



HexLicense Components VCL edition

© 2016 Lauritz Computing

HexLicense Components

License management for Embarcadero Delphi

by Lauritz Computing

All programmers that sell their software will sooner or later need to add some security to their applications; but even more important is to keep track of what serial numbers belongs to what customer. After all, a customer has invested in your product and the serial number is what secures access to the functionality you provide.

HexLicense Components VCL edition

© 2016 Lauritz Computing

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Printed: oktober 2016 in vestfold, Norway

Publisher

Lauritz Computing

Managing Editor

Linda Lauritzen

Technical Editors

Jon Lennart Aasenden

Glenn Dufke

Production

Jon Lennart Aasenden

Special thanks to:

All the people that helped make this manual and helped out testing the product. Especially Glenn who tested and expanded on the Firemonkey edition.

Also, Linda at Lauritz Computing who have followed the process from the beginning. Thank you going over the work.

To our customers who keeps giving us positive feedback, tips and suggestions for new versions.

Table of Contents

Foreword

Part I Introduction 2

- 1 Compatibility 3
- 2 License management 4
- 3 Piracy 4

Part II Getting started 6

- 1 Generating a root key 6
- 2 Generating serial numbers 7
- 3 Preparing your online store 8
 - Root-Key constant 8
 - Exporting serial numbers 8
 - Save your key to disk 9

Part III The Delphi side of things 11

- 1 Selecting a storage mechanism 12
- 2 Connecting the root-key 12
- 3 Storage events 13
 - ItDayTrial 13
 - ItFixed 13
 - ItRunTrial 13

Part IV A Delphi example 15

- 1 The code 16

Part V Using HexLicense server-side 21

- 1 Using HexLicense to build a serial server 21
- 2 Keeping track of licenses 21
- 3 The startup sequence 21
- 4 Multiple use of the same license 22

Part VI The components 24

- 1 THexLicense 24
 - Events 24
 - WarnDebugger..... 24
 - OnLicenseObtained..... 24
 - OnBeforeLicenseLoaded..... 24
 - OnAfterLicenseLoaded..... 25
 - OnLicenseBegins..... 25
 - OnLicenseExpires..... 25
 - OnBeforeLicenseUnLoaded..... 25

OnAfterLicenseUnLoaded	25
OnLicenseCountDown	25
Properties	25
Automatic	25
License	26
Duration	26
Provider	27
Software	27
FixedStart	27
FixedEnd	27
Storage	27
SerialNumber	28
Runtime properties	28
Active	28
SerialKey	29
LicenseState	29
LastExecuted	29
DurationLeft	30
Bought	30
Methods	30
BeginSession	30
EndSession	31
Execute	32
Buy	33
2 THexSerialNumber	33
Methods	33
GetSerial	33
Validate	34
Clear	34
Spin	34
3 THexSerialMatrix	35
4 THexCustomLicenseStorage	36
Properties	36
Events	36
OnDataExists	36
OnReadData	37
OnWriteData	37
Encryption	37
CanEncode	37
EncodeData	38
DecodeData	38
How does RC4 work	38
5 THexFileLicenseStorage	39
6 THexRegistryStorage	39
7 THexOwnerStorage	40
8 THexSerialGenerator	40
Methods	40
Events	40
Part VII Number engines	42
1 Linear vs. non linear	42

2	Number engines and matrixes	43
3	Exploring number sequences	44

Part VIII HexTools 50

1	Using HexTools	50
2	Class declaration	50
3	Getting an instance	51
4	Methods	51
	DiskSerial	51
	MacAddress	52
	IPAddress	52
	CalcCRC	53
5	Properties	53
	Failed	53
	LastError	54

Part IX More useful code 56

1	Bios identifier and disk serial	56
2	Local IP adresse	57
3	Network adapter MAC adresse	57

0

Foreword

License management is first of all about the customer, not the hacker. License management is about keeping track of what product a customer has bought, the duration of such a license, trial types and building up a customer relationship.

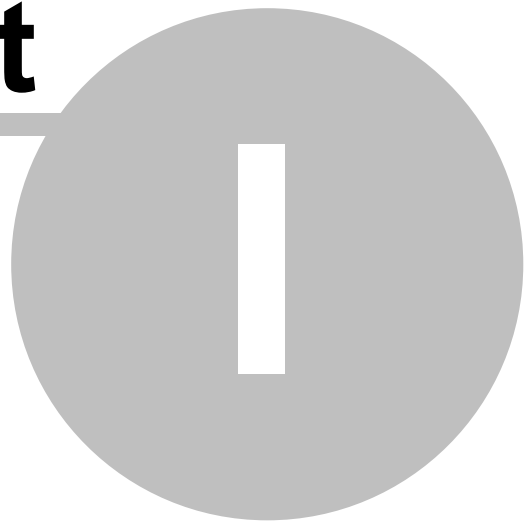
Hexlicense is not just another key generator. It doesn't simply play around with letters or encode a customers name into a fancy hex string. Those solutions are trivial at best and dangerous at worst. Hexlicense uses a technique not unlike certificates key chains (but much easier to use).

With HexLicense you generate licenses from a root key, which is an array of 12 random bytes (you can also input whatever values you wish). From that key the license generator application can generate thousands of unique serial numbers, serial numbers which can only be mathematically validated with that exact root-key.

HexLicense

License management for
Embarcadero Delphi programmers

Part



1 Introduction

You have a killer application and want to get it onto the marketplace. But how will you deal with serial numbers? How will you handle trial periods? How will you generate serial numbers in bulk, which online vendors need in order to retail your product?

This is where our license software comes in. Turning your application into a trial version is now a matter of adding some components, setting a few properties – and then deciding what to disable in your code during the trial. With plenty of events and straight forward mechanisms, the license manager component package will save you a lot of time!

Our components are all about generating serial numbers, validation and management of trial periods from within your Delphi application. How you store the license data (file and registry storage components ship with the package) is up to you. If you want to handle the license data yourself, just drop a THexOwnerLicenseStorage component and take charge via the event handlers.

This model gives you the benefit of not having to deal with the low-level stuff (like encryption, data signatures, date and time checking, file validation etc). You also have ample freedom to expand how data is managed, so you are not boxed into “our way of doing things”.

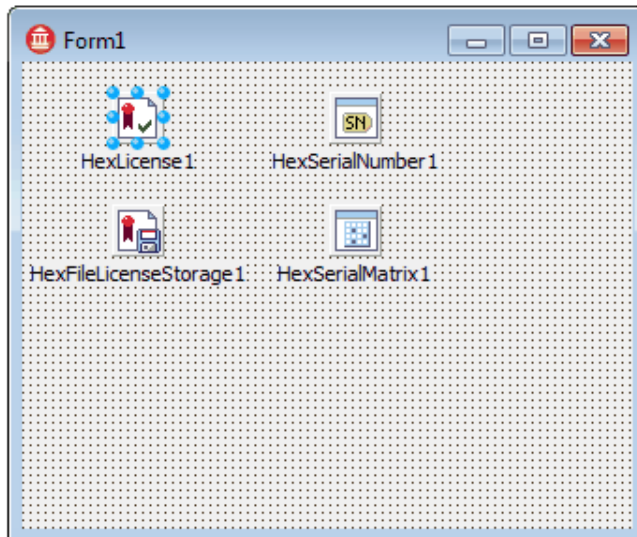
If you have a custom server solution or ordering system online, you can easily extend serial-number validation by calling your server to check usage. You can also use the license generator to create as many serial numbers (based on a root key) as you wish. These lists can be uploaded to all major merchant systems, making sure each customer is assigned a single, unique serial number on purchase.



Forgetting to factor in license management for your software is very common

1.1 Compatibility

The HexLicense component package is written in pure object pascal, no assembler or external libraries are used in the codebase. It is compatible with all versions of Delphi from version 5 through XE Berlin and beyond, both 32 and 64 bit. The package is designed to be platform independent.

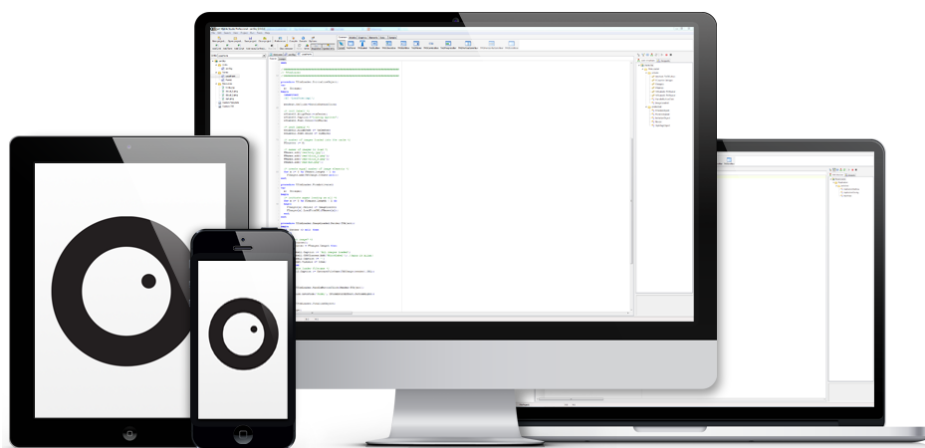


Lazarus and freepascal

We have customers that use Lazarus and have been given permission to adapt the sourcecode for freepascal. Hexlicense is also used on Android and iOS, although it makes little sense for Apple mobile devices -which uses license provisioning by default.

Smart Mobile Studio

HexLicense has also been implemented as a Node.js server solution, where the generation and validation of licenses are dealt with server-side. This solution was written in Smart Mobile Studio, which compiles object pascal to pristine JavaScript - with all the benefits of classes, inheritance and polymorphism we are accustomed to.



1.2 License management

License management is first of all about the customer, not the hacker. This is a very important distinction to make.

HexLicense helps you to protect your software from casual piracy, in fact - a hacker would have to disassemble your program in order to bypass the system.

It is important to recognize that HexLicense was designed to give you, the developer, the tools you need to deal with license keys and trial functionality. It was designed to keep ordinary customers from abusing your licenses, but focus is first and foremost on giving the developer a uniform way of linking customers to their purchases.

1.3 Piracy

There is no such thing as an un-crackable product. If a company makes such a claim I would avoid them at all cost, because lying about something so fundamentally incorrect is simply fraud.

Microsoft spent millions trying to come up with a scheme to make Windows “un-breakable”, yet only hours after Windows 7 was released – a fully working cracked pirate version could be downloaded from the internet. Same with Windows 10 and all previous version of their operative system.

The same goes for huge corporations like Adobe, Apple and all the others. The open-source tools for disassembling and reverse-engineering software is just as evolved as the products to compile and build software.

If you are going to put your trust in a company or security product, at least make sure they are honest and genuinely interested in buying you time, which is what serial number protection is all about.

In short, serial number protection gives you the following advantages:

- Non technical users are stopped from abusing their license
- Serious customers recognize the value of a proper license
- By carefully planning your release cycle and the use of keys, you can minimize loss even if your product is hacked. Only the hacked build and its root-key will have been compromised. By generating new root-keys and keeping track of these and their serial-numbers in a database, the damage caused by piracy can be greatly reduced.

There are several steps you can take to make your product more time consuming to break, like refusing to run the application if a system-wide debugger is running and also to disable memory dumps. These steps are publicly known, and while effective – they wont stop the most experienced hackers from getting at your code. Nothing will. Thats just reality. We would be lying if we told you otherwise.

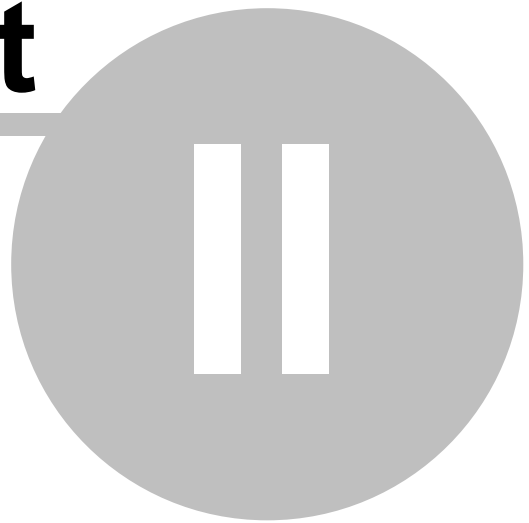
But at least with HexLicense you wont make it easy for them, and it will stop ordinary users from copying your work.

