VS Code Extension for Pascaline

Implementation Assessment

Prepared for the Pascal-P6 Project

February 2026

Scope

1. Syntax Coloring
2. Cross-Referencing of Symbols (Go to Definition, Find References)
3. External Debugger Integration

1. Overview

This document describes the type and amount of work needed to produce a Visual Studio Code extension that supports the Pascaline programming language. Pascaline is a formal extension of ISO 7185 Pascal, adding classes, modules, exception handling, operator overloading, parallel programming constructs, container types, and a comprehensive standard library platform. The Pascal-P6 compiler is the reference implementation.

The extension would provide three major capabilities: syntax coloring for the editor, cross-referencing of symbols (go to definition, find all references, hover information), and integration with the Pascal-P6 debugger for source-level debugging within VS Code.

2. Syntax Coloring (TextMate Grammar)

2.1 Deliverable

A TextMate grammar file (.tmLanguage.json) that defines regex-based tokenization rules for Pascaline source files. This file is bundled with the VS Code extension and provides syntax highlighting in the editor.

2.2 Language Elements to Cover

ISO 7185 Pascal keywords:

```
program  begin  end  if  then  else  while  do  for  to  downto
repeat  until  case  of  with  goto  var  const  type  procedure
function  label  packed  array  record  set  file  nil  not  and
or  in  div  mod
```

Pascaline additional word-symbols (30+):

```
forward  module  uses  private  external  view  fixed  process
monitor  share  class  is  xor  overload  override  reference
joins  static  inherited  self  virtual  try  except  extends
on  result  operator  start  task  atom  property  channel
stream  out
```

Additional tokenization targets:

- Comments: { }, (* *), and ! line comments
- String literals with character escape sequences (Annex E)
- Numeric literals: decimal, hexadecimal ($), octal (&), binary (%)
- Underscore break character (_) in identifiers and numbers
- Operators and special symbols: `+ - * / = < > <> <= >= := .. @ ^`
- Block scoping constructs: program, module, class, process, monitor, share, channel, stream
- Built-in types: integer, real, char, boolean, text, string, pstring, byte, vector, matrix
- Standard procedures/functions: read, write, readln, writeln, new, dispose, etc.

2.3 Approach

Use an existing Pascal TextMate grammar as a starting point (several open-source grammars exist for ISO 7185/Turbo Pascal). Extend it substantially for the Pascaline-specific constructs. The grammar file uses regex patterns organized into repository rules, which are well-documented by the VS Code extension API.

2.4 Effort Estimate

Estimated: 1-2 weeks for one developer. The work is straightforward but thorough, covering a large number of keyword and syntax cases. Testing across the existing Pascaline codebase (compiler source, test files, sample programs) would validate coverage.

Complexity: Low to moderate.

3. Cross-Referencing / Symbol Navigation (Language Server)

3.1 Deliverable

A Language Server Protocol (LSP) implementation providing:

- Go to Definition - navigate to where a symbol is declared
- Find All References - locate every use of a symbol across files
- Hover Information - display type signatures and documentation on hover
- Document/Workspace Symbol Outline - structured view of symbols
- Signature Help - parameter hints when calling procedures/functions
- Code Completion - context-aware suggestions

3.2 Why This Is the Largest Component

Cross-referencing requires deep understanding of the Pascaline language semantics: scoping rules, the module/uses system, class inheritance hierarchies, overloaded procedure resolution, and the full type system. This is fundamentally a compiler-frontend problem. Unlike syntax coloring (which is regex-based and approximate), symbol resolution must be semantically correct.

3.3 Approach A: LSP Wrapping the Existing Compiler (Recommended)

Leverage the Pascal-P6 compiler (pcom) as the parsing and analysis engine. The compiler already parses the full Pascaline language and builds complete symbol tables with type information, scoping, and cross-references.

