BOBC Protocol – Version 0.0

<u>B</u>lochsTech <u>O</u>pen <u>B</u>itcoin <u>C</u>ard

Contents

1 Introduction

The name of the protocol, BOBC, stands for **B**lochsTech **O**pen **B**itcoin **C**ard. This does not imply ownership, but only origin. The BOBC protocol is open license, see licensing section 7.

The BOBC protocol is meant to be used by Bitcoin smartcards and terminals to establish clear and secure communication.
The key aspect of the protocol is that the terminal is assumed to be an attacker at all times and as such the protocol is designed to be secure under those conditions.

The protocol is defined as a set of APDU commands meant to be executed by an internet connected terminal on a BOBC card.

1.1 Version history

Version:	Changes:
0.0	First protocol and document version.

2 Meta protocol

2.1 Standardizing body

These standards shall be governed/released by the board of the Danish Bitcoin Foundation with the board having 7 votes and BlochsTech having one additional vote concerning the official standards. Decisions can be made with majority votes. Hence known as "the standardizing body".

BlochsTech may with a 1 year notice change the standardizing body or the standardizing body may do so immediately with a 1/3 majority vote and the approval of BlochsTech.

The standardizing body shall NOT have the power to revoke the open Apache license nor can it prevent others from modifying and releasing this document. The standardizing body exists primarily as an official outlet for standard releases – as such third parties must remove the standardizing body as the document origin if they do modify it and release it.

2.2 Versioning and roadmap

The BOBC protocol starts at version 0.0. The decimals shall be extremely minor changes and document changes. All versions of same major version shall be practically identical.

It is planned that there shall be major versions 0 to 5. The decimals shall count up as follows:

0.1, 0.2, … 0.9, 0.10, …

As much as possible the BOBC major versions shall be as backwards compatible as possible. Even version 0.0 for instance shall allow for more than 8 Bitcoin decimals.

The plan is to release a major version 1.0 after about 1 year of version 0 operation and experiences. Then another version 2.0 2-5 years after that, version 3.0 5-10 years after that and so on.

The plan is to make the major versions increasingly stable and more friendly towards future backwards compatibility and keeping them few, while also incorporating needs that cannot be foreseen at present.

2.3 General concepts

Operations shall as far as possible remain unchanged and instead new ones added for future functionality.

Future operations shall be made as optional as possible.

The protocol includes methods for establishing the state of the card in a controlled manor rather than through exception handling.

The exact exception code messages are not covered by the protocol and they may change – though the general meaning of the exception codes should remain unchanged.

Commands sending more total data than 256 bytes are changed so that they send packages instead. This is because some devices cannot handle longer commands.

The protocol does not distinguish between NFC or contact connections nor different cards or device types. If a phone wants to emulate a card or someone wants to make a Java based card this should be supported.

The protocol does assume either an NFC ISO-14443 or contact ISO-7816 connection between the devices.

The protocol also assumes that one device acts as a "terminal"/master and the other acts as the "card"/slave. Only deviation from this is that the "card" can communicate that it has its own internet connection and will refuse "terminal" data.

Card devices are expected to support the SHA-256 and ECDSA SECp256K1 algorithms, but NOT any others such as RIPEMD.

BOBC 0 is restricted to claiming TX outputs requiring a single signature from the card and expects strictly "pay-to-pub-key-hash"-outputs for input usage.
The card is expected to support creating outputs for pub-key-hash as well as script-hash addresses.

BOBC is limited to the Bitcoin network, though with a few changes or if the alternative is similar can be applied to other networks.

3 Process description – with flowcharts

Will be added at a later time. For now please refer to section 4 and the open source reference implementation of a BOBC terminal.

4 Commands

For reference while reading the following - see section 5.2 with card data types.

Inspecting the open source reference terminal program is also highly recommended. For this section classes BitcoinCard.java and CardTaskUtil.java should be most informative.

All command parameters should always be filled, if the terminal is not supposed to provide them just fill them with zeroes. Likewise the card may in some cases return nothing, but zeroes in some parameters.

All parameters work two-way, meaning that data can be put into them by both the terminal and the card. You should read the command and parameter descriptions to learn how each parameter is expected to be used.

This is the byte map template for commands, parameters make up the "data":

[2 bytes command bytes] [2 bytes: 00 00] [1 byte data length] [DATA] [1 byte response length]

4.1 Network command

Parameters

Command bytes (80 00)

Parameters:
2 bytes integer “returnValue” – is ignored by card.

