Using Product Activation in your application

Version 2004-03-26

Licenturion GmbH

Please direct any suggestions or questions to tech@licenturion.com.

Table of Contents

1 Overview

This document provides the information that you need to integrate Licenturion Product Activation into your applications. Let's first have a look at how this licensing scheme works.

1.1 Product Activation

The idea underlying Product Activation is to enable the use of a each copy of a software application or the use of certain features of each copy exclusively on a single computer. Establishing such an exclusive binding between a given copy and a given computer is called *activating* this copy *for this computer*. Activation for a computer requires interaction of your end-users with you or with Licenturion.

Let us assume that you were selling a shareware word processor. You would then, for example, choose to restrict the shareware version by disabling the "save" and "save as" functions. In this way an end-user would be able to install and evaluate your word processor on his or her computer but he or she would not be able to use it for any real work. In order to obtain a fully functional version, the end-user would have to pay you and ask you or Licenturion for assistance in activating the copy of your word processor for his or her computer to enable the "save" and "save as" functions - exclusively on this computer.

For classic (non-shareware) software sales activation would typically be required to use your software application at all. Activation for a single computer would then be included in the sales price and be carried out by the end-user in the course of installing your software.

So, how does this prevent piracy? After your software application has been installed on the end-user's computer he or she activates it for this computer in order to be able to use it at all or to use all of its features. Let's assume that the end-user illicitly installed your software application on a second computer. He or she would then have to activate, with your or Licenturion's assistance, the application a second time for the second computer. Noticing, however, the end-user's attempt to activate your application for a different computer, you or Licenturion would deny activation and thus prevent any illicit use of your software application on more than one computer.

In order to be able to establish a binding between a copy of a software application and a given computer, we need means to uniquely identify each distributed copy of a software application and to identify the computer that this copy is to be bound to without disclosing any sensitive information about this computer.

1.1.1 Identifying individual copies

Identifying a copy of a software application is easy. We just have to supply a unique Serial Number with each copy, e.g. by means of a sticker on the CD-ROM jewel case. For Product Activation we use Serial Numbers that are sequences of 16 characters, e.g.

P7TQ-BS7X-S9AJ-ETM9

Serial Numbers are case-insensitive. Each of the 16 characters is one of the following 26 letters and digits.

A B C D E F G H J K L M N P Q R S T W X Y Z 3 4 7 9

We neither use the letter O (it resembles a zero), nor the letter I (it could easily be mistaken for a one), nor the letters U and V (they might look similar in some fonts). Instead, we use four digits that do not resemble any letters (as in 2 vs. Z, 5 vs. S, 6 vs. G, and 8 vs. B). In addition, we recommend that you use a clearly recognizable font when printing Serial Numbers, e.g. Adrian Frutiger's OCR-B.

1.1.2 Identifying a computer

We need a similar concept to identify a single computer. The idea is to assign the computer an identifier that - if possible - is unique and that cannot be copied to other computers. Let's have a quick look at why we need these two conditions.

- When binding a copy of your software application to a single computer, the application only remembers that it has been bound to a computer with an identifier of, say, X. So, it will not run or provide its full functionality on any computer with an identifier of, say, Y. If, however, the end-user was able to easily find a second computer with an identifier of X, the copy of your software application would also work on this computer. Hence, identifiers should be unique.
- Assume that a copy of your software application has been bound to a single computer with an identifier of X. If it was possible to copy this identifier to a second computer, making this computer have the same identifier, namely X, as the first computer, your application would also work on the second computer. Hence, identifiers should not be copyable.

Achieving both goals at the same time is hard and therefore we do not use truly unique identifiers. We just take care that the identifiers of two arbitrarily chosen computers are different with a very high probability, which is enough for our purposes.

Product Activation identifies a computer by considering nine characteristics, e.g. the make and model, of a variety of hardware components contained in the computer and constructs a Hardware Hash - the identifier for a computer - from the gathered information. A Hardware Hash thus represents the hardware configuration of a computer. Note that the term *hardware configuration* comprises, in the context of this manual, only some selected hardware components and not the full hardware configuration of the computer.

As computers typically differ in many hardware components, chances that any two computers yield the same Hardware Hash are slim. In addition, copying hardware components from one computer to another is not possible. So, Hardware Hashes meet the two conditions described above.

Hardware Hashes are sequences of 12 characters, e.g.

LNKJ-BLR7-7TNZ

Like Serial Numbers, Hardware Hashes are case-insensitive. Each of the characters is selected from the set of 26 letters and digits that we also use for Serial Numbers.

The hardware components represented by the Hardware Hash and their considered characteristics are

- one of the installed harddrives - make and model
- one of the installed CD-ROM drives - make and model
- one of the installed SCSI host adapters or IDE controllers - make and model
- one of the installed graphics boards - make and model
- the first CPU in the computer - make and model, serial number
- the installed RAM - size
- one of the available disk volumes - volume serial number
- one of the installed Ethernet adapters - Ethernet address

Typically, the end-user may not specify of which harddrive, CD-ROM drive, etc., the characteristic is included in the Hardware Hash. The hardware components to be used are automatically determined. However, in customized Product Activation (see section 1.1.13) advanced end-users can themselves select the hardware components to be considered.

From each of the collected characteristics, with the exception of the CPU serial number and the Ethernet address, a numerical value between 0 and 7, i.e. a 3-bit value, is derived. The CPU serial number and the Ethernet address are mapped to numerical values between 0 and 511, i.e. a 9-bit value.

A value of 0 indicates that the corresponding characteristic is not available. If a computer, for example, did not have any CD-ROM drive installed, the value representing the make and model of one of the installed CD-ROM drives would be 0. If the installed CPU did not support a CPU serial number, the respective value would be 0. And so on. Any value different from 0 indicates that the corresponding characteristic is available. In this case the value is the result of passing a text representation of the characteristic through a hash function.

As log2(26) is roughly 4.7, each of the 12 characters of a Hardware Hash represents about 4.7 bits. A complete 12-character Hardware Hash thus represents a 56-bit value. We use big-endian "character ordering," so the first character of a Hardware Hash represents the most significant 4.7 bits.

The 40 least significant bits of the 56-bit value represent the hardware configuration. The remaining 16 bits contain a CRC-16 checksum to guard against typographic errors. The following table gives the structure of Hardware Hashes in full detail.

Bits	**Value(s)**	**Characteristic**
0 - 2	0 - 7	one of the installed harddrives (make and model)
3 - 5	0 - 7	one of the installed CD-ROM drives (make and model)
6 - 8	0 - 7	one of the installed SCSI host adapters or IDE controllers (make and model)
9 - 11	0 - 7	one of the installed graphics boards (make and model)
12 - 14	0 - 7	the first CPU in the computer (make and model)
15 - 17	0 - 7	the installed RAM (size)
18 - 20	0 - 7	one of the available disk volumes (volume serial number)
21 - 29	0 - 511	the first CPU in the computer (CPU serial number)
30 - 38	0 - 511	one of the installed Ethernet adapters (Ethernet address)
39	0 - 1	1 = the computer is a notebook computer 0 = the computer is not a notebook computer
40 - 55	0 - 65535	CRC-16 of bits 0 - 39

So, what about the end-users' privacy? Doesn't Licenturion learn which components the end-users' computers are made of? When mapping, for example, the wealth of makes and models of harddrives to only seven different values, many makes and models are mapped to the same value. Hence it is not possible to determine from a given value the make and model that the value has been derived from. It's a one-way thing. We are able to verify whether the make and model of a harddrive match the value 3. Just pass the make and model through the hash function and see whether the result is 3. Merely knowing the value 3 we are, however, unable to identify the make and model that has led to this value. For characteristics that are even more variable - i.e. CPU serial numbers and Ethernet addresses - we can even afford to use 511 different values instead of only seven. There are substantially more CPU serial numbers and Ethernet addresses than harddrive makes and models out there, because CPU serial numbers and Ethernet addresses are unique for each CPU and Ethernet adapter, respectively. Hence, dividing, for example, the CPU serial numbers into 511 groups instead of only seven groups does no harm to the end-users' privacy. There are still enough CPU serial numbers that map to the same value.

So, it is highly likely that two different computers yield two different Hardware Hashes, but a Hardware Hash nevertheless does not disclose which hardware components the corresponding computer contains.

1.1.3 The activation process

During activation an end-user supplies the Serial Number identifying his or her copy of your software product to be activated and the Hardware Hash identifying his or her computer for which this copy is to be activated to the Licenturion server. The corresponding URL is

http://act.licenturion.com

The Licenturion server then generates an *Activation Code* linking the Serial Number and the hardware configuration as represented by the Hardware Hash, i.e. not the full hardware configuration of the computer but the components selected for inclusion in the Hardware Hash. The Activation Code thus links the given copy of your software product (identified by the Serial Number) to the computer for which it is to be activated (identified by the Hardware Hash). It constitutes the binding between these two that has been introduced at the beginning of section 1.1. Activation Codes are sequences of 32 characters, e.g.

MLXB-GNF9-TGND-34BW-PS4K-KCC3-FDWA-GRX4

Activation Codes are case-insensitive. Each character is taken from the set of 26 characters that is also used for Serial Numbers and Hardware Hashes.

The end-user then supplies the Activation Code to your application, the application establishes the validity of the Activation Code, and from that point considers itself or some of its features activated.

This process is completely transparent and enables end-users to verify that their privacy is respected. The only information transmitted to the Licenturion server obviously consists of the Serial Number and the Hardware Hash. The former is supplied with your application and thus cannot contain any personal information. The contents of the latter are fully documented in section 1.1.2 and, accordingly, represent the hardware configuration of the end-users' computers without undermining their right to privacy.

While this manual process is fully transparent, it is not very comfortable - although 32-character Activation Codes are relatively benign when compared to other people's solutions. We therefore expect the vast majority of end-users to perform automatic activation. With automatic activation your application asks the end-user for his or her Serial Number, determines the Hardware Hash for the computer on which it is running, automatically transmits the Serial Number and the Hardware Hash to the Licenturion server, receives the corresponding Activation Code from the server, and activates itself without further intervention by the end-user.

This mechanism is also suitable for easily integrating the manual activation process into your own website. The idea is that your web-server collects the Serial Number and the Hardware Hash from the end-user, transmits the information to our web-server in the described way, receives the Activation Code or an error message, and displays it to the end-user.

The following table summarizes those parts of the manual activation process that are visible to an end-user. The "Software application" column describes the end-user's interaction with his or her copy of your software application, the "Licenturion server" column describes the end-user's interaction with the Licenturion server.

Step	Software application	Licenturion server
1	The software application creates and displays the Hardware Hash.	-
2	-	The end-user enters his or her Serial Number and the Hardware Hash.

Step	Software application	Licenturion server
3	-	The server returns the Activation Code corresponding to the Serial Number and the hardware configuration represented by the Hardware Hash.
4	The end-user enters the Activation Code.	-

The table for automatic activation is considerably shorter.

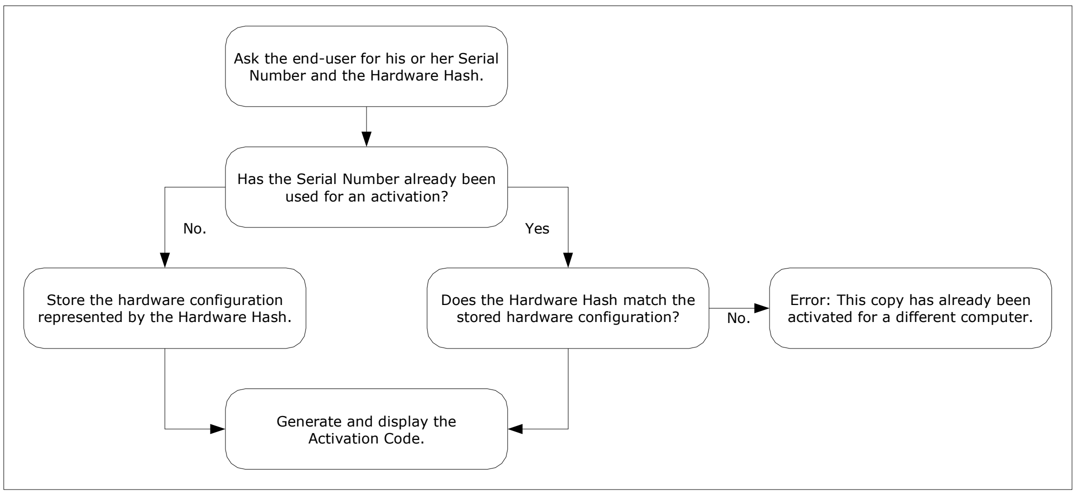
Step	Software application	Licenturion server
1	The end-user enters his or her Serial Number.	-

When a Serial Number is used for activation for the first time, the hardware configuration associated with it is stored by the Licenturion server. Subsequent activations are only allowed as long as they are related to the same hardware configuration, i.e. to the same computer.

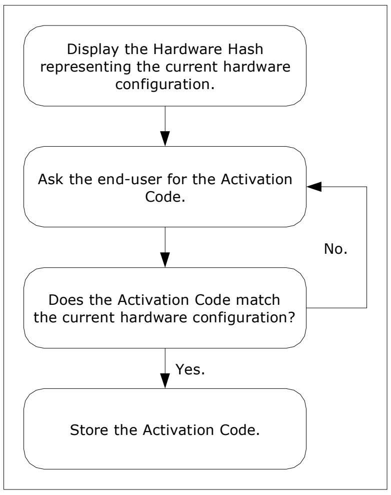
This restriction is imposed to prevent end-users from using the same Serial Number over and over again to successively activate your software application for as many computers as they like instead of the intended single computer. If the Licenturion server notices that the Serial Number entered by an end-user has already been used in an activation for a different computer, i.e. in conjunction with a different hardware configuration, it refuses to give out an Activation Code and returns an error message instead.

So, why does the Licenturion server allow multiple activations for the same computer? Let's assume that the end-user reformats his or her harddrive and thus loses all stored information, in particular the information that his or her copy of your software application has already been activated. When reinstalling his or her copy of your software application, the end-user will have to activate it again. As this is clearly something to allow, the Licenturion server grants end-users as many activations as they like, as long as the supplied hardware configuration does not change, i.e. as long as the activations refer to the same computer.

Actually the whole process is a bit more complica..., err, sophisticated, but we will neglect the details for now. Just remember that in case of a first-time activation the supplied hardware configuration is stored by the Licenturion server. The following chart illustrates the manual activation process as described up to this point. The automatic activation process works analogously.


Ask the end-user for his or her Serial Number and the Hardware Hash.
Has the Serial Number already been used for an activation?
No.
Yes
Store the hardware configuration represented by the Hardware Hash.
Does the Hardware Hash match the stored hardware configuration?
No.
Error: This copy has already been activated for a different computer.
Generate and display the Activation Code.

After the end-user has entered a valid Activation Code matching the current hardware configuration, the software application will store it for later reference. The following chart summarizes the activation process from the perspective of the software application.


Display the Hardware Hash representing the current hardware configuration.
Ask the end-user for the Activation Code.
No.
Does the Activation Code match the current hardware configuration?
Yes.
Store the Activation Code.

1.1.4 Server Cookies

Automatic activation employs a simple HTTP-based request/response protocol for the communication with the Licenturion server and respects the proxy settings of Internet Explorer. We do not use SSL, since HTTPS is often blocked by corporate firewalls as encrypted traffic cannot be content-scanned. Protection against eavesdroppers that try to harvest valid Serial Numbers in front of the Licenturion server by analyzing the incoming network traffic is supplied by a Diffie-Hellman scheme instead.

The data transmitted to the Licenturion server during automatic activation is called a Server Cookie. A Server Cookie consists of 174 bytes, as described in the following table.

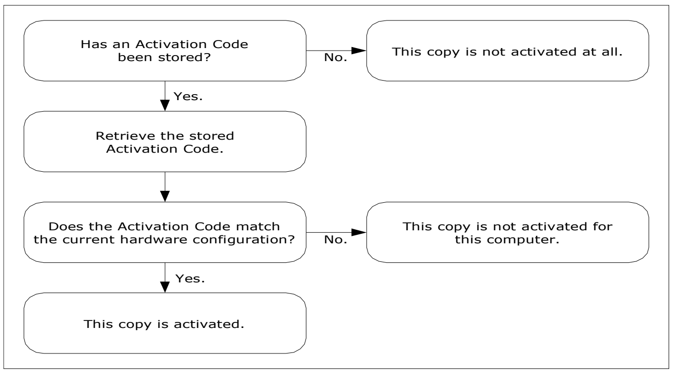
Bytes	Contents
0 - 3	Product ID represented as a 32-bit value (little endian byte-order)
4 - 7	Cryptographic checksum for the following bytes (verified by the server)
8 - 27	ASCII representation of the Serial Number (encrypted with the shared Diffie-Hellman secret).
28 - 42	ASCII representation of the Hardware Hash (encrypted with the shared Diffie-Hellman secret).
43 - 170	the public Diffie-Hellman key of the client (used by the server to create the shared Diffie-Hellman secret)
171 - 173	Length of the additional data represented as a 24-bit value (little endian byte-order, currently unused and set to zero)

The Server Cookie is converted into an ASCII representation of 2 x 174 = 348 characters and transmitted to the Licenturion server inside a POST request as a URL-encoded parameter named "serverCookie". The ASCII conversion simply maps the two four-bit halves of each byte to the sixteen letters 'A' through 'P'. The most significant four bits are stored first. The byte value 0x21 would, for example, thus map to the 2-character sequence "CB".

The Licenturion server returns a 4-line ASCII file that contains status information and, if activation is granted, the Activation Code corresponding to the data contained in the Server Cookie.

1.1.5 Am I activated?

An installed copy of a software application considers itself to be activated, if an Activation Code matching the current hardware configuration has been stored. The following chart illustrates the behavior of the software application when determining whether it is activated.


Has an Activation Code been stored?
No.
This copy is not activated at all.
Yes.
Retrieve the stored Activation Code.
Does the Activation Code match the current hardware configuration?
No.
This copy is not activated for this computer.
Yes.
This copy is activated.

1.1.6 Security

Imagine that a software pirate was able to generate valid Serial Numbers himself or herself. He or she would then be able to happily produce loads of fresh Serial Numbers and use each Serial Number to activate the same copy of your software application for a different computer. Fortunately, protecting Serial Numbers is easy as their validity is only verified by the Licenturion server. All 16-character sequences that constitute valid Serial Numbers share the same construction plan, i.e. they all share a common structure. If a given sequence complies with the construction plan, it is considered to be a valid Serial Number. If it does not, it is considered to be random garbage. This construction plan can be kept securely on the server and is never disclosed to the outside world.

Speaking more technically, Serial Numbers are protected by a message authentication code. The secret construction plan corresponds to the secret cryptographic key used to create and verify the message authentication code on the Licenturion server.

Now imagine that a software pirate was able to generate valid Activation Codes matching given hardware configurations. He or she would then be able to perform activations himself or herself, without the assistance - and thus without the control - of the Licenturion server. That's why protecting Activation Codes is equally important, although a bit more complicated. All Activation Codes share a single construction plan that describes how to derive a valid Activation Code from a given hardware configuration. However, since Activation Codes are validated by your software application and not by the Licenturion server, we have the following problem. In order to be able to check whether a 32-character sequence given by an end-user is a valid Activation Code, i.e. based on a given hardware configuration and generated according to the construction plan, your application needs to know the construction plan underlying the creation of valid Activation Codes. However, if your application knows the construction plan, a software pirate can extract the construction plan from your application, analyze it and use the gained insight to write a program that illicitly generates sequences that your application will consider valid Activation Codes.

That's why the idea underlying Activation Codes as implemented by Licenturion Product Activation is to keep their construction plan secret and equip your application only with the ability to determine whether a sequence given by the end-user has been derived from a given hardware configuration in compliance with the secret construction plan - without, however, actually knowing the secret construction plan. In this way, the ability to create valid Activation Codes is separated from the ability to verify whether a given sequence is a valid Activation Code for a given hardware configuration. A pirate analyzing your application can, at maximum, learn how to verify Activation Codes. He or she cannot find out how to generate valid Activation Codes.

This separation of creation and verification of Activation Codes is based on public key cryptography. An Activation Code is a digital signature over a message containing the hardware configuration to which your application is to be bound. The secret creation plan corresponds to the secret key employed during signature creation, the ability to verify Activation Codes corresponds to knowledge of the public key for signature verification. The rocket science part of the whole story is that Activation Codes as implemented by Licenturion

Product Activation consist of only 32 characters, while conventional digital signatures would typically result in Activation Codes with a length of more than 200 characters.

