✅ Parsing Delphi syntax
- ✅ Building AST
- ✅ Outputting JSON representation

Code Generator (We Build This):

- Read DelphiAST JSON
- Walk AST nodes
- Emit C++20 code that calls RTL functions
- **This is deliberately simple - just syntax translation**

C++ Runtime Library / RTL (We Build This):

- Implements ALL Delphi semantics in C++20
- Provides functions/classes that behave exactly like Delphi
- **This is where the complexity lives**
- Header-only where possible (templates)
- Compiled library for complex runtime features

C++ Compiler (Existing):

- ✅ Compiles C++20 to machine code
- ✅ Optimizes
- ✅ Links

2.3 Data Flow Example

Input Delphi:

```
program Hello;
begin
  WriteLn('Hello, World!');
end.
```

DelphiAST JSON (simplified):

```json
{
  "type": "PROGRAM",
  "name": "Hello",
  "children": [
    {
      "type": "STATEMENTS",
      "children": [
        {
          "type": "CALL",
          "children": [
            {"type": "IDENTIFIER", "name": "WriteLn"},
            {
              "type": "EXPRESSIONS",
              "children": [
                {"type": "LITERAL", "value": "Hello, World!", "literalType": "string"}
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

Generated C++:

```cpp
#include "nitropascal_rtl.h"

int main() {
    np::WriteLn("Hello, World!");
    return 0;
}
```

Compiled Result:

- Native executable for target platform
- Runs on Windows/Linux/macOS/etc.

---

3. THE KEY INSIGHT: RTL WRAPPING STRATEGY

3.1 The Fundamental Breakthrough

Instead of:

- Code generator contains complex logic to map Delphi semantics to C++
- Generator must understand operator precedence, type conversions, control flow differences, etc.
- 11+ units of complex translation logic

We Do:

- **Wrap ALL Delphi semantics in C++ RTL functions/classes**
- Code generator is **trivial** - just emit function calls
- All complexity lives in RTL (which is written once, tested once, reused forever)

3.2 Why This Works

Delphi `for` loop:

```
for i := 1 to 10 do
  WriteLn(i);
```

Traditional approach (complex codegen):

```cpp
// Generator must know:
// - Delphi for-to is inclusive (includes 10)
// - Range is evaluated once
// - Iterator can't be modified
// - Must handle expressions in range
for (int i = 1; i <= 10; i++) {  // Note: <=, not <
    std::cout << i << std::endl;
}
```

RTL wrapping approach (trivial codegen):

RTL provides:

```cpp
// RTL: nitropascal_rtl.h
template<typename Func>
void ForLoop(int start, int end, Func body) {
    for (int i = start; i <= end; i++) {
        body(i);
    }
}
```

Generator just emits:

```cpp
np::ForLoop(1, 10, [&](int i) {
    np::WriteLn(i);
});
```

Result:

- ✅ Delphi semantics guaranteed (inclusive range, etc.)
- ✅ Simple codegen (just emit function call)
- ✅ All complexity in RTL (written once, correct once)

3.3 This Works For EVERYTHING

Control Flow → RTL Functions:

- `for...to` → `ForLoop(start, end, lambda)`
- `for...downto` → `ForLoopDownto(start, end, lambda)`
- `while...do` → `WhileLoop(condition_lambda, body_lambda)`
- `repeat...until` → `RepeatUntil(body_lambda, condition_lambda)`
- `with` → `WithScope(object, body_lambda)`
- `try...except...finally` → `TryExceptFinally(...)`

Types → RTL Classes:

- `String` → `np::String` class (UTF-16, 1-based indexing, Delphi semantics)
- `array of T` → `np::DynArray<T>` (1-based, SetLength, Copy)
- `set of T` → `np::Set<T>`
- `TList<T>` → `np::TList<T>` (wraps std::vector)

Operators → RTL Functions:

- `div` → `np::Div(a, b)` (integer division)
- `mod` → `np::Mod(a, b)`
- `shl` → `np::Shl(a, n)`
- `shr` → `np::Shr(a, n)`
- `in` → `np::In(element, set)`

Functions → RTL Functions:

- `WriteLn(...)` → `np::WriteLn(...)`
- `Length(s)` → `np::Length(s)`
- `SetLength(arr, n)` → `np::SetLength(arr, n)`
- `Copy(s, i, len)` → `np::Copy(s, i, len)` (1-based!)
- `IntToStr(i)` → `np::IntToStr(i)`

3.4 Code Generator Becomes Trivial

The entire code generator is just:

```
procedure TWalkAST(ANode: TJSONObject);
var
  LNodeType: string;
begin
  LNodeType := GetNodeType(ANode);

  case LNodeType of
    'CALL': EmitFunctionCall(ANode);      // Just emit: FunctionName(args)
    'FOR': EmitForLoop(ANode);            // Just emit: np::ForLoop(start, end, lambda)
    'WHILE': EmitWhileLoop(ANode);        // Just emit: np::WhileLoop(condition, body)
    'LITERAL': EmitLiteral(ANode);        // Just emit: the literal value
    'IDENTIFIER': EmitIdentifier(ANode);  // Just emit: the identifier name
    'ADD': EmitBinaryOp('+', ANode);      // Just emit: (left + right)
    // etc...
  end;
end;
```

That's it. No complex logic. Just map AST node types to output syntax.

---

4. THE C++ RUNTIME LIBRARY (RTL)

4.1 RTL Organization

```text
nitropascal_rtl/
├── include/
│   └── nitropascal/
│       ├── core.h           // Core definitions
│       ├── io.h             // Write/WriteLn/ReadLn
│       ├── string.h         // String class (UTF-16)
│       ├── containers.h     // DynArray, TList, TDictionary
│       ├── controlflow.h    // ForLoop, WhileLoop, etc.
│       ├── operators.h      // Div, Mod, Shl, Shr, In
│       ├── system.h         // Length, SetLength, Copy, etc.
│       └── nitropascal_rtl.h // Main include (includes all above)
└── src/
    ├── string.cpp           // String implementation (if not header-only)
    ├── containers.cpp       // Container implementations
    └── system.cpp           // System function implementations
```

4.2 Core RTL Examples

4.2.1 I/O Functions (io.h)

```cpp
#pragma once
#include <iostream>
#include <string>

namespace np {
    // Write (no newline)
    template<typename... Args>
    void Write(Args&&... args) {
        (std::cout << ... << args);
    }

    // WriteLn (with newline)
    template<typename... Args>
    void WriteLn(Args&&... args) {
        (std::cout << ... << args) << std::endl;
    }

    // WriteLn with no args (empty line)
    inline void WriteLn() {
        std::cout << std::endl;
    }

    // ReadLn
    template<typename T>
    void ReadLn(T& value) {
        std::cin >> value;
    }
}
```

Why variadic templates:

- Handles any number of arguments: `WriteLn(x)`, `WriteLn(x, y, z)`
- Handles any types: `WriteLn("X=", 42, " Y=", 3.14)`
- Uses C++17 fold expressions: `(std::cout << ... << args)`

4.2.2 Control Flow (controlflow.h)

```cpp
#pragma once
#include <functional>

namespace np {
    // for i := start to end do body
    template<typename Func>
    void ForLoop(int start, int end, Func body) {
        for (int i = start; i <= end; i++) {
            body(i);
        }
    }

    // for i := start downto end do body
    template<typename Func>
    void ForLoopDownto(int start, int end, Func body) {
        for (int i = start; i >= end; i--) {
            body(i);
        }
    }

    // while condition do body
    template<typename CondFunc, typename BodyFunc>
    void WhileLoop(CondFunc condition, BodyFunc body) {
        while (condition()) {
            body();
        }
    }

    // repeat body until condition
    template<typename BodyFunc, typename CondFunc>
    void RepeatUntil(BodyFunc body, CondFunc condition) {
        do {
            body();
        } while (!condition());
    }
}
```

Why lambdas:

- Captures Delphi semantics exactly
- Range evaluated once (start/end can't change)
- Iterator can't be modified in body
- Simple to generate from codegen

4.2.3 String Class (string.h - sketch)

```cpp
#pragma once
#include <string>
#include <cstdint>

namespace np {
    // Delphi String = UTF-16, 1-based indexing
    class String {
    private:
        std::u16string data_;

    public:
        String() = default;
        String(const char16_t* s) : data_(s) {}
        String(const std::u16string& s) : data_(s) {}

