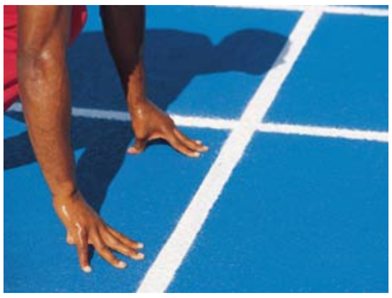
maXbox Starter 50

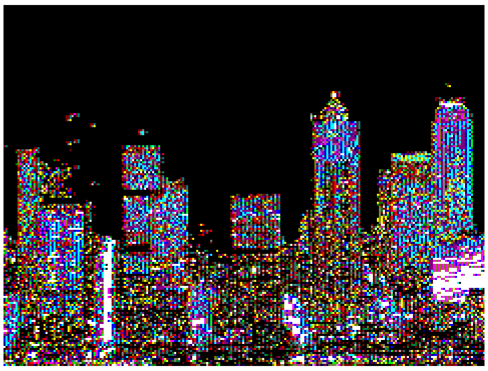


Work with Big Numbers II

1.1 Crypto Cipher Attack

Performing 5 billion multiplications on huge numbers, in one single thread isn't that easy. There are not many implementations of multi-precision BigInteger, BigDecimal or BigRational types, built from baseline in our programming tools.
Best of components and libraries use higher level algorithms like Burnikel-Ziegler, Karatsuba, Toom-Cook, mX4 etc. to make things fast and accurate even for very large integers.

We focus on a real world example from a PKI topic RSA to sign a public key included in a certificate. They usually operate with huge integers by multiplication of big primes and that's exactly what we want.
Each example in the following tutorial is shown in detailed small steps so it is easily reproduced in your own system behaviour. At first I show a simple solution to handle big numbers.



We know the `max` value for an `int64` is 9223372036854775807.
If we multiply 9223372036854775807 x 9223372036854775807 using the following routine we get:
85070591730234615847396907784232501249.

If we don't have a special routine we only get:

8.5070591730234616E37

And now you see the missing difference:

85070591730234615847396907784232501249
8.5070591730234616E37

I just drop the decimal point now:

85070591730234615847396907784232501249
85070591730234616E37

The whole number has a length of 38 and we lost 20 signs, left only 18. So we lose precision or need any rounding.

// big number ex.:

```
function AddNumStrings (Str1, Str2: string): string;
var
  i : integer;
  carryStr : string;
  worker : integer;
  workerStr : string;
begin
  Result:= inttostr (length(Str1));
  Result:= '';
  carryStr:= '0';
  // make numbers the same length
  while length(Str1) < length(Str2) do
    Str1:= '0' + Str1;
  while length(Str1) > length(Str2) do
    Str2:= '0' + Str2;
  i:= 0;
  while i < length(Str1) do begin
    worker:= strtoint(copy(Str1, length(str1)-i, 1)) +
              strtoint(copy(Str2, length(str2)-i, 1)) +
              strtoint (carryStr);
    if worker > 9 then begin
      workerStr:= inttostr(worker);
      carryStr:= copy(workerStr, 1, 1);
      result:= copy(workerStr, 2, 1) + result;
    end else begin
      result:= inttostr(worker) + result;
      carryStr:= '0';
    end;

    inc(i);
  end; { while }
  if carryStr <> '0' then
    result:= carryStr + result;
end;
```

This function will add two numbers that are represented by their string digits, so we use strings as the representation of big integers.

Without that as a we said, we don't get the whole number:

```
    i64e:= 9223372036854775807
     PrintF('mulu %18.f ',[i64e*i64e])
     >>> 8.50705917302346158E37
```

How does it work?
Go back to the basics and calculate the answer the same way you would if you were doing it with pencil and paper. Use string variables to hold the text representation of your numbers as digits and create functions that will add and multiply those number strings. You already know the algorithms, you learned it as a (script) kid!
Nowadays kids code a lot of with sensor kits with Arduino or Raspy, I call them sensor kids;-).

Next example shows the multiplication with this school concept:

```
function MultiplyNumStrings (Str1, Str2 : string): string;
var
  i,j : integer;
  worker : integer;
  carryStr, workerStr, tempResult: string;
begin
  Result:= '';
  carryStr:= '0';
  tempResult:= '';
  // process each digit of str1
  for i:= 0 to length(Str1) - 1 do begin
    while length(tempResult) < i do
      tempResult:= '0'+ tempResult;

    // process each digit of str2
    for j:= 0 to length(Str2) - 1 do begin
      worker:= (strtoint(copy(Str1, length(str1)-i, 1)) *
                 strtoint(copy(Str2, length(str2)-j, 1)))+
                strtoint (carryStr);
      if worker > 9 then begin
        workerStr:= inttostr(worker);
        carryStr:= copy(workerStr, 1, 1);
        tempResult:= copy(workerStr, 2, 1)+ tempResult;
      end else begin
        tempResult:= inttostr(worker)+ tempResult;
        carryStr:= '0';
      end;
    end; { for }
    if carryStr <> '0' then
      tempResult:= carryStr + tempResult;
    carryStr:= '0';

    result:= addNumStrings (tempResult, Result);
    tempResult:= '';
  end; { for }

  if carryStr <> '0' then
    result:= carryStr + result;
end;
```

But performing billions multiplications on huge numbers, in one single thread? Unless you've got a state-of-the-art over-clocked CPU cooled with liquid fluid helium, you'd have to wait a whole lot for this to complete and finish. However if you do have, you'd just have to wait for a very long time with a sort of brute force attack.

Each algorithm has a benchmark and also in our school example there's some room to improve. You should replace `strtoint(copy(S, Index, 1))` with `Ord(S[1])-Ord('0')` since all of the characters are '0'..'9', and replace any remaining `copy(S, Index, 1)` with `S[Index]`, which then allows some places, like `carryStr`, to replace String with Char.

Now we can get the real int64:

```
writeln(bigPow(2,64))
```

> 18446744073709551616

not only

```
 maxcalcF('2^64')
 >>> 1.84467440737096E19
```

Big Decimal is equally built for ease of use and reliability. It builds on top of Big Integer so the internal representation is a Big Integer for the significant digits before decimal point, and a scale to indicate the decimals after decimal point.

The most important aspect that is emphasized in this example is that you should apply big numbers when you need to get added precision or really large numbers, that means, if you don't understand the impact of the resulting calculation the most of the time a rounded or truncated result is good enough in your own code or when you add a new feature or fix a bug then you can switch to a big numb library:

```
writeln(bigMulu('9223372036854775807','9223372036854775807'))

writeln(MuluNumStrings('9223372036854775807','9223372036854775807'))
```

```
procedure getUTCTime;
var
  s: string;
  lHTTP: TIdHTTP;
  lReader: string; //TStringReader;
begin
  lHTTP:= TIdHTTP.Create(nil);
  try
    lReader:= lHTTP.Get('http://tycho.usno.navy.mil/cgi-bin/timer.pl');
    //while lReader.Peek > 0 do begin
      if Pos('UTC',lreader) > 0 then
        Writeln(lreader);
  finally
    //lhttp.close;
    lHTTP.Free;
    //lReader.Free;
  end;
 end;
```

1.2 The RSA Crypt System

Next real word example is the following teaching crypt system:

```
Ref of Herdt P.119 Version 10: Network Security

1. Modulus Prime1: 23  &  Prime2: 47
2. RSA Modul Public [p*q] as N: 1081 //Modulus
3. Phi(N) Private: 1012 = 22*46
4. Public RSA Exponent: 3
5. Private D: 675
6. Public (3,1081) - Private (675,1081)
```

What we want in the end is to sign a message or a hash, whatever, with that simple two functions:

```
  cipher c[i]    = m[i]^e (mod N)
  decipher m[i] = c[i]^d (mod N)
```

