# TBCX for Tcl 9.1:
# Design, Format, and Implementation Notes

TBCX Project

September 2025

**Abstract**

This document describes the TBCX (Tcl ByteCode eXchange) file format and the accompanying `tbcx` extension for Tcl 9.1. TBCX allows precompiling Tcl scripts to a compact, portable, little-endian binary representation that can be loaded later without recompilation. We cover the public API, the exact on-disk format, loader behavior, and implementation details relevant to embedding and tooling.

## 1 Overview and Goals

TBCX goals:

- **Zero source recompilation at load time**: ship precompiled bodies.
- **Portability**: files are little-endian and validated at load.
- **Determinism**: dictionaries and jump tables are serialized in a stable order.
- **Safety**: only supported AuxData kinds are encoded; inputs are size-checked.

The package provides `tbcx::save`, `tbcx::savechan`, `tbcx::savefile`, `tbcx::loadchan`, `tbcx::loadfile`, and `tbcx::dumpfile`. The package version is `1.0`.

## 2 Public API

### 2.1 Saving

`tbcx::save` *script outPath*
: Compile *script* and write a TBCX file.

`tbcx::savechan` *script channelName*
: Compile and write to an open channel; the command configures the channel for binary I/O.

`tbcx::savefile` *in.tcl outPath*
: Read *in.tcl*, compile, and write a TBCX file.

### 2.2 Loading and Inspecting

`tbcx::loadfile` *in.tbcx*
: Load and evaluate a TBCX file in the current namespace.

`tbcx::loadchan` *channelName*
: Load and evaluate from a channel (configured for binary).

`tbcx::dumpfile` *filename*
: Return a text dump (header, sections, disassembly) for inspection.

## 3 Stream Header

All integers are little-endian. The header layout is:

```
struct TbcxHeader {
    u32 magic;       // "TBCX" = 0x58434254
    u32 format;      // 9
    u32 tcl_version; // maj<<24 | min<<16 | patch<<8 | type
    u64 codeLenTop;
    u32 numCmdsTop;    // always 0 in this version
    u32 numExceptTop;
    u32 numLitsTop;
    u32 numAuxTop;
    u32 numLocalsTop;
    u32 maxStackTop;
};
```

The loader validates *magic* and *format* before continuing.

## 4 Compiled Block Encoding

A compiled block serializes one `ByteCode` object in five parts:

1. **Code**: `u32` byte length, then raw code bytes.
2. **Literals**: `u32` count, then tagged literals (Section 5).
3. **AuxData**: `u32` count, then tagged AuxData (Section 6).
4. **Exceptions**: `u32` count, then range table (fields as `u32`).
5. **Epilogue**: `u32` maxStack, `u32` reserved (0), `u32` numLocals.

For top-level compilation the default namespace is the current one at load time. For nested `lambdaExpr` and `bytecode` literals the stream precedes the block with a namespace FQN (LPString) and ensures that namespace exists before instantiation.

### 4.1 Limits

Enforced maxima: code $\leq$ 1 GiB; literals $\leq$ 64 M; AuxData $\leq$ 64 M; exceptions $\leq$ 64 M; any LPString $\leq$ 16 MiB.

## 5 Literals

Each literal begins with a `u32` tag:

| | |
|---|---|
| 0 | **BIGNUM**: `u8` sign (0=zero, 1=+, 2=−), `u32` magLen, then `magLen` LE bytes. |
| 1 | **BOOLEAN**: `u8` (0/1). |
| 2 | **BYTEARR**: `u32` length + bytes. |
| 3 | **DICT**: `u32` pair count, then key literal + value literal (keys sorted by UTF-8). |
| 4 | **DOUBLE**: 64-bit IEEE-754 as `u64`. |
| 5 | **LIST**: `u32` element count, then that many nested literals. |
| 6 | **STRING**: LPString (`u32` length + bytes). |
| 7 | **WIDEINT**: signed 64-bit two's complement (`u64` payload). |
| 8 | **WIDEUINT**: unsigned 64-bit integer (promoted to bignum by loader if needed). |
| 9 | **LAMBDA_BC**: ns FQN (LPString), `u32` *numArgs*; for each arg: name (LPString), `u8` hasDefault, op |
| 10 | **BYTECODE**: ns FQN (LPString) + compiled block. |

**Lambda bodies.** For **LAMBDA_BC** the saver marshals the public *args* list and compiles the body using proc semantics, creating a temporary `Proc` whose argument locals match the lambda signature.

## 6 AuxData

Supported AuxData tags:

**JT_STR (0)**
`u32` count; for each entry: LPString key, `u32` target offset.

**JT_NUM (1)**
`u32` count; for each entry: `u64` key, `u32` target offset.

**DICT_UPDATE (2)**
`u32` length; then that many local indices.

**NEW_FOREACH (3), FOREACH (4)**
`u32` numLists, `u32` loopCtTemp, `u32` firstValueTemp, duplicate `numLists`; for each list: `u32` numVars, then indices.

The loader maps either FOREACH tag to the closest available core AuxData type.

## 7 Procedures, Classes, and Methods

### 7.1 Static capture and sections

The saver parses the source text to capture static definitions: `proc name args body`, `namespace eval` *`ns`* `body`, and `oo::define` forms for methods/ctors/dtors, plus `oo::class create`. For each captured proc it stores three LPStrings (name as written, ns FQN, and the canonical args list) followed by the compiled body. Classes are written with their FQN and a list of superclasses (0 for now). Methods are written with class FQN, kind (inst/class/ctor/dtor), optional name, args list, and compiled body.

### 7.2 Load-time reconstruction

The loader installs temporary shims:

- **proc shim**: on `proc name args body`, if the fully-qualified name and *args* match an entry from the file, substitute the precompiled body; otherwise forward unchanged.
- **oo::define shim**: for `method`, `classmethod`, `constructor`, and `destructor`, substitute the precompiled body when the *args* list matches.

After reading all sections the loader evaluates the top-level block (with the current namespace as default), then restores the original commands and discards the shims.

### 7.3 Locals and variable frames

For each compiled body the saver records `numLocals`. The loader extends the `Proc`'s compiled-local list to at least that length to keep Tcl's invariants when freeing or redefining the proc later.

## 8 Disassembly Tool

The command `tbcx::dumpfile` produces a readable dump: header, compiled-block structure, AuxData, exceptions, and a disassembly. The decoder table is aligned to Tcl 9.1's `tclInstructionTable`. Opcodes marked "deprecated" correspond to legacy one-byte forms and may still be shown to aid debugging.

## 9 Portability and Limits

Streams are little-endian; both writer and reader swap on big-endian hosts. Size limits at save and load prevent pathological inputs (1 GiB code; pools up to 64 M elements; LPString up to 16 MiB).

## 10 Examples

### 10.1 Proc and class

```
set s {
  proc add {a b} { expr {$a + $b} }
  oo::class create C
  oo::define C method m {x} { expr {$x * 2} }
}
tbcx::save $s out.tbcx
tbcx::loadfile out.tbcx
puts [add 2 3] ;# => 5
puts [[C new] m 10] ;# => 20
```

### 10.2 Channels

```
set ch [open "bundle.tbcx" w]
fconfigure $ch -translation binary -eofchar {}
tbcx::savechan {proc p {} {return ok}} $ch
close $ch

set ch [open "bundle.tbcx" r]
fconfigure $ch -translation binary -eofchar {}
tbcx::loadchan $ch
close $ch
```

## 11 Build and Initialization Notes

The extension uses Tcl's stubs and TomMath stubs. During initialization the package resolves the necessary `Tcl_ObjType`s (`bytecode`, `list`, `dict`, `int`, `double`, `boolean`, `bytearray`, `bignum`, and optionally `lambdaExpr`) and the AuxData type pointers (`JumptableInfo`, `JumptableNumInfo`, `DictUpdateInfo`, `ForeachInfo`, `NewForeachInfo`).

**Endianness.** Endianness is detected once (via `tcl_platform(byteOrder)` or a probe) and cached; streams are always written and read as little-endian.

## Acknowledgments

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

## 12 License