Algorand Transaction Execution Approval Language

May 19, 2023

Abstract

Algorand allows transactions to be effectively signed by a small program. If the program evaluates to true then the transaction is allowed. This document defines the language and bytecode format.

Contents

1 The Algorand Virtual Machine (AVM) and TEAL.

The AVM is a bytecode based stack interpreter that executes programs associated with Algorand transactions. TEAL is an assembly language syntax for specifying a program that is ultimately converted to AVM bytecode. These programs can be used to check the parameters of the transaction and approve the transaction as if by a signature. This use is called a *Smart Signature*. Starting with v2, these programs may also execute as *Smart Contracts*, which are often called *Applications*. Contract executions are invoked with explicit application call transactions.

Programs have read-only access to the transaction they are attached to, the other transactions in their atomic transaction group, and a few global values. In addition, *Smart Contracts* have access to limited state that is global to the application, per-account local state for each account that has opted-in to the application, and additional per-application arbitrary state in named *boxes*. For both types of program, approval is signaled by finishing with the stack containing a single non-zero uint64 value, though `return` can be used to signal an early approval which approves based only upon the top stack value being a non-zero uint64 value.

1.1 The Stack

The stack starts empty and can contain values of either uint64 or byte-arrays (byte-arrays may not exceed 4096 bytes in length). Most operations act on the stack, popping arguments from it and pushing results to it. Some operations have *immediate* arguments that are encoded directly into the instruction, rather than coming from the stack.

The maximum stack depth is 1000. If the stack depth is exceeded or if a byte-array element exceeds 4096 bytes, the program fails. If an opcode is documented to access a position in the stack that does not exist, the operation fails. Most often, this is an attempt to access an element below the stack – the simplest example is an operation like `concat` which expects two arguments on the stack. If the stack has fewer than two elements, the operation fails. Some operations, like `frame_dig` and `proto` could fail because of an attempt to access above the current stack.

1.2 Scratch Space

In addition to the stack there are 256 positions of scratch space. Like stack values, scratch locations may be uint64s or byte-arrays. Scratch locations are initialized as uint64 zero. Scratch space is accessed by the `load(s)` and `store(s)` opcodes which move data from or to scratch space, respectively. Application calls may inspect the final scratch space of earlier application calls in the same group using `gload(s)(s)`

1.3 Versions

In order to maintain existing semantics for previously written programs, AVM code is versioned. When new opcodes are introduced, or behavior is changed, a new version is introduced. Programs carrying old versions are executed with their original semantics. In the AVM bytecode, the version is an incrementing integer, currently 6, and denoted vX throughout this document.

1.4 Execution Modes

Starting from v2, the AVM can run programs in two modes: 1. LogicSig or *stateless* mode, used to execute Smart Signatures 2. Application or *stateful* mode, used to execute Smart Contracts

Differences between modes include: 1. Max program length (consensus parameters LogicSigMaxSize, MaxAppTotalProgramLen & MaxExtraAppProgramPages) 2. Max program cost (consensus parameters LogicSigMaxCost, MaxAppProgramCost) 3. Opcode availability. Refer to opcodes document for details. 4. Some global values, such as LatestTimestamp, are only available in stateful mode. 5. Only Applications can observe transaction effects, such as Logs or IDs allocated to ASAs or new Applications.

1.5 Execution Environment for Smart Signatures

Smart Signatures execute as part of testing a proposed transaction to see if it is valid and authorized to be committed into a block. If an authorized program executes and finishes with a single non-zero uint64 value on the stack then that program has validated the transaction it is attached to.

The program has access to data from the transaction it is attached to (`txn` op), any transactions in a transaction group it is part of (`gtxn` op), and a few global values like consensus parameters (`global` op). Some "Args" may be attached to a transaction being validated by a program. Args are an array of byte strings. A common pattern would be to have the key to unlock some contract as an Arg. Be aware that Smart Signature Args are recorded on the blockchain and publicly visible when the transaction is submitted to the network, even before the transaction has been included in a block. These Args are *not* part of the transaction ID nor of the TxGroup hash. They also cannot be read from other programs in the group of transactions.

A program can either authorize some delegated action on a normal signature-based or multisignature-based account or be wholly in charge of a contract account.

- If the account has signed the program (by providing a valid ed25519 signature or valid multisignature for the authorizer address on the string "Program" concatenated with the program bytecode) then: if the program returns true the transaction is authorized as if the account had signed it. This allows an account to hand out a signed program so that other users can carry out delegated actions which are approved by the program. Note that Smart Signature Args are *not* signed.
- If the SHA512_256 hash of the program (prefixed by "Program") is equal to authorizer address of the transaction sender then this is a contract account wholly controlled by the program. No other signature is necessary or possible. The only way to execute a transaction against the contract account is for the program to approve it.

The bytecode plus the length of all Args must add up to no more than 1000 bytes (consensus parameter LogicSigMaxSize). Each opcode has an associated cost and the program cost must total no more than 20,000 (consensus parameter LogicSigMaxCost). Most opcodes have a cost of 1, but a few slow cryptographic operations have a much higher cost. Prior to v4, the program's cost was estimated as the static sum of all the opcode costs in the program (whether they were actually executed or not). Beginning with v4, the program's cost is tracked dynamically, while being evaluated. If the program exceeds its budget, it fails.

1.6 Execution Environment for Smart Contracts (Applications)

Smart Contracts are executed in ApplicationCall transactions. Like Smart Signatures, contracts indicate success by leaving a single non-zero integer on the stack. A failed Smart Contract call to an ApprovalProgram is not a valid transaction, thus not written to the blockchain. An ApplicationCall with OnComplete set to ClearState invokes the ClearStateProgram, rather than the usual ApprovalProgram. If the ClearStateProgram fails, application state changes are rolled back, but the transaction still succeeds, and the Sender's local state for the called application is removed.

