Frequently Asked Questions (FAQ)

A comprehensive collection of frequently asked questions about STUNIR.

Table of Contents

- General Questions
- Installation Questions
- Usage Questions
- Troubleshooting Questions
- Performance Questions
- Security Questions
- Integration Questions

General Questions

G1: What is STUNIR?

STUNIR (Spec To Universal Intermediate Representation) is a toolkit for generating deterministic, verifiable code from specifications. It converts high-level module definitions into multiple target languages while maintaining cryptographic verification of all outputs.

G2: What problem does STUNIR solve?

STUNIR addresses several challenges:
- **Reproducibility**: Ensures the same input always produces the same output
- **Multi-language support**: Generate code for Python, Rust, C, and more from a single spec
- **Verification**: Cryptographic receipts prove artifact integrity
- **Auditability**: Provenance tracking for compliance requirements

G3: Who should use STUNIR?

STUNIR is ideal for:
- Developers needing deterministic builds
- Teams maintaining code in multiple languages
- Organizations with compliance/audit requirements
- Projects requiring reproducible code generation

G4: Is STUNIR production-ready?

Yes! STUNIR 2.0 is production-ready with:
- Comprehensive test coverage
- Full documentation
- Stable API
- Active maintenance

G5: What languages are supported?

Built-in targets:
- Python 3.x
- Rust
- C89 (ANSI C)
- C99
- x86/x86_64 Assembly
- ARM/ARM64 Assembly

Via custom emitters:
- TypeScript
- Go
- Swift
- Kotlin
- Any language you implement

G6: How is STUNIR different from other code generators?

Key differentiators:
1. **Determinism**: Byte-for-byte reproducible output
2. **Verification**: Built-in receipt and manifest system
3. **Multi-target**: Single IR to multiple languages
4. **Provenance**: Complete build history tracking

G7: Is STUNIR open source?

Yes, STUNIR is open source. See the LICENSE file for details.

G8: Where can I find the source code?

The source code is available on GitHub at the STUNIR repository.

G9: How can I contribute?

See the CONTRIBUTING.md (../CONTRIBUTING.md) guide for:
- Code contributions
- Documentation improvements
- Bug reports
- Feature requests

G10: What's the release schedule?

STUNIR follows semantic versioning:
- **Patch releases** (2.0.x): Bug fixes, as needed
- **Minor releases** (2.x.0): New features, quarterly
- **Major releases** (x.0.0): Breaking changes, annually

Installation Questions

I1: What are the system requirements?

Minimum:
- Python 3.8+

- 100MB disk space
- Any modern OS (Linux, macOS, Windows)

Recommended:
- Python 3.10+
- Git (for version control)
- GHC 9.0+ (for native tools)

I2: How do I install STUNIR?

```
# Clone the repository
git clone https://github.com/stunir/stunir.git
cd stunir

# (Optional) Install dependencies
pip install -r requirements.txt
```

I3: Can I install via pip?

Not currently. Pip installation is planned for v3.0.

I4: Do I need to install all dependencies?

No. The core functionality uses only Python standard library. Dependencies are optional:
- `pyyaml`: For YAML config files
- `rich`: For colorful CLI output

I5: How do I verify my installation?

```
# Check Python version
python3 --version  # Should be 3.8+

# Test core functionality
python3 -c "import json, hashlib; print('OK')"

# Run a simple test
python3 tools/ir_emitter/emit_ir.py --help
```

Usage Questions

U1: How do I create a spec?

Create a JSON file with this structure:

```
{
  "name": "my_module",
  "version": "1.0.0",
  "functions": [
    {
      "name": "my_function",
      "params": [{"name": "x", "type": "i32"}],
      "returns": "i32"
    }
  ],
  "exports": ["my_function"]
}
```

U2: What types are supported?

Type	Description
`i32`	32-bit signed integer
`i64`	64-bit signed integer
`f32`	32-bit float
`f64`	64-bit float
`bool`	Boolean
`str`	String
`void`	No value

U3: How do I generate IR from a spec?

```
python3 tools/ir_emitter/emit_ir.py spec.json output/ir/module.ir.json
```

U4: How do I generate code for a specific language?

```
python3 tools/emitters/emit_code.py \
  --target python \
  --ir module.ir.json \
  --output output/python/
```

U5: How do I generate code for multiple languages?

```
for target in python rust c99; do
  python3 tools/emitters/emit_code.py \
    --target $target \
    --ir module.ir.json \
    --output output/$target/
done
```

U6: How do I create a manifest?

```
# IR manifest
python3 manifests/ir/gen_ir_manifest.py --ir-dir asm/ir/

# Targets manifest
python3 manifests/targets/gen_targets_manifest.py --targets-dir output/
```

U7: How do I verify artifacts?

```
# Verify specific manifest
python3 manifests/ir/verify_ir_manifest.py receipts/ir_manifest.json

# Strict verification (all manifests)
./scripts/verify_strict.sh --strict
```

U8: Can I use STUNIR in a CI/CD pipeline?

Yes! Example GitHub Actions workflow:

```
- name: Generate and verify
  run: |
    python3 tools/ir_emitter/emit_ir.py spec.json ir.json
    python3 manifests/ir/gen_ir_manifest.py
    ./scripts/verify_strict.sh --strict
```

U9: How do I create a custom emitter?

```
class MyEmitter:
    TYPE_MAP = {'i32': 'Int', 'f64': 'Double'}

    def emit(self, ir):
        # Generate code from IR
        return generated_code
```

See examples/python/custom_emitters.py (../examples/python/custom_emitters.py) for details.

U10: How do I process multiple specs at once?

```
# Using shell
for spec in specs/*.json; do
  python3 tools/ir_emitter/emit_ir.py "$spec" "output/$(basename $spec)"
done

# Using Python
# See examples/python/batch_processing.py
```

Troubleshooting Questions

T1: Why do I get “Invalid JSON” errors?

Common causes:
- Trailing commas: `{"a": 1,}` is invalid
- Single quotes: Use double quotes for strings
- Missing commas between elements

Fix: Validate your JSON:

```
python3 -m json.tool < spec.json
```

T2: Why do I get “Missing required field” errors?

Your spec is missing a required field. Required fields:
- `name`
- `version`
- `functions`
- `exports`

T3: Why are my hashes different each time?

Possible causes:
1. **Timestamps**: Ensure you’re not including variable timestamps
2. **Unordered iteration**: Use sorted keys
3. **Floating point**: Use consistent representation

T4: Why does verification fail?

Check:
1. Files weren’t modified after manifest generation
2. Encoding is UTF-8 (not UTF-16)
3. Line endings are consistent (LF, not CRLF)
4. Manifest is up to date

Fix:

```
# Regenerate manifest
python3 manifests/ir/gen_ir_manifest.py --ir-dir asm/ir/
```

T5: Why is a file missing from the manifest?

The file wasn’t in the directory when the manifest was generated.

Fix: Regenerate the manifest after adding files.

T6: Why do I get “Hash mismatch” errors?

The file content differs from what’s recorded in the manifest.

Causes:
- File was modified
- Different encoding
- Line ending changes

T7: Why does the Haskell build fail?

```
# Update dependencies
cabal update

# Clean and rebuild
cd tools/native/haskell/stunir-native
cabal clean
cabal build
```

T8: Why is the output different on different machines?

Check:
- Python version matches
- Locale/encoding settings
- Line ending configuration (git `core.autocrlf`)

T9: How do I debug determinism issues?

```
# Compare outputs
for i in 1 2 3; do
  python3 emit.py spec.json | sha256sum
done

# If hashes differ, use diff
diff <(python3 emit.py spec.json) <(python3 emit.py spec.json)
```

T10: Where are log files stored?

STUNIR doesn't create log files by default. Enable logging:

```
export STUNIR_LOG_LEVEL=DEBUG
python3 tools/ir_emitter/emit_ir.py spec.json ir.json 2>&1 | tee stunir.log
```

Performance Questions

P1: How long does IR generation take?

Typical times:
- Small spec (1-10 functions): < 100ms
- Medium spec (10-100 functions): 100-500ms
- Large spec (100+ functions): 500ms-2s

P2: Can I parallelize code generation?

Yes! Use parallel processing:

```
# Using GNU parallel
find specs/ -name "*.json" | parallel python3 process.py {}

# Using Python ThreadPoolExecutor
# See examples/python/batch_processing.py
```

P3: How can I speed up builds?

1. **Parallel processing**: Use multiple workers
2. **Incremental builds**: Only rebuild changed files
3. **SSD storage**: Faster I/O helps
4. **Native tools**: Use Haskell tools for heavy lifting

P4: What's the memory usage?

Typical usage:
- Core operations: ~50MB
- Large specs: ~100-200MB
- Batch processing: Depends on parallelism

P5: Is there a caching system?

Currently, no built-in caching. Planned for v3.0.

Workaround: Implement your own:

```
import hashlib
import os

def cached_emit(spec_path, output_path, cache_dir=".cache"):
    spec_hash = compute_file_hash(spec_path)
    cache_file = f"{cache_dir}/{spec_hash}.json"

    if os.path.exists(cache_file):
        return read_file(cache_file)

    result = emit(spec_path)
    write_file(cache_file, result)
    return result
```

Security Questions

S1: Is the generated code safe?

STUNIR generates deterministic, reviewable code. However:
- Review all generated code before deployment
- The security of output depends on your spec
- STUNIR doesn't add security vulnerabilities

S2: Are the hashes cryptographically secure?

Yes. STUNIR uses SHA-256, which is:
- Collision-resistant
- One-way (irreversible)
- Widely trusted

S3: Can manifests be tampered with?

Manifests include self-referential hashes. Tampering is detectable:
- Changing any entry invalidates the manifest hash
- Verification fails if tampered

S4: How do I verify artifact integrity?

```
# Verify using manifest
python3 manifests/ir/verify_ir_manifest.py manifest.json

# Manual verification
sha256sum file.json  # Compare with manifest
```

S5: Is there signing support?

Not currently. GPG/PGP signing is planned for v3.0.

Workaround:

```
gpg --sign --armor manifest.json
```

Integration Questions

N1: Can I use STUNIR with existing build systems?

Yes! STUNIR integrates with:
- **Make**: Call Python scripts from Makefile
- **CMake**: Add custom commands
- **Bazel**: Define custom rules
- **npm/yarn**: Use as pre-build step

N2: How do I integrate with GitHub Actions?

```yaml
name: Build
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.10'
      - name: Build
        run: |
          python3 tools/ir_emitter/emit_ir.py spec.json ir.json
          ./scripts/verify_strict.sh
```

N3: How do I integrate with GitLab CI?

```yaml
build:
  image: python:3.10
  script:
    - python3 tools/ir_emitter/emit_ir.py spec.json ir.json
    - ./scripts/verify_strict.sh
```

N4: Can I use STUNIR with Docker?

Yes! Example Dockerfile:

```
FROM python:3.10-slim
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
ENTRYPOINT ["python3", "tools/ir_emitter/emit_ir.py"]
```

N5: How do I use STUNIR programmatically?

```
import json
import sys
sys.path.insert(0, 'tools/ir_emitter')
from emit_ir import spec_to_ir, canonical_json

# Load spec
with open('spec.json') as f:
    spec = json.load(f)

# Generate IR
ir = spec_to_ir(spec)

# Write output
print(canonical_json(ir))
```

N6: Is there an API?

STUNIR provides a CLI interface. A Python API is planned for v3.0.

N7: Can I embed STUNIR in my application?

Yes, import the modules directly:

```
from tools.ir_emitter.emit_ir import spec_to_ir
from manifests.base import canonical_json, compute_sha256
```

Additional Resources

- User Guide (USER_GUIDE.md)
- API Reference (API.md)
- Glossary (GLOSSARY.md)
- Tutorials (tutorials/)
- Examples (../examples/)
- Migration Guides (migration/)

Still Have Questions?

If your question isn’t answered here:

1. **Search GitHub Issues**: Your question may already be answered
2. **Open a Discussion**: For general questions
3. **Open an Issue**: For bugs or feature requests
4. **Check the Documentation**: Comprehensive guides available

---

Feedback

Was this FAQ helpful?

- Found an answer: :thumbsup:
- Missing information: Open an issue (https://github.com/stunir/stunir/issues/new)
- Suggestion: Start a discussion (https://github.com/stunir/stunir/discussions/new)

Last updated: 2026-01-28