Description

Gets the crypto currency network the card was made for. The Bitcoin network has the Id 0. The correct terminal behavior is to show an explanatory message and then disconnect the card if the network is unsupported.

4.2 Protocol command

Parameters

Command bytes (80 01)

Parameters:
2 bytes integer “returnValue” – fill with the preferred protocol version of the terminal.

Description

The terminal should send over its preferred protocol.
The card should either reply with the same value or the closest protocol it has. So a version 5 terminal and a version 4 card might agree to use protocol version 4 for instance.

Both card and terminal should save in their internal state the version selected.

If the card keeps sending unsupported version numbers or invalid ones then the terminal should disconnect with a reason message shown.

4.3 Addresses command

Parameters

Command bytes (80 02)

Parameters:
Variable length string “returnValue” – used by card.

Description

The terminal will call this one with no parameters, but the card will return a list of “:”-separated Bitcoin addresses (as ASCII text).

As soon as this is received by the terminal it can begin to get network data for the card on another thread.

4.4 RequestPayment command

Parameters

Command bytes (80 03)

Parameters:
2 byte integer “errorCode”.

1 byte “requiresPin” – If 1 PIN is required, else 0.
2 byte integer “AmountMantissa” – Charge amount mantissa.
1 byte “AmountExponent” – Charge amount exponent.
2 byte integer “FeeMantissa” – Fee amount mantissa.
1 byte “FeeExponent” – Fee amount exponent.
2 byte integer “TerminalMantissa” – Terminal amount mantissa.
1 byte “TerminalExponent” – Terminal amount exponent.
1 byte “Decimals” – Amount of decimals (see section 5.2 “custom float”).
1 byte “ReceiverAddressType” – Type of Bitcoin address being paid (0 or 5).
20 bytes “ReceiverAddress” – RipeMD160 hash of Bitcoin address.
1 byte “TerminalAddressType” –Type of Bitcoin address being paid (0 or 5).
20 bytes “TerminalAddress” – RipeMD160 hash of Bitcoin terminal address.
8 bytes string “ReturnValue” – Checksum message.

Description

This method should be called after calling the MaxAmount and WaitingCharge commands to establish respectively the maximum amount the card can be charged and whether the card has already been charged correctly, wrongly or not at all.

If the card has already been charged the terminal has two options. If the charged amount is higher than the new amount or the same, the new amount can be charged straight away.
If the new amount is higher then the terminal must charge the card 0 Bitcoins and give Pin code etc. to clear the old charged amount/addresses.

The method will if used correctly return no error.

The return value is the checksum from the card (ASCII string), the terminal should display this to the card holder. It will be an encrypted version of the amount charge to the card. Only the card holder will know the algorithm and can thereby check the amount charged before giving the Pin and the terminal has no chance to fake the displayed amount.

“Amount” is the charge amount that the merchant will receive. “Terminal amount” is used only if the terminal app itself requires a fee. “Fee” is the Bitcoin miner’s fee.

4.5 GivePINGetTx command

Parameters

Command bytes (80 04)

Parameters:
2 bytes integer “ErrorCode”
2 bytes integer “Pin” – Numeric value of pincode.
1 byte “EndOfTXStream” – Length of data package.
245 bytes array “TXBytes” – Data package.

Description

This method should only be called after all other methods have been handled and after the card is charged the correct amount.

The pin parameter is used to give the pin code. If it is wrong the card will lock itself for 1 minute extra. If a pin code is not required the method may be called with anything being the pin code and it will work.
If a pin code is required the card must be unlocked before this method works.

If the pin is accepted the parameter TXBytes is filled with data. For TX's longer than 245 bytes the data is split up and the EndOfTXStream signifies the end of data.
It will be zero until the last package. If it is zero the GivePinGetTX command should be called again.

When simply re-calling to get data parameters do not matter at all.

When this command is called the card will delete all unverified sources before doing anything.

4.6 GetSources command

Parameters

Command bytes (80 05)

Parameters:
2 bytes integer "ErrorCode"
1 byte "NextSourceIndex" – Source index starting at 0.
4 byte integer "OutIndex" – Format as in Bitcoin protocol.
32 bytes array "TXHash" – Format as in Bitcoin protocol.
8 bytes integer "Value" – Format as in Bitcoin protocol.
1 byte "Verified" – 1 if card has had merkle branch and block header sent to it.