Smart Contracts have access to everything a Smart Signature may access (see previous section), as well as the ability to examine blockchain state such as balances and contract state (their own state and the state of other contracts). They also have access to some global values that are not visible to Smart Signatures because the values change over time. Since smart contracts access changing state, nodes must rerun their code to determine if the ApplicationCall transactions in their pool would still succeed each time a block is added to the blockchain.

Smart contracts have limits on their execution cost (700, consensus parameter MaxAppProgramCost). Before v4, this was a static limit on the cost of all the instructions in the program. Starting in v4, the cost is tracked dynamically during execution and must not exceed MaxAppProgramCost. Beginning with v5, programs costs are pooled and tracked dynamically across app executions in a group. If `n` application invocations appear in a group, then the total execution cost of all such calls must not exceed `n`*MaxAppProgramCost. In v6, inner application calls become possible, and each such call increases the pooled budget by MaxAppProgramCost at the time the inner group is submitted with `itxn_submit`.

Executions of the ClearStateProgram are more stringent, in order to ensure that applications may be closed out, but that applications also are assured a chance to clean up their internal state. At the beginning of the execution of a ClearStateProgram, the pooled budget available must be MaxAppProgramCost or higher. If it is not, the containing transaction group fails without clearing the app's state. During the execution of the ClearStateProgram, no more than MaxAppProgramCost may be drawn. If further execution is attempted, the ClearStateProgram fails, and the app's state *is cleared*.

1.6.1 Resource availability

Smart contracts have limits on the amount of blockchain state they may examine. Opcodes may only access blockchain resources such as Accounts, Assets, Boxes, and contract state if the given resource is *available*.

- A resource in the "foreign array" fields of the ApplicationCall transaction (`txn.Accounts`, `txn.ForeignAssets`, and `txn.ForeignApplications`) is *available*.
- The `txn.Sender`, `global CurrentApplicationID`, and `global CurrentApplicationAddress` are *available*.
- Prior to v4, all assets were considered *available* to the `asset_holding_get` opcode, and all applications were *available* to the `app_local_get_ex` opcode.
- Since v6, any asset or contract that was created earlier in the same transaction group (whether by a top-level or inner transaction) is *available*. In addition, any account that is the associated account of a contract that was created earlier in the group is *available*.
- Since v7, the account associated with any contract present in the `txn.ForeignApplications` field is *available*.
- Since v9, there is group-level resource sharing. Any resource that is available in *some* top-level transaction in a transaction group is available in *all* v9 or later application calls in the group, whether those application calls are top-level or inner.
- When considering whether an asset holding or application local state is available by group-level resource sharing, the holding or local state must be available in a top-level transaction without considering group sharing. For example, if account A is made available in one transaction, and asset

X is made available in another, group resource sharing does *not* make A's X holding available.

- Top-level transactions that are not application calls also make resources available to group-level resource sharing. The following resources are made available by other transaction types.

  1. `pay` - `txn.Sender`, `txn.Receiver`, and `txn.CloseRemainderTo` (if set).
  2. `keyreg` - `txn.Sender`
  3. `acfg` - `txn.Sender`, `txn.ConfigAsset`, and the `txn.ConfigAsset` holding of `txn.Sender`.
  4. `axfer` - `txn.Sender`, `txn.AssetReceiver`, `txn.AssetSender` (if set), `txnAssetCloseTo` (if set), `txn.XferAsset`, and the `txn.XferAsset` holding of each of those accounts.
  5. `afrz` - `txn.Sender`, `txn.FreezeAccount`, `txn.FreezeAsset`, and the `txn.FreezeAsset` holding of `txn.FreezeAccount`. The `txn.FreezeAsset` holding of `txn.Sender` is *not* made available.

- A Box is *available* to an Approval Program if *any* transaction in the same group contains a box reference (`txn.Boxes`) that denotes the box. A box reference contains an index `i`, and name `n`. The index refers to the `ith` application in the transaction's ForeignApplications array, with the usual convention that 0 indicates the application ID of the app called by that transaction. No box is ever *available* to a ClearStateProgram.

Regardless of *availability*, any attempt to access an Asset or Application with an ID less than 256 from within a Contract will fail immediately. This avoids any ambiguity in opcodes that interpret their integer arguments as resource IDs *or* indexes into the `txn.ForeignAssets` or `txn.ForeignApplications` arrays.

It is recommended that contract authors avoid supplying array indexes to these opcodes, and always use explicit resource IDs. By using explicit IDs, contracts will better take advantage of group resource sharing. The array indexing interpretation may be deprecated in a future version.

1.7 Constants

Constants can be pushed onto the stack in two different ways:

1. Constants can be pushed directly with `pushint` or `pushbytes`. This method is more efficient for constants that are only used once.
2. Constants can be loaded into storage separate from the stack and scratch space, using two opcodes `intcblock` and `bytecblock`. Then, constants from this storage can be pushed onto the stack by referring to the type and

index using `intc`, `intc_[0123]`, `bytec`, and `bytec_[0123]`. This method is more efficient for constants that are used multiple times.

The assembler will hide most of this, allowing simple use of `int 1234` and `byte 0xcafed00d`. Constants introduced via `int` and `byte` will be assembled into appropriate uses of `pushint|pushbytes` and `{int|byte}c, {int|byte}c_[0123]` to minimize program size.

The opcodes `intcblock` and `bytecblock` use proto-buf style variable length unsigned int, reproduced here. The `intcblock` opcode is followed by a varuint specifying the number of integer constants and then that number of varuints. The `bytecblock` opcode is followed by a varuint specifying the number of byte constants, and then that number of pairs of (varuint, bytes) length prefixed byte strings.

1.7.1 Named Integer Constants

1.7.1.1 OnComplete

An application transaction must indicate the action to be taken following the execution of its approvalProgram or clearStateProgram. The constants below describe the available actions.