License management is about taking care of your customer base; plain and simple.

HexLicense

License management for
Embarcadero Delphi programmers

Part



2 Getting started

The HexLicense package consists of three parts:

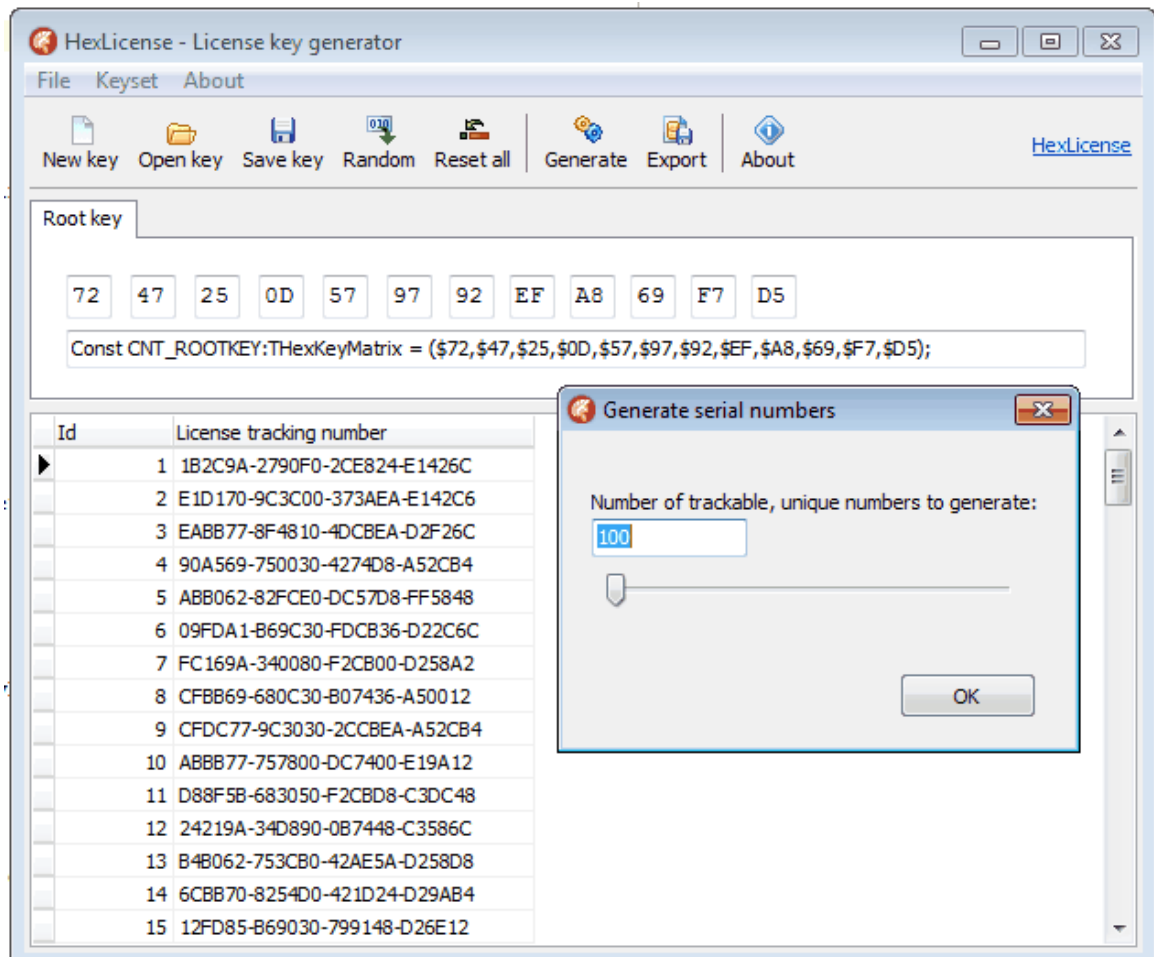
- The serial number generator application
- The Delphi license components
- The Delphi plug-in menu (starts the key generator from the IDE)

The license component package consists of the already mentioned 7 non-visual components. These components can be dropped on a form or a data-module. They require very little work on your part and can just as easily be dropped into an already existing application as a completely new program.

2.1 Generating a root key

Adding serial management to your application begins with the serial number generator. The generator (called “keygen” from now on) allows you to generate a random root key, which is the basis from which all compatible serial numbers are derived.

Optionally you may type in your own root-key sequence, but you should try to keep your root-keys as unique as possible.



In the picture above the root key is defined by the 12 hexadecimal input boxes. You can use your own values or simply click “new key” to make a random number sequence.

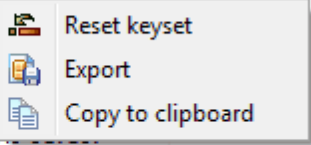
When the key changes the application automatically generates the delphi code you see in the text-edit box below the key values. You need to copy this source-code into your Delphi application and preferably store it in a separate unit. A hexadecimal key is much harder to locate for hackers, since it's stored with the machine code as opposed to a resource string or text constant.

2.2 Generating serial numbers

Once you have generated or typed in a root-key, the keygen application can be used to generate a large amount of unique serial numbers. Each of them derived from this root-key alone, which is also the only numeric sequence capable of validating them.

Note: While you can in theory create a massive amount of serial numbers from the same key, I strongly advice against this. All numbers once large enough begins to suffer from atrophy, meaning that the algorithm must search through a growing spectrum to locate an unused and reproducible combination.

Id		License tracking number
	1	D8E0B6-3F3C38-6388DC-9A45BD
▶	2	6C54C4-543C30-9990F0-A81736
	3	B4A8D2-004B40-6388DC-9A45BD
	4	6C54E0-A83C48-9070
	5	7870C4-D24B70-AB88
	6	F0FCD2-FC7828-8788
	7	24A89A-543C38-6370DC-8CFD00
	8	9054D2-694B50-99886E-622EBD
	9	FCE0EE-3F2D40-9960E6-9AE66C
	10	84708C-7E4B40-6C60D2-541787
	11	FC1CFC-2A4B68-6C8882-70E687



It's the exact same problem which make technologies like bit-coin extremely hard to work with. In that terminology locating a reproducible combination is referred to as “mining”.

In short: The more serial numbers you generate from the same root-key, the larger the distance between reproducible number combinations. The distance between matches grows exponentially as you mint and exhaust the base numbers.

As such I recommend that you generate an absolute maximum of 6000 serial numbers per key. But I strongly urge you to create a new key for every 1000 licenses you put into circulation.

2.3 Preparing your online store

Most online store-fronts support pre-generated lists of serial numbers. Typically you are expected to upload a serial-number file to the server as a normal text-file. Lately some vendors also accept XML and JSON formats as well.



Personally I would like to recommend [Gumroad](#) as a vendor. They allow you to upload serial lists and their service is simplicity itself for both customer and provider. Another vendor which accepts serial-number lists is [ShareIT](#), albeit their solution is more troublesome for the customer.

To generate serial numbers, simply click the “generate” button from the toolbar and select the amount. Please note that the more serials you mint, the longer it will take to produce the list (!)

Having generated your serial numbers there are two things you must do:

- Copy the Root-key const into your delphi project
- Export your serial numbers
- Save your key to disk (keep safe)

2.3.1 Root-Key constant

As mentioned, the keygen will automatically create a line of Delphi code for you; A single const array declaration which contains your root-key. This must be copied into the source-code of your application.

When you start your application the THexSerialMatrix component will ask for the root key. It does this through the OnGetKeyMatrix event.

2.3.2 Exporting serial numbers

Simply click the Export button from the keygen toolbar and you will be presented with several file-formats to choose from. You can choose to save the serial number list as:

- Text file
- XML dataset (TClientDataset)
- Binary dataset (TClientDataset binary format)
- JSON file

Since the majority of online vendors support plain text-files, this is the best format to pick. I have also added JSON support since Node.js based servers written in [Smart Mobile Studio](#) are in circulation.

2.3.3 Save your key to disk

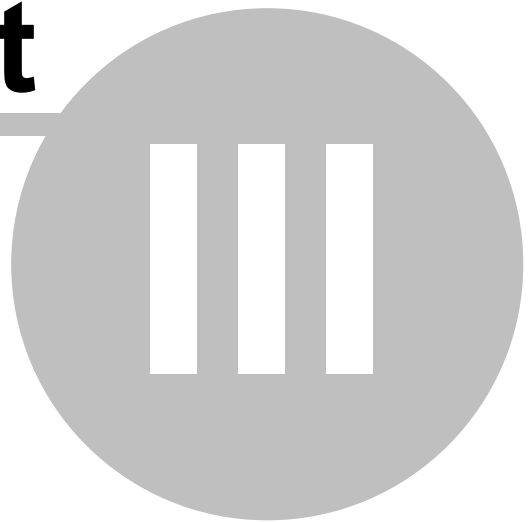
Having created a root-key, generated serial numbers and exported them to the file-system, remember to save your key as well. While you can just type in the root key, saving it in a safe place makes it easier to keep track of your number series.

Please note that saving your key does not save any generated lists. You are meant to use a different root-key ever time you mint new serial numbers.

HexLicense

License management for
Embarcadero Delphi programmers

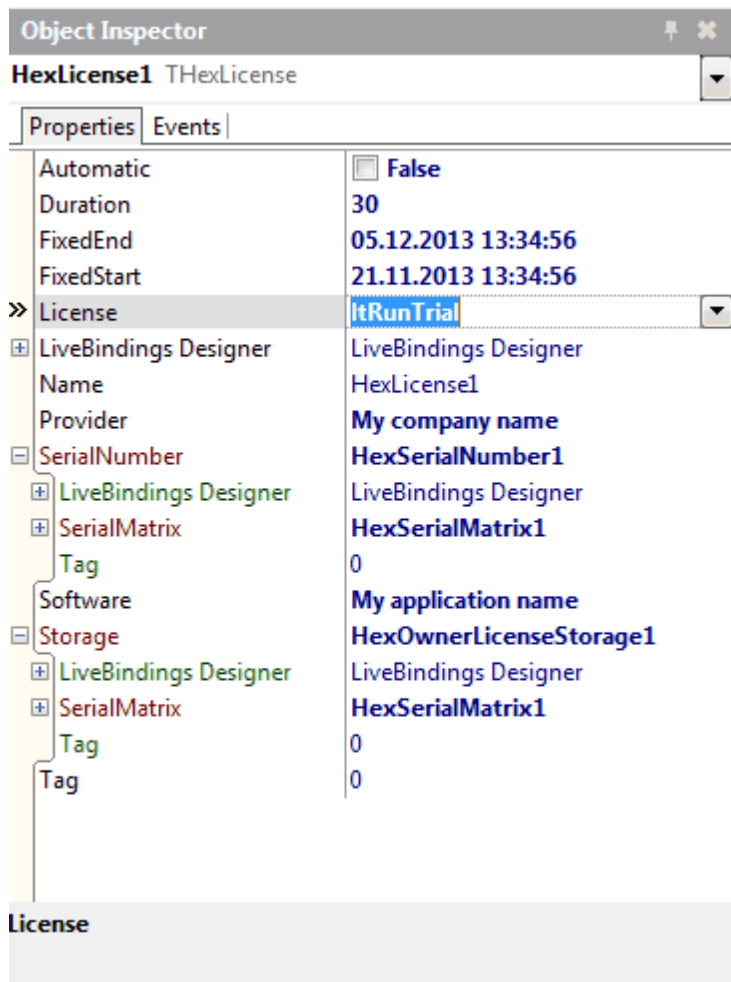
Part



3 The Delphi side of things

Now that you have done the preliminary work it's time to work with the components from Delphi. Before we start, a few words about architecture:

Delphi allows you to auto-create datamodules as part of your application's startup sequence. Datamodules are, as you probably know, "invisible forms" (actually they are components with resource persistency) which can host non-visual components exclusively.



