STUNIR Emitter Patterns Specification

Overview

This document captures the common patterns across all 4 STUNIR pipeline emitters (SPARK, Python, Rust, Haskell) to enable automated emitter generation.

Common Structure Across All Pipelines

1. File Organization

SPARK (Ada)

```
targets/spark/<category>/
├── <category>_emitter.ads    # Specification (interface)
├── <category>_emitter.adb    # Implementation (body)
└── <category>_emitter_main.adb  # CLI entry point (optional)
```

Python

```
targets/<category>/
├── __init__.py
├── emitter.py             # Main emitter class
└── types.py               # Type definitions (optional)
```

Rust

```
targets/rust/<category>/
├── mod.rs               # Main module
└── <sub_modules>.rs     # Additional modules
```

Haskell

```
targets/haskell/src/STUNIR/Emitters/
└── <Category>.hs        # Single file module
```

2. Core Components

All emitters contain these fundamental components:

A. Configuration Record/Struct

Defines target-specific settings:
- Architecture (ARM, x86, RISC-V, etc.)
- Feature flags (SIMD, floating-point, stdlib usage)
- Memory constraints (stack size, heap size)
- Optimization settings

SPARK:

```
type Config_Type is record
   Architecture : Architecture_Type;
   Feature_X    : Boolean;
   Size_Setting : Positive;
end record;
```

Python:

```
class EmitterConfig:
    def __init__(self, arch='arm', feature_x=False, size_setting=1024):
        self.arch = arch
        self.feature_x = feature_x
        self.size_setting = size_setting
```

Rust:

```
pub struct Config {
    pub architecture: Architecture,
    pub feature_x: bool,
    pub size_setting: usize,
}
```

Haskell:

```
data Config = Config
  { architecture :: Architecture
  , featureX :: Bool
  , sizeSetting :: Int
  }
```

B. Type Mapping System

Maps STUNIR IR types to target language types:

Common IR Types:
- `i8`, `i16`, `i32`, `i64` (signed integers)
- `u8`, `u16`, `u32`, `u64` (unsigned integers)
- `f32`, `f64` (floating point)
- `bool`, `char`, `string`
- `void`, `pointer`, `array`, `struct`

SPARK:

```
function Map_IR_Type_To_Target (IR_Type : IR_Data_Type) return Type_Name_String;
```

Python:

```
TYPE_MAP = {
    'i32': 'int32_t',
    'i64': 'int64_t',
    'f32': 'float',
    'bool': 'uint8_t',
}
```

Rust:

```rust
fn map_ir_type(ir_type: &str) -> &'static str {
    match ir_type {
        "i32" => "i32",
        "i64" => "i64",
        _ => "i32",
    }
}
```

Haskell:

```haskell
mapIRType :: Text -> Text
mapIRType "i32" = "int32_t"
mapIRType "i64" = "int64_t"
mapIRType _ = "int32_t"
```

C. Main Emission Functions

Module/Package Level:
- Input: IR module data
- Output: Generated files (header, source, build scripts)
- Returns: Result with file paths and hashes

Function Level:
- Input: IR function data
- Output: Function code as string
- Returns: Status/Result

Statement Level:
- Input: Single IR statement
- Output: Target language statement(s)
- Returns: Generated code string

D. File Generation and Tracking

All emitters track generated files with:
- Path (relative to output directory)
- Content hash (SHA-256)
- File size
- Timestamp

SPARK:

```
type Generated_File_Record is record
   Path     : Path_String;
   Hash     : Hash_String;
   Size     : Natural;
end record;
```

Python:

```
{
    'path': 'output.c',
    'sha256': 'abc123...',
    'size': 1024,
}
```

E. Error Handling

SPARK:

```
type Emitter_Status is (Success, Error_Invalid_Input, Error_Buffer_Overflow,
Error_IO);
```

Python:

```
class EmitterError(Exception):
    pass
```

Rust:

```
pub enum EmitterError {
    UnsupportedTarget(String),
    InvalidConfiguration(String),
    GenerationFailed(String),
}
```

Haskell:

```
data EmitterError
  = UnsupportedTarget Text
  | InvalidConfiguration Text
  | GenerationFailed Text
```

3. Standard Header Comments

All generated files include:

```
/* STUNIR Generated Code */
/* Module: <module_name> */
/* Category: <category> */
/* Generator: <SPARK|Python|Rust|Haskell> Pipeline */
/* Timestamp: <iso8601> */
/* DO-178C Level A compliant (SPARK only) */
```

4. Common Operations

A. Initialization/Setup

- Parse configuration
- Validate inputs
- Initialize state

B. IR Parsing

- Read IR JSON/structure
- Extract functions, types, data
- Build internal representation

C. Code Generation

- Module prologue (headers, imports)
- Type definitions
- Function declarations
- Function implementations
- Module epilogue

D. Output Formatting

- Indentation (4 spaces standard)
- Line wrapping (80-120 chars)
- Comment formatting
- Whitespace normalization