Value	Name	Description
0	NoOp	Only execute the `ApprovalProgram` associated with this application ID, with no additional effects.
1	OptIn	Before executing the `ApprovalProgram`, allocate local state for this application into the sender's account data.
2	CloseOut	After executing the `ApprovalProgram`, clear any local state for this application out of the sender's account data.
3	ClearState	Don't execute the `ApprovalProgram`, and instead execute the `ClearStateProgram` (which may not reject this transaction). Additionally, clear any local state for this application out of the sender's account data as in `CloseOutOC`.
4	UpdateApplication	After executing the `ApprovalProgram`, replace the `ApprovalProgram` and `ClearStateProgram` associated with this application ID with the programs specified in this transaction.

Value	Name	Description
5	DeleteApplication	After executing the `ApprovalProgram`, delete the application parameters from the account data of the application's creator.

1.7.1.2 TypeEnum constants

Value	Name	Description
0	unknown	Unknown type. Invalid transaction
1	pay	Payment
2	keyreg	KeyRegistration
3	acfg	AssetConfig
4	axfer	AssetTransfer
5	afrz	AssetFreeze
6	appl	ApplicationCall

1.8 Operations

Most operations work with only one type of argument, uint64 or bytes, and fail if the wrong type value is on the stack.

Many instructions accept values to designate Accounts, Assets, or Applications. Beginning with v4, these values may be given as an *offset* in the corresponding Txn fields (Txn.Accounts, Txn.ForeignAssets, Txn.ForeignApps) *or* as the value itself (a byte-array address for Accounts, or a uint64 ID). The values, however, must still be present in the Txn fields. Before v4, most opcodes required the use of an offset, except for reading account local values of assets or applications, which accepted the IDs directly and did not require the ID to be present in they corresponding *Foreign* array. (Note that beginning with v4, those IDs *are* required to be present in their corresponding *Foreign* array.) See individual opcodes for details. In the case of account offsets or application offsets, 0 is specially defined to Txn.Sender or the ID of the current application, respectively.

This summary is supplemented by more detail in the opcodes document.

Some operations immediately fail the program. A transaction checked by a program that fails is not valid. An account governed by a buggy program might not have a way to get assets back out of it. Code carefully.

In the documentation for each opcode, the stack arguments that are popped are referred to alphabetically, beginning with the deepest argument as `A`. These arguments are shown in the opcode description, and if the opcode must be of a specific type, it is noted there. All opcodes fail if a specified type is incorrect.

If an opcode pushes more than one result, the values are named for ease of exposition and clarity concerning their stack positions. When an opcode manipulates the stack in such a way that a value changes position but is otherwise unchanged, the name of the output on the return stack matches the name of the input value.

1.8.1 Arithmetic, Logic, and Cryptographic Operations

Opcode	Description
`sha256`	SHA256 hash of value A, yields [32]byte
`keccak256`	Keccak256 hash of value A, yields [32]byte
`sha512_256`	SHA512_256 hash of value A, yields [32]byte
`sha3_256`	SHA3_256 hash of value A, yields [32]byte
`ed25519verify`	for (data A, signature B, pubkey C) verify the signature of ("ProgData" || program_hash || data) against the pubkey => {0 or 1}
`ed25519verify_bare`	for (data A, signature B, pubkey C) verify the signature of the data against the pubkey => {0 or 1}
`ecdsa_verify v`	for (data A, signature B, C and pubkey D, E) verify the signature of the data against the pubkey => {0 or 1}
`ecdsa_pk_recover v`	for (data A, recovery id B, signature C, D) recover a public key
`ecdsa_pk_decompress v`	decompress pubkey A into components X, Y
`vrf_verify s`	Verify the proof B of message A against pubkey C. Returns vrf output and verification flag.
`+`	A plus B. Fail on overflow.
`-`	A minus B. Fail if B > A.
`/`	A divided by B (truncated division). Fail if B == 0.
`*`	A times B. Fail on overflow.
`<`	A less than B => {0 or 1}
`>`	A greater than B => {0 or 1}
`<=`	A less than or equal to B => {0 or 1}
`>=`	A greater than or equal to B => {0 or 1}
`&&`	A is not zero and B is not zero => {0 or 1}
`||`	A is not zero or B is not zero => {0 or 1}
`shl`	A times 2^B, modulo 2^64
`shr`	A divided by 2^B
`sqrt`	The largest integer I such that I^2 <= A
`bitlen`	The highest set bit in A. If A is a byte-array, it is interpreted as a big-endian unsigned integer. bitlen of 0 is 0, bitlen of 8 is 4

Opcode	Description
`exp`	A raised to the Bth power. Fail if A == B == 0 and on overflow
`==`	A is equal to B => {0 or 1}
`!=`	A is not equal to B => {0 or 1}
`!`	A == 0 yields 1; else 0
`len`	yields length of byte value A
`itob`	converts uint64 A to big-endian byte array, always of length 8
`btoi`	converts big-endian byte array A to uint64. Fails if len(A) > 8. Padded by leading 0s if len(A) < 8.
`%`	A modulo B. Fail if B == 0.
`|`	A bitwise-or B
`&`	A bitwise-and B
`^`	A bitwise-xor B
`~`	bitwise invert value A
`mulw`	A times B as a 128-bit result in two uint64s. X is the high 64 bits, Y is the low
`addw`	A plus B as a 128-bit result. X is the carry-bit, Y is the low-order 64 bits.
`divw`	A,B / C. Fail if C == 0 or if result overflows.
`divmodw`	W,X = (A,B / C,D); Y,Z = (A,B modulo C,D)
`expw`	A raised to the Bth power as a 128-bit result in two uint64s. X is the high 64 bits, Y is the low. Fail if A == B == 0 or if the results exceeds 2^128-1
`getbit`	Bth bit of (byte-array or integer) A. If B is greater than or equal to the bit length of the value (8*byte length), the program fails
`setbit`	Copy of (byte-array or integer) A, with the Bth bit set to (0 or 1) C. If B is greater than or equal to the bit length of the value (8*byte length), the program fails
`getbyte`	Bth byte of A, as an integer. If B is greater than or equal to the array length, the program fails
`setbyte`	Copy of A with the Bth byte set to small integer (between 0..255) C. If B is greater than or equal to the array length, the program fails
`concat`	join A and B

1.8.2 Byte Array Manipulation

Opcode	Description
`substring s e`	A range of bytes from A starting at S up to but not including E. If E < S, or either is larger than the array length, the program fails
`substring3`	A range of bytes from A starting at B up to but not including C. If C < B, or either is larger than the array length, the program fails
`extract s l`	A range of bytes from A starting at S up to but not including S+L. If L is 0, then extract to the end of the string. If S or S+L is larger than the array length, the program fails
`extract3`	A range of bytes from A starting at B up to but not including B+C. If B+C is larger than the array length, the program fails`extract3` can be called using `extract` with no immediates.
`extract_uint16`	A uint16 formed from a range of big-endian bytes from A starting at B up to but not including B+2. If B+2 is larger than the array length, the program fails
`extract_uint32`	A uint32 formed from a range of big-endian bytes from A starting at B up to but not including B+4. If B+4 is larger than the array length, the program fails
`extract_uint64`	A uint64 formed from a range of big-endian bytes from A starting at B up to but not including B+8. If B+8 is larger than the array length, the program fails
`replace2 s`	Copy of A with the bytes starting at S replaced by the bytes of B. Fails if S+len(B) exceeds len(A)`replace2` can be called using `replace` with 1 immediate.
`replace3`	Copy of A with the bytes starting at B replaced by the bytes of C. Fails if B+len(C) exceeds len(A)`replace3` can be called using `replace` with no immediates.
`base64_decode e`	decode A which was base64-encoded using *encoding* E. Fail if A is not base64 encoded with encoding E
`json_ref r`	key B's value, of type R, from a valid utf-8 encoded json object A

The following opcodes take byte-array values that are interpreted as big-endian unsigned integers. For mathematical operators, the returned values are the shortest byte-array that can represent the returned value. For example, the zero value is the empty byte-array. For comparison operators, the returned value is a uint64.

Input lengths are limited to a maximum length of 64 bytes, representing a 512 bit unsigned integer. Output lengths are not explicitly restricted, though only `b*` and `b+` can produce a larger output than their inputs, so there is an implicit length limit of 128 bytes on outputs.

Opcode	Description
`b+`	A plus B. A and B are interpreted as big-endian unsigned integers
`b-`	A minus B. A and B are interpreted as big-endian unsigned integers. Fail on underflow.
`b/`	A divided by B (truncated division). A and B are interpreted as big-endian unsigned integers. Fail if B is zero.
`b*`	A times B. A and B are interpreted as big-endian unsigned integers.
`b<`	1 if A is less than B, else 0. A and B are interpreted as big-endian unsigned integers
`b>`	1 if A is greater than B, else 0. A and B are interpreted as big-endian unsigned integers
`b<=`	1 if A is less than or equal to B, else 0. A and B are interpreted as big-endian unsigned integers
`b>=`	1 if A is greater than or equal to B, else 0. A and B are interpreted as big-endian unsigned integers
`b==`	1 if A is equal to B, else 0. A and B are interpreted as big-endian unsigned integers
`b!=`	0 if A is equal to B, else 1. A and B are interpreted as big-endian unsigned integers
`b%`	A modulo B. A and B are interpreted as big-endian unsigned integers. Fail if B is zero.
`bsqrt`	The largest integer I such that I^2 <= A. A and I are interpreted as big-endian unsigned integers

These opcodes operate on the bits of byte-array values. The shorter input array is interpreted as though left padded with zeros until it is the same length as the other input. The returned values are the same length as the longer input. Therefore, unlike array arithmetic, these results may contain leading zero bytes.

Opcode	Description
`b|`	A bitwise-or B. A and B are zero-left extended to the greater of their lengths

Opcode	Description
`b&`	A bitwise-and B. A and B are zero-left extended to the greater of their lengths
`b^`	A bitwise-xor B. A and B are zero-left extended to the greater of their lengths
`b~`	A with all bits inverted

1.8.3 Loading Values

Opcodes for getting data onto the stack.

Some of these have immediate data in the byte or bytes after the opcode.

Opcode	Description
`intcblock uint ...`	prepare block of uint64 constants for use by intc
`intc i`	Ith constant from intcblock
`intc_0`	constant 0 from intcblock
`intc_1`	constant 1 from intcblock
`intc_2`	constant 2 from intcblock
`intc_3`	constant 3 from intcblock
`pushint uint`	immediate UINT
`pushints uint ...`	push sequence of immediate uints to stack in the order they appear (first uint being deepest)
`bytecblock bytes ...`	prepare block of byte-array constants for use by bytec
`bytec i`	Ith constant from bytecblock
`bytec_0`	constant 0 from bytecblock
`bytec_1`	constant 1 from bytecblock
`bytec_2`	constant 2 from bytecblock
`bytec_3`	constant 3 from bytecblock
`pushbytes bytes`	immediate BYTES
`pushbytess bytes ...`	push sequences of immediate byte arrays to stack (first byte array being deepest)
`bzero`	zero filled byte-array of length A
`arg n`	Nth LogicSig argument
`arg_0`	LogicSig argument 0
`arg_1`	LogicSig argument 1
`arg_2`	LogicSig argument 2
`arg_3`	LogicSig argument 3
`args`	Ath LogicSig argument
`txn f`	field F of current transaction
`gtxn t f`	field F of the Tth transaction in the current group

Opcode	Description
`txna f i`	Ith value of the array field F of the current transaction`txna` can be called using `txn` with 2 immediates.
`txnas f`	Ath value of the array field F of the current transaction
`gtxna t f i`	Ith value of the array field F from the Tth transaction in the current group`gtxna` can be called using `gtxn` with 3 immediates.
`gtxnas t f`	Ath value of the array field F from the Tth transaction in the current group
`gtxns f`	field F of the Ath transaction in the current group
`gtxnsa f i`	Ith value of the array field F from the Ath transaction in the current group`gtxnsa` can be called using `gtxns` with 2 immediates.
`gtxnsas f`	Bth value of the array field F from the Ath transaction in the current group
`global f`	global field F
`load i`	Ith scratch space value. All scratch spaces are 0 at program start.
`loads`	Ath scratch space value. All scratch spaces are 0 at program start.
`store i`	store A to the Ith scratch space
`stores`	store B to the Ath scratch space
`gload t i`	Ith scratch space value of the Tth transaction in the current group
`gloads i`	Ith scratch space value of the Ath transaction in the current group
`gloadss`	Bth scratch space value of the Ath transaction in the current group
`gaid t`	ID of the asset or application created in the Tth transaction of the current group
`gaids`	ID of the asset or application created in the Ath transaction of the current group

1.8.3.1 Transaction Fields

1.8.3.1.1 Scalar Fields

Index	Name	Type	In	Notes
0	Sender	[]byte		32 byte address
1	Fee	uint64		microalgos

Index	Name	Type	In	Notes
2	FirstValid	uint64		round number
3	FirstValidTime	uint64	v7	UNIX timestamp of block before txn.FirstValid. Fails if negative
4	LastValid	uint64		round number
5	Note	[]byte		Any data up to 1024 bytes
6	Lease	[]byte		32 byte lease value
7	Receiver	[]byte		32 byte address
8	Amount	uint64		microalgos
9	CloseRemainderTo	[]byte		32 byte address
10	VotePK	[]byte		32 byte address
11	SelectionPK	[]byte		32 byte address
12	VoteFirst	uint64		The first round that the participation key is valid.
13	VoteLast	uint64		The last round that the participation key is valid.
14	VoteKeyDilution	uint64		Dilution for the 2-level participation key
15	Type	[]byte		Transaction type as bytes
16	TypeEnum	uint64		Transaction type as integer
17	XferAsset	uint64		Asset ID
18	AssetAmount	uint64		value in Asset's units
19	AssetSender	[]byte		32 byte address. Source of assets if Sender is the Asset's Clawback address.
20	AssetReceiver	[]byte		32 byte address
21	AssetCloseTo	[]byte		32 byte address
22	GroupIndex	uint64		Position of this transaction within an atomic transaction group. A stand-alone transaction is implicitly element 0 in a group of 1
23	TxID	[]byte		The computed ID for this transaction. 32 bytes.
24	ApplicationID	uint64	v2	ApplicationID from ApplicationCall transaction
25	OnCompletion	uint64	v2	ApplicationCall transaction on completion action
27	NumAppArgs	uint64	v2	Number of ApplicationArgs
29	NumAccounts	uint64	v2	Number of Accounts
30	ApprovalProgram	[]byte	v2	Approval program
31	ClearStateProgram	[]byte	v2	Clear state program
32	RekeyTo	[]byte	v2	32 byte Sender's new AuthAddr

Index	Name	Type	In	Notes
33	ConfigAsset	uint64	v2	Asset ID in asset config transaction
34	ConfigAssetTotal	uint64	v2	Total number of units of this asset created
35	ConfigAssetDecimals	uint64	v2	Number of digits to display after the decimal place when displaying the asset
36	ConfigAssetDefaultFrozen	uint64	v2	Whether the asset's slots are frozen by default or not, 0 or 1
37	ConfigAssetUnitName	[]byte	v2	Unit name of the asset
38	ConfigAssetName	[]byte	v2	The asset name
39	ConfigAssetURL	[]byte	v2	URL
40	ConfigAssetMetadataHash	[]byte	v2	32 byte commitment to unspecified asset metadata
41	ConfigAssetManager	[]byte	v2	32 byte address
42	ConfigAssetReserve	[]byte	v2	32 byte address
43	ConfigAssetFreeze	[]byte	v2	32 byte address
44	ConfigAssetClawback	[]byte	v2	32 byte address
45	FreezeAsset	uint64	v2	Asset ID being frozen or un-frozen
46	FreezeAssetAccount	[]byte	v2	32 byte address of the account whose asset slot is being frozen or un-frozen
47	FreezeAssetFrozen	uint64	v2	The new frozen value, 0 or 1
49	NumAssets	uint64	v3	Number of Assets
51	NumApplications	uint64	v3	Number of Applications
52	GlobalNumUint	uint64	v3	Number of global state integers in ApplicationCall
53	GlobalNumByteSlice	uint64	v3	Number of global state byteslices in ApplicationCall
54	LocalNumUint	uint64	v3	Number of local state integers in ApplicationCall
55	LocalNumByteSlice	uint64	v3	Number of local state byteslices in ApplicationCall
56	ExtraProgramPages	uint64	v4	Number of additional pages for each of the application's approval and clear state programs. An ExtraProgramPages of 1 means 2048 more total bytes, or 1024 for each program.
57	Nonparticipation	uint64	v5	Marks an account nonparticipating for rewards

Index	Name	Type	In	Notes
59	NumLogs	uint64	v5	Number of Logs (only with `itxn` in v5). Application mode only
60	CreatedAssetID	uint64	v5	Asset ID allocated by the creation of an ASA (only with `itxn` in v5). Application mode only
61	CreatedApplicationID	uint64	v5	ApplicationID allocated by the creation of an application (only with `itxn` in v5). Application mode only
62	LastLog	[]byte	v6	The last message emitted. Empty bytes if none were emitted. Application mode only
63	StateProofPK	[]byte	v6	64 byte state proof public key
65	NumApprovalProgramPages	uint64	v7	Number of Approval Program pages
67	NumClearStateProgramPages	uint64	v7	Number of ClearState Program pages

1.8.3.1.2 Array Fields

Index	Name	Type	In	Notes
26	ApplicationArgs	[]byte	v2	Arguments passed to the application in the ApplicationCall transaction
28	Accounts	[]byte	v2	Accounts listed in the ApplicationCall transaction
48	Assets	uint64	v3	Foreign Assets listed in the ApplicationCall transaction
50	Applications	uint64	v3	Foreign Apps listed in the ApplicationCall transaction
58	Logs	[]byte	v5	Log messages emitted by an application call (only with `itxn` in v5). Application mode only
64	ApprovalProgramPages	[]byte	v7	Approval Program as an array of pages
66	ClearStateProgramPages	[]byte	v7	ClearState Program as an array of pages

Additional details in the opcodes document on the `txn` op.

Global Fields

Global fields are fields that are common to all the transactions in the group. In particular it includes consensus parameters.

Index	Name	Type	In	Notes
0	MinTxnFee	uint64		microalgos
1	MinBalance	uint64		microalgos
2	MaxTxnLife	uint64		rounds
3	ZeroAddress	[]byte		32 byte address of all zero bytes
4	GroupSize	uint64		Number of transactions in this atomic transaction group. At least 1
5	LogicSigVersion	uint64	v2	Maximum supported version
6	Round	uint64	v2	Current round number. Application mode only.
7	LatestTimestamp	uint64	v2	Last confirmed block UNIX timestamp. Fails if negative. Application mode only.
8	CurrentApplicationID	uint64	v2	ID of current application executing. Application mode only.
9	CreatorAddress	[]byte	v3	Address of the creator of the current application. Application mode only.
10	CurrentApplicationAddress	[]byte	v5	Address that the current application controls. Application mode only.
11	GroupID	[]byte	v5	ID of the transaction group. 32 zero bytes if the transaction is not part of a group.
12	OpcodeBudget	uint64	v6	The remaining cost that can be spent by opcodes in this program.
13	CallerApplicationID	uint64	v6	The application ID of the application that called this application. 0 if this application is at the top-level. Application mode only.
14	CallerApplicationAddress	[]byte	v6	The application address of the application that called this application. ZeroAddress if this application is at the top-level. Application mode only.

Asset Fields

Asset fields include `AssetHolding` and `AssetParam` fields that are used in the `asset_holding_get` and `asset_params_get` opcodes.

Index	Name	Type	Notes
0	AssetBalance	uint64	Amount of the asset unit held by this account
1	AssetFrozen	uint64	Is the asset frozen or not

Index	Name	Type	In	Notes
0	AssetTotal	uint64		Total number of units of this asset
1	AssetDecimals	uint64		See AssetParams.Decimals
2	AssetDefaultFrozen	uint64		Frozen by default or not
3	AssetUnitName	[]byte		Asset unit name
4	AssetName	[]byte		Asset name
5	AssetURL	[]byte		URL with additional info about the asset
6	AssetMetadataHash	[]byte		Arbitrary commitment
7	AssetManager	[]byte		Manager address
8	AssetReserve	[]byte		Reserve address
9	AssetFreeze	[]byte		Freeze address
10	AssetClawback	[]byte		Clawback address
11	AssetCreator	[]byte	v5	Creator address

App Fields

App fields used in the `app_params_get` opcode.

Index	Name	Type	Notes
0	AppApprovalProgram	[]byte	Bytecode of Approval Program
1	AppClearStateProgram	[]byte	Bytecode of Clear State Program
2	AppGlobalNumUint	uint64	Number of uint64 values allowed in Global State
3	AppGlobalNumByteSlice	uint64	Number of byte array values allowed in Global State
4	AppLocalNumUint	uint64	Number of uint64 values allowed in Local State
5	AppLocalNumByteSlice	uint64	Number of byte array values allowed in Local State
6	AppExtraProgramPages	uint64	Number of Extra Program Pages of code space
7	AppCreator	[]byte	Creator address
8	AppAddress	[]byte	Address for which this application has authority

Account Fields

Account fields used in the `acct_params_get` opcode.

Index	Name	Type	In	Notes
0	AcctBalance	uint64		Account balance in microalgos
1	AcctMinBalance	uint64		Minimum required balance for account, in microalgos
2	AcctAuthAddr	[]byte		Address the account is rekeyed to.
3	AcctTotalNumUint	uint64	v8	The total number of uint64 values allocated by this account in Global and Local States.
4	AcctTotalNumByteSlice	uint64	v8	The total number of byte array values allocated by this account in Global and Local States.
5	AcctTotalExtraAppPages	uint64	v8	The number of extra app code pages used by this account.
6	AcctTotalAppsCreated	uint64	v8	The number of existing apps created by this account.
7	AcctTotalAppsOptedIn	uint64	v8	The number of apps this account is opted into.
8	AcctTotalAssetsCreated	uint64	v8	The number of existing ASAs created by this account.
9	AcctTotalAssets	uint64	v8	The numbers of ASAs held by this account (including ASAs this account created).
10	AcctTotalBoxes	uint64	v8	The number of existing boxes created by this account's app.
11	AcctTotalBoxBytes	uint64	v8	The total number of bytes used by this account's app's box keys and values.