What we do next is simple, we encrypt the letter 'A' as `ASCII 65` with the following public key of `Public (3,1081)`:

```
    writeln(RSAEncrypt('65','3','1081'))
>>> 51
   writeln(RSADecrypt('51','675','1081'));
>>> 65
```

What we also do is more explicit with RSA:

```
 writeln('RSA cipher message:')
m:= 'A';
c:= RSAEncrypt(intToStr(ord(m[1])),'3','1081')
writeln(c)
writeln(RSADecrypt(c,'675','1081'))

writeln('RSA sign message:')
m:= 'A';
c:= RSAEncrypt(intToStr(ord(m[1])),'675','1081')
writeln(c)
m:= RSADecrypt(c,'3','1081')
writeln(chr(strToInt(m)))
```

As you can see we convert the string A to a ASCII number 65 and back to string, cause the big numb library handles only strings as numbers.

For this crypt system to work, the system must guarantee that it is (effectively) impossible to decrypt the message without knowledge of the private key. In particular, it must be impossible to decrypt using the public key, or to derive the private key from the public key.

And here's the whole implementation step by step, especially to find the private key D[1]:

```
//1. Init - Choose two prime numbers p and q and e.
Prime1:= 23           //13;  //7;  //11;
Prime2:= 47           //23;  //11; //7;
RSA_Exponent:= 3;     //5;   //17; //5;

PrintF('1. Modulus Prime1: %d  &  Prime2: %d',[prime1, prime2])
//2. Let n = p*q.
Writeln('2. RSA Modul Public [p*q] as N: '+inttostr(setPublic))

//3. less than n with no factors in common to n phi(n)=(p-1)(q-1)
PrintF('3. Phi(N) Private: %d = %d*%d',
                              [getPHI_N,prime1-1,prime2-1])

//4. Choose e <n; e is relative prime to phi(n).
Writeln('4. Public RSA Exponent: '+inttostr(getEPublic))

//5. Find d such that e*d mod phi(n)=1.
Writeln('5. Private D: '+inttostr(getPrivate))

//6. Public key is (e,n), private key is (d, n).
PrintF('6. Public (%d,%d) - Private (%d,%d)',
         [getEPublic,setPublic,getPrivate,setPublic])
Writeln('');
```

[1]OpenSSL actually produces a public - private key pair.

```
function setPublic: integer;
begin
  result:= prime1 * prime2;
end;

function getPHI_N: integer;
begin
  result:= (prime1-1) * (prime2-1);
end;

function getEPublic: integer;
begin
  result:= RSA_exponent; //round(Power(2,4)+1);
end;

//Find d such that: e*d mod phi(n) = 1. (as inverse modulo)!
// more than one d is possible
function getPrivate: integer;
begin
  for it:= 1 to 1000 do //or n depends on bigint
    if (getEPublic * it mod getPHI_N = 1) then begin
      //writeln(itoa(it))
      result:= it;
    end
end;
```

Explanation:
Alice lets her **public** key be known **to** everyone, but keeps the **private** key secret. Bob may send a confidential **message to** Alice like this:

```
1. B gets A's public key (you can get it from web).
2. B encrypts a message with A's public key, and sends it.
3. A decrypts a message with her private key.
```

You see in our example the public and private is:

```
Public (3,1081) - Private (675,1081)
```

At least we take a look at the big numbers.
Generating the public and private keys.
Pick two large prime numbers, p and q. Let n=pq. Typically, n is a number which in binary is written with 1024 bits (in decimal, that's about wide 308 digits).
Pick e relatively prime to (p-1)(q-1) like Euler said. Now find d such that ed=1 mod (p-1)(q-1)[2]. You can use Euclid's algorithm to find this d. The pair of numbers (e, n) is the public key. The pair of numbers (d, n) is the private key. The two primes p,q are no longer needed, and can be

[2] e*d mod (p-1)(q-1) = 1

discarded, but should never be revealed. E is most of the time the same number: Exponent (24 bits): 65537