Work items:

- **Compiler modification:** Add a mode to pcom that emits structured symbol table data (definitions, references, types, scopes, file positions) in a machine-readable format such as JSON. This involves instrumenting the existing symbol table management code to output its contents. Estimated: 2-4 weeks.
- **LSP server framework:** Implement the LSP server, likely in TypeScript (the standard for VS Code language servers), that communicates with the editor via JSON-RPC and invokes the modified compiler to obtain symbol data. Estimated: 2-3 weeks.
- **LSP feature implementation:** Map the compiler symbol data to each LSP feature (textDocument/definition, textDocument/references, textDocument/hover, textDocument/completion, textDocument/documentSymbol, etc.). Each feature requires parsing compiler output and formatting LSP responses. Estimated: 6-8 weeks.
- **Incremental analysis:** For editor responsiveness, implement incremental or cached parsing so that the compiler is not re-invoked from scratch on every keystroke. This may involve file-level caching, dirty-file tracking, and background re-analysis. Estimated: 2-4 weeks.
- **Cross-file resolution:** Handle the Pascaline module/uses system to resolve symbols across compilation units. The compiler already handles this during compilation; the work is exposing it through the LSP. Estimated: 1-2 weeks (included in above).
- **Testing:** Validate across the full Pascal-P6 codebase (compiler source, libraries, test programs). Estimated: 2-3 weeks.

3.4 Approach B: Standalone Parser (Not Recommended)

An alternative would be writing a new parser specifically for IDE use, for example using Tree-sitter or a hand-written parser in TypeScript. This would mean re-implementing a significant portion of the compiler frontend: lexer, parser, symbol table builder, type checker, and scope resolution. Given that the Pascal-P6 compiler already handles the full grammar correctly, this approach would duplicate effort and risk divergence from the actual language specification.

Estimated: 4-8 months. Not recommended.

3.5 Effort Estimate (Approach A)

Estimated: 3-6 months for one experienced developer familiar with both the Pascal-P6 compiler internals and the LSP specification.

Complexity: High.

4. External Debugger Integration (Debug Adapter Protocol)

4.1 Deliverable

A Debug Adapter Protocol (DAP) implementation that connects VS Code to the Pascal-P6 debugger, providing a graphical debugging experience within the editor.

4.2 Existing Debugger Capabilities

Pascal-P6 already includes a comprehensive source-level debugger built into pint (the P-code interpreter). The debugger supports:

- Breakpoints by source line number or routine name (b command)
- Tracepoints at source lines or instructions (tp, tpi commands)
- Single-stepping: source-level (s), instruction-level (si), over routines (so)
- Step out / return from routine (ret command)
- Watch variables (w command)
- Print expressions, locals, globals, parameters (p, pl, pg, pp commands)
- Stack traces / call frame dumps (df command)
- Memory inspection and modification (d, e commands)
- Source listing at current position (l command)
- Instruction tracing (ti/nti commands)

The native code generator (pgen) also produces GDB-compatible debug information via GCC with -g3 flags, including source line diagnostics. This provides a second debugging path through GDB.

4.3 Target A: DAP Adapter for pint (Recommended First Target)

Write a Debug Adapter that drives the pint interpreter in debug mode via stdin/stdout, translating between the DAP protocol and pint debug commands.

DAP request mapping:

DAP Request	pint Command	Notes
setBreakpoints	b <line>	Map file/line to module/line
setFunctionBreakpoints	b <routine>	Direct mapping
continue	r (run)	Resume execution
next	so (step over)	Source-level step over
stepIn	s (step)	Source-level step into
stepOut	ret (return)	Return from routine
stackTrace	df (dump frames)	Parse frame output
scopes/variables	pl, pg, pp	Parse variable output
evaluate	p <expr>	Print expression value

Work items:

- **DAP server framework:** Implement the adapter in TypeScript using the vscode-debugadapter library, which handles the DAP protocol. Estimated: 1-2 weeks.
- **Process management:** Launch and manage the pint process, handle stdin/stdout communication, detect

prompts and parse output. Estimated: 1-2 weeks.

