```
BIP: 174
Layer: Applications
Title: Partially Signed Bitcoin Transaction Format
Author: Andrew Chow <achow101@gmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0174
Status: Final
Type: Standards Track
Created: 2017-07-12
License: BSD-2-Clause
```

Introduction

Abstract

This document proposes a binary transaction format which contains the information necessary for a signer to produce signatures for the transaction and holds the signatures for an input while the input does not have a complete set of signatures. The signer can be offline as all necessary information will be provided in the transaction.

The generic format is described here in addition to the specification for version 0 of this format.

Copyright

Motivation

Creating unsigned or partially signed transactions to be passed around to multiple signers is currently implementation dependent, making it hard for people who use different wallet software from being able to easily do so. One of the goals of this document is to create a standard and extensible format that can be used between clients to allow people to pass around the same transaction to sign and combine their signatures. The format is also designed to be easily extended for future use which is harder to do with existing transaction formats.

Signing transactions also requires users to have access to the UTXOs being spent. This transaction format will allow offline signers such as air-gapped wallets and hardware wallets to be able to sign transactions without needing direct access to the UTXO set and without risk of being defrauded.

Specification

The Partially Signed Bitcoin Transaction (PSBT) format consists of key-value maps. Each map consists of a sequence of key-value records, terminated by a

`0x00` byte [1].

```
 :=   * *
:= 0x70 0x73 0x62 0x74 0xFF
:= * 0x00
:= * 0x00
:= * 0x00
:=
:=
:=
```

Where:

A compact size unsigned integer representing the type. This compact size unsigned integer must be minimally encoded, i.e. if the value can be represented using one byte, it must be represented as one byte. There can be multiple entries with the same within a specific , but the must be unique.
The compact size unsigned integer containing the combined length of and
The compact size unsigned integer containing the length of .
Magic bytes which are ASCII for psbt **Why use 4 bytes for psbt?** The

transaction format needed to start with a 5 byte header which uniquely identifies it. The first bytes were chosen to be the ASCII for psbt because that stands for Partially Signed Bitcoin Transaction.

followed by a separator of `0xFF`[2]. This integer must be serialized in most significant byte order.

The currently defined global types are as follows:

Name			Description
Unsigned Transaction	`PSBT_GLOBAL_UNSIGNED_TX = 0x00`	None	No key data
Extended Public Key	`PSBT_GLOBAL_XPUB = 0x01`		The 78 byte serialized e
Transaction Version	`PSBT_GLOBAL_TX_VERSION = 0x02`	None	No key data
Fallback Locktime	`PSBT_GLOBAL_FALLBACK_LOCKTIME = 0x03`	None	No key data
Input Count	`PSBT_GLOBAL_INPUT_COUNT = 0x04`	None	No key data
Output Count	`PSBT_GLOBAL_OUTPUT_COUNT = 0x05`	None	No key data
Transaction Modifiable Flags	`PSBT_GLOBAL_TX_MODIFIABLE = 0x06`	None	No key data
PSBT Version Number	`PSBT_GLOBAL_VERSION = 0xFB`	None	No key data

[1] **Why is the separator here `0x00` instead of `0xff`?** The separator here is used to distinguish between each chunk of data. A separator of 0x00 would mean that the unserializer can read it as a key length of 0, which would never occur with actual keys. It can thus be used as a separator and allow for easier unserializer implementation.

[2] **Why Use a separator after the magic bytes?** The separator is part of the 5 byte header for PSBT. This byte is a separator of `0xff` because this will cause any non-PSBT unserializer to fail to properly unserialize the PSBT as a normal transaction. Likewise, since the 5 byte header is fixed, no transaction in the non-PSBT format will be able to be unserialized by a PSBT unserializer.

Name		Description
Proprietary Use Type	PSBT_GLOBAL_PROPRIETARY = 0xFC	Compact size unsigned i

The currently defined per-input types are defined as follows:

Name		
Non-Witness UTXO	PSBT_IN_NON_WITNESS_UTXO = 0x00	None
Witness UTXO	PSBT_IN_WITNESS_UTXO = 0x01	None
Partial Signature	PSBT_IN_PARTIAL_SIG = 0x02	
Sighash Type	PSBT_IN_SIGHASH_TYPE = 0x03	None
Redeem Script	PSBT_IN_REDEEM_SCRIPT = 0x04	None
Witness Script	PSBT_IN_WITNESS_SCRIPT = 0x05	None
BIP 32 Derivation Path	PSBT_IN_BIP32_DERIVATION = 0x06	
Finalized scriptSig	PSBT_IN_FINAL_SCRIPTSIG = 0x07	None
Finalized scriptWitness	PSBT_IN_FINAL_SCRIPTWITNESS = 0x08	None
Proof-of-reserves commitment	PSBT_IN_POR_COMMITMENT = 0x09	None
RIPEMD160 preimage	PSBT_IN_RIPEMD160 = 0x0a	`<20-byte hash>`
SHA256 preimage	PSBT_IN_SHA256 = 0x0b	`<32-byte hash>`
HASH160 preimage	PSBT_IN_HASH160 = 0x0c	`<20-byte hash>`
HASH256 preimage	PSBT_IN_HASH256 = 0x0d	`<32-byte hash>`
Previous TXID	PSBT_IN_PREVIOUS_TXID = 0x0e	None
Spent Output Index	PSBT_IN_OUTPUT_INDEX = 0x0f	None
Sequence Number	PSBT_IN_SEQUENCE = 0x10	None
Required Time-based Locktime	PSBT_IN_REQUIRED_TIME_LOCKTIME = 0x11	None
Required Height-based Locktime	PSBT_IN_REQUIRED_HEIGHT_LOCKTIME = 0x12	None
Taproot Key Spend Signature	PSBT_IN_TAP_KEY_SIG = 0x13	None
Taproot Script Spend Signature	PSBT_IN_TAP_SCRIPT_SIG = 0x14	`<32 byte xonlypub`
Taproot Leaf Script	PSBT_IN_TAP_LEAF_SCRIPT = 0x15	
Taproot Key BIP 32 Derivation Path	PSBT_IN_TAP_BIP32_DERIVATION = 0x16	`<32 byte xonlypub`
Taproot Internal Key	PSBT_IN_TAP_INTERNAL_KEY = 0x17	None
Taproot Merkle Root	PSBT_IN_TAP_MERKLE_ROOT = 0x18	None
Proprietary Use Type	PSBT_IN_PROPRIETARY = 0xFC	

The currently defined per-output [4] types are defined as follows:

[3]**Why can both UTXO types be provided?** Many wallets began requiring the full previous transaction (i.e. `PSBT_IN_NON_WITNESS_UTXO`) for segwit inputs when PSBT was already in use. In order to be compatible with software which were expecting `PSBT_IN_WITNESS_UTXO`, both UTXO types must be allowed.

[4]**Why do we need per-output data?** Per-output data allows signers to verify that the outputs are going to the intended recipient. The output data can also be use by signers to determine which outputs are change outputs and verify that the change is returning to the correct place.

Name		
Redeem Script	`PSBT_OUT_REDEEM_SCRIPT = 0x00`	None
Witness Script	`PSBT_OUT_WITNESS_SCRIPT = 0x01`	None
BIP 32 Derivation Path	`PSBT_OUT_BIP32_DERIVATION = 0x02`	
Output Amount	`PSBT_OUT_AMOUNT = 0x03`	None
Output Script	`PSBT_OUT_SCRIPT = 0x04`	None
Taproot Internal Key	`PSBT_OUT_TAP_INTERNAL_KEY = 0x05`	None
Taproot Tree	`PSBT_OUT_TAP_TREE = 0x06`	None
Taproot Key BIP 32 Derivation Path	`PSBT_OUT_TAP_BIP32_DERIVATION = 0x07`	`<32 byte xonlypubkey`
Proprietary Use Type	`PSBT_OUT_PROPRIETARY = 0xFC`	

Types can be skipped when they are unnecessary. For example, if an input is a witness input, then it should not have a Non-Witness UTXO key-value pair.

If the signer encounters key-value pairs that it does not understand, it must pass those key-value pairs through when re-serializing the transaction.

All keys must have the data that they specify. If any key or value does not match the specified format for that type, the PSBT must be considered invalid. For example, any key that has no data except for the type specifier must only have the type specifier in the key.

Handling Duplicated Keys

Keys within each scope should never be duplicated; all keys in the format are unique. PSBTs containing duplicate keys are invalid. However implementors will still need to handle events where keys are duplicated when combining transactions with duplicated fields. In this event, the software may choose whichever value it wishes.[5]

Proprietary Use Type

For all global, per-input, and per-output maps, the type `0xFC` is reserved for proprietary use. The proprietary use type requires keys that follow the type with a compact size unsigned integer representing the length of the string identifer, followed by the string identifier, then a subtype, and finally any key data.

The identifier can be any variable length string that software can use to identify whether the particular data in the proprietary type can be used by it. It can also be the empty string although this is not recommended.

[5] **Why can the values be arbitrarily chosen?** When there are duplicated keys, the values that can be chosen will either be valid or invalid. If the values are invalid, a signer would simply produce an invalid signature and the final transaction itself would be invalid. If the values are valid, then it does not matter which is chosen as either way the transaction is still valid.

The subtype is defined by the proprietary type user and can mean whatever they want it to mean. The subtype must also be a compact size unsigned integer in the same form as the normal types. The key data and value data are defined by the proprietary type user.

The proprietary use type is for private use by individuals and organizations who wish to use PSBT in their processes. It is useful when there are additional data that they need attached to a PSBT but such data are not useful or available for the general public. The proprietary use type is not to be used by any public specification and there is no expectation that any publicly available software be able to understand any specific meanings of it and the subtypes. This type must be used for internal processes only.

Version 0

Partially Signed Bitcoin Transactions version 0 is the first version of the PSBT format. Version 0 PSBTs must either omit PSBT_GLOBAL_VERSION or include it and set it to 0. Version 0 PSBTs must include PSBT_GLOBAL_UNSIGNED_TX, if omitted, the PSBT is invalid.

Roles

Using the transaction format involves many different roles. Multiple roles can be handled by a single entity, but each role is specialized in what it should be capable of doing.

Creator

The Creator creates a new PSBT. It must create an unsigned transaction and place it in the PSBT. The Creator must create empty input and output fields.

Updater

The Updater must only accept a PSBT. The Updater adds information to the PSBT that it has access to. If it has the UTXO for an input, it should add it to the PSBT. The Updater should also add redeemScripts, witnessScripts, and BIP 32 derivation paths to the input and output data if it knows them.

A single entity is likely to be both a Creator and Updater.

Signer

The Signer must only accept a PSBT. The Signer must only use the UTXOs provided in the PSBT to produce signatures for inputs. Before signing a non-witness input, the Signer must verify that the TXID of the non-witness UTXO matches the TXID specified in the unsigned transaction. Before signing a witness input, the Signer must verify that the witnessScript (if provided) matches the hash specified in the UTXO or the redeemScript, and the redeemScript (if provided) matches the hash in the UTXO. The Signer may choose to fail to sign a segwit input if a non-witness UTXO is not provided. [6] The Signer should not need any additional data sources, as all necessary information is provided in the PSBT format. The Signer must only add data to a PSBT. Any signatures created by the Signer must be added as a "Partial Signature" key-value pair for the respective input it relates to. If a Signer cannot sign a transaction, it must not add a Partial Signature.

The Signer can additionally compute the addresses and values being sent, and the transaction fee, optionally showing this data to the user as a confirmation of intent and the consequences of signing the PSBT.

Signers do not need to sign for all possible input types. For example, a signer may choose to only sign Segwit inputs.

A single entity is likely to be both a Signer and an Updater as it can update a PSBT with necessary information prior to signing it.

Data Signers Check For For a Signer to only produce valid signatures for what it expects to sign, it must check that the following conditions are true:

- If a non-witness UTXO is provided, its hash must match the hash specified in the prevout
- If a witness UTXO is provided, no non-witness signature may be created
- If a redeemScript is provided, the scriptPubKey must be for that redeemScript
- If a witnessScript is provided, the scriptPubKey or the redeemScript must be for that witnessScript
- If a sighash type is provided, the signer must check that the sighash is acceptable. If unacceptable, they must fail.
- If a sighash type is not provided, the signer should sign using SIGHASH_ALL, but may use any sighash type they wish.

Simple Signer Algorithm A simple signer can use the following algorithm to determine what and how to sign

```
sign_witness(script_code, i):
    for key, sighash_type in psbt.inputs[i].items:
        if sighash_type == None:
            sighash_type = SIGHASH_ALL
        if IsMine(key) and IsAcceptable(sighash_type):
```

[6] **Why would non-witness UTXOs be provided for segwit inputs?** The sighash algorithm for Segwit specified in BIP 143 is known to have an issue where an attacker could trick a user to sending Bitcoin to fees if they are able to convince the user to sign a malicious transaction multiple times. This is possible because the amounts in `PSBT_IN_WITNESS_UTXO` of other segwit inputs can be modified without effecting the signature for a particular input. In order to prevent this kind of attack, many wallets are requiring that the full previous transaction (i.e. `PSBT_IN_NON_WITNESS_UTXO`) be provided to ensure that the amounts of other inputs are not being tampered with.

```
            sign(witness_sighash(script_code, i, input))

sign_non_witness(script_code, i):
    for key, sighash_type in psbt.inputs[i].items:
        if sighash_type == None:
            sighash_type = SIGHASH_ALL
        if IsMine(key) and IsAcceptable(sighash_type):
            sign(non_witness_sighash(script_code, i, input))

for input, i in enumerate(psbt.inputs):
    if witness_utxo.exists:
        if redeemScript.exists:
            assert(witness_utxo.scriptPubKey == P2SH(redeemScript))
            script = redeemScript
        else:
            script = witness_utxo.scriptPubKey
        if IsP2WPKH(script):
            sign_witness(P2PKH(script[2:22]), i)
        else if IsP2WSH(script):
            assert(script == P2WSH(witnessScript))
            sign_witness(witnessScript, i)
    else if non_witness_utxo.exists:
        assert(sha256d(non_witness_utxo) == psbt.tx.input[i].prevout.hash)
        if redeemScript.exists:
            assert(non_witness_utxo.vout[psbt.tx.input[i].prevout.n].scriptPubKey == P2SH(re
            sign_non_witness(redeemScript, i)
        else:
            sign_non_witness(non_witness_utxo.vout[psbt.tx.input[i].prevout.n].scriptPubKey,
    else:
        assert False
```

Change Detection Signers may wish to display the inputs and outputs to users for extra verification. In such displays, signers may wish to identify which outputs are change outputs in order to omit them to avoid additional user confusion. In order to detect change, a signer can use the BIP 32 derivation paths provided in inputs and outputs as well as the extended public keys provided globally.

For a single key output, a signer can observe whether the master fingerprint for the public key for that output belongs to itself. If it does, it can then derive the public key at the specified derivation path and check whether that key is the one present in that output.

For outputs involving multiple keys, a signer can first examine the inputs that it is signing. It should determine the general pattern of the script and internally produce a representation of the policy that the script represents. Such a policy can include things like how many keys are present, what order they are in, how many signers are necessary, which signers are required, etc. The signer can then use the BIP 32 derivation paths for each of the pubkeys to find which global extended public key is the one that can derive that particular public key. To do so, the signer would extract the derivation path to the highest hardened index and use that to lookup the public key with that index and master fingerprint. The signer would construct this script policy with extended public keys for all of the inputs and outputs. Change outputs would then be identified as being the outputs which have the same script policy as the inputs that are being signed.

Combiner

The Combiner can accept 1 or many PSBTs. The Combiner must merge them into one PSBT (if possible), or fail. The resulting PSBT must contain all of the key-value pairs from each of the PSBTs. The Combiner must remove any duplicate key-value pairs, in accordance with the specification. It can pick arbitrarily when conflicts occur. A Combiner must not combine two different PSBTs. PSBTs can be uniquely identified by `0x00` global transaction typed key-value pair. For every type that a Combiner understands, it may refuse to combine PSBTs if it detects that there will be inconsistencies or conflicts for that type in the combined PSBT.

The Combiner does not need to know how to interpret scripts in order to combine PSBTs. It can do so without understanding scripts or the network serialization format.

In general, the result of a Combiner combining two PSBTs from independent participants A and B should be functionally equivalent to a result obtained from processing the original PSBT by A and then B in a sequence. Or, for participants performing fA(psbt) and fB(psbt): Combine(fA(psbt), fB(psbt)) == fA(fB(psbt)) == fB(fA(psbt))

Input Finalizer

The Input Finalizer must only accept a PSBT. For each input, the Input Finalizer determines if the input has enough data to pass validation. If it does, it must construct the `0x07` Finalized scriptSig and `0x08` Finalized scriptWitness and place them into the input key-value map. If scriptSig is empty for an input, `0x07` should remain unset rather than assigned an empty array. Likewise, if no scriptWitness exists for an input, `0x08` should remain unset rather than assigned an empty array. All other data except the UTXO and unknown fields in the input key-value map should be cleared from the PSBT. The UTXO should be kept to allow Transaction Extractors to verify the final network serialized transaction.

Transaction Extractor

The Transaction Extractor must only accept a PSBT. It checks whether all inputs have complete scriptSigs and scriptWitnesses by checking for the presence of `0x07` Finalized scriptSig and `0x08` Finalized scriptWitness typed records. If they do, the Transaction Extractor should construct complete scriptSigs and scriptWitnesses and encode them into network serialized transactions. Otherwise the Extractor must not modify the PSBT. The Extractor should produce a fully valid, network serialized transaction if all inputs are complete.

The Transaction Extractor does not need to know how to interpret scripts in order to extract the network serialized transaction. However it may be able to in order to validate the network serialized transaction at the same time.

A single entity is likely to be both a Transaction Extractor and an Input Finalizer.

Encoding

A PSBT can be represented in two ways: in binary (as a file) or as a Base64 string using the encoding described in RFC4648.

Binary PSBT files should use the `.psbt` file extension. A MIME type name will be added to this document once one has been registered.

Extensibility

The Partially Signed Transaction format can be extended in the future by adding new types for key-value pairs. Backwards compatibilty will still be maintained as those new types will be ignored and passed-through by signers which do not know about them.

Version Numbers

The Version number field exists only as a safeguard in the event that a backwards incompatible change is introduced to PSBT. If a parser encounters a version number it does not recognize, it should exit immediately as this indicates that the PSBT will contain types that it does not know about and cannot be ignored. Current PSBTs are Version 0. Any PSBT that does not have the version field is version 0. It is not expected that any backwards incompatible change will be introduced to PSBT, so it is not expected that the version field will ever actually be seen.

Updaters and combiners that need to add a version number to a PSBT should use the highest version number required. For example, if a combiner sees two PSBTs for the same transaction, one with version 0, and the other with version 1, then it should combine them and produce a PSBT with version 1. If an updater is updating a PSBT and needs to add a field that is only available in version 1, then it should set the PSBT version number to 1 unless a version higher than that is already specified.

Procedure For New Fields

New fields should first be proposed on the bitcoin-dev mailing list. If a field requires significant description as to its usage, it should be accompanied by a separate BIP. The field must be added to the field listing tables in the Specification section. Although some PSBT version 0 implementations encode types as uint8_t rather than compact size, it is still safe to add >0xFD fields to PSBT 0, because these old parsers ignore unknown fields, and is prefixed by its length.

Procedure For New Versions

New PSBT versions must be described in a separate BIP. The BIP may reference this BIP and any components of PSBT version 0 that are retained in the new version. Any new fields described in the new version must be added to the field listing tables in the Specification section.

Compatibility

This transaction format is designed so that it is unable to be properly unserialized by normal transaction unserializers. Likewise, a normal transaction will not be able to be unserialized by an unserializer for the PSBT format.

Examples

Manual CoinJoin Workflow

2-of-3 Multisig Workflow

Test Vectors

The following are invalid PSBTs:

- Case: Network transaction, not PSBT format
  - Bytes in Hex:
    `0200000001268171371edff285e937adeea4b37b78000c0566cbb3ad64641713ca42171bf6000000006`
  - Base64 String:
    `AgAAAAEmgXE3Ht/yhek3re6ks3t4AAwFZsuzrWRkFxPKQhcb9gAAAABqRzBEAiBwsiRRI+a/R01gxbUMBD1`
- Case: PSBT missing outputs
  - Bytes in Hex:
    `70736274ff01007502000000001268171371edff285e937adeea4b37b78000c0566cbb3ad64641713ca4`
  - Base64 String:
    `cHNidP8BAHUCAAAAASaBcTce3/KF6Tet7qSze3gADAVmy7OtZGQXE8pCFxv2AAAAAAD+////AtPf9QUAAAA`
- Case: PSBT where one input has a filled scriptSig in the unsigned tx
  - Bytes in Hex:
    `70736274ff0100fd0a010200000002ab0949a08c5af7c49b8212f417e2f15ab3f5c33dcf153821a8139`
  - Base64 String:

cHNidP8BAP0KAQIAAAACqwlJoIxa98SbghL0F+LxWrP1wz3PFTghqB0fh3pbe+QAAAAAakcwRAIgR1lmF5f

- Case: PSBT where inputs and outputs are provided but without an unsigned tx
  - Bytes in Hex:
    70736274ff000100fda5010100000000010289a3c71eab4d20e0371bbba4cc698fa295c9463afa2e397
  - Base64 String:
    cHNidP8AAQD9pQEBAAAAAAECiaPHHqtNIOA3G7ukzGmPopXJRjr6Ljl/hTPMti+VZ+UBAAAAFxYAFL4Y0V
- Case: PSBT with duplicate keys in an input
  - Bytes in Hex:
    70736274ff0100750200000001268171371edff285e937adeea4b37b78000c0566cbb3ad64641713ca4
  - Base64 String:
    cHNidP8BAHUCAAAAASaBcTce3/KF6Tet7qSze3gADAVmy7OtZGQXE8pCFxv2AAAAAAD+////AtPf9QUAAAA
- Case: PSBT with invalid global transaction typed key
  - Bytes in Hex:
    70736274ff020001550200000001279a2323a5dfb51fc45f220fa58b0fc13e1e3342792a85d7e36cd63
  - Base64 String:
    cHNidP8CAAFVAgAAAAEnmiMjpd+1H8RfIg+liw/BPh4zQnkqhdfjbNYzO1y8OQAAAAAA/////wGgWuoLAAA
- Case: PSBT with invalid input witness utxo typed key
  - Bytes in Hex:
    70736274ff0100550200000001279a2323a5dfb51fc45f220fa58b0fc13e1e3342792a85d7e36cd6333
  - Base64 String:
    cHNidP8BAFUCAAAAASeaIyOl37UfxF8iD6WLD8E+HjNCeSqF1+Ns1jM7XLw5AAAAAAD/////AaBa6gsAAAA
- Case: PSBT with invalid pubkey length for input partial signature typed key
  - Bytes in Hex:
    70736274ff0100550200000001279a2323a5dfb51fc45f220fa58b0fc13e1e3342792a85d7e36cd6333
  - Base64 String:
    cHNidP8BAFUCAAAAASeaIyOl37UfxF8iD6WLD8E+HjNCeSqF1+Ns1jM7XLw5AAAAAAD/////AaBa6gsAAAA
- Case: PSBT with invalid redeemscript typed key
  - Bytes in Hex:
    70736274ff0100550200000001279a2323a5dfb51fc45f220fa58b0fc13e1e3342792a85d7e36cd6333
  - Base64 String:
    cHNidP8BAFUCAAAAASeaIyOl37UfxF8iD6WLD8E+HjNCeSqF1+Ns1jM7XLw5AAAAAAD/////AaBa6gsAAAA
- Case: PSBT with invalid witnessscript typed key
  - Bytes in Hex:
    70736274ff0100550200000001279a2323a5dfb51fc45f220fa58b0fc13e1e3342792a85d7e36cd6333
  - Base64 String:
    cHNidP8BAFUCAAAAASeaIyOl37UfxF8iD6WLD8E+HjNCeSqF1+Ns1jM7XLw5AAAAAAD/////AaBa6gsAAAA
- Case: PSBT with invalid pubkey in input BIP 32 derivation paths typed key
  - Bytes in Hex:
    70736274ff0100550200000001279a2323a5dfb51fc45f220fa58b0fc13e1e3342792a85d7e36cd6333

  - Base64 String:
    `cHNidP8BAFUCAAAAASeaIyOl37UfxF8iD6WLD8E+HjNCeSqF1+Ns1jM7XLw5AAAAAAD/////AaBa6gsAAAA`

- Case: PSBT with invalid non-witness utxo typed key
  - Bytes in Hex:
    `70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb`
  - Base64 String:
    `cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQp`

- Case: PSBT with invalid final scriptsig typed key
  - Bytes in Hex:
    `70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb`
  - Base64 String:
    `cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQp`

- Case: PSBT with invalid final script witness typed key
  - Bytes in Hex:
    `70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb`
  - Base64 String:
    `cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQp`

- Case: PSBT with invalid pubkey in output BIP 32 derivation paths typed key
  - Bytes in Hex:
    `70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb`
  - Base64 String:
    `cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQp`

- Case: PSBT with invalid input sighash type typed key
  - Bytes in Hex:
    `70736274ff0100730200000001301ae986e516a1ec8ac5b4bc6573d32f83b465e23ad76167d68b38e73`
  - Base64 String:
    `cHNidP8BAHMCAAAAATAa6YblFqHsisW0vGVz0y+DtGXiOtdhZ9aLOOcwtNvbAAAAAAD/////AnR7AQAAAAA`

- Case: PSBT with invalid output redeemScript typed key
  - Bytes in Hex:
    `70736274ff0100730200000001301ae986e516a1ec8ac5b4bc6573d32f83b465e23ad76167d68b38e73`
  - Base64 String:
    `cHNidP8BAHMCAAAAATAa6YblFqHsisW0vGVz0y+DtGXiOtdhZ9aLOOcwtNvbAAAAAAD/////AnR7AQAAAAA`

- Case: PSBT with invalid output witnessScript typed key
  - Bytes in Hex:
    `70736274ff0100730200000001301ae986e516a1ec8ac5b4bc6573d32f83b465e23ad76167d68b38e73`
  - Base64 String:
    `cHNidP8BAHMCAAAAATAa6YblFqHsisW0vGVz0y+DtGXiOtdhZ9aLOOcwtNvbAAAAAAD/////AnR7AQAAAAA`

- Case: PSBT with unsigned tx serialized with witness serialization format
  - Bytes in Hex:
    `70736274ff01007802000000000101268171371edff285e937adeea4b37b78000c0566cbb3ad6464171`
  - Base64 String:

`cHNidP8BAHgCAAAAAAEBJoFxNx7f8oXpN63upLN7eAAMBWbLs61kZBcTykIXG/YAAAAAAP7///8C09/1BQA`

The following are valid PSBTs:

- Case: PSBT with one P2PKH input. Outputs are empty
  - Bytes in Hex:
    `70736274ff0100750200000001268171371edff285e937adeea4b37b78000c0566cbb3ad64641713ca4`
  - Base64 String:
    `cHNidP8BAHUCAAAAASaBcTce3/KF6Tet7qSze3gADAVmy7OtZGQXE8pCFxv2AAAAAAD+////AtPf9QUAAAA`
- Case: PSBT with one P2PKH input and one P2SH-P2WPKH input. First input is signed and finalized. Outputs are empty
  - Bytes in Hex:
    `70736274ff0100a00200000002ab0949a08c5af7c49b8212f417e2f15ab3f5c33dcf153821a8139f877`
  - Base64 String:
    `cHNidP8BAKACAAAAAqsJSaCMWvfEm4IS9Bfi8Vqz9cM9zxU4IagTn4d6W3vkAAAAAAD+////qwlJoIxa98S`
- Case: PSBT with one P2PKH input which has a non-final scriptSig and has a sighash type specified. Outputs are empty
  - Bytes in Hex:
    `70736274ff0100750200000001268171371edff285e937adeea4b37b78000c0566cbb3ad64641713ca4`
  - Base64 String:
    `cHNidP8BAHUCAAAAASaBcTce3/KF6Tet7qSze3gADAVmy7OtZGQXE8pCFxv2AAAAAAD+////AtPf9QUAAAA`
- Case: PSBT with one P2PKH input and one P2SH-P2WPKH input both with non-final scriptSigs. P2SH-P2WPKH input's redeemScript is available. Outputs filled.
  - Bytes in Hex:
    `70736274ff0100a00200000002ab0949a08c5af7c49b8212f417e2f15ab3f5c33dcf153821a8139f877`
  - Base64 String:
    `cHNidP8BAKACAAAAAqsJSaCMWvfEm4IS9Bfi8Vqz9cM9zxU4IagTn4d6W3vkAAAAAAD+////qwlJoIxa98S`
- Case: PSBT with one P2SH-P2WSH input of a 2-of-2 multisig, redeem-Script, witnessScript, and keypaths are available. Contains one signature.
  - Bytes in Hex:
    `70736274ff0100550200000001279a2323a5dfb51fc45f220fa58b0fc13e1e3342792a85d7e36cd6333`
  - Base64 String:
    `cHNidP8BAFUCAAAAASeaIyOl37UfxF8iD6WLD8E+HjNCeSqF1+Ns1jM7XLw5AAAAAAD/////AaBa6gsAAAA`
- Case: PSBT with one P2WSH input of a 2-of-2 multisig. witnessScript, keypaths, and global xpubs are available. Contains no signatures. Outputs filled.
  - Bytes in Hex:
    `70736274ff01005202000000019dfc6628c26c5899fe1bd3dc338665bfd55d7ada10f6220973df2d386`
  - Base64 String:
    `cHNidP8BAFICAAAAAZ38ZijCbFiZ/hvT3DOGZb/VXXraEPYiCXPfLTht7BJ2AQAAAAD/////AfA9zR0AAAA`
- Case: PSBT with unknown types in the inputs.
  - Bytes in Hex:
    `70736274ff01003f0200000001fffffffffffffffffffffffffffffffffffffffffffffffffffffffff`

  - Base64 String:
    cHNidP8BAD8CAAAAAf//AAAAAAD/////AQAAAAAAAAAAA

- Case: PSBT with ‘PSBT_GLOBAL_XPUB‘.
  - Bytes in Hex:
    70736274ff01009d0100000002710ea76ab45c5cb6438e607e59cc037626981805ae9e0dfd9089012a
  - Base64 String:
    cHNidP8BAJ0BAAAAAnEOp2q0XFy2Q45gflnMA3YmmBgFrp4N/ZCJASq7C+U1AQAAAAD/////GQmU1qizyM

- Case: PSBT with global unsigned tx that has 0 inputs and 0 outputs
  - Bytes in Hex:
    70736274ff01000a0000000000000000000000
  - Base64 String:
    cHNidP8BAAoAAAAAAAAAAAAAAA==

- Case: PSBT with 0 inputs
  - Bytes in Hex:
    70736274ff01004c020000000002d3dff505000000001976a914d0c59903c5bac2868760e90fd521a4
  - Base64 String:
    cHNidP8BAEwCAAAAAALT3/UFAAAAABl2qRTQxZkDxbrChodg6Q/VIaRmWqdlIIisAOH1BQAAAAAXqRQ1Re

Fails Signer checks

- Case: A Witness UTXO is provided for a non-witness input
  - Bytes in Hex:
    70736274ff0100a00200000002ab0949a08c5af7c49b8212f417e2f15ab3f5c33dcf153821a8139f87
  - Base64 String:
    cHNidP8BAKACAAAAAqsJSaCMWvfEm4IS9Bfi8Vqz9cM9zxU4IagTn4d6W3vkAAAAAAD+////qwlJoIxa98

- Case: redeemScript with non-witness UTXO does not match the script-
  PubKey
  - Bytes in Hex:
    70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887b
  - Base64 String:
    cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQ

- Case: redeemScript with witness UTXO does not match the scriptPubKey
  - Bytes in Hex:
    70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887b
  - Base64 String:
    cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQ

- Case: witnessScript with witness UTXO does not match the redeemScript
  - Bytes in Hex:
    70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887b
  - Base64 String:
    cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQ

The private keys in the tests below are derived from the following master private key:

- Extended Private Key:
  `tprv8ZgxMBicQKsPd9TeAdPADNnSyH9SSUUbTVeFszDE23Ki6TBB5nCefAdHkK8Fm3qMQR6sHwA56zqRmKmxnHk`
    - Seed:
      `cUkG8i1RFfWGWy5ziR11zJ5V4U4W3viSFCfyJmZnvQaUsd1xuF3T`

A creator creating a PSBT for a transaction which creates the following outputs:

- scriptPubKey: `0014d85c2b71d0060b09c9886aeb815e50991dda124d`,
  Amount: `1.49990000`
- scriptPubKey: `001400aea9a2e5f0f876a588df5546e8742d1d87008f`,
  Amount: `1.00000000`

and spends the following inputs:

- TXID: `75ddabb27b8845f5247975c8a5ba7c6f336c4570708ebe230caf6db5217ae858`,
  Index: `0`
- TXID: `1dea7cd05979072a3578cab271c02244ea8a090bbb46aa680a65ecd027048d83`,
  Index: `1`

must create this PSBT:

- Bytes in Hex:
  `70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abd`
- Base64 String:
  `cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQpoqka`

Given the above PSBT, an updater with only the following:

- Redeem Scripts:
    - `5221029583bf39ae0a609747ad199addd634fa6108559d6c5cd39b4c2183f1ab96e07f2102dab61ff49`
    - `00208c2353173743b595dfb4a07b72ba8e42e3797da74e87fe7d9d7497e3b2028903`
- Witness Scripts:
    - `522103089dc10c7ac6db54f91329af617333db388cead0c231f723379d1b99030b02dc21023add904f3`
- Previous Transactions:
    - `0200000000010158e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abdd7501000`
    - `0200000001aad73931018bd25f84ae400b68848be09db706eac2ac18298babee71ab656f8b0000000004`
- Public Keys
    - Key: `029583bf39ae0a609747ad199addd634fa6108559d6c5cd39b4c2183f1ab96e07f`,
      Derivation Path: `m/0'/0'/0'`
    - Key: `02dab61ff49a14db6a7d02b0cd1fbb78fc4b18312b5b4e54dae4dba2fbfef536d7`,
      Derivation Path: `m/0'/0'/1'`
    - Key: `03089dc10c7ac6db54f91329af617333db388cead0c231f723379d1b99030b02dc`,
      Derivation Path: `m/0'/0'/2'`
    - Key: `023add904f3d6dcf59ddb906b0dee23529b7ffb9ed50e5e86151926860221f0e73`,
      Derivation Path: `m/0'/0'/3'`
    - Key: `03a9a4c37f5996d3aa25dbac6b570af0650394492942460b354753ed9eeca58771`,
      Derivation Path: `m/0'/0'/4'`
    - Key: `027f6399757d2eff55a136ad02c684b1838b6556e5f1b6b34282a94b6b50051096`,
      Derivation Path: `m/0'/0'/5'`

Must create this PSBT:

- Bytes in Hex:
  70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abd
- Base64 String:
  cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQpoqka

An updater which adds SIGHASH_ALL to the above PSBT must create this PSBT:

- Bytes in Hex:
  70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abd
- Base64 String:
  cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQpoqka

Given the above updated PSBT, a signer that supports SIGHASH_ALL for P2PKH and P2WPKH spends and uses RFC6979 for nonce generation and has the following keys:

- cP53pDbR5WtAD8dYAW9hhTjuvvTVaEiQBdrz9XPrgLBeRFiyCbQr (m/0'/0'/0')
- cR6SXDoyfQrcp4piaiHE97Rsgta9mNhGTen9XeonVgwsh4iSgw6d (m/0'/0'/2')

must create this PSBT:

- Bytes in Hex:
  70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abd
- Base64 String:
  cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQpoqka

Given the above updated PSBT, a signer with the following keys:

- cT7J9YpCwY3AVRFSjN6ukeEeWY6mhpbJPxRaDaP5QTdygQRxP9Au (m/0'/0'/1')
- cNBc3SWUip9PPm1GjRoLEJT6T41iNzCYtD7qro84FMnM5zEqeJsE (m/0'/0'/3')

must create this PSBT:

- Bytes in Hex:
  70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abd
- Base64 String:
  cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQpoqka

Given both of the above PSBTs, a combiner must create this PSBT:

- Bytes in Hex:
  70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abd
- Base64 String:
  cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQpoqka

Given the above PSBT, an input finalizer must create this PSBT:

- Bytes in Hex:
  70736274ff01009a020000000258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abd
- Base64 String:

`cHNidP8BAJoCAAAAAljoeiG1ba8MI76OcHBFbDNvfLqlyHV5JPVFiHuyq911AAAAAAD/////g40EJ9DsZQpoqka`

Given the above PSBT, a transaction extractor must create this Bitcoin transaction:

- Bytes in Hex:
  `0200000000010258e87a21b56daf0c23be8e7070456c336f7cbaa5c8757924f545887bb2abdd7500000000d`

Given these two PSBTs with unknown key-value pairs:

- Bytes in Hex:
  `70736274ff01003f0200000001fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`
  - Base64 String:
    `cHNidP8BAD8CAAAAAf//////////////////////////////////////////AAAAAAD/////AQAAAAAAA`
- Bytes in Hex:
  `70736274ff01003f0200000001fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`
  - Base64 String:
    `cHNidP8BAD8CAAAAAf//////////////////////////////////////////AAAAAAD/////AQAAAAAAA`

A combiner which orders keys lexicographically must produce the following PSBT:

- Bytes in Hex:
  `70736274ff01003f0200000001fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`
- Base64 String:
  `cHNidP8BAD8CAAAAAf//////////////////////////////////////////AAAAAAD/////AQAAAAAAAAAAA2c`

Rationale

Reference implementation

The reference implementation of the PSBT format is available at https://github.com/achow101/bitcoin/tree/psbt.

Acknowledgements

Special thanks to Pieter Wuille for suggesting that such a transaction format should be made and for coming up with the name and abbreviation of PSBT.

Thanks to Pieter Wuille, Gregory Maxwell, Jonathan Underwood, Daniel Cousens and those who commented on the bitcoin-dev mailing list for additional comments and suggestions for improving this proposal.