If you are adding license management to an already existing application, chances are that you want to keep it separate from your already existing codebase. If that is the case, simply add a blank datamodule to your project and make sure Delphi automatically creates it when your application starts.

You start by dropping the following components on your datamodule or form:

- THexLicense
- THexSerialNumber
- THexSerialMatrix

3.1 Selecting a storage mechanism

Next you need to select a storage mechanism. By default HexLicense ships with 3 different storage adapters, these are:

- THexFileLicenseStorage
- THexRegistryLicenseStorage
- THexOwnerLicenseStorage

I strongly urge you to pick THexOwnerLicenseStorage. Using this storage adapter you can handle the entire storage of license information via ordinary Delphi events. It is imperative that you find a safe place to store the actual license data – here you are only limited by operative system credentials and creativity. Some typical places to store data are:

- Encoded into the pixel-buffer of an image
- Stored as a file inside a zip-file cabinet (password protected)
- Stored as a resource inside an .exe or .dll file (MSDN: [UpdateResource API](#))
- Stored as a TAG chunk of an image file

3.2 Connecting the root-key

The component THexSerialMatrix has a single event called “OnGetKeyMatrix”. You must respond to this event and return the const array generated by the codegen. An example of such an event handler is:

```
procedure TfrmMain.HexSerialMatrix1GetKeyMatrix
  (Sender: TObject; var Value: THexKeyMatrix);
Const
  CNT_ROOTKEY: THexKeyMatrix = ($4F, $9B, $BD, $7E,
    $79, $8E, $40, $98, $93, $EC, $FB, $D0);
begin
  Value := CNT_ROOTKEY;
end;
```

This is more or less all the manual coding you have to do. Next you simply need to connect each component to its corresponding published properties.

THexLicense must have it's properties “SerialNumber” and “Storage” connected to the THexSerialNumber and THexStorage component you selected (THexOwnerLicenseStorage is recommended).

THexSerialNumber needs to have the property SerialMatrix connected to the THexSerialMatrix component you dropped on the form or datamodule.

And finally, THexOwnerLicenseStorage must have it's SerialMatrix property connected to the same THexSerialMatrix component as THexSerialNumber.

This may sound very complex but it's very easy and self-explanatory when you examine the properties for these components.

3.3 Storage events

The component THexOwnerLicenseStorage exposes only three events. These are:

- OnDataExists
- OnReadData
- OnWriteData

The component does not really care how or where you store the license data (which is just a small chunk of encrypted data), it only cares that these three events are handled properly.

When your application starts the first event to fire on this component is OnDataExists.

If no data can be found (meaning, that it's the first time the user starts your program) a blank license is initialized and immediately stored through OnWriteData.

What is written inside the license file greatly depends on what type of license you want to use. As of writing you can choose between:

- ItDayTrial
- ItFixed
- ItRunTrial

3.3.1 ItDayTrial

Day trial allows the user to test your product for a finite amount of days. Typically 14 or 30 days of free use, before the customer must buy a full license if he or she wants to continue using your application. The amount of days is defined by the property "duration" of THexLicense.

3.3.2 ItFixed

While rare, this type of license allows you to compile a fixed start and stop date into your application – and your program will only work within the range of those dates. The start and stop-dates are defined by the "fixedStart" and "fixedEnd" TDateTime properties exposed by THexLicense.

3.3.3 ItRunTrial

This kind of license allows the user to start your application a finite amount of times, typically 100 times for commercial applications or 200 times for share-ware. The amount of runs is defined by the "duration" property of THexLicense.

You define your license type completely through the properties of THexLicense. This component also exposes all the events you need to handle activation, trial "time out" and other measurements. By setting the property "Automatic" to true the license session is initialized immediately when the application starts. If you want a finer grain of control over how the license management system works, you can set disable to false and activate the component by calling the method "BeginSession ()" whenever you mean it's most appropriate for your program.

HexLicense

License management for
Embarcadero Delphi programmers

Part

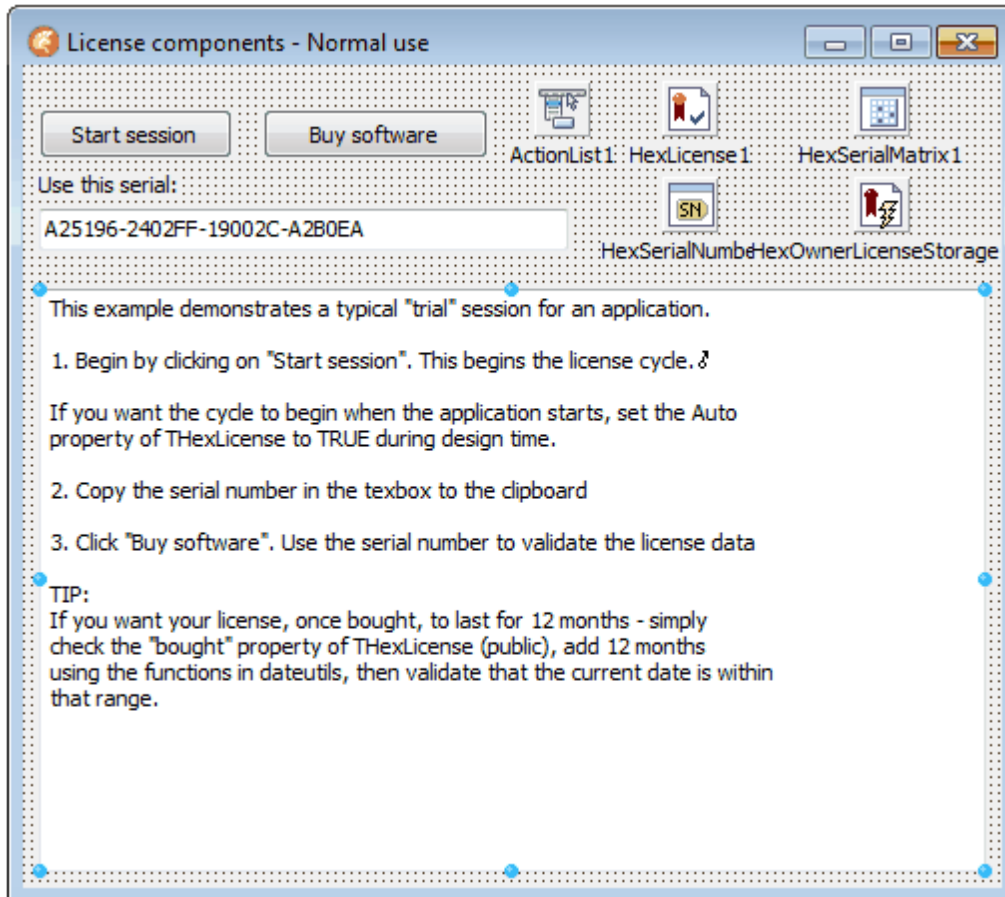


IV

4 A Delphi example

The following code is taken from one of the demonstrations that comes with the Delphi software package. As you can see from the code everything is explained throughout the unit.

The license components are very easy to use, even for novice Delphi developers. Each component is logically named according to the work it does, and each even is equally named for simplicity of use.



4.1 The code

The code is very simple and straight forward. Notice how you activate and validate a license with a single call, and also how the serial matrix is fetched.

```
unit mainform;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,
  Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls,
  Vcl.ActnList, hexmgrlicense, System.Actions;

type
  TForm1 = class(TForm)
    HexLicense1: THexLicense;
    HexSerialMatrix1: THexSerialMatrix;
    HexSerialNumber1: THexSerialNumber;
    HexOwnerLicenseStorage1: THexOwnerLicenseStorage;
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    ActionList1: TActionList;
    acStart: TAction;
    acAuth: TAction;
    procedure HexOwnerLicenseStorage1DataExists(Sender: TObject;
      var Value: Boolean);
    procedure HexOwnerLicenseStorage1WriteData(sender: TObject; Stream: TStream;
      var Failed: Boolean);
    procedure HexOwnerLicenseStorage1ReadData(Sender: TObject; Stream: TStream;
      var Failed: Boolean);
    procedure HexLicense1LicenseBegins(Sender: TObject);
    procedure HexLicense1LicenseObtained(Sender: TObject);
    procedure HexLicense1LicenseExpires(Sender: TObject);
    procedure HexSerialMatrix1GetKeyMatrix(Sender: TObject;
      var Value: THexKeyMatrix);
    procedure acStartExecute(Sender: TObject);
    procedure acAuthExecute(Sender: TObject);
    procedure acAuthUpdate(Sender: TObject);
    procedure acStartUpdate(Sender: TObject);
    procedure HexLicense1AfterLicenseLoaded(Sender: TObject);
  private
    (* Below is our temporary, in-memory only storage.
       In a real live application you would use either
       a filestream, or perhaps better: embed the stream
       inside a protected zipfile or some other mechanism which
       makes it harder for users to alter the data *)
    FStream: TMemoryStream;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```

{$R *.dfm}

uses dateutils;

#####
// Delphi action handlers - these just handle button states
//#####

procedure TForm1.acStartExecute(Sender: TObject);
begin
    (* Start the license cycle *)
    HexLicense1.BeginSession;
end;

procedure TForm1.acStartUpdate(Sender: TObject);
begin
    if not (csDestroying in ComponentState)
    and not (csLoading in ComponentState)
    and not (csCreating in ControlState) then
        TAction(sender).Enabled:=hexlicense1.Active=False;
end;

procedure TForm1.acAuthExecute(Sender: TObject);
begin
    (* Display the serial dialog *)
    Hexlicense1.Execute;
end;

procedure TForm1.acAuthUpdate(Sender: TObject);
begin
    if not (csDestroying in ComponentState)
    and not (csLoading in ComponentState)
    and not (csCreating in ControlState) then
        TAction(sender).Enabled:=Hexlicense1.Active
        and (hexlicense1.LicenseState<>THexLicenseState.lsValid);
end;
#####
// License component event handlers
//#####

procedure TForm1.HexLicense1AfterLicenseLoaded(Sender: TObject);
begin
    (* Check if the program is bought *)
    if hexlicense1.LicenseState=lsValid then
        Begin
            (* a year ago? Time to upgrade! *)
            if dateutils.DaysBetween(now,hexlicense1.Bought)>365 then
                Begin
                    showmessage('This license has expired (12 months have passed)');
                end;
            end;
        end;
end;

procedure TForm1.HexLicense1LicenseBegins(Sender: TObject);
const
    CNT_TEXT = 'Welcome to your trial version of %s!%s'
    + 'This is the first time you run our product';
begin
    showmessage(Format(CNT_TEXT,[HexLicense1.Software,#13]));
end;

```



```
procedure TForm1.HexLicense1LicenseExpires(Sender: TObject);
const
  CNT_TEXT = 'Your trial license has expired!%s'
    + 'Some functions will be disabled';
begin
  showmessage(Format(CNT_TEXT, [#13]));
end;

procedure TForm1.HexLicense1LicenseObtained(Sender: TObject);
const
  CNT_TEXT = 'Thank you for buying %s!';
begin
  showmessage(Format(CNT_TEXT, [HexLicense1.Software]));
end;

procedure TForm1.HexOwnerLicenseStorage1DataExists(Sender: TObject;
  var Value: Boolean);
begin
  (* If data cannot be found, a new license is created.
    So here we just check if we have stored anything.
    Since this is a "in memory" example, this event will always
    return false, and a new license is created every time you
    run the example.
    In a real application you would check if you have a license
    file (perhaps inside a protected space, or embedded inside
    a picture or something. Be clever where you store the data *)
  value:=assigned(FStream);
end;

procedure TForm1.HexOwnerLicenseStorage1ReadData(Sender: TObject;
  Stream: TStream; var Failed: Boolean);
begin
  (* The components will read the license at different
    situations, including when the license is created.
    But writedata is always invoked before anything is read
    in such a scenario (to initialize the session).
    In a real life example you would return the license-data
    from some other medium here, like a file - stored "somewhere" *)
  FStream.Position:=0;
  Stream.CopyFrom(FStream, FStream.Size);
end;

procedure TForm1.HexOwnerLicenseStorage1WriteData(sender: TObject;
  Stream: TStream; var Failed: Boolean);
begin
  (* Since this is an "in memory" example, we make sure we create
    our storage stream if it doesnt exist *)
  if FStream=NIL then
    FStream:=TMemoryStream.Create;

  (* Reset the size of the target, since we want to update the data *)
  FStream.Size:=0;

  (* now copy over the new license data to the target *)
  FStream.CopyFrom(Stream, Stream.Size);
end;

procedure TForm1.HexSerialMatrix1GetKeyMatrix(Sender: TObject;
  var Value: THexKeyMatrix);
```

