This app demonstrates how to use the NBitcoin library to create a bitcoin transaction and sign it offline. The app is written in WPF and uses the NBitcoin Nuget package.

Using the application:

1. Start the application
2. There is a test extended key already hardcoded in the app. If you have your own extended private key, you can enter it into the Ext Private Key field and click the arrow button to get the extended public key and derived addresses used in transactions. If using your own key, you will need to fill in the first three fields in the form. The receive and change address are from the same extended address so the transaction will send bitcoin to the same extended address.
3. Click the arrow button at the top of the form. This will retrieve the extended balance (first 10 addresses) for the extended public key. If the total balance is low (below 0.001 BTC), you can get more test bitcoins from a bitcoin faucet (i.e. <https://testnet-faucet.mempool.co/>). Note that test coins are worthless except for testing.
4. Enter an amount to send in the test transaction.
5. Click the arrow button next to the fee amount. This will retrieve the recommended transaction fee from the bitcoin blockchain. The sum of the send and fee amounts should be less than the total balance received in step 3.
6. Click the Create Tx button. This will retrieve information from the blockchain needed for the transaction and serialize it to to json.
7. Click the Sign Tx button this will take the json data and extended private key then use NHibernate to create a signed transaction hex. This step can be done offline (to protect the private key):
   * Copy the json data
   * Copy the extended private key
   * Start the app on an offline computer
   * Fill in the json and extended private key in the offline app
   * Click the Sign Tx button
   * Copy the signed transaction hex to the online application for broadcast
8. Click the Broadcast Tx button. If broadcast is successful, a transaction hex should be returned. You can use this hex to look up you transaction on the test blockchain: https://live.blockcypher.com/btc-testnet/  
   Remember that your transaction is on the testnet blockchain. It will not appear on the mainnet (real bitcoin) chain.

Notes concerning the application:

* The app is hardcoded to use testnet. You can switch to mainnet (real bitcoin) by using the flag at the top of MainWindow.xaml.cs.
* The app may be used by several people so you will probably want to use your own extended private key. You can create a new random key using the code in MainWindow.xaml.cs (CreateAddresses)
* The bitcoin node address and ElectrumX server address are hardcoded in the application (TxUtil.cs). If a node is down, you can find other servers here:  
  Mainnet ElectrumX & Bitcoin: <https://uasf.saltylemon.org/electrum>  
  Mainnet ElectrumX: <https://1209k.com/bitcoin-eye/ele.php>  
  Testnet ElectrumX: <https://1209k.com/bitcoin-eye/ele.php?chain=tbtc>

The main purpose of this article is to describe how to use NBitcoin to create a basic transaction and sign that transaction offline. The included project contains a WPF project for demonstrating the steps. This article will cover the main methods to create, sign, and broadcast the transaction. Note that we will be using an extended public key to fund the transaction and an extended private key to sign the transaction. For balances, we will use an ElectrumX server and for broadcasting, we will use a bitcoind server. Bitcoind is the main server application used by the bitcoin blockchain. ElectrumX is an indexer, allowing us to extract metadata from the blockchain.

The extended private key is the parent key of the private keys used to sign UTXOs (unspent transaction outputs). An extended key can have millions of child keys. It is best practice to never use the same child key twice. Private keys should be kept secret since they allow someone to spend the bitcoins in a given account.

The extended public key is the parent of the public keys used for spending and receiving bitcoin. Each public key has a single corresponding private key. Public keys are used for receiving bitcoin but a private key is required for sending bitcoin. Note that a public key can be derived from a private key, but not vis-a-versa.

Deriving child keys using NBitcoin  
This method displays all the derived keys for an extended key. If the extended key is private, this method will derive private and public keys. It will also display change and receiving addresses. Change addresses are generally used for receiving change from a transaction, but this not required.

using NBitcoin.BitcoinCore;

using NBitcoin.Protocol;

using NBitcoin;

public static string GetDerivedKeysAll(string extKey, uint cnt, bool testnet)

{

try

{

string result = "";

Network ntwk = testnet ? Network.TestNet : Network.Main;

bool IsPrv = extKey.Substring(1,3) == "prv"; //xprv...

ExtPubKey epbk;

if (IsPrv)

epbk = ExtKey.Parse(extKey).Neuter();

else

epbk = ExtPubKey.Parse(extKey);

result += "\n" + (IsPrv ? "Private" : "Public") + " key entered. Testnet=" + (ntwk == Network.TestNet) +"\n";

bool chg;

if (IsPrv) //show derived private keys

{

ExtKey k = ExtKey.Parse(extKey);

result += "\n" + "Ext Private Key: " + k.ToString(ntwk);

result += "\n" + "-- Derived Private Keys --";

chg = false;

result += "\n" + " >> Recipient Addresses";

for (uint ctr=0; ctr<cnt; ctr++)

result += "\n" + "" + ctr + ") " + k.Derive(chg ? 1u : 0u).Derive(ctr).PrivateKey.ToString(ntwk) + " : " + k.Derive(chg ? 1u : 0u).Derive(ctr).ToString(ntwk);

result += "\n" + " >> Change Addresses";

chg = true;

for (uint ctr=0; ctr<cnt; ctr++)

result += "\n" + "" + ctr + ") " + k.Derive(chg ? 1u : 0u).Derive(ctr).PrivateKey.ToString(ntwk) + " : " + k.Derive(chg ? 1u : 0u).Derive(ctr).ToString(ntwk);

result += "\n";