1.1.7 Tolerating hardware modifications

Sometimes end-users add new hardware components to their computers, exchange old hardware components for new and better ones, or remove hardware components that they do not need any longer. Implementing the activation mechanism as described up to this point would have two major implications in case the hardware of a given computer was modified.

- All software applications installed on the computer would have to be activated again as the stored Activation Code would not match the new hardware configuration any longer (see section 1.1.5).
- Still worse, the Licenturion server would deny activation as the new hardware configuration would not match the hardware configuration stored for the affected end-user's Serial Number during first-time activation (see section 1.1.3).

1.1.8 The scoring mechanism

Obviously, we need a way to tolerate hardware modifications. Licenturion Product Activation offers a configurable scoring mechanism that enables you to specify up to which point hardware changes are to be tolerated. This mechanism is employed by

- your software application to determine whether it still considers itself activated after the hardware modifications and
- the Licenturion server to determine whether to grant activation in spite of the modified hardware configuration.

To implement the scoring mechanism the present hardware configuration is compared to the original hardware configuration.

- In case of your software application the original hardware configuration is the hardware configuration at the time of its activation. So, how does your software application know the original hardware configuration? When we described the activation process we oversimplified a bit. During activation the software application does not only store the Activation Code, but also the Hardware Hash representing the current, i.e. the original, hardware configuration.
- In the case of the Licenturion server the original hardware configuration is the hardware configuration supplied for first-time activation. As explained before, it is stored by the server to be considered for subsequent activations.

As a result of the comparison of the present hardware configuration and the original hardware configuration each characteristic is assigned one of the following five states.

State	Description
present and matching	The characteristic has been available in the hardware configuration at activation time, it is still available in the present hardware configuration, and it has not changed. **Example:** A computer has had a CD-ROM drive at activation time and the CD-ROM drive has not been exchanged or removed since then.
missing and matching	The characteristic neither has been available in the hardware configuration at activation time nor is it available in the present hardware configuration. **Example:** A computer has not had a CD-ROM drive at activation time and it still does not have a CD-ROM drive.

State	Description
added	The characteristic has not been available in the hardware configuration at activation time but it is available in the present hardware configuration. **Example:** A computer has not had a CD-ROM drive at activation time but a CD-ROM drive has been added since then.
removed	The characteristic has been available in the hardware configuration at activation time but it is not available in the present hardware configuration. **Example:** A computer has had a CD-ROM drive at activation time but the CD-ROM drive has been removed since then.
changed	The characteristic has been available in the hardware configuration at activation time, it is still available in the present hardware configuration, but it has changed. **Example:** A computer has had a CD-ROM drive at activation time but the CD-ROM drive has been exchanged for another CD-ROM drive model since then.

When comparing the present hardware configuration to the original hardware configuration an individual score value is assigned to each characteristic that is either present and matching or missing and matching. Characteristics that are added, removed, or changed, are assigned an individual score value of zero. Then the total score for the present hardware configuration is calculated by adding the nine individual score values. So, the more characteristics are present and matching or missing and matching, the higher the total score. If the obtained total score is greater than a configurable threshold, the hardware configuration is considered to identify the same computer as the original hardware configuration.

Let's have a closer look at what "present and matching" means. Assume that an end-user's computer contains only one harddrive when activating his or her copy of your software application for the first time. So, this harddrive's characteristic is included in the original hardware configuration. Later he or she adds a second harddrive to his or her computer. This, however, will **not** change the "present and matching" status of the harddrive characteristic. As long as the first harddrive is not removed or exchanged, the corresponding characteristic will always remain present and matching. Generally, as long as the harddrive characteristic contained in the original configuration matches **any** of the installed harddrives, it will remain present and matching. Adding harddrives will therefore never change the state of a characteristic.

This is not only the case for harddrives but for all characteristics in the hardware configuration. As long as end-users just add hardware components, all characteristics will remain present and matching.

1.1.9 Assigning individual score values

While we want to allow end-users to modify their hardware configuration, we definitely do not want to be too tolerant. Imagine the following scenario. Audrey legitimately buys a copy of your software application. She activates it using the Serial Number that you have supplied to her. Audrey then hands her copy along with her Serial Number to her friend Donna. Donna installs Audrey's copy on her computer and tries to activate it for this computer. If the Licenturion server was too tolerant with respect to hardware changes, it would possibly fail to recognize that it is now handling an activation for a different computer, handle Donna's computer as if it were Audrey's computer with some hardware modifications, and let Donna successfully activate.

We cannot completely eliminate this problem. However, by carefully selecting the parameters of the scoring mechanism, we can substantially reduce the probability that two arbitrarily chosen computers are erroneously considered to be the same computer by the activation mechanism. Our solution is to assign the individual score values based on probabilities. The lower the probability for an arbitrarily chosen computer to yield "missing and matching" or "present and matching" for a given characteristic, the higher the associated individual score value.

Let's have a look at Ethernet addresses to discuss this idea. It is still relatively probable today that neither Audrey's nor Donna's computer has an Ethernet adapter. So, two hardware configurations not including an Ethernet adapter, i.e. the corresponding characteristic missing and matching, is not a strong indicator that these hardware configurations represent the same computer. Simply because it is relatively probable that two arbitrarily chosen computers both do not have an Ethernet adapter.

However, if the hardware configurations both contain an Ethernet adapter and the corresponding characteristic is the same in both hardware configurations, this is a strong indicator that these two hardware configurations represent the same computer. Remember that a present Ethernet adapter results in one of 511 different values for the corresponding characteristic. So, it is relatively improbable that the Ethernet adapters of two arbitrarily chosen computers have the same characteristic.

That's why a missing and matching Ethernet adapter characteristic by default contributes an individual score of 1 to the total score, whereas a present and matching Ethernet adapter characteristic contributes by default the much higher individual score of 3. The same is true for CPU serial numbers.

For other characteristics, e.g. the characteristic representing one of the installed harddrives, a status of missing and matching is relatively unlikely. After all, most computers do contain at least one harddrive. That's why these characteristics are by default given an individual score value of 2 in case their status is missing and matching.

The two individual score values for missing and matching or present and matching characteristics, respectively, can be set on the Licenturion server. The configured values are considered by the Licenturion server when deciding whether to grant an activation, i.e. whether a hardware configuration passed with a Serial Number is sufficiently similar to the hardware configuration supplied during first-time activation for this Serial Number. They are also used by your software application to determine whether it still considers itself activated after hardware changes, i.e. whether the new hardware configuration is sufficiently similar to the original hardware configuration.

The defaults for the individual score values are listed in the following table.

Characteristic	**present and matching**	**missing and matching**
one of the installed harddrives (make and model)	1	2
one of the installed CD-ROM drives (make and model)	1	1
one of the installed SCSI host adapters or IDE controllers (make and model)	1	2
one of the installed graphics boards (make and model)	1	2
the first CPU in the computer (make and model)	1	1
the installed RAM (size)	1	2
one of the available disk volumes (volume serial number)	1	2
the first CPU in the computer (CPU serial number)	3	1
one of the installed Ethernet adapters (Ethernet address)	3	1

To give you an idea how likely it is that, based on your configured individual score values and the set threshold, Audrey's computer is sufficiently similar to Donna's computer to subvert the protection scheme by reusing Audrey's Serial Number to illicitly activate Audrey's copy of your software application also on Donna's computer, the Licenturion server calculates four probabilities. Audrey's hardware configuration is assumed to be the original hardware configuration to which Donna's hardware configuration is compared.

It is assumed that both computers contain at least one harddrive, CD-ROM drive, SCSI host adapter or IDE controller, graphics board, CPU, and disk volume, as well as some RAM. So, when comparing the first computer's hardware configuration to the second computer's hardware configuration, the first seven of the above nine characteristics will either be in the "present and matching" state or in the "changed" state. The four probabilities are then calculated under the following four individual assumptions for the remaining two characteristics.

1. Neither computer's CPU supports a CPU serial number and neither computer contains an Ethernet adapter, i.e. the CPU serial number characteristic and the Ethernet adapter characteristic are missing and matching. For the default individual score values given above and the default threshold this probability is 1.02%.
2. Both computer's CPUs support a CPU serial number but neither computer contains an Ethernet adapter, i.e. the CPU serial number characteristic is either present and matching or changed and the Ethernet adapter characteristic is missing and matching. For the default individual score values given above and the default threshold this probability is 0.15%.
3. Neither computer's CPU supports a CPU serial number but both computers contain an Ethernet adapter, i.e. the CPU serial number characteristic is missing and matching and the Ethernet adapter characteristic is either present and matching or changed. For the default individual score values given above and the default threshold this probability is 0.15%.
4. Both computer's CPUs support a CPU serial number and both computers contain an Ethernet adapter, i.e. the CPU serial number characteristic and the Ethernet adapter characteristic are either present and matching or changed. For the default individual score values given above and the default threshold this probability is 0.03%.

The four probabilities are displayed by the Licenturion server in a 2 by 2 *coincidence matrix*.

1.1.10 Notebook computers

Although notebook computers are hardly ever upgraded, inserting or removing PCMCIA cards and plugging or unplugging the notebook computer from a docking station can lead to substantial hardware changes. Product Activation is therefore typically more tolerant for notebook computers than it is for desktop computers. As described before two hardware configurations are considered to refer to the same computer if the total score obtained by adding the nine individual score values is higher than a given threshold. This threshold is actually two thresholds. One threshold is used for desktop computers and the second threshold is used for notebooks.

Like the individual score values the threshold values can be set on the Licenturion server. Again, the configured values are considered by the Licenturion server when deciding whether to grant an activation and by your software application to determine whether it still considers itself activated after hardware changes.

The default threshold for notebook computers is 3, the threshold for desktop computers defaults to 6. If the comparison of two hardware configurations yields a score of 3 or more in the case of a notebook or 6 or more in the case of a desktop, the two hardware configurations are considered to identify the same computer.

When creating the coincidence matrix, the threshold for desktop computers is used to calculate the four required probabilities.

1.1.11 Activating for more than one computer

Every now and then one of your end-users may want to deinstall your software application from one computer and reinstall it on another computer. As this is typically not an every day situation, allowing your end-users to do so every few weeks is typically sufficient.

For this purpose the Licenturion server offers you to specify the duration of a reset period for your Serial Numbers. The reset period starts when a Serial Number is used for first-time activation. Until the end of the reset period it is possible to use the Serial Number again for activation only if the hardware configuration specified for an activation is, as detailed before, sufficiently similar to the hardware configuration stored by the Licenturion server during first-time activation. However, after the reset period is over, the next activation for the Serial Number will be considered a first-time activation again, the provided hardware configuration will be stored replacing the previously stored hardware configuration, and a new reset period will start. The default duration of the reset period is 60 days. So the Licenturion server by default basically forgets every 60 days about all previous activations for a Serial Number.

If you prefer to be still more tolerant, you may offer your end-users to perform more than one first-time activation during the reset period. Suppose that an end-user tried to activate with a hardware configuration substantially differing from the hardware configuration given during first-time activation and that the reset period was not over. Instead of denying the activation, the end-user would be allowed to override the denial for a specified number of times per reset period. Each activation granted by this mechanism is considered a first-time activation, i.e. the new provided hardware configuration is stored replacing the previously stored hardware configuration, with the only difference that the reset period is not restarted.

As this feature can be abused by your end-users, it is disabled by default by setting the maximal number of first-time activations per reset period to 1. This number can, however, be increased.

1.1.12 Modifying the counter of first-time activations

It is conceivable that one of your end-users nevertheless requires more first-time activations than he or she is allowed in the running reset period. In this case, he or she would have to contact you and ask you to reset or decrease the counter of first-time activations for his or her Serial Number. You can easily accomplish this via the Licenturion server.

1.1.13 Customized Product Activation

You can offer advanced end-users to further tailor the activation process to their needs. The idea is to let an end-user specify which hardware components are used to form the hardware configuration instead of having the activation mechanism perform this selection automatically.

Let us assume that an end-user uses a removable harddrive for data transfer between different computers. He or she could then ensure that this harddrive is not included in the hardware configuration, since removing it would otherwise modify the hardware configuration.

1.1.14 Activating individual features of a software application

Up to now we have considered a software application to be a monolithic block. We would only activate the complete software application or nothing at all. However, Licenturion Product Activation also offers to activate individual parts or features of a piece of software. For this purpose, Activation Codes contain a 32-bit payload, which can be extracted by the software application from a given Activation Code.

The payload is opaque to the Licenturion server. It does not need to know how to interpret the payload. It just knows that the payload is a 32-bit value and can therefore be given a set of rules along the lines of "If somebody activates using Serial Number A, then return an Activation Code that has the 32-bit value X as its payload, if somebody activates using Serial Number B, then return an Activation Code that has the 32-bit value Y as its payload, ..."

The interpretation of the payload is up to the software application. It could extract the payload from an Activation Code and, for example, use it as a bit-mask, each of the 32 bits enabling or disabling a certain feature of the application.

1.1.15 Time-limited Product Activation

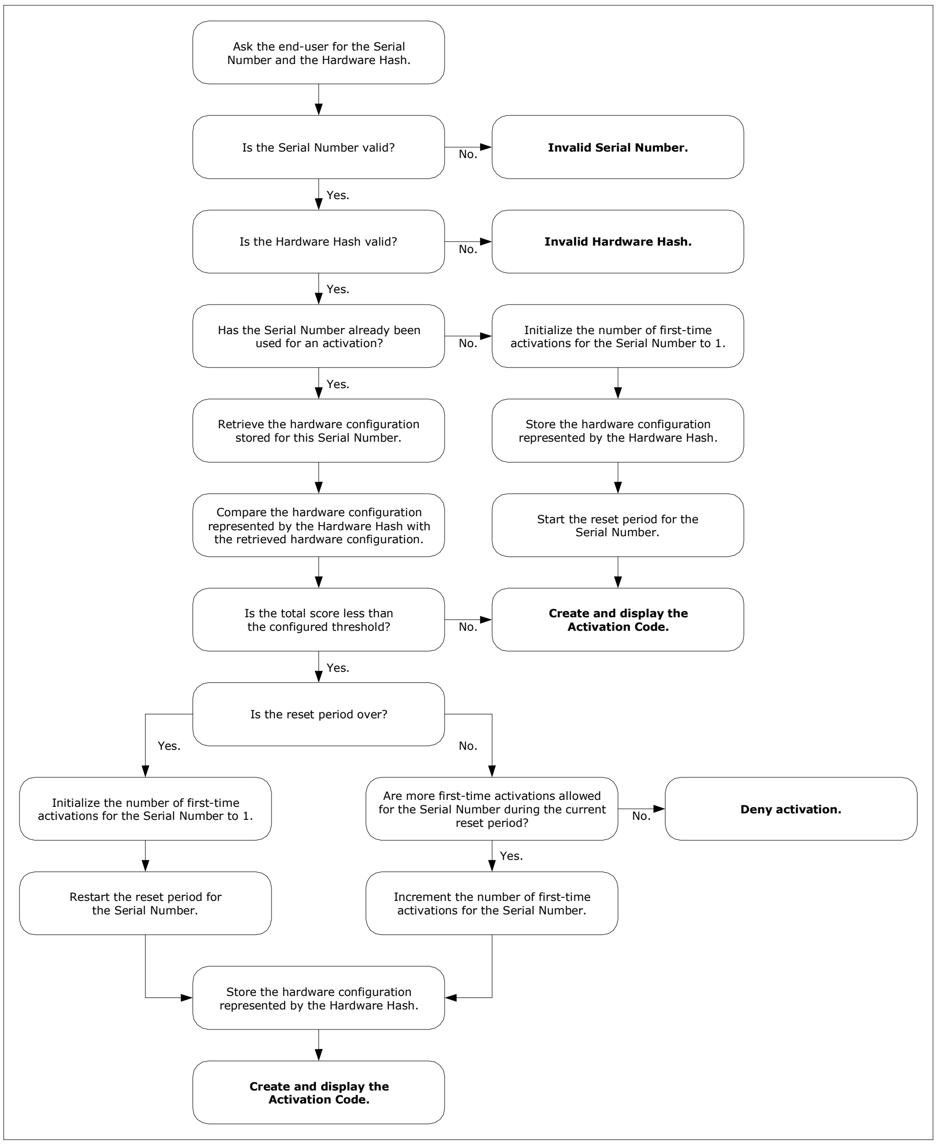
The server can also be instructed to use the most significant 13 bits of the 32-bit payload for storing an expiration date in the Activation Code, still leaving the least significant 19 bits for user-defined content. After the expiration date of an Activation Code is reached, an activation performed with this Activation Code becomes invalid and the software application returns into its original state before activation. These temporary activations are typically used in shareware scenarios to implement trial periods, during which a fully activated version of a piece of software can be evaluated at no cost.

The expiration date is specified as an absolute point in time, e.g. "valid until February 1st, 2003" - as opposed to "valid for 30 days." This prevents a quite common kind of attack that simply resets the software application's idea of how many days have passed since activation to zero and thus extends trials periods *ad infinitum*.

1.2 The complete picture

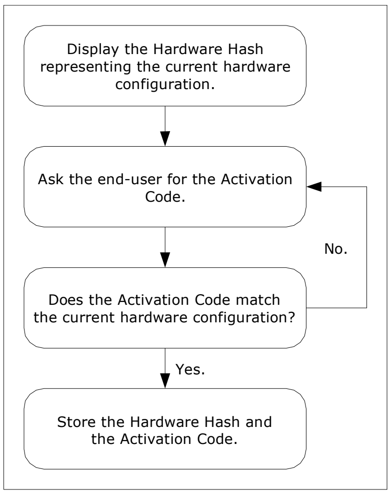
Let's summarize everything that we now know about the activation process.

1.2.1 Activation on the server side



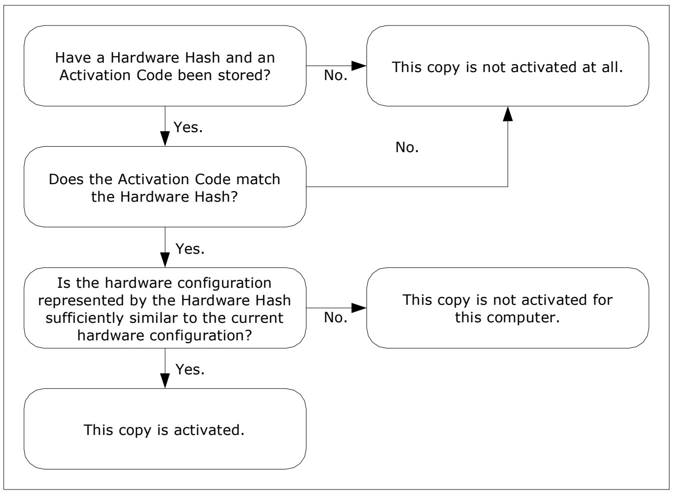
Ask the end-user for the Serial Number and the Hardware Hash.
Is the Serial Number valid?
No.
Invalid Serial Number.
Yes.
Is the Hardware Hash valid?
No.
Invalid Hardware Hash.
Yes.
Has the Serial Number already been used for an activation?
No.
Initialize the number of first-time activations for the Serial Number to 1.
Yes.
Retrieve the hardware configuration stored for this Serial Number.
Store the hardware configuration represented by the Hardware Hash.
Compare the hardware configuration represented by the Hardware Hash with the retrieved hardware configuration.
Start the reset period for the Serial Number.
Is the total score less than the configured threshold?
No.
Create and display the Activation Code.
Yes.
Is the reset period over?
Yes.
No.
Initialize the number of first-time activations for the Serial Number to 1.
Are more first-time activations allowed for the Serial Number during the current reset period?
No.
Deny activation.
Yes.
Restart the reset period for the Serial Number.
Increment the number of first-time activations for the Serial Number.
Store the hardware configuration represented by the Hardware Hash.
Create and display the Activation Code.

1.2.2 Activation on the software application side



Display the Hardware Hash representing the current hardware configuration.
Ask the end-user for the Activation Code.
No.
Does the Activation Code match the current hardware configuration?
Yes.
Store the Hardware Hash and the Activation Code.

1.2.3 Am I activated?


Have a Hardware Hash and an Activation Code been stored?
No.
This copy is not activated at all.
Yes.
No.
Does the Activation Code match the Hardware Hash?
Yes.
Is the hardware configuration represented by the Hardware Hash sufficiently similar to the current hardware configuration?
No.
This copy is not activated for this computer.
Yes.
This copy is activated.