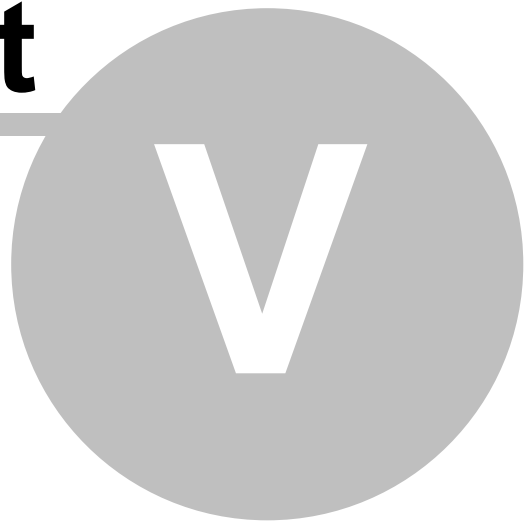
```
Const
  AMatrix: THexKeyMatrix =
    ($4E,$FC,$DC,$93,$02,$5A,$BE,$EC,$9D,$D5,$53,$58);
begin
  (* This is where we return the key matrix generated
    by the keygen application *)
  Value:=AMatrix;
end;

end.
```

HexLicense

License management for
Embarcadero Delphi programmers

Part



5 Using HexLicense server-side

HexLicense is a component based architecture. It is written in standard object pascal with no dependencies on system libraries, external DLL's or operative system functionality.

This means that HexLicense can be used both by VCL visual applications, non-visual applications; Firemonkey desktop applications, mobile applications and services.

5.1 Using HexLicense to build a serial server

Many customers have asked if HexLicense can be implemented server-side, effectively moving serial number identification and validation away from local applications.

This is ofcourse possible. In fact, one of the reasons HexLicense was organized as non-visual components was to ensure that it could be deployed in different situations. Not just in single executables.

Being able to use the HexLicense components on your server does not mean that no validation need occur in the program. Actually, your program should implement HexLicense as usual.

It must be understood that the server-side validation is for your benefit, not the customer.

5.2 Keeping track of licenses

A classical scheme for keeping track of licenses is (just an example):

- A database with all generated serial numbers, customers and root-keys
- A RPC or REST service exposing validation methods
- Capability to mark a serial number as "bought" and also store a disk checksum or serial number from the customer.
- Small list of roughly 10 records keeping track of when your program was last used (time, date), network mac address and IP address should also be stored in this table

5.3 The startup sequence

When your application starts, HexLicense will issue its events and you should validate the license as per usual; including expiration (please see introduction and quick-start on how to use Hexlicense) of the license or otherwise factors in need of attention.

When your license and session is considered valid, you should request a harddisk checksum or serial number, which identifies the harddisk your software has been installed to.

When calling your RPC/REST service, the following information should be supplied:

- Application serial number
- Harddisk serial number or checksum
- Duration of the license (expiration date, bought date)
- Remote IP
- User's network MAC address

When invoked, your server should look-up the serial number in the database. A bought license

should already be registered to a customer and the license marked as “valid”.

If the harddisk serial number is registered (which you should send to your server when your product is first bought) it should be compared with the one sent as a parameter.

Also, the bought-date and expiration-date on file should be compared to the information given as parameters in the validation call.

If all matches up, the server should return a OK and the program continue.

If it doesnt match up, you have to chose how to respond. You can:

- Log IP address of license offender for use in legal persecution
- Mark license as invalid, effectively terminating relation with that customer
- Mark license as invalid server-side, clear license locally and reduce the application back to trial mode.

I would urge you to respond with the latter two options: terminate your relationship with the customer based on breach of license agreement, and further clear the license locally and reduce the application back to trial mode.

It depends greatly on how much time and resources you are willing to spend hunting down pirates in other countries (and possibly continents). In some cases it may be worth while, but history demonstrates that legal action against a customer in another countries is difficult.

5.4 Multiple use of the same license

You should log the use of licenses regardless of the above, especially the time, mac and IP address is of interest here.

If a license is used at the same time, but in different locations; something you can pick up by tracking the IP through a whois() service, which can be automated via Delphi's Indy components, it is highly likely that your software license is being used on more than one computer.

The use of Whols is quite vital here, because if your software has been installed on a virtual-machine (VMWare, VirtualBox, Bochs etc), all the hardware information will be identical. The only thing that uniquely identifies each user is the IP, which can be traced back to an ISP or national region. Customers rarely change ISP and region, so this gives you an indication.

NOTE: ISP/Region checks should only be performed when your server notice that the same license is being used by different IP's at the same time. Customer can after all move house or use your program on a laptop traveling (hence they will have a different ISP depending on where they are).

If the use of a license is noticed during the same time-frame, or from an IP not in the same locale (look up the IP using whois() services) it is highly possible that someone is using the same license on different machines. Its important to use the Whols() service here, otherwise it may be difficult to distinguish between virtual computers should the software be installed on a virtual machine and copied.

HexLicense

License management for
Embarcadero Delphi programmers

Part



VI

6 The components

The entire package consists of seven non-visual components and the license generator application. Starting to use the package is a matter of dropping the desired components on a form, starting the generator application, generating a root key and a bulk of serial numbers, pasting the root-key into your code and setting the trial mode.

That is more or less all you need to do for the most basic setup. Now let's go through the components so you get a feel for what they do and how they connect.

6.1 THexLicense



THexLicense is the central component. It allows you to define the trial mode (fixed time range, day trial, run trial) and it's values. You can also set it to automatically start on loading, which means you don't have to manually call the component for it to initialize. This component must be connected to a THexSerialNumber component and a THexLicenseStorage adapter.

6.1.1 Events

[THexLicense](#) publishes events that each play a vital role in the internal dynamics of the component suite. While they are each important they are not hard to understand or use. For a complete example check out the [Delphi example code](#) at the end of this manual. The demonstration also ships with the product and you can set breakpoints and study the code live as it runs in Delphi.

6.1.1.1 WarnDebugger

This event will fire if HexLicense notice a global debugger running in the background. You should issue a message dialog explaining that you cannot allow this, or simply quit if this event ever fires.

Note: As of version 1.0.1 this event is deprecated. Modern debuggers largely go unnoticed and professional hackers use root-kits and virtual environments making it near impossible to detect their presence.

6.1.1.2 OnLicenseObtained

This event fires when a customer has entered a valid serial number into the register dialog, or when the THexLicense.Buy() method is manually used to register a product.

6.1.1.3 OnBeforeLicenseLoaded

This event is issued before the THexLicense attempts to load license data. Depending on the situation this makes room for adjustments, double-checking and/or opening storage devices where your license file resides.

6.1.1.4 OnAfterLicenseLoaded

This event fires after THexLicense has successfully loaded the license data. It allows you to check if the data is in order, if you are in trial mode or if its the first time the program has been executed.

6.1.1.5 OnLicenseBegins

This event fires the first time a license file is created. In other words, when the license session begins.

6.1.1.6 OnLicenseExpires

This event fires when the license period has expired. Typically you would inform the customer that its time to renew the license, and present a dialog with options.

How you respond to an expired license is up to you, some disable automatic updates if that is a part of the agreement, others reduces the program back to trial mode.

6.1.1.7 OnBeforeLicenseUnLoaded

This event fires as the license keyfile data is erased from memory. This typically happens when you close your program, manually shut down the session, or the components are destroyed.

6.1.1.8 OnAfterLicenseUnLoaded

This event fires after the keyfile data has been erased from memory, much like OnBeforeLicenseUnloaded this happens when a session is manually stopped, the components destroyed or your application is closing down.

6.1.1.9 OnLicenseCountDown

This event fires when you have chosen a trial mode involving days or amount of runs. If for example you only allow the program to be started 100 times, then this will fire on each run, decrementing automatically the count you have specified in the THexLicense.Duration property.

6.1.2 Properties

[THexLicense](#) publishes several properties, each of which are important and must be set at design-time. The property [License](#) is of special importance since that defines the trial type the component operates with, which in turn affects what properties like [Duration](#) mean.

6.1.2.1 Automatic

Datatype: Boolean

```
property Automatic: boolean read FAuto write FAuto stored true;
```

This is a boolean property that defines if you want THexLicense to automatically start when the application is executed. Some prefer to manually initiate the license session, but if you set this property to TRUE, the component will automatically call the BeginSession() method.

Notes:

See also *THexLicense.BeginSession()*

6.1.2.2 License

Datatype: THexLicenseType

```
type
  THexLicenseType = (ltDayTrial, ltRunTrial, ltFixed);
```

This property defines the trial model you want to use in your application. As of writing you can pick 3 types:

- ltDayTrial
- ltRunTrial
- ltFixed

ltDayTrail

This trial type allows the customer to use the product for a number of days. The amount of days can be defined through the duration property.

ltRunTrial

This trial type restricts the user to only execute the program a fixed amount of times before a valid serial number must be bought and applied.

ltFixed

This trial model allows the program to be used for a limited, specified time only. This trial mode is often used in public beta programs and makes sure that after a given date - the program can easily be rendered useless.

Notes:

Always remember that how you respond to an expired license, regardless of model, is 100% under your command. You can chose to simply exit the program, inform the user or present options.

6.1.2.3 Duration

Datatype: Integer

```
property Duration: integer read FData.lDuration write StoreDuration stored true;
```

This property defines the duration for [ltDayTrial](#) and [ltRunTrial](#) as set in the [THexLicense.License](#) property.

If set to ltDayTrial the duration property represents the amount of days the customer is allowed to use the application.

If the THexLicense.License property is set to ltRunTrial, the number represents the amount of time the customer can start the application.

6.1.2.4 Provider

Datatype: String

```
property Provider: string read Getprovider write SetProvider stored true;
```

This property should contain the name of your company. It will be displayed in the activation dialog.

6.1.2.5 Software

Datatype: String

```
property Software: string read GetSoftware write SetSoftware stored true;
```

This property should contain the name of your program. If you use the built-in license activation dialog, this string is used to represent the program the customer is buying.

6.1.2.6 FixedStart

Datatype: TDateTime

```
property FixedStart: TDateTime read FData.lFixedStart write SetFixedStartDate stored true;
```

When using the fixed trial model (see [THexLicense.License](#)) this field must be populated on compilation if you ship a pre-generated license keyfile with your product. It's corresponding end-date is defined by the property [THexLicense.FixedEnd](#).

6.1.2.7 FixedEnd

Datatype: TDateTime

```
property FixedEnd: TDateTime read FData.lFixedEnd write SetFixedEndDate stored true;
```

When using the fixed trial model (see [THexLicense.License](#)) this field must be populated on compilation. It marks the last date the product can be started. It's corresponding start-date is defined by the property [THexLicense.FixedStart](#).

6.1.2.8 Storage

Datatype: THexCustomLicenseStorage

```
property Storage: THexCustomLicenseStorage read FStorage write SetStorage;
```

In order for THexLicense to actually manage your license, it needs to store data. This data is basically a keyfile (or license file if you wish). It is a small, encrypted piece of data that contains the information about your license model. The THexLicense component manages everything for you, but naturally - it must be connected to a storage mechanism.

HexLicense ships with 3 storage mechanisms, these are:

- [THexFileLicenseStorage](#)

- [THexRegistryLicenseStorage](#)
- [THexOwnerStorage](#)

I strongly urge you to be creative about where and how you store your data! [THexOwnerStorage](#) gives you a huge advantage in that you get to decide where the data should be stored. You can place it inside the pixel-buffer of an image, inside a protected zipfile your program needs to run, inside a DLL file -- the more clever you are about it the better.

By default the license data is heavily encrypted. This is done automatically, so the binary data dealt with by [THexOwnerStorage](#) won't make much sense if people try to use a hex-editor on it.

