# Using Product Keys and Personal Product Keys 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 Keys and Licenturion Personal Product Keys into your applications. Let's first have a look at how these two licensing schemes work.

## 1.1 Product Keys

Product Keys are case-insensitive sequences of 32 characters, e.g.

AJLQ-PMYA-LAXE-CDSH-HTKQ-EHG7-Z3ZR-BRBX

The idea is to make your application or certain features of your application available to an end-user only after he or she has supplied a valid Product Key to your application. Let us assume that you were selling a shareware word processor. You would then, for example, choose to restrict the shareware version by disabling - or "locking" - the "save" and "save as" functions. In this way an end-user would be able to install and evaluate your word processor 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 buy a Product Key from you, supply it to the word processor installed on his or her computer, and thus enable - or "unlock" - the "save" and "save as" functions.

Each of the 32 characters of a Product Key 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.

Obviously, your application must be able to determine whether the sequence of letters and digits supplied by the end-user is a valid Product Key or just random garbage. In addition, nobody but you should be able to generate valid Product Keys.

The former requirement can easily be met. All sequences that constitute valid Product Keys 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 Product Key. If it does not comply with the construction plan, it is considered to be random garbage. If your application knows the construction plan it is hence able to judge whether the sequence entered by the end-user is a valid Product Key or whether it is not.

So, what about the latter requirement? Here the whole story becomes a bit more complicated, since we have the following problem. In order to be able to check whether a sequence given by an end-user is a valid Product Key your application needs to know the construction plan underlying the creation of valid Product Keys. 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 Product Keys - a "key generator". This is quite common. Just search the Internet for the terms "key generator" or "keygen".

That's why the idea underlying Licenturion Product Keys 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 matches the secret construction plan - without, however, actually knowing the secret construction plan. In this way, the ability to create valid Product Keys is separated from the ability to verify whether a given sequence is a valid Product Key. A pirate analyzing your application can, at maximum, learn how to verify Product Keys. He or she cannot find out how to generate valid Product Keys.

This separation of creation and verification of Product Keys is based on public key cryptography. Product Keys are digitally signed messages. The secret creation plan

corresponds to the secret key employed during signature creation, the ability to verify Product Keys corresponds to knowledge of the public key for signature verification. The rocket science part of the whole story is that Licenturion Product Keys consist of only 32 characters, while conventional digital signatures would typically result in Product Keys with a length of more than 200 letters.

Product Keys are generated by the Licenturion server and can then be downloaded. For evaluation purposes the acquisition of up to ten Product Keys is free.

### 1.2 Personal Product Keys

In addition to "key generators" for illicitly generating valid Product Keys, any schemes working similarly to Product Keys face the threat of end-users sharing their Product Key with other end-users or using a single Product Key to unlock more than one installation of your application. The former threat can be mitigated by psychologically discouraging end-users from disclosing their Product Key to others. This is the idea behind Personal Product Keys. Both threats can also be addressed by technical means - as implemented by Licenturion Product Activation - instead of psychological means.

Personal Product Keys bind a Product Key to the identity of the end-user that the Product Key has been issued to. They consist of a case-insensitive Product Key, e.g.

G7B9-G9E7-FGGF-KYED-F497-YPJK-7Q7P-BYGJ

and an identification string - the *user ID* - of maximally 200 characters that uniquely identifies the end-user, e.g.

Donna Haywood

The user ID is case-sensitive and may only consist of (7-bit) ASCII characters.

A Personal Product Key will only be considered valid, if both of its parts, i.e. the Product Key and the user ID, are presented. When unlocking your application or selected features of your application, end-users therefore always have to supply their identification string in addition to their Product Key. So, sharing a Personal Product Key with other end-users requires sharing the user ID. This prevents anonymous sharing of Personal Product Keys and establishes a psychological barrier that discourages end-users from disclosing their Personal Product Key to others.

Personal Product Keys also employ digital signatures based on public key cryptography. The ability to create a Product Key that, together with a given user ID, constitutes a valid Personal Product Key is separated from the ability to determine whether a Product Key and a User ID given by an end-user form a valid Personal Product Key. In order to keep the construction plan for valid Personal Product Keys secret and to foil the creation of "key generators", your application, as described for Product Keys, only implements the latter.

Resembling Product Keys, Personal Product Keys are generated by the Licenturion server and can then be downloaded. For evaluation purposes the acquisition of up to ten Personal Product Keys is free.

### 1.3 Unlocking individual features of a software application

(Personal) Product Keys contain a 32-bit payload. The interpretation of the payload is up to the software application. It could extract the payload from a (Personal) Product Key and, for example, use it as a bit-mask, each of the 32 bits enabling or disabling a certain feature of the application.

The 32-bit payload is protected against illicit modification. This is achieved by the same mechanism that prevents software pirates from writing a "key generator", i.e. digital signatures based on public key cryptography.