Description

The terminal should sent the index of the source it is interested in – starting with 0. The Card will respond with the source and the index of the next source.

If the card returns 0 as the index of the next source then the card has no more sources.

A "Source" is the key data from a TX output required to create a new TX input. The card makes the sources by itself from the TX data the terminal is sending.

The terminal should use this command to learn what the card is missing and what it already knows.

4.7 GiveTX command

Parameters

Command bytes (80 06)

Parameters:
2 bytes integer "ErrorCode"
1 byte "Accepted" – Card sets to 1 if end reached and TX is valid.
1 byte "EndOfTXStream" – Length of data package.
246 bytes array "TXBytes" – Data package.

Description

This command is used to load the card with TX data it can use to make and sign new transactions. Data is split into packages of 246 bytes in the variable TXBytes and EndOfTXStream is set to 0. When the last package is sent set EndOfTXStream to the correct value from 1 to 246.

If the card responds to the last package with Accepted equal to 1 the terminal should proceed to use the hashes and header commands to let the card verify that the TX, just received, is in a block and is valid.

4.8 GiveHeader command

Parameters

Command bytes (80 07)

Parameters:
2 bytes integer “ErrorCode”
1 byte “Accepted” – Card sets to 1 if difficulty is high enough.
32 byte array “VerifyingTXHash” – TXHash terminal will start to verify.
80 bytes array “Data” – Bitcoin block header, format as in Bitcoin protocol.

Description

This command is used after the GiveTX command and before the GiveHash command. The card will first hash the header to check that this is a real Bitcoin block header with a high difficulty level.

Then the card will save the merkle root. Next the terminal should call the GiveHash command to verify that the TXHash is in the just verified header.

4.9 GiveHash command

Parameters

Command bytes (80 08)

Parameters:
1 byte “Accepted” – Card sets to 1 if valid hash.
1 byte “RightNode” – Sent hash is the right node in a merkle branch.
32 bytes array “Data” – Hash in a merkle branch.

Description

This command is called after the GiveHeader command and is used to verify a given TX is in fact in the blockchain – specifically the block given in the GiveHeader command.

To understand this command you should understand merkle trees and TXes in Bitcoin blocks. A merkle branch links a leaf node to the merkle tree root using only the strictly necessary hashes.

The first hash comes from the TX hash, then the neighbor of the TX hash is given in this command – using the bool “RightNode” to specify which side of the TX hash this neighbor is on.

The TX hash and the neighbor hash are concatenated and hashed into a “special hash”. This command is then called again with the neighbor of the special hash.

This continues until the merkle root is reached. The card will return accepted = 1.

4.10 DelayUnlockCard command

Parameters

Command bytes (80 09)

Parameters:
2 byte integer “returnValue” – Card sets to how many seconds the card is locked.

Description

When this command is called the card will do time wasting logic for 1 second. It will return how many more times the command must be called before the card is “unlocked”.

Only when the card is unlocked will it accept a pin code etc..

If the card is already unlocked this command will return immediately so a terminal can always call this command to check if the card is unlocked.

4.11 MaxAmount command

Parameters

Command bytes (80 0A)

Parameters:
2 byte integer “AmountMantissa” – Max amount mantissa.
1 byte “AmountExp” – Max amount exponent.

Description

When this command is called the card will return the maximum value that the card can be charged. Hence this command should be called before charging the card to avoid error codes being sent back.

This is because the max amount of cards can and should change to reflect user behavior and Bitcoin price.

If the terminal needs to charge a large amount the correct approach is to charge the card multiple times – asking the card holder for the Pin code each time.

(See section 5.2 for card data types.)

4.12 WaitingCharge command

Parameters

Command bytes (80 0B)

Parameters:
2 byte integer “AmountMantissa” – Charged amount mantissa.
1 byte “AmountExp” – Charged amount exponent.
2 byte integer “FeeMantissa” – Saved fee mantissa.
1 byte “FeeExp” – Saved fee exponent.
2 byte integer “TerminalMantissa” – Saved terminal mantissa.
1 byte “TerminalExp” – Saved terminal exponent.
1 byte “ReceiverAddressType” – Type of Bitcoin address being paid (0 or 5).
20 bytes “ReceiverAddress” – RipeMD160 hash of Bitcoin address.
1 byte “TerminalAddressType” –Type of Bitcoin address being paid (0 or 5).
20 bytes “TerminalAddress” – RipeMD160 hash of Bitcoin terminal address.
2 byte integer “CardFeeMantissa” – Card fee mantissa.
1 byte “CardFeeExp” – Card fee exponent.
1 byte “RequiresPin” – If equal to 1 pin is required.
8 byte ASCII string “VignereCode” – Check code from card.
1 byte “IsResetRequest” – If equal to 1 giving the Pin will finish resetting the saved charge.

Description

This command is called with all parameters set to 0 bytes and the card will return its current charge state. This command lets the terminal know how many bitcoins the card needs to finish a transaction and other information.

If IsResetRequest is set Pin is always required and all other values and receiving addresses can be ignored, except the VignereCode. The terminal should show the reset VignereCode/check code and get the Pin from the user to reset the card.

This command is most useful of course if connection has been temporarily lost or card user has left another merchant/terminal mid-transaction.

“Amount” is the charge amount that the merchant will receive. “Terminal amount” is used only if the terminal app itself requires a fee. “Fee” is the Bitcoin miner’s fee.

The “card fee” is the cards own fee and should only be used to calculate whether the card knows enough sources to complete a payment – if this is unexpectedly not the case due to the card fee the terminal should display the message “No funds” or similar.

4.13 DumpTXSources command

Parameters

Command bytes (80 0C)

Parameters:
<None>

Description

This command will cause the card to dump all of its known and unspent sources. Use incase card somehow has been loaded with invalid sources.

This command will NOT unload spent sources as that would permit one merchant to double spend the funds of the previous too easily.

The spent sources are automatically cleared as new transactions are being made. If a spent source is in fact not spent, simply wait and the card will clear it – usually after 10 new sources have been used.

4.14 Decimals command

Parameters

Command bytes (80 0D)

Parameters:
2 byte integer “ExpextedDecimals” – Number of decimals expected in the Bitcoin protocol.

Description

This command will return how many decimals the card can work with. For Bitcoin and BOBC 0.0 cards this is currently always 8, but perhaps one day it will be more.

This value affects the custom float type used in the BOBC protocol as the base unit used is the smallest number possible. So if Bitcoin has 12 decimals instead of 8 in the future all the float exponents should be incremented by 4 to get the same Bitcoin amount.

See section 5.2 for more.

4.15 WantData command

Parameters

Command bytes (80 0E)

Parameters:
2 byte integer “WantResponse” – 1 if BOBC card/device requires data from the terminal.

Description

Will usually be 1. If it is 0 it means that the card has its own internet connection.

Hence if it is 0 the terminal should skip all source loading procedures.

4.16 MaxSources command

Parameters

Command bytes (80 0F)

Parameters:
2 byte integer “MaxResponse” – Number of sources card has memory for.

Description

Card returns the number of sources it has set memory aside for. Attempting to load more sources than this value will prompt an errorcode.

Should be used before the GiveTX command.

4.17 ResetPinCode command

Parameters

Command bytes (80 10)

Parameters:
2 byte integer “ErrorCode” – Error code.
2 byte integer “PukCodeValue” – Numeric value of Puk code.
2 byte integer “NewPinCode” – Numeric value of new Pin code.

Description

Command lets the card user change his or hers Pin code. This should be done from time to time to prevent a large number of malicious terminals from saving and sharing Pin codes efficiently.

Card users should strive to only use this command on trusted terminals or at their most trusted merchants.

Card must be delay unlocked before the command can be called.

4.18 Debug command

Parameters

Command bytes (80 FF)

Parameters:
255 byte string “value” – Debug value. Usually ASCII string.

Description

Returns whatever the card has saved in a global EEPROM debug parameter.

Command can be used for debugging purposes. Exact behavior is not governed by BOBC protocol, but may contain a message after a failed command for instance or such.

4.19 Other commands

As described in this chapter APDU commands 80 00 (0) to 80 10 (16) and 80 FF (255) are currently used by the BOBC protocol.

If a device is a BOBC device the BOBC protocol also reserves the commands 80 11 to 80 FE (17-254) for future use.

Other device functions, whether additions to BOBC or unrelated software, should use different command Ids outside the 80 range.

5 Miscellaneous card specifications

5.1 Application Id

The "ApplicationID" of the smart card should be set to "BlochstechOpenBitcoinCard".

5.2 Card data types

Byte: This should self-explanatory. Numeric value from 0-255.

Integer: Given bytes AA BB the numeric value of AA is AA*256 and so on.