As the class names hint to, [THexRegistryLicenseStorage](#) allows you to store license information in the registry - and [THexFileLicenseStorage](#) stores it as a file.

While perfectly reasonable, these two mechanisms represent the most vulnerable options, and are only included for legacy reasons.

Notes:

If you want to store more information, [THexOwnerStorage](#) is the perfect candidate. Simply append your data to the stream (see events for [THexOwnerStorage](#)) when saving, and read extract it again when the component issues a read event.

6.1.2.9 SerialNumber

Datatype: [THexSerialNumber](#)

```
property SerialNumber: THexSerialNumber read FSerial write SetSerialnrClass;
```

While [THexLicense](#) manages and maintains the actual license, serial numbers are dealt with by an isolated component called [THexSerialNumber](#). No matter what trial model you use, an instance of [THexSerialNumber](#) must be present and connected to [THexLicense](#).

6.1.3 Runtime properties

In order to respond to the state of the present license, some properties must by name be read-only and available at runtime. You can implement more vigorous testing of when the program was bought, when the trial was started - to make sure tampering with the Windows clock is noticed. [HexLicense](#) follows up on this, but there will always be situations where manual checking is important -- especially if you include the license state as a part of your [TAction.OnUpdate](#) enable/disable mechanisms.

6.1.3.1 Active

Datatype: Boolean

```
property Active: boolean read FActive;
```

This is a read only property that informs you if a user-session is active. You should check this at various places in your program before allowing the use of functionality.

6.1.3.2 SerialKey

Datatype: String

```
property Serialkey: string read GetSerialNumber;
```

This is a read only property that returns the serial-number used to activate the software.
This is only populated after a successful call to [Buy\(\)](#) has executed and the trial state expires.

6.1.3.3 LicenseState

Datatype: THexLicenseState

```
property LicenseState: THexLicenseState read FData.LrState;
```

This is a read-only property that returns the state of the current license. The following values are returned:

- IsPending
- IsExpired
- IsValid
- IsError

IsPending

This state means the program is still in trial mode and is pending validation

IsExpired

This state means the license has expired and the program should inform the user that a new license must be bought

IsValid

This state means that a valid serial number has been assigned the product, the license is valid and the program should continue to function as normal.

IsError

This state means the components could not validate the current state. This can be due to data corruption (if someone has tampered with the license-file, or the stream has been damaged in some way) but it may also be due to hacking.

6.1.3.4 LastExecuted

Datatype: TDateTime

```
property LastExecuted: TDateTime read FData.LrLastRun;
```

This run-time property returns the date and time the program was executed last. This can be a helpful property to determine if the user has tampered with the windows clock in an attempt to continue using the trial features beyond it's legal range.

6.1.3.5 DurationLeft

Datatype: Integer

property DurationLeft: integer read FDurationLeft;

This read-only property returns the duration left before the trial period expires. The meaning of the value depends on what trial model you have selected. If you are using [ItDayTrial](#) the integer represents the number of days left; if you used [ItRunTrial](#) it represents the number of times the user is allowed to start the program.

6.1.3.6 Bought

Datatype: TDateTime

property Bought: TDateTime read FData.IrBought;

This property returns the date when the product was bought (read: when a valid serial number was entered to activate the product). This is a handy property that helps you calculate if the user has tampered with the windows clock to extend the use of the program.

It is more commonly used to calculate when a subscription must be renewed. If you sell software with a 12 month subscription plan, a simple call to `System.DateUtils.DaysBetween()` in the Delphi RTL returns how many days have gone by since the product was activated.

```
procedure TForm1.HexLicense1AfterLicenseLoaded(Sender: TObject);
begin
    if hexlicense1.LicenseState=lsValid then
    begin
        (* a year ago? Time to upgrade! *)
        if dateutils.DaysBetween(now,hexlicense1.Bought)>365 then
        begin
            showmessage('This license has expired (12 months have passed!)');
        end;
    end;
end;
```

6.1.4 Methods

The public methods of `THexLicense` are few but central to the product. With the product now shipping in source-code form you can also inherit from the component to compose your own variations, and make use of the strictly protected procedures as well.

6.1.4.1 BeginSession

Type: Procedure

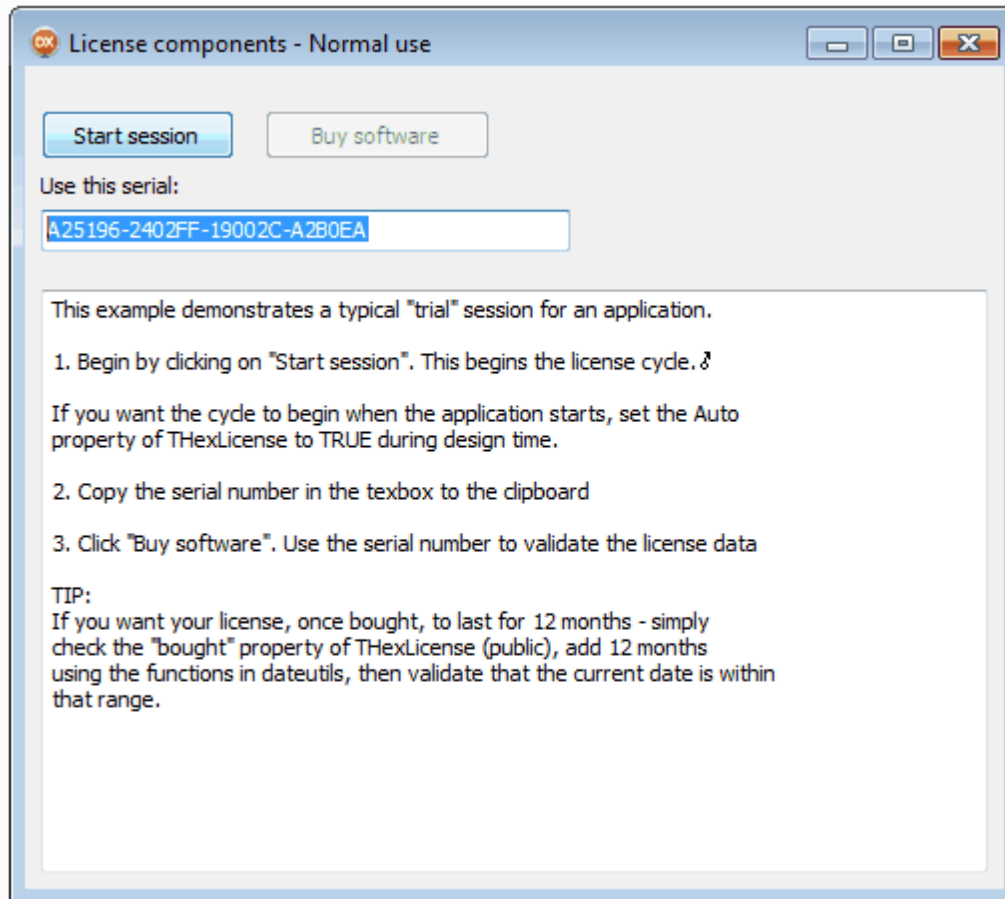
```
procedure BeginSession();
```

When a program managed by `HexLicense` executes, the duration of its use is called a session. Like all sessions it begins by the program starting, and ends by its termination. The procedure `BeginSession()` is the method that starts the whole license cycle and will issue the event-chain that asks for the root-key, requests the license file data and performs validation.

`THexLicense` let's you choose if you want the session to start automatically. By default the property

[THexLicense.Automatic](#) is set to true, which means that `BeginSession()` is called the moment all components are loaded and ready. This happens when the form or datamodule you have placed the components is created and becomes active.

If you however want better control over when the session starts, perhaps after a loading form or splash-screen, then you can set the property `Automatic` to false and call `BeginSession()` manually when it's more convenient to do so.



The demo program that ships with HexLicense demonstrate manual initiation of the user-session.

The demo program also uses in-memory streams to hold the license data (as a demonstration of how you can stream the license key anywhere you wish). By studying the code you should have no problems adapting the components to your application or service.

6.1.4.2 EndSession

Type: Procedure

```
procedure EndSession();
```

As with [BeginSession\(\)](#) this method is called by `THexLicense` automatically if the property [Automatic](#) is set to true. If you however want to control the user-session manually, you can call these functions as you see fit. Simply set the `Automatic` property to false in the designer and invoke these methods at an appropriate place in your code.

Notes:

See [BeginSession\(\)](#) for more information

6.1.4.3 Execute

Type: Function

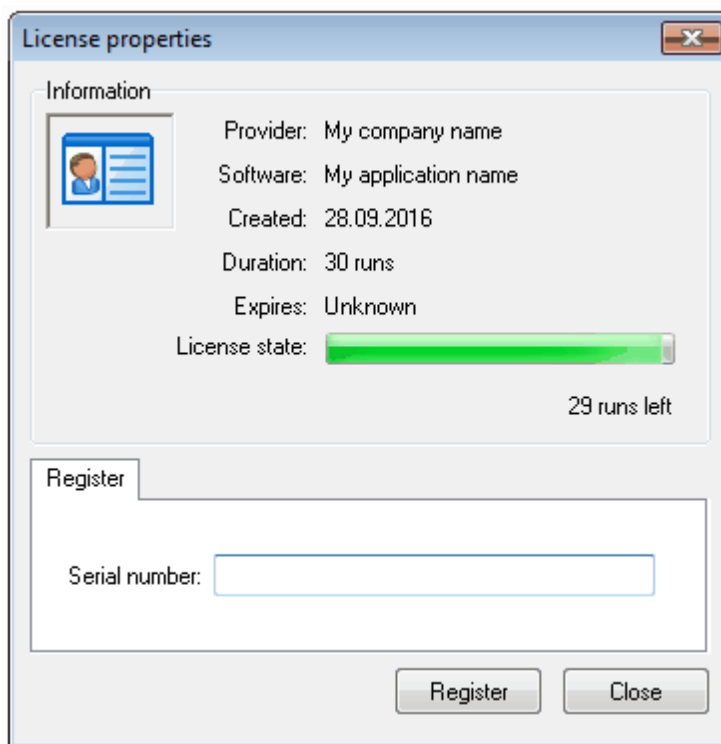
Datatype: Boolean

```
function Execute: boolean;
```

The Execute() method invokes the software activation form. The form itself allows a customer to input the serial number he has bought, which in turn disables the trial mode and allows the full use of the program.

The [THexLicense](#) component will call this method automatically when the trial model has expired, but you can also call the function manually. It is common to have a register option in the about menu of a program.

The function will return true if the serial-number validates against the root-key. The method also calls the method Buy() which marks the license as paid-for and thus disables the trial period. The event [OnLicenseObtained](#) is triggered to inform you about this important change of state.



Calling Execute() displays the built-in registration form

6.1.4.4 Buy

Type: Procedure

```
procedure Buy(const aSerialNumber: string);
```

If you have set THexLicense.Automatic to true, most methods are called automatically through the call-chain. If you however want to use your own dialog to present the user with other options (like getting a serial directly from your servers) then you can call the Buy() method directly.

This procedure will validate the serial-number against the root-key, and if it was generated by that key - the license file data is changed and the trial mode ends. The events for storing data in your chosen storage-mechanism will fire and the license is now in order.

6.2 THexSerialNumber



THexSerialNumber deals exclusively with the serial number.

In order for this component to do its work it must be connected to a THexSerialMatrix component.

THexSerialNumber has no published events. It encapsulate the means of validating a serial number through it's connection to THexSerialMatrix. This connection is exposed as the SerialMatrix property:

```
property SerialMatrix: THexSerialMatrix read FMatrix write SetMatrix;
```

For THexSerialNumber to function it must be connected to an instance of THexSerialMatrix. Likewise, THexLicense must be connected to THexSerialNumber for the circuit to be complete.

6.2.1 Methods

While THexSerialNumber is short on events and properties, it does contain vital methods for the internal workings of serial number validation and generation. It must be underlined however that these methods are rarely called by developers. They serve little function outside the scope of either validating or generating new serial numbers - tasks which are both automated and dealt with by the THexLicense and THexSerialGenerator components.

6.2.1.1 GetSerial

Type: function

```
function GetSerial(): string;
```

This function returns the serial-number presently in the buffer. The formula used to generate unique serial numbers from a common root-key can only come from nature, and is based on the fibbonanci expansion series coupled with german enigma encryption.

6.2.1.2 Validate

Type: function

```
function Validate(Serial: string): boolean;
```

This function is rarely used directly. It is called primarily by THexLicense when validating a serial-number against the root-key.

It is also used by THexSerialGenerator when producing new serial number combination based on the common root-key.

6.2.1.3 Clear

Type: procedure

```
procedure Clear();
```

This method clears the internal serial number buffer. Like most methods in this component this is rarely called directly by developers, it's called when needed by THexLicense and THexSerialGenerator.

6.2.1.4 Spin

Type: Procedure

```
procedure Spin();
```

This method is used to twist the number series into it's next position. Each digit in a serial number can be regarded as a number-wheel that grows from the initial root-key, much like a tree will always balance it's weight through the use of the Fibonacci formula.

This method is primarily used by THexSerialGenerator and have no meaning outside the scope of generating unique number sequences.

6.3 THexSerialMatrix



THexSerialMatrix is the component that handles the “root key”. You provide the root-key via an event handler. The root key is generated with the serial-number generator application that ships with the package (you can also write your own).

THexSerialMatrix has no public methods or properties, **with the exception of one very important event.**

When HexLicense starts it needs the root-key as generated by the keygen application. The event OnGetKeyMatrix must be populated and the constant you created with the key generator returned through that event.

The event is declared as such:

```
property OnGetKeyMatrix: THexGetKeyMatrixEvent
  read FOnGetMatrix write FOnGetMatrix;
```

The event declaration is designed to retrieve the byte-array constant:

```
type
  THexKeyMatrix = packed Array[0..11] of byte;
```

Where the event itself is straight forward:

```
type
  THexGetKeyMatrixEvent = Procedure (Sender: TObject;
    var Value: THexKeyMatrix) of object;
```

A typical event handler in your program looks like this:

```
procedure TForm1.HexSerialMatrix1GetKeyMatrix(Sender: TObject;
  var Value: THexKeyMatrix);
const
  AMatrix: THexKeyMatrix =
    ($4E,$FC,$DC,$93,$02,$5A,$BE,$EC,$9D,$D5,$53,$58);
begin
  (* This is where we return the key matrix generated
    by the keygen application *)
  Value:=AMatrix;
end;
```

6.4 THexCustomLicenseStorage

THexCustomLicenseStorage is the base-class for the storage mechanism components. It defines a set of very basic protected methods that descendant components must implement.

THexCustomLicenseStorage also contains a default encryption engine (RC4) that is applied to all read/write operations for the license data.

THexCustomLicenseStorage also allows indirect access to its IO methods. It implements the interface IHexLicenseStorage.

The interface declaration looks like this:

```
IHexLicenseStorage = Interface
  ['{894A9A51-6287-48BB-B6B3-6B195123DB02}']
  procedure ReadData(Stream: TStream; var Failed: boolean);
  procedure WriteData(Stream: TStream; var Failed: boolean);
  function DataExists: boolean;
  function Ready: boolean;
end;
```

The Ready() method can be called to check if the component can be used. If the dependency on [THexSerialMatrix](#) is not met, the function returns false.

6.4.1 Properties

THexCustomLicenseStorage exposes only one property, namely its link to [THexSerialMatrix](#).

```
property SerialMatrix: THexSerialMatrix read FMatrix write SetMatrix;
```

When dropping components on a form or datamodule in Delphi, simply connect the components together in with the property inspector. In most cases Delphi does this automatically. Inter-linked components use Delphi's Notification() scheme to know when it has been bound to another component.

6.4.2 Events

THexCustomLicenseStorage exposes only the most fundamental for dealing with read-write operations. It has only one property dependency towards THexSerialMatrix, and the events are the most central when adding THexLicense to your application.

6.4.2.1 OnDataExists

Type: Event

```
property OnDataExists: THexDataExistsEvent read FOnExists write FOnExists;
```

This event fires before a read-operation begins. Please note that this event exists also on components that target specific media, like THexFileLicenseStorage even though that component will use FileExists() by default. These events are exposed to give you the means of overriding default behavior should it be needed.

Note: This event is mostly handled when you use THexOwnerStorage.

6.4.2.2 OnReadData

Type: Event

```
property OnReadData: THexReadLicenseEvent read FOnRead write FOnRead;
```

This event fires when a read operation should be performed. Please note that if you handle this event on media-specific components (like THexFileLicenseStorage), you effectively override the default behavior.

Note: This event is mostly handled when you use THexOwnerStorage, which is highly recommended that you use. That way you can hide the data cleverly in disguise as something else.

6.4.2.3 OnWriteData

Type: Event

```
property OnWriteData: THexWriteLicenseEvent read FOnWrite write FOnWrite;
```

This event fires when a write operation should be performed. Please note that if you handle this event on media-specific components (like THexFileLicenseStorage), you effectively override the default behavior.

Note: Like with OnReadData, this event is mostly handled when you use THexOwnerStorage. I highly recommend you use THexOwnerStorage as your medium because that gives you more options to hide you license data.

6.4.3 Encryption

THexCustomLicenseStorage implements the extremely fast and efficient RC4 ([Rivest cipher version 4](#)) encryption engine. In cryptography RC4 has later been replaced by RC5 and more advanced encryption engines, but for personal use RC4 represents extremely good protection.

The encryption methods exposed by THexCustomLicenseStorage are simple and universal, they are abstracted from the actual implementation

- [CanEncode](#)
- [EncodeData](#)
- [DecodeData](#)

These methods are strictly protected.

6.4.3.1 CanEncode

Type: Function

```
function CanEncode: boolean;
```

Before reading or writing the license data this method is invoked asking if the encryption engine is

available. If no encryption should be applied the implementation must return false. Since RC4 is built into the baseclass you dont need to override any of the encryption methods unless you want to use a different cipher.

6.4.3.2 EncodeData

Type: Procedure

```
procedure EncodeData(pBuffer: pointer; Length: integer);
```

This method is called to encrypt a piece of data. It is called automatically on read/write operations of the license information. Since RC4 is built into the baseclass you dont need to override this method unless you want to use a different cipher. In which case you must also override [DecodeData\(\)](#).

6.4.3.3 DecodeData

Type: Procedure

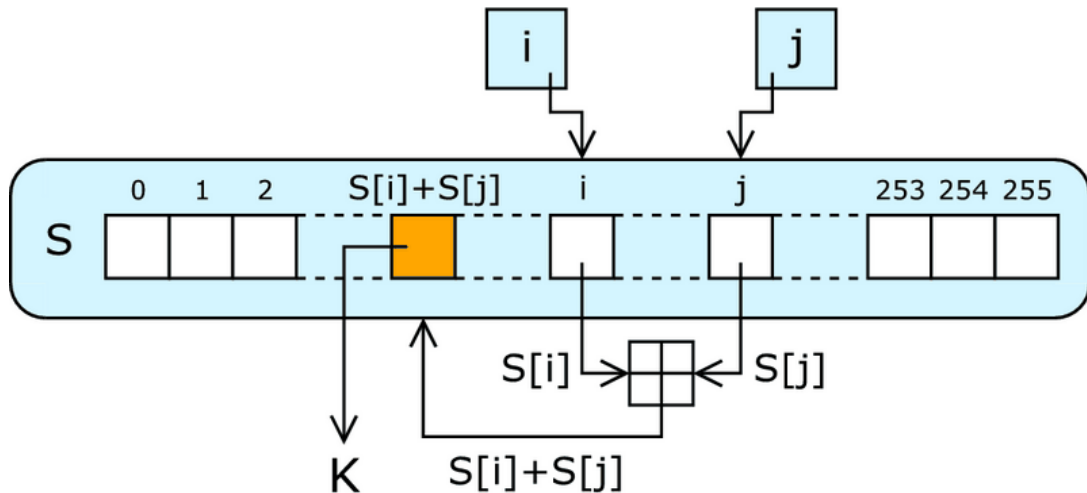
```
procedure DecodeData(pBuffer: pointer; Length: integer);
```

This method is called to decrypt a piece of data. It is called automatically on read/write operations of the license information. Since RC4 is built into the baseclass you dont need to override this method unless you want to use a different cipher. In which case you must also override [EncodeData\(\)](#).

6.4.3.4 How does RC4 work

RC4 creates a pseudorandom stream of bits that is also referred to as a keystream. Similar to other stream ciphers, the keystream can be leveraged for encryption operations by combining it with plaintext using the XOR operation. Decryption works similarly through the involution of the ciphertext. In order to create the keystream, the RC4 cipher will use a secret internal state comprised of two items: 1 – Two, eight bit pointers that are represented by the letters “i” and “j,” and 2 – A permutation of all 256 possible bytes that is connoted by the letter “S.”

The RC4 permutation is initialized using a variable length key. This key will typically range between 40 and 256 bits that uses the KSA (key scheduling algorithm). Once the permutation is created, the stream of bits will be created using the PRGA (pseudo-random generation algorithm).



Most cryptologic agencies work with super-computers and clusters. To break an RC4 encrypted piece of data took thousands of CPU cores working non-stop over a period of weeks. In later times more efficient models of attack have been created, but again these agencies use supercomputers.

Example: The Shamir attack used around 10 million messages with a rotating cipher, and was able to break 104 keys using 40,000 frames with 50% probability, and 85,000 frames with 95% probability.

In human terms: they were able to break a http cookie encrypted with RC4 in *74 hours* using a cluster.

You can read more about RC4 here: <http://www.tech-faq.com/rc4.html>

6.5 THexFileLicenseStorage



THexFileLicenseStorage inherits from [THexCustomLicenseStorage](#) and provides automatic storage of license data on the filesystem.

Please note that this component is sensitive to the security permissions of the running application.

For properties and methods, please see documentation for [THexCustomLicenseStorage](#).

6.6 THexRegistryStorage



THexRegistryStorage inherits from [THexCustomLicenseStorage](#) and allows you to automatically read, write and create a license file in the Windows registry. Much like THexFileLicenseStorage it is subject to the same security permissions as the running application.

For properties and methods, please see documentation for [THexCustomLicenseStorage](#).

6.7 THexOwnerStorage



THexOwnerStorage inherits from [THexCustomLicenseStorage](#) and allows you to override and take charge of the read, write and create process. Due to the changes in Windows during the last 8 years it is highly recommended that you work with this component, since both registry and fixed path IO is discouraged in modern software development. Using this component allows you fill freedom in where you store the license data, only security permissions and creativity sets boundaries.

For properties and methods, please see documentation for [THexCustomLicenseStorage](#).

6.8 THexSerialGenerator



THexSerialGenerator is the component used by the license key application to create new keys and produce serial numbers in bulk. It exists as a component so it can be easily added server-side if you already have an existing ordering system and user-database available.

6.8.1 Methods

THexSerialGenerator expose only one new method compared to it's ancestor, THexSerialNumber. And that is the generate method.

The declaration looks like this:

```
procedure Generate(Count:integer); virtual;
```

When you want to generate new serial numbers, all components must be connected (THexLicense, THexSerialGenerator, THexSerialMatrix and THexOwnerStorage) before you call this method.

Please note that **THexSerialGenerator takes the place of THexSerialNumber**. It inherits from THexSerialNumber and thus you can omit that from the setup.

6.8.2 Events

Type: Event.

THexSerialGenerator only expose one event, the OnSerialnumberAvailable event:

```
property OnSerialnumberAvailable: THexSerialNumberAvailableEvent  
    read FOnAvailable write FOnAvailable;
```

This event fires as the generator component runs through it's sequencing of unique numbers. By handling this event you can cache up the new serial numbers.

Please note that the generator will produce already existing numbers from time to time, this is due to how it exhausts a permutation sequence (growth ring). So you must always check that a serial-number is unique before placing it in a list (just check if it's already in the list, if so - disregard it).

HexLicense

License management for
Embarcadero Delphi programmers

Part



VII

7 Number engines

License management and software security are two different things. The concepts overlap, naturally, since the whole purpose of a license system is to enable or disable features based on the premise of bought access. But there are differences that are important to understand.

Ultimately license management is about just that: to manage licenses. Which in practical terms can be reduced to: being able to produce serial numbers that are non-linear yet can be validated against its origins.

This is where license management and cryptography differs greatly. Being able to produce a non-linear serial number has really very little to do with certificates, private and public keys, checksum validation and all the intricate aspects that software security entails.



7.1 Linear vs. non linear

A linear number is a normal integer value that grows in a linear, forward fashion. Let's say you have a serial number that is "1001". Now that's a very bad serial number, because the first thing a hacker is going to do is to try "1002". Perhaps you have been clever about it and added 28 or 64 for each valid number — but a brute force password finder will cut through "schemes" like this in milliseconds. It's not even worthy of debate.

Term:	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Term Value:	2	9	16	23	30	37	44	51
Common Difference:		+7	+7	+7	+7	+7	+7	+7

Linear numbers should never be used as the basis for a license management system

Non-linear numbers are more complex: first of all they consist of more parts. Not just a single value, but several values. Each of which must match or have a range of matches in order to validate. It is very hard to use traditional hacking tools to figure out a non-linear number, because you have to find out the valid number ranges for each part of the whole.

	4	7	12	19	28
First difference		6	10	14	18
Second difference			4	4	4

Non-linear means that more factors must match in order to produce a valid range. The more factors, the harder it is to reverse engineer

Also, if you don't know the number that was used to start with, the root key, then trying to reverse

engineer it is indeed tricky.

One way of thinking about this is a bicycle lock: you know, those small code wheels you have to turn to input the code? Now non-linear number generators can be thought of as a collection of such wheels, but with 256 (0..255 = one byte) possible settings per wheel. Not just 8 or 10.

This gives you: 12^{256} different combinations (roughly 1.8 billion). And out of those the cipher allocates a percentage or range that is deemed valid within the scope of each byte. HexLicense allows for a maximum of 64 out of 256 per segment to be valid, which gives you 4.7 million possible combinations. This is further reduced by the way numbers are grown. The gaps as a number advances non-linearly are invalid and unused, and the total they represent must further be subtracted from the total of possible combinations.

HexLicense has been tested with up to 100.000 generated serial numbers from a single root-key. This is a very low number compared to the possible combinations involved, but it also helps limit brute force attacks. Besides, minting a new root-key after having sold 100.000 copies of your program is a luxury problem, not a technical problem.

7.2 Number engines and matrixes

There really is no “proper” way when it comes to creating a number generator. You can’t go to school and take a course in generating unique license strings. It all comes down to the formula and how you deploy it.

Another fact to remember is that number engines and random number generators is not the same thing. The latter being a field within mathematics and science; The first belonging more in the realm of natural mathematics, esotericism and number games. Despite this, the way scientific random number generators work is surprisingly low-tech until you enter the realm of physics.

But its important not to mix these topics up, because science is about availing something, while the other is about masking and hiding something.

The majority of engines that generate unique licenses today either use encryption to mask data, which is not what HexLicense is about; or they rely on alternative mathematics. Hexlicense uses quite ancient number sequences, like the [Lucas numbers](#), the [Fibonacci numbers](#) and [sepsephos gematria](#).

We are adding support for more formulas in order to bring a greater diversity to the available serial lists. As of writing the following ancient formulas have been added to the library:

- Beatus gematria matrix
- Hebrew gematria matrix
- Sepsephos greek gematria matrix
- Latin common gematria matrix
- Egyptian gematria matrix (*)

(*) [A study of numbers](#), R.A Schwaller de lubics

There is no risk involved in knowing these things because the amount of combinations would still require a herculean effort to reverse engineer. What is important for this company is that our users understand the technology, because that helps them deploy the software in ways that serve them better.

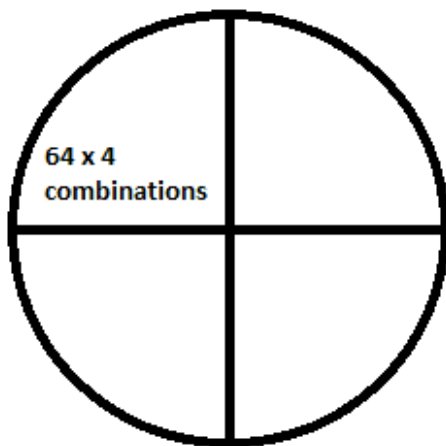
7.3 Exploring number sequences

The following Smart Pascal code generates both the Lucas and Fibonacci number sequences. The code itself is useless without a framework that puts the numbers to work, that creates structure and uses the numbers to produce sequences that can be validated. Which is what HexLicense does for you.

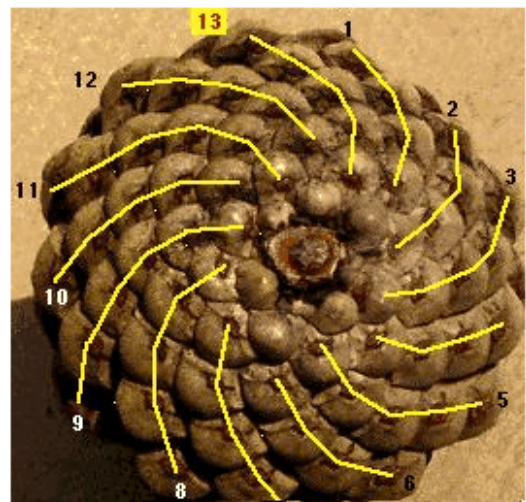
Please note that this code is placed here for educational purposes only. This code is very different from what can be found inside HexLicense, which is more compact and deals with the numbers head on.

The code is written in [Smart Pascal](#) and can be compiled to Javascript with [Smart Mobile Studio](#). It is written to be generic and should also compile with Delphi or freepascal with little or no changes.

The number series classes are a great way to explore the patterns that govern growth and, essentially, life as we know it.



Each hex number when seen as a growth seed, becomes a 4 x 64 matrix when the Lucas number sequence is applied. Fibonacci restriction makes sure the numbers never go out of alignment.



Hexlicense uses patterns from nature to control non-linear numbers

type

```
T8bitNumberSequence = Array[0..255] of integer;

TNumberSeries = class(TObject)
private
    FCache: T8bitNumberSequence;
protected
    function BuildNumberSeries: T8bitNumberSequence; virtual; abstract;
    function GetItem(Index: integer): byte; virtual;
    procedure SetItem(Index: integer; const Value: byte); virtual;
    function GetCount: integer; virtual;
public
    property Data[Index: integer]: byte read GetItem write SetItem;
    property Count: integer read GetCount;
    function ToString: string; virtual;
```

```

    function ToNearest(const Value: integer): integer; virtual;
    constructor Create;virtual;
end;

TLucasSequence = class(TNumberSeries)
protected
    function BuildNumberSeries: T8bitNumberSequence;override;
    function DoFindNearest(const Value: integer): integer;virtual;
public
    function ToNearest(const Value: integer): integer; override;
end;

TFibonacciSequence = class(TNumberSeries)
protected
    function BuildNumberSeries: T8bitNumberSequence;override;
public
    function ToNearest(const Value: integer): integer; override;
end;

#####
// TFibonacciSequence
#####

function TFibonacciSequence.BuildNumberSeries: T8bitNumberSequence;
var
    x: integer;
    a,b : integer;
begin
    result[low(result)] := 0;
    result[low(result)+1] := 1;
    for x:=low(result)+2 to high(result) do
    begin
        a:= result[x-2];
        b:= result[x-1];
        result[x] := a + b;
    end;
end;

function TFibonacciSequence.ToNearest(const Value: integer): integer;
var
    LForward: integer;
    LBackward: integer;
    LForwardDistance: integer;
    LBackwardsDistance: integer;
begin
    (* Note: the round() function always rounds upwards to the closest
       whole number. Which in a formula can result in the routine
       returning the next number even though the previous number
       is closer.
       To remedy this we do a distance compare between "number" and
       "number-1", so make sure we pick the closest match in distance *)
    LForward := round( TFloat.Power( ( (1+SQRT(5)) / 2), value) / SQRT(5) );
    LBackward := round( TFloat.Power( ( (1+SQRT(5)) / 2), value-1) / SQRT(5) );

    if LForward <> LBackward then
    begin
        LForwardDistance := LForward - value;
        LBackwardsDistance := Value - LBackward;

        if (LForwardDistance < LBackwardsDistance) then

```

```

        result := LForward else
        result := LBackward;
    end else
        result := LForward;
    end;

//#####
// TLucasSequence
//#####

function TLucasSequence.DoFindNearest(const Value: integer): integer;
var
    LBestDiff: integer;
    LDistance: integer;
    LMatch: integer;
    a,b,c : integer;
begin
    LBestDiff := MAX_INT;
    LMatch := -1;

    a := 2;
    b := 1;
    repeat
        c := a + b;
        LDistance := c - value;
        if (LDistance >= 0) then
            begin
                if (LDistance < LBestDiff) then
                    begin
                        LBestDiff := LDistance;
                        LMatch := c;
                    end;
            end;
    until (c > value) or (LBestDiff = 0);

    if (LMatch > 0) then
        result := LMatch else
        result := value;
    end;

function TLucasSequence.ToNearest(const Value: integer): integer;
var
    LForward: integer;
    LBackward: integer;
    LForwardDistance: integer;
    LBackwardsDistance: integer;
begin
    (* Note: Lucas is a bit harder to work with than fibonacci. So instead
       of using a formula we have to actually search a bit.
       It is however important to search both ways, since the distance
       backwards can be closer to the number than forward *)
    LForward := DoFindNearest(Value);
    LBackward := DoFindNearest(value-1);

    if LForward <> LBackward then
        begin

```

```

    LForwardDistance := LForward - value;
    LBackwardsDistance := Value - LBackward;

    if (LForwardDistance < LBackwardsDistance) then
        result := LForward else
        result := LBackward;
    end else
        result := LForward;
    end;

function TLucasSequence.BuildNumberSeries: T8bitNumberSequence;
var
    x: integer;
    a,b : integer;
begin
    result[low(result)] := 2;
    result[low(result)+1] := 1;
    for x:=low(result)+2 to high(result) do
        begin
            a:= result[x-2];
            b:= result[x-1];
            result[x] := a + b;
        end;
    end;

#####
// TNumberSeries
#####

constructor TNumberSeries.Create;
begin
    inherited create;
    FCache := BuildNumberSeries;
end;

function TNumberSeries.ToNearest(const Value: integer): integer;
var
    x: integer;
    FDiff: integer;
    LBestDiff: integer;
    LIndex: integer;
begin
    result := -1;
    LBestDiff := MAX_INT;
    LIndex := -1;

    for x:=low(FCache) to high(FCache) do
        begin
            FDiff := value - FCache[x];
            if (FDiff >= 0) then
                begin
                    if FDiff < LBestDiff then
                        begin
                            LBestDiff := FDiff;
                            LIndex := x;
                        end;
                    end;
                end;

            if ( FCache[x] > value ) then
                break;
        end;
    end;

```

```
end;

if LIndex>=0 then
  result := FCache[LIndex];
end;

function TNumberSeries.ToString: string;
var
  x: integer;
begin
  for x:=low(FCache) to high(FCache) do
    begin
      if x<high(FCache) then
        result := result + FCache[x].toString() + ', ' else
        result := result + FCache[x].toString();
      end;
    end;
  end;

function TNumberSeries.GetItem(Index: integer): byte;
begin
  result := FCache[Index];
end;

procedure TNumberSeries.SetItem(Index: integer; const Value: byte);
begin
  FCache[Index] := value;
end;

function TNumberSeries.GetCount: integer;
begin
  result := length(FCache);
end;
```

HexLicense

License management for
Embarcadero Delphi programmers

Part



8 HexTools

HexTools is a utility class that is implemented specifically for each supported platform HexLicense supports.

The system abstracts you from the underlying complexity of each platform and gives you clean, no-nonsense and concrete API that is the same across platforms.

In other words: the code you write for Windows will work just as well under Android or OS X. As long as you stick to the class and its functions, you're home free.

As of writing the following platforms are supported:

- Windows 32/64 bits
- Google Android
- Apple iOS
- Apple OS X
- Linux (experimental under freepascal)

8.1 Using HexTools

In order to use HexTools you must first include the unit in your project. Depending on what component library you use (Firemonkey or VCL) the unit names are:

- FMX.hextools.pas (Firemonkey)
- hextools.pas (VCL)

Since VCL is Windows only, the VCL unit only contains code that executes under Windows 32 and 64 bit.

8.2 Class declaration

Since each operative system deals with getting information differently (the difference between iOS and Android is quite substantial), HexTools needs an instance.

Had the operative systems been more similar we might have gotten away with pure class functions, but alas that is not the case.

But you must never create the class yourself. We have hidden the implementation away from the interface section to make sure you don't spawn an instance by mistake.

While this won't create huge problems right now, the way we implement our code may change in the future - and dependency injection helps us shield you from any re-engineering on our part.

The class declaration is

```
THexTools = class(TInterfacedObject)
private
    FLastError: string;
protected
    procedure SetLastError(const ErrorText: string); virtual;
    procedure ClearLastError; virtual;
    function GetFailed: boolean; virtual;
public
    property LastError: string read FLastError;
```

```
property Failed: boolean read GetFailed;

class function CalcCRC(const Stream: TStream): longword; virtual;
function DiskSerial: string; virtual; abstract;
function MacAddress: string; virtual; abstract;
function IPAddress: string; virtual; abstract;
end;
```

8.3 Getting an instance

To get an instance of HexTools, simply call the GetHexTools() unit function. This function has the following declaration:

```
function GetHexTools(var Access: THexTools): boolean;
```

And can be used in typical service style programming:

```
procedure TForm1.UseHexTools;
var
  LInstance: THexTools
begin
  if GetHexTools(LInstance) then
  begin
    try
      // use the methods here
    finally
      LInstance := nil;
    end;
  end;
end;
```

Note: THexTools inherits from TInterfacedObject. This means you don't have to release the instance by calling free (it can actually cause problems if you do). By setting the reference pointer to NIL you effectively force the instance to go out of scope. It's not really needed but I tend to do this. The last thing you want is a java wrapper floating around in memory acting like a singleton. So better safe than sorry.,

8.4 Methods

The public methods of THexTools are few but very valuable. They are typically used when registering an installation with your server and represents the most commonly used factors when building a server that keeps track of licenses and users.

8.4.1 DiskSerial

Type: Function
Datatype: String

```
function DiskSerial: string;
```

This function returns a string that identifies the primary storage device. For a Windows PC this

would be the boot drive (C). Mobile devices typically only have one storage device.

Note: *The content of the string varies greatly from device to device and platform to platform.*

The content of the string doesn't really matter. It is derived from the operative system and will uniquely identify the primary storage device.

Should this change during the life-cycle of your application - you know the program has been copied to another computer.

What you want to do is this:

1. When the user enters a valid serial-number, you grab the disk serial
2. You append the serial to the HexLicense data stream (see [THexCustomLicenseStorage](#))
3. Whenever the application is started from then on, check if there is data appended to the license stream, and compare what is on file with a call to `DiskSerial()`

If it differs, the program has been copied.

You should also ship the info to your server when doing online registration. How you decide to deal with multiple installations etc. is naturally up to you.

But most companies tend to allow 1-3 installs per license. But to enforce this you must also connect to your server once in a while to check.

It can be wise to do this with as little intrusion as possible. There is a balance between defending your software and ruining the experience.

8.4.2 MacAddress

Type: Function

Datatype: String

```
function MacAddress: string;
```

This function returns a string that identifies the network adapter's Mac address that was used when registering the software.

On mobile devices this never really changes. PC's and Mac's however can have multiple adapters, so this is not a property you want to enforce too hard.

But changes in adapter after registration is good indication that an online check is due.

Note: *The content of the string varies greatly from device to device and platform to platform.*

Like with `DiskSerial()`, the content of the string doesn't really matter. It is derived from the operative system and will uniquely identify the network adapter.

8.4.3 IPAddress

Type: Function

Datatype: String

```
function IPAddress: string;
```

This function returns a string with the current WAN IP of the computer or device. If the device is not connected to the internet, 127.0.0.1 is returned.

Note: Take some care in checking this on mobile devices. Never put this on interval without asking the user, because calling this function can cause the mobile device to connect and use 3G/4G data without your intention. Depending on where you live and the pricing on mobile data - that can cause some irritation.

8.4.4 CalcCRC

Type: Function
Datatype: String

```
class function CalcCRC(const Stream: TStream): longword; virtual;
```

As the name implies this function calculates a CRC (cyclic redundancy check) checksum for the content of the stream parameter. Notice that the CalcCRC() function is implemented in object pascal itself with no dependencies on the operative system.

It is also implemented as a class function. This can be handy if you need to calculate the CRC of a stream anywhere in your code without calling GetHexTools() first:

```
var
  LCRC: longword;
begin
  // A class function doesnt need an instance.
  // Here we call the CRC calculation routine directly without getting
  // an instance of HexTools first
  LCRC := THexTools.CalcCRC(MyStream);
end;
```

8.5 Properties

HexTools only expose two properties at this time, namely those dealing with errors. When using HexTools you want to check the Failed() property to see if any errors occurred. If Failed() returns true, you can read the error message via the LastError() property.

8.5.1 Failed

Type: property
Datatype: Boolean

This property returns a boolean value representing the success or failure of the last HexTools function call.

While rare, exceptions can occur. For instance if something is wrong with the network, or a disk has a read/write error. In such cases calls to DiskSerial() and MacAddress() can raise exceptions.

Since exceptions under Java (Android) can be difficult, we decided to mute any exceptions and capture the message/status instead.

So even if something goes wrong, no exceptions will bubble up to the surface and stop your code.

Example:

```
var
```

```
LDiskSerial: string;

LDiskSerial := LHexTools.DiskSerial();
if not LHexTools.Failed then
begin
    // No errors occurred, use the value here
end else
begin
    // Something went wrong, write code to deal with errors here
    Showmessage(LHexTools.LastError);
end;
```

8.5.2 LastError

Type: property
Datatype: String

This property returns a the error message from the last exception or error that occurred. You can check for errors using `LHexTools.Failed()`.

The content of the string is purely dependent on the error, platform and operative system.

How you deal with errors should they occur is of great importance. If your program has been trying to register with your server 200 times and every single time it fails, then you might want to reduce the program back to trial mode (as an example).

HexLicense

License management for
Embarcadero Delphi programmers

Part



IX

9 More useful code

There are plenty of examples on how to obtain a harddisk serial number, the network MAC and IP address online. Use google to find a solution which best matches your program.

You will also find plenty of solutions on websites like Torry's Delphi pages, Github and EDN.

Below are examples on how you can obtain system information. Please note that these are only examples from various places around the internet. They must always be tested, and you should preferably look up the various API calls to make sure they are not deprecated before using this code.

We do not provide any warranty regarding the code below. They are provided as examples only, and you use the code purely at your own risk.

9.1 Bios identifier and disk serial

```
uses
  ActiveX,
  ComObj;

var
  FSWbemLocator : OLEVariant;
  FWMIService   : OLEVariant;

function GetWMISTring(const WMIClass, WMIProperty:string): string;
const
  wbemFlagForwardOnly = $000000020;
var
  FWbemObjectSet: OLEVariant;
  FWbemObject   : OLEVariant;
  oEnum          : IEnumvariant;
  iValue         : LongWord;
begin;
  Result:='';
  FWbemObjectSet:= FWMIService.ExecQuery(Format
    ('Select %s from %s',[WMIProperty, WMIClass]),'WQL',wbemFlagForwardOnly);
  oEnum          := IUnknown(FWbemObjectSet._NewEnum) as IEnumVariant;
  if oEnum.Next(1, FWbemObject, iValue) = 0 then
    if not VarIsNull(FWbemObject.Properties_.Item(WMIProperty).Value) then
      Result:=FWbemObject.Properties_.Item(WMIProperty).Value;
      FWbemObject:=Unassigned;
    end;
end;

function GetHarddiskSerial(var Bios,Media:String):Boolean;
begin
  result := false;

  try
    FSWbemLocator := CreateOleObject('WbemScripting.SWbemLocator');
    FWMIService   := FSWbemLocator.ConnectServer('localhost', 'root\CIMV2', '', '');
    Bios:=GetWMISTring('Win32_BIOS','SerialNumber');
    Media:=GetWMISTring('Win32_PhysicalMedia','SerialNumber');
  except
    on exception do
      begin
        Bios := '';
        Media := '';
        exit;
      end;
  end;
```

```

        end;
    end;

    result := true;
end;
```

9.2 Local IP adresse

Getting the local IP address can be done in multiple ways, here is one which obtains it directly from winsocket:

```

uses Winsock;

Function GetIPAddress():String;
type
    pu_long = ^u_long;
var
    varTWSADData : TWSADData;
    varPHostEnt : PHostEnt;
    varTInAddr : TInAddr;
    namebuf : Array[0..255] of char;
begin
    If WSStartup($101,varTWSADData) <> 0 Then
        Result := 'No. IP Address'
    Else Begin
        gethostname(namebuf,sizeof(namebuf));
        varPHostEnt := gethostbyname(namebuf);
        varTInAddr.S_addr := u_long(pu_long(varPHostEnt^.h_addr_list^)^);
        Result := 'IP Address: '+inet_ntoa(varTInAddr);
    End;
    WSACleanup;
end;
```

9.3 Network adapter MAC adresse

Obtaining the active network adapter's MAC address can likewise be done in a variety of ways, here is one example calling netbios directly for the information:

```

uses nb30;

function GetMACAddress: string;
var
    NCB: PNCB;
    Adapter: PAdapterStatus;

    URetCode: PAnsiChar;
    RetCode: Ansichar;
    I: integer;
    Lenum: PlanaEnum;
    _SystemID: string;
    TMPSTR: string;
begin
    Result := '';
    _SystemID := '';
    Getmem(NCB, SizeOf(TNCB));
    Fillchar(NCB^, SizeOf(TNCB), 0);

    Getmem(Lenum, SizeOf(TLanaEnum));
```



```

Fillchar(Lenum^, SizeOf(TLanaEnum), 0);

Getmem(Adapter, SizeOf(TAdapterStatus));
Fillchar(Adapter^, SizeOf(TAdapterStatus), 0);

Lenum.Length      := chr(0);
NCB.ncb_command   := chr(NCBENUM);
NCB.ncb_buffer    := Pointer(Lenum);
NCB.ncb_length    := SizeOf(Lenum);
RetCode           := Netbios(NCB);

i := 0;
repeat
  Fillchar(NCB^, SizeOf(TNCB), 0);
  Ncb.ncb_command   := chr(NCBRESET);
  Ncb.ncb_lana_num  := lenum.lana[I];
  RetCode           := Netbios(Ncb);

  Fillchar(NCB^, SizeOf(TNCB), 0);
  Ncb.ncb_command   := chr(NCBASTAT);
  Ncb.ncb_lana_num  := lenum.lana[I];
  // Must be 16
  Ncb.ncb_callname  := '*          ';

  Ncb.ncb_buffer := Pointer(Adapter);

  Ncb.ncb_length := SizeOf(TAdapterStatus);
  RetCode        := Netbios(Ncb);
  //---- calc _systemId from mac-address[2-5] XOR mac-address[1]...
  if (RetCode = chr(0)) or (RetCode = chr(6)) then
  begin
    _SystemId := IntToHex(Ord(Adapter.adapter_address[0]), 2) + '-' +
      IntToHex(Ord(Adapter.adapter_address[1]), 2) + '-' +
      IntToHex(Ord(Adapter.adapter_address[2]), 2) + '-' +
      IntToHex(Ord(Adapter.adapter_address[3]), 2) + '-' +
      IntToHex(Ord(Adapter.adapter_address[4]), 2) + '-' +
      IntToHex(Ord(Adapter.adapter_address[5]), 2);
  end;
  Inc(i);
until (I >= Ord(Lenum.Length)) or (_SystemID <> '00-00-00-00-00-00');
FreeMem(NCB);
FreeMem(Adapter);
FreeMem(Lenum);
GetMacAddress := _SystemID;
end;

```