Migration Guide: Hash Manifests → Semantic IR

Target Audience: STUNIR users, integrators, and developers
STUNIR Version: 2.0 (Semantic IR release)
Last Updated: 2026-01-30

Executive Summary

STUNIR 2.0 introduces **Semantic IR**—a fundamental architectural improvement that replaces hash-based manifests with true semantic intermediate representation. This guide helps you migrate existing STUNIR projects to the new system.

Key Benefits of Semantic IR:
- ✅ True semantic equivalence (not just byte-level equality)
- ✅ Better error messages with type information
- ✅ Optimization passes for smaller/faster code
- ✅ Richer debugging information
- ✅ Foundation for advanced features (pattern matching, effects, dependent types)

Migration Effort:
- **Simple projects:** 1-2 hours
- **Complex projects:** 1-2 days
- **Custom tooling:** 1-2 weeks

1. Understanding the Changes

1.1 What's Different?

Aspect	Old (Hash Manifests)	New (Semantic IR)
IR Format	SHA-256 hashes of files	AST-based JSON structure
Equivalence	Byte-level (`sha256(A) == sha256(B)`)	Semantic (`normalize(A) == normalize(B)`)
Type System	Implicit, inferred at emission	Explicit, checked during parsing
Transformations	None	Optimizations, normalization
Error Detection	Late (during code generation)	Early (during parsing)
File Extension	`.txt` manifest	`.sir.json` (Semantic IR JSON)

1.2 Breaking Changes

⚠️ **The following will require code changes:**

1. **IR File Format:** Old `asm/spec_ir.txt` → New `asm/spec_ir.sir.json`
2. **Tool Interface:** Parser output is now JSON AST, not hash manifest
3. **Receipts:** Receipt format includes IR node hashes, not file hashes
4. **Custom Emitters:** Must implement visitor pattern for new IR nodes
5. **Verification:** Hash-based verification replaced by semantic validation

1.3 Backward Compatibility

STUNIR 2.0 provides:
- Legacy mode flag: `--use-legacy-ir` (supported for 2 release cycles)
- Migration tool: `stunir migrate-to-semantic-ir`
- Compatibility layer for reading old receipts

Deprecation Timeline:
- **STUNIR 2.0 (Current):** Semantic IR default, legacy available
- **STUNIR 2.1 (Q3 2026):** Legacy deprecated, warnings issued
- **STUNIR 3.0 (Q1 2027):** Legacy removed, Semantic IR only

2. Migration Paths

2.1 Path A: Automatic Migration (Recommended)

For standard STUNIR projects using default tools

```
# Step 1: Update STUNIR to 2.0
git pull origin main
./scripts/build.sh

# Step 2: Run migration tool
./tools/migrate_to_semantic_ir.sh --repo-root /path/to/your/project

# Step 3: Verify migration
./scripts/verify.sh --semantic-ir

# Step 4: Test your build
./scripts/build.sh
```

What the migration tool does:
1. Parses your old specs into Semantic IR
2. Regenerates receipts with new format
3. Updates build scripts to use Semantic IR
4. Backs up old files to `.stunir_migration_backup/`
5. Generates migration report

Expected output:

```
[MIGRATE] Reading spec files from spec/...
[MIGRATE] ✓ Parsed 12 spec files successfully
[MIGRATE] ✓ Generated Semantic IR: asm/spec_ir.sir.json
[MIGRATE] ✓ Updated receipts in receipts/
[MIGRATE] ✓ Backup created: .stunir_migration_backup/
[MIGRATE]
[MIGRATE] Migration complete! Next steps:
[MIGRATE]   1. Review changes: git diff
[MIGRATE]   2. Test build: ./scripts/build.sh
[MIGRATE]   3. Commit: git add . && git commit -m "Migrate to Semantic IR"
```

2.2 Path B: Manual Migration

For projects with custom build scripts or integrations

Step 1: Update Spec Files (If Needed)

Most specs work as-is, but you may need to add explicit type annotations:

Old Spec (implicit types):

```
{
  "functions": [
    {
      "name": "add",
      "parameters": [
        {"name": "a"},
        {"name": "b"}
      ],
      "body": "return a + b;"
    }
  ]
}
```

New Spec (explicit types preferred):

```
{
  "functions": [
    {
      "name": "add",
      "parameters": [
        {"name": "a", "type": "i32"},
        {"name": "b", "type": "i32"}
      ],
      "return_type": "i32",
      "body": "return a + b;"
    }
  ]
}
```

Note: Type inference still works, but explicit types give better error messages.