### 1.4 Time-limited (Personal) Product Keys

When generating (Personal) Product Keys, the Licenturion server can be instructed to use the most significant 13 bits of the 32-bit payload for storing an expiration date in the (Personal) Product Keys, still leaving the least significant 19 bits for user-defined content. When the expiration date of a (Personal) Product Key is reached, the (Personal) Product Key becomes invalid and the software application returns into the state that it had originally been in before the (Personal) Product Key was entered by the end-user. These temporary (Personal) Product Keys are typically used in shareware scenarios to implement trial periods, during which a fully unlocked 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 the entry of a (Personal) Product Key to zero and thus extends trials periods *ad infinitum*.

### 1.5 Implementing (Personal) Product Keys

End-users are typically required to enter their (Personal) Product Key only once, e.g. during installation, when running your software application for the first time, or, in the shareware case, when opting to upgrade from the shareware version of your software application to the full version. If the entered information is valid, it is stored. Verifying whether the end-user has already entered a valid (Personal) Product Key is then as easy as retrieving the stored information and checking whether it constitutes a valid (Personal) Product Key.

The functionality required for Product Keys and Personal Product Keys can be made available to your application as

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

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

**IMPORTANT:** If you choose to integrate (Personal) Product Keys into your application using the COM approach or the DLL approach, you have to supply your lickey.dll file to your end-users in the distribution package of your application. **Never** install your lickey.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 lickey.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 (Personal) Product Keys, which includes the other vendor's version of the lickey.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 lickey.dll file is unique**. Your DLL only recognizes your (Personal) Product Keys and the other vendor's DLL only recognizes that vendor's (Personal) Product Keys. Technically speaking, your DLL contains your public key and the other vendor's DLL contains the other vendor's public key. 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 (Personal) Product Keys any longer. So, by installing your lickey.dll file into a private location such as the installation folder of your executable file(s), you ensure that your lickey.dll file is not touched by anyone else and that your application always uses your lickey.dll file.

Each of the unique versions of the lickey.dll file is unambiguously identified by a Product ID. To find out which Product ID your lickey.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.

### 1.5.1 The LicKey COM component

Before a COM component is available it must be registered. The LicKey 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-lickey-dll

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

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

D:\LicKeySDK\licreg.exe

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

To unregister the LicKey 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 LicKey component to be unregistered instead of it being registered.

**IMPORTANT:** Keep in mind that you must also register the LicKey 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 LicKey COM component is correctly registered and unregistered.

The LicKey COM component includes a type library which is also registered and unregistered via the self-registration mechanism. The LicKey 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

LicKeyLib 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

LicKeyLib 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 lickey.dll file.

**IMPORTANT:** Double-check that you are using the correct type library! If more than one LicKey COM component have been registered on a computer, e.g. by other vendors also using (Personal) Product Keys or by yourself using (Personal) Product Keys for more than one product, all LicKey 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 LicKey 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.

### 1.5.2 The LicKey dynamic link library (DLL)

In addition to the LicKey COM component, the lickey.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 lickey.dll. For C and C++ development, we supply the header file lickey.h along with the import library lickey.lib in the top-level directory of this ZIP archive.

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

**IMPORTANT:** As mentioned above, make absolutely sure that your application uses the correct lickey.dll file by installing your lickey.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 lickey.dll take the Product ID of your lickey.dll file as their first argument. Each function of lickey.dll then verifies whether the Product ID passed by your application matches the Product ID of the lickey.dll file. If a mismatch is detected, an error is returned. This has the following effect. Let us assume that your lickey.dll file has a Product ID of 1234ABCD. Your application therefore passes 1234ABCD as the first argument to all functions in your lickey.dll and lickey.dll notices on each function call that the Product ID is correct. Let us now assume that, perhaps because your lickey.dll file was accidentally overwritten by the end-user with another vendor's lickey.dll file in spite of all the care you have taken, your application erroneously invokes a function in the wrong lickey.dll which has a Product ID of, say, 2345BCDE. So, your application still passes 1234ABCD - but to the wrong lickey.dll. The wrong lickey.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.

### 1.5.3 The LicKey static library

A static library is supplied for use by C or C++ developers who prefer the (Personal) Product Key functionality to reside inside their executable files instead of the separate lickey.dll file. The static library lickeys.lib exports the same functions as the dynamic link library lickey.dll. The function declarations are identical and, hence, the header file lickey.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 lickey.h header file.

**IMPORTANT:** Include the standard windows.h header file before including lickey.h in your source code, since lickey.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 <lickey.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. Be sure to link you application against the corresponding import libraries.

### 1.6 Available APIs

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

#### 1.6.1 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 asking the end-user for his or her Product Key, return a "canceled" result code if he or she pushes the cancel button of the dialog box, otherwise determine whether the characters entered by him or her constitute a valid Product Key, return an "invalid Product Key" result code in case they don't, and otherwise store the Product Key in the Windows registry and return a "valid Product Key" result code. 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 (Personal) Product Key functionality within much less than an hour.

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

#### 1.6.2 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 is a valid Product Key.

#### 1.6.3 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, see section 6.

### 1.6.4 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 1.5.3). 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.

## 1.7 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 might 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**

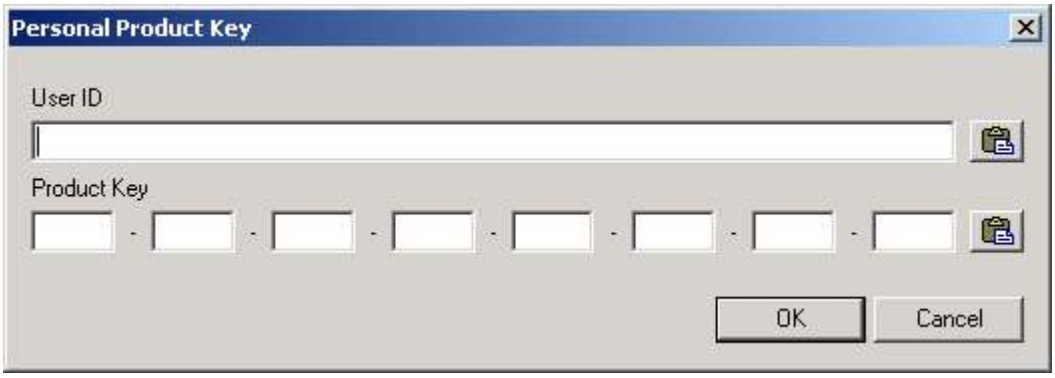
There are no compatibility-breaking enhancements, yet.

# 3 The standard API

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

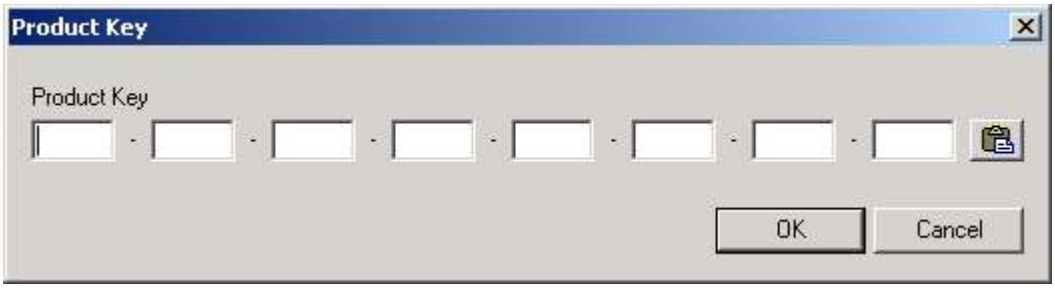
## 3.1 Basic functionality

Asking the end-user for a Product Key or a Personal Product Key takes a single function call. For Personal Product Keys the API function is **WinPersonalProductKey()**, which interacts with the end-user via the following dialog box.




The dialog box only accepts valid characters for the Product Key, i.e. characters from the set of 26 letters and digits that are used to construct valid Product Keys. Two paste buttons allow a Product Key or a User ID to be pasted from the clipboard.

Analogously, the API function to be used for Product Keys is **WinProductKey()**. Its dialog box looks like this:




Again, the dialog box only accepts valid characters, i.e. characters from the set of 26 letters and digits that are used to construct valid Product Keys. The paste button allows a Product Key to be pasted from the clipboard.

If the end-user enters a valid Personal Product Key or a valid Product Key, respectively, it is automatically stored in the Windows registry. If the entered information is invalid or if the end-user clicks the cancel button, the functions return suitable result codes. In the former case tell the end-user that the entered information is invalid and call WinPersonalProductKey() or WinProductKey() again. The dialog box will then contain the previously entered information, unless you have invoked **WinResetDialog()** to clear its contents.

To determine yourself whether the Windows registry contains a valid Personal Product Key or a valid Product Key, i.e. whether the end-user has already entered a valid (Personal) Product Key, use the **LoadPersonalProductKey()** function or the **LoadProductKey()** function.

The **CleanRegistry()** function removes a stored (Personal) Product Key from the Windows registry. It is typically called by the uninstallation routine of your software product.

If successful, WinPersonalProductKey(), WinProductKey(), LoadPersonalProductKey(), and LoadProductKey() also return the 32-bit payload embedded in the (Personal) Product Key entered by the end-user or retrieved from the Windows registry.

If you have instructed the Licenturion server to use the most significant 13 bits of the payload to carry an expiration date for the (Personal) Product Key, use **SplitPayload()** to convert the 32-bit payload into the number of days left before the (Personal) Product Key expires and the remaining 19 (least significant) