STUNIR API Reference

Complete reference documentation for all STUNIR public APIs.

Table of Contents

Python Tools

IR Emitter

Module: `tools.ir_emitter.emit_ir`

Converts STUNIR spec files to deterministic Intermediate Representation.

Functions

`canonical_json(data: Any) -> str`

Generate RFC 8785 / JCS subset canonical JSON.

```python
from tools.ir_emitter.emit_ir import canonical_json

data = {"z": 1, "a": 2, "m": 3}
result = canonical_json(data)
# Result: '{"a":2,"m":3,"z":1}'
```

Parameters:
- `data`: Any JSON-serializable Python object

Returns: Canonical JSON string with sorted keys and no extra whitespace

---

`compute_sha256(data: Union[bytes, str]) -> str`

Compute SHA-256 hash of provided data.

```
from tools.ir_emitter.emit_ir import compute_sha256

hash_value = compute_sha256("hello world")
# Result: 'b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9'
```

Parameters:
- `data` : Bytes or string to hash

Returns: Hex-encoded SHA-256 hash

`spec_to_ir(spec_data: Dict) -> Dict`

Transform a spec dictionary into an IR dictionary.

```
from tools.ir_emitter.emit_ir import spec_to_ir

spec = {
    "module": "example",
    "functions": [{"name": "main", "body": [...]}]
}
ir = spec_to_ir(spec)
```

Parameters:
- `spec_data` : Parsed spec JSON as dictionary

Returns: IR dictionary with normalized structure

Validators

Module: `tools.security.validation`

Input validation and sanitization utilities.

Classes

`InputValidator`

Comprehensive input validator for STUNIR operations.

```python
from tools.security.validation import InputValidator

validator = InputValidator(
    max_file_size=1024*1024,  # 1MB
    max_json_depth=50,
    base_dir="/project",
    allow_symlinks=False
)

# Validate a file
result = validator.validate_file("input.json")
if result.valid:
    print(f"Safe path: {result.value}")
else:
    print(f"Errors: {result.errors}")

# Validate JSON file
result = validator.validate_json_file("config.json")
if result.valid:
    config = result.value  # Parsed JSON
```

Constructor Parameters:
- `max_file_size` : Maximum allowed file size in bytes (default: 100MB)
- `max_json_depth` : Maximum JSON nesting depth (default: 50)
- `base_dir` : Base directory for path validation (optional)
- `allow_symlinks` : Whether to allow symlinks (default: False)

Methods:
- `validate_file(path)` → `ValidationResult`
- `validate_json_file(path)` → `ValidationResult`
- `validate_directory(path)` → `ValidationResult`

`ValidationResult`

Result of a validation operation.

```python
@dataclass
class ValidationResult:
    valid: bool
    value: Optional[Any] = None
    errors: List[str] = field(default_factory=list)
    warnings: List[str] = field(default_factory=list)
```

Attributes:
- `valid` : Whether validation passed
- `value` : Sanitized value (if valid)
- `errors` : List of validation errors
- `warnings` : List of non-critical warnings

Functions

`sanitize_path(path, base_dir=None, allow_absolute=False, allow_symlinks=False)`

Sanitize a file path to prevent security issues.

```python
from tools.security.validation import sanitize_path

# Basic usage
result = sanitize_path("config/settings.json")
if result.valid:
    safe_path = result.value

# With base directory constraint
result = sanitize_path("../etc/passwd", base_dir="/project")
# result.valid = False
# result.errors = ['Path escapes base directory']

# Allow absolute paths
result = sanitize_path("/tmp/data.json", allow_absolute=True)
```

Parameters:
- `path`: The path to sanitize
- `base_dir`: If provided, ensure path is within this directory
- `allow_absolute`: Whether to allow absolute paths (default: False)
- `allow_symlinks`: Whether to allow symlinks (default: False)

Returns: `ValidationResult`

`validate_identifier(name, max_length=256, allow_dots=False)`

Validate an identifier (module name, function name, etc.).

```python
from tools.security.validation import validate_identifier

# Valid identifiers
validate_identifier("my_module").valid  # True
validate_identifier("_private").valid   # True
validate_identifier("CamelCase").valid  # True

# Invalid identifiers
validate_identifier("123start").valid   # False (starts with number)
validate_identifier("with-dash").valid  # False (contains dash)
validate_identifier("").valid           # False (empty)

# Allow dots for qualified names
validate_identifier("module.submodule", allow_dots=True).valid  # True
```

Parameters:
- `name`: The identifier to validate
- `max_length`: Maximum allowed length (default: 256)
- `allow_dots`: Whether to allow dots (default: False)

Returns: `ValidationResult`

`validate_json_input(data, max_size=100MB, max_depth=50, schema=None)`

Validate JSON input for safety and conformance.

```python
from tools.security.validation import validate_json_input

# Validate JSON string
result = validate_json_input('{"key": "value"}')
if result.valid:
    parsed = result.value

# Validate with size limit
result = validate_json_input(large_json, max_size=1024*1024)

# Validate bytes
result = validate_json_input(b'{"key": "value"}')
```

Parameters:
- `data`: JSON string, bytes, or already-parsed dict
- `max_size`: Maximum size in bytes (default: 100MB)
- `max_depth`: Maximum nesting depth (default: 50)
- `schema`: Optional JSON schema for validation

Returns: `ValidationResult`

Manifest Generators

Module: `manifests`

Deterministic manifest generation and verification.

Base Classes

`BaseManifestGenerator`

Abstract base class for manifest generators.

```python
from manifests.base import BaseManifestGenerator

class CustomManifestGenerator(BaseManifestGenerator):
    def __init__(self, source_dir: str, output_path: str):
        super().__init__(
            manifest_type="custom",
            source_dir=source_dir,
            output_path=output_path
        )

    def _collect_entries(self) -> List[Dict]:
        # Override to collect entries
        entries = []
        for path in self.scan_directory(self.source_dir):
            entries.append({
                "name": path.name,
                "path": str(path),
                "hash": self.compute_file_hash(path),
            })
        return entries
```

Methods:
- `generate()` → Generate manifest and write to output
- `_collect_entries()` → Override to collect manifest entries
- `scan_directory(dir_path)` → Scan directory for files
- `compute_file_hash(path)` → Compute SHA-256 hash

`BaseManifestVerifier`

Abstract base class for manifest verifiers.

```python
from manifests.base import BaseManifestVerifier

class CustomManifestVerifier(BaseManifestVerifier):
    def _verify_entries(self, entries: List[Dict]) -> Tuple[bool, List[str]]:
        errors = []
        for entry in entries:
            if not os.path.exists(entry["path"]):
                errors.append(f"Missing: {entry['path']}")
        return len(errors) == 0, errors
```

Error System

Module: `tools.errors`

Comprehensive error handling with codes and suggestions.

Error Classes

`StunirError`

Base class for all STUNIR errors.

```python
from tools.errors import StunirError

try:
    risky_operation()
except StunirError as e:
    print(f"Error {e.code}: {e.message}")
    print(f"Category: {e.category}")
    print(f"Suggestion: {e.suggestion}")
    print(f"Recoverable: {e.is_recoverable}")
```

Attributes:
- `code`: Error code (e.g., “E3001”)
- `message`: Human-readable message
- `context`: Additional context dict
- `suggestion`: Actionable fix suggestion
- `cause`: Original exception

Methods:
- `to_dict()` → Convert to dictionary
- `to_json()` → Convert to JSON string

Specialized Error Classes

```
from tools.errors import (
    IOError,          # E1xxx - File system errors
    JSONError,        # E2xxx - JSON parsing errors
    ValidationError,  # E3xxx - Input validation errors
    VerificationError,# E4xxx - Verification failures
    SecurityError,    # E6xxx - Security violations
    UsageError,       # E5xxx - CLI usage errors
    ConfigError,      # E7xxx - Configuration errors
)

# Creating errors with context
raise ValidationError(
    "E3001",
    "Invalid module name",
    field="spec.modules[0].name",
    value="",
    expected="non-empty identifier"
)

raise IOError("E1001", "Config file not found", path="/missing/file")

raise VerificationError(
    "E4001",
    "Hash mismatch",
    expected_hash="abc123...",
    actual_hash="def456...",
    path="output.json"
)
```

Error Handler

```
from tools.errors import ErrorHandler

handler = ErrorHandler(verbose=True, json_output=False)

try:
    process_files()
except StunirError as e:
    handler.handle(e, exit_code=1)

# Batch error collection
for file in files:
    try:
        process(file)
    except StunirError as e:
        handler.collect(e)

if handler.has_errors():
    handler.report()
```

Rust Native Tools

Crypto Module

Module: `stunir_native::crypto`

Cryptographic hashing for files and directories.

Functions

`hash_file(path: &Path) -> Result<String>`

Compute SHA-256 hash of a file.

```rust
use stunir_native::crypto::hash_file;
use std::path::Path;

let hash = hash_file(Path::new("input.json"))?;
println!("SHA-256: {}", hash);
```

Security:
- Validates file size (max 1GB)
- Rejects symlinks
- Uses streaming for large files

`hash_directory(path: &Path, depth: usize) -> Result<String>`

Compute Merkle tree hash of a directory.

```rust
use stunir_native::crypto::hash_directory;
use std::path::Path;

let hash = hash_directory(Path::new("./src"), 0)?;
println!("Directory hash: {}", hash);
```

Security:
- Maximum depth limit (100 levels)
- Deterministic ordering (sorted paths)
- No symlink following

`hash_path(path: &Path) -> Result<String>`

Hash a path, auto-detecting file or directory.

```rust
use stunir_native::crypto::hash_path;

let hash = hash_path(Path::new("target"))?;
```

Canonical Module

Module: `stunir_native::canonical`

JSON canonicalization (RFC 8785 subset).