1.3 Implementation methods

Licenturion Product Activation functionality can be made available to your software application as

- a COM component (implemented by licact.dll),
- a dynamic link library (also implemented by licact.dll, licact.lib is the corresponding import library), or
- a static library (implemented by licacts.lib).

For C and C++ programmers using the dynamic link library (DLL) or the static library, the header file licact.h is supplied. All files can be found in the top-level directory of this ZIP archive.

IMPORTANT: If you choose to integrate Product Activation into your application using the COM approach or the DLL approach, you have to supply your licact.dll file to your end-users in the distribution package of your application. **Never** install your licact.dll file into any folder shared with other applications, e.g. the Windows folder or the System32 folder! **Never! Jamais! Niemals!** Always use the folder into which you install the executable file(s) of your application. Because here's what is going to happen otherwise according to Murphy's law, should you decide, for example, to install your licact.dll file into the System32 folder: In addition to your application the end-user will also install another vendor's application, which is also protected by Product Activation, which includes the other vendor's version of the licact.dll file, which is also installed into the System32 folder. Obviously, the other vendor's DLL will overwrite your DLL during the installation of the other vendor's application. The problem is now that **every licact.dll file is unique**. Your DLL only recognizes your Activation Codes and the other vendor's DLL only recognizes that vendor's Activation Codes. Technically speaking, your DLL contains your public key and the other vendor's DLL contains the other vendor's public key. In addition, your DLL contains your individual score values and the other vendor's DLL contains the other vendor's individual score values. So, if your DLL is overwritten with the other vendor's DLL, things will get seriously messed up and your application will not be able to recognize your Activation Codes any longer. So, by installing your licact.dll file into a private location such as the installation folder of your executable file(s), you ensure that your licact.dll file is not touched by anyone else and that your application always uses your licact.dll file.

IMPORTANT: Your DLL contains your individual score values. They are used when applying the previously described scoring mechanism to determine whether your software application is activated for the computer that it is running on. If you **change the individual score values** on the Licenturion server be sure to **download the updated ZIP archive** containing the new DLL with the updated individual score values!

Each of the unique versions of the licact.dll file is unambiguously identified by a Product ID. To find out which Product ID your licact.dll file has been assigned, have a look at the PersonalInfo.txt ASCII text file in the top-level directory of this ZIP archive. You will find a string of eight hex digits. This is your Product ID.

1.3.1 The LicAct COM component

Before a COM component is available it must be registered. The LicAct COM component supports self-registration. To trigger self-registration of the component use licreg.exe, which can also be found in the top-level directory of this ZIP archive. It is invoked as follows, either from the command prompt or via "Run..." in the Windows start menu:

```
licreg.exe path-to-licact-dll
```

If you omit the path-to-licact-dll part then licreg.exe will display a file selection dialog box allowing you to specify the correct path for the licact.dll file.

Let us assume that you have unpacked the ZIP archive to drive D:. The corresponding invocation of licreg.exe would then be as follows:

D:\LicActSDK\licreg.exe D:\LicActSDK\licact.dll

If you invoked licreg.exe without the path argument, i.e. as

D:\LicActSDK\licreg.exe

then a file selection dialog box would appear, allowing you to specify the correct path for the licact.dll file.

To unregister the LicAct COM component again, use licunr.exe, which is also located in the top-level directory of this ZIP archive. It is used in exactly the same way as licreg.exe, the only difference being that it causes the LicAct component to be unregistered instead of it being registered.

IMPORTANT: Keep in mind that you must also register the LicAct COM component during the installation process of your application on your end-users' computers and unregister it during uninstallation. If your installation program does not support self-registering COM components, you will have to run licreg.exe and licunr.exe in the way described above during installation and uninstallation, respectively, to ensure that your LicAct COM component is correctly registered and unregistered.

The LicAct COM component includes a type library which is also registered and unregistered via the self-registration mechanism. The LicAct type library is what is typically visible to you in your COM-aware development environment, e.g. Visual Basic. It is identified by a string that looks as follows

LicActLib 1.0 Type Library [XXXXXXXX]

where the eight "X" characters represent the Product ID. Let us assume your PersonalInfo.txt file tells you that your Product ID is 1234ABCD. Your type library would thus be named

LicActLib 1.0 Type Library [1234ABCD]

This name can also be found in your PersonalInfo.txt file. It is called the "Type Library Help String".

The type library is embedded in the licact.dll file.

IMPORTANT: Double-check that you are using the correct type library! If more than one LicAct COM component have been registered on a computer, e.g. by other vendors also using Product Activation or by yourself using Product Activation for more than one product, all LicAct type libraries will be listed in your COM-aware development environment, each with a different Product ID between the square brackets. Make absolutely sure that you use the type library bearing your Product ID and not somebody else's!

If you intend to use the COM component without using the type library, the PersonalInfo.txt file contains additional information, such as the class ID or the interface ID for the LicAct object.

IMPORTANT: To register successfully the type library requires OLEAUT32.DLL, version 2.20 or better to be installed. The very first release of Windows 95 included version 2.1 of this DLL, whereas Windows 95 OSR2 (Windows 95 B) fortunately contained version 2.20 already. The problem is that today's tools produce type libraries in a format that is not supported by OLEAUT32.DLL versions below 2.20. On affected Windows 95 systems registration of the component will fail. In this case update OLEAUT32.DLL, e.g. by installing Internet Explorer 3.0 or later or the redistributable DCOM95 update, version 1.3, which is available from the Microsoft website.

1.3.2 The LicAct dynamic link library (DLL)

In addition to the LicAct COM component, the licact.dll file also accommodates classic DLL functionality. All functions accessible via COM are also available through the standard DLL mechanism, i.e. they are exported by licact.dll. For C and C++ development, we supply the header file licact.h along with the import library licact.lib in the top-level directory of this ZIP archive.

IMPORTANT: Include the standard windows.h header file before including licact.h in your source code, since licact.h requires the definition of HINSTANCE and HWND.

IMPORTANT: As mentioned above, make absolutely sure that your application uses the correct licact.dll file by installing your licact.dll file into the same folder as the executable file (s) of your application on your end-users' computers. Never use any shared folders such as the Windows folder or the System32 folder.

As an additional protective measure all functions exported by licact.dll take the Product ID of your licact.dll file as their first argument. Each function of licact.dll then verifies whether the Product ID passed by your application matches the Product ID of the licact.dll file. If a mismatch is detected, an error is returned. This has the following effect. Let us assume that your licact.dll file has a Product ID of 1234ABCD. Your application therefore passes 1234ABCD as the first argument to all functions in your licact.dll and licact.dll notices on each function call that the Product ID is correct. Let us now assume that, perhaps because your licact.dll file was accidentally overwritten by the end-user with another vendor's licact.dll file in spite of all the care you have taken, your application erroneously invokes a function in the wrong licact.dll which has a Product ID of, say, 2345BCDE. So, your application still passes 1234ABCD - but to the wrong licact.dll. The wrong licact.dll will then detect the mismatch between the passed Product ID of 1234ABCD and the expected Product ID of 2345BCDE and return an error code to your application stating that 1234ABCD is not what it expected the application to pass.

1.3.3 The LicAct static library

A static library is supplied for use by C or C++ developers who prefer the Product Activation functionality to reside inside their executable files instead of the separate licact.dll file. The static library licacts.lib exports the same functions as the dynamic link library licact.dll. The function declarations are identical and, hence, the header file licact.h used for the dynamic link library also applies to the static library.

The only difference is that the static library needs to be initialized before any of its functions is invoked. In the DLL case, initialization is automatically performed inside the DllMain() function. To initialize the static library we mimic what DllMain() does - which is calling the __LicInitContext() function. Note the **two underscores** at the beginning of the function name. __LicInitContext() takes the instance handle of the running executable as its first argument. The second argument is the address of a pointer named __LicContext. Again, note the **two underscores**. The required declarations are contained in the licact.h header file.

IMPORTANT: Include the standard windows.h header file before including licact.h in your source code, since licact.h requires the definition of HINSTANCE and HWND.

The following example illustrates the use of __LicInitContext() in a typical C program.

```
#include <windows.h>
#include <licact.h>

int WINAPI WinMain(HINSTANCE Inst, HINSTANCE Prev, LPSTR Cmd, int Show)
{
  __LicInitContext(Inst, &__LicContext);

  /*
   *  [... more code ...]
   */
}
```

IMPORTANT: You may be required to add certain resources to executable files that are linked against the static library. See section 3.2 for details.

The code in the static library requires functions from KERNEL32.DLL, USER32.DLL, GDI32.DLL, ADVAPI32.DLL, NETAPI32.DLL, and SHELL32.DLL. Be sure to link your application against the corresponding import libraries.

1.4 Available APIs

No matter which implementation method you chose, you always have two APIs at your disposal, the standard API and the advanced API.

1.4.1 Standard API

The standard API is a high-level interface, i.e. it offers relatively powerful functions that do a lot of things in one fell swoop. You could, for example, invoke only one function and watch the standard API open a dialog box telling the end-user his or her Hardware Hash and asking for his or her Activation Code, return a "canceled" result code if he or she pushes the cancel button of the dialog box, otherwise determine whether the letters and digits entered by him or her constitute an Activation Code that matches the current hardware configuration, and then return an "everything OK" or "invalid Activation Code" result code, respectively. While it is possible to customize the appearance of the GUI, the underlying program logic will always be the same. Still, we expect the standard API to meet the needs of the majority of developers. It is light-weight and it should be possible to implement Product Activation functionality within much less than an hour.

IMPORTANT: If you use the standard API in conjunction with the licacts.lib static library be sure to link the required dialog box resources to your executable. The necessary resources can be taken from the actres.rc and resource.h files in the Src subdirectory of this ZIP archive. See section 3.2 for more information on this subject.

1.4.2 Advanced API

For developers that prefer to take care of GUI-related things themselves or that require customized program logic, the advanced API is a better choice. It is a low-level interface, i.e. it offers functions that carry out limited and very specific tasks, like determining whether a given string constitutes a valid Activation Code.

1.4.3 API conventions

The names of all functions exported by the DLL and the static library start with the two-letter prefix "Lic" to prevent name clashes with other people's libraries. The functions exposed by the COM component do not bear this prefix. Apart from that the function names used by the DLL, the static library, and the COM component are all the same.

With the exception of ErrorString() (when using the COM component) or LicErrorString() (when using the DLL or the static library) all API functions return a 32-bit integer result code. In case of success the result code is zero. In case of failure the returned positive non-zero result code specifies the error encountered while executing the API function. ErrorString() and LicErrorString() can then be used to map the result code to a text string representation of the error.

All DLL functions adhere to the STDCALL calling convention.

The DLL and the static library do not support Unicode. Nor does the COM component. Although the COM component uses Unicode strings to interface with the outside world its internal string representation is ANSI strings. Full Unicode support is planned for future versions of the DLL and the COM component.

For the definitions of constants used in the API specifications, e.g. LIC_NO_ERROR or LIC_MODE_CONVENTIONAL see section 7.

1.4.4 Thread-safety

All API functions are thread-safe but some do not operate exclusively on thread-local storage. Some functions use global storage to maintain state between successive function calls in order to keep the API simple. When using COM global storage is allocated on a per-object basis. In case of the DLL or the static library the allocation is performed on a per-process basis automatically in DllMain() (DLL) or manually by calling __LicInitContext() (static library, see section 1.3.3). Although synchronization is in place to ensure thread-safety, i.e. that multi-threading does not corrupt global storage, any execution of an API function in any thread may overwrite information stored during any previous execution of an API function in any thread.

1.5 Source code

The Src subdirectory of this ZIP archive contains C source code for most parts of the libraries and the COM component. For the missing parts we have included object files. To build everything with the given makefile, the command line tools of Visual C++ 6.0 - cl.exe, link.exe, lib.exe, etc. - must be installed. Then simply run

```
nmake all
```

in this subdirectory to build the static library and the DLL/COM component.

2 Compatibility with earlier versions

2.1 Overview

Every now and then we will probably think of enhancements to our technology which are useful, but which would also make a new version of the libraries and the COM component incompatible with previous versions. To still enable software developers to always work with the latest version of the libraries/COM component and nevertheless be compatible with the initial release any compatibility-breaking enhancements will always be switched off by default. However, software developers can selectively enable these enhancements at their discretion. So, you will always have access to the latest version containing the latest bug fixes, but it is up to you to enable or disable those enhancements that break compatibility. In this way you will always be able to update existing installations of your software with the latest version of the libraries/COM component.

If you are not interested in enabling compatibility-breaking enhancements, just skip the remainder of this section. However, if you do not have an existing user-base that depends on compatibility, you should definitely learn how to enable all the latest bells and whistles in the libraries/COM component.

Compatibility is configured via the **SetCompatibility()** function. It takes a single integer argument of which bits 0 through 29 are used. Each of these 30 bits is linked to a single enhancement that breaks the compatibility between the current version of the libraries/COM component and their initial release. Setting one of these bits enables the corresponding enhancement and clearing one of these bits disables the corresponding enhancement. By default, i.e. if we do not call SetCompatibility() in our application, all compatibility-breaking enhancements are disabled.

The following table helps us map the bits that we want to set to the integer value to be passed to SetCompatibility(). We simply add the numbers given in the "Value" column for all bits that we want to set. The resulting number is an integer with the intended bits set and all remaining bits clear.

Bit	Value	Bit	Value	Bit	Value	Bit	Value
0	1	8	256	16	65,536	24	16,777,216
1	2	9	512	17	131,072	25	33,554,432
2	4	10	1,024	18	262,144	26	67,108,864
3	8	11	2,048	19	524,288	27	134,217,728
4	16	12	4,096	20	1,048,576	28	268,435,456
5	32	13	8,192	21	2,097,152	29	536,870,912
6	64	14	16,384	22	4,194,304		
7	128	15	32,768	23	8,388,608		

To set bits 0, 1, and 2 and clear all other bits, for example, we would have to pass 1 + 2 + 4 = 7 as integer parameter to SetCompatibility(). This would enable the enhancements linked to bits 0, 1, and 2. Passing 0 as the integer parameter would not enable any enhancements, i.e. it would disable all of them. This is the default.

2.2 SetCompatibility

Enables or disables compatibility-breaking enhancements in the libraries/COM component.

COM - SetCompatibility(Mask)

	Direction	**Type (C)**	**Type (Visual Basic)**
Mask	in	int	Long

	Description
Mask	Bits 0 through 29 of this integer parameter are used. Each of these 30 bits enables or disables one of the compatibility-breaking enhancements. Setting a bit enables the corresponding enhancement. Clearing a bit disables the corresponding enhancement. Have a look at the remarks below for additional information.

DLL - LicWinSetResourceInstance(ProductId, Inst)

	Direction	**Type (C)**
ProductId	in	const char *
Mask	in	int

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Mask	Bits 0 through 29 of this integer parameter are used. Each of these 30 bits enables or disables one of the compatibility-breaking enhancements. Setting a bit enables the corresponding enhancement. Clearing a bit disables the corresponding enhancement. Have a look at the remarks below for additional information.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

The following bits are currently used to enable or disable compatibility-breaking enhancements.

Bit	**Associated enhancement**
0	A new method for creating Hardware Hashes. This method has better statistical properties than the original method. However, for the same hardware configuration it generates a completely different Hardware Hash than the original method. This breaks compatibility with the initial release of the libraries/COM component.

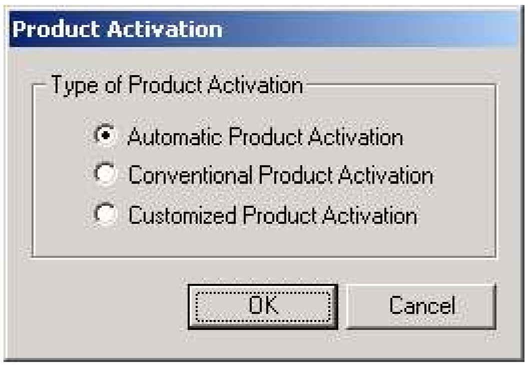
3 The standard API

Let's now have a look at the quick and easy way of integrating Product Activation into your application. We consider the functions that get you going first. Then we describe how the default GUI can be customized.

3.1 Basic functionality

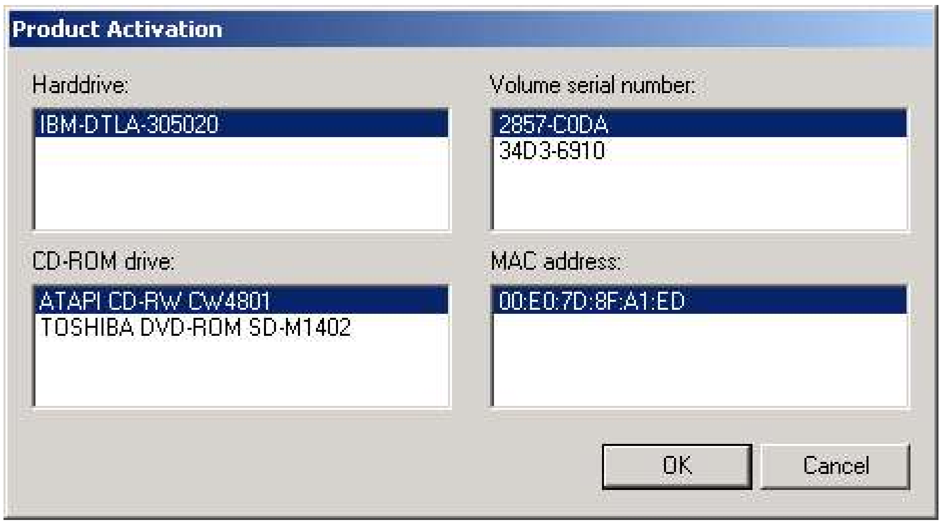
When using the standard API, activation comprises only three simple steps. Each step is executed by invoking a single API function.

- **Step 1**: A dialog box is displayed asking the end-user whether he or she prefers automatic, conventional, or customized Product Activation.


Product Activation
Type of Product Activation
Automatic Product Activation
Conventional Product Activation
Customized Product Activation
OK
Cancel

  The corresponding API function is **WinStep1()**. This step can be omitted if you prefer your software application to always perform a preselected type of Product Activation without asking the end-user.

- **Step 2**: The hardware configuration is determined. In case of customized Product Activation a dialog box is displayed allowing the end-user to select one of the detected harddrives, CD-ROM drives, volume serial numbers, and Ethernet adapters for inclusion in the hardware configuration.



Product Activation

Harddrive:
IBM-DTLA-305020

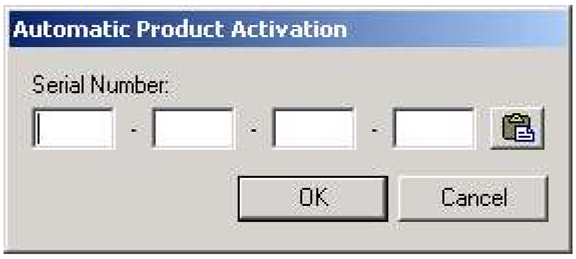
Volume serial number:
2857-C0DA
34D3-6910

CD-ROM drive:
ATAPI CD-RW CW4801
TOSHIBA DVD-ROM SD-M1402

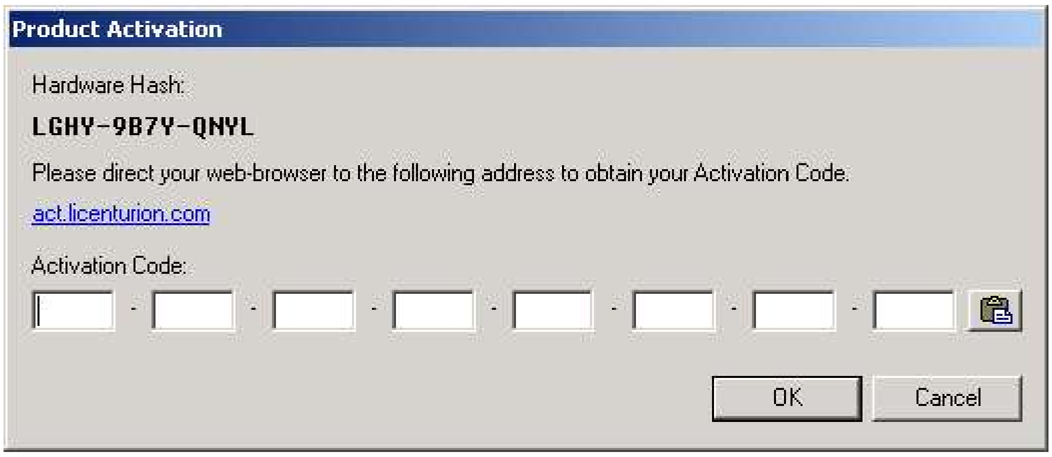
MAC address:
00:E0:7D:8F:A1:ED

  In the case of automatic or conventional Product Activation no interaction with the end-user takes place. The corresponding API function is **WinStep2()**.

- **Step 3**: In case of automatic Product Activation a dialog box appears which asks the end-user to type in his or her Serial Number. The dialog box only accepts valid characters, i.e. characters from the set of 26 letters and digits that are used to construct valid Serial Numbers. A paste button allows a Serial Number to be pasted from the clipboard.



In the case of conventional or customized Product Activation a dialog box is shown which contains the Hardware Hash representing the hardware configuration determined in the previous step besides the URL of the activation homepage and which asks the end-user to type in an Activation Code. The dialog box only accepts valid characters, i.e. characters from the set of 26 letters and digits that are used to construct valid Activation Codes. A paste button allows an Activation Code to be pasted from the clipboard.


Product Activation

Hardware Hash:

LGHY-9B7Y-QNYL

Please direct your web-browser to the following address to obtain your Activation Code.

act.licenturion.com

Activation Code:

The corresponding API function is **WinStep3()**. In the case of automatic Product Activation the Licenturion server is automatically sent the entered Serial Number and the Hardware Hash representing the current hardware configuration and the status value returned by this function indicates whether whether the Licenturion server has supplied a valid Activation Code.

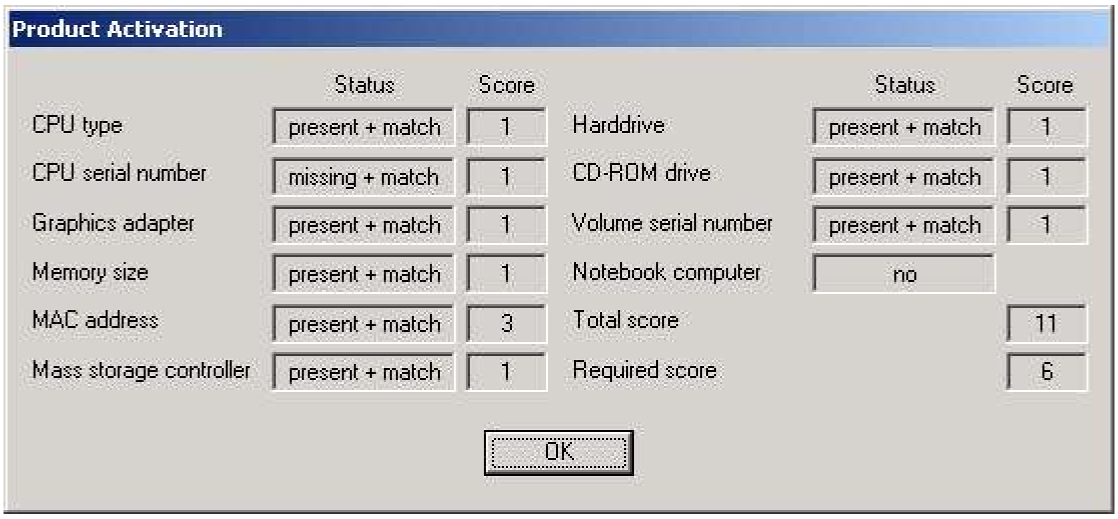
In the case of conventional or customized Product Activation the entered Activation Code is verified and the status value indicates whether the given Activation Code is valid.

If the status value indicates success, your software product is now activated. In case of an error you must repeat this step to enable the end-user to correct the entered Serial Number or Activation Code. The dialog box will then be opened again, containing the Serial Number or the Activation Code previously entered by the end user. In order to display an empty dialog box instead, invoke **WinResetDialog()** before repeating this step.

To determine whether to initiate the activation process check whether your software application has already been activated. In case it is, you simply skip activation or tell the end-user that his or her copy of your software application is already activated. The corresponding API function is **CheckActivation()**.

If successful, CheckActivation() and WinStep3() also return the 32-bit payload embedded in the Activation Code entered by the end-user or received from the Licenturion server. If you have instructed the Licenturion server to use the most significant 13 bits of the payload to carry an expiration date for the Activation Code, use **SplitPayload()** to retrieve the number of days left before the Activation Code expires and the remaining 19 (least significant) user-specified bits of the payload.

WinStatus() displays a dialog box detailing the activation status of your software application. For every characteristic the dialog box indicates whether it is present and matching, missing and matching, or whether it has been added, removed, or changed since activation time. The assigned individual score values are also given, as is the obtained total score and the configured threshold that the total score must reach.



Product Activation

	Status	Score		Status	Score
CPU type	present + match	1	Harddrive	present + match	1
CPU serial number	missing + match	1	CD-ROM drive	present + match	1
Graphics adapter	present + match	1	Volume serial number	present + match	1
Memory size	present + match	1	Notebook computer	no	
MAC address	present + match	3	Total score		11
Mass storage controller	present + match	1	Required score		6

The functions of the standard API store and retrieve Hardware Hashes and Activation Codes in and from the Windows registry. By default

HKEY_LOCAL_MACHINE\SOFTWARE\Licenturion GmbH\XXXXXXXX

is used. The eight "X" characters represent the Product ID of the product that the stored information belongs to. For activation the end-user requires privileges that enable him or her to create those registry keys ("Licenturion GmbH" and "XXXXXXXX") that do not yet exist and associate values with the "XXXXXXXX" key. For the NT line of operating systems (Windows NT, Windows 2000, and Windows XP) this means, that the end-user must be logged in as an administrator. As this is not necessarily the case, the alternative user-specific fallback location

HKEY_CURRENT_USER\SOFTWARE\Licenturion GmbH\XXXXXXXX

is tried if the end-user does not have the required privileges to write to the HKEY_LOCAL_MACHINE location. This leads to the following two scenarios on NT-based operating systems.

- The end-user has permission to write to the HKEY_LOCAL_MACHINE location. As the registration information stored at this location is visible to all user accounts, this activates the software application for all users.
- The end-user does not have permission to write to the HKEY_LOCAL_MACHINE location, i.e. he or she is not logged in as an administrator, and the activation information, i.e. the Hardware Hash and the Activation Code, is stored at the HKEY_CURRENT_USER location instead. As this location is user-specific, this only activates the software application for the single user that performs the activation.

To manually deactivate your software application, call **CleanRegistry()**. This API function removes any activation information related to your software application, i.e. the Hardware Hash representing the original hardware configuration and the Activation Code, from the Windows registry. It is typically used during the uninstallation of your application.

On NT-based operating systems The "Licenturion GmbH" and "XXXXXXXX" subkeys are created with "Full Control" permissions for "Everyone". CleanRegistry() will thus successfully deactivate the software application, even if invoked by an ordinary end-user without administrative privileges. Moreover, owing to the liberal "Full Control" permissions assigned to the "Licenturion GmbH" registry key, subsequent activations of the software application will succeed at the HKEY_LOCAL_MACHINE location even for unprivileged end-users.

The top-level directory of this ZIP archive contains subdirectories with example source code that illustrate the application of the standard API.

Subdirectory	**Contents**
Standard	Compact C source code that uses the DLL. The compiled executable can be found in the top-level directory of this ZIP archive (Standard.exe).
Standard-VB	Compact Visual Basic source code that uses the COM component.

The top-level directory also contains a Visual C++ workspace (Examples.dsw).

3.1.1 CheckActivation

Determines whether your software product has already been activated.

COM - CheckActivation(Payload)

	Direction	**Type (C)**	**Type (Visual Basic)**
Payload	out	int *	Long

	Description
Payload	If he function is successful, this parameter receives the 32-bit payload of the stored Activation Code.

DLL - LicCheckActivation(ProductId, Payload)

	Direction	**Type (C)**
ProductId	in	const char *
Payload	out	int *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Payload	Must point to an integer that, if the function succeeds, is set to the 32-bit payload of the stored Activation Code.

Result codes

- LIC_NO_ERROR
    - The function completed successfully, i.e. the software application is activated.
- LIC_ERROR_CANNOT_READ_FROM_REGISTRY
    - The registry key containing the Hardware Hash representing the original hardware configuration and the Activation Code could not be opened. The software application may not yet have been activated.
    - The Hardware Hash value or the Activation Code value could not be read from the opened registry key.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION
    - The current hardware configuration is too different from the original hardware configuration.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.
    - The function could not free memory on the process heap.
- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The Activation Code retrieved from the registry is invalid or does not match the retrieved Hardware Hash.

- LIC_ERROR_INVALID_HARDWARE_HASH
    - The Hardware Hash retrieved from the registry is invalid.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

The function tries to retrieve the Hardware Hash representing the original hardware configuration and the Activation Code for your software product from the registry. If it succeeds, if the Activation Code matches the original hardware configuration, and if the original hardware configuration is sufficiently similar to the current hardware configuration, your software product is considered activated, the Payload parameter is set to the payload of the stored Activation Code and LIC_NO_ERROR is returned.

3.1.2 WinStep1

Displays a dialog box that asks the end-user whether he or she prefers automatic Product Activation, conventional Product Activation, or customized Product Activation.

COM - WinStep1(Mode)

	Direction	**Type (C)**	**Type (Visual Basic)**
Mode	out	int *	Long

	Description
Mode	If the function is successful, this parameter is set to • LIC_MODE_AUTOMATIC, • LIC_MODE_CONVENTIONAL, or • LIC_MODE_CUSTOMIZED to reflect whether the end-user has selected automatic, conventional, or customized Product Activation.

DLL - LicWinStep1(ProductId, Mode)

	Direction	**Type (C)**
ProductId	in	const char *
Mode	out	int *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Mode	Must point to an integer that, if the function succeeds, is set to • LIC_MODE_AUTOMATIC, • LIC_MODE_CONVENTIONAL, or • LIC_MODE_CUSTOMIZED to reflect whether the end-user has selected automatic, conventional, or customized Product Activation.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CANCEL
    - The end-user pushed the cancel button.
- LIC_ERROR_CANNOT_CREATE_DIALOG
    - The dialog box could not be displayed. Make sure that the standard API tries to access the dialog box resources using the correct resource IDs.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

If you prefer not to offer your end-users the choice between automatic, conventional, and customized Product Activation, skip this function and always pass the LIC_MODE_AUTOMATIC constant, the LIC_MODE_CONVENTIONAL constant, or the LIC_MODE_CUSTOMIZED constant to WinStep2.

3.1.3 WinStep2

COM - WinStep2(HardwareHash, Mode)

Determines the current hardware configuration and returns a Hardware Hash representing the current hardware configuration. If customized Product Activation is specified, a dialog box is displayed that enables the end-user to specify which harddrive, CD-ROM drive, volume serial number, and Ethernet adapter to include in the hardware configuration.

	Direction	**Type (C)**	**Type (Visual Basic)**
HardwareHash	out	BSTR *	String
Mode	in	int	Long

	Description
HardwareHash	If the function succeeds, this string receives the Hardware Hash for the current hardware configuration.
Mode	Must be set to • LIC_MODE_AUTOMATIC, • LIC_MODE_CONVENTIONAL, or • LIC_MODE_CUSTOMIZED to indicate whether automatic, conventional, or customized Product Activation is desired. The value to be used is typically obtained from the WinStep1 function.

DLL - LicWinStep2(ProductId, HardwareHash, Mode)

	Direction	**Type (C)**
ProductId	in	const char *
HardwareHash	out	char *
Mode	in	int

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
HardwareHash	Must point to a 15-byte buffer that, if the function is successful, receives the Hardware Hash for the determined hardware configuration as a null-terminated ASCII string.
Mode	Must be set to • LIC_MODE_AUTOMATIC, • LIC_MODE_CONVENTIONAL, or • LIC_MODE_CUSTOMIZED to indicate whether automatic, conventional, or customized Product Activation is desired. The value to be used is typically obtained from the LicWinStep1 function.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CANCEL

    - The end-user pushed the cancel button.
- LIC_ERROR_CANNOT_CREATE_DIALOG
    - The dialog box could not be displayed. Make sure that the standard API tries to access the dialog box resources using the correct resource IDs.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - One or more of the dialog controls could not be accessed. Make sure that the standard API tries to access the dialog controls using the correct control IDs.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

This function is based on the CollectCandidates(), GetCandidate(), SelectCandidate(), and CreateHardwareHash() functions included in the advanced API.

3.1.4 WinStep3

Automatic Product Activation: Displays a dialog box that requests the end-user to enter his or her Serial Number, automatically transmits the Serial Number and the Hardware Hash representing the current hardware configuration to the Licenturion server, and obtains an Activation Code. If the Activation Code matches the current hardware configuration, the activation process is completed by storing the Activation Code and the Hardware Hash in the registry.

Conventional and customized Product Activation: Displays a dialog box containing the Hardware Hash to be used for activation that requests the end-user to enter the Activation Code obtained from the Licenturion server. If the Activation Code matches the current hardware configuration, the activation process is completed by storing the Activation Code and the Hardware Hash in the registry.

COM - WinStep3(Payload, HardwareHash, Mode)

	Direction	**Type (C)**	**Type (Visual Basic)**
Payload	out	int *	Long
HardwareHash	in	BSTR	String
Mode	in	int	Long

	Description
Payload	If he function is successful, this parameter receives the 32-bit payload of the Activation Code used for activation.
HardwareHash	Passes the Hardware Hash to be used for activation to the function. The specified Hardware Hash is typically the Hardware Hash returned by the WinStep2 function.
Mode	Must be set to • LIC_MODE_AUTOMATIC, • LIC_MODE_CONVENTIONAL, or • LIC_MODE_CUSTOMIZED to indicate whether automatic, conventional, or customized Product Activation is desired. The value to be used is typically obtained from the WinStep1 function.

DLL - LicWinStep3(ProductId, Payload, HardwareHash, Mode)

	Direction	**Type (C)**
ProductId	in	const char *
Payload	out	int *
HardwareHash	in	const char *
Mode	in	int

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Payload	Must point to an integer that, if he function is successful, receives the 32-bit payload of the Activation Code used for activation.

	Description
HardwareHash	Must point to a null-terminated ASCII string specifying the Hardware Hash to be used for activation. The specified Hardware Hash is typically the Hardware Hash returned by the LicWinStep2 function.
Mode	Must be set to • LIC_MODE_AUTOMATIC, • LIC_MODE_CONVENTIONAL, or • LIC_MODE_CUSTOMIZED to indicate whether automatic, conventional, or customized Product Activation is desired. The value to be used is typically obtained from the WinStep1 function.

Result codes

- LIC_NO_ERROR
    - The function completed successfully, i.e. the software application is now activated.
- LIC_ERROR_ACTIVATION_DENIED
    - The Licenturion server denied activation. The Serial Number entered by the end-user has probably reached its maximal number of first-time activations.
- LIC_ERROR_CANCEL
    - The end-user pushed the cancel button.
- LIC_ERROR_CANNOT_CREATE_DIALOG
    - The dialog box could not be displayed. Make sure that the standard API tries to access the dialog box resources using the correct resource IDs.
- LIC_ERROR_CANNOT_WRITE_TO_REGISTRY
    - The registry key to store the Hardware Hash representing the original hardware configuration and the Activation Code could not be created or opened.
    - The Hardware Hash value or the Activation Code value could not be set in the created or opened registry key.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION
    - The current hardware configuration is too different from the hardware configuration given by the specified Hardware Hash.
- LIC_ERROR_COMMUNICATION
    - The Licenturion server could not be contacted for automatic Product Activation.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - One or more of the dialog controls could not be accessed. Make sure that the standard API tries to access the dialog controls using the correct control IDs.
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.

  - The function could not free memory on the process heap.
  - The Licenturion server could not decrypt the information transmitted during automatic Product Activation.
  - The Licenturion server sent an unexpected response to our request.
- LIC_ERROR_INVALID_ACTIVATION_CODE
  - The Activation Code entered by the end-user or automatically obtained from the Licenturion server is invalid or does not match the specified Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
  - An invalid Hardware Hash was specified.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
  - An invalid Product ID was specified.
- LIC_ERROR_INVALID_SERIAL_NUMBER
  - The end-user entered an invalid Serial Number.
- LIC_ERROR_NOT_ENOUGH_MEMORY
  - The function could not allocate memory on the process heap.

Remarks

This function invokes the Activate() function or the AutoActivate() function included in the advanced API to perform the activation.

3.1.5 WinStatus

Displays a dialog box detailing the activation status of your software product.

COM - WinStatus()

No parameters.

DLL - LicWinStatus(ProductId)

	Direction	**Type (C)**
ProductId	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CANNOT_CREATE_DIALOG
    - The dialog box could not be displayed. Make sure that the standard API tries to access the dialog box resources using the correct resource IDs.
- LIC_ERROR_CANNOT_READ_FROM_REGISTRY
    - The registry key containing the Hardware Hash representing the original hardware configuration and the Activation Code could not be opened.
    - The Hardware Hash value or the Activation Code value could not be read from the opened registry key.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - One or more of the dialog controls could not be accessed. Make sure that the standard API tries to access the dialog controls using the correct control IDs.
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.
    - The function could not free memory on the process heap.
- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The Activation Code retrieved from the registry is invalid or does not match the retrieved Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
    - The Hardware Hash retrieved from the registry is invalid.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY

- The function could not allocate memory on the process heap.

Re marks

This function invokes the GetActivationStatus() function included in the advanced API to obtain the status information.

3.1.6 WinResetDialog

Resets the contents of the dialog boxes displayed by WinStep1 and WinStep3.

COM - WinResetDialog()

No parameters.

DLL - LicWinResetDialog(ProductId)

	Direction	**Type (C)**
ProductId	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

No remarks.

3.1.7 CleanRegistry

Deactivates the software product by removing the Hardware Hash representing the original hardware configuration and the Activation Code stored during activation from the Windows registry.

COM - CleanRegistry()

No parameters.

DLL - LicCleanRegistry(ProductId)

	Direction	**Type (C)**
ProductId	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CANNOT_DELETE_FROM_REGISTRY
    - The registry key that contains the Hardware Hash representing the original hardware configuration and the Activation Code could not be deleted.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

The Hardware Hash representing the original hardware configuration and the Activation Code are stored in the Windows registry during the activation process by the Activate() function or the AutoActivate() function included in the advanced API.

3.1.8 SplitPayload

This function is passed an Activation Code payload which contains an expiration date. The expiration date is extracted and compared to the current date. If the Activation Code has not yet expired, the function succeeds and returns the number of days remaining until the expiration date and the remaining 19 user-defined bits of the payload.

COM - SplitPayload(Left, PayloadOut, PayloadIn)

	Direction	**Type (C)**	**Type (Visual Basic)**
Left	out	int *	Long
PayloadOut	out	int *	Long
PayloadIn	in	int	Long

	Description
Left	If the function is successful, this parameter receives the number of days remaining until the expiration date.
PayloadOut	If the function is successful, this parameter receives the 19 (least significant) user-defined bits of the given payload.
PayloadIn	Passes the 32-bit payload of the Activation Code.

DLL - LicSplitPayload(ProductId, Left, PayloadOut, PayloadIn)

	Direction	**Type (C)**
ProductId	in	const char *
Left	out	int *
PayloadOut	out	int *
PayloadIn	in	int

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Left	Must point to an integer that, if the function is successful, is set to the number of days remaining until the expiration date.
PayloadOut	Must point to an integer that, if the function is successful, is set to the 19 (least significant) user-defined bits of the given payload.
PayloadIn	Must be set to the 32-bit payload of the Activation Code.

Result codes

- LIC_NO_ERROR
  - The function completed successfully, i.e. the given payload contains an expiration date that has not yet been reached.
- LIC_ERROR_EXPIRED
  - The expiration date in the given payload has been reached, i.e. the Activation Code has expired.
  - The end-user has tried to fool the expiration mechanism by turning back the system clock.

Remarks

No remarks.

3.1.9 ErrorString

COM - ErrorString(ResultCode)

Returns the text representation corresponding to the given result code.

	Direction	Type (C)	Type (Visual Basic)
ResultCode	in	int	Long

	Description
ResultCode	Must be set to the result code for which the text representation is to be obtained.

DLL - LicErrorString(ResultCode)

	Direction	Type (C)
ResultCode	in	int

	Description
ResultCode	Must be set to the result code for which the text representation is to be obtained.

Remarks

This function does not return a result code. It returns the text representation as a string.