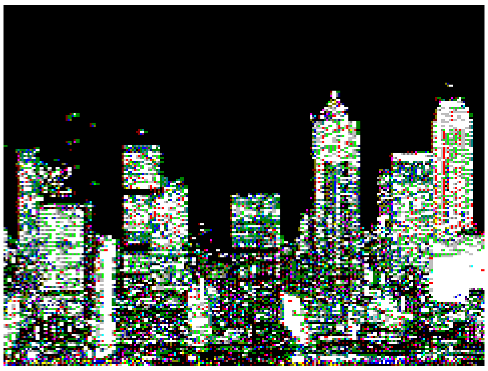
```
procedure encryptMessage(amess: string; acipher: TStrings);
var k,l,i: integer; //int64;
    pst: string;
begin
  for i:= 1 to length(amess) do begin
    l:= ord(amess[i])-OFFSET;
    //write(inttostr(l)+ 'd ') //debug
    pst:= PowerBig(l,getEPublic);
    k:= modbig(pst,setPublic);
    acipher.add(intToStr(k))
  end;
end;

function decryptMessage(alist: TStrings): string;
var k,i: int64;
    pst: string;
begin
  for i:= 0 to alist.count -1 do begin
    pst:= PowerBig(strtoint(alist[i]),getPrivate);
    k:= modBig(pst,setPublic);
    //k:= f mod setPublic;
    result:= result+ Chr(k+OFFSET);
  end;
end;
```

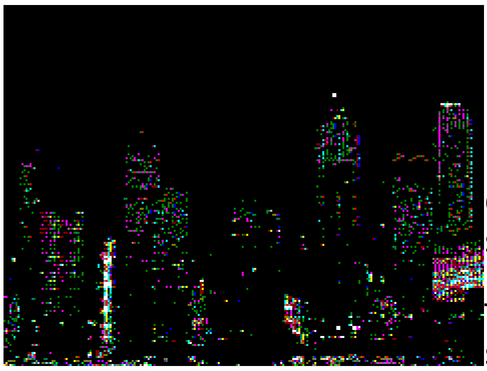
We need `powerBig` and `modBig` to deal with large numbers.
Conclusion:

```
The public key is (e,n), private key is (d, n).
Med mod n = M (this holds if e*d mod phi(n) = 1)
C=Me mod n
Me=C mod n
```

In real the message is divided into blocks, each block corresponding to a number less than n. For example, for binary data, the blocks will be (log2 n) bits. An exponentiation cipher (e) utilizing Euler's Theorem.



1.3 Beyond Big Numbers

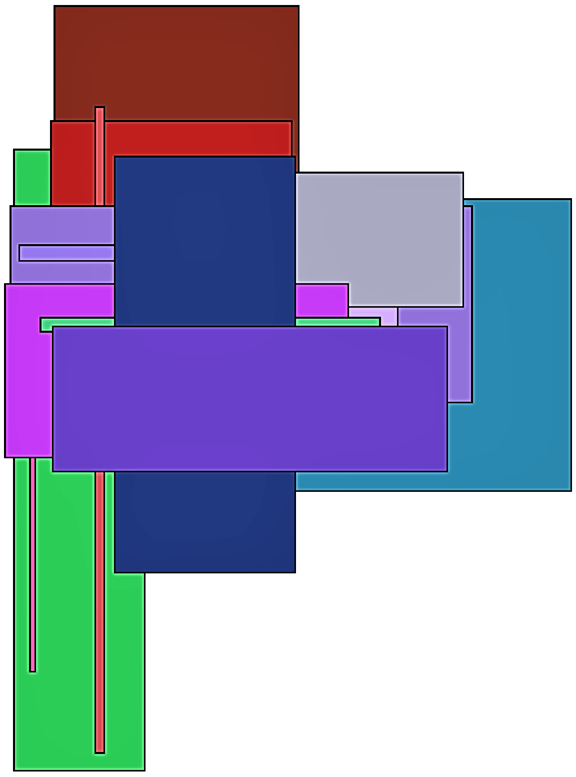
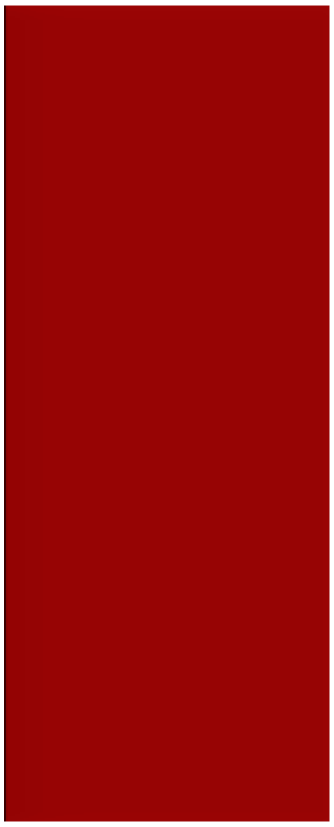


Challenging domains such as robot-assisted search and rescue, operations in space require humans to interact with robots. These interactions may be in the form of supervisory control, which connotes a high human involvement with limited robot automation (e.g., semi-autonomy, where the robot is truly autonomous for portions of the task or mixed-initiative systems, where the robot and human are largely interchangeable.

The interactions between humans and robots are often interchangeably referred to as “coordination” or “collaboration”.
The application of a script to interleave human and robot coordination is both logical and natural. Scripts simplify the relationship between human and robot making the task comprehensible to both novice and expert system users.

```
procedure StartBtnClick(Sender: TObject);
{User clicked start}
var  i:integer;
  drawtime:integer;
  startcount,stopcount:int64;
begin
  for i:= 2 to count do begin
    {put center of robo on the point}
    Robo.left:= saved[i].x-Robo.width div 2;
    Robo.top:= saved[i].y-Robo.height div 2;
    queryperformanceCounter(startcount);{Get time before we repaint}
    application.processmessages;
    queryPerformanceCounter(stopcount);{Get time after repaint}
    drawtime:= (stopcount-startcount) div freq;  {Compute ms to repaint}
    writeln('test freq ms: '+itoa(drawtime))
    sleep(max(0,sleepms-drawtime));   {wait whatever time is left, if any}
  end;
end;
```

The ability to simplify a task as a series of simple steps is necessary for this comprehension. The available or possible actions for the human operator at any particular time are clear in the script because of the GUI. This approach can be applied to any task, level of manmach coordination.

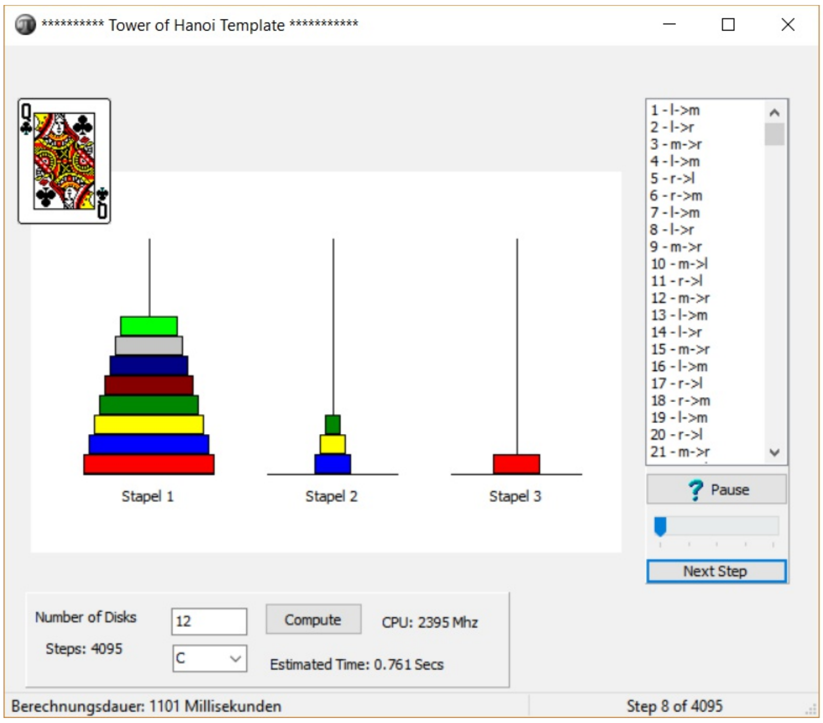
Due to the highly proprietary nature of robot software, most producers of robot hardware also provide their own software and firmware. While this is not unusual in other automated control systems, the lack of standards of programming methods for robots does pose certain challenges.

For example, there are over 30 different manufacturers of industrial robots, so there are also 30 different robot programming languages required.
Fortunately, there are enough similarities between the different robots that it is possible to gain a broad-based understanding of robot programming without having to learn each manufacturer's proprietary language.

Another interesting approach is worthy of mention. All robotic applications need or explore parallelism and event-based programming.
Robot Operating System is an open-source platform for robot programming using Python, CPascal and C++. Java, Lisp, Lua, Pascal and Pharo are supported but still in experimental stage.

https://en.wikipedia.org/wiki/Robot_Operating_System

Another example is the tower of hanoi which can be solved for example by Lego mindstorm or other Arduino frameworks and can produce large numbers to resolve:



********** Tower of Hanoi Template ***********

```
1 - l->m
2 - l->r
3 - m->r
4 - l->m
5 - r->l
6 - r->m
7 - l->m
8 - l->r
9 - m->r
10 - m->l
11 - r->l
12 - m->r
13 - l->m
14 - l->r
15 - m->r
16 - l->m
17 - r->l
18 - r->m
19 - l->m
20 - r->l
21 - m->r
```

Number of Disks: 12
Steps: 4095
C
CPU: 2395 Mhz
Estimated Time: 0.761 Secs
Berechnungsdauer: 1101 Millisekunden
Step 8 of 4095

The script is called: `examples\712_towerofhanoi_animation.pas`

Programming errors represent a serious safety consideration, particularly in large industrial robots. The power and size of industrial robots mean they are capable of inflicting severe injury if programmed incorrectly or used in an unsafe manner. Due to the mass and high-speeds of industrial robots, it is always unsafe for a human to remain in the work area of the robot during automatic operation.
Two concepts should improve this safety:

- Console Capture
- Exception Handling

1.4 Console Capture DOS Big Decimals

I'm trying to move a part of SysTools to Win64. There is a certain class `TStDecimal` which is a fixed-point value with a total of 38 significant digits. The class itself uses a lot of ASM code.

```
function BigDecimal(aone: float; atwo: integer): string;
begin
  with TStDecimal.create do begin
    try   //assignfromint(aone)
      assignfromfloat(aone) //2
      RaiseToPower(atwo)  //23
      result:= asstring
    finally
      free
    end;
  end;
end;
```

But then I want to test some Shell Functions on a DOS Shell or command line output. The code below allows to perform a command in a DOS Shell and capture it's output to the maXbox console. The captured output is sent “real-time” to the `Memo2` parameter as console output in maXbox:

```
    srlist:= TStringlist.create;
      ConsoleCapture('C:\', 'cmd.exe', '/c dir *.*',srlist);
      writeln(srlist.text)
    srlist.Free;
```

But you can redirect the output `srlist.text` anywhere you want.
For example you can capture the output of a DOS console and input into a textbox, or you want to capture the command start of demo app and input into your app that will do further things.

```
 ConsoleCapture('C:\', 'cmd.exe', '/c ipconfig',srlist);
 ConsoleCapture('C:\', 'cmd.exe', '/c ping 127.0.0.1',srlist);
```

☝ It is important to note that some special events like `/c java -version` must be captured with different parameters like /k or in combination.