1.8.4 Flow Control

Opcode	Description
`err`	Fail immediately.
`bnz target`	branch to TARGET if value A is not zero
`bz target`	branch to TARGET if value A is zero
`b target`	branch unconditionally to TARGET
`return`	use A as success value; end
`pop`	discard A
`popn n`	remove N values from the top of the stack
`dup`	duplicate A

Opcode	Description
`dup2`	duplicate A and B
`dupn n`	duplicate A, N times
`dig n`	Nth value from the top of the stack. dig 0 is equivalent to dup
`bury n`	replace the Nth value from the top of the stack with A. bury 0 fails.
`cover n`	remove top of stack, and place it deeper in the stack such that N elements are above it. Fails if stack depth <= N.
`uncover n`	remove the value at depth N in the stack and shift above items down so the Nth deep value is on top of the stack. Fails if stack depth <= N.
`frame_dig i`	Nth (signed) value from the frame pointer.
`frame_bury i`	replace the Nth (signed) value from the frame pointer in the stack with A
`swap`	swaps A and B on stack
`select`	selects one of two values based on top-of-stack: B if C != 0, else A
`assert`	immediately fail unless A is a non-zero number
`callsub target`	branch unconditionally to TARGET, saving the next instruction on the call stack
`proto a r`	Prepare top call frame for a retsub that will assume A args and R return values.
`retsub`	pop the top instruction from the call stack and branch to it
`switch target ...`	branch to the Ath label. Continue at following instruction if index A exceeds the number of labels.
`match target ...`	given match cases from A[1] to A[N], branch to the Ith label where A[I] = B. Continue to the following instruction if no matches are found.

1.8.5 State Access

Opcode	Description
`balance`	balance for account A, in microalgos. The balance is observed after the effects of previous transactions in the group, and after the fee for the current transaction is deducted. Changes caused by inner transactions are observable immediately following `itxn_submit`

Opcode	Description
`min_balance`	minimum required balance for account A, in microalgos. Required balance is affected by ASA, App, and Box usage. When creating or opting into an app, the minimum balance grows before the app code runs, therefore the increase is visible there. When deleting or closing out, the minimum balance decreases after the app executes. Changes caused by inner transactions or box usage are observable immediately following the opcode effecting the change.
`app_opted_in`	1 if account A is opted in to application B, else 0
`app_local_get`	local state of the key B in the current application in account A
`app_local_get_ex`	X is the local state of application B, key C in account A. Y is 1 if key existed, else 0
`app_global_get`	global state of the key A in the current application
`app_global_get_ex`	X is the global state of application A, key B. Y is 1 if key existed, else 0
`app_local_put`	write C to key B in account A's local state of the current application
`app_global_put`	write B to key A in the global state of the current application
`app_local_del`	delete key B from account A's local state of the current application
`app_global_del`	delete key A from the global state of the current application
`asset_holding_get f`	X is field F from account A's holding of asset B. Y is 1 if A is opted into B, else 0
`asset_params_get f`	X is field F from asset A. Y is 1 if A exists, else 0
`app_params_get f`	X is field F from app A. Y is 1 if A exists, else 0
`acct_params_get f`	X is field F from account A. Y is 1 if A owns positive algos, else 0
`log`	write A to log state of the current application
`block f`	field F of block A. Fail unless A falls between txn.LastValid-1002 and txn.FirstValid (exclusive)

1.8.6 Box Access

All box related opcodes fail immediately if used in a ClearStateProgram. This behavior is meant to discourage Smart Contract authors from depending upon the availability of boxes in a ClearState transaction, as accounts using ClearState are under no requirement to furnish appropriate Box References. Authors would do well to keep the same issue in mind with respect to the availability of Accounts, Assets, and Apps though State Access opcodes *are* allowed in ClearState programs because the current application and sender account are sure to be *available*.

Opcode	Description
`box_create`	create a box named A, of length B. Fail if A is empty or B exceeds 32,768. Returns 0 if A already existed, else 1
`box_extract`	read C bytes from box A, starting at offset B. Fail if A does not exist, or the byte range is outside A's size.
`box_replace`	write byte-array C into box A, starting at offset B. Fail if A does not exist, or the byte range is outside A's size.
`box_del`	delete box named A if it exists. Return 1 if A existed, 0 otherwise
`box_len`	X is the length of box A if A exists, else 0. Y is 1 if A exists, else 0.
`box_get`	X is the contents of box A if A exists, else ''. Y is 1 if A exists, else 0.
`box_put`	replaces the contents of box A with byte-array B. Fails if A exists and len(B) != len(box A). Creates A if it does not exist

1.8.7 Inner Transactions

The following opcodes allow for "inner transactions". Inner transactions allow stateful applications to have many of the effects of a true top-level transaction, programatically. However, they are different in significant ways. The most important differences are that they are not signed, duplicates are not rejected, and they do not appear in the block in the usual away. Instead, their effects are noted in metadata associated with their top-level application call transaction. An inner transaction's `Sender` must be the SHA512_256 hash of the application ID (prefixed by "appID"), or an account that has been rekeyed to that hash.

In v5, inner transactions may perform `pay`, `axfer`, `acfg`, and `afrz` effects. After executing an inner transaction with `itxn_submit`, the effects of the transaction are visible begining with the next instruction with, for example, `balance` and `min_balance` checks. In v6, inner transactions may also perform `keyreg` and `appl` effects. Inner `appl` calls fail if they attempt to invoke a program with version less than v4, or if they attempt to opt-in to an app with a ClearState Program less than v4.

In v5, only a subset of the transaction's header fields may be set: `Type`/`TypeEnum`, `Sender`, and `Fee`. In v6, header fields `Note` and `RekeyTo` may also be set. For the specific (non-header) fields of each transaction type, any field may be set. This allows, for example, clawback transactions, asset opt-ins, and asset creates in addition to the more common uses of `axfer` and `acfg`. All fields default to the zero value, except those described under `itxn_begin`.

Fields may be set multiple times, but may not be read. The most recent setting is used when `itxn_submit` executes. For this purpose `Type` and `TypeEnum` are considered to be the same field. When using `itxn_field` to set an array field (`ApplicationArgs` `Accounts`, `Assets`, or `Applications`) each use adds an element to the end of the the array, rather than setting the entire array at once.

`itxn_field` fails immediately for unsupported fields, unsupported transaction types, or improperly typed values for a particular field. `itxn_field` makes acceptance decisions entirely from the field and value provided, never considering previously set fields. Illegal interactions between fields, such as setting fields that belong to two different transaction types, are rejected by `itxn_submit`.