For the COM component the corresponding type for the returned string is "BSTR" or "String" for C and Visual Basic, respectively. If there is not enough memory to create the string, an empty string is returned.

The DLL returns a pointer to the null-terminated ASCII representation of the string. The corresponding C type is "char *". No memory allocation is necessary in the DLL case, so the function never fails.

3.2 Customizing the user interface

The five dialog boxes used by the standard API are defined by five dialog box resources. For the COM object and the DLL licact.dll contains default dialog box resources. The static library does not contain any default dialog box resources and you have to supply them yourself by linking your executable with your own default dialog box resources. You might want to use the actres.rc file and its corresponding resource.h file in the Src subdirectory of this ZIP archive as a starting point.

The default dialog box resources may be overridden to customize the appearance of the user interface. Your custom dialog box resources can be specified in two ways.

- Tell the standard API to load a DLL containing your custom dialog box resources. That's what **WinLoadResourceDll()** does. To unload a loaded resource DLL use **WinFreeResourceDll()**.
- Tell the standard API to retrieve your custom dialog box resources from an already loaded module, e.g. an already loaded DLL or your executable. That's what **WinSetResourceInstance()** does.

If you override the default dialog box resources, no default dialog box resources are required. So, if you link against the static library and use one of the above two methods to specify your custom dialog box resources, you do not need to link against any custom dialog box resources.

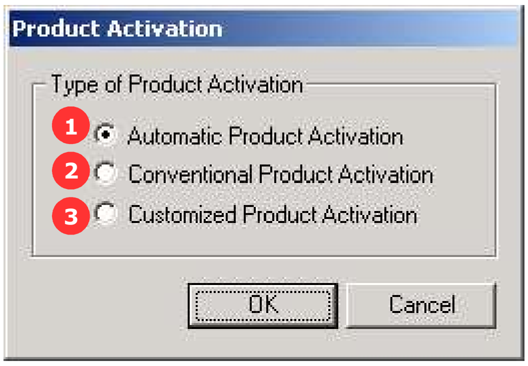
Each standard API function dealing with a dialog box accesses the corresponding dialog box resource using a resource ID as follows.

API function	**Resource ID**
WinStep1()	1972
WinStep2()	1973
WinStep3() - conventional or customized	1974
WinStep3() - automatic	1975
WinStatus()	1976

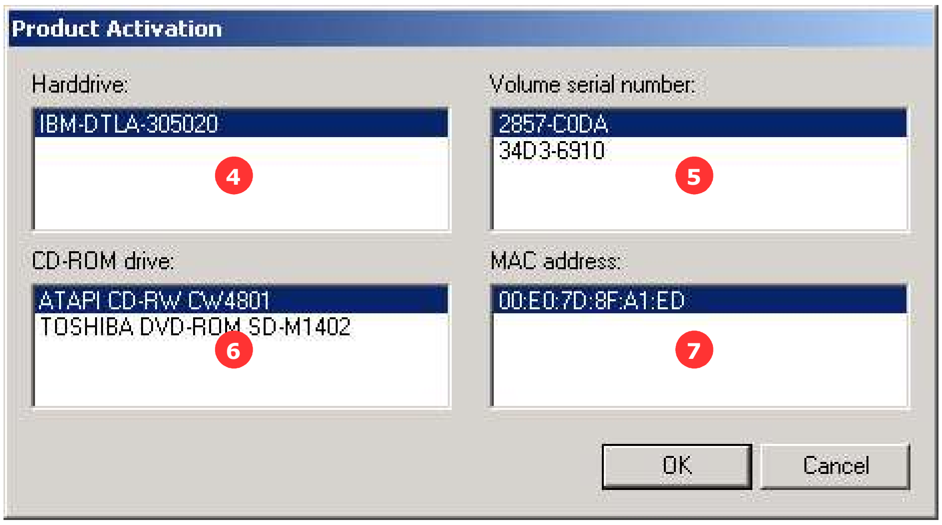
The main application icon is required to have a resource ID of 1977, the icon on the paste button of the two WinStep3() dialogs needs to have a resource ID of 1978. If you do not use an API function then you do not need to care about the corresponding resource IDs.

In the following five screen captures of the default dialog boxes the dialog controls accessed by the standard API functions have been marked with numbers.

- **WinStep1() dialog**


Product Activation
Type of Product Activation
1
Automatic Product Activation
2
Conventional Product Activation
3
Customized Product Activation
OK
Cancel

- **WinStep2() dialog**


Product Activation

Harddrive:

IBM-DTLA-305020

Volume serial number:

2857-C0DA
34D3-6910

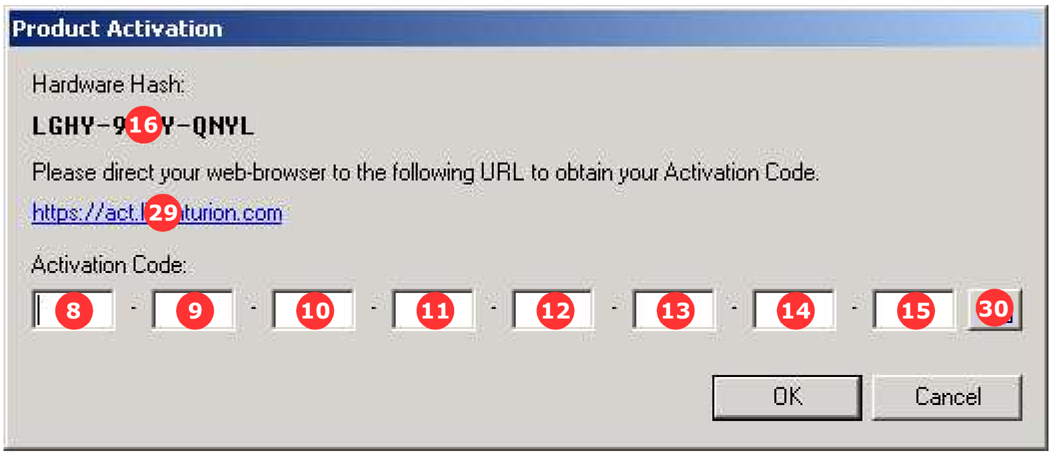
CD-ROM drive:

ATAPI CD-RW CW4801
TOSHIBA DVD-ROM SD-M1402

MAC address:

00:E0:7D:8F:A1:ED

- **WinStep3() dialogs**


Product Activation

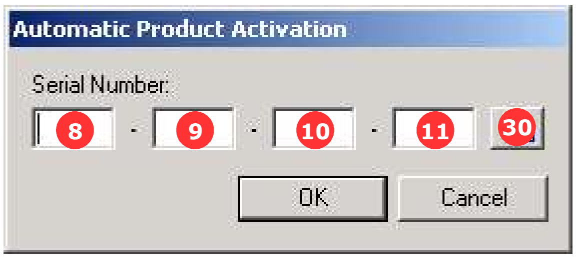
Hardware Hash:

LGHY-9 Y-QNYL

Please direct your web-browser to the following URL to obtain your Activation Code.

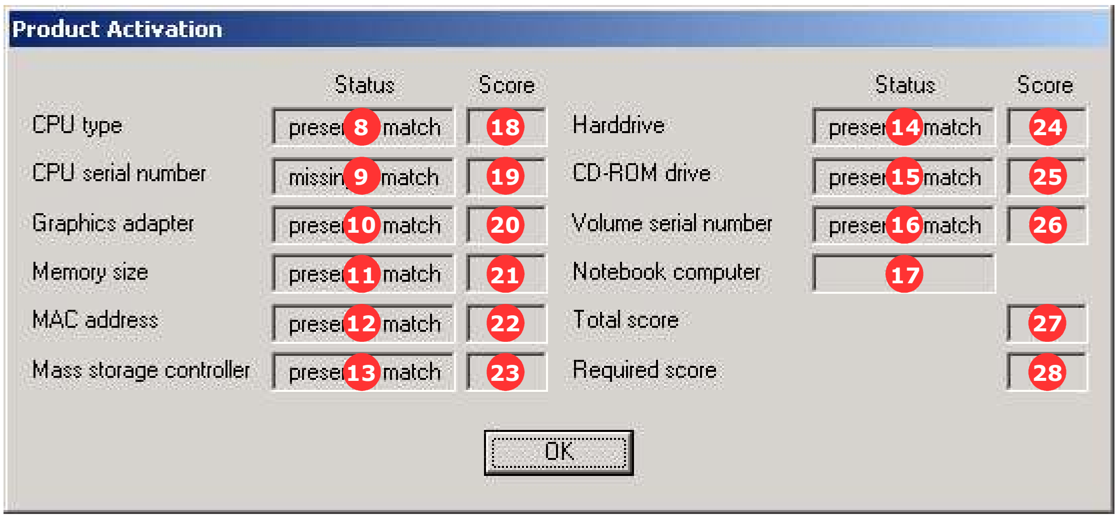
https://act. turion.com

Activation Code:


Automatic Product Activation

Serial Number:

- **WinStatus() dialog**



The following table lists the type and the control ID that the standard API expects each of the 30 controls to have.

#	Type	Control ID
1	radio button control	1001
2	radio button control	1002
3	radio button control	1003
4	list box control	1004
5	list box control	1005
6	list box control	1006
7	list box control	1007
8	edit control	1008
9	edit control	1009
10	edit control	1010
11	edit control	1011
12	edit control	1012
13	edit control	1013
14	edit control	1014
15	edit control	1015

#	Type	Control ID
16	edit control	1016
17	edit control	1017
18	edit control	1018
19	edit control	1019
20	edit control	1020
21	edit control	1021
22	edit control	1022
23	edit control	1023
24	edit control	1024
25	edit control	1025
26	edit control	1026
27	edit control	1027
28	edit control	1028
29	static text control	1029
30	button control	1030

Note that the paste button must have the BS_ICON style. Otherwise the standard API will be unable to assign the icon to the button. The static text control is subclassed by the standard API and modified to look like a hyperlink.

The OK button is always expected to be a button control and have the standard control ID of 1 (IDOK). Analogously, the cancel button is always expected to be a button control and to have the standard control ID of 2 (IDCANCEL). All remaining controls may have arbitrary types and control IDs as the standard API only touches the 30 marked controls as well as the OK button and the cancel button.

If you prefer to assign different resource IDs - e.g. if your software applications already uses one or more of the resource IDs 1972 through 1978 for different purposes - or different control IDs, you have to tell the standard API which IDs you use by calling the **WinMapIdValues()** function. This function can also be used to switch to a plain paste button without the BS_ICON style, to disable support for the paste button, or to disable support for the hyperlink.

Customizing the standard API may lead to errors, e.g. wrongly assigned control IDs, that can be a bit tricky to track down. The **WinEnableDebug()** function assists you in determining whether you have assigned the right control IDs to your dialog controls. Just call this function before invoking any other standard API function. Message boxes will then tell you what the data extracted from the dialog by the standard API functions, e.g. the Serial Number, looks like.

By default the desktop window is used as the parent window for all standard API dialog boxes. The **WinSetParent()** function selects an alternative parent window.

The default registry keys for storing the Activation Code and the Hardware Hash during the activation process can be overridden by calling **SetRegistryInfo()**.

IMPORTANT: CleanRegistry() does not only remove the created values from the registry, but also the key that contains the values. In the default case, for example, the "XXXXXXXX" subkey would be removed. Keep this in mind when overriding the default registry keys.

The only aspect that requires additional customization is the text assigned to each characteristic by WinStatus(). The default "added", "removed", "changed", "missing + match", and "present + match" may be overridden with the **WinMapStatusTexts()** function as well as the "yes" and "no" status that indicates whether the computer is recognized as a notebook computer.

The top-level directory of this ZIP archive contains subdirectories with example source code that illustrate the customization of the standard API.

Subdirectory	**Contents**
ResourceDll	Source code for a resource DLL that localizes the user interface of the standard API to German. Simply pass it to WinLoadResourceDll().
Standard	Compact C source code. Simply uncomment the customization functions.

The top-level directory also contains a Visual C++ workspace (Examples.dsw).

3.2.1 WinLoadResourceDll

Loads the resource DLL that contains the dialog box resources to be used by WinStep1(), WinStep2(), WinStep3(), and WinStatus().

COM - WinLoadResourceDll(DllPath)

	Direction	**Type (C)**	**Type (Visual Basic)**
DllPath	in	BSTR	String

	Description
DllPath	Passes the path of the DLL to be loaded to the function. For relative paths the DLL is located as specified for the LoadLibrary() function included in the Win32 API.

DLL - LicWinLoadResourceDll(ProductId, DllPath)

	Direction	**Type (C)**
ProductId	in	const char *
DllPath	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
DllPath	Must point to a null-terminated ASCII string specifying the path of the DLL to be loaded. For relative paths the DLL is located as specified for the LoadLibrary() function included in the Win32 API.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_LIBRARY_NOT_FOUND
    - The specified DLL file could not be found.
- LIC_ERROR_STRING_TOO_LONG (COM only)
    - The specified path was longer than 260 characters.

Remarks

WinFreeResourceDll() should be used to unload the resource DLL when it is not needed any longer.

WinSetResourceInstance() takes precedence over this function.

3.2.2 WinFreeResourceDll

Unloads a resource DLL previously loaded with WinLoadResourceDll().

COM - WinFreeResourceDll()

No parameters.

DLL - LicWinFreeResourceDll(ProductId)

	Direction	**Type (C)**
ProductId	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NO_LIBRARY_LOADED
    - No library was loaded.

Remarks

No remarks.

3.2.3 WinSetResourceInstance

Tells WinStep1(), WinStep2(), WinStep3(), and WinStatus() to retrieve the required dialog box resources from the given module.

COM - WinSetResourceInstance(Inst)

	Direction	**Type (C)**	**Type (Visual Basic)**
Inst	in	int	Long

	Description
Inst	Passes the instance handle of the module to the function. As instance handles are not supported by COM it has to be cast to an integer. Set this parameter to zero to clear a previously set instance.

DLL - LicWinSetResourceInstance(ProductId, Inst)

	Direction	**Type (C)**
ProductId	in	const char *
Inst	in	HINSTANCE

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Inst	Must be set to the instance handle of the module. Set this parameter to zero to clear a previously set instance.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

This function takes precedence over WinLoadResourceDll(). If you have used WinSetResourceInstance(), invoke it with an Inst parameter of zero before loading a resource DLL.

3.2.4 WinMapIdValues

Tells WinStep1(), WinStep2(), WinStep3(), and WinStatus() to use the given resource IDs and control IDs instead of the default to access dialog box resources and dialog controls.

COM - WinMapIdValues(Values)

	Direction	**Type (C)**	**Type (Visual Basic)**
Values	in	SAFEARRAY(int) *	Array of Long

	Description
Values	Passes an array with 37 values and an index range from 0 through 36 to the function. The first five elements (index 0 through 4) in this array specify the new resource IDs of the five dialog boxes. The sixth element (index 5) specifies the new resource ID of the main application icon. The seventh element (index 6) specifies the new resource ID of the paste button icon. The remaining 30 elements (index 7 through 36) specify the new control IDs to be used for accessing the dialog controls. All IDs must be ordered according to the tables given in section 3.2. For the default settings this array would thus contain (1972, 1973, 1974, 1975, 1976, 1977, 1978, 1001, 1002, 1003, ..., 1029, 1030).

DLL - LicWinMapIdValues(ProductId, Values, Length)

	Direction	**Type (C)**
ProductId	in	const char *
Values	in	int *
Length	in	int

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Values	Must point to an array of 37 integers. The first five elements (index 0 through 4) in this array specify the new resource IDs of the five dialog boxes. The sixth element (index 5) specifies the new resource ID of the main application icon. The seventh element (index 6) specifies the new resource ID of the paste button icon. The remaining 30 elements (index 7 through 36) specify the new control IDs to be used for accessing the dialog controls. All IDs must be ordered according to the tables given in section 3.2. For the default settings this array would thus contain (1972, 1973, 1974, 1975, 1976, 1977, 1978, 1001, 1002, 1003, ..., 1029, 1030).
Length	Must be set to the number of elements in the Values array, i.e. to 37.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - The SAFEARRAY could not be accessed (COM only).
- LIC_ERROR_INVALID_ARRAY_FORMAT

    - The specified array had more or less than 37 elements.
    - The first element of the specified array had an index different from 0. (COM only)
    - The elements of the specified array had an invalid type. (COM only)
    - The specified array was not one-dimensional. (COM only)
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

If the resource ID of the paste icon (index 6) is set to zero, the standard API does not try to assign any icon to the paste button. Set this array element to zero if you use a standard paste button with a text caption.

If the control ID of the paste button (index 36) is set to zero, the standard API skips the handling of messages related to the paste button. Set this array element to zero if you use WinStep3 dialog boxes without paste button.

If the control ID of the static text control (index 35) is set to zero, the standard API does not try to convert the static text control into a hyperlink. Set this array element to zero if you do not want your WinStep3 dialog box for conventional or customized Product Activation to contain the hyperlink to the activation web-site.

If the resource ID of the main application icon (index 5) is set to zero, the standard API does not try to set the application icon with WM_SETICON.

3.2.5 WinMapStatusTexts

Tells WinStatus() to use the given status strings instead of the default.

COM - WinMapStatusTexts(Texts)

	Direction	**Type (C)**	**Type (Visual Basic)**
Texts	in	SAFEARRAY(BSTR) *	Array of String

	Description
Texts	Passes an array with 7 strings and an index range from 0 through 6 to the function. The first 5 elements of the array (index 0 through 4) replace the "added", "removed", "changed", "missing + match", and "present + match" status strings. The remaining 2 elements (index 5 and 6) replace the "yes" and "no" status strings used to indicate whether the computer is considered a notebook computer. The strings must not be longer than 50 characters.

DLL - LicWinMapStatusTexts(ProductId, Texts)

	Direction	**Type (C)**
ProductId	in	const char *
Texts	in	const char const **

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Texts	Must point to an array of 7 pointers to null-terminated ASCII strings. The first 5 strings replace the "added", "removed", "changed", "missing + match", and "present + match" status strings. The remaining 2 strings replace the "yes" and "no" status strings used to indicate whether the computer is considered a notebook computer. The strings must not be longer than 50 characters excluding the terminating null character.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - The SAFEARRAY could not be accessed (COM only).
- LIC_ERROR_INVALID_ARRAY_FORMAT (COM only)
    - The specified array had more or less than 7 elements.
    - The first element of the specified array had an index different from 0.
    - The elements of the specified array had an invalid type.
    - The specified array was not one-dimensional.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

- LIC_ERROR_STRING_TOO_LONG
    - One of the specified strings was longer than 50 characters.

Remarks

No remarks.

3.2.6 SetRegistryInfo

Specifies the two registry keys to be used for storing the Hardware Hash and the Activation Code during the activation process.

IMPORTANT: CleanRegistry does not only remove the created values from the registry, but also the key that contains the values. In the default case, for example, the "XXXXXXXX" subkey would be removed. Keep this in mind when using SetRegistryInfo to override the default registry keys.

COM - SetRegistryInfo(Key1, Path1, Key2, Path2)

	Direction	**Type (C)**	**Type (Visual Basic)**
Key1	in	int	Long
Path1	in	BSTR	String
Key2	in	int	Long
Path2	in	BSTR	String

	Description
Key1	Passes the registry key to serve as the root for the Path1 parameter. Must be set to one of the following seven constants. • LIC_HKEY_CLASSES_ROOT • LIC_HKEY_CURRENT_USER • LIC_HKEY_LOCAL_MACHINE • LIC_HKEY_USERS • LIC_HKEY_PERFORMANCE_DATA • LIC_HKEY_CURRENT_CONFIG • LIC_HKEY_DYN_DATA
Path1	Passes the path of the registry key to be used for storing the activation information. This parameter is relative to the key given by the Key1 parameter. The string length must not exceed 500 characters.
Key2	The equivalent to Key1 for the fallback location.
Path2	The equivalent to Path1 for the fallback location.

DLL - LicSetRegistryInfo(ProductId, Key1, Path1, Key2, Path2)

	Direction	**Type (C)**
ProductId	in	const char *
Key1	in	int
Path1	in	const char *
Key2	in	int
Path2	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.

	Description
Key1	Specifies the registry key to serve as the root for the Path1 parameter. Must be set to one of the following seven constants. • LIC_HKEY_CLASSES_ROOT • LIC_HKEY_CURRENT_USER • LIC_HKEY_LOCAL_MACHINE • LIC_HKEY_USERS • LIC_HKEY_PERFORMANCE_DATA • LIC_HKEY_CURRENT_CONFIG • LIC_HKEY_DYN_DATA
Path1	Must point to a null-terminated ASCII string specifying the path of the registry key to be used for storing the activation information. This parameter is relative to the key given by the Key1 parameter. The string length must not exceed 500 characters excluding the terminating null character.
Key2	The equivalent to Key1 for the fallback location.
Path2	The equivalent to Path1 for the fallback location.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_STRING_TOO_LONG
    - One of the specified paths was longer than 500 characters.

