Design and Implementation of TBCX: A Bytecode Serialization Framework for Tcl 9.1

Miguel Bañón

Abstract

This paper presents `tbcx`, a library that serializes and deserializes Tcl 9.1 bytecode into a compact, portable binary format (TBCX). The library allows scripts to be compiled once and then stored, transmitted, and quickly restored into a running interpreter. We describe the Tcl 9.1 bytecode model, the full save/load pipeline (`tbcx::save`, `tbcx::load`), the function-level architecture, and performance/portability considerations (including a little-endian fast path). We also provide guidance for future maintainers and an executable appendix for hands-on testing using `tbcx::savefile`/`tbcx::loadfile`.

1 Introduction

Tcl compiles scripts to bytecode at runtime and caches the result, but this compiled form traditionally does not persist across processes. `tbcx` addresses that gap with a robust serializer and loader for Tcl 9.1 bytecode. The accompanying disassembler (`tbcxdump.c`) helps validate and inspect generated TBCX streams.

2 Tcl 9.1 Bytecode Compiler (Background)

A compiled script is represented by a `ByteCode` structure holding: (i) a linear instruction array, (ii) a literal table of `Tcl_Obj` constants, (iii) `AuxData` (e.g., jump tables, dict update descriptors, foreach metadata), (iv) exception ranges, and (v) metadata (max stack, number of locals, namespace). The dumper shows the instruction set and decoded structures for analysis.

3 Serialization with `tbcx::save`

The serializer lives in `tbcxsave.c` and is orchestrated by `EmitTopLevelAndProcs`, which scans, compiles, and emits all components of a script to a TBCX stream.

3.1 Entry Points and High-Level Flow

User-facing commands (all funneled through `EmitTopLevelAndProcs` after opening/obtaining a Tcl channel): `tbcx::save`, `tbcx::savechan`, and `tbcx::savefile`.

1. **Proc capture (static + dynamic).** Static `proc` forms are parsed by `CollectProcsFromScript`; dynamic forms (created at runtime) are captured by running the script in a child interpreter via `CollectProcsByEval`; lists are deduplicated with `MergeProcLists`.
2. **Top-level filtering and compile.** The script is filtered to remove redundant proc creation forms (`BuildTopLevelFiltered`) and compiled to `ByteCode` with `GetByteCodeFromScript`.
3. **Header and core blocks.** `WriteHeaderEx` emits the `TbcxHeader` (magic/format/flags, lengths, counts, locals, stack). Then the raw bytecode, literals (`WriteLiteral`), supported AuxData (`WriteAux`), and exception ranges are written; `AssertAuxCoverage` ensures only supported Aux kinds appear.

4. **Procedures.** Each proc body is compiled via `CompileProcBodyForSave` (backed by the core compiler), optionally patched by `PeepholeNeutralizeProcCreates` to neutralize embedded `proc` creation sequences, and then emitted (code, literals, aux, ranges, max stack, local count).

5. **OO constructs.** Classes/methods are collected both statically (`CollectOOFromScript`) and dynamically (`CollectOOByEval`); lists are merged via `MergeClassLists`/`MergeMethLists`; method bodies are compiled with `CompileMethodBodyForSave` prior to emission.

6. **Namespace bodies.** `ScanNamespaceEvalBodies` finds `namespace eval` sites by scanning bytecode; `CompileNsBodyForSave` compiles those bodies and they are serialized with their target namespace and literal index for reattachment on load.

3.2 Serialization Helpers

Low-level writers `wr`/`wr1`/`wr4`/`wr8` wrap `Tcl_WriteRaw`; `WriteLiteral` handles booleans/ints/-doubles/bignums/bytearrays/lists/dicts/strings; `WriteAux_*` serializes jump tables (string/num), dict-update, and foreach metadata; `AssertAuxCoverage` checks completeness.

3.3 Lambda Serialization and Upgrade

Anonymous functions ("lambdas") are represented as list literals like `{argList body}` (optionally with namespace). **On save**, lambdas are not tagged specially; they are serialized as ordinary lists by `WriteLiteral`, keeping the format simple. **On load**, lambdas are "upgraded" by `PrewarmLambdaLiterals`, which scans literal tables and converts lambda-shaped values to compiled bytecode (`tbcxTyLambda` $\rightarrow$ `tbcxTyBytecode`) so later calls are fast.

4 Deserialization and Execution with `tbcx::load`

Loading lives in `tbcxload.c`. Both `tbcx::loadfile` and `tbcx::loadchan` call `LoadFromChannel`, which reconstructs all blocks and then executes the top-level.

4.1 Top-Level Flow

1. **Header and validation.** Read/validate `TbcxHeader` (magic/format/bounds).

2. **Top-level `ByteCode`.** `ReadByteCode` allocates a fresh `ByteCode`, copies raw instruction bytes, rebuilds literals (`ReadOneLiteral`), decodes AuxData (`GetAuxTypeByKind`), exception ranges, and finalizes. `TbcxFinalizeByteCode` marks it precompiled and wires interpreter/namespace epochs.

