```
 1:  ************************************************
 2:   maXbox Starter 47
 3:  ************************************************
 4:
 5:
 6:  Work with real big RSA Cryptography
 7:  ----------------------------------
 8:  Max Kleiner
 9:
10:  //Zwei Worte werden Dir im Leben viele Türen öffnen - "ziehen" und "stossen".
11:
12:  A student of mine wrote a while ago asking me to clarify exactly how people used
    public key sharing to send encrypted messages. I had read a code a lot and thought
    I understood the concepts, but I cant explain it to with out a refresher.
13:
14:  In its simplest forms, an individual has public and private keys, each including
15:  of two values.
16:  For the public key the values are n of p*q, the so called "modulus", and E, a well
    known encrypting integer prime with the value: Const E = 65537;.
17:
18:  The private key values are also n, the same modulus that appears in the public key,
     and d, a big number which can decrypt any message encrypted using the public key.
19:
20:  There are obviously two cases:
21:
22:    1. Encrypting with the public key, and then decrypting with the private key.
23:       For a message or data
24:    2. Encrypting with the private key, and then decrypting with the public key.
25:       For a digital signature
26:
27:  Lets begin with a big real world example:
28:
29:  1. First we have to generate two very large primes, a(p) and a(q). These numbers
    must be random and not too close to each other. Here are the numbers that we
    generated by using Rabin-Miller primality tests:
30:
31:  Const
32:   ap =
33: '12131072439211271897323671531612440428472427633701410925634549312301964373042 08
34: '56193241973653224168665410170573613652141717117137979742993348710628298035 41';
35:   aq =
36: '12027524255478748885956220793734512128733387803682075433653899983955179850988 79
37: '78998691469008091316111533468170508320960221601463663463918124709871054152 33';
38:
39:   The number is of a RSA crypto system with 1024 bits (2^1024), so the big number n
    (modulus)
40:   has a length of 309:
41:
42:   maxcalcF('ln(2^512)/ln(10)') >>> 154.127357779958  //one prime
43:   maxcalcF('ln(2^1024)/ln(10)') >>> 308.254715559917 //modulus
44:
45:  2. Second with these two large prime numbers, we can calculate n and phi(n):
46:
47:  n = ap * aq
48:  phi(n) = (ap - 1) * (aq - 1)
49:
50:  The RSA generation technique depends on the fact that it is generally difficult to
    factor a number which is the product of two "large" prime numbers. This is the way
    that RSA determines the n value!
51:
52:  n is >>> writeln(bigMulu(ap,aq)) =
53:
54:   14590676800758332323018693934907063529240187237535716439958187101987343879 9005
55:   35893836957140267014980212181808629246742282815702292207674690654340122488 9672
56:   47240792696998710058129010319931785875366371086235765651050788371429711563 7342
57:   78891146353510271203276516651841172685983798867211183720508552634661874005 3
58:
```

```
 59:  phi(n) is >>> writeln(bigMulu(ap-1,aq-1)) =
 60:
 61:   14590676800758332323018693934907063529240187237535716439958187101987343879900 5
 62:   35893836957140267014980212181808629246742282815702292207674690654340122488964 8
 63:   31381123227996631730139777785236530154784827347887129722205858745715289160645 9
 64:   26971811926897116355507080264399952954964411681194751651393818429668352128 0
 65:
 66:  The Euler phi-function assigns to an integer n the number of numbers less than n
      and relatively prime to n. For example phi(12)= 4. Why?
 67:   The only numbers less than 12 which are relatively prime to 12 are 1,5,7,11 so
      phi(12) = 4.
 68:
 69:   writeln('EulerPhi '+itoa(GetEulerPhi(12)));
 70:   >>> 4
 71:
 72:   What about a prime as phi(11)?
 73:   Since 11 is prime, all the numbers less than 11 are relatively prime to 11, so
      phi(11) = 10.
 74:
 75:   writeln('EulerPhi '+itoa(GetEulerPhi(11)));
 76:   >>> 10
 77:
 78:  This can be checked by counting the number of numbers less than N which have a
      factor in common with N by Eulers Theorem of phi of n.
 79:
 80:  3. Third we have to find the private Key, the difficult part of RSA.