Step 2: Update Build Scripts

Replace old parser invocations:

Old:

```
# Old hash-based IR generation
tools/spark/bin/stunir_spec_to_ir_main \
  --spec-dir spec/ \
  --output asm/spec_ir.txt
```

New:

```
# New semantic IR generation
tools/spark/bin/stunir_spec_to_semantic_ir \
  --spec-dir spec/ \
  --output asm/spec_ir.sir.json \
  --format json
```

Step 3: Update Emitter Invocations

Emitters now consume Semantic IR:

Old:

```
tools/spark/bin/stunir_ir_to_code_main \
  --ir-file asm/spec_ir.txt \
  --target c99 \
  --output build/generated/
```

New:

```
tools/spark/bin/stunir_semantic_ir_to_code \
  --ir-file asm/spec_ir.sir.json \
  --target c99 \
  --output build/generated/
```

Step 4: Update Custom Tools

If you have custom tools that read IR:

Old (reading hash manifest):

```
import hashlib

with open('asm/spec_ir.txt') as f:
    for line in f:
        sha, path = line.strip().split('\t')
        # Process file hash
```

New (reading Semantic IR):

```
import json

with open('asm/spec_ir.sir.json') as f:
    ir = json.load(f)

for func in ir['declarations']:
    if func['kind'] == 'function_decl':
        print(f"Function: {func['name']}")
        print(f"  Return type: {func['type']['return_type']}")
        # Process semantic structure
```

2.3 Path C: Gradual Migration

For large projects migrating incrementally

STUNIR 2.0 supports hybrid mode:

```
# Use legacy IR for most targets, Semantic IR for select targets
./scripts/build.sh \
  --legacy-ir \
  --semantic-ir-targets=c99,rust,wasm
```

This allows you to:
1. Migrate critical targets first
2. Validate output quality
3. Gradually migrate remaining targets
4. Minimize risk

Recommended order:
1. Start with C99 (most mature)
2. Add Rust, Python (high usage)
3. Add embedded targets (ARM, AVR)
4. Add GPU targets (CUDA, OpenCL)
5. Complete remaining targets

3. Common Migration Scenarios

3.1 Scenario: Standard Embedded Project

Project: MAVLink message handler for ArduPilot

Migration steps:

```
cd /path/to/ardupilot_stunir_project

# 1. Backup current state
git checkout -b pre-semantic-ir-backup
git checkout main

# 2. Run migration tool
./tools/migrate_to_semantic_ir.sh --repo-root .

# 3. Review generated IR
cat asm/spec_ir.sir.json | jq '.declarations[] | {name, kind}'

# 4. Test build
./scripts/build.sh

# 5. Compare output with old version
diff -r build/generated/ .stunir_migration_backup/build/generated/

# 6. Validate on hardware (if applicable)
make upload

# 7. Commit migration
git add .
git commit -m "Migrate to STUNIR 2.0 Semantic IR"
```

3.2 Scenario: Multi-Target Project

Project: Algorithm library generating C, Rust, Python, WASM

Migration strategy:

1. Use hybrid mode initially
2. Migrate one target at a time
3. Validate each target before proceeding

```
# Phase 1: Migrate C99 only
./scripts/build.sh \
  --semantic-ir-targets=c99 \
  --legacy-ir-targets=rust,python,wasm

# Validate C99 output
./tests/validate_c99.sh

# Phase 2: Add Rust
./scripts/build.sh \
  --semantic-ir-targets=c99,rust \
  --legacy-ir-targets=python,wasm

# Validate Rust output
./tests/validate_rust.sh

# Phase 3: Add Python
./scripts/build.sh \
  --semantic-ir-targets=c99,rust,python \
  --legacy-ir-targets=wasm

# Phase 4: Complete migration (all targets)
./scripts/build.sh --semantic-ir
```

3.3 Scenario: Custom Emitter

Project: Custom emitter for proprietary embedded platform

Migration checklist:

1. **Update emitter interface:**
   ```ada
   – Old interface
   procedure Emit_Code(
   IR_File : String;
   Output : String
   );

– New interface
with STUNIR.Semantic_IR;
procedure Emit_Code(
IR : STUNIR.Semantic_IR.IR_Module;
Output : String
);
```

1. **Implement visitor pattern:**
   ```ada
   package My_Custom_Emitter is
   type Custom_Emitter is new STUNIR.Code_Generator with private;

   overriding procedure Visit_Function(
   Emitter : in out Custom_Emitter;
   Func : Function_Decl
   );

   – Implement all required visit methods
   end My_Custom_Emitter;
   ```

2. **Update code generation logic:**
   - Replace hash lookups with IR traversal
   - Use type information from IR nodes
   - Leverage semantic information for optimizations