E. Testing Pattern

Each emitter has:
- Unit tests for type mapping
- Integration tests for full module emission
- Compliance tests (syntax validation)
- Hash consistency tests

5. Target-Specific Patterns

Assembly Emitters

- Prologue/epilogue generation
- Register allocation hints
- Stack frame management
- Architecture-specific instructions

Polyglot Emitters (C, Rust, Python output)

- Header file generation
- Include guards
- Standard library imports
- Build system integration

VM/Bytecode Emitters (JVM, WASM, BEAM)

- Instruction encoding
- Stack management
- Module format compliance

Lisp Dialect Emitters

- S-expression formatting
- Package/namespace definitions
- Macro definitions
- Proper indentation

Functional Emitters (Haskell, ML, F#)

- Type class definitions
- Pattern matching
- Algebraic data types

Emitter Categories

Current Categories

1. **assembly** - ARM, x86, MIPS, RISC-V
2. **polyglot** - C89, C99, Rust, Python, JavaScript
3. **embedded** - Bare-metal C for microcontrollers
4. **wasm** - WebAssembly text and binary
5. **lisp** - Common Lisp, Scheme, Clojure, Racket, etc.
6. **oop** - Java, C++, C#
7. **functional** - Haskell, OCaml, F#
8. **systems** - Ada, D, Zig
9. **mobile** - Swift, Kotlin, Java (Android)
10. **gpu** - CUDA, OpenCL, HLSL, GLSL
11. **constraints** - MiniZinc, Answer Set Programming
12. **scientific** - MATLAB, Julia, R, Fortran
13. **business** - COBOL, RPG
14. **bytecode** - JVM bytecode, BEAM bytecode
15. **fpga** - Verilog, VHDL, Chisel
16. **expert_systems** - CLIPS, Prolog
17. **grammar** - ANTLR, Lex/Yacc
18. **lexer** - Lexer generators
19. **parser** - Parser generators
20. **planning** - PDDL, domain-specific planners

Required Fields for New Emitter

Specification Input (YAML/JSON)

```yaml
category: <category_name>            # e.g., "json", "xml", "protobuf"
description: <human_description>     # e.g., "JSON serialization emitter"
output_types:                        # Target formats
  - <type1>                          # e.g., "json", "schema"
  - <type2>
features:                            # Optional features
  - <feature1>                       # e.g., "pretty_print", "validation"
  - <feature2>
ir_fields:                           # Required IR fields
  required:
    - <field1>                       # e.g., "functions", "types"
  optional:
    - <field2>
architectures:                       # Applicable architectures (if relevant)
  - <arch1>                          # e.g., "arm", "x86", "any"
dependencies:                        # External dependencies
  spark:
    - <package>                      # e.g., "Ada.Strings.Bounded"
  python:
    - <module>                       # e.g., "json", "hashlib"
  rust:
    - <crate>                        # e.g., "serde_json"
  haskell:
    - <package>                      # e.g., "aeson"
examples:
  input:                             # Example IR input
    <ir_structure>
  output:                            # Expected output
    <generated_code>
```

Template Variables

Standard variables available in all templates:

- `{{CATEGORY}}` - Category name (lowercase)
- `{{CATEGORY_UPPER}}` - Category name (uppercase)
- `{{CATEGORY_TITLE}}` - Category name (title case)
- `{{DESCRIPTION}}` - Human-readable description
- `{{TIMESTAMP}}` - ISO 8601 timestamp
- `{{GENERATOR}}` - Pipeline name (SPARK/Python/Rust/Haskell)
- `{{OUTPUT_TYPES}}` - List of output types
- `{{FEATURES}}` - List of features
- `{{AUTHOR}}` - STUNIR Team
- `{{LICENSE}}` - MIT License

Validation Requirements

Generated emitters must:
1. ✅ Compile/parse in target language
2. ✅ Follow language-specific style guidelines

3. ✅ Include all required error handling
4. ✅ Generate deterministic output (same input → same output)
5. ✅ Include comprehensive comments
6. ✅ Provide example usage
7. ✅ Include test scaffolding
8. ✅ Update build system files

Build System Integration

SPARK

Add to `targets/spark/stunir_emitters.gpr`:

```
for Source_Dirs use ("src", "<category>");
```

Python

Add to `setup.py` or `pyproject.toml`:

```
packages=['targets.<category>']
```

Rust

Add to `targets/rust/lib.rs`:

```
pub mod <category>;
```

Haskell

Add to `targets/haskell/stunir-emitters.cabal`:

```
exposed-modules: STUNIR.Emitters.<Category>
```

Testing Strategy

Unit Tests

- Type mapping correctness
- Configuration parsing
- Error handling

Integration Tests

- Full IR → code pipeline
- Multiple functions/types
- Edge cases

Compliance Tests

- Syntax validation (compile/parse)
- Style guide compliance
- Hash determinism

Cross-Pipeline Tests

- All 4 pipelines generate equivalent output
- Confluence testing

Note: This specification is used by `tools/emitter_generator/generate_emitter.py` to scaffold new emitters.