3. **Procedures.** For each proc, `ReadByteCode` loads the body; `InstallPrecompiledProc` creates a proc, attaches the precompiled body (`ByteCodeSetInternalRep`), and sets `numCompiledLocals`.

4. **Classes/methods.** Classes via `oo::class create`, superclasses via `oo::define superclass`; methods are bound by wrapping compiled bodies in `Tcl_Objs` and issuing `oo::define method/constructor/destructor`.

5. **Namespace bodies.** Nested `namespace eval` blocks are loaded as `ByteCode` and reattached at their literal indices; `EnsureNamespaceFromString` resolves or creates the target namespaces.

6. **Execution.** The top-level block is installed as a temporary proc and invoked; the wrapper is then removed.

4.2 Supporting Functions and Safety

`ReadOneLiteral` reconstructs booleans, (wide)ints, doubles, bignums, byte arrays, lists, dicts (with bounds checks). `PrewarmLambdaLiterals` upgrades lambda lists to compiled form. `FreeLoadedByteCode` cleans up on error.

5 Performance and Portability

5.1 Endian-Stable Encoding with LE Fast Path

All multibyte integers in TBCX are serialized little-endian. On little-endian hosts, `putle32`/`putle64` become raw `memcpy` (no byte swaps); on big-endian hosts, `TBCX_BSWAP32`/`TBCX_BSWAP64` ensure correctness. Host endianness is detected via `tbcxHostIsLE`, and read helpers `le16`/`le32`/`le64` mirror this symmetry.

5.2 Direct Channel I/O and Minimal Copies

Writers `wr`/`wr1`/`wr4`/`wr8` call `Tcl_WriteRaw` directly; readers use `ReadAll` to fetch exact blocks with bounds checks. This minimizes copies and surprises from buffering.

5.3 Precompiled Attachment & Epoch Wiring

`TbcxFinalizeByteCode` marks blocks as precompiled and attaches interpreter/namespace epochs so the engine can skip parsing/compilation and keep resolution in sync. Bodies are attached via `ByteCodeSetInternalRep` where appropriate.

5.4 Semantic Peephole Patching

`PeepholeNeutralizeProcCreates` replaces embedded proc-creation sequences with benign forms so load does not replay definitions, preserving semantics and trimming load time.

5.5 Header Layout (for reference)

Header Field	Description
Magic (32b)	TBCX identifier
Format (32b)	Version (9 for Tcl 9.1)
Flags (32b)	Reserved (V1 = 0)
CodeLen (64b)	Top-level bytecode length
NumCmds (32b)	Number of compiled commands
NumExcept (32b)	Exception ranges
NumLiterals (32b)	Literal count
NumAux (32b)	Aux data count
NumLocals (32b)	Local variable slots
MaxStack (32b)	Maximum stack depth

See `TbcxHeader` in `tbcx.h`.

6 Guidance for Future Maintainers

Types/Aux coverage. Extend serializer/deserializer when Tcl adds new `AuxData` or literal types; see `AssertAuxCoverage` and the reader's `GetAuxTypeByKind`.
Namespaces. Use `EnsureNamespaceFromString` for predictable resolution/creation; avoid surprising globals.

Memory management. Manage refcounts for literals; free Aux client data. `FreeLoadedByteCode` shows the correct teardown path.
Channels/pipelines. Prefer `savechan`/`loadchan` to plumb compression, encryption, or authentication without changing the TBCX format. Initialization wires public commands in `tbcx.c`.

7 Conclusion

`tbcx` brings persistent, portable bytecode to Tcl 9.1 with a clear, symmetric save/load design and strong performance characteristics (LE fast path, precompiled attachment, and semantic patching). The library is suitable for large applications seeking faster startup, code distribution, and secure deployment, and provides a maintainable foundation for future evolution.

Appendix: Hands-On Round-Trip (Tcl Shell)

Sample Source (`app.tcl`)

```
namespace eval ::demo {
  proc greet {name} { return "Hello, $name!" }
}

oo::class create ::demo::Greeter {
  constructor {who} { set myWho $who }
  method hello {} { return "Hello from [info object class [self]]" }
}
```

Compile & Save

```
package require tbcx
tbcx::savefile app.tcl app.tbcx
```

Internally, `savefile` compiles the input and writes: header, top-level block, procs, classes/methods, and namespace bodies (`Tbcx_SaveFileObjCmd` → `EmitTopLevelAndProcs`).

Load & Run

```
package require tbcx
tbcx::loadfile app.tbcx

demo::greet World
# -> Hello, World!

set obj [::demo::Greeter new "Ada"]
$obj hello
```

`tbcx::loadfile` reconstructs the top-level block, installs precompiled proc bodies, creates classes, binds methods, and attaches namespace-eval bodies (`LoadFromChannel`, `InstallPrecompiledProc`, `ReadByteCode`).

Acknowledgments

Thanks to the Tcl core for public/internal APIs enabling safe precompilation and to the reviewers who improved the serializer/loader design.