3. **Add tests:**
   ```bash
   # Test with Semantic IR
   ./test_custom_emitter.sh --ir-file test_inputs/simple.sir.json
   ```

4. Troubleshooting

4.1 Common Issues

Issue 1: Type Mismatch Errors

Symptom:

```
[ERROR] Type mismatch in function 'calculate':
  Expected: i32
  Got: f64
  Location: main.spec:42:10
```

Cause: Semantic IR enforces type safety that was implicit before

Solution: Add explicit type annotations or fix type errors:

```
{
  "name": "calculate",
  "parameters": [
    {"name": "x", "type": "f64"}  // ← Explicit type
  ],
  "return_type": "f64",  // ← Explicit return type
  "body": "return x * 2.0;"
}
```

Issue 2: Undefined Variable Errors

Symptom:

```
[ERROR] Undefined variable: 'result'
  Location: main.spec:55:5
```

Cause: Name resolution pass detected undefined reference

Solution: Declare variable before use:

```
{
  "kind": "block",
  "statements": [
    {"kind": "var_decl", "name": "result", "type": "i32", "value": 0},
    {"kind": "assign", "target": "result", "value": "..."}
  ]
}
```

Issue 3: Semantic IR Validation Fails

Symptom:

```
[ERROR] IR validation failed:
  Invalid node reference: n_12345
  Location: function 'main'
```

Cause: Corrupted or manually-edited IR file

Solution: Regenerate IR from spec:

```
# Delete corrupted IR
rm asm/spec_ir.sir.json

# Regenerate from spec
./scripts/build.sh --regenerate-ir
```

Issue 4: Performance Regression

Symptom: Build times increased significantly (>2x)

Cause: Semantic IR adds parsing and analysis overhead

Solution 1 - Enable caching:

```
export STUNIR_ENABLE_IR_CACHE=1
./scripts/build.sh
```

Solution 2 - Use incremental compilation:

```
./scripts/build.sh --incremental
```

Solution 3 - Profile and optimize:

```
./scripts/build.sh --profile
# Review profile.json for bottlenecks
```

Issue 5: Legacy Mode Not Working

Symptom:

```
[ERROR] Legacy mode flag not recognized
```

Cause: Using wrong binary version

Solution: Ensure you're using STUNIR 2.0+:

```
./tools/spark/bin/stunir_spec_to_ir_main --version
# Should output: STUNIR 2.0 or higher
```

4.2 Getting Help

Resources:
- Documentation: `docs/SEMANTIC_IR_SPECIFICATION.md`
- Examples: `examples/semantic_ir/`
- Issue Tracker: https://github.com/emstar-en/STUNIR/issues
- Community Forum: https://forum.stunir.dev/

Reporting Migration Issues:
1. Check existing issues first
2. Provide minimal reproduction case
3. Include STUNIR version: `./tools/spark/bin/stunir_spec_to_ir_main --version`
4. Include error logs and IR files (if applicable)

5. Validation and Testing

5.1 Validation Checklist

After migration, verify:

- [ ] Build completes successfully
- [ ] All target outputs generated
- [ ] Output matches expected behavior
- [ ] Tests pass
- [ ] Receipts validate correctly
- [ ] Performance acceptable

5.2 Testing Strategy

Test 1: Functional Equivalence

```
# Generate code with old and new system
./scripts/build.sh --legacy-ir --output build/legacy/
./scripts/build.sh --semantic-ir --output build/semantic/

# Compare outputs (should be semantically equivalent)
./tests/compare_outputs.sh build/legacy/ build/semantic/
```

Test 2: Runtime Behavior

```
# Compile and run both versions
cd build/legacy/c99 && make && ./test_runner
cd build/semantic/c99 && make && ./test_runner

# Compare results
diff legacy_results.txt semantic_results.txt
```

Test 3: Performance

```
# Benchmark both versions
./benchmark.sh --legacy-ir > legacy_bench.txt
./benchmark.sh --semantic-ir > semantic_bench.txt

# Compare performance
./compare_benchmarks.py legacy_bench.txt semantic_bench.txt
```

6. Advanced Topics

6.1 Custom Semantic Passes

STUNIR 2.0 allows custom analysis and transformation passes:

```
-- my_custom_pass.ads
with STUNIR.Semantic_IR;
package My_Custom_Pass is
   procedure Run_Pass(Module : in out STUNIR.Semantic_IR.IR_Module);
   -- Custom analysis or transformation
end My_Custom_Pass;
```