Remarks

The location specified by Key1 and Path1 is tried first. If this location cannot be written to (store) or does not contain any activation information (retrieve) the location specified by Key2 and Path2 is tried. The idea is to have Key1 and Path1 specify a location that requires administrative privileges on NT-based operating systems. If the end-user is not logged in as an administrator, Key2 and Path2 provide the fallback location. By default the location tried first is

HKEY_LOCAL_MACHINE\SOFTWARE\Licenturion GmbH\XXXXXXXX

with

HKEY_CURRENT_USER\SOFTWARE\Licenturion GmbH\XXXXXXXX

as the fallback location. The eight "X" characters represent the Product ID of the product that the activation information belongs to.

The LIC_HKEY_* constants are derived from the Windows HKEY_* constants by clearing the most significant bit. The Windows constant 0x80000002 (HKEY_LOCAL_MACHINE), for example, thus becomes 2 (LIC_HKEY_LOCAL_MACHINE). We use the proprietary LIC_HKEY_* constants to have low constants that are also nicely representable in decimal notation.

To instruct, for example, the standard and advanced APIs to use

HKEY_CURRENT_USER\SOFTWARE\My Company\My Product\Activation

as the fallback location for storing the activation information, pass

- LIC_HKEY_LOCAL_MACHINE as Key2 and
- "SOFTWARE\My Company\My Product\Activation" as Path2

to this function.

IMPORTANT: Remember that in this example calling CleanRegistry will delete the "Activation" subkey!

3.2.7 WinEnableDebug

Enables debug message boxes that display the data retrieved from the dialog boxes.

COM - WinEnableDebug()

No parameters.

DLL - LicWinEnableDebug(ProductId)

	Direction	**Type (C)**
ProductId	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

No remarks.

3.2.8 WinSetParent()

Specifies the parent window for all dialog boxes displayed by the standard API.

COM - WinSetParent(Win)

	Direction	Type (C)	Type (Visual Basic)
Win	in	int	Long

	Description
Win	Passes the window handle of the desired parent window to the function. As window handles are not supported by COM it has to be cast to an integer.

DLL - LicWinSetParent(ProductId, Win)

	Direction	Type (C)
ProductId	in	const char *
Win	in	HWND

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Win	Must be set to the window handle of the desired parent window.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

No remarks.

4 The advanced API

The advanced API offers low-level functionality that enables you to create a HardwareHash representing the current hardware configuration and to perform an activation with this HardwareHash and a given Serial Number (automatic Product Activation) or a given Activation Code (conventional Product Activation or customized Product Activation). Moreover you may obtain detailed information about the current activation status of your software application.

Let's first have a look at how a Hardware Hash representing the present hardware configuration is created. As discussed before a Hardware Hash comprises nine characteristics. For each of the nine characteristics a set of candidates is determined and then for each characteristic one candidate of its candidate set is selected for inclusion in the Hardware Hash. Consider the example of the make and model of one of the installed harddrives. For this characteristic the set of candidates consists of the makes and models of all installed harddrives. From among these candidates one candidate, i.e. the make and model of one of the installed harddrives, is selected to be included in the Hardware Hash. In case of automatic or conventional Product Activation the selection is done without intervention of the end-user. In case of customized Product Activation the end-user is allowed to select the candidate to be included in the Hardware Hash for each characteristic. All this is achieved using five functions.

- **CollectCandidates()** determines the candidates for all nine characteristics and preselects a default candidate for each characteristic.
- **GetCandidate**() retrieves one of the candidates for a given characteristic and indicates whether the retrieved candidate is selected. Using this function you can enumerate all candidates for each characteristic and present them to the end-user to implement customized Product Activation. Each candidate is identified by an index value. Invoke this function with an index of 0 to obtain the first candidate, invoke this function with an index of 1 to obtain the second candidate, etc. Repeat this until the function returns an error code indicating that the specified index is invalid, which means that there are no more candidates left to be enumerated.
- **SelectCandidate()** selects a candidate for a given characteristic. Use this function to reflect your end-user's choice when implementing customized Product Activation. Candidates are identified analogously to GetCandidate(), i.e. by the same index values.
- **CreateHardwareHash()** creates a Hardware Hash based on the selected candidate from each group.
- **FreeCandidates()** releases the candidates collected by CollectCandidates() when they are no longer needed, i.e. after the Hardware Hash has been created.

For conventional Product Activation GetCandidate() and SelectCandidate() are not needed. Just stick to the candidates automatically preselected by CollectCandidates().

To implement conventional or customized Product Activation display the created Hardware Hash to your end-user and ask him or her for his or her Activation Code. Then pass both to the **Activate()** function to perform the activation. If the given Activation Code is valid for the given Hardware Hash, both are stored in the Windows registry.

To implement automatic Product Activation ask your end-user for his or her Serial Number and pass it along with the created Hardware Hash to **AutoActivate()**. The function contacts the Licenturion server, transmits the Serial Number and the Hardware Hash, obtains the Activation Code, and, if the obtained Activation Code is valid for the given Hardware Hash, stores the Activation Code along with the Hardware Hash in the Windows registry.

Just like the standard API the advanced API uses

HKEY_LOCAL_MACHINE\SOFTWARE\Licenturion GmbH\XXXXXXXX

and, if this fails, falls back to

HKEY_CURRENT_USER\SOFTWARE\Licenturion GmbH\XXXXXXXX

for storing and retrieving the Hardware Hash and the Activation Code for the software application.

The registry keys used for storing the activation information can be overridden, as was the case for the standard API before, with **SetRegistryInfo()**.

To obtain details of the activation state of your software application, call **GetActivationStatus**(), which returns things like the state of each characteristic, the corresponding individual score values, etc. It retrieves the activation information from the registry and compares the hardware configuration represented by the retrieved Hardware Hash with the current hardware configuration.

Manual deactivation is done by calling the **CleanRegistry()** function included in the standard API. To check whether your software application is already activated use the **CheckActivation()** function, which is also included in the standard API. Both functions have been discussed before.

If you prefer to handle the activation information yourself instead of having the advanced API store it for you in the Windows registry, use the **SimpleCheckActivation()** instead of Activate() and CheckActivation(), **SimpleAutoActivate()** instead of AutoActivate(), and **SimpleGetActivationStatus()** instead of GetActivationStatus(). These functions do not store and retrieve the activation information in and from the Windows registry. Instead the information is passed as input and output parameters to and from the functions.

Conventional and customized Product Activation would be implemented as follows.

- Generate a Hardware Hash as described above and display it to your end-user.
- Ask the end-user for his or her Activation Code.
- Pass the Activation Code and the Hardware Hash to SimpleCheckActivation() to verify the validity of the Activation Code.
- If the Activation Code is valid, store it along with the Hardware Hash.

Automatic Product Activation would require the following steps.

- Generate a Hardware Hash as described above.
- Ask the end-user for his or her Serial Number.
- Pass the Serial Number and the Hardware Hash to SimpleAutoActivate().
- If SimpleAutoActivate() succeeds, it returns a valid Activation Code. Store it along with the Hardware Hash.

To find out whether your software application is activated, retrieve the stored activation information and verify that it is valid by passing it to SimpleCheckActivation().

IMPORTANT: If we did not verify the validity of the retrieved activation information, a software pirate could - instead of activating - store arbitrary data at the location that our software application uses to store the activation information, e.g. in the Windows registry. So, do not rely on the activation information just "being there." The retrieved data could be anything. Make sure that it really **is valid activation information**.

IMPORTANT: Verifying the activation information only once during activation, then setting an "I am activated" flag, e.g. in the Windows registry, and just verifying whether this flag is set to determine whether our software application is activated is also a bad idea. A software pirate could just set this flag himself or herself and trick your software application into believing that it has already been activated. Always store and verify the full activation information. In contrast to a simple flag, it is **computationally infeasible to fake activation information**.

For a detailed view of the activation state of your application retrieve the activation information and pass it to SimpleGetActivationStatus().

The six functions that operate on Activation Codes - SimpleCheckActivation(), SimpleGetActivationStatus(), GetActivationStatus(), SimpleAutoActivate(), AutoActivate(), and Activate() - also return, if successful, the 32-bit payload embedded in the handled Activation Code. As has been described for the standard API, use **SplitPayload()** if you have instructed the Licenturion server to use the most significant 13 bits of the payload to carry an expiration date for the Activation Code.

The top-level directory of this ZIP archive contains subdirectories with example source code that illustrate the application of the advanced API.

Subdirectory	**Contents**
ActWiz	Rather large C++ source code that uses the static library. The compiled executable can be found in the top-level directory of this ZIP archive (ActWiz.exe).
Console	Compact C source code that uses the DLL. The compiled executable can be found in the top-level directory of this ZIP archive (Console.exe).
Status	Compact C source code that uses the DLL. The compiled executable can be found in the top-level directory of this ZIP archive (Status.exe).

The top-level directory also contains a Visual C++ workspace (Examples.dsw).

4.1 CollectCandidates

Create a collection of all candidates for all characteristics and preselect a default candidate for each characteristic.

COM - CollectCandidates(CandHand)

	Direction	**Type (C)**	**Type (Visual Basic)**
CandHand	out	int *	Long

	Description
CandHand	If the function succeeds, this parameter receives the handle identifying the created collection of candidates.

DLL - LicCollectCandidates(ProductId, CandHand)

	Direction	**Type (C)**
ProductId	in	const char *
CandHand	out	unsigned long *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
CandHand	Must point to an integer that, if the function succeeds, receives the handle identifying the created collection of candidates.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

The obtained handle is used to specify the created collection of candidates in subsequent calls to GetCandidate(), SelectCandidate(), and CreateHardwareHash(). If the collection of candidates is not required any longer, use FreeCandidates() to release it.

The handle actually is a pointer to a heap location. It is thus only valid in a given process context.

4.2 GetCandidate

Retrieves from the given collection of candidates the specified candidate of the specified characteristic.

COM - GetCandidate(String, Selected, CandHand, Char, Index)

	Direction	**Type (C)**	**Type (Visual Basic)**
String	out	BSTR *	String
Selected	out	int *	Long
CandHand	in	int	Long
Char	in	int	Long
Index	in	int	Long

	Description
String	If the function succeeds, this string receives the text representation of the specified candidate of the specified characteristic.
Selected	If the function succeeds, this parameter is set to 1 if the specified candidate is the currently selected candidate of the specified characteristic and otherwise set to 0.
CandHand	Passes the handle identifying the collection of candidates to be operated on.
Char	Passes the characteristic that the candidate to be retrieved is associated with. The values identifying the nine characteristics are as follows. • LIC_HARDDRIVE • LIC_CD_ROM_DRIVE • LIC_MASS_STORAGE_CONTROLLER • LIC_GRAPHICS_ADAPTER • LIC_CPU_TYPE • LIC_MEMORY_SIZE • LIC_VOLUME_SERIAL_NUMBER • LIC_CPU_SERIAL_NUMBER • LIC_MAC_ADDRESS
Index	Passes the index of the candidate to be retrieved for the specified characteristic (see below remarks).

DLL - LicGetCandidate(ProductId, String, Selected, CandHand, Char, Index)

	Direction	**Type (C)**
ProductId	in	const char *
String	out	char **
Selected	out	int *
CandHand	in	unsigned long
Char	in	int
Index	in	int

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.

	Description
String	Must point to a string pointer that, if the function succeeds, is set to point to a null-terminated ASCII string giving the text representation of the specified candidate of the specified characteristic.
Selected	Must point to an integer that, if the function succeeds, is set to 1 if the specified candidate is the currently selected candidate of the specified characteristic and otherwise set to 0.
CandHand	Must be set to the handle identifying the collection of candidates to be operated on.
Char	Must be set to the characteristic that the candidate to be retrieved is associated with. The following nine values identify the nine characteristics. • LIC_HARDDRIVE • LIC_CD_ROM_DRIVE • LIC_MASS_STORAGE_CONTROLLER • LIC_GRAPHICS_ADAPTER • LIC_CPU_TYPE • LIC_MEMORY_SIZE • LIC_VOLUME_SERIAL_NUMBER • LIC_CPU_SERIAL_NUMBER • LIC_MAC_ADDRESS
Index	Must be set to the index of the candidate to be retrieved for the specified characteristic (see below remarks).

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INVALID_INDEX
    - A candidate with the specified index does not exist for the specified characteristic.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY (COM only)
    - The function could not allocate memory on the process heap.

Remarks

The handle identifying the collection of candidates must be obtained from CollectCandidates().

Let us assume you would like to enumerate and print all candidates for the "make and model of one of the installed harddrives" characteristic. You would the proceed as follows.

1. Define an integer variable i and set i = 0.
2. Invoke CollectCandidates() to create a collection of candidates and obtain the associated handle.
3. Invoke GetCandidate() with the handle obtained in the previous step, the Char parameter set to LIC_HARDDRIVE, and the Index parameter set to i.
4. If the return value is LIC_ERROR_INVALID_INDEX then continue with step 8.

5. Print the text representation of the candidate as returned via the String parameter.
6. Set i = i + 1.
7. Go back to step 3.
8. Invoke FreeCandidates() with the handle obtained in step 2.
9. That's it.

4.3 SelectCandidate

Selects in the given collection of candidates the specified candidate of the specified characteristic for inclusion in the Hardware Hash.

COM - SelectCandidate(CandHand, Char, Index)

	Direction	**Type (C)**	**Type (Visual Basic)**
CandHand	in	int	Long
Char	in	int	Long
Index	in	int	Long

	Description
CandHand	Passes the handle identifying the collection of candidates to be operated on.
Char	Passes the characteristic for which the candidate is to be selected. The values identifying the nine characteristics are as follows. • LIC_HARDDRIVE • LIC_CD_ROM_DRIVE • LIC_MASS_STORAGE_CONTROLLER • LIC_GRAPHICS_ADAPTER • LIC_CPU_TYPE • LIC_MEMORY_SIZE • LIC_VOLUME_SERIAL_NUMBER • LIC_CPU_SERIAL_NUMBER • LIC_MAC_ADDRESS
Index	Passes the index of the candidate to be selected for the specified characteristic.

DLL - LicSelectCandidate(ProductId, CandHand, Char, Index)

	Direction	**Type (C)**
ProductId	in	const char *
CandHand	in	unsigned long
Char	in	int
Index	in	int

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
CandHand	Must be set to the handle identifying the collection of candidates to be operated on.

	Description
Char	Must be set to the characteristic for which the candidate is to be selected. The values identifying the nine characteristics are as follows. • LIC_HARDDRIVE • LIC_CD_ROM_DRIVE • LIC_MASS_STORAGE_CONTROLLER • LIC_GRAPHICS_ADAPTER • LIC_CPU_TYPE • LIC_MEMORY_SIZE • LIC_VOLUME_SERIAL_NUMBER • LIC_CPU_SERIAL_NUMBER • LIC_MAC_ADDRESS
Index	Must be set to the index of the candidate to be selected for the specified characteristic.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INVALID_INDEX
    - A candidate with the specified index does not exist for the specified characteristic.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

The handle identifying the collection of candidates must be obtained from CollectCandidates().

GetCandidate() and SelectCandidate() use the same index values to identify candidates for a given characteristic.

4.4 CreateHardwareHash

Creates a Hardware Hash based on the nine candidates selected for the nine characteristics.

COM - CreateHardwareHash(HardwareHash, CandHand)

	Direction	**Type (C)**	**Type (Visual Basic)**
HardwareHash	out	BSTR *	String
CandHand	in	int	Long

	Description
HardwareHash	If the function succeeds, this string receives the generated Hardware Hash.
CandHand	Passes the handle identifying the collection of candidates to be operated on.

DLL - LicCreateHardwareHash(ProductId, HardwareHash, CandHand)

	Direction	**Type (C)**
ProductId	in	const char *
HardwareHash	out	char *
CandHand	in	unsigned long

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
HardwareHash	Must point to a 15-byte buffer that, if the function is successful, receives the generated Hardware Hash as a null-terminated ASCII string.
CandHand	Must be set to the handle identifying the collection of candidates to be operated on.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY (COM only)
    - The function could not allocate memory on the process heap.

Remarks

No remarks.

4.5 FreeCandidates

Releases the specified collection of candidates.

COM - FreeCandidates(CandHand)

	Direction	**Type (C)**	**Type (Visual Basic)**
CandHand	in	int	Long

	Description
CandHand	Passes the handle identifying the collection of candidates to be released.

DLL - LicFreeCandidates(ProductId, CandHand)

	Direction	**Type (C)**
ProductId	in	const char *
CandHand	in	unsigned long

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
CandHand	Must be set to the handle identifying the collection of candidates to be released.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_INTERNAL
    - The function could not free heap memory. Perhaps you passed an invalid handle or tried to release the same collection of candidates more than once.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

No remarks.

4.6 Activate

Activates your software application using the specified Hardware Hash and Activation Code and, if the Activation Code matches the Hardware Hash, returns the payload contained in the Activation Code.

COM - Activate(Payload, HardwareHash, ActivationCode)

	Direction	**Type (C)**	**Type (Visual Basic)**
Payload	out	int *	Long
HardwareHash	in	BSTR	String
ActivationCode	in	BSTR	String

	Description
Payload	If the function succeeds, this parameter receives the payload carried by the supplied ActivationCode parameter.
HardwareHash	Passes the Hardware Hash.
ActivationCode	Passes the Activation Code.

DLL - LicActivate(ProductId, Payload, HardwareHash, ActivationCode)

	Direction	**Type (C)**
ProductId	in	const char *
Payload	out	int *
HardwareHash	in	const char *
ActivationCode	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Payload	Must point to an integer that, if the the function is successful, receives the payload carried by the supplied ActivationCode parameter.
HardwareHash	Must point to a null-terminated ASCII string specifying the Hardware Hash.
ActivationCode	Must point to a null-terminated ASCII string specifying the Activation Code.

Result codes

- LIC_NO_ERROR
    - The function completed successfully, i.e. the software application is now activated.
- LIC_ERROR_CANNOT_WRITE_TO_REGISTRY
    - The registry key to store the Hardware Hash representing the original hardware configuration and the Activation Code could not be created or opened.
    - The Hardware Hash value or the Activation Code value could not be set in the created or opened registry key.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION

    - The current hardware configuration is too different from the hardware configuration given by the specified Hardware Hash.

- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.
    - The function could not free memory on the process heap.
- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The passed Activation Code is invalid or does not match the specified Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
    - An invalid Hardware Hash was specified.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

If the Activation Code matches the Hardware Hash, both strings are stored in the Windows registry, turning the software product into activated state.

4.7 AutoActivate

Activates your software application using the specified Hardware Hash and Serial Number by contacting the Licenturion server and obtaining an Activation Code. If the Activation Code matches the Hardware Hash and the original hardware configuration as represented by the Hardware Hash sufficiently resembles the current hardware configuration, the function succeeds and returns the payload contained in the Activation Code.

COM - AutoActivate(Payload, SerialNumber, HardwareHash)

	Direction	**Type (C)**	**Type (Visual Basic)**
Payload	out	int *	Long
SerialNumber	in	BSTR	String
HardwareHash	in	BSTR	String

	Description
Payload	If the function succeeds, this parameter receives the payload carried by the supplied ActivationCode parameter.
SerialNumber	Passes the Serial Number.
HardwareHash	Passes the Hardware Hash.

DLL - LicAutoActivate(ProductId, Payload, SerialNumber, HardwareHash)

	Direction	**Type (C)**
ProductId	in	const char *
Payload	out	int *
SerialNumber	in	const char *
HardwareHash	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Payload	Must point to an integer that, if the the function is successful, receives the payload carried by the supplied ActivationCode parameter.
SerialNumber	Must point to a null-terminated ASCII string specifying the Serial Number.
HardwareHash	Must point to a null-terminated ASCII string specifying the Hardware Hash.

Result codes

- LIC_NO_ERROR
    - The function completed successfully, i.e. the software application is now activated.
- LIC_ERROR_ACTIVATION_DENIED
    - The Licenturion server denied activation. The passed Serial Number has probably reached its maximal number of first-time activations.
- LIC_ERROR_CANNOT_WRITE_TO_REGISTRY
    - The registry key to store the Hardware Hash representing the original hardware configuration and the Activation Code could not be created or opened.