Functions

`normalize(json_str: &str) -> Result<String>`

Normalize JSON to canonical form.

```rust
use stunir_native::canonical::normalize;

let input = r#"{"z": 1, "a": 2}"#;
let canonical = normalize(input)?;
assert_eq!(canonical, r#"{"a":2,"z":1}"#);
```

`normalize_and_hash(json_str: &str) -> Result<String>`

Normalize JSON and compute SHA-256 hash.

```rust
use stunir_native::canonical::normalize_and_hash;

let hash = normalize_and_hash(r#"{"key": "value"}"#)?;
```

`json_equal(json_a: &str, json_b: &str) -> Result<bool>`

Check if two JSON strings are semantically equivalent.

```rust
use stunir_native::canonical::json_equal;

assert!(json_equal(
    r#"{"x": 1, "y": 2}"#,
    r#"{"y":2,"x":1}"#
)?);
```

IR Module

Module: `stunir_native::ir_v1`

Intermediate Representation data structures.

Types

`Spec`

Input specification format.

```rust
use stunir_native::ir_v1::Spec;

let spec: Spec = serde_json::from_str(json)?;
assert_eq!(spec.kind, "spec");
```

Fields:
- `kind: String` - Always “spec”

- `modules: Vec<SpecModule>` - Source modules
- `metadata: HashMap<String, String>` - Optional metadata

`IrV1`

Intermediate Representation version 1.

```
use stunir_native::ir_v1::IrV1;

let ir = IrV1::new("my_module");
ir.add_function(func);
ir.validate()?;
```

Fields:
- `kind: String` - Always "ir"
- `generator: String` - Generator identifier
- `ir_version: String` - "v1"
- `module_name: String` - Primary module name
- `functions: Vec<IrFunction>` - Function definitions
- `modules: Vec<IrModule>` - External dependencies
- `metadata: IrMetadata` - Preserved metadata

Receipt Module

Module: `stunir_native::receipt`

Build receipt management.

Types

`Receipt`

Build receipt linking inputs to outputs.

```
use stunir_native::receipt::Receipt;

let receipt = Receipt::new("build-001")
    .add_tool("stunir-native", "0.1.0")
    .add_tool("rustc", "1.75.0")
    .with_timestamp("2026-01-28T12:00:00Z");

receipt.add_input("input.json", "abc123...");
receipt.add_output("output.bin", "def456...");

// Verify against actual files
if receipt.verify()? {
    println!("Build verified!");
}
```

Haskell Native Tools

Manifest Module

Module: `Stunir.Manifest`

Deterministic IR bundle manifest generation.

Functions

`generateIrManifest :: FilePath -> IO IrManifest`

Generate manifest from IR directory.

```
import Stunir.Manifest

manifest <- generateIrManifest "asm/ir"
writeIrManifest manifest "receipts/ir_manifest.json"
```

`computeFileHash :: FilePath -> IO Text`

Compute SHA-256 hash of a file.

```
hash <- computeFileHash "asm/ir/module.dcbor"
```

Provenance Module

Module: `Stunir.Provenance`

Build provenance tracking and C header generation.

Functions

`generateCHeader :: Provenance -> Text`

Generate minimal C header.

```
import Stunir.Provenance

let prov = Provenance { prov_epoch = 1234, ... }
let header = generateCHeader prov
writeFile "provenance.h" header
```

`generateCHeaderExtended :: Provenance -> Text`

Generate comprehensive C header with all fields.

CLI Commands

stunir-native

```
# Hash a file or directory
stunir-native hash <path>

# Canonicalize JSON
stunir-native canon <input.json> [output.json]

# Verify receipts
stunir-native verify <manifest.json>

# Generate IR manifest
stunir-native gen-ir-manifest --ir-dir asm/ir --out receipts/ir_manifest.json

# Generate provenance
stunir-native gen-provenance --extended
```

Python Tools

```
# Emit IR from spec
python -m tools.ir_emitter.emit_ir spec.json output.json

# Generate manifest
python -m manifests.ir.gen_ir_manifest --ir-dir asm/ir --output manifest.json

# Verify manifest
python -m manifests.ir.verify_ir_manifest manifest.json --ir-dir asm/ir

# Validate input
python -m tools.security.validation
```

Error Codes Quick Reference

Code	Category	Description
E1xxx	IO	File system errors
E2xxx	JSON	JSON parsing errors
E3xxx	Validation	Input validation errors
E4xxx	Verification	Hash/signature mismatches
E5xxx	Usage	CLI usage errors
E6xxx	Security	Security violations
E7xxx	Config	Configuration errors

See Also

- User Guide (USER_GUIDE.md) - Getting started tutorial
- Architecture (ARCHITECTURE.md) - System design documentation
- Contributing (../CONTRIBUTING.md) - Development guidelines
- Deployment (DEPLOYMENT.md) - Production deployment guide

Generated for STUNIR v0.1.0