Opcode	Description
`itxn_begin`	begin preparation of a new inner transaction in a new transaction group
`itxn_next`	begin preparation of a new inner transaction in the same transaction group
`itxn_field f`	set field F of the current inner transaction to A
`itxn_submit`	execute the current inner transaction group. Fail if executing this group would exceed the inner transaction limit, or if any transaction in the group fails.
`itxn f`	field F of the last inner transaction
`itxna f i`	Ith value of the array field F of the last inner transaction
`itxnas f`	Ath value of the array field F of the last inner transaction
`gitxn t f`	field F of the Tth transaction in the last inner group submitted
`gitxna t f i`	Ith value of the array field F from the Tth transaction in the last inner group submitted
`gitxnas t f`	Ath value of the array field F from the Tth transaction in the last inner group submitted

2 Assembler Syntax

The assembler parses line by line. Ops that only take stack arguments appear on a line by themselves. Immediate arguments follow the opcode on the same line, separated by whitespace.

The first line may contain a special version pragma `#pragma version X`, which directs the assembler to generate bytecode targeting a certain version. For instance, `#pragma version 2` produces bytecode targeting v2. By default, the assembler targets v1.

Subsequent lines may contain other pragma declarations (i.e., `#pragma <some-specification>`), pertaining to checks that the assembler should perform before agreeing to emit the program bytes, specific optimizations, etc. Those declarations are optional and cannot alter the semantics as described in this document.

"`//`" prefixes a line comment.

2.1 Constants and Pseudo-Ops

A few pseudo-ops simplify writing code. `int` and `byte` and `addr` and `method` followed by a constant record the constant to a `intcblock` or `bytecblock` at the beginning of code and insert an `intc` or `bytec` reference where the instruction appears to load that value. `addr` parses an Algorand account address base32 and converts it to a regular bytes constant. `method` is passed a method signature and takes the first four bytes of the hash to convert it to the standard method selector defined in ARC4

`byte` constants are:

```
byte base64 AAAA...
byte b64 AAAA...
byte base64(AAAA...)
byte b64(AAAA...)
byte base32 AAAA...
byte b32 AAAA...
byte base32(AAAA...)
byte b32(AAAA...)
byte 0x0123456789abcdef...
byte "\x01\x02"
byte "string literal"
```

`int` constants may be `0x` prefixed for hex, `0o` or `0` prefixed for octal, `0b` for binary, or decimal numbers.

`intcblock` may be explicitly assembled. It will conflict with the assembler gathering `int` pseudo-ops into a `intcblock` program prefix, but may be used if code only has explicit `intc` references. `intcblock` should be followed by space separated int constants all on one line.

`bytecblock` may be explicitly assembled. It will conflict with the assembler if there are any `byte` pseudo-ops but may be used if only explicit `bytec` references are used. `bytecblock` should be followed with byte constants all on one line, either 'encoding value' pairs (`b64 AAA...`) or 0x prefix or function-style values (`base64(...)`) or string literal values.

2.2 Labels and Branches

A label is defined by any string not some other opcode or keyword and ending in ':'. A label can be an argument (without the trailing ':') to a branching instruction.

Example:

```
int 1
bnz safe
err
safe:
pop
```

3 Encoding and Versioning

A compiled program starts with a varuint declaring the version of the compiled code. Any addition, removal, or change of opcode behavior increments the version. For the most part opcode behavior should not change, addition will be infrequent (not likely more often than every three months and less often as the language matures), and removal should be very rare.

For version 1, subsequent bytes after the varuint are program opcode bytes. Future versions could put other metadata following the version identifier.

It is important to prevent newly-introduced transaction types and fields from breaking assumptions made by programs written before they existed. If one of the transactions in a group will execute a program whose version predates a transaction type or field that can violate expectations, that transaction type or field must not be used anywhere in the transaction group.

Concretely, the above requirement is translated as follows: A v1 program included in a transaction group that includes a ApplicationCall transaction or a non-zero RekeyTo field will fail regardless of the program itself.

This requirement is enforced as follows:

- For every transaction, compute the earliest version that supports all the fields and values in this transaction.
- Compute the largest version number across all the transactions in a group (of size 1 or more), call it `maxVerNo`. If any transaction in this group has a program with a version smaller than `maxVerNo`, then that program will fail.

In addition, applications must be v4 or greater to be called in an inner transaction.

3.1 Varuint

A 'proto-buf style variable length unsigned int' is encoded with 7 data bits per byte and the high bit is 1 if there is a following byte and 0 for the last byte. The lowest order 7 bits are in the first byte, followed by successively higher groups of 7 bits.

4 What AVM Programs Cannot Do

Design and implementation limitations to be aware of with various versions.

- Stateless programs cannot lookup balances of Algos or other assets. (Standard transaction accounting will apply after the Smart Signature has authorized a transaction. A transaction could still be invalid by other accounting rules just as a standard signed transaction could be invalid. e.g. I can't give away money I don't have.)
- Programs cannot access information in previous blocks. Programs cannot access information in other transactions in the current block, unless they are a part of the same atomic transaction group.
- Smart Signatures cannot know exactly what round the current transaction will commit in (but it is somewhere in FirstValid through LastValid).
- Programs cannot know exactly what time its transaction is committed.
- Programs cannot loop prior to v4. In v3 and prior, the branch instructions `bnz` "branch if not zero", `bz` "branch if zero" and `b` "branch" can only branch forward.
- Until v4, the AVM had no notion of subroutines (and therefore no recursion). As of v4, use `callsub` and `retsub`.
- Programs cannot make indirect jumps. `b`, `bz`, `bnz`, and `callsub` jump to an immediately specified address, and `retsub` jumps to the address currently on the top of the call stack, which is manipulated only by previous calls to `callsub` and `retsub`.