    - The Hardware Hash value or the Activation Code value could not be set in the created or opened registry key.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION
    - The current hardware configuration is too different from the hardware configuration given by the specified Hardware Hash.
- LIC_ERROR_COMMUNICATION
    - The Licenturion server could not be contacted.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.
    - The function could not free memory on the process heap.
    - The Licenturion server could not decrypt our request.
    - The Licenturion server sent an unexpected response to our request.
- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The Activation Code obtained from the Licenturion server is invalid or does not match the specified Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
    - An invalid Hardware Hash was specified.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_INVALID_SERIAL_NUMBER
    - An invalid Serial Number was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

The obtained Activation Code and the Hardware Hash are stored in the Windows registry, turning the software product into activated state.

The necessary communication with the Licenturion server consists of a single HTTP POST request to port 80 of the server and the corresponding HTTP response. The request contains a Server Cookie (see section 1.1.4), which is an encrypted representation of the given Serial Number and the given Hardware Hash. The response is a plain ASCII file that contains the Activation Code. We have chosen to implement our own encryption scheme on top of HTTP as HTTPS traffic is typically restricted by corporate firewalls. Encryption is necessary as Serial Numbers have to be protected against eavesdroppers while they traverse the Internet. Otherwise a software pirate could just break into a computer topologically close to the Licenturion server and harvest lots of valid Serial Numbers.

The link to the Licenturion server is established using the WinINet API. The settings of Internet Explorer, e.g. proxy settings, are thus respected.

4.8 GetActivationStatus

Provides a detailed view of the activation state of your software application. The function retrieves the Activation Code and the Hardware Hash representing the original hardware configuration from the Windows registry, validates the Activation Code, and compares the original hardware configuration to the current hardware configuration.

COM - GetActivationStatus(Stat, Scores, Total, Threshold, Notebook)

	Direction	**Type (C)**	**Type (Visual Basic)**
Stat	out	SAFEARRAY (unsigned char) *	Array of Byte
Scores	out	SAFEARRAY(int) *	Array of Long
Total	out	int *	Long
Threshold	out	int *	Long
Notebook	out	int *	Long

	Description
Stat	If the function succeeds, this array receives the states of the nine characteristics. Each array element is set to one of the following five values to indicate one of the five possible states. • LIC_STATUS_ADDED • LIC_STATUS_REMOVED • LIC_STATUS_CHANGED • LIC_STATUS_MISSING_AND_MATCHING • LIC_STATUS_PRESENT_AND_MATCHING The nine elements of this array map to the characteristics as follows. • index 0: make and model of one of the installed harddrives • index 1: make and model of one of the installed CD-ROM drives • index 2: make and model of one of the installed SCSI host adapters or IDE controllers • index 3: make and model of one of the installed graphics boards • index 4: make and model of the first CPU in the computer • index 5: size of the installed RAM • index 6: volume serial number of one of the available disk volumes • index 7: CPU serial number of the first CPU in the computer • index 8: Ethernet address of one of the installed Ethernet adapters
Scores	If the function succeeds, this array receives the individual score values assigned to the nine characteristics. The nine elements have the index values 0 to 8 and map to the characteristics in the same way as specified for the elements of the above Stat array.
Total	If the function succeeds, this parameter receives the total score value calculated by the scoring mechanism by adding all individual score values.
Threshold	If the function succeeds, this parameter receives the total score value required to consider this software product activated.
Notebook	If the function succeeds, the parameter is set to 1 if the computer is considered to be a notebook computer and otherwise set to 0.

DLL - LicGetActivationStatus(ProductId, Stat, Scores, Total, Threshold, Notebook)

	Direction	Type (C)
ProductId	in	const char *
Stat	out	unsigned char *
Scores	out	int *
Total	out	int *
Threshold	out	int *
Notebook	out	int *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Stat	Must point to a buffer of nine bytes that, if the function succeeds, receives the states of the nine characteristics. Each byte in the buffer is set to one of the following five values to indicate one of the five possible states. • LIC_STATUS_ADDED • LIC_STATUS_REMOVED • LIC_STATUS_CHANGED • LIC_STATUS_MISSING_AND_MATCHING • LIC_STATUS_PRESENT_AND_MATCHING The nine bytes in this buffer map to the characteristics as follows. • index 0: make and model of one of the installed harddrives • index 1: make and model of one of the installed CD-ROM drives • index 2: make and model of one of the installed SCSI host adapters or IDE controllers • index 3: make and model of one of the installed graphics boards • index 4: make and model of the first CPU in the computer • index 5: size of the installed RAM • index 6: volume serial number of one of the available disk volumes • index 7: CPU serial number of the first CPU in the computer • index 8: Ethernet address of one of the installed Ethernet adapters
Scores	Must point to a buffer with space for nine integers that, if the function succeeds, receives the individual score values assigned to the nine characteristics. The nine integers map to the characteristics in the same order as specified for the elements of the above Stat buffer.
Total	Must point to an integer that, if the function succeeds, is set to the total score value calculated by the scoring mechanism by adding all individual score values.
Threshold	Must point to an integer that, if the function succeeds, is set to the total score value required to consider this software product activated.
Notebook	Must point to an integer that, if the function succeeds, is set to 1 if the computer is considered to be a notebook computer and otherwise set to 0.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CANNOT_READ_FROM_REGISTRY

    - The registry key containing the Hardware Hash representing the original hardware configuration and the Activation Code could not be opened.
    - The Hardware Hash value or the Activation Code value could not be read from the opened registry key.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION
    - The current hardware configuration is too different from the original hardware configuration given by the retrieved Hardware Hash.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.
    - The function could not free memory on the process heap.
    - One of the SAFEARRAYs could not be accessed (COM only).
- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The Activation Code retrieved from the registry is invalid or does not match the retrieved Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
    - The Hardware Hash retrieved from the registry is invalid.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

The function is successful if it either returns either LIC_NO_ERROR or LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION. In the former case, your software application is considered activated. In the latter case substantial hardware changes have been applied to the computer since activation that invalidate the activation. Your software application is not activated but the Stat, Scores, Total, Threshold, and Notebook parameters have still been set to the correct values.

4.9 SimpleCheckActivation

This function behaves like CheckActivation. The only difference is that it does not try to retrieve the Activation Code and the Hardware Hash from the Windows Registry. Both are passed as parameters instead.

COM - SimpleCheckActivation(Payload, HardwareHash, ActivationCode)

	Direction	**Type (C)**	**Type (Visual Basic)**
Payload	out	int *	Long
HardwareHash	in	BSTR	String
ActivationCode	in	BSTR	String

	Description
Payload	If he function is successful, this parameter receives the 32-bit payload of the given Activation Code.
HardwareHash	Passes the Hardware Hash that represents the original hardware configuration.
ActivationCode	Passes the Activation Code.

DLL - LicSimpleCheckActivation(ProductId, Payload, HardwareHash, ActivationCode)

	Direction	**Type (C)**
ProductId	in	const char *
Payload	out	int *
HardwareHash	in	const char *
ActivationCode	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Payload	Must point to an integer that, if the function succeeds, is set to the 32-bit payload of the given Activation Code.
HardwareHash	Must point to a null-terminated ASCII string specifying the Hardware Hash representing the original hardware configuration.
ActivationCode	Must point to a null-terminated ASCII string specifying the Activation Code.

Result codes

- LIC_NO_ERROR
    - The function completed successfully, i.e. the given Activation Code and the given Hardware Hash constitute valid activation information for the software application.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION
    - The current hardware configuration is too different from the original hardware configuration as specified by the Hardware Hash.
- LIC_ERROR_INTERNAL
    - Somebody has illicitly modified the individual score values contained in the

licact.dll file.

    - The function could not free memory on the process heap.

- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The given Activation Code retrieved from the registry is invalid or does not match the given Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
    - The given Hardware Hash is invalid.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

If the Activation Code matches the original hardware configuration as given by the passed Hardware Hash, and if the original hardware configuration is sufficiently similar to the current hardware configuration, your software product is considered activated, the Payload parameter is set to the payload of the passed Activation Code and LIC_NO_ERROR is returned.

Assume that we would like to implement a variant of Product Activation where the Hardware Hash representing the original hardware configuration and the Activation Code are stored in a file during the activation process. To perform conventional or customized Product Activation in this scenario, we would proceed as follows.

- Create a Hardware Hash based on the current hardware configuration. (CollectCandidates, GetCandidate, SelectCandidate, CreateHardwareHash, FreeCandidates)
- Display the Hardware Hash to the end-user and ask him or her for the matching Activation Code.
- Invoke this function with the created Hardware Hash and the entered Activation Code.
- If the result is not LIC_NO_ERROR, tell the end-user that the activation did not work out.
- Otherwise store the entered Activation Code along with the generated Hardware Hash in the file.

To check whether our software application has already been activated, we would go through the following steps.

- Retrieve the Activation Code and the Hardware Hash from the file.
- Pass both to this function.
- If the result is LIC_NO_ERROR, the software application is already activated.
- Otherwise it is not activated, yet.

4.10 SimpleAutoActivate

This function behaves like AutoActivate. The only difference is that it does not store the given Hardware Hash and the Activation Code obtained from the Licenturion server in the Windows registry. Instead, the Activation Code is returned to the caller.

COM - SimpleAutoActivate(ActivationCode, Payload, SerialNumber, HardwareHash)

	Direction	**Type (C)**	**Type (Visual Basic)**
ActivationCode	out	BSTR *	String
Payload	out	int *	Long
SerialNumber	in	BSTR	String
HardwareHash	in	BSTR	String

	Description
ActivationCode	If the function succeeds, this strings receives the Activation Code returned by the Licenturion server.
Payload	If the function succeeds, this parameter receives the payload carried by the Activation Code returned by the Licenturion server.
SerialNumber	Passes the Serial Number.
HardwareHash	Passes the Hardware Hash representing the original hardware configuration to be used.

DLL - LicSimpleAutoActivate(ProductId, ActivationCode, Payload, SerialNumber, HardwareHash)

	Direction	**Type (C)**
ProductId	in	const char *
ActivationCode	out	char *
Payload	out	int *
SerialNumber	in	const char *
HardwareHash	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
ActivationCode	Must point to a 40-byte buffer that, if the function is successful, receives the Activation Code returned by the Licenturion server.
Payload	Must point to an integer that, if the the function is successful, receives the payload carried by the Activation Code returned by the Licenturion server.
SerialNumber	Must point to a null-terminated ASCII string specifying the Serial Number.
HardwareHash	Must point to a null-terminated ASCII string specifying the Hardware Hash representing the original hardware configuration to be used.

Result codes

- LIC_NO_ERROR
    - The function completed successfully, i.e. the ActivationCode parameter contains a valid Activation Code for the software application and the Payload parameter has been set to the corresponding payload.

- LIC_ERROR_ACTIVATION_DENIED
    - The Licenturion server denied activation. The passed Serial Number has probably reached its maximal number of first-time activations.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION
    - The current hardware configuration is too different from the hardware configuration given by the specified Hardware Hash.
- LIC_ERROR_COMMUNICATION
    - The Licenturion server could not be contacted.
- LIC_ERROR_INTERNAL
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.
    - The function could not free memory on the process heap.
    - The Licenturion server could not decrypt our request.
    - The Licenturion server sent an unexpected response to our request.
- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The Activation Code obtained from the Licenturion server is invalid or does not match the specified Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
    - An invalid Hardware Hash was specified.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_INVALID_SERIAL_NUMBER
    - An invalid Serial Number was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

Assume that we would like to implement a variant of Product Activation where the Hardware Hash representing the original hardware configuration and the Activation Code are stored in a file during the activation process. To perform automatic Product Activation in this scenario, we would proceed as follows.

- Create a Hardware Hash based on the current hardware configuration. (CollectCandidates, GetCandidate, SelectCandidate, CreateHardwareHash, FreeCandidates)
- Ask the end-user for his or her Serial Number.
- Invoke this function with the created Hardware Hash and the entered Serial Number.
- If the result is not LIC_NO_ERROR, tell the end-user that the activation did not work out.
- Otherwise store the obtained Activation Code along with the generated Hardware Hash in the file.

4.11 SimpleGetActivationStatus

This function behaves like GetActivationStatus. The only difference is that it does not try to retrieve the Activation Code and the Hardware Hash from the Windows registry. Both are passed as parameters instead.

COM - SimpleGetActivationStatus(Stat, Scores, Total, Threshold, Notebook, HardwareHash, ActivationCode)

	Direction	Type (C)	Type (Visual Basic)
Stat	out	SAFEARRAY (unsigned char) *	Array of Byte
Scores	out	SAFEARRAY(int) *	Array of Long
Total	out	int *	Long
Threshold	out	int *	Long
Notebook	out	int *	Long
HardwareHash	in	BSTR	String
ActivationCode	in	BSTR	String

	Description
Stat	If the function succeeds, this array receives the states of the nine characteristics. Each array element is set to one of the following five values to indicate one of the five possible states. • LIC_STATUS_ADDED • LIC_STATUS_REMOVED • LIC_STATUS_CHANGED • LIC_STATUS_MISSING_AND_MATCHING • LIC_STATUS_PRESENT_AND_MATCHING The nine elements of this array map to the characteristics as follows. • index 0: make and model of one of the installed harddrives • index 1: make and model of one of the installed CD-ROM drives • index 2: make and model of one of the installed SCSI host adapters or IDE controllers • index 3: make and model of one of the installed graphics boards • index 4: make and model of the first CPU in the computer • index 5: size of the installed RAM • index 6: volume serial number of one of the available disk volumes • index 7: CPU serial number of the first CPU in the computer • index 8: Ethernet address of one of the installed Ethernet adapters
Scores	If the function succeeds, this array receives the individual score values assigned to the nine characteristics. The nine elements have the index values 0 to 8 and map to the characteristics in the same way as specified for the elements of the above Stat array.
Total	If the function succeeds, this parameter receives the total score value calculated by the scoring mechanism by adding all individual score values.
Threshold	If the function succeeds, this parameter receives the total score value required to consider this software product activated.
Notebook	If the function succeeds, the parameter is set to 1 if the computer is considered to be a notebook computer and otherwise set to 0.

	Description
HardwareHash	Passes the Hardware Hash that represents the original hardware configuration.
ActivationCode	Passes the Activation Code matching the given Hardware Hash.

DLL - LicGetActivationStatus(ProductId, Stat, Scores, Total, Threshold, Notebook, HardwareHash, ActivationCode)

	Direction	Type (C)
ProductId	in	const char *
Stat	out	unsigned char *
Scores	out	int *
Total	out	int *
Threshold	out	int *
Notebook	out	int *
HardwareHash	in	const char *
ActivationCode	in	const char *

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Stat	Must point to a buffer of nine bytes that, if the function succeeds, receives the states of the nine characteristics. Each byte in the buffer is set to one of the following five values to indicate one of the five possible states. • LIC_STATUS_ADDED • LIC_STATUS_REMOVED • LIC_STATUS_CHANGED • LIC_STATUS_MISSING_AND_MATCHING • LIC_STATUS_PRESENT_AND_MATCHING The nine bytes in this buffer map to the characteristics as follows. • index 0: make and model of one of the installed harddrives • index 1: make and model of one of the installed CD-ROM drives • index 2: make and model of one of the installed SCSI host adapters or IDE controllers • index 3: make and model of one of the installed graphics boards • index 4: make and model of the first CPU in the computer • index 5: size of the installed RAM • index 6: volume serial number of one of the available disk volumes • index 7: CPU serial number of the first CPU in the computer • index 8: Ethernet address of one of the installed Ethernet adapters
Scores	Must point to a buffer with space for nine integers that, if the function succeeds, receives the individual score values assigned to the nine characteristics. The nine integers map to the characteristics in the same order as specified for the elements of the above Stat buffer.
Total	Must point to an integer that, if the function succeeds, is set to the total score value calculated by the scoring mechanism by adding all individual score values.

	Description
Threshold	Must point to an integer that, if the function succeeds, is set to the total score value required to consider this software product activated.
Notebook	Must point to an integer that, if the function succeeds, is set to 1 if the computer is considered to be a notebook computer and otherwise set to 0.
HardwareHash	Must point to a null-terminated ASCII string specifying the Hardware Hash that represents the original hardware configuration.
ActivationCode	Must point to a null-terminated ASCII string specifying the Activation Code matching the given Hardware Hash.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION
    - The current hardware configuration is too different from the original hardware configuration given by the Hardware Hash.
- LIC_ERROR_INTERNAL
    - Somebody has illicitly modified the individual score values contained in the licact.dll file.
    - The function could not free memory on the process heap.
    - One of the SAFEARRAYs could not be accessed (COM only).
- LIC_ERROR_INVALID_ACTIVATION_CODE
    - The given Activation Code is invalid or does not match the given Hardware Hash.
- LIC_ERROR_INVALID_HARDWARE_HASH
    - An invalid Hardware Hash was specified.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_NOT_ENOUGH_MEMORY
    - The function could not allocate memory on the process heap.

Remarks

The function is successful if it either returns either LIC_NO_ERROR or LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION. In the former case, your software application is considered activated. In the latter case substantial hardware changes have been applied to the computer since activation that invalidate the activation. Your software application is not activated but the Stat, Scores, Total, Threshold, and Notebook parameters have still been set to the correct values.

Assume that we would like to implement a variant of Product Activation where the Hardware Hash representing the original hardware configuration and the Activation Code have been stored in a file during the activation process. To display the activation details of our software application, we would proceed as follows.

- Retrieve the Activation Code and the Hardware Hash from the file.
- Pass both to this function.
- If the result is LIC_NO_ERROR, the software application is activated. Display the returned status information. The returned total score will be greater than or equal to the returned threshold.
- If the result is LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION, the software is not activated anymore, because the hardware configuration has been substantially changed. Still, display the returned status information. The returned total score will be lower than the returned threshold.
- For all other result codes, display an error message.

5 Wrapping executables

The wrap.exe command line tool contained in the top-level directory of this ZIP archive puts a *wrapper* around a given executable to obtain a *wrapped executable*. The wrapper consist of new program code that is injected into a given executable by wrap.exe. When the wrapped executable is run this new program code is executed before the original code of the executable and has two effects: It improves the robustness of the executable against attacks by software pirates and it enables us to add Product Activation to software for which we do not have the source code.

5.1 Improved robustness

The wrapper improves the robustness of an executable against illicit modification by a software pirate. While the cryptographic wizardry underlying Product Activation reliably protects us from key generators, an attacker is still able to modify the executable that uses Product Activation. Suppose that we have written a simple function that implements Product Activation by means of the standard API and that looks as follows.

```
void HandleProductActivation(void)
{
  int Res;
  int Payload;
  int Mode;
  char HardwareHash[15];

  Res = LicCheckActivation(PRODUCT_ID, &Payload);

  /*
   *  no Activation Code in the registry
   */

  if (Res != LIC_NO_ERROR)
  {
    Res = LicWinStep1(PRODUCT_ID, &Mode);

    if (Res != LIC_NO_ERROR)
      ExitProcess(0);

    Res = LicWinStep2(PRODUCT_ID, HardwareHash, Mode);

    if (Res != LIC_NO_ERROR)
      ExitProcess(0);

    Res = LicWinStep3(PRODUCT_ID, &Payload, HardwareHash, Mode);

    if (Res != LIC_NO_ERROR)
      ExitProcess(0);
  }
}
```

The function first checks whether the end-user has activated before by searching the registry for valid activation data with LicCheckActivation(). If the registry does not contain valid activation data, the end-user is guided through the activation process by LicWinStep1(), LicWinStep2() and LicWinStep3(). If the end-user fails to successfully complete the activation, i.e. if any of the three functions returns a value different from LIC_NO_ERROR, the program exits by calling ExitProcess(). With a little knowledge of assembly language it is not very hard for a skilled attacker to modify this function in our executable. He or she could, for example, change the following line of our function

```
Res = LicCheckActivation(PRODUCT_ID, &Payload);
```

into something like the following line.

```
Res = LIC_NO_ERROR;
```

So, instead of actually searching the registry for valid activation data the function would unconditionally assume that the registry already contains valid activation data. Our piece of software would thus never ask the end-user to activate but nevertheless work as if he or she had already activated it.

Illicitly modifying our executable like this is called *cracking* the executable. A wrapper makes cracking our executable much harder. This is typically achieved by a combination of encryption and anti-debugger devices, which try to foil reverse engineering of our program by tracing its execution with a debugger. The wrapper supports two forms of protection: static and dynamic protection.