Here's the solution with `GetDosOutput()`:

```
writeln('GetDosOut: '+GetDosOutput('java -version','c:\'));
```

or like powercfg or the man-pages in Linux

```
writeln('GetDosOut: '+GetDosOutput('help dir','c:\'));
 GetDosOutput('powercfg energy -output
                         c:\maxbox\osenergy.htm','c:\')
```

1.5 Exception Handling

A few words how to handle Exceptions within maXbox:
Prototype:

```
procedure RaiseException(Ex: TIFException; const Msg: String);
```

Description: Raises an exception with the specified catch message.
Example:

```
begin
  RaiseException(erCustomError,'Your message goes here');
// The following line will not be executed because of the
exception!
  MsgBox('You will not see this.', 'mbInformation', MB_OK);
end;
```

This is a simple example of a actual script that shows how to do try except with raising a exception and doing something with the exception message.

procedure Exceptions_On_maXbox;

```
var filename,emsg:string;
begin
   filename:= '';
   try
     if filename = '' then
       RaiseException(erCustomError,
                    'Exception: File name cannot be blank');
   except
     emsg:= ExceptionToString(ExceptionType,ExceptionParam);

       //do act with the exception message i.e. email it or
       //save to a log etc

     writeln(emsg)
   end;
end;
```

☝ The `ExceptionToString()` returns a message associated with the current exception. This function with parameters should only be called from within an except section.

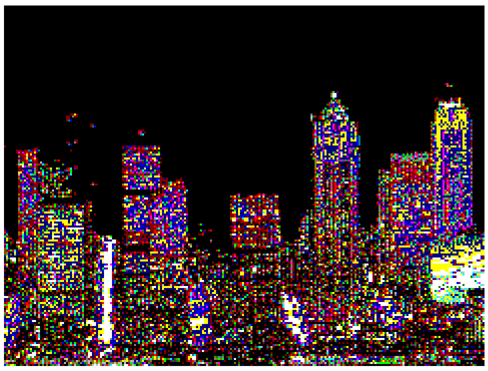
After completing the tasks as directed, the compiler proceeds to its second step where it checks for syntax errors (violations of the rules of the language) and converts the source code into an object code that contains machine language instructions, a data area, and a list of items to be resolved when he object file is linked to other object files.

At least there are two ways to install and configure your box into a directory you want. The first way is to use the unzip command-line tool or IDE, which is discussed above.
That means no installation needed. Another way is to copy all the files to navigate to a folder you like, and then simply drag and drop another scripts into the /examples directory.
The only thing you need to backup is the ini file **`maxboxdef.ini`** with your history or another root files with settings that have changed.

A namespace is a logical grouping of types (mostly to avoid name coll-isions). An assembly can contain types in multiple namespaces (`System.DLL` contains a few...), and a single namespace can be spread across assemblies (e.g. `System.Threading`).

- An assembly provides a fundamental unit of physical code grouping.
- A namespace provides a fundamental unit of logical code grouping.

1.6 Crypto Libraries with BigInt



With the namespace `System.Security.Cryptography` you do have access to huge numbers too:

```
System.Object
  System.Security.Cryptography.SymmetricAlgorithm
    System.Security.Cryptography.Aes
      System.Security.Cryptography.AesCryptoServiceProvider
```

This overview provides a synopsis of the encryption methods and practices supported by the .NET Framework, including the ClickOnce manifests, Suite B, and Cryptography Next Generation (CNG) support introduced in the .NET Framework 3.5.

This overview contains the following sections:

- Cryptographic Primitives
- Secret-Key Encryption
- Public-Key Encryption
- Digital Signatures
- Hash Values
- Random Number Generation
- ClickOnce Manifests
- Suite B Support
- Related Topics

You simply put the line above on the boot script and make sure the ini file has it set to Yes. BOOTSCRIPT=Y //enabling load a boot script

“Wise men speak: Hand made by robots.

Feedback @ max@kleiner.com
Literature: Kleiner et al., Patterns konkret, 2003, Software & Support

https://github.com/maxkleiner/maXbox4/releases

https://en.wikipedia.org/wiki/Robot_software

EKON 21 Cologne 2017 Input for Robots, Big Numbers & Components

1. AsyncPRO,	2. BigInteger,
3. Hotlog,	4. WMILib,
5. StBarCode,	6. XMLUtils,
7. LockBox,	8. cX509Certificate,
9. OpenGL,	10. WaveUnit,
11. OpenSSL,	12. Kronos,
13. HiResTimer,	14. Kmemo,
15. BigDecimals,	16. SynEdit,
17. SFTP,	18. Sensors,
19. PasScript,	20. ALJSON
21. CryptographicLib	22. RSA_Engine
23. cHugeInt.pas	24. cCipherRSA;

1.7 Appendix External links of RSA & Big Numbers

- "*The Basics - Robot Software*". Seattle Robotics Society.
- G.W. Lucas, "*Rossum Project*".
- "*Mobile Autonomous Robot Software* (MARS)". Georgia Tech Research Corporation.
- "*Tech Database*". robot.spawar.navy.mil.
- Adaptive Robotics Software at the Idaho National Laboratory
- A review of robotics software platforms Linux Devices.
- ANSI/RIA R15.06-1999 American National Standard for Industrial Robots and Robot Systems - Safety Requirements (revision of ANSI/RIA R15.06-1992)

https://www.academia.edu/31097592/Work_with_RSA_maxbox_starter47.pdf

http://www.softwareschule.ch/download/maxbox_starter47.pdf

https://www.academia.edu/31112544/Work_with_microservice_maXbox_starter48.pdf

1.8 References

Examples of Tutorial:

www.softwareschule.ch/examples/210_public_private_cryptosystem5_ibz_herdt2.txt

Examples of Big RSA Chain Routines:

http://www.softwareschule.ch/examples/750_RSA_Toolproof4.txt

http://www.rvelthuis.de/programs/bigintegers.html http://www.delphiforfun.org/programs/Library/big_integers.htm

http://stackoverflow.com/questions/6892937/huge-number-in-delphi

OpenSSL explain:

http://stackoverflow.com/questions/5244129/use-rsa-private-key-to-generate-public-key

- *O. Nnaji, Bartholomew. Theory of Automatic Robot Assembly and Programming (1993 ed.). Springer. p. 5. ISBN 978-0412393105. Retrieved 8 February 2015.*
- *"Robot programming languages". Fabryka robotów. Retrieved 8 February 2015.*
- https://www.madboa.com/geek/openssl/#key-rsa

```
             _od#HMM6&*MMMH::-_
          _dHMMMR??MMM? ""|  `"'-?Hb_
       .~HMMMMMMMMHMMM#M?        `*HMb.
     ./?HMMMMMMMMMMM"*"""           &MHb.
    /'|MMMMMMMMMMM'             -   `*MHM\
   /  |MMMMMMHHM''                   .MMMHb
  |   9HMMP   .Hq,                   TMMMMMH
 /     |MM\,H-""&&6\__               `MMMMMMb
|       `""HH#,       \             - MMMMMMM|
|           `HoodHMM###.              `9MMMMMH
|              .MMMMMMMM##\             `*"?HM
|         ..  ,HMMMMMMMMMMMo\.              |M
|             |M'MMMMMMMM'MMMMMHo           |M
|              ?MMM'MMMMMMM'MMMM*           |H
|.              `#MMMM'MM'MMMMM'           .M|
 \                `MMMMMMMMMMM*            |P
 `\                MMMMMMMMT"'            ,H
  `\              `MMMMMMH?              ./
    \.            |MMMH#"               ,/
     `\.          |MMP'               ./'
       `~\        `HM:.-    .       ,/'
          "-\_       '_\   .      _.-"
             "-\-#odMM\_,oo==-"
```

Zwei Worte werden Dir im Leben viele Türen öffnen - "ziehen" und "stossen".