        // 1-based indexing (Delphi style)
        char16_t operator[](int index) const {
            return data_[index - 1];  // Convert to 0-based
        }

        int Length() const {
            return static_cast<int>(data_.length());
        }

        String operator+(const String& other) const {
            return String(data_ + other.data_);
        }

        // For std::cout output
        friend std::ostream& operator<<(std::ostream& os, const String& s);

        // More Delphi string operations...
    };

    // Global functions that work on String
    int Length(const String& s);
    String Copy(const String& s, int start, int count);
    int Pos(const String& substr, const String& s);
    // etc...
}
```

Key Points:

- UTF-16 internally (matches Delphi)
- 1-based indexing (Delphi convention)
- Implicit conversions where needed
- Provides all Delphi string functions

4.2.4 Operators (operators.h)

```cpp
#pragma once

namespace np {
    // Integer division (Delphi: div)
    inline int Div(int a, int b) {
        return a / b;  // C++ / is integer division for ints
    }

    // Modulo (Delphi: mod)
    inline int Mod(int a, int b) {
        return a % b;
    }

    // Shift left (Delphi: shl)
    inline int Shl(int value, int shift) {
        return value << shift;
    }

    // Shift right (Delphi: shr)
    inline int Shr(int value, int shift) {
        return value >> shift;
    }

    // Set membership (Delphi: in)
    template<typename T, typename SetType>
    bool In(const T& element, const SetType& set) {
        return set.contains(element);
    }
}
```

4.3 Type Mappings

Delphi Type	C++ RTL Type	Notes
`Integer`	`int32_t`	Fixed size
`Cardinal`	`uint32_t`	Fixed size
`Int64`	`int64_t`	Fixed size
`Byte`	`uint8_t`	Fixed size
`Word`	`uint16_t`	Fixed size
`Boolean`	`bool`	Direct mapping
`Char`	`char16_t`	UTF-16
`String`	`np::String`	UTF-16, 1-based
`Double`	`double`	Direct mapping

Delphi Type	C++ RTL Type	Notes
`Single`	`float`	Direct mapping
`Pointer`	`void*`	Direct mapping
`array of T`	`np::DynArray<T>`	1-based, dynamic
`array[a..b] of T`	`std::array<T, size>` or `T[size]`	Static
`set of T`	`np::Set<T>`	Bitset or set
`record`	`struct`	Direct mapping
`class`	`class`	With RTL support for constructors/destructors

---

5. CODE GENERATOR DESIGN

5.1 High-Level Structure

Single Unit Approach:

```
unit NitroPascal.CodeGen;

type
  TCodeGenerator = class
  private
    FOutput: TStringBuilder;
    FIndentLevel: Integer;

    // Helper methods
    procedure Emit(const AText: string);
    procedure EmitLine(const AText: string);
    procedure IncIndent;
    procedure DecIndent;

    // Node walking
    procedure WalkNode(const ANode: TJSONObject);

    // Section processors (organized by Delphi concerns)
    procedure ProcessProgram(const ANode: TJSONObject);
    procedure ProcessUnit(const ANode: TJSONObject);
    procedure ProcessConstants(const ANode: TJSONObject);
    procedure ProcessTypes(const ANode: TJSONObject);
    procedure ProcessVariables(const ANode: TJSONObject);
    procedure ProcessFunctions(const ANode: TJSONObject);
    procedure ProcessStatements(const ANode: TJSONObject);

    // Statement emitters
    procedure EmitCall(const ANode: TJSONObject);
    procedure EmitAssignment(const ANode: TJSONObject);
    procedure EmitFor(const ANode: TJSONObject);
    procedure EmitWhile(const ANode: TJSONObject);
    procedure EmitRepeat(const ANode: TJSONObject);
    procedure EmitIf(const ANode: TJSONObject);
    procedure EmitCase(const ANode: TJSONObject);

    // Expression emitters
    function EmitExpression(const ANode: TJSONObject): string;
    function EmitBinaryOp(const AOp: string; const ANode: TJSONObject): string;
    function EmitLiteral(const ANode: TJSONObject): string;
    function EmitIdentifier(const ANode: TJSONObject): string;

  public
    constructor Create;
    destructor Destroy; override;

    function GenerateFromJSON(const AJSON: string): string;
  end;
```

5.2 Organization by Delphi Concerns

Delphi program structure:

```
program MyProgram;

const
  // Constants section

type
  // Type declarations section

var
  // Variable declarations section

// Functions and procedures

begin
  // Main program block statements
end.
```

Code generator mirrors this:

- `ProcessConstants()` - handles const section
- `ProcessTypes()` - handles type section
- `ProcessVariables()` - handles var section
- `ProcessFunctions()` - handles procedure/function declarations
- `ProcessStatements()` - handles statement blocks (begin/end)

5.3 Core Algorithm

```
procedure TCodeGenerator.WalkNode(const ANode: TJSONObject);
var
  LNodeType: string;
begin
  LNodeType := GetNodeType(ANode);

  case LNodeType of
    'PROGRAM': ProcessProgram(ANode);
    'UNIT': ProcessUnit(ANode);
    'LIBRARY': ProcessLibrary(ANode);
    'CONSTANTS': ProcessConstants(ANode);
    'TYPES': ProcessTypes(ANode);
    'VARIABLES': ProcessVariables(ANode);
    'FUNCTION', 'PROCEDURE': ProcessFunction(ANode);
    'STATEMENTS': ProcessStatements(ANode);
    'CALL': EmitCall(ANode);
    'ASSIGN': EmitAssignment(ANode);
    'FOR': EmitFor(ANode);
    'WHILE': EmitWhile(ANode);
    'REPEAT': EmitRepeat(ANode);
    'IF': EmitIf(ANode);
    'CASE': EmitCase(ANode);
    // etc...
  else
    raise Exception.CreateFmt('Unknown node type: %s', [LNodeType]);
  end;
end;
```

5.4 Simple Emit Examples

5.4.1 Function Call

Delphi:

```
WriteLn('Hello', 42);
```

JSON (simplified):

```json
{
  "type": "CALL",
  "children": [
    {"type": "IDENTIFIER", "name": "WriteLn"},
    {
      "type": "EXPRESSIONS",
      "children": [
        {"type": "LITERAL", "value": "Hello", "literalType": "string"},
        {"type": "LITERAL", "value": "42", "literalType": "integer"}
      ]
    }
  ]
}
```

Code Generator:

```
procedure TCodeGenerator.EmitCall(const ANode: TJSONObject);
var
  LChildren: TJSONArray;
  LFuncName: string;
  LArgs: string;
begin
  LChildren := GetNodeChildren(ANode);

  // First child is function name
  LFuncName := GetNodeAttribute(LChildren[0], 'name');

  // Second child is arguments
  LArgs := EmitArguments(LChildren[1]);

  // Emit: np::FunctionName(args);
  EmitLine(Format('np::%s(%s);', [LFuncName, LArgs]));
end;
```

Output:

```cpp
np::WriteLn("Hello", 42);
```

5.4.2 For Loop

Delphi:

```
for i := 1 to 10 do
  WriteLn(i);
```

JSON (simplified):

```json
{
  "type": "FOR",
  "children": [
    {"type": "IDENTIFIER", "name": "i"},
    {"type": "FROM", "children": [{"type": "LITERAL", "value": "1"}]},
    {"type": "TO", "children": [{"type": "LITERAL", "value": "10"}]},
    {
      "type": "STATEMENTS",
      "children": [
        {"type": "CALL", ...}
      ]
    }
  ]
}
```

Code Generator:

```
procedure TCodeGenerator.EmitFor(const ANode: TJSONObject);
var
  LIterator: string;
  LStart: string;
  LEnd: string;
  LBody: string;
begin
  // Extract iterator, start, end
  LIterator := ...;
  LStart := EmitExpression(FindNode('FROM'));
  LEnd := EmitExpression(FindNode('TO'));

  // Emit: np::ForLoop(start, end, [&](int iterator) {
  EmitLine(Format('np::ForLoop(%s, %s, [&](int %s) {', [LStart, LEnd, LIterator]));
  IncIndent;

  // Emit body statements
  ProcessStatements(FindNode('STATEMENTS'));

  DecIndent;
  EmitLine('});');
end;
```

Output:

```cpp
np::ForLoop(1, 10, [&](int i) {
    np::WriteLn(i);
});
```

That's it! Simple mapping.

---

6. COMPLETE MAPPING REFERENCE

6.1 Control Flow Constructs

FOR...TO

Delphi:

```
for i := 1 to 10 do
  Statement;
```

C++ Output:

```cpp
np::ForLoop(1, 10, [&](int i) {
    Statement;
});
```

FOR...DOWNTO

Delphi:

```
for i := 10 downto 1 do
  Statement;
```

C++ Output:

```cpp
np::ForLoopDownto(10, 1, [&](int i) {
    Statement;
});
```

WHILE...DO

Delphi:

```
while condition do
  Statement;
```

C++ Output:

```cpp
np::WhileLoop([&]() { return condition; }, [&]() {
    Statement;
});
```

REPEAT...UNTIL

Delphi:

```
repeat
  Statement;
until condition;
```

C++ Output:

```cpp
np::RepeatUntil([&]() {
    Statement;
}, [&]() { return condition; });
```

IF...THEN...ELSE

Delphi:

```
if condition then
  TrueStatement
else
  FalseStatement;
```

C++ Output:

```cpp
if (condition) {
    TrueStatement;
} else {
    FalseStatement;
}
```

Note: IF is direct C++ syntax - no need for RTL wrapper

CASE...OF

Delphi:

```
case value of
  1: Statement1;
  2: Statement2;
else
  ElseStatement;
end;
```

C++ Output:

```cpp
switch (value) {
    case 1:
        Statement1;
        break;
    case 2:
        Statement2;
        break;
    default:
        ElseStatement;
        break;
}
```

Note: CASE maps to switch - direct C++ syntax

6.2 Operators

Delphi	C++ Emission	Notes
`+`	`+`	Direct
`-`	`-`	Direct
`*`	`*`	Direct
`/`	`/`	Float division (direct)
`div`	`np::Div(a, b)`	Integer division
`mod`	`np::Mod(a, b)`	Modulo
`shl`	`np::Shl(a, n)`	Shift left
`shr`	`np::Shr(a, n)`	Shift right
`and`	`&&`	Logical AND (direct)
`or`	`||`	Logical OR (direct)
`xor`	`^`	Bitwise XOR (direct)
`not`	`!`	Logical NOT (direct)
`=`	`==`	Equality (direct)
`<>`	`!=`	Inequality (direct)
`<`	`<`	Less than (direct)
`>`	`>`	Greater than (direct)
`<=`	`<=`	Less or equal (direct)

Delphi	C++ Emission	Notes
`>=`	`>=`	Greater or equal (direct)
`:=`	`=`	Assignment (direct)
`in`	`np::In(elem, set)`	Set membership

6.3 Type Declarations

Record

Delphi:

```
type
  TPoint = record
    X: Integer;
    Y: Integer;
  end;
```

C++ Output:

```cpp
struct TPoint {
    int32_t X;
    int32_t Y;
};
```

Class (Simple)

Delphi:

```
type
  TMyClass = class
  private
    FValue: Integer;
  public
    procedure SetValue(AVal: Integer);
    function GetValue: Integer;
  end;
```

C++ Output:

```cpp
class TMyClass {
private:
    int32_t FValue;
public:
    void SetValue(int32_t AVal);
    int32_t GetValue();
};
```

Dynamic Array

Delphi:

```
var
  arr: array of Integer;
```

C++ Output:

```cpp
np::DynArray<int32_t> arr;
```

Static Array

Delphi:

```
var
  arr: array[0..9] of Integer;
```

C++ Output:

```cpp
std::array<int32_t, 10> arr;  // or: int32_t arr[10];
```

6.4 Functions and Procedures

Procedure

Delphi:

```
procedure DoSomething(AValue: Integer);
begin
  WriteLn(AValue);
end;
```

C++ Output:

```cpp
void DoSomething(int32_t AValue) {
    np::WriteLn(AValue);
}
```

Function

Delphi:

```
function Add(A, B: Integer): Integer;
begin
  Result := A + B;
end;
```

C++ Output:

```cpp
int32_t Add(int32_t A, int32_t B) {
    int32_t Result;
    Result = A + B;
    return Result;
}
```

Note: Delphi's implicit `Result` variable becomes explicit local variable + return

6.5 String Operations

Delphi	C++ RTL Call	Notes
`Length(s)`	`np::Length(s)`	String length
`Copy(s, i, len)`	`np::Copy(s, i, len)`	Substring (1-based!)
`Pos(sub, s)`	`np::Pos(sub, s)`	Find substring
`s[i]`	`s[i]`	Index (1-based in np::String)
`s + t`	`s + t`	Concatenation (overloaded +)
`IntToStr(i)`	`np::IntToStr(i)`	Convert int to string
`StrToInt(s)`	`np::StrToInt(s)`	Convert string to int
`Format(fmt, args)`	`np::Format(fmt, args)`	String formatting

6.6 Program/Unit/Library Structure

Program

Delphi:

```
program MyProgram;
begin
  WriteLn('Hello');
end.
```

C++ Output:

```cpp
#include "nitropascal_rtl.h"

int main() {
    np::WriteLn("Hello");
    return 0;
}
```

Unit

Delphi:

```
unit MyUnit;

interface

procedure DoSomething;

implementation

procedure DoSomething;
begin
  WriteLn('Hello');
end;

end.
```

C++ Output (MyUnit.h):

```cpp
#pragma once
#include "nitropascal_rtl.h"

namespace MyUnit {
    void DoSomething();
}
```

C++ Output (MyUnit.cpp):

```cpp
#include "MyUnit.h"

namespace MyUnit {
    void DoSomething() {
        np::WriteLn("Hello");
    }
}
```

Library

Delphi:

```
library MyLibrary;

function Add(A, B: Integer): Integer; export;
begin
  Result := A + B;
end;

exports
  Add;

end.
```

C++ Output:

```cpp
#include "nitropascal_rtl.h"

extern "C" {
    __declspec(dllexport) int32_t Add(int32_t A, int32_t B) {
        int32_t Result;
        Result = A + B;
        return Result;
    }
}
```

---

7. IMPLEMENTATION ROADMAP

Phase 1: Proof of Concept

Goal: Get one complete example working end-to-end

Tasks:

1. Create minimal RTL (just WriteLn)
2. Create simple code generator (just handles PROGRAM + CALL + LITERAL)
3. Test with test01.json example
4. Compile and run

Deliverables:

- `nitropascal_rtl.h` (just WriteLn)
- `NitroPascal.CodeGen.pas` (minimal walker)
- Generated C++ compiles and runs

Success Criteria:

```
program test01;
begin
  WriteLn('Hello world, welcome to NitroPascal!');
end.
```

→ Compiles to C++ → Executes correctly

Phase 2: Core Features

Goal: Support basic procedural programming

Features to Add:

- Variables (var section)
- Constants (const section)
- Assignment statements
- Arithmetic expressions (+, -, *, /, div, mod)
- Comparison operators (=, <>, <, >, ⇐, >=)
- Boolean operators (and, or, not)
- If/then/else
- For loops (to/downto)
- While loops
- Repeat/until loops
- Functions and procedures
- Basic types (Integer, Boolean, Double, String)

RTL Extensions:

- ForLoop, ForLoopDownto
- WhileLoop, RepeatUntil
- Basic String class
- Type conversion functions (IntToStr, StrToInt)

Phase 3: Advanced Features

Goal: Support advanced Delphi features

Features to Add:

- Records (struct)
- Static arrays
- Dynamic arrays
- Case statements
- Type declarations
- Units (separate compilation)
- String operations (Length, Copy, Pos, etc.)
- Set types
- Pointers
- With statements

RTL Extensions:

- Full String class
- DynArray
- Set
- Memory management functions

Phase 4: Object-Oriented

Goal: Support classes and objects

Features to Add:

- Class declarations
- Constructors/destructors
- Methods
- Properties (via getter/setter methods)
- Inheritance
- Virtual methods
- Interfaces (maybe)

RTL Extensions:

- Base object infrastructure
- Memory management for objects
- RTTI support (if needed)

Phase 5: Standard Library (Ongoing)

Goal: Implement Delphi RTL equivalents

Features to Add:

- TList
- TDictionary<K,V>
- TStringList
- File I/O (TFile, TDirectory)
- Exception handling
- Threading (TThread)
- More...

---

8. COMPLETE EXAMPLES

8.1 Example 1: Simple Program

Delphi Source:

```
program Hello;
begin
  WriteLn('Hello, World!');
end.
```

Generated C++:

```cpp
#include "nitropascal_rtl.h"

int main() {
    np::WriteLn("Hello, World!");
    return 0;
}
```

8.2 Example 2: Variables and Expressions

Delphi Source:

