34 Hashing and Bitcoin

Information

The term "hash" originates by analogy with its non-technical meaning, to "chop and mix". Hash functions often "chop" the input domain into many sub-domains that get "mixed" e.g. add the first three digits of the key, add the last three digits, concatenate the two resulting digit strings then map into the output range by applying modulo N.

Hashing

Purpose: To understand the use of hashing functions in cryptography and in hash tables

Chapter 33 introduced digital signatures whereby a digest of the message is signed instead of the message m. The message digest is shorter than the message m and is generated by applying a **hash function** H to the message m. The message digest $H(m)$ is typically between 128 and 512 bits, compared to thousands or millions of bits for the message m itself. Signing $H(m)$ is therefore much less computationally intensive than signing m directly because signing $H(m)$ involves working with fewer bits.

In general, a hashing or hash function H maps input data, m, of an arbitrary length in a random way to an output of fixed length. The output, a fixed-length random value, is called the hash value or just hash.

Hashing functions have many uses besides generating message digests.

For example, hash functions are also used as mapping functions in hash tables, a data structure which is considered later in this chapter.

Hash functions used in cryptography are different from hash functions used in hash tables. Cryptographic hash functions have specific security properties, notably:

> A cryptographic hash function should be a **one way function**, i.e. a function which is practically infeasible to invert, i.e. given a message m it should be relatively easy to compute its hash value $H(m)$, but given a hash value x it should not be possible to find an m such that $H(m) = x$ given current computational resources.

Hash table mapping functions are less stringent in this respect.

Both cryptographic and hash table functions should have **good collision resistance**. A collision occurs when $H(m_1) = H(m_2)$ for two different inputs m_1 and m_2 but hash table functions have ways of coping when collisions do occur.

Theoretically avoiding collisions is impossible because there are more possible input values than there are possible output values.

However, the collision-resistance requirement simple means that, although collisions exist, encountering these should be minimised in the case of hash tables or avoided altogether in the case of cryptographic hash functions.

It is also important that both types of hash function should be quick to compute.

Real hash functions are distinguished from the **ideal hash function** as follows:

The **ideal hash function** is a random mapping from all possible input values to the set of possible output values i.e. output hash values are of equal likelihood;

Real hash functions only attempt to be indistinguishable from a random mapping.

Secure hash functions

We saw in the previous chapter that ciphers protect data confidentiality (i.e. attempt to prevent data sent over a communication link from being read if intercepted).

We also saw that hash functions protect data integrity by attempting to detect when data have been modified whether that data is encrypted or not - message digest and signed message digest.

If a **hash function** is **secure**, two distinct pieces of data should always have different hashes. A file's hash (the result of applying the hash function to the file) can thus serve as an identifier. Even if a single bit is changed in the file, the hash of the file will be completely different.

Secondly, the output from a secure hash function should be unpredictable.

The console mode program shown in *Table 34.1* calls function `THashSHA2.GetHashString`, a class function, which applies the **Secure Hash Algorithm-2** (**SHA2**) to a single character to produce a 256-bit hash value output. The function `GetHashString` returns a hexadecimal string which it generates by dividing each 256-bit hash value into 64, 4-bit blocks before treating each block as a single hexadecimal digit which it then maps to its equivalent hexadecimal character digit, e.g. hexadecimal digit C is mapped to character 'c'.

256 bits map to 64, 4-bit blocks which map to 64 characters chosen from the set ['0'..'9', 'a'..'f'].

The output from this program for the characters 'a', 'b' and 'c' is shown in *Figure 34.1*. Although the bit patterns for 'a', 'b' and 'c' differ by only one or two bits ('a' is the bit pattern 01100001, 'b' is 01100010 and 'c' is 01100011) their hash values are completely different. Given only these three hashes, it is impossible to predict the SHA2 - 256 hash of 'd', etc, i.e. the hash values are *unpredictable*. The function `THashSHA2.GetHashString` returns a random string each time it receives an input.

```
Program CryptoHashOnACharUsingSHA2_256Project;
{$APPTYPE CONSOLE}
{$R *.res}
Uses
  System.SysUtils, System.Hash;
Var
  Ch : Char;
Begin
  Repeat
    Write('Input char to hash: ');
    Readln(Ch);
    Writeln('Hash value SHA2 of ', '''', Ch, '''', ' = '
            + THashSHA2.GetHashString(Ch, SHA256));
    Writeln('Hash value SHA2 of ', '''', Succ(Ch), '''', ' = '
            + THashSHA2.GetHashString(Succ(Ch), SHA256));
    Writeln('Hash value SHA2 of ', '''', Succ(Succ(Ch)), '''', ' = '
            + THashSHA2.GetHashString(Succ(Succ(Ch)), SHA256));
    Write('Another go (Y/N)? ');
    Readln(Ch);
  Until Ch In ['N', 'n'];
End.
```

Table 34.1 CryptoHashOnACharUsingSHA2_256Project.dpr

```
Input char to hash: a
Hash value SHA2 of 'a' = ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
Hash value SHA2 of 'b' = 3e23e8160039594a33894f6564e1b1348bbd7a0088d42c4acb73eeaed59c009d
Hash value SHA2 of 'c' = 2e7d2c03a9507ae265ecf5b5356885a53393a2029d241394997265a1a25aefc6
Another go (Y/N)? y
Input char to hash: a
Hash value SHA2 of 'a' = ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb
Hash value SHA2 of 'b' = 3e23e8160039594a33894f6564e1b1348bbd7a0088d42c4acb73eeaed59c009d
Hash value SHA2 of 'c' = 2e7d2c03a9507ae265ecf5b5356885a53393a2029d241394997265a1a25aefc6
```

Figure 34.1 SHA2 - 256 hash values of the characters 'a', 'b', 'c'

Preimage resistance

In practice, the security of a hash function, $H(m)$, where m is any message, is judged by whether an attacker will find m, given the generated hash value x. We call m the **preimage** in this scenario and the security property of the hash function that resists discovery of m given hash value x, **preimage resistance**.

To illustrate the difficulty that an attacker will experience, consider a hash function that outputs hash values of length 256 bits and which behaves like a truly random function. There are 2^{256} equally likely hash values. If just 1024-bit length messages are considered, there are 2^{1024} possible messages. Therefore, on average each possible 256-bit hash value will have $2^{1024}/_{2^{256}} = 2^{768}$ preimages of 1024 bits each. 2^{768} is approximately 1.6×10^{231}.
If execution time of a preimage search algorithm for a match with hash value x - *Table 34.2* - for one possible message m is, say, 5×10^{-7} seconds, then to test half the possible messages of length 1024 bits would take 4×10^{224} seconds, an enormous amount of time. And then there are other possible bit length messages.....

```
Function FindPreImage(x : THashValue) : TMessage
  Var
    m : TMessage
  Repeat
    m ← GenerateRandomMessage
    If H(m) = x
      Then Exit;
  Until  False
  Result ← m
End Function
```

Information

SHA2-256 is built into Apple Macs. Choose Terminal and then type

echo -n 'hello!' | shasum - a 256

to pass the string 'hello!' through a pipe (|) to the function shasum.

Install NotePad++ on Windows. This editor has a SHA-256 option under Tools.

Table 34.2 Pseudocode for preimage search algorithm for a secure hash function h

Preimage resistance may be divided into **first-preimage resistance** and **second-preimage resistance**.

First-preimage resistance means the degree of resistance to discovering any message that maps to a given hash value.

Second-preimage resistance means the degree of resistance for a given message m_1 of finding a second message m_2 that hashes to the same value as m_1.

First-preimage resistance

First-preimage resistance is important in the case of passwords which are hashed and saved in a database. When a user logs in, the hash of the entered password is computed and compared against the stored hash value. The intention of storing hashes of passwords instead of storing passwords directly is to keep passwords stored in a system confidential. As passwords are usually of limited length, a salt (random data) is added to the password before hashing to make it more difficult to discover the password given the hash value. Storing passwords in this way relies crucially on the hash not being reversed by an attacker, i.e. first-preimage resistance.

Second-preimage resistance

Second-preimage resistance is important in protocols which focus on message integrity. A match between the original message and its hash should be confirmation that the document has not been altered.

Second-preimage resistance is also important in a commitment scheme which is a cryptographic primitive that allows one to commit to a chosen value (or chosen statement) while keeping it hidden from others, with the ability to reveal the committed value later.

Interactions in a commitment scheme take place in two phases:

- the commit phase during which a value is chosen and specified
- the reveal phase during which the value is revealed and checked

Figure 34.2 shows an example in which Bob sends the hash value of an even or odd number chosen from some prearranged fixed but large range of integers. The chosen number encodes the setting of a single bit known only to Bob at this stage. In the commitment phase, Bob chooses an even number if the bit is to be set to one, otherwise he chooses an odd number. Alice on receipt of the hashed value makes a guess as to the setting of Bob's bit and replies by sending her guess as a single bit. Upon receiving Alice's guess, Bob enters the reveal stage by sending the original number. On receipt, Alice verifies this number by comparing the hash of this number with the hash value received during the commit stage. If Bob uses a hash function that has weak second-preimage resistance then it is possible that Bob knows of a different number with opposite parity (evenness or oddness) that produces the same hash value as the committed number. Bob then has the option to determine the outcome whether Alice guesses correctly or not by exploiting the second-preimage weakness in the hash function.

What might Bob's bit sequence represent? One case is the sequence represents the head|tail outcome of a series of coin tosses which Alice is required to guess.

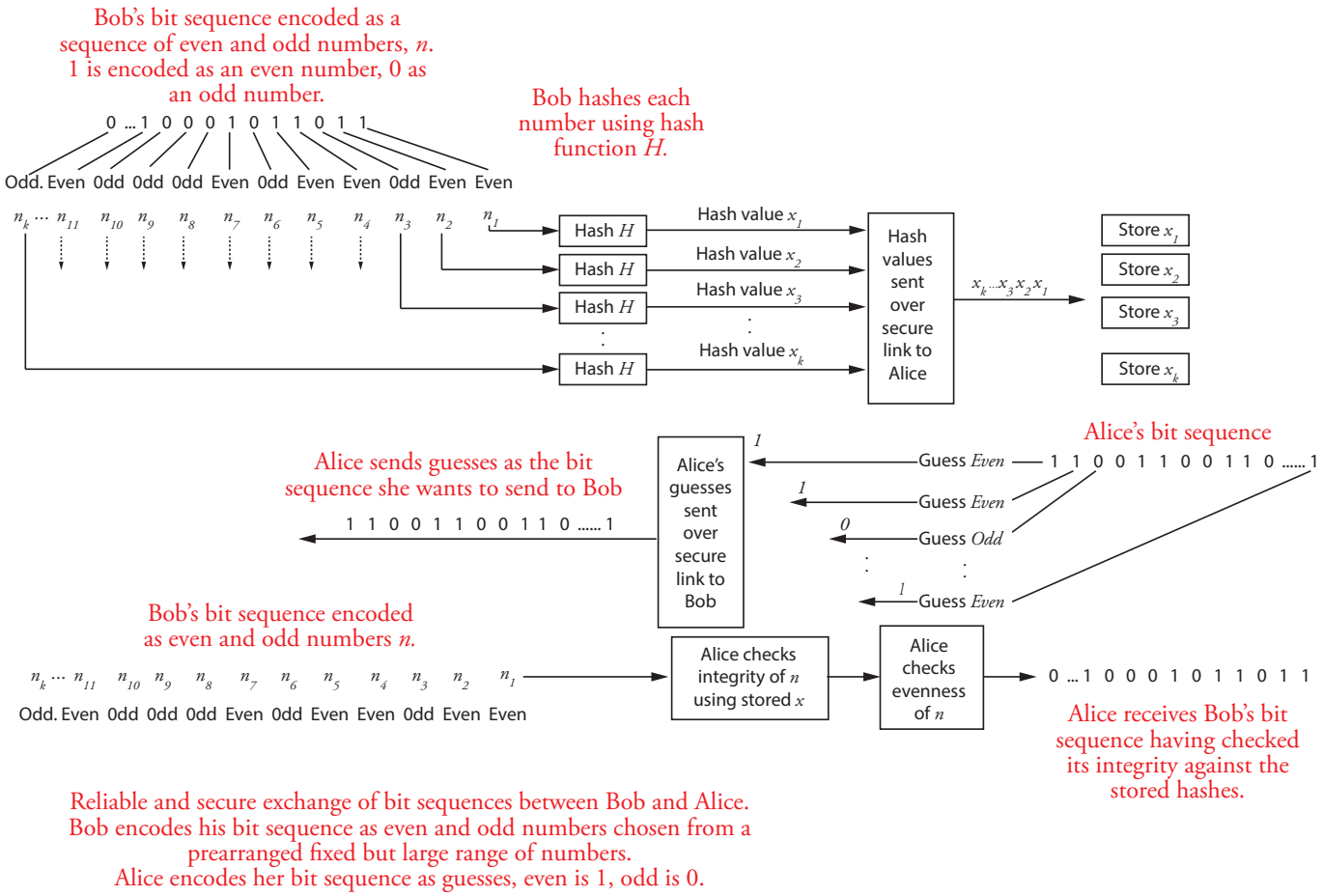


Figure 34.2 Commitment scheme

Collision resistance

Whatever hash function is used, there are always more possible messages by design than hash values so collisions must exist. This is the **pigeonhole principle** in action.

> The pigeonhole principle states that if there are n holes and m pigeons to put in these holes, and if m is greater than n, at least one hole must contain more than one pigeon.

The **collision resistance** of a hash function is a measure of how hard it is to find collisions. A collision resistant hash function is one which should make it infeasible for attackers to find two distinct messages that hash to the same value.

Collision resistance is related to second-preimage resistance. If it is possible to find second-preimages for a hash function, it is also possible to find collisions.

It is actually faster to find collisions than it is to find preimages thanks to the **birthday attack**. The birthday attack is a restatement of the **birthday problem** of calculating the probability that in a set of n randomly selected people, at least two people share the same birthday.

Let $p(n)$ be the probability that in a set of n randomly chosen people at least two people share the same birthday.

Then $1 - p(n)$ is the probability that every single one of them has distinct birthdays.

The number of ways to pick n distinct birthdays from a set of 366 days (when the order in which you pick the birthdays matters) is

$$366 \times 365 \times \ldots\ldots \times (367 - n)$$

because each successive birthday has one fewer choice of days left.

The number of possibilities for the birthdays for n people is 366^n.(although not all equally likely because of the leap year, but we will ignore this)

Therefore,

$$1 - p(n) = \frac{366 \times 365 \times \ldots\ldots \times (367 - n)}{366^n}$$

$$1 - p(n) = \frac{366!}{366^n \times (366 - n)!}$$

$$p(n) = 1 - \frac{366!}{366^n \times (366 - n)!}$$

If $n = 24$

$$366! = 9.19 \times 10^{780}$$

$$366^n = 366^{24} = 3.34 \times 10^{61}$$

$$(366 - n)! = (366 - 24)! = 342! = 5.95 \times 10^{719}$$

$$366^{24} \times 342! = 3.34 \times 10^{61} \times 5.95 \times 10^{719} = 19.87 \times 10^{780}$$

Therefore,

$$p(24) = 1 - \frac{366!}{366^n \times (366 - 24)!} = 1 - \frac{9.19 \times 10^{780}}{19.87 \times 10^{780}} = 1 - 0.46 = 0.54$$

The probability that in a set of 24 randomly chosen people at least two people share the same birthday is 54%.

If the calculation is repeated but with $n = 72$ the probability is close to 100%.

We are now ready to answer the question:

How many values does an attacker need to compute before the probability of a collision is greater than 50%?

Suppose that a hash function is chosen with a 64-bit range, i.e. its output is a 64-bit nonnegative integer less than 2^{64} (0 .. 2^{64} - 1). Just as with the birthday problem, the probability of a collision from n random samples is

$$p_M(n) = 1 - \frac{M!}{M^n \times (M - n)!}$$

where $M = 2^{64}$ the number of possible hash values.

For $n^2 << M$

$$p_M(n) \approx \frac{n^2}{2M}$$

For $p_M(n) = 0.5$

$$0.5 \approx \frac{n^2}{2M}$$

i.e. n is of the order of $\sqrt{M}$. For $M = 2^{64}$, $\sqrt{M} = 2^{32}$ which is approximately 4 x 10^9. Trialling this number of sample messages is feasible (*hash*(*sample value*)). This vulnerability necessitates the use of a larger hash range in practical applications.

If a 256-bit range is chosen, i.e. hash values of nonnegative integers less than 2^{256}, then for $n^2 << M$, n is of the order of $\sqrt{2^{256}}$, i.e. 2^{128} which is approximately 3 x 10^{38}. The number of values an attacker needs to compute before the probability of a collision is greater than 50% is now large enough to be infeasible by a brute force approach.

How can an attacker exploit a collision vulnerability in a hashing function?

> An attacker typically begins by constructing two messages with the same hash value where one message appears legitimate. For example, when an attacker, *X*, discovers that the message
>
> "I, *SomeName*, agree to pay *X* the sum of £5000.00 on 01/03/2021."
>
> has the same hash as
>
> "I, *SomeName*, agree to pay *X* the sum of £50000.00 on 09/03/2021."
>
> then *X* can try to get the victim, *SomeName*, to digitally sign the first message. The attacker *X* can then claim that *SomeName* actually signed the second message. *SomeName* signs the hash of the first and genuine message, i.e. signs the message digest, with his/her private key and the attacker *X* retrieves the hash from the digitally signed message digest by using *SomeName*'s public key. *X* now attempts to prove that the hash for the confirmed signature matches the second bogus message.

Collisions have been announced for the following hash functions

- SHA-0
- MD4
- MD5
- HAVAL-128
- RIPEMD
- SHA-1.

Hash function construction

The simplest way to hash a message is to split it into chunks and process each chunk consecutively using a similar algorithm. This is called **iterative hashing**. It comes in two main forms:

- Using a technique based on a **compression function** that transforms an input to a smaller output. This is known as the Merkle-Damgård construction after cryptographers Raplh Mekle and Ivan Damgård. MD4, MD5, SHA-1, and the SHA-2 family are examples.
- Using a technique that transforms an input, a binary string of any length, and returns a binary string with any requested length, such that any two different inputs give two different outputs, i.e. permutations. Such functions are called **sponge** functions. An example is **Keccak** which is also known as SHA-3.

Hashing file contents in Delphi

Start a new application **New|Window VCL Application - Delphi**.

Save the project as `CryptographicHashOnFileProject.dproj` and its unit as `CryptographicHashOnFileUnit.pas` in folder `CryptographicHashOnFileProject`.

Add the following components to the form:

- 3 x **TPanel**
- 6 x **TButton**
- 2 x **TEdit**
- 2 x **TLabel**
- 2 x **TMemo**
- 1 x **TOpenDialog**

Configure and **rename** these as shown in *Figure 34.3* and *Figure 34.4*.

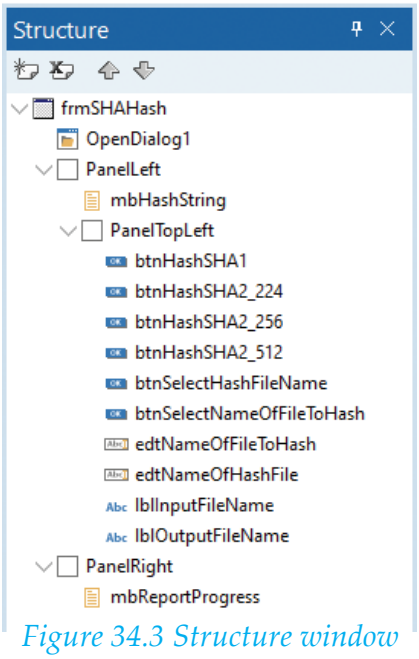


Figure 34.3 Structure window

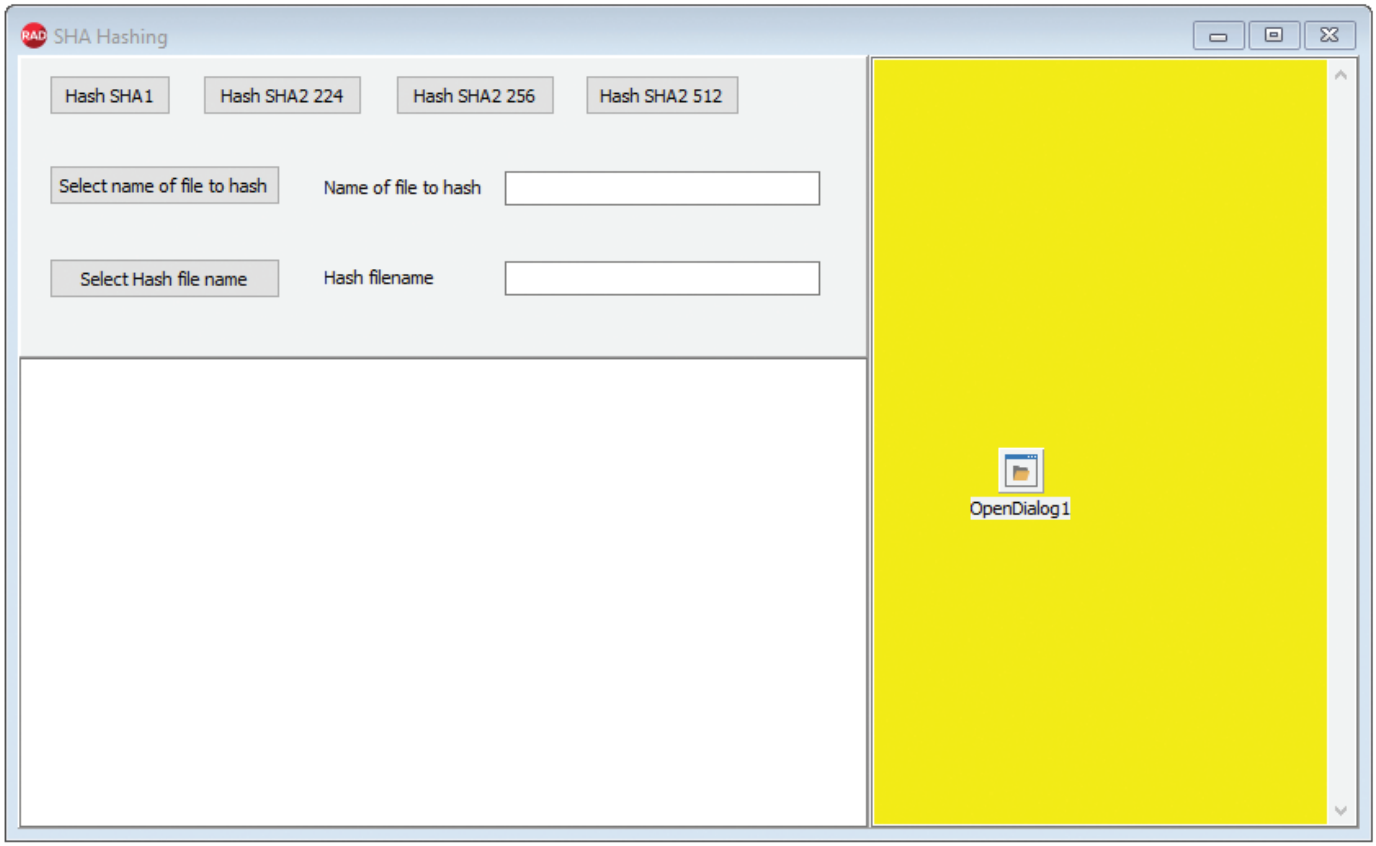


Figure 34.4 CryptographicHashOnFileProject user interface design

Double click the buttons to create event-handlers.

Add the following to the class definition as shown in *Table 34.3*.

```
Procedure SHA1HashFile(InputFileName, HashFileName : String);
Procedure SHA2_512HashFile(InputFileName, HashFileName : String);
Procedure SHA2_256HashFile(InputFileName, HashFileName : String);
Procedure SHA2_224HashFile(InputFileName, HashFileName : String);
```

Place the cursor in each of these in turn and press **Ctrl+Shift+C** to create the skeleton procedure definition in the **Implementation** section.

Add the code shown in *Table 34.4* to the body of `Procedure SHA2_256HashFile`.

Repeat this process for the other procedures except `SHA1HashFile` changing the references `SHA256` to the appropriate references for the respective procedure, e.g. SHA256 ↦ SHA512.

`SHA1HashFile`. `GetHashStringFromFile` body code is similar for all but

```
OutputStringList.Add(THashSHA1.GetHashStringFromFile(InputFileName));
```

which has only one parameter.

```
Type
  TfrmSHAHash = Class(TForm)
                  PanelTopLeft : TPanel;
                  PanelRight : TPanel;
                  mbReportProgress : TMemo;
                  btnHashSHA1 : TButton;
                  btnHashSHA2_512 : TButton;
                  edtNameOfFileToHash: TEdit;
                  lblInputFileName : TLabel;
                  edtNameOfHashFile: TEdit;
                  lblOutputFileName : TLabel;
                  btnSelectNameOfFileToHash : TButton;
                  btnSelectHashFileName : TButton;
                  OpenDialog1 : TOpenDialog;
                  PanelLeft : TPanel;
                  mbHashString : TMemo;
                  btnHashSHA2_256 : TButton;
                  btnHashSHA2_224 : TButton;
                  Procedure btnHashSHA1Click(Sender : TObject);
                  Procedure btnHashSHA2_512Click(Sender : TObject);
                  Procedure btnSelectHashFileNameClick(Sender : TObject);
                  Procedure btnSelectNameOfFileToHashClick(Sender : TObject);
                  Procedure SHA1HashFile(InputFileName, HashFileName : String);
                  Procedure SHA2_512HashFile(InputFileName, HashFileName : String);
                  Procedure SHA2_256HashFile(InputFileName, HashFileName : String);
                  Procedure SHA2_224HashFile(InputFileName, HashFileName : String);
                  Procedure btnHashSHA2_256Click(Sender : TObject);
                  Procedure btnHashSHA2_224Click(Sender : TObject);
                End;
Var
  frmSHAHash: TfrmSHAHash;
```

Table 34.3 Class definition TfrmSHAHash and variable declaration frmSHAHash

Add the code shown in *Table 34.5* to the body of event handler `btnHashSHA2_256Click`.

Repeat for the other `btnHash....Click` event handlers replacing references to SHA2 256 to the relevant hashing algorithm.

Add the code shown in *Table 34.6* to the bodies of the **OpenDialog** event handlers.

Save All (**SHIFT+CTRL+S**).

Click Run (**F9**) to build and run the application `CryptographicHashOnFileProject`.

Figure 34.5 shows `CryptographicHashOnFileProject` in execution. `Books.txt` is an 80 kB text file downloaded from http://www.gutenberg.org/files/64684/64684-0.txt.

```
Procedure TfrmSHAHash.SHA2_256HashFile(InputFileName, HashFileName : String);
    Var
      OutputStream : TFileStream;
      OutputStringList : TStringList;
    Begin
      If FileExists(InputFileName)
      Then
        Begin
          OutputStringList := TStringList.Create;
          OutputStringList.Add(THashSHA2.GetHashStringFromFile(InputFileName, SHA256));
          OutputStream := TFileStream.Create(HashFileName, fmCreate);
          OutputStringList.SaveToStream(OutputStream);
          mbHashString.Lines.Add(OutputStringList.Text);
          mbReportProgress.Lines.Add('Size of contents of hash file '
           + ExtractFileName(HashFileName) + ' produced with SHA2 256 is in bits '
           + IntToStr(4 * Length(THashSHA2.GetHashStringFromFile(InputFileName, SHA256)))
           + '.' + #10#13#10#13);
          OutputStream.Free;
          OutputStringList.Free;
        End
      Else ShowMessage('File doesn''t exist');
    End;
```

Table 34.4 Procedure that applies SHA2-256 hash function to input file

```
Procedure TfrmSHAHash.btnHashSHA2_256Click(Sender : TObject);
  Begin
    SHA2_256HashFile(edtNameOfFileToHash.Text, edtNameOfHashFile.Text);
    mbReportProgress.Lines.Add('Input file ' + ExtractFileName(edtNameOfFileToHash.Text)
      + ' hashed with SHA2 256.' + #10#13#10#13);
    mbReportProgress.Lines.Add('Output file ' + ExtractFileName(edtNameOfHashFile.Text)
      + ' produced with SHA2 256.' + #10#13#10#13);
  End;
```

Table 34.5 Event handler for btnHashSHA2_256Click

```
 Procedure TfrmSHAHash.btnSelectNameOfFileToHashClick(Sender : TObject);
    Begin
      If OpenDialog1.Execute
      Then edtNameOfFileToHash.Text := OpenDialog1.FileName;
    End;

  Procedure TfrmSHAHash.btnSelectHashFileNameClick(Sender : TObject);
    Begin
      If OpenDialog1.Execute
        Then edtNameOfHashFile.Text := OpenDialog1.FileName;
    End;
```

Table 34.6 Event handlers for the input and output file dialogues

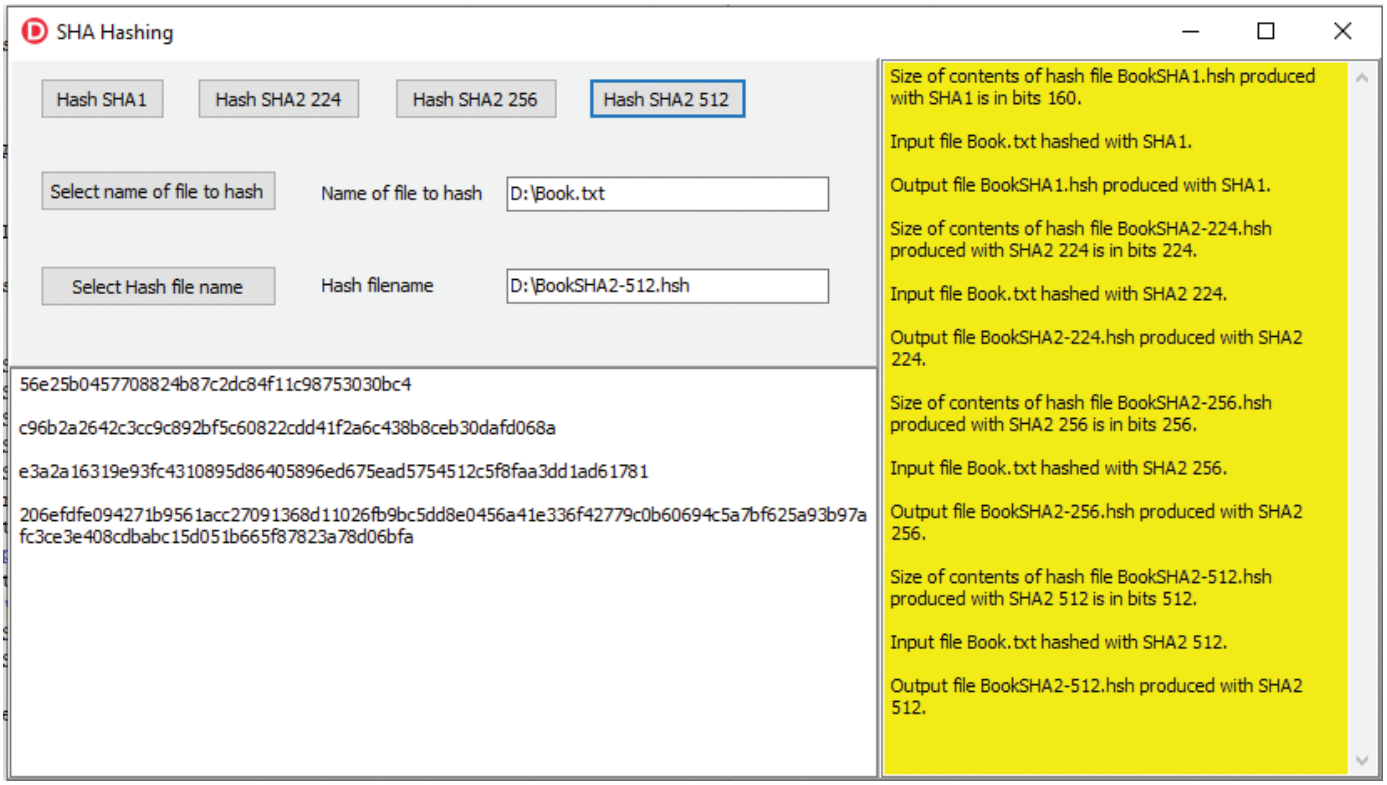


Figure 34.5 CryptographicHashOnFileProject in execution

Bitcoin® and Blockchain

The Bitcoin project was the first to bring together technologies from the 1970s, 80s and 90s to create something novel that solved important problems associated with a digital currency based on a distributed ledger system.

According to Wikipedia[1]:

> *A ledger is a book or collection of accounts in which account transactions are recorded. Each account has an opening or carry-forward balance, and would record transactions as either a debit or credit in separate columns, and the ending or closing balance.*

Bitcoin uses

- a peer-to-peer network protocol (see BitTorrent for an example of a peer-to-peer network) consisting of **miner** nodes (miners compete to add blocks to the blockchain) and nodes that keep the network operating (these nodes validate blocks, confirm transactions and send updates to ledgers in real time) - *Figure 34.6*.
- some core cryptographic functions - SHA-256, RIPEMD160, Elliptic Curve Digital Signature Algorithm (ECDSA), public key/private key encryption.
- Game theory - a dynamically adjusting equilibrium system that uses economics at a global scale - the energy cost incurred in solving a challenge (proof of work) - to enable a trustless distributed system to function successfully.

Miners create blocks containing transactions picked from a pool of transactions.
Each miner that successfully solves a challenge may then attach their block to the chain of blocks called Bitcoin's blockchain - *Figure 34.7*.
The challenge is of sufficient computational difficulty to deny an algorithmic solution so miners are left with no choice but to use a brute-force approach. The level of difficulty is set so that finding a solution takes approximately 10 minutes. Miners using hashing algorithm SHA-256 have to find the nonce (number used only once) that when concatenated with a hash of the previous block, a hash of the transactions in the block and a timestamp produces a

1 Wikipedia - Text under CC-BY-SA license - https://en.wikipedia.org/wiki/Ledger

hash with a required number of leading zeroes - *Figure 34.8*. The hash value produced in this way becomes the new block's hash value.

Block hash = SHA-256 Hash(Hash of previous block + Transactions hash + Timestamp + Nonce)

https://bitnodes.io site is ©ADDY YEOW

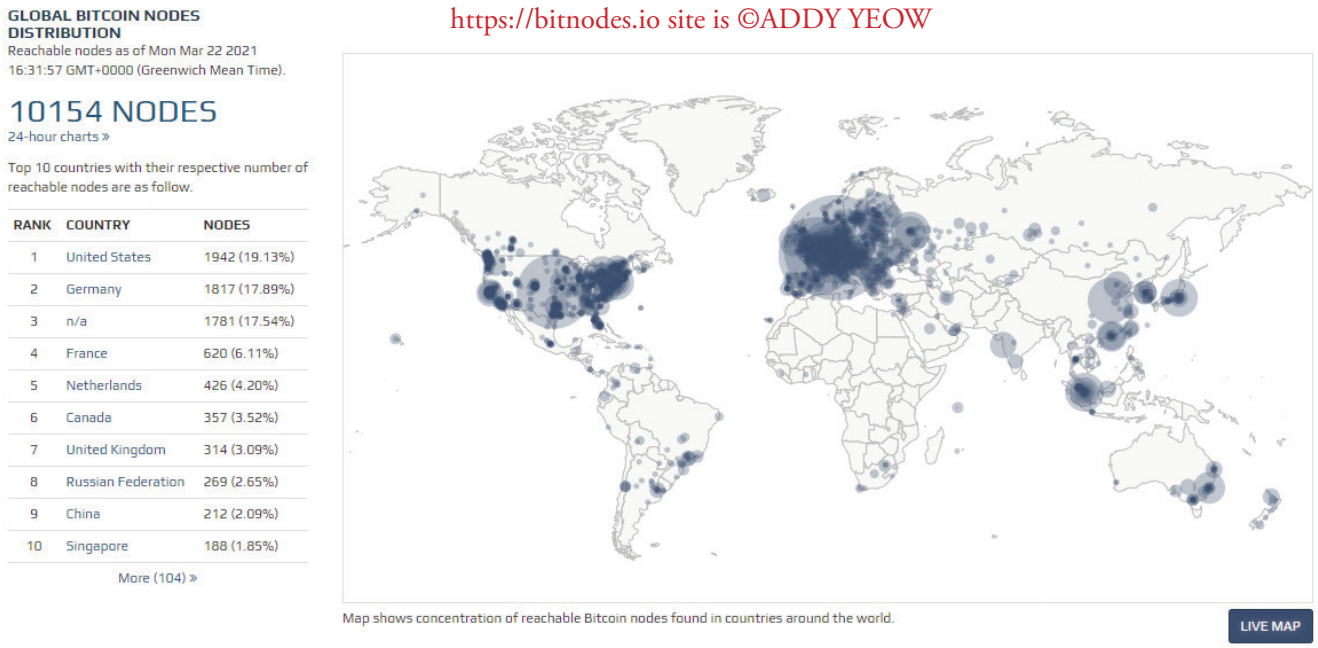


Figure 34.6 Map of bitcoin nodes

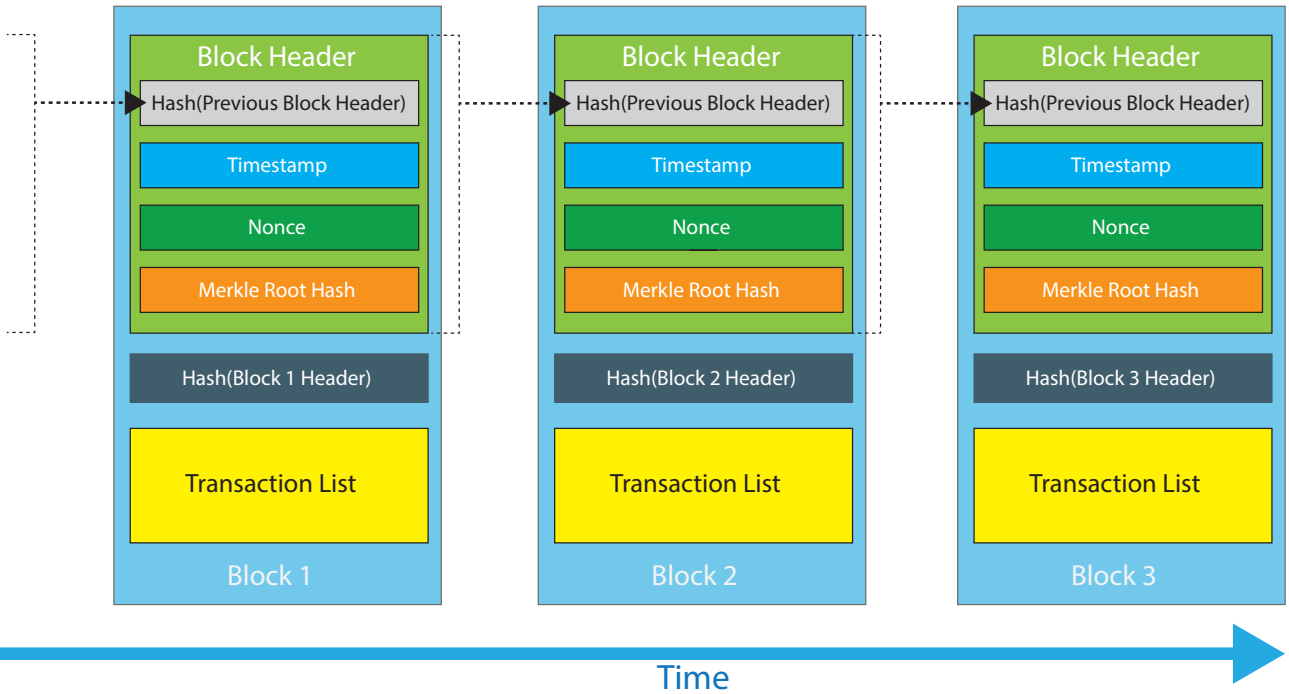


Figure 34.7 Structure of a block in Bitcoin's blockchain

The first miner to solve the challenge has their block of transactions added to the blockchain once their solution has been validated by other nodes in the network (the new block contains everything needed to validate the block's hash, i.e. verify that the challenge has been solved). As each node in the network contains a local copy of the blockchain, the local copies are updated by propogating the validated new block through the network. Very occasionally, two miners arrive at a solution about the same time (their block's timestamp will verify this) and start the propogation process at about the same time. Imagine that these two miners are on opposite sides of the globe, then for a while nodes closer to one miner will add this miner's block while nodes closer to the other will get that

miner's block. Thus for a while, half the network will have one blockchain version and the other half the other. What should be done when the new blocks cross over? When this happens a fork is added to the blockchain to accommodate the other block. On receiving a validated block, miners stop working on the current challenge -some miners will be using transactions that occur in the accepted block - and start a new block challenge with other transactions from the unprocessed pool. The arrival of the second block with a timestamp similar to the newly added block signals that a fork must occur. The winner of the next challenge round will add their validated block to the oldest branch of the fork i.e. the branch before the fork arose in their local copy. After five new blocks have been added, one branch of the fork must be longer than the other. The longest branch is then kept and the shorter one abandoned, with its transactions returned to the unprocessed transaction pool if they have not been included in any of the blocks in the retained branch.

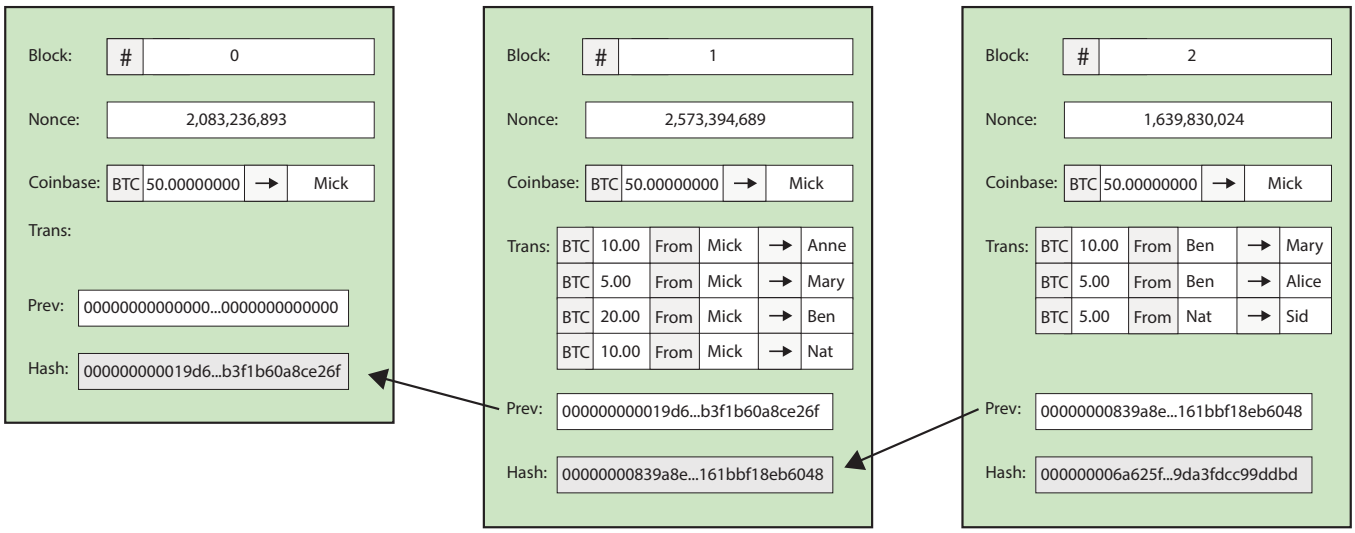


Figure 34.8 Simplified view of blocks in the blockchain

The blockchain is a chain of blocks containing every transaction that has occurred with the very first, a special transaction called a **coinbase** transaction, recorded in block 0. **Coinbase** transaction are assigned their own space in the block. In Bitcoin, miners are rewarded with bitcoins everytime they succeed in solving the challenge. *Figure 34.8* shows that miner Mick has been rewarded three times with 50 BTC each time. This is recorded in the **coinbase** field of the block. Whilst Bitcoin was still in its *proof of concept* stage, the blocks were currency-generation-blocks similar to block 0 in *Figure 34.8*. Spending came later when it was clearly demonstrated that Bitcoin had solved the **coordination problem for shared ledgers**:

- Shared ledgers (each node in the network has their own local copy of the ledger) with multiple collaborators are vulnerable to errors infiltrating the ledger because it is hard to coordinate the actions of multiple users when they are acting independently.
- In a "trustless" community of anonymous users, in addition to honest mistakes, the system must also guard against malicious users who are potentially seeking to defraud others.

Balances are not recorded in the blockchain, instead to answer the question, "How many BTCs does Mick have currently?" requires that a local copy of the ledger is searched for all Mick's transactions, both inputs and outputs. The answer to the question is then given by the sum of the inputs minus the sum of the outputs. This is also the approach used to prevent Mick from spending what he doesn't have.
The blockchain of Bitcoin may be explorered at www.blockchain.com. *Figure 34.9* shows block 2.

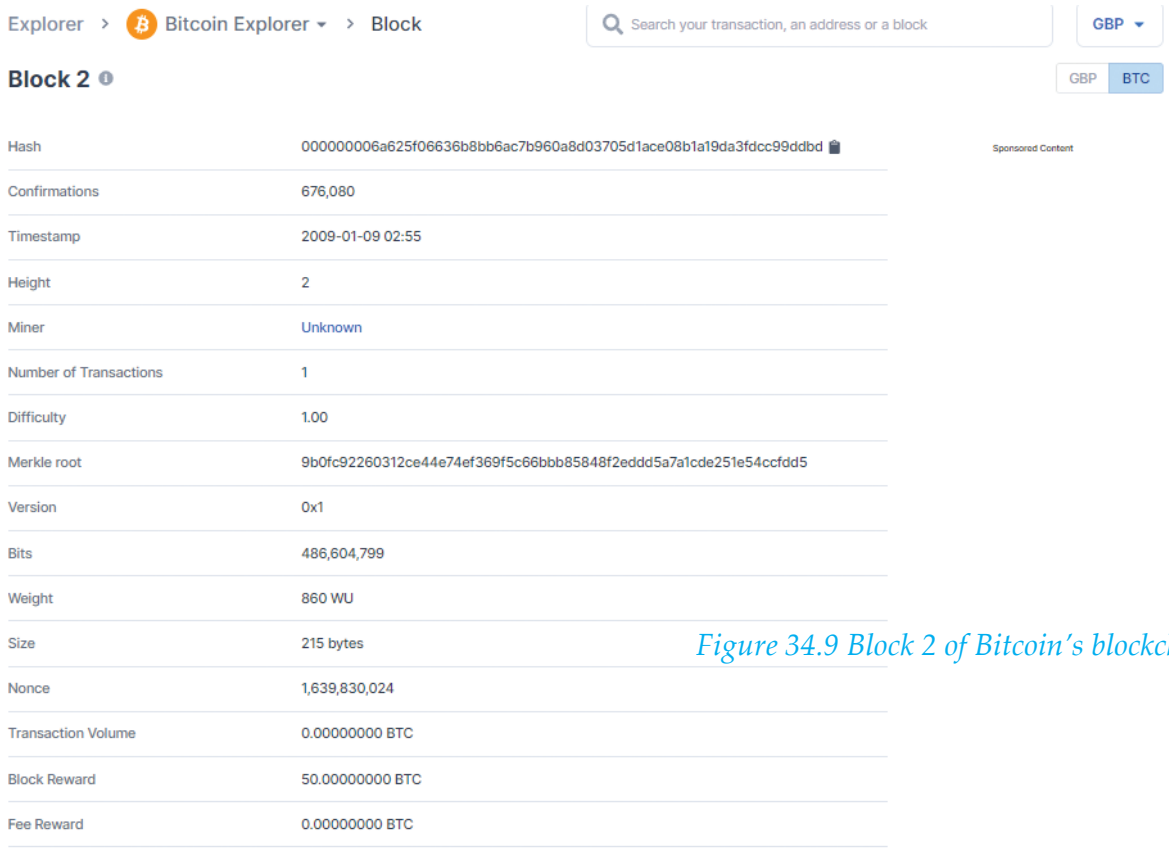


Figure 34.9 Block 2 of Bitcoin's blockchain

Figure 34.10 shows the first two transactions for block 400000. In addition to collecting a reward in btc for mining a block, miners may also collect a fee per transaction. The sender in the second transaction has Bitcoin address

13XSrVkweo5Dzm3yuykFw4P63N63MA6bTd

This uses a modified Base 58 binary-to-text encoding known as **Base58Check**[2]. This sender sent 0.19206072 BTC to address 1HU1LDBXUg73f2ro2e2dB3XY8cFoYLFgZZ. The transaction itself is protected with a hash value

0de586d0c74780605c36c0f51dcd850d1772f41a92c549e3aa36f9e78e905284

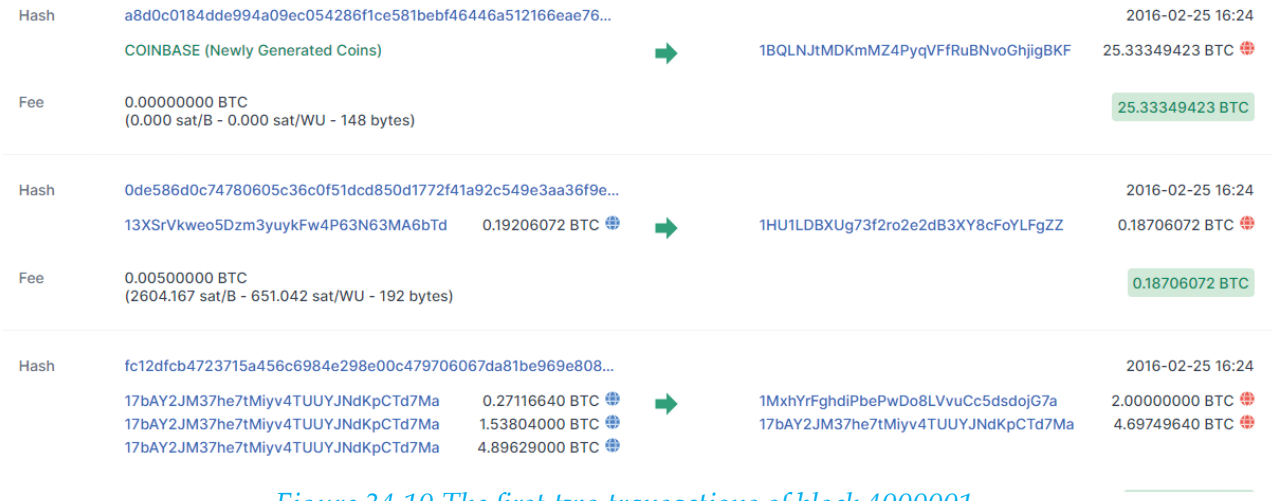


Figure 34.10 The first two transactions of block 4000001

Bitcoin relies on public key cryptography, where a private key – comparable to an account password –is used to authorise ("sign") a movement of funds stored in an address. The address, which can be thought of as an account number, is derived from the public key that mathematically corresponds to the private key. Each owner transfers Bitcoin by digitally signing a hash of the previous transaction(s) (assigning Bitcoin to the sender - any change is just

2 https://en.bitcoin.it/wiki/Base58Check_encoding

sent back to the sender) and the public key of the next owner. Transactions are therefore digitally signed records that reassign ownership of Bitcoins to new addresses. Miners verify that a transaction is genuine using the sender's public key. Software programs, commonly referred to as wallets, handle the management of the key pairs - see https://www.coinbase.com/ for an example of how to create a wallet and to start using Bitcoin. Satoshi Nakamoto, the anonymous creator of the Bitcoin network, actually defined Bitcoin as the chain of digital signatures that come together to form a blockchain.

A user's Bitcoin network wallet - e.g https://www.coinbase.com/ - monitors the user's Bitcoin addresses (they can have more than one public/private key pair) and keeps a record of all the transactions associated with these Bitcoin addresses to create the user's balance from the Bitcoin ledger.

Figure 34.11 shows the hash generated for the string "Hello World!" by Delphi program `HashingWithSHA256Project.exe`. Load this program and make small alterations to the string, e.g. remove the '!' character. Notice how the hash value changes in an unpredictable and seemingly random way. Of course, SHA-256 is deterministic, i.e. the same output is produced for a given input. However, its output passes statistical tests used to determine randomness. *Table 34.7* shows the unit for this program, `HashingWithSHA256Unit.pas`.

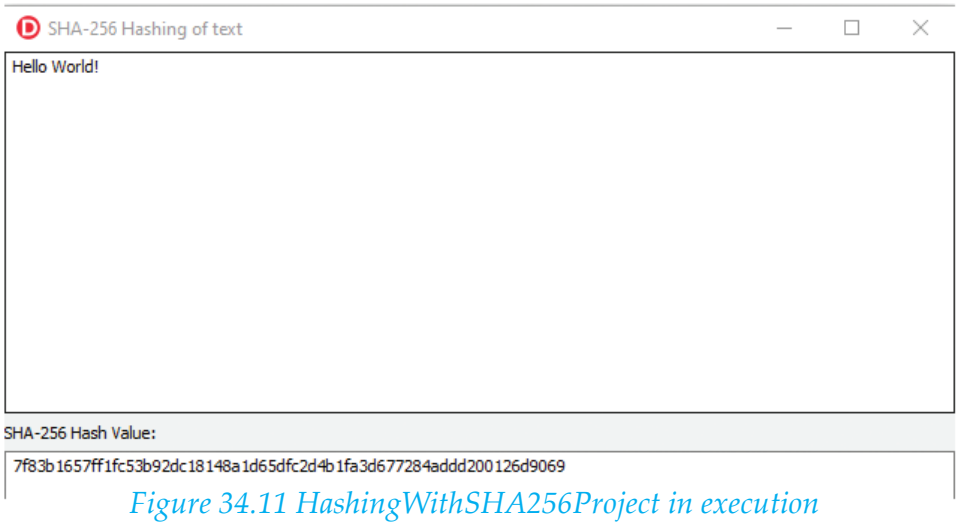


Figure 34.11 HashingWithSHA256Project in execution

Figure 34.11 shows the construction of a binary tree of hashes known as a **Merkle tree**, which is used primarily to verify the data held within, without revealing what that data is, and at speed. Alteration of any of the underlying data will be readily revealed.

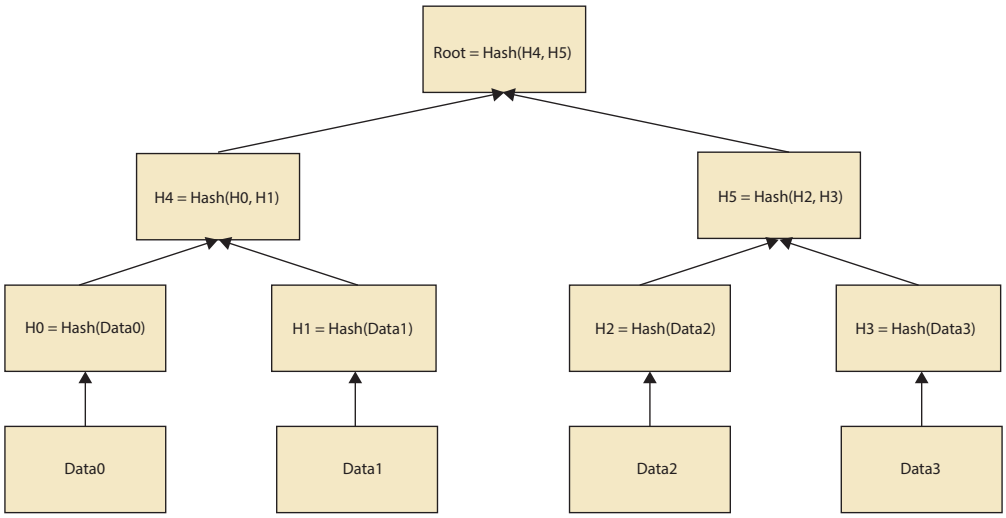


Figure 34.11 Merkle tree - binary data tree of hashes

```
Unit HashingWithSHA256Unit;
Interface
Uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls, System.Hash;
Type
  TfrmSHA256HashingOfText = Class(TForm)
                              mbText : TMemo;
                              mbSHA256HashValue : TMemo;
                              lblSHA256HashValue : TLabel;
                              Procedure mbTextChange(Sender : TObject);
                            End;
Var
  frmSHA256HashingOfText: TfrmSHA256HashingOfText;

Implementation
{$R *.dfm}
  Procedure TfrmSHA256HashingOfText.mbTextChange(Sender : TObject);
    Begin
      mbSHA256HashValue.Clear;
      mbSHA256HashValue.Lines.Add(THashSHA2.GetHashString(mbText.Text));
    End;
End.
```

P

Table 34.7 HashingWithSHA256Unit.pas

Simulating a blockchain in Delphi

Figure 34.12 shows a simulated blockchain created in Delphi in which eleven blocks are linked by a backward chain.

Start a new multi-device application **New|Multi-Device Application|Blank Application**.

Save the project as `BlockChainProject.dproj` and its unit as `BlockChainUnit.pas` in folder `BlockChainProject`.

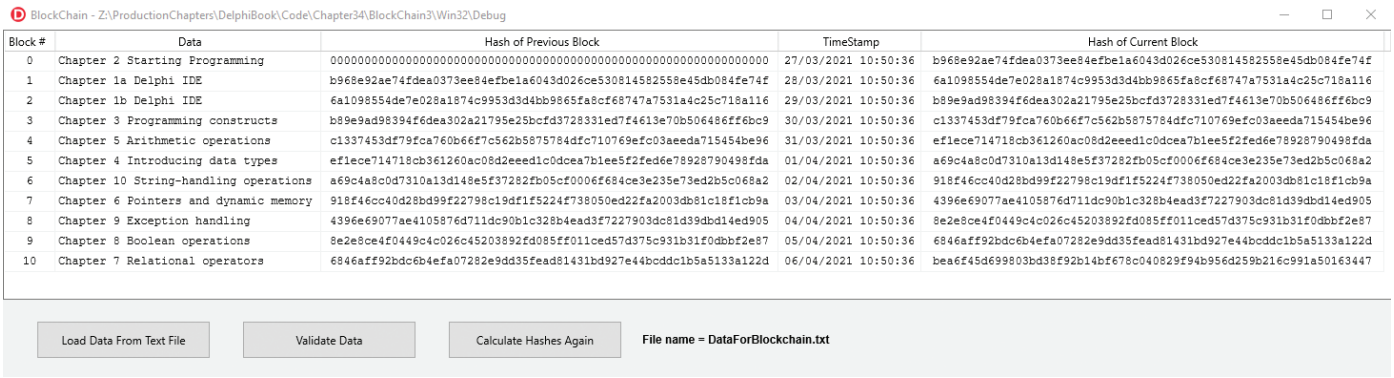


Figure 34.12 Simulated blockchain - BlockchainProject.exe

Add the following components to the form:

- 2 x **TLayout**
- 3 x **TButton**
- 1 x **TLabel**
- 1 x **TGrid**
- 1 x **TFDMemTable**(an in-memory dataset)
- 1 x **TOpenDialog**

Use TClientDataSet instead for VCL applications

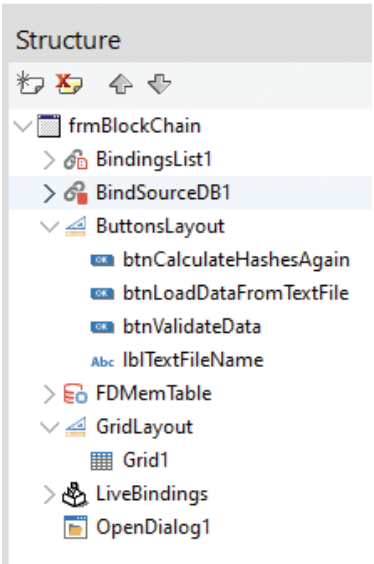


Figure 34.13

Configure and **rename** these as shown in *Figure 34.12* and *Figure 34.13*.

Set Align property of `ButtonsLayout` to **Bottom**.

Set Align property of `GridLayout` to **Client**.

Set Align property of `Grid1` to **Client**.

Set Width property of `lblTextFileName` to 753 and `Text` property to empty string.

Select `FDMemTable`|**Fields** then right click **Fields** to bring up window shown in *Figure 34.14*.

Click **New Field** option to bring up the **New Field** window shown in *Figure 34.15*.

Enter the name `Index` for the **Name** field and **Integer** for the **Type** field.

Click **OK**.

Repeat to create four more new fields with name and type as follows

- `Data` - **String**
- `HashOfPreviousBlock` - **String**
- `TimeStamp` - **DateTime**
- `HashOfCurrentBlock` - **String**

The name assigned to each field is the field's **DisplayLabel** property value whilst the actual field name is a concatenation of `FDMemTable` and name - see *Figure 34.16* and the **Object Inspector**.

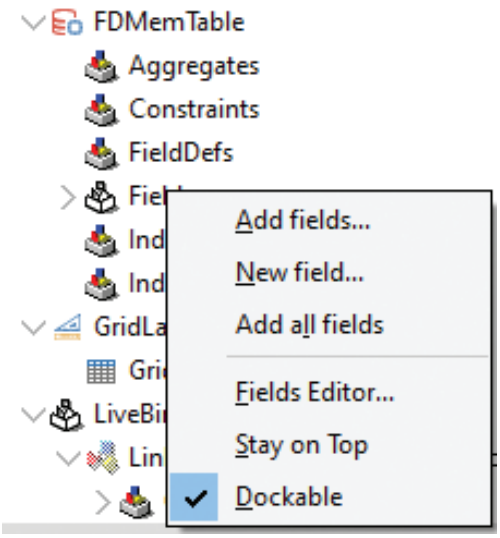


Figure 34.14 Fields popup window

Right click in the grid area of the form to bring up the window shown in *Figure 34.17*.

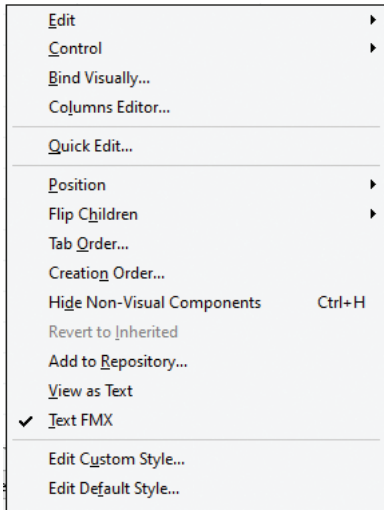


Figure 34.17

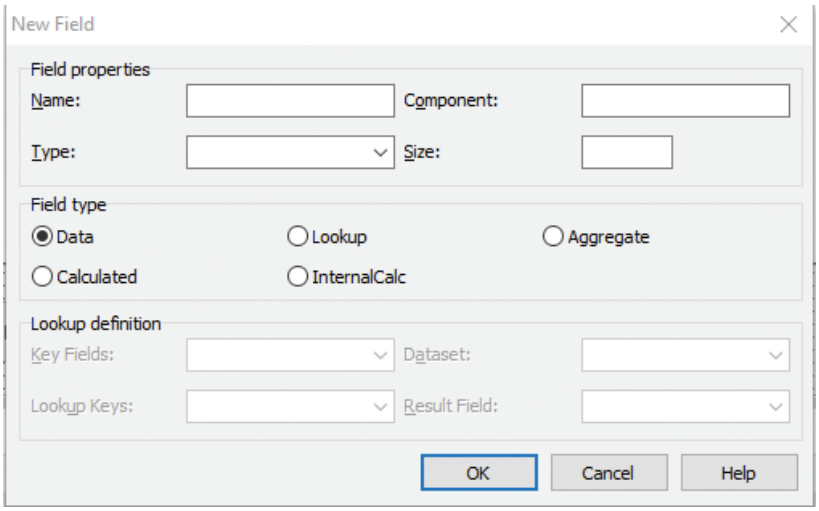


Figure 34.15 New Field window

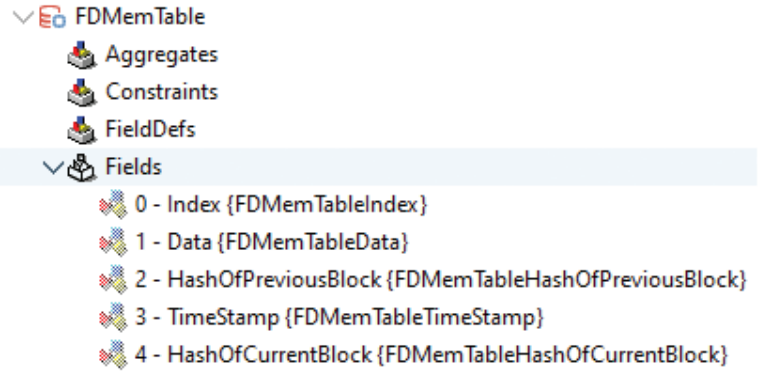


Figure 34.16 The five fields of FDMemTable

Click **Bind Visually...** option to bring up the **LiveBindings Designer** window shown in *Figure 34.18* (without the connections between `FDMemTable` and `Grid1`).

Click and **hold** the mouse button down on the `Index` field of `FDMemTable`.

Drag the mouse to the * field of `Grid1` and release the mouse button. A connecting line with an arrowhead at each end is created anchored at one end to the `Index` field of `FDMemTable` and at the other end to a newly created field `Column[0]`. **Repeat** for the other fields of `FDMemTable` to add four more connecting lines.

Figure 34.19 shows the user interface after connecting `FDMemTable` to `Grid1`. **Note** that **LiveBindings Designer** has added two new components to manage the connection, **BindingsList1** and **BindSourceDB**.

Save All (**SHIFT+CTRL+S**).

Page references to update

Add the `Private` section to the class definition `TfrmBlockChain` as shown in *Table 34.8* - page 560.

Place the cursor in `CalculateHashes` and **press** **Shift+Ctrl+C** to create the skeleton for this procedure in the **Implementation** section.

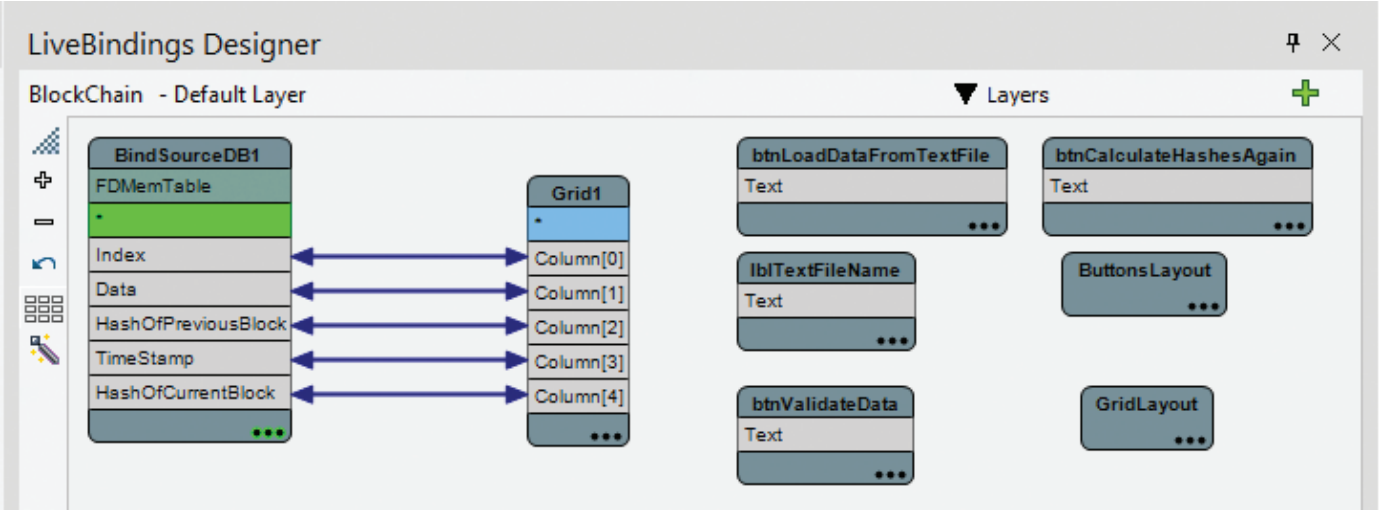


Figure 34.18 LiveBindings Design window

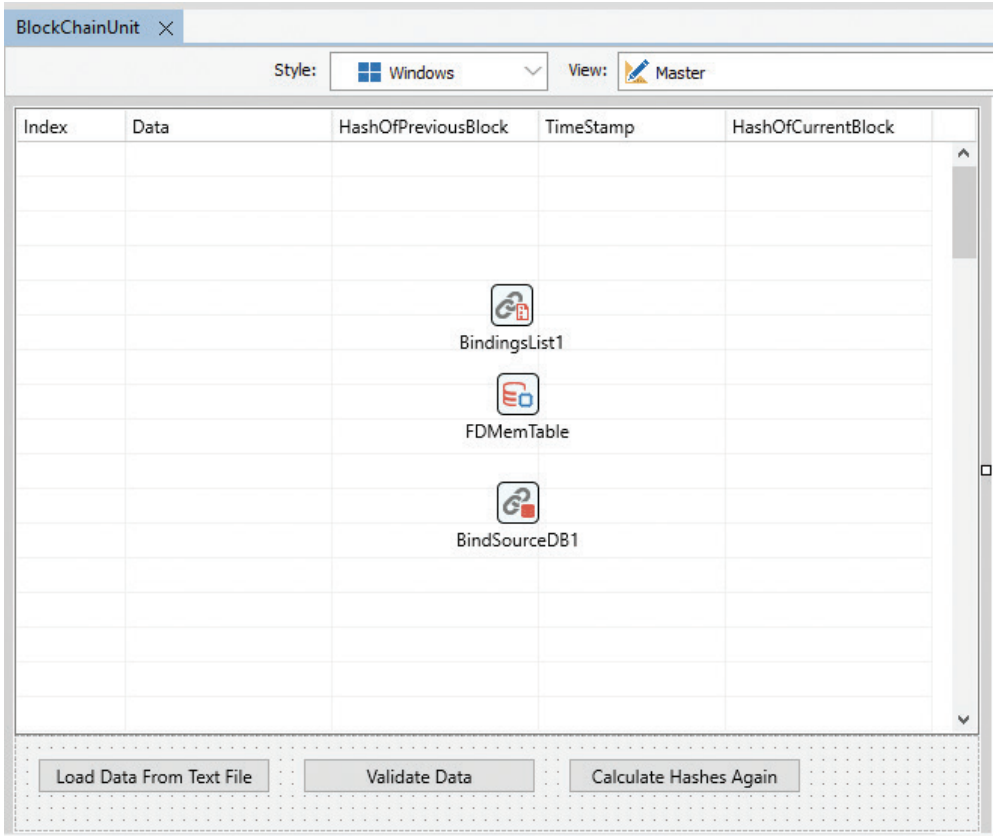


Figure 34.19 Design View of User Interface after using LiveBindings Designer

Define the constant `cnstGenesisBlockAddress` then add the code shown in *Table 34.8* to the body of procedure `CalculatesHashes`. **Add** `System.Hash` to the **Uses** clause in the **Interface** section.

Double click button `Load Data From Text File` to create an event handler. **Add** the code shown in *Table 34.9* to its body - page 561. **Add** `System.IOUtils` to the **Uses** clause in the **Interface** section.

Double click the **OnCreate** field in the **Events** page of `frmBlockChain` to create an event handler `FormCreate`.

Add the code shown in *Table 34.11* to the body of this event handler - page 562.

Select `Grid1` in the **Object Inspector,** open the `LinkGridToDataSourceBindSourceDB1` - *Figure 34.20*.

Click on the ellipsis (**...**) field of the **Columns** property to bring up the editing window shown in *Figure 34.21*.

Page references to update

Select each column in *Figure 34.21* in turn and edit the column in the **Object Inspector** as shown in *Figure 34.22*.

Use the following values

```
0 - Index
      Alignment taCenter
      TextWidth 8
1 - Data
      Alignment taLeftJustify
      TextWidth 44
2 - HashOfPreviousBlock
      Alignment taCenter
      TextWidth 68
3 - TimeStamp
      Alignment taCenter
      TextWidth 22
4 - HashOfCurrentBlock
      Alignment taCenter
      TextWidth 68
```

Select `FDMemTable`|**Fields** - *Figure 34.23*.

Select each column in *Figure 34.23* in turn and edit the column in the **Object Inspector** as shown in *Figure 34.24*.

Use the following values

```
0 - Index
      Alignment taCenter
      TextWidth 8
1 - Data
      Alignment taLeftJustify
      TextWidth 44
      Size 44
2 - HashOfPreviousBlock
      Alignment taCenter
      TextWidth 64
      Size 64
3 - TimeStamp
      Alignment taCenter
      TextWidth 22
4 - HashOfCurrentBlock
      Alignment taCenter
      TextWidth 64
      Size 64
```

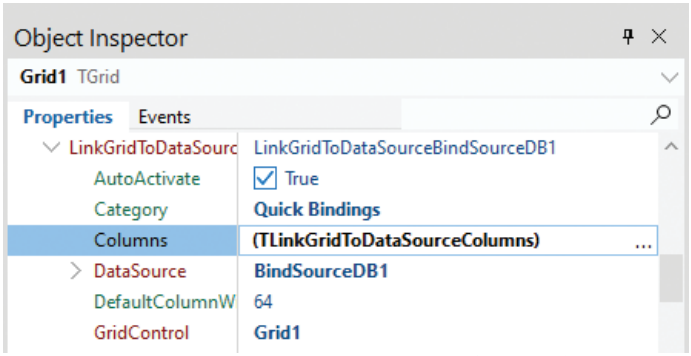


Figure 34.20 Object Inspector for Grid1

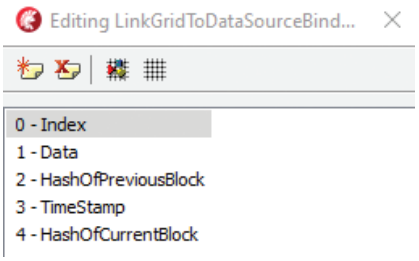


Figure 34.21 Editing LinkGridToDataSourceBindSourceDB1.Columns

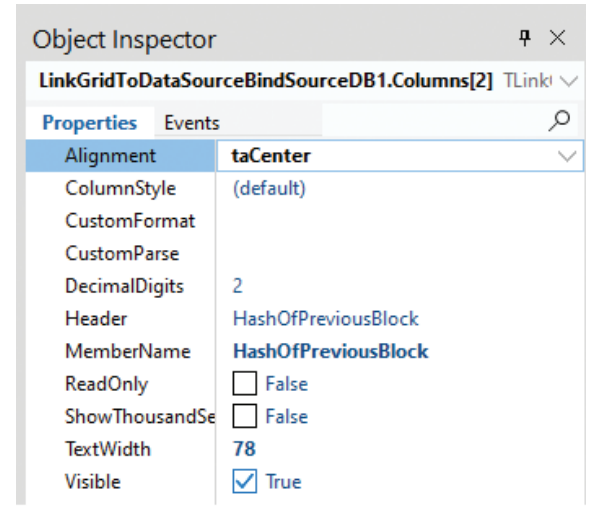


Figure 34.22 Editing Column[2] - setting TextWidth to 78

Figure 34.25

Double click button `Validate Data` to create an event handler.

Add the code shown in *Table 34.11* to its body - page 562.

Double click button `Calculate Hashes Again` to create an event handler. **Add** the code shown in *Table 34.9* to its body - page 561.

Change the `frmBlockChain` properties as follows

```
Caption = 'BlockChain'
ClientHeight = 340
ClientWidth = 1431
```

Save All (**SHIFT+CTRL+S**).

Download `DataForBlockchain.txt`. - *Table 34.10*.

Click **Run** (**F9**) to build and run the application `BlockChainProject`.

Click `Load Data From Text File` button and **locate** and **load** `DataForBlockchain.txt`. *Figure 34.12* shows the result.

Click `Validate Data` button to validate the blockchain - *Figure 34.25*.

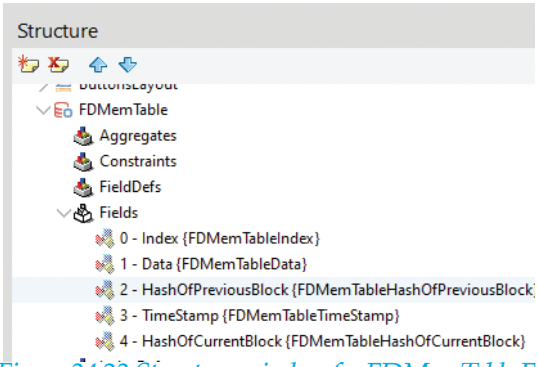


Figure 34.23 Structure window for FDMemTable Fields

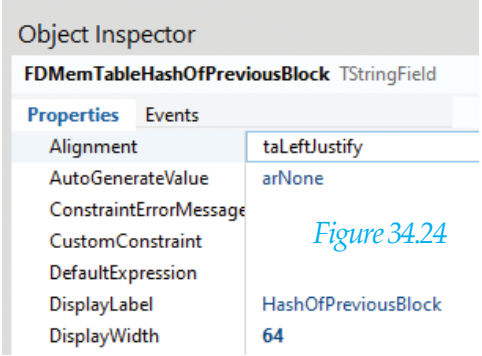


Figure 34.24

```
Unit BlockChainUnit;
Interface
Uses
  System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants, System.IOUtils,
  FMX.Controls, FMX.Forms, FMX.Graphics, FMX.Dialogs, FireDAC.Stan.Intf, FireDAC.Stan.Option,
  FireDAC.Stan.Param,  FireDAC.Stan.Error, FireDAC.DatS, FireDAC.Phys.Intf, FireDAC.DApt.Intf,
  System.Rtti, FMX.Grid.Style, Data.Bind.EngExt, Fmx.Bind.DBEngExt, Fmx.Bind.Grid,
  System.Bindings.Outputs, Fmx.Bind.Editors, Data.Bind.Components, Data.Bind.Grid, FMX.StdCtrls,
  FMX.Layouts, Data.Bind.DBScope, FMX.Controls.Presentation, FMX.ScrollBox, FMX.Grid, Data.DB,
  FireDAC.Comp.DataSet, FireDAC.Comp.Client, System.Hash, FMX.Types, FMX.Memo.Types, FMX.Memo, FMX.Edit;
Type
  TfrmBlockChain = Class(TForm)
                     FDMemTable : TFDMemTable;
                     Grid1 : TGrid;
                     BindingsList1 : TBindingsList;
                     BindSourceDB1 : TBindSourceDB;
                     ButtonsLayout : TLayout;
                     btnCalculateHashesAgain : TButton;
                     btnValidateData : TButton;
                     LinkGridToDataSourceBindSourceDB1 : TLinkGridToDataSource;
                     FDMemTableIndex : TIntegerField;
                     FDMemTableData : TStringField;
                     FDMemTableHashOfPreviousBlock : TStringField;
                     btnLoadDataFromTextFile: TButton;
                     FDMemTableTimeStamp : TDateTimeField;
                     OpenDialog1 : TOpenDialog;
                     lblTextFileName : TLabel;
                     FDMemTableHashOfCurrentBlock : TStringField;
                     GridLayout : TLayout;
                     Procedure btnCalculateHashesAgainClick(Sender : TObject);
                     Procedure btnValidateDataClick(Sender : TObject);
                     Procedure btnLoadDataFromTextFileClick(Sender : TObject);
                     Procedure FormCreate(Sender : TObject);
                     Private
                       CalculatingHashesFlag : Boolean;
                       Procedure CalculateHashes;
                   End;
Var
  frmBlockChain : TfrmBlockChain;
Implementation
{$R *.fmx}
  Const
    cnstGenesisBlockAddress = '0000000000000000000000000000000000000000000000000000000000000000';
  Procedure TfrmBlockChain.CalculateHashes;
    Var
      strString, strHash, strPreviousHash : String;
      Begin
        If CalculatingHashesFlag
          Then Exit;
        CalculatingHashesFlag := True;
        FDMemTable.First;
        FDMemTable.BeginBatch;
        LinkGridToDataSourceBindSourceDB1.Active := False;
        Try
          While Not FDMemTable.Eof
            Do
              Begin
                If (FDMemTableHashOfPreviousBlock.AsString = cnstGenesisBlockAddress)
                  Then strPreviousHash := cnstGenesisBlockAddress
                  Else strPreviousHash := strHash;
                strString := FDMemTableIndex.AsString + FDMemTableData.AsString + strPreviousHash
                               + FDMemTableTimeStamp.AsString;
                strHash := THashSHA2.GetHashString(strString);
                FDMemTable.Edit;
                FDMemTableHashOfCurrentBlock.AsString := strHash;
                FDMemTableHashOfPreviousBlock.AsString := strPreviousHash;
                FDMemTable.Post;
                FDMemTable.Next;
              End;
        Finally
          CalculatingHashesFlag := False;
          FDMemTable.EndBatch;
          LinkGridToDataSourceBindSourceDB1.Active := True;
        End;
      End;
```

Table 34.8 Part 1 of BlockChainUnit.pas

```
Procedure TfrmBlockChain.btnCalculateHashesAgainClick(Sender : TObject);
  Begin
    CalculateHashes;
  End;

Procedure TfrmBlockChain.btnLoadDataFromTextFileClick(Sender : TObject);
  Var
    strlistLines : TStringList;
    intIndex: Integer;
    dtDateTime : TDateTime;
    FileName : String;
    CurrentDir : String;
  Const
    cnstDataFileName = 'DataForBlockChain.txt';
  Begin
    CurrentDir := GetCurrentDir;
    frmBlockChain.Caption := 'BlockChain - ' + CurrentDir;
    If OpenDialog1.Execute
      Then FileName := OpenDialog1.FileName
      Else FileName := CurrentDir + '\' + cnstDataFileName;
    If TFile.Exists(FileName)
      Then
        Begin
          FDMemTable.First;
          FDMemTable.BeginBatch;
          CalculatingHashesFlag := True;
          LinkGridToDataSourceBindSourceDB1.Active := False;
          Try
            FDMemTable.EmptyDataSet;
            strlistLines := TStringList.Create;
            Try
              strlistLines.LoadFromFile(FileName);
              dtDateTime := Now;
              For intIndex := 0 To strlistLines.Count - 1
                Do
                  Begin
                    FDMemTable.Insert;
                    FDMemTableIndex.Value := intIndex;
                    FDMemTableData.AsString := strlistLines[intIndex];
                    dtDateTime := dtDateTime + 1;
                    FDMemTableTimeStamp.AsDateTime := dtDateTime;
                    If intIndex = 0
                      Then FDMemTableHashOfPreviousBlock.Value := cnstGenesisBlockAddress;
                    FDMemTable.Post;
                  End;
            Finally
              strlistLines.Free;
            End;
          Finally
            FDMemTable.EndBatch;
            LinkGridToDataSourceBindSourceDB1.Active := True;
            CalculatingHashesFlag := False;
          End;
          CalculateHashes;
          lblTextFileName.Text := 'File name = ' + ExtractFileName(FileName);
        End
      Else ShowMessage('File with name ' + FileName + ' doesn''t exist');
  End;
```

Table 34.9 Part 2 of BlockChainUnit.pas

Now **change** the data, e.g. 'Chapter 1b Delphi IDE' to 'Chapter 1 Delphi IDE'.

Click `Validate Data` button to validate the blockchain but the blockchain is no longer valid - *Figure 34.26*.

Click `Calculate Hashes Again` button to regenerate the blockchain hashes. Now confirm that the blockchain is valid again.

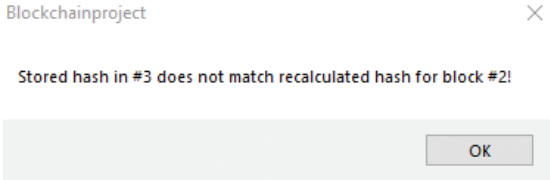


Figure 34.26

Chapter 2 Starting Programming
Chapter 1a Delphi IDE
Chapter 1b Delphi IDE
Chapter 3 Programming constructs
Chapter 5 Arithmetic operations
Chapter 4 Introducing data types
Chapter 10 String-handling operations
Chapter 6 Pointers and dynamic memory
Chapter 9 Exception handling
Chapter 8 Boolean operations
Chapter 7 Relational operators

Table 34.10 DataForBlockchain.txt

```
Procedure TfrmBlockChain.btnValidateDataClick(Sender : TObject);
  Var
    strString, strHash : String;
    intIndex : Integer;
  Begin
    FDMemTable.First;
    FDMemTable.BeginBatch;
    Try
      While Not FDMemTable.EOF
        Do
          Begin
            strString := FDMemTableIndex.AsString + FDMemTableData.AsString
                          + FDMemTableHashOfPreviousBlock.AsString + FDMemTableTimeStamp.AsString;
            strHash := THashSHA2.GetHashString(strString);
            FDMemTable.Next;
            If (Not FDMemTable.EOF) And (FDMemTableHashOfPreviousBlock.AsString <> strHash)
              Then
                Begin
                  intIndex := FDMemTableIndex.Value;
                  ShowMessage(Format('Stored hash in #%d does not match recalculated hash for block #%d!',
                                     [intIndex, intIndex - 1]));
                  Exit;
                End;
          End;
    Finally
      FDMemTable.EndBatch;
    End;
    ShowMessage('Blockchain is valid!');
  End;

 Procedure TfrmBlockChain.FormCreate(Sender : TObject);
   Begin
     FDMemTable.Active := True;
     LinkGridToDataSourceBindSourceDB1.Columns[0].Header := Format('%7s', ['Block #']); //Centre heading
     LinkGridToDataSourceBindSourceDB1.Columns[1].Header := Format('%42s', ['Data']); //Centre heading
     LinkGridToDataSourceBindSourceDB1.Columns[2].Header := Format('%73s', ['Hash of Previous Block']);
     LinkGridToDataSourceBindSourceDB1.Columns[3].Header := Format('%22s', ['TimeStamp']);
     LinkGridToDataSourceBindSourceDB1.Columns[4].Header := Format('%73s', ['Hash of Current Block']);
   End;
End.
```

Table 34.11 Part 3 of BlockChainUnit.pas

`BlockChainProject.exe` calculates the hash of the current block as follows

```
strString := FDMemTableIndex.AsString + FDMemTableData.AsString + strPreviousHash
               + FDMemTableTimeStamp.AsString;
strHash := THashSHA2.GetHashString(strString);
```

This differs from how **Bitcoin** generates the hash for the current block which is as follows

Block hash = SHA-256 Hash(Hash of previous block + Transactions hash + Timestamp + Nonce)

`BlockChainProject.exe` uses the block's index instead of a nonce. This makes calculating the hash very quick. In Bitcoin, in order to deter miners from attempting to rewrite transactions in previous blocks, the opposite is true. If a miner alters a block before the current block, let's say the sixth block back, then the miner will have to recalculate the hashes for the following five blocks. But in the time it will take the miner to do this, assuming that the hardware used is not more than five times faster than other miners' hardware, five new blocks can be added by other miners, thus defeating the miner's attempt to alter the chaining without being detected.

Miners have to calculate the nonce - a number that is used only once - that when concatenated with the transaction data, timestamp and hash of previous block generates a hash with a specific number of leading zeros (the greater the number of zeros the more difficult the task). The only way to do this is by brute force, incrementing the nonce value in a loop until the required number of leading hexadecimal zeros is reached. *Table 34.12* illustrates the principle using a simple message string 'Hello!' concatenated with a series of numbers starting with 0. The SHA 256 column shows the hash result expressed in hexadecimal. Note that none of the hash values has leading hexadecimal zeros. *Figure 34.9* shows a block of bitcoin with a hash containing 8 leading hexadecimal zeros, i.e. 32 leading binary digits (this block was added in 2009).

Message + Nonce	SHA 256 Hash
'Hello!'	5e9f00ca1f8a276af5de8b6373ccffd498ff22926c518721b75c29ae6a73d5d6
'Hello!0'	a4dde72d413d81baf052b997e3f1e389600df10bfe68ffd965cd6cf31bebb0c0
'Hello!1'	d4004ddb95a8d91cb70211c4d66cfd52efa947843c99c088138a49ce4e5ac019
'Hello!2'	32204efe17020ee4f5408d5c6729df928a894ea474c517f7de8b55c3f9507402
'Hello!209234516784532198765'	330b0fe5d1a886539705b7ee01c6b922405567aedad5fea8e591caf477fa5f6f
'Hello!2092345167845321987651 737610956784289765770022675126 5432789'	798d9f61396d906144d499d7c1b534ad52eb9d95fd41f82abd015fb58a9b79a4

Table 34.12 Hashes for a simple message string 'Hello!' concatenated with a nonce

The nonce determination is expressed slightly differently in reality.

The nonce is that value that generates a hash interpreted as a number which is numerically smaller than the target called the difficulty target.

The current target value using SHA 256 is a number in the range

$$2^{(256-1-k)} \text{ to } 2^{(256-k)}-1 \text{ inclusive}$$

where k is the number of leading zeros. SHA 256 uses 256 bits.

Table 34.13 illlustrates the possible ranges if k = 1 for 4, 5, 6 and n bits.

Total number of bits	Smallest binary number with one leading zero	Largest binary number with one leading zero	Smallest denary number with one leading zero	Largest denary number with one leading zero
4	0100	0111	4	7
5	01000	01111	8	15
6	010000	011111	16	31
n	01000...0000	01111...1111	$2^{(n-2)}$	$2^{(n-1)} - 1$

Table 34.13 Range of numbers with a single leading zero for a given number of bits

Suppose the current number of leading zeros that is required is set at 44, i.e. k = 44 (the year is currently 2021).

There are approximately $2^{212} - 2^{211} = 9 \times 2^{211} \approx 3 \times 10^{64}$ different target numbers between 2^{211} and $2^{212} - 1$, i.e for k = 44.

But there are 2^{256} different numbers given 256 bits, i.e. $\approx 10^{77}$.

The fraction of these which lie in the range for k = 44 is therefore $\frac{9 \times 2^{211}}{2^{256}} \approx 3 \times 10^{-13}$.

This means that roughly 4×10^{12} hashes must be calculated on average to find a hash less than the target value.

If the hardware is capable of executing 10^{10} searches of block hash space per second, locating a hash which satisfies the target will take 400 seconds on average or 7 minutes.

In bitcoin, the time-consuming and energy-expensive nonce calculation outlined above generates information that meets the specified conditions when successful. This information is taken as a proof that work has been done called **Proof of Work (PoW) to solve the challenge**. The purpose of PoW (Proof-of-work algorithm) is to check if calculations were indeed conducted during the creation process for a new block of cryptocurrency. The checking process or verification can be done very quickly. (The term mining is used by analogy with mining for precious metal - correct nonces are 'rare' and costly to produce).[3]

3 The two example programs on bitcoin were inspired by a webinair given by Jim McKeeth http://delphi.org/2018/02/delphi-and-the-blockchain-more-than-just-bitcoin-and-cryptocurrency/

Simulating a blockchain mining in Delphi

Start a new **Windows VCL application.**

Save the project as `BlockChainMiningProject.dproj` and its unit as `BlockChainMiningUnit.pas` in folder `BlockChainMIningProject`.