Long: Same as integer, but can be 4 bytes.

Float/Custom float type: ZeitControl cards support a type called "Single" that is a float type. Due to uncertain compatibility with this and Java/C#/other implementations as well as other BOBC cards a simplified float was made.
The mantissa is always an integer of 2 bytes and signifies the lowest value possible – that means 1/10^8 in the Bitcoin system, or "Satoshi".
If more decimals are added the basic unit will also be shifted down.
The exponent is the power of base 10 and is an unsigned byte type ranging from 0 to 255.
By grounding the type this way instead of around 0 arbitrary precision can be reached for practical purposes, 0.0000001BTC can be expressed as well as 211BTC (211.0000001 would be rounded to 211 though).
You should use the mantissa for as much precision as possible and remember to round it up at the cutoff.
You can check the reference implementations method "toCardFloatType" in the TypeConverter class to see how it can be done.
When used in card commands this type is always seen as a 2 byte integer mantissa and a 1 byte exponent.
This should be easy to understand and convert on all platforms.

String*n: A string with a specified length. Strings can be used either as pure byte data arrays or as actual ASCII strings. Read the command descriptions to know how a string is used.

String: Same as the limited string except it is variable length. This is not used in the BOBC protocol as calls will fail unexpectedly if the string is too long anyway.

5.3 Dust limit handling

The dust limit in the Bitcoin network is currently 5460 satoshis to be exact. No output can be below this amount. For now this limit will be the same in all 0.0 BOBC cards. In the future as this limit is lowered a function will be added for polling or setting the limit.

In BOBC 0.0 dust amounts are handled the following way:

Amount being paid: Card returns error 17.

Miner's fee: This is allowed to be under the dust limit because it is not a TX output as such.

Card's fee: Card saves the amount, once enough is accumulated the fees are paid in one TX. (Hence some TXs from the card will have a larger than expected fee and some no fee at all)

Terminal app fee: Amount is removed from TX and not paid. Terminal's responsibility that the fees it is charging are above dust limits.

Leftover amount: Added to amount being paid. Hence some paid amounts may be slightly larger than expected.

6 Error codes

If a vendor makes their own error codes they should use codes equal to or above 512 / 02 00. Codes below this are reserved for use in future BOBC standards.

6.1 Error table

These descriptions/messages may change slightly between vendors and should not be relied upon in logic, use state checking functions instead.

Error code:	Description:
0 / 00 00	No error.
1 / 00 01	Undefined/unknown contact manufacturer.
2 / 00 02	Wrong command order.
3 / 00 03	Data/Command/String/Index too long/out of bounds.
4 / 00 04	Wrong TX format version.
5 / 00 05	Invalid amount (BOBC supports polling the card for its max charge before charging it, larger puchases have to be split up).
6 / 00 06	Invalid address.
7 / 00 07	Not enough funds or card needs to be updated with TX sources.
8 / 00 08	Card has not been unlocked (it is instant to check by calling delayunlock, if card is already unlocked!) (No PIN amounts do not require time unlocking)
9 / 00 09	Card has no space for new sources, use or dump existing sources./Not enough storage.
10 / 00 0A	Does not contain expected data. Unspent TX out may not be owned by the card or script is not standard.
11 / 00 0B	Source is already known or spent.
12 / 00 0C	Device does not accept data, may have own connection and/or enough data. (Mostly useful for non-card BOBC implementing devices.)
13 / 00 0D	Not enough difficulty, block header sent should have a difficulty of at least 1/10.000 of cards average figure.
14 / 00 0E	Decimals value not supported (BOBC-0 cards usually expect 8).
15 / 00 0F	Card must be reset with a zero charge + pin to charge higher amounts or card is already waiting for a reset pin, before it can be charged again.
16 / 00 10	ECpDSA library error, contact manufacturer.
17 / 00 11	Charged amount is below the dust limit (5460 satoshis for all 0.0 cards).

7 Licensing

The BOBC protocol, this document and the reference prototype terminal implementation are released under the Apache free software license 2.0.

The source code of the BlochsTech BOBC card is NOT released under this license, but remains proprietary software.
However anyone may freely use the BOBC protocol and the reference *terminal* implementation under the Apache License to create their own proprietary or open card and/or terminal without infringing on any copyrights.

The BOBC logo is a trademark owned by the standardizing body and should only be used on a product if said product adheres to the BOBC protocol (any version).

Figure: The trademarked BOBC logo.