```
program Math;
var
  x, y: Integer;
  result: Integer;
begin
  x := 10;
  y := 20;
  result := x + y * 2;
  WriteLn('Result: ', result);
end.
```

Generated C++:

```cpp
#include "nitropascal_rtl.h"

int main() {
    int32_t x;
    int32_t y;
    int32_t result;

    x = 10;
    y = 20;
    result = x + y * 2;
    np::WriteLn("Result: ", result);

    return 0;
}
```

8.3 Example 3: For Loop

Delphi Source:

```
program Loop;
var
  i: Integer;
begin
  for i := 1 to 10 do
    WriteLn('Count: ', i);
end.
```

Generated C++:

```cpp
#include "nitropascal_rtl.h"

int main() {
    np::ForLoop(1, 10, [&](int i) {
        np::WriteLn("Count: ", i);
    });

    return 0;
}
```

8.4 Example 4: Function

Delphi Source:

```
program Functions;

function Add(A, B: Integer): Integer;
begin
  Result := A + B;
end;

var
  x: Integer;
begin
  x := Add(5, 7);
  WriteLn('5 + 7 = ', x);
end.
```

Generated C++:

```cpp
#include "nitropascal_rtl.h"

int32_t Add(int32_t A, int32_t B) {
    int32_t Result;
    Result = A + B;
    return Result;
}

int main() {
    int32_t x;

    x = Add(5, 7);
    np::WriteLn("5 + 7 = ", x);

    return 0;
}
```

8.5 Example 5: Record

Delphi Source:

```
program Records;

type
  TPoint = record
    X: Integer;
    Y: Integer;
  end;

var
  p: TPoint;
begin
  p.X := 10;
  p.Y := 20;
  WriteLn('Point: (', p.X, ', ', p.Y, ')');
end.
```

Generated C++:

```cpp
#include "nitropascal_rtl.h"

struct TPoint {
    int32_t X;
    int32_t Y;
};

int main() {
    TPoint p;

    p.X = 10;
    p.Y = 20;
    np::WriteLn("Point: (", p.X, ", ", p.Y, ")");

    return 0;
}
```

8.6 Example 6: Unit

Delphi Source (MathUtils.pas):

```
unit MathUtils;

interface

function Square(X: Integer): Integer;

implementation

function Square(X: Integer): Integer;
begin
  Result := X * X;
end;

end.
```

Delphi Source (Main.pas):

```
program Main;
uses MathUtils;

begin
  WriteLn('Square of 5: ', Square(5));
end.
```

Generated C++ (MathUtils.h):

```cpp
#pragma once
#include "nitropascal_rtl.h"

namespace MathUtils {
    int32_t Square(int32_t X);
}
```

Generated C++ (MathUtils.cpp):

```cpp
#include "MathUtils.h"

namespace MathUtils {
    int32_t Square(int32_t X) {
        int32_t Result;
        Result = X * X;
        return Result;
    }
}
```

Generated C++ (Main.cpp):

```cpp
#include "nitropascal_rtl.h"
#include "MathUtils.h"

int main() {
    np::WriteLn("Square of 5: ", MathUtils::Square(5));
    return 0;
}
```

---

9. DESIGN PRINCIPLES

9.1 Correctness Over Performance

Principle: The generated C++ must be **correct** first. Performance can be optimized later.

Example: It's okay to use lambdas and function calls instead of raw loops if it guarantees correct Delphi semantics.

9.2 Simple Code Generator

Principle: The code generator should be as simple as possible. All complexity belongs in the RTL.

Why:

- Simple codegen is easier to debug
- Simple codegen is less likely to have bugs
- Complexity in RTL can be unit-tested once
- Complex codegen is hard to maintain

9.3 RTL Wrapping Everything

Principle: When in doubt, wrap it in an RTL function.

Example:

- Not sure if Delphi `mod` is exactly C++ `%`? → Wrap it: `np::Mod(a, b)`
- Not sure about string indexing? → Wrap it: `np::String` class with 1-based indexing
- Not sure about loop semantics? → Wrap it: `np::ForLoop()`

Benefits:

- If behavior is wrong, fix it once in RTL
- Generated code stays simple
- Easy to extend later

9.4 Explicit Over Implicit

Principle: Make behavior explicit in generated code.

Example:

```cpp
// Good: Explicit Result variable
int32_t MyFunc() {
    int32_t Result;
    Result = 42;
    return Result;
}

// Bad: Implicit behavior hidden in RTL magic
int32_t MyFunc() {
    // Where is Result?
    return __np_implicit_result;
}
```

9.5 No Magic

Principle: Generated C++ should be readable and understandable.

Why: Users might need to debug the generated C++. It should be clear what's happening.

Example:

```cpp
// Good: Clear what's happening
np::ForLoop(1, 10, [&](int i) {
    np::WriteLn(i);
});

// Bad: Magic preprocessor macros
NP_FOR(i, 1, 10) {
    NP_WRITELN(i);
}
```

9.6 Compilation Unit Separation

Principle: Delphi units → C++ compilation units (separate .h/.cpp files)

Why:

- Matches Delphi's compilation model
- Faster compilation (parallel)
- Better organization
- Easier to debug

9.7 Namespaces for Units

Principle: Each Delphi unit becomes a C++ namespace

Why:

- Prevents name conflicts
- Matches Delphi's unit scoping
- Clear where functions come from

Example:

```cpp
// Unit System.SysUtils
namespace System {
namespace SysUtils {
    np::String IntToStr(int32_t value);
}
}

// Usage:
auto s = System::SysUtils::IntToStr(42);
```

9.8 Header-Only Where Possible

Principle: RTL templates should be header-only

Why:

- Easier to use (just include)
- Compiler can inline and optimize
- No linking issues

When Not: Complex implementations that don't need to be templates

9.9 Standard C++20

Principle: Use only standard C++20 features, no compiler extensions

Why:

- Maximum portability
- Works with zig/c++ (standard C++20)
- Future-proof

9.10 Error Messages

Principle: When code generation fails, provide helpful error messages

Example:

```
FErrorManager.AddError(
  NP_ERROR_CODEGEN,
  LNode.Line,
  LNode.Col,
  Format('Cannot generate code for unsupported node type: %s', [LNodeType])
);
```

---

10. CRITICAL SUCCESS FACTORS

10.1 What Makes This Work

1. **RTL wrapping** - All Delphi semantics in C++
2. **Simple codegen** - Just syntax translation
3. **DelphiAST** - Parsing is already solved
4. **C++ as IR** - Leverage existing compilers
5. **Incremental approach** - Build features one at a time

10.2 What Would Make This Fail

1. **Trying to do too much at once** - Start small
2. **Complex codegen** - Keep it simple, use RTL
3. **Perfect semantics** - Good enough is okay for v1
4. **Performance obsession** - Correctness first
5. **Feature creep** - Stick to core language first

10.3 How to Stay on Track

1. **Follow the roadmap** - Phase 1, then 2, then 3
2. **Test each feature** - Write test cases
3. **Keep codegen simple** - If it's complex, wrap it in RTL
4. **Read this DESIGN** - When stuck, re-read relevant sections
5. **One feature at a time** - Don't try to do everything

---

11. QUICK REFERENCE

11.1 For New Sessions

When starting a new session:

1. Read this DESIGN.md completely
2. Understand: RTL wrapping is the KEY
3. Understand: Codegen is just syntax translation
4. Check roadmap for current phase
5. Ask user what specific feature to implement

The pattern is always:

1. What Delphi construct?
2. What should C++ output be?
3. Does RTL need a wrapper? (usually yes)
4. Implement RTL wrapper first
5. Update codegen to emit calls to it
6. Test

11.2 When Stuck

If code generator is getting complex: → You're doing it wrong. Wrap it in RTL instead.

If unsure about semantics: → Test in real Delphi, match that behavior in RTL.

If file getting too large: → Split by concern (const, type, var, statements, etc.)

If losing context: → Update SESSION.md, stop and ask for guidance.

11.3 Remember

- **This is a compiler, not a transpiler**
- **C++ is our "assembly language"**
- **RTL wrapping makes codegen trivial**
- **Simple is better than clever**
- **Correctness over performance**
- **One feature at a time**

END OF DESIGN

This document is the definitive guide for NitroPascal development.

When in doubt, refer back to this document. It contains everything needed to build the compiler successfully.

Key Takeaway: Wrap Delphi semantics in C++ RTL. Code generator becomes trivial syntax translation. That's the entire architecture.