Register custom pass:

```
./scripts/build.sh \
  --custom-pass my_custom_pass.so \
  --pass-order "name_resolution,type_check,my_custom_pass,optimize"
```

6.2 IR Introspection

Query Semantic IR programmatically:

```
import json
from stunir import SemanticIR

# Load IR
with open('asm/spec_ir.sir.json') as f:
    ir = SemanticIR.load(f)

# Query functions
for func in ir.functions():
    print(f"Function: {func.name}")
    print(f"  Parameters: {[p.name for p in func.parameters]}")
    print(f"  Calls: {func.get_all_function_calls()}")

# Type analysis
type_info = ir.get_type_info()
print(f"Total types: {len(type_info.all_types())}")
print(f"Custom types: {len(type_info.custom_types())}")
```

6.3 Optimizations

Enable optimization passes:

```
# Default (basic optimizations)
./scripts/build.sh --semantic-ir

# Aggressive optimizations
./scripts/build.sh --semantic-ir --optimize=aggressive

# Specific passes
./scripts/build.sh --semantic-ir \
  --enable-pass constant_folding \
  --enable-pass dead_code_elimination \
  --enable-pass function_inlining
```

7. FAQ

Q: Can I use both legacy and Semantic IR in the same project?
A: Yes, in hybrid mode (STUNIR 2.0 only). Different targets can use different IR formats.

Q: Will my old receipts still validate?
A: Yes, STUNIR 2.0 includes a compatibility layer for old receipt format.

Q: Is performance worse with Semantic IR?
A: Initial parsing is slower (~2x), but optimizations can produce faster output code. Net effect depends on use case.

Q: Can I manually edit Semantic IR files?
A: Technically yes (it's JSON), but not recommended. Use spec files as the source of truth.

Q: What happens to my custom emitters?
A: They need to be updated to consume Semantic IR. See Section 3.3 for migration guide.

Q: Can I keep using hash manifests forever?
A: No, legacy mode will be removed in STUNIR 3.0 (Q1 2027).

Q: Does Semantic IR work with all 28 target categories?
A: Yes, all emitters have been updated in STUNIR 2.0.

Q: What if I find a bug in Semantic IR?
A: Report it on GitHub. In the meantime, use legacy mode: `--use-legacy-ir`

8. Migration Checklist

Use this checklist to track your migration:

Pre-Migration

- [ ] Backup current project: `git checkout -b pre-semantic-ir-backup`
- [ ] Update STUNIR to 2.0: `git pull origin main && ./scripts/build.sh`
- [ ] Read migration guide (this document)
- [ ] Review Semantic IR specification: `docs/SEMANTIC_IR_SPECIFICATION.md`

Migration

- [ ] Run migration tool: `./tools/migrate_to_semantic_ir.sh`
- [ ] Review generated IR: `cat asm/spec_ir.sir.json | jq`
- [ ] Update build scripts (if custom)
- [ ] Update custom emitters (if any)
- [ ] Update test scripts

Validation

- [ ] Build completes: `./scripts/build.sh`
- [ ] Verification passes: `./scripts/verify.sh --semantic-ir`
- [ ] Tests pass: `./tests/run_all_tests.sh`
- [ ] Output comparison (old vs new)
- [ ] Runtime testing (if applicable)
- [ ] Performance benchmarking

Post-Migration

- [ ] Commit changes: `git add . && git commit -m "Migrate to Semantic IR"`
- [ ] Update documentation

- [ ] Notify team/users
- [ ] Monitor for issues
- [ ] Delete backup branch (once stable): `git branch -D pre-semantic-ir-backup`

9. Conclusion

Migrating to Semantic IR is a significant upgrade that brings STUNIR's architecture in line with its goal of true semantic code generation. While the migration requires some effort, the benefits—better error detection, optimization opportunities, and foundation for future features—make it worthwhile.

Key Takeaways:
1. Most projects can use automatic migration tool
2. Backward compatibility maintained for 2 release cycles
3. Hybrid mode allows gradual migration
4. Performance may vary (better output, slower parsing)
5. Type safety catches errors earlier

Next Steps:
1. Complete the migration checklist (Section 8)
2. Run your first Semantic IR build
3. Validate output and performance
4. Provide feedback to STUNIR team

Document Status: ✅ Migration Guide Complete
For Help: See Section 4.2 or file an issue on GitHub
Related Documents:
- `docs/SEMANTIC_IR_SPECIFICATION.md` - Design details
- `docs/SEMANTIC_IR_IMPLEMENTATION_PLAN.md` - Implementation roadmap
- `docs/SEMANTIC_IR_USER_GUIDE.md` - User documentation (coming soon)