- **Command translation:** Map each DAP request to the corresponding sequence of pint commands and parse the text output back into structured DAP responses. Estimated: 2-3 weeks.
- **Source mapping:** Maintain the mapping between editor file/line positions and the module/line format that pint uses. Estimated: 1 week.
- **Testing:** Validate with sample programs and the compiler itself. Estimated: 1 week.

4.4 Target B: GDB via Native Code (pgen)

Since pgen compiles to native AMD64 code via GCC with -g3 debug info, GDB can already debug the resulting executables. VS Code has existing debug adapters for GDB (cppdbg, CodeLLDB). The work would be ensuring that the DWARF debug information correctly maps Pascaline source lines and verifying the experience works end-to-end.

Estimated: 2-4 weeks. This could serve as a complementary debugging option alongside the pint adapter.

4.5 Effort Estimate

Target A (pint DAP adapter): 4-6 weeks for one developer. Target B (GDB integration): 2-4 weeks additional.

Complexity: Moderate.

5. Summary and Total Effort

Component	Approach	Effort	Complexity
Syntax Coloring	TextMate grammar	1-2 weeks	Low-moderate
Symbol Navigation	LSP wrapping pcom	3-6 months	High
Debugger (pint)	DAP adapter	4-6 weeks	Moderate
Debugger (GDB)	Existing adapters	2-4 weeks	Low-moderate
Extension scaffold	package.json, etc.	1 week	Low

Total estimated effort: 5-9 months for one experienced developer

5.1 Recommended Implementation Order

The following phased approach allows incremental delivery of value:

- **Phase 1 (weeks 1-2):** Extension scaffolding and syntax coloring. This provides immediate visual benefit and establishes the extension infrastructure.
- **Phase 2 (weeks 3-8):** Debugger adapter for pint. This is high-value and moderate effort, giving developers interactive debugging within VS Code.
- **Phase 3 (months 3-9):** Language Server for symbol navigation. This is the largest and most complex component, but builds on the foundation of the earlier phases.
- **Phase 4 (optional):** GDB debugging support for natively compiled programs, providing a second debugging path.

5.2 Prerequisites and Dependencies

- Familiarity with the Pascal-P6 compiler internals (pcom symbol table structure)
- VS Code Extension API and TypeScript development
- Language Server Protocol specification
- Debug Adapter Protocol specification
- The Pascaline language specification (doc/pascaline.txt)
- The P6 compiler documentation (doc/the_p6_compiler.txt)

5.3 AI-Assisted Development Estimate

With an AI coding assistant such as Claude used as an active development partner, the estimates compress significantly. The biggest gains are on boilerplate and protocol code (TextMate grammar, LSP/DAP message handling, TypeScript scaffolding). The least compressible parts are architectural decisions about how to modify the compiler to export symbol data and how to handle incremental re-analysis -- those require someone who understands the P6 compiler internals.

Component	Without AI	With AI	Reason
Syntax coloring	1-2 weeks	2-3 days	AI generates bulk of .tmLanguage.json
LSP framework	2-3 weeks	1 week	Boilerplate-heavy; AI excels here
LSP features	6-8 weeks	3-4 weeks	Human judgment still needed
Compiler mod	2-4 weeks	2-3 weeks	Deep internals; less compressible

Incremental analysis	2-4 weeks	1-2 weeks	Architecture needs a human
DAP debugger	4-6 weeks	2-3 weeks	Protocol translation is mechanical
Testing	2-3 weeks	1-2 weeks	AI generates cases; human validates
Extension scaffold	1 week	1-2 days	Nearly fully automatable

Revised total with AI assistance: 3-4 months (down from 5-9 months)

The developer's role shifts from writing code to directing, reviewing, and testing. The bottleneck becomes understanding what to build, not building it.

5.4 Risk Factors

- **Incremental parsing performance:** The compiler was designed for batch compilation, not interactive use. Adapting it for responsive editor feedback may require significant caching or architectural changes.
- **Cross-module resolution:** The module/uses system requires multi-file analysis. Ensuring this works correctly and efficiently in an LSP context adds complexity.
- **Debugger output parsing:** The pint debugger communicates via text output. Robust parsing of this output across all edge cases requires careful implementation.