 81:  Euclid gives a process which we now call the Euclidean Algorithm. This process,
      essentially repeated long division, can be used (The Extended Euclidean Algorithm)
      to find the private key:
 82:
 83:  //Find d such that: e*d mod phi(n) = 1. (as inverse modulo)
 84:  function getPrivate: integer;
 85:  begin
 86:   for it:= 1 to big(1200) do //it(n) depends on bigint
 87:     if (getEPublic * it mod getPHI_N = 1) then begin
 88:       result:= it;
 89:       privKey:= result;
 90:       break;
 91:     end
 92:   end;
 93:
 94:  This multiplicative inverse is the private key [e*d=1 mod phi(n)]. The common
      notation for expressing the private key is d or in our case addpkey:
 95:
 96:  Const  addpkey = // - the private key -
 97: 894894250092744443682285459217730939196695860658842574454978544564876748396298 18
 98: 390934941973262879616797970608917283679875499331574161113854088813275488110588 24
 99: 719307758252727843790650401568062342355006724004246666565423238350292221549362 32
100: 894721388664458187891279461234078077257026266440910365023725451397 13';
101:
102:   Now we do the proof of the crypto pudding:
103:
104:   Find d such that: e*d mod phi(n)=1
105:
106:     Extended Euclidean Algorithm with real world example e*d mod phi(n)=1
107:
108:     writeln(modbig2(bigmulu(addpkey,65537),bigmulu(ap1,aq1)));
109:      >>> 1
110:
111:      Or with a smaller example:
112:      e*d mod phi(n)=1  >>> 7·103=721 mod 120 =1. [p=11 and q=13.] //invmod()
113:      maXcalcF('7 * 103 % 120') >>> 1
114:
115:  4. Fourth we are able to cipher data with:
116:
117:      encrypt: m^e mod n = 9^7 mod 143 = 48 = c;  //9 is the message
118:      decrypt: c^d mod n = 48^103 mod 143 = 9 = m;
```

```
119:      below: Crypto Systems Ref 2
120:
121:   Our big encrypt: m^e mod n, m= 19766202164023008896244827187751 50
122:
123:   writeln(modpowbig2('19766202164023008896244827187751 50',65537,bigmulu(ap,aq)))
124:   s_encrypt:= modpowbig2('19766202164023008896244827187751 50',65537,bigmulu(ap,aq));
125:   s_decrypt:= modpowbig2(s_encrypt,addpkey,bigmulu(ap,aq))
126:   writeln('decrypt: '+s_decrypt)
127:
128:   RSA is the single most useful applied concept for building cryptographic
129:   protocols.
130:
131:   Ref: http://www.softwareschule.ch/maxbox.htm
132:
133:   ..\examples\210_RSA_crypto_complete8hybrid.txt
134:   ..\examples\750_ibz_cryptomem_RSA_proof_64.txt
135:
136:     Crypto Systems Ref:
137:  1. Prime1: 13 and Prime2: 23
138:  2. RSA Modul Public [p*q] as N: 299
139:  3. phi(N) Private: 264 = 12*22
140:  4. RSA Exponent: 17
141:  5. Private D: 233
142:  6. Public (17,299) - Private (233,299) //}
143:
144:     Crypto Systems Ref 2:
145:  1. Prime1: 11  &  Prime2: 13
146:  2. RSA Modul Public [p*q] as N: 143
147:  3. Phi(N) Private: 120 = 10*12
148:  4. Public RSA Exponent: 7
149:  5. Private D: 103
150:  6. Public (7,143) - Private (103,143)
151:
152:   Lets choose two primes: p=11 and q=13.
153:   Hence the modulus is n=p×q=143. The totient of n phi(n)=(p-1)·(q-1)=120.
154:
155:  Doc:
156:   http://doctrina.org/How-RSA-Works-With-Examples.html
157:   http://www.delphiforfun.org/Programs/Math_Topics/RSA_KeyDemo.htm
158:   http://www.softwareschule.ch/download/maxbox_functions.txt
159:
160:   There are only 10 types of people: those who understand binary and those who
      don't.
161:
162:                 _od#HMM6&*MMMH::-_
163:           _dHMMMR??MMM? ""| `"'-?Hb_
164:        .~HMMMMMMMMHMMM#M?         `*HMb.
165:      ./?HMMMMMMMMMMM"*""""          &MHb.
166:     /'|MMMMMMMMMM'                -   `*MHM\
167:    /  |MMMMMMHHM''                    .MMMHb
168:   |   9HMMP   .Hq,                    TMMMMMH
169:  /     |MM\,H-""&&6\__                 `MMMMMMb
170: |       `""HH#,       \               - MMMMMMM|
171: |            `HoodHMM###.               `9MMMMMH
172: |              .MMMMMMM##\                 `*"?HM
173: |          ..  ,HMMMMMMMMMMo\.                 |M
174: |              |MMMMMMMMMMMMMMMHo              |M
175: |               ?MMMMMMMMMMMMMMM*              |H
176: |.               `#MMMMMMMMMMMM'              .M|
177:  \                  `MMMMMMMMMM*              |P
178:  `\                  MMMMMMMT"'              ,H
179:   `\                `MMMMMH?                ./
180:     \.              |MMMH#"                ,/
181:      `\.            |MMP'               ./'
182:        `~\          `HM:.-    .       ,/'
183:           "-\_        '_\ .      _.-"
184:               "-\-#odMM\_,oo==-"
```