}

result += "\n" + "Ext Public Key: " + epbk.ToString(ntwk);

result += "\n" + "-- Derived Public Keys --";

chg = false;

result += "\n" + " >> Recipient Addresses";

for (uint ctr=0; ctr<cnt; ctr++)

result += "\n" + "" + ctr + ") " + epbk.Derive(chg ? 1u : 0u).Derive(ctr).PubKey.GetAddress(ntwk) + " : " + epbk.Derive(chg ? 1u : 0u).Derive(ctr).ToString(ntwk);

result += "\n" + " >> Change Addresses";

chg = true;

for (uint ctr=0; ctr<cnt; ctr++)

result += "\n" + "" + ctr + ") " + epbk.Derive(chg ? 1u : 0u).Derive(ctr).PubKey.GetAddress(ntwk) + " : " + epbk.Derive(chg ? 1u : 0u).Derive(ctr).ToString(ntwk);

return result;

}

catch(Exception ex)

{

return "Error: " + ex.Message;

}

}

Getting the extended balance  
To get the balance of an extended public key, we must derive all the child keys and sum the balances of the child keys. We will send an RPC call to an ElectrumX server. Here is the method to get the balance of a single child address. Note that there is a confirmed and unconfirmed balance. An unconfirmed balance indicates that the latest transaction has not yet been included in a bitcoin block. The server we use is a public ElectrumX server used for bitcoin testing (testnet). This does not require the NBitcoin library. It is included for the sake of completion.

using System.Net.Sockets;  
using System.Text;

//sum all UTXOs to get balance for single address

public static long GetBalance(string address, string server = "testnet.qtornado.com", int port = 51001)

{

//balance for single address - ElectrumX

using (var socket = new TcpClient(server, port)) //("18.221.223.44",50001))

{

String json = "{\"id\": \"1\", \"method\": \"blockchain.address.get\_balance\", \"params\": [\""+address+"\"], \"jsonrpc\" : \"1.0\"}\n";

var body = Encoding.UTF8.GetBytes(json);

using (var stream = socket.GetStream())

{

stream.Write(body, 0, body.Length);

byte[] bb=new byte[10000];

int k=stream.Read(bb,0,10000);

string resp = Encoding.UTF8.GetString(bb,0,k);

if (resp.Contains("error")) && !resp.Replace(" ","").Contains("error\":null"))

throw new Exception("ERROR:"+resp);

else

{

//{"jsonrpc": "2.0", "id": "1", "result": {"confirmed": 0, "unconfirmed": 0}}

int pos = resp.IndexOf("confirmed")+11;

int pos2 = resp.IndexOf(',', pos);

string qq = resp.Substring(pos, pos2-pos);

long q = long.Parse(qq);

return q;

}

}

}

}

Getting the estimated transaction fee  
For a transaction to be processed, a miners fee must be included in the transaction. ElectrumX provides a call to estimate the current fee based on the previous blocks. The fee is returned in satoshis, the smallest usable unit in the bitcoin system. A satoshi is 1/100000000 bitcoin. This does not require the NBitcoin library. It is included for the sake of completion.

using System.Net.Sockets;  
using System.Text;

//estimate tx fee based on previous # blocks

public static long EstimateFee(int blocks, string server = "testnet.qtornado.com", int port = 51001)

{

using (var socket = new TcpClient(server, port))

{

String json = "{\"id\": \"1\", \"method\": \"blockchain.estimatefee\", \"params\": [\"" + blocks + "\"], \"jsonrpc\" : \"1.0\"}\n";

var body = Encoding.UTF8.GetBytes(json);

using (var stream = socket.GetStream())

{

stream.Write(body, 0, body.Length);

byte[] bb = new byte[10000];

int k = stream.Read(bb, 0, 10000);

string resp = Encoding.UTF8.GetString(bb, 0, k);

if (resp.Contains("error")) && !resp.Replace(" ","").Contains("error\":null"))

throw new Exception("ERROR:" + resp);

else

{

resp = resp.Replace('}', ',');

//"{\"result\": 1e-05, \"error\": null, \"id\": \"1\"}\n"

int pos = resp.IndexOf("result") + 9; //{"jsonrpc": "2.0", "id": "1", "result": 1.015e-05}

int pos2 = resp.IndexOf(',',pos); //{"jsonrpc": "2.0", "id": "1", "result": 1.015e-05}

string qq = resp.Substring(pos, pos2 - pos); // 1.015e-05

decimal q = (decimal)double.Parse(qq);

return (long)(q \* 100000000); //satoshis

}

}

}

}

To create a new extended private key, just use this code:

ExtKey pvtKey = new ExtKey();

string extPrvKey = (pvtKey).GetWif(Network.TestNet).ToString(); //extended private key

For this article, I'll use this extended (testnet) private key:

tprv8j2PWZc5LeVdQWSA58MjB8ZBLQM3xDMFufv26kZ3wUcKeSR419MG6LJ6QQsTodNJ2fWU1NsodWsRMZ8KAPJkSbaKXK3vEdkSn2Qo1v7FnxV

From the extended private key, we can derive the extended public key:

ExtPubKey pubKey = pvtKey.Neuter();

string extPubKey = (pubKey).GetWif(Network.TestNet).ToString(); //extended public key

tpubDFiReyeKV2BJHyTwxn2KaYDHuRrz7YYAUyWoPGbMMkQiUvfpdYArGpuxaZPYCmRmNyFhg62sX6EPripSo6wG5hbGSerKTAmuN9z6c1vEbcF

From the extended keys, we can derive the private and public keys for transactions. The public key is used to identify an account. The private key is needed to spend bitcoin from that account. In each line, the first derive call specifies account type (0=receive, 1=change). The second derive call indicates the address index which can be over a million.

string pvtAddr = pvtKey.Derive(0).Derive(0).PrivateKey.ToString(Network.TestNet);

string pubAddr = pubKey.Derive(0).Derive(0).PubKey.GetAddress(Network.TestNet).ToString();

For this article, we'll use these addresses for transactions:

Private Send Key: cUxutv9UwUuRJdRtrfRwSXFCWQFNE7qX5Z41TG4DGxu6rP6XH59i  
Public Send Address: mzpzv4n9UmaAjC7gBJmqktfQQwrDRmWT5Z  
Public Receive Address: mhAmxzttQ6H91NRHZfg6iY6LgiGTRzdeGr  
Public Change Address: mx4iJ5roJwiDnpe6cE1GUSd8P4NTBhUh1f

All these addresses were derived using the code snippets shown above. For our test transactions, we'll send bitcoin to ourselves using addresses derived from the extended keys.

There are 3 main steps to a transaction:

* Creating the transaction
* Signing the transaction
* Broadcasting the signed transaction

Creating the transaction

When creating the transaction, there are several pieces of information needed:

1. The source account (public address)
2. Private key coinciding with the public source account address
3. The change address
4. The receiving address (the address receiving the funds)
5. The amount to send
6. The transaction fee

When sending bitcoin, the entire value of the source account is spent. The change address is used to receive change (the amount we keep). Several source addresses can be used in a single transaction, but for our test, we will just use a single address.

The transaction fee is not specified directly. When sending funds, we must specify two target addresses: the receiving address and the change address. We will specify the exact amount to send to each address. The remaining funds are considered the transaction fee.

In the application, the addresses are already set for testing. You can provide your own addresses using the code shown below:

//create random ext key

ExtKey pvtKey = new ExtKey();

string extPrvKey = (pvtKey).GetWif(Network.TestNet).ToString(); //extended private key, can derive all addresses\keys from this

ExtPubKey pubKey = pvtKey.Neuter(); //extended public key, can derive all public addresses from this

string extPubKey = (pubKey).GetWif(Network.TestNet).ToString(); //extended public key string

string pvtSrcKey = pvtKey.Derive(0).Derive(0).PrivateKey.ToString(Network.TestNet); //need this to spend btc

string pubSrcAddr = pubKey.Derive(0).Derive(0).PubKey.GetAddress(Network.TestNet).ToString(); //source from bitcoin fountain (or USD)

string pubRecAddr = pubKey.Derive(0).Derive(1).PubKey.GetAddress(Network.TestNet).ToString(); //use to receive funds

string pubChgAddr = pubKey.Derive(1).Derive(0).PubKey.GetAddress(Network.TestNet).ToString(); //use as change address

Using the application

Creating the transaction:

Open the app. Click the top right (>>) button. This will give you the total balance for the provided extended public address (first 11 receive and change addresses). If the balance is low (or zero) you can fund one of the addresses with a bitcoin faucet:

<https://testnet-faucet.mempool.co/>

Remember that testnet coins are worthless and are only useful for testing.

You can enter any send amount as long as one of the addresses has enough to cover the transaction and fee.

To get the fee amount, click the >> button next to Fee Amount. This will set the recommended fee amount. Higher fees will result in faster transaction processing. If the fee is set to low, the transaction may never be processed.

Click the "Create Tx" button. The application will generate json string containing the required transaction data. Note there will be entries for InputUTXOs. These are the transactions that provided the funds to our source account. These are the funds we will spend.

Signing the transaction

Once we have the transaction information, we pass the data to the NBitcoin library along with the private spend key for each address we are spending from. In the application, ensure the extended private key field is filled in and click the Sign Tx button. This produce the signed transaction hex used to broadcast the transaction on the blockchain.

Broadcasting the transaction

After using the application to sign the transaction, click the Broadcast Tx button. This will broadcast the transaction to the blockchain. This is not done through NBitcoin, but directly through a bitcoin node. You can the check the status of the transaction using a bitcoin testnet blockchain explorer:

https://live.blockcypher.com/btc-testnet/

Note that the transaction may take a minute or two to show up in the explorer. You can also use the explorer to check the current balance of your addresses.