Add the following components to the form:

- 3 x **TPanel** - **rename** `TopPanel`, `MiddlePanel`, `BottomPanel`, **clear** their captions.
- 1 x **TListBox** - **add** to `BottomPanel`, **rename** `lbxNonceTargetHash`.
- 1 x **TMemo** - **add** to `TopPanel`, **rename** `mbMessage` **clear Lines** property.
- 1 x **TButton** - **add** to `MiddlePanel`, **rename** `btnMine`, **set Caption** property to `Mine`.
- 2 x **TLabel** - **add** to `MiddlePanel`, **rename** the first `lblDifficulty`, **set** its **Caption** property to `Difficulty`. and **rename** the second `lblElapsedMilliSeconds`, **set** its **Caption** property to the empty string
- 1 x **TSpinEdit** - **add** to `MiddlePanel`, **rename** `spbDifficulty`, **set MinValue** property to 1 and **MinValue** property to 5.
- 1 x **TSplitter**

Configure and **rename** these as shown in *Figure 34.27*.

Set Align property of `TopPanel` to **Top**.

Set Align property of `Splitter1` to **Top**.

Set Align property of `MiddlePanel` to **Top**.

Set Align property of `BottomPanel` to **Client**.

Set Align property of `mbMessage` to **Client**.

Set Align property of `lbxNonceTargetHash` to **Client**.

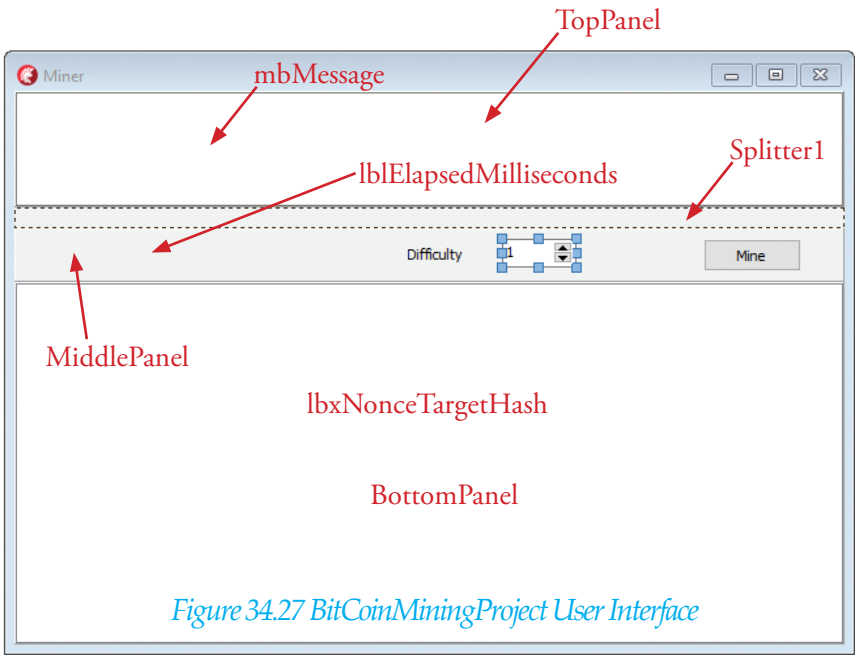


Figure 34.27 BitCoinMiningProject User Interface

Double click button `btnMine` to create an **OnClick** event handler.

Add the code shown in *Table 34.14*.

Save All (**SHIFT+CTRL+S**).

Click Run (**F9**) to build and run the application `BlockChainMiningProject`.

Try values of difficulty 1 to 5 for the message 'Hello!'.

Figure 34.28 shows the outcome for difficulty 5 which takes 6 minutes approximately to find a solution.

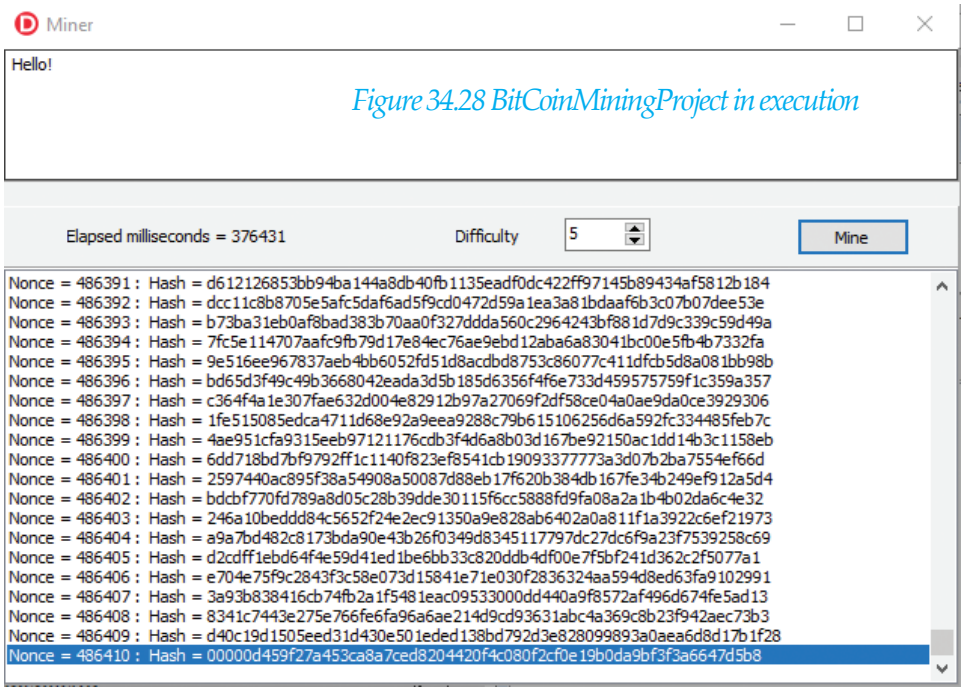


Figure 34.28 BitCoinMiningProject in execution

```
Unit BitCoinMiningUnit;
Interface
Uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls, Vcl.Samples.Spin,
  Vcl.ExtCtrls;
Type
  TfrmMiner = Class(TForm)
                TopPanel : TPanel;
                mbMessage : TMemo;
                BottomPanel : TPanel;
                Splitter1 : TSplitter;
                lbxNonceTargetHash : TListBox;
                spbDifficulty : TSpinEdit;
                lblDifficulty : TLabel;
                MiddlePanel : TPanel;
                btnMine : TButton;
                lblElapsedMilliseconds : TLabel;
                Procedure btnMineClick(Sender : TObject);
              End;
Var
  frmMiner: TfrmMiner;
Implementation
{$R *.dfm}
Uses System.Hash, System.Diagnostics;
  Procedure TfrmMiner.btnMineClick(Sender : TObject);
    Var
      strHash: string;
      intNonce: Integer;
      intDifficulty: Integer;
      StopWatch : TStopWatch;
    Begin
      lbxNonceTargetHash.Clear;
      intNonce := 0;
      intDifficulty := Trunc(spbDifficulty.Value);
      StopWatch.Start;
      Repeat
        strHash := THashSHA2.GetHashString(mbMessage.Text + intNonce.ToString);
        lbxNonceTargetHash.Items.Add(Format('%s = %d :  %s', [‘Nonce’, intNonce, ‘Hash = ‘ + strHash]));
        Inc(intNonce);
      Until strHash.Substring(0, intDifficulty) = StringOfChar('0', intDifficulty);
      lbxNonceTargetHash.ItemIndex := lbxNonceTargetHash.Items.Count - 1;
      StopWatch.Stop;
      lblElapsedMilliseconds.Caption := 'Elapsed milliseconds = '
                                        + IntToStr(StopWatch.ElapsedMilliseconds);
    End;
End.
```

Table 34.14 BitCoinMiningUnit.pas

Hash Tables
Tables

Using a table to store records

A table in computer science is a data structure of rows and columns, an example is shown in *Table 34.15*. This table consists of 4 rows of data in three columns, labelled `ULN, Forename, Surname`. Each row stores a single record of three fields containing data for an individual student as follows:

- student's unique learner number `(ULN)` consisting of eight digits, e.g. 34567890
- Forename
- Surname

An individual record is uniquely identified by its key field, `ULN`.

The rows of this table are indexed with the first row that contains a student record being labelled with index 0, the second with index 1, and so on.

	ULN	Forename	Surname
0	34567890	Fred	Bloggs
1	90002789	Mary	Smith
2	74432167	Ahmed	Khan
3	24567813	Sarah	White

Table 34.15 Student records stored in a table

This table will occupy a part of the computer's RAM (main memory). It can also be stored permanently in backing store or secondary storage, e.g. magnetic disk. However, to be searched or manipulated, it must first be copied from secondary store to RAM.

Searching the table for a record

The table could be searched for a particular record by starting at the row labelled with index 0 and scanning the entries in turn until the record is found if it is present, or the end of the table is reached. This is known as linear search which is one of several ways that an existing record can be 'looked up'.

Inserting a new record into the table

Table 34.16 shows a table similar to *Table 34.15* but this table has three empty rows following the four rows of data. A new record could be inserted in the first empty row, a second new record in the next row and so on until the table is full.

	ULN	Forename	Surname
0	34567890	Fred	Bloggs
1	90002789	Mary	Smith
2	74432167	Ahmed	Khan
3	24567813	Sarah	White
4			
5			
6			

Table 34.16 Student records stored in a table with room for new records

Deleting a record in the table

The row containing the student record to be deleted is located by searching from row 0. Once found, the data in the row is deleted. To avoid gaps appearing in the table, the occupied rows following this row are moved up to remove the gap.

Limitations of this type of table and table access

A problem surfaces with the operations of searching, inserting and deleting described above when the table contains a large number of records, e.g. 10,000. It just takes too much of the computer's time to perform these operations. One solution is to use a hash table based on a well-chosen hash function.

Hash table

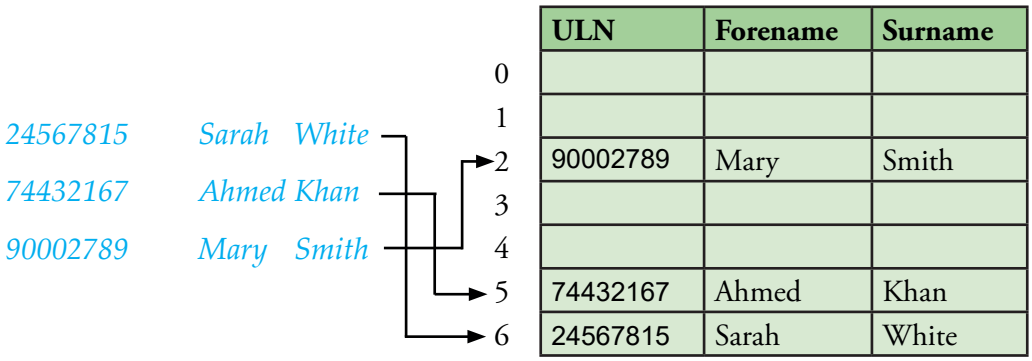
A hash table resembles an ordinary table as described above but differs in the method used to access the rows of the table.

A row of a hash table is accessed directly when looking up, inserting and deleting a record, i.e. it does not start from row 0 every time but instead goes directly to the required row. Movement of records when deleting a record is also eliminated.

Table 34.17 shows a hash table that has gone from being empty to containing 3 records located in three different rows with indices, 2, 5, and 6, respectively, as input data.

Key concept

Hash table:
The table gets its name from the method used to determine the row to use.
The hash value generated by applying a hash function to a key is the table index where the record with this key should be stored if the row is free.



	ULN	Forename	Surname
0			
1			
2	90002789	Mary	Smith
3			
4			
5	74432167	Ahmed	Khan
6	24567815	Sarah	White

Table 34.17 Hash table storing three student records

The table gets the name **hash** because of the method used to generate the address or row number. A randomising function called a **hash function** is applied to the record's key, in this case the 8-digit unique learner number or ULN, to map the possible 8-digit ULN values into a much smaller range of values, the possible row numbers. This process is known as **hashing**.

If the ULN values were used directly as specifiers of row addresses, we would have to accommodate addresses covering all possible values of an 8-digit number, 10^8 addresses in total, even though only a small subset of ULNs might be required, e.g. those used in a particular school.

For ease of understanding, the number of rows for the table in *Table 34.17* has been made small intentionally at seven, and labelled 0, 1, 2, ..., 5, 6.

Hash function

The hash function takes as input the record's key (hash key) and outputs the row address of the row for this record. The output is called the hash value or hash.

In our example, the hash value ranges from 0 to 6 for the seven rows of the given table. A hash function, `H`, that will map 8-digit `ULN`s to the set
{0, 1, 2, ..., 5, 6} is shown below

```
H(ULN) = ULN Mod 7
```

Mod is the modulo arithmetic operator which calculates the remainder after integer division (see *Chapters 5* and *33*).

Table 34.18 shows three possible values of `ULN` being mapped to 2, 5 and 6 respectively e.g. 90002789 when divided by 7 gives 12857541 with a remainder of 2.

ULN	H(ULN)
90002789	2
74432167	5
24567815	6

Table 34.18 Some hash values produced by hash function H applied to ULN keys

Key concept

Hash function:
Is a function H, applied to a key k, which generates a hash value H(k) of range smaller than the domain of values of k,
e.g.
H : {00000000..99999999}
→ {0..6}

Questions

1. Calculate `H(ULN)` for the following `ULN`s
   (a) 31258745 (b) 62517493 (c) 49981627
   Hint: The scientific mode of Microsoft Windows calculator has a `Mod` operator.

Key concept

Hash key:
Is the key that the hash function is applied to.

Simple hashing functions

Hashing and hash tables are a way that memory locations for records can be assigned so that records can be retrieved quickly.

A hashing function must be relatively quick to compute whilst at the same time generating an even spread of values for the given inputs, the keys.

Another way that the latter can be expressed is that each hash value generated by the hashing function should be equally probable.

Key concept

Hashing:
The process of applying a hash function to a key to generate a hash value.

Achieving this depends on both the particular key values being hashed, and the particular hash function employed.

The value of N in modulo N (e.g. Mod 7) is chosen to be prime because this can contribute to producing an even spread of hash values.

One simple hash function that attempts to achieve these objectives, sums the squares of the ASCII codes of each character of `Key`, as shown in *Table 34.19* in pseudo-code.

The `Ord` function returns the ASCII code of a given character,

e.g. `Ord('A') = 65`.

The individual characters of `Key` are accessed using array indexing starting at 0, e.g. `Key[0]` accesses the first character in the string.

The algorithm generates hash values in the range 0 ... 522 because `Sum` is Modded with 523, a prime number.

Suppose that `Key` stores a string, then the steps to convert `Key` into a storage-address returned in `Hash` are as follows:

```
Sum ← 0
For i ← 1 To Length(Key)
  Sum ← Sum + Ord(Key[i]) * Ord(Key[i])
Endfor
Hash ← Sum Mod 523
```

Table 34.19 Hashing algorithm that calculates a storage address in range 0 to 522

Looking up a record in a hash table

A record with a given key can be looked up by just calculating the hash of its key and checking the associated storage location.

English-French dictionary example

In this example, English words and their French equivalents are stored in records in a hash table, `HashTable`, using a hashing function, `H`, based on the hashing algorithm shown in *Table 34.19*. Each record must have a key field which uniquely identifies the record. In this case, the key is the English word.

The hashing function, `H`, assigns hash table memory location `H(k)` to the record with key, `k`.

In our English-French dictionary example, `H(k)` could be `H('BEACH')` where `k = 'BEACH'` for the record containing the English word `'BEACH'` and the equivalent French word `'PLAGE'`.

Information

The term "hash" originates by analogy with its non-technical meaning, to "chop and mix". Hash functions often "chop" the input domain into many sub-domains that get "mixed" e.g. add the first three digits of the key, add the last three digits, concatenate the two resulting digit strings then map into the output range by applying modulo N.

The storage structure, `HashTable`, that will be used with this address has the following data structure:

```
THashTableArray = Array[0..522] Of TRecord
```

Where the data structure `TRecord` is defined as follows

```
TRecord = Record
            EnglishWord : String
            FrenchWord : String
          End
```

Questions

2 Calculate `H(k)` for the following keys, `k`

(a) PEN (b) CAT (c) NOW (d) WON

(ASCII codes for the characters `'A'...'Z'` map to the range 65 ... 90 - see *Chapter 4*)

Programming tasks

1. Write a program to store English words and their French equivalents in a hash table which is an array or its equivalent with addresses in range 0 to 522. The English word and its French equivalent should be stored together in a record or equivalent data structure at an address which is calculated by the hashing function, `H`, described above. The table should be initialised so that every key field stores the string `'-1'` to indicate that this field's record has yet to be used to store an English-French word pair. Use your program to temporarily store the English words, PEN, CAT, NOW and their French equivalents.
(English word with its French equivalent:
PEN – PLUME, CAT – CHAT, NOW – MAINTENANT)

2. Extend your program so that after storing the English-French word pairs for PEN, CAT and NOW, the program uses the hashing function, `H`, to retrieve the French equivalent when the user enters PEN, CAT or NOW. Use a loop to enable the user to continue to look up the French equivalent until the user decides otherwise.

Collisions

The hash values calculated in Questions 2(c) and 2(d) are identical because the English words contain the same letters, but arranged in a different order (NOW and WON). So both words hash to the same address. This situation is known as a **collision**. Clearly, there is only space at this address for one English-French word pair.

Collisions can be resolved in two ways:

1. Store the record in the "next available" empty location in the table, or
2. Store a pointer in each table location that points to a list of records that have all collided at this table location, otherwise set the pointer value to null.

Key concept

Collision:
A collision occurs when two or more different keys hash to the same hash value. For the hash table this means a hash value of a location in the hash table which is already occupied.

Method 1 – closed hashing or open addressing

The first way of resolving a collision is to *rehash* which means to generate a new table row address at which to store the English-French word pair.
One rehash method, called **linear rehash**, calculates a new address by adding one to the original address before testing that the location with this new address is empty, e.g. indicated by `'-1'` in the `EnglishWord` field.

The rehash step may have to be repeated until an empty slot is found.

To avoid going off the end of the table, the new address is made to wrap around to the beginning of the hash table, if necessary and assuming there is an empty slot, by using modular arithmetic as follows:

```
Repeat
  Address ← (Address + 1) Mod 523
Until HashTable[Address].EnglishWord = '-1'
```

This method is an example of **closed hashing** or **open addressing** because other row addresses of the hash table are open to being used but access to addresses outside the hash table are closed off.

Key concept

Closed hashing or open addressing:
Method in which a collision is resolved by storing the record in the "next available" location.

The table, `HashTable`, is an array whose addresses run from 0 to 522.
The table is initialised with 523 empty English-French word pair records in which every `EnglishWord` field has the string `'–1'` stored in it to indicate that this field is unoccupied and the whole record is empty.
Table 34.20 shows an algorithm expressed in pseudo-code for inserting an English-French word pair into an initialised `HashTable`. The English word to insert is supplied in `WordInE` and its French equivalent in `WordInF`.
Each row of the hash table has space for a record with two fields, `EnglishWord` and `FrenchWord`.

```
Address ← Hash(WordInE)
If HashTable[Address].Key = '-1'
                  {-1 indicates field is empty}
  Then
    Begin
      HashTable[Address].EnglishWord ← WordInE
      HashTable[Address].FrenchWord ← WordInF
    End
  Else
    If Not(HashTable[Address].EnglishWord = WordInE)
                               {not already stored}
      Then
        Begin
          {find empty slot)
          Repeat
            Address ← (Address + 1) Mod 523
          Until (HashTable[Address].EnglishWord = '-1')
            Or (HashTable[Address].EnglishWord = WordInE)
                                      {already stored}
          If (HashTable[Address].EnglishWord = '-1')
              Then
                Begin
                  HashTable[Address].EnglishWord ← WordInE
                  HashTable[Address].FrenchWord ← WordInF
                End
        End
```

Table 34.20 Hashing algorithm incorporating a linear rehash that inserts an English-French word pair into a hash table

Clearly for this algorithm to work the hash table must have at least one empty row.

Searching for a specific record in a hash table accommodating collisions

Table 34.21 shows an algorithm expressed in pseudo-code that can be used to search for an English-French word pair in a hash table, `HashTable`, given an English word stored in the variable `WordInE`.
The English word may or may not be present in the hash table.
If it is, then its French equivalent is returned in variable, `Retrieve` otherwise message `'Not found'` is returned in `Retrieve`.
Clearly for this algorithm to work the hash table must have at least one empty row.

```
Address ←  Hash(WordInE)
Found ←  False
Repeat
  If HashTable[Address].EnglishWord = WordInE
    Then Found ←  True
    Else Address ←  (Address + 1) Mod 523
Until Found Or (HashTable[Address].EnglishWord = '-1')
If Found
  Then Retrieve ←  HashTable[Address].FrenchWord
  Else Retrieve ←  'Not found'
```

Table 34.21 Hashing algorithm that uses a linear rehash method to search a hash table for the French equivalent of a given English word

Setting up a hash table that uses closed hashing

Method 1 (closed hashing or open addressing) requires that the number of rows in the table exceeds by about a third the maximum number of records that will ever be stored in the table. When every record has been stored in the table, the table should still contain empty rows (i.e. table should never be more than roughly two thirds full). If this isn't the case then search times will be extended as will the time to insert new records.

Although this might seem a waste of storage space, there is a very good reason for working in this way. Studies have shown that the number of collisions depends on

- the hash keys
- the hash function
- the ratio of total number of records to total number of possible locations available to these records in the hash table.

A perfect hash function hashes all the hash keys to hash values without the occurrence of a single collision. That is why it is called perfect.

However, finding a perfect hash function is extremely difficult.

The effectiveness of a hash function is very sensitive to the hash key values. These are not always fully known in advance.

Using a ratio of roughly two thirds for total number of records to total number of hash table locations seems to minimise collisions for hash functions that are close to perfect. The hash table shown in *Table 34.22* has six student records and seven rows. One improvement could be to change the number of rows to 9 or even better, 11, a prime number. **Using a prime number for modulo arithmetic helps to minimise collisions**.

However, the hash function could be further improved as well as it is far from being perfect.

The aim is to make each hash value generated by the hash function equally likely when the function is applied to any of the possible hash keys, i.e. no one particular hash value should be more favoured than any other.

	ULN	Forename	Surname
0	34567876	Fred	Bloggs
1			
2	90002789	Mary	Smith
3	64156906	Alex	Black
4	24567805	Visha	Baal
5	74432167	Ben	Brown
6	90002985	Shena	Patel

Table 34.22 Hash table with not enough rows to minimise collisions

Questions

3 Copy and complete *Table 34.23*.

ULN	ULN Mod 7	ULN Mod 11
24567805		
34567876		
64156906		
74432167		
90002789		
90002985		

Table 34.23

Questions

4 Insert the ULNs from *Table 34.23* into a copy of the hash table shown in *Table 34.24* using the hashing function,

```
H(ULN) = ULN Mod 7
```

The student `Forename` and `Surname` fields do not need to be completed.
You should deal with any collision by performing a linear rehash until an empty slot is found.

	ULN	Forename	Surname
0			
1			
2			
3			
4			
5			
6			

Table 34.24 Hash table

5 Insert the ULNs from *Table 34.23* into a copy of the hash table shown in *Table 34.25* using the hashing function,

```
H(ULN) = ULN Mod 11
```

The student `Forename` and `Surname` fields may be omitted for convenience.

	ULN	Forename	Surname
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Table 34.25 Hash table

6 Explain how the hash table in *Table 34.25* when populated with student records would be used to look up the forename and surname of student with ULN = 24567805.

7 Explain how the hash table in *Table 34.24* when populated with student records would be used to look up the forename and surname of student with ULN = 24567805.

8 Why is it necessary to store the key field in a hash table even when an application using this hash table must already know the value of the key field?

Investigation

1. Devise an experiment to investigate collisions on a hash table that is to store 6000 student records. Use a random number generator to generate unique student ID numbers. Try different ratios of total number of records to total number of table rows in the hash table.
2. The hash function `H` that we have used so far is far from perfect for many data sets that we wish to store in a hash table. Investigate other hashing functions.

Method 2 - open hashing or closed addressing

In this alternative method of dealing with collisions, the hash table is extended to include a pointer field. The pointer field for each row is initialised to the null pointer value when the table is set up (⊣).

When a collision occurs the colliding record is linked to the corresponding table row by changing the pointer field of this row to point to the colliding record as shown in *Figure 34.29*.

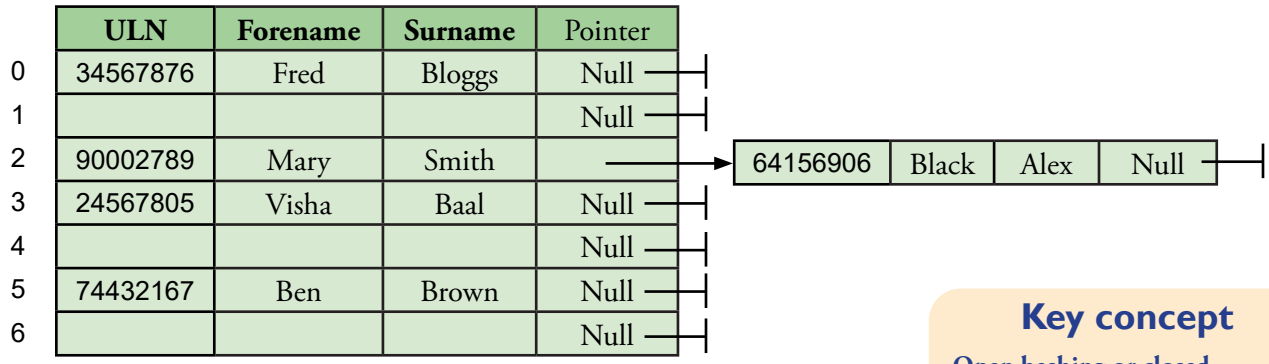


Figure 34.29 Hash table that uses open hashing

Another record colliding with row 2 will be linked or chained to the record of `'Mary Smith'` by changing the pointer field of Mary's record to point to this record and so on, thus forming a chain of linked records or a linked list.

Method 2 is called **open hashing** or **closed addressing** because locations outside the table are open for use by the hashing algorithm, i.e. the linked list locations, whilst other row addresses are closed off.

Key concept

Open hashing or closed addressing:
In a collision, the other rows of the hash table are closed to the colliding record which must, instead, be attached to the addressed table row in a chain or linked list of other colliding records. The table row uses a pointer field to point to the linked list.

Deleting a record

Care must be exercised when an entry in a hash table is deleted.

Closed hashing

In closed hashing, collisions are resolved by rehashing and storing the colliding record in another row whose table index is the rehash value.

However, if the entry at the original hash value table index or any of the rehash value table entries are deleted and the deleted entry remains empty, searching can be stopped prematurely before all potential matching entries have been examined.

Therefore, a deleted entry must be distinguishable from an entry that has never been used. This requires a special marker to be present in the key field part of the hash table entry when the entry is not in use. The special marker will use one value to indicate that this entry has never been used and a different value to indicate that it has been used but the entry has been deleted.

The special marker values should not use any value that potentially could occur in the key fields of the data set to be stored in the hash table. A search should now continue until an empty unused slot (indicated by the special marker) is encountered and not just an empty slot (which might have been used previously).

Open hashing

In open hashing, collisions are resolved by chaining the colliding record to the table entry slot whose index is the hash. Care must be taken when deleting the record in the table row when the row has a nonempty chain.

A special marker can be left in the key field to signal that there is at least one record in a chain (linked list) attached to the row so that a search does not fail to look at the chain when seeking a match.

There are at least two alternatives that do not rely on a special marker.

In alternative one, the search examines the pointer field of an empty slot to see if a chain is attached.

In alternative two, the first record in the chain is moved into the table slot whilst preserving its link to the rest of the chain.

Information

The definitions assigned to the terms closed hashing and open hashing have been interchanged over the years so care needs to be exercised when interpreting them. The key is to focus on concept/method not name and to make sure that you understand the former.

Questions

9 An empty hash table is set up for open hashing. The following hashing function is to be used to store variable names beginning with an uppercase letter in range A...Z, as well as other information.

`H(VariableName)=(code for first letter of VariableName x 11) Mod M`

Where `M` is the number of rows in the hash table.

Using `M = 5` and coding letters of the alphabet as follows, A=1, B=2, ..., Z=26 show the contents of the hash table after inserting the following variable names:

CHECK, OVERTIME, MAIN, P, URL, TAXRATE, INDEX, N, GENDER

You may ignore in your answer the other information associated with each variable name.

10 (a) Using the hashing algorithm expressed in pseudo-code below, calculate the hash value for the hash key `'PEN'` stored in string variable `Key`. You will need access to an ASCII code table to map characters to their equivalent ASCII codes. This is performed in the pseudo-code by the function `Ord`. The `Length` function returns the number of characters in the string. The symbol `'*'` means multiply.

```
Sum ← 0
For i ← 0 To Length(Key)- 1
  Sum ← Sum + Ord(Key[i]) * Ord(Key[i])
EndFor
HashValue ← Sum Mod 523
```

(b) Now repeat the exercise with the made-up word `'NEP'`.

(c) Can you see that there is a problem? What is the problem?

(d) Describe **two** ways that could be used to overcome this problem.

11 Explain why care must be exercised when deleting an entry in a hash table that uses closed hashing and on which searching occurs after deletion.

12 A person owns *n* distinct pairs of socks, which are kept in an unmatched pile in a drawer.

Individual socks are pulled from the drawer blindly, then identified and placed in a separate pile according to identity.

(a) How many individual socks must the person pull from the drawer to ensure that two are pulled that match?

(b) In what respect does this process resemble a hash table and open hashing?

Questions

13 In an application, student records are identified by their key field, the student's unique learner number (`ULN`) consisting of eight digits, e.g. 34567890. The application has to process a `ULN` allocated in the range 1000000 to 99999999 but it will never have to deal with more than 500 `ULN`s.

(a) Explain why when storing student records in a table in memory it would not be sensible to use the `ULN` as the row address for the record, e.g. 34128496.

(b) Explain why the use of a hash table would be a better option for this application.

14 (a) State **two** advantages of using hashing and the hash table approach over the alternative approach which just stores records in an ordinary table starting from the first row.

(b) It is noticed that after inserting many records into a hash table that uses closed hashing, searches are taking much longer than they did.

(i) Explain why this may be the case

(ii) Suggest a solution that could potentially restore searching times to what they were.

15 Explain why it is necessary to store the hash key in a hash table.