5.1.1 Static protection

Static protection means that wrap.exe encrypts the original program code carried inside an executable and then injects new program code into the executable that decrypts the encrypted original program code at runtime. When the wrapped executable is run, the injected code is executed first, which then decrypts the encrypted original program code and, after decryption, initiates execution of the decrypted original program code. As the injected code contains anti-debugger devices, it is hard to execute it in presence of a debugger. However, without executing the injected code the original program code will remain encrypted and thus safe from reverse engineering and modification.

Static protection is enabled by default and cannot be disabled. When injecting the wrapper into the executable to be protected, a password has to be specified that is used by wrap.exe to encrypt the original program code.

5.1.2 Dynamic protection

With static protection the injected code decrypts the complete original program code when the executable is run. Although it is hard to execute the injected code in presence of a debugger, an attacker is able to attach a debugger to our running program *after* the injected code has been executed. At this point the original program code has been completely decrypted and the attacker is thus able to gain access to a decrypted version of the complete original program code. In contrast, dynamic protection divides the original program code into chunks of 4096 bytes named *pages* and only decrypts those pages that are needed at a certain point in time. If pages are not needed anymore, the wrapper re-encrypts them. Thus, at any point in time, only the needed pages are in the decrypted state. All other pages are in the encrypted state. To illustrate this idea suppose that we have a program that contains two functions and which has the following structure.

```
void Sub(void)
{
  /*
   *  program code for Sub
   */
}

void Func(void)
{
  Sub();
}
```

The function Func() simply invokes a second function named Sub(). Now let us have a look at what dynamic protection does in principle. Suppose that the program code for Func() is located in page #1 and the program code for Sub() in page #2. Initially both pages are still encrypted. When Func() is called, page #1 is decrypted to unveil the program code for Func(), which can then be executed. Page #2 remains encrypted. Once Func() calls Sub(), page #1 is re-encrypted to hide the program code for Func() and page #2 is decrypted to unveil the program code for Sub() instead, enabling execution of Sub(). When Sub() returns, page #2 is re-encrypted to hide the program code for Sub() and page #1 is decrypted to unveil the program code for Func() instead and thus enable further execution of Func(). Finally, when Func() returns, page #1 is re-encrypted to hide the program code for Func(). So, only the single page that contains the currently executed program code is in the decrypted state at any point in time.

As can easily be seen, switching execution from one function to another function requires the decryption of the page that contains the code of the function to be entered and the re-encryption of the page that contains the function to be left. Although encryption and decryption are fast operations, their overhead reduces the performance of the protected executable. To counter this performance penalty the wrapper offers to leave the most recently needed *n* pages in the decrypted state, where *n* is a configurable number. If we chose *n* to be 2 in the above example, the wrapper would first decrypt page #1 to get access to the code for Func(). When Func() calls Sub(), the wrapper would additionally decrypt page #2 without re-encrypting page #1, as it is allowed to keep up to 2 pages in the decrypted state at the same time. This saves on encryptions and decryptions and thus reduces the performance penalty. However, the higher the number of pages that are visible at the same time, the larger the part of your program code that an attacker sees (2 pages = 2 x 4096 bytes = 8192 bytes) when he or she attaches a debugger to your running program. The value chosen for *n* is thus always a trade-off between performance and security. We would typically start out with a small value, run our program, and, if we are not satisfied with its performance, increase *n*.

Dynamic protection is not enabled by default. When enabling it, the number of pages that may be kept in the decrypted state at the same time by the wrapper must be specified.

5.2 Protecting without source code

Up to now we have assumed that we have access to the source code of the executable to be protected. We would then simply add calls to functions of the standard API or the advanced API to our source code to implement Product Activation. However, if the piece of software that we want to protect were written by somebody else and this somebody did not disclose the corresponding source code to us, we could not add these function calls. Luckily, the wrapper allows us to work our way around this problem.

In addition to decrypting the original program code and to providing anti-debugger devices the wrapper code injected for static protection can be instructed to load a dynamic link library (DLL) and call a function in this library to obtain the password to be used for decryption. If the function returns a password, the wrapper uses it to decrypt the original program code. If the function does not return a password, no decryption takes place as no password is available and the wrapper simply terminates the running executable. This enables us to write our own DLL that implements Product Activation via the standard API or the advanced API and hook this DLL into the wrapper and thus into the executable to be protected - although we do not have the source code of the executable. As a result the wrapped executable will only work if our DLL says "Go!".

So, by means of the described mechanism we can make the wrapper load our self-made DLL and call our function inside. Our function would then, for example, determine whether the registry already contains valid activation data. If it did, the function would indicate to the wrapper - by returning the decryption password - that everything is on track and that it shall proceed with the decryption of the original program code and initiate execution of the original program code. If the registry did not contain valid activation data, the end-user would be guided through the activation process. If the end-user successfully completed the activation procedure, our function would again indicate to the wrapper - by returning the decryption password - that it shall proceed with the decryption of the original program code and initiate execution of the original program code. Otherwise our function would not return a password and thus indicate to the wrapper that program execution is to be aborted.

The wrapper expects our DLL to be named LicWrap.dll. The search rules of LoadLibrary() apply. Placing the LicWrap.dll DLL in the same directory as the wrapped executable is therefore a good choice. The function to be called needs to be named LicWrap(), needs to use the STDCALL calling convention, and is required to have the following prototype.

```
BOOL __stdcall LicWrap(char *ModulePath, char *Password)
```

The wrapper looks for a function named "_LicWrap@8" as well as a function named "LicWrap".

If the function wants to return a decryption password, it must copy it to the buffer pointed to by the "Password" parameter. In this case the return value must be TRUE. The "Password" buffer has a size of 31 bytes. The length of the returned password must thus not exceed 30 characters. If the function does not want to return a decryption password, it must return FALSE.

To enable the function to verify the integrity of the protected executable as an additional layer of security, the "ModulePath" parameter points to the path of the protected executable. The function can then, for example, use the path to open the executable file, calculate a checksum over it, and return FALSE if the checksum indicates that the file has been illicitly modified. However, this additional integrity check is strictly optional.

If we used "secret" as the encryption password, a minimalistic implementation of LicWrap(), which always instructs the wrapper to do the decryption and which does not verify the integrity of the protected executable, would look as follows.

```
BOOL __stdcall LicWrap(char *ModulePath, char *Password)
{
  lstrcpy(Password, "secret");
  return TRUE;
}
```

The LicWrap.dll DLL contained in the top-level directory of this ZIP archive is an implementation of Product Activation to be used with the wrapper. Its C source code is available in the LicWrap subdirectory.

When the wrapper is injected into the executable the DLL to be used has to be specified. Why is this necessary? After all, the wrapper always looks for a DLL named LicWrap.dll. So, why is it necessary to specify the DLL to be used? When the wrapper is injected, a checksum over the specified DLL is calculated and embedded into the wrapper. This enables the wrapper to verify the integrity of the DLL between loading the DLL and calling LicWrap(). Illicit modification of the DLL by a software pirate hence becomes harder. So, the DLL must be specified to allow wrap.exe to calculate the checksum to be embedded into the wrapper. The injected wrapper does not know about the specified DLL and always uses LicWrap.dll as the DLL name. Hence, the wrapper always compares the checksum that wrap.exe calculated over the specified DLL with the checksum of the LicWrap.dll DLL that it has loaded. There's an important lesson to be learned from this.

IMPORTANT: When you modify your LicWrap.dll DLL (even if you simply re-compile it), you **must always re-wrap** the executable to generate a new wrapped executable as the new version of the LicWrap.dll DLL will have a checksum that is different from the checksum of the old DLL version! The effect of re-wrapping with the new LicWrap.dll DLL is to create a new wrapped executable that contains the checksum of the new DLL. Running the old protected executable that still contains the old checksum in conjunction with the new LicWrap.dll DLL would cause the wrapper to think that the DLL has been modified and a corresponding error message (error code 05) to be displayed.

5.3 Error messages

If the wrapper encounters a problem, it displays a message box that contains a two-digit error code describing the problem. The following table lists all possible error codes and their meaning.

Error code	Meaning
01	The wrapper could not load the LicWrap.dll DLL. Remember that the wrapper uses the same search strategy as LoadLibrary(). Did you place the LicWrap.dll DLL in a directory where it can be found?

Error code	Meaning
02	The path of the loaded LicWrap.dll DLL could not be determined. The wrapper needs the path to open the file for calculating its checksum.
03	The wrapper was unable to allocate 512 kilobytes of temporary buffer space. The buffer space is required for calculating the checksum over the loaded LicWrap.dll DLL.
04	The wrapper was unable to read data from the LicWrap.dll file for calculating the checksum over it.
05	The checksum calculated over the LicWrap.dll file was different from the checksum embedded into the wrapper by wrap.exe. This indicates that the LicWrap.dll file has been changed since the wrapper was injected into the protected executable.
06	The 512 kilobytes allocated for the temporary buffer could not be freed.
07	The wrapper was unable to find the LicWrap() function in the loaded LicWrap.dll DLL. Remember that LicWrap() must adhere to the prototype given above. The wrapper looks for a function exported as "_LicWrap@8" or "LicWrap".
08	The path of the running wrapped executable could not be determined. The wrapper needs the path to pass it to LicWrap().
09	The wrapper could not change the memory protection mode of the original program code in the running wrapped executable from read-only to read-write. For decryption the wrapper needs read-write access to this part of the loaded in-memory executable.
10	The original program code yielded an invalid checksum. After decrypting the original program code, the wrapper tests its integrity by calculating a checksum. If the original program code was not illicitly modified by an attacker, the most likely cause is that the decryption failed because LicWrap() returned a password that did not match the password that was specified when injecting the wrapper into the executable. In this case make sure that the same password is used in both places.
11	The wrapper could not reset the memory protection mode of the original program code in the loaded in-memory executable.
12	The process ID of the running wrapped executable could not be determined.
13	A temporary file required for dynamic protection could not be created.
14	A temporary file required for dynamic protection could not be written.
15	The auxiliary process required for dynamic protection could not be created.

5.4 Putting everything to work

5.4.1 Static protection

Suppose we simply want to apply static protection to the Notepad executable that comes with Windows and that the notepad.exe executable is located in C:\WINDOWS\system32. This is the most basic form of using wrap.exe. It just requires the input file (in our case this would be "C:\WINDOWS\system32\notepad.exe"), the output file (let's use "out.exe"), and a password (let's use "blah") for the encryption of the original program code of notepad.exe. If we open a command prompt and change the current directory to the top-level directory of this ZIP archive, the following command will accomplish the task.

```
D:\LicActSDK>wrap C:\WINDOWS\system32\notepad.exe out.exe blah
```

The resulting out.exe executable now contains the original program code taken from the notepad.exe executable plus the injected wrapper code. Just run the out.exe executable and you will... not notice anything apart from a short delay before the Notepad window appears. This delay is due to the injected wrapper code being executed before the original program code taken from the notepad.exe executable. So, we have now successfully protected Notepad using static protection.

IMPORTANT: The generated wrapped executable (in this case "out.exe") is composed of the full contents of the executable to be protected (in this case "notepad.exe") plus the injected wrapper. The wrapped executable is therefore fully self-contained. There is no need to distribute the original unprotected executable ("notepad.exe") together with the wrapped executable ("out.exe").

5.4.2 Hooking into the wrapper

Suppose that we now want to protect Notepad with Product Activation. To accomplish this we use the LicWrap.dll DLL in the top-level directory of this ZIP archive. As the LicWrap.dll DLL returns "secret" as the password, we will have to use this password when invoking wrap.exe. The DLL to be used is specified with the "-w" ("wrapper DLL") option, as in "-w LicWrap.dll". The following command will hence do the job.

```
D:\LicActSDK>wrap C:\WINDOWS\system32\notepad.exe out.exe secret -w LicWrap.dll
```

Again, the resulting out.exe executable consists of the original program code taken from the notepad.exe executable plus the injected wrapper code. If we run the out.exe executable, the activation process will be initiated. If this does not happen, then your registry probably already contains valid activation data. Otherwise, after we have successfully activated, the corresponding activation data will be stored in the registry and the Notepad window will be opened. So, again, the injected wrapper code is executed first, which then calls LicWrap() in the LicWrap.dll DLL, and, if the LicWrap.dll DLL says "Go!" because the end-user has successfully activated or valid activation data have been found in the registry, initiates the execution of the original program code taken from the notepad.exe executable.

5.4.3 Dynamic protection

Suppose that we finally want to apply dynamic protection to Notepad without, however, hooking the LicWrap.dll DLL into the wrapper. Dynamic protection is enabled with the "-r" ("runtime encryption/decryption") option. This option takes as its single argument the number *n* of pages that may be in the decrypted state at the same time, as in "-r 4" for four pages. The minimal number of pages that can be specified is 4, which corresponds to 4 x 4096 bytes = 16384 bytes. The maximal number is 500, which corresponds to 500 x 4096 bytes = 2000 kilobytes. The lower the number, the higher the level of protection and the higher the performance penalty. The following command will do the job, this time with "bob" as the password.

```
D:\LicActSDK>wrap C:\WINDOWS\system32\notepad.exe out.exe bob -r 4
```

As static protection is always in place, even in presence of dynamic protection, it will take a few moments before we see the Notepad window when we run the out.exe executable. In addition, Notepad performance will be slightly degraded as the wrapper is continuously encrypting and decrypting pages in the background while the out.exe executable is running.

To hook the LicWrap.dll DLL into the wrapper and at the same time use dynamic protection simply combine the "-w" option and the "-r" option. If you use the LicWrap.dll DLL contained in the top-level directory of this ZIP archive, do not forget to always use "secret" as the password given to wrap.exe.

5.5 Other important things

- The current version of the wrap.exe tool does not support wrapping DLLs. The only supported files are executables.
- The current version of the wrap.exe tool does not support executables with atypical

characteristics, e.g. executables that contain relocation information. If the wrap.exe tool aborts with an error message that suggests that the executable to be wrapped contains data that cannot be handled, try adding the "-s" ("strip") option to the command line. This option will try to remove the atypical characteristics from the executable.

- Use the "-q" ("quiet") option to suppress the verbose output that is enabled by default.
- Dynamic protection currently does not work reliably on some 486 processors.
- Perform extensive tests with your wrapped executables, especially when using dynamic protection! Any wrapper has to apply functionality supplied by Windows in a way in which it was never meant to be used. So, the wrapper has a higher potential of causing trouble than "normal" Windows applications, although it is well-tested and did not cause any problems in our test environments. Still, in theory it might be the case that your executable cannot be dynamically protected - or even wrapped at all.

6 High Availability

Up to now we have always talked about *the* activation server. However, to ensure high availability of the activation service even in the presence of natural disasters, airplane crashes, network connections cut by construction work, etc., we actually maintain activation servers at two geographically distinct sites. Each of the two sites is in itself redundant (server hardware, network hardware), so that the additional redundancy offered by two sites is really only for true worst-case scenarios.

6.1 Background

The first site is act.licenturion.com. This is the master site that in the normal course of operation receives and processes activation requests. Changes to the activation database on act.licenturion.com are replicated to the hot-standby site act2.licenturion.com in real-time, so that act2.licenturion.com mirrors the state of the master site at any time. It does not, however, accept activation requests. Only if failure of the master site is detected, cuts the hot-standby site its link to the master site and starts accepting activation requests.

Based on this, high availability for automatic activation is implemented as follows.

- Try to reach act.licenturion.com. The timeout is 20 seconds.
- If act.licenturion.com is reachable and works properly, use it to activate.
- If act.licenturion.com is not reachable or if it does not work properly, try to reach act2.licenturion.com. Again, the timeout is 20 seconds.
- If act2.licenturion.com is reachable and works properly, use it to activate.
- If act2.licenturion.com is not reachable or if it does not work properly, return an error.

High availability is thus achieved by having the activation client determine whether the master site or the hot-standby site currently is on duty. Simple, quite manageable, and pretty effective.

For manual activation the analogous mechanism would be to tell the end-users "Simply direct your web-browser to act.licenturion.com first and, if nothing happens there, try act2.licenturion.com." Although the hot-standby site comes into play only in, as mentioned above, true worst-case scenarios, we should do better than this. The advanced API therefore contains one other function, **GetActivationUrl()**. The idea is that your application supplies a "perform manual activation" button. When this button is clicked, your application calls **GetActivationUrl()** to determine the URL to be used for manual activation, i.e. either a URL based on act.licenturion.com or a URL based on act2.licenturion.com. Your application then uses the Win32 function **ShellExecute()** to open the returned URL in the end-user's web-browser.

As an aside, WinStep3() uses this function to determine which URL to open when the end-user clicks on the "act.licenturion.com" link.

6.2 GetActivationUrl()

This function tries to reach act.licenturion.com for 20 seconds. If act.licenturion.com is reachable and works properly, a URL based on act.licenturion.com is returned. Otherwise the function tries to reach act2.licenturion.com for 20 seconds. If act2.licenturion.com is reachable and works properly, a URL based on act2.licenturion.com is returned.

COM - GetActivationUrl(Url)

	Direction	**Type (C)**	**Type (Visual Basic)**
Url	out	BSTR *	String

	Description
Url	If the function succeeds, this parameter receives the URL to be used for manual activation.

DLL - LicGetActivationUrl(ProductId, Url)

	Description
ProductId	Must point to a null-terminated ASCII string specifying the Product ID of your licact.dll file.
Url	Must point to a buffer of at least 500 bytes that, if the the function is successful, receives the null-terminated ASCII string specifying the URL to be used for manual activation.

Result codes

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_COMMUNICATION
    - Neither act.licenturion.com nor act2.licenturion.com could be reached.
- LIC_ERROR_INTERNAL
    - The mutex protecting the global storage could not be obtained.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.

Remarks

No remarks.

7 Constants

7.1 Result codes

Constant	**Value**
LIC_NO_ERROR	0
LIC_ERROR_ALREADY_ACTIVATED	1
LIC_ERROR_INTERNAL	2
LIC_ERROR_NOT_ENOUGH_MEMORY	3
LIC_ERROR_INVALID_INDEX	4
LIC_ERROR_CANNOT_READ_FROM_REGISTRY	5
LIC_ERROR_CANNOT_WRITE_TO_REGISTRY	6
LIC_ERROR_INVALID_HARDWARE_HASH	7
LIC_ERROR_CHANGED_HARDWARE_CONFIGURATION	8
LIC_ERROR_CANNOT_DELETE_FROM_REGISTRY	9
LIC_ERROR_INVALID_PRODUCT_ID	11
LIC_ERROR_CANCEL	12
LIC_ERROR_LIBRARY_NOT_FOUND	13
LIC_ERROR_NO_LIBRARY_LOADED	14
LIC_ERROR_CANNOT_CREATE_DIALOG	16
LIC_ERROR_INVALID_ARRAY_FORMAT	17
LIC_ERROR_STRING_TOO_LONG	18
LIC_ERROR_INVALID_ACTIVATION_CODE	19
LIC_ERROR_INVALID_SERIAL_NUMBER	20
LIC_ERROR_COMMUNICATION	21
LIC_ERROR_ACTIVATION_DENIED	22
LIC_ERROR_EXPIRED	23

7.2 Activation modes

Constant	**Value**
LIC_MODE_AUTOMATIC	0
LIC_MODE_CONVENTIONAL	1
LIC_MODE_CUSTOMIZED	2

7.3 Type of hardware components

Constant	**Value**
LIC_HARDDRIVE	0
LIC_CD_ROM_DRIVE	1

Constant	Value
LIC_MASS_STORAGE_CONTROLLER	2
LIC_GRAPHICS_ADAPTER	3
LIC_CPU_TYPE	4
LIC_MEMORY_SIZE	5
LIC_VOLUME_SERIAL_NUMBER	6
LIC_CPU_SERIAL_NUMBER	7
LIC_MAC_ADDRESS	8

7.4 States of characteristics

Constant	Value
LIC_STATUS_ADDED	0
LIC_STATUS_REMOVED	1
LIC_STATUS_CHANGED	2
LIC_STATUS_MISSING_AND_MATCHING	3
LIC_STATUS_PRESENT_AND_MATCHING	4

7.5 Predefined registry keys

Constant	Value
LIC_HKEY_CLASSES_ROOT	0
LIC_HKEY_CURRENT_USER	1
LIC_HKEY_LOCAL_MACHINE	2
LIC_HKEY_USERS	3
LIC_HKEY_PERFORMANCE_DATA	4
LIC_HKEY_CURRENT_CONFIG	5
LIC_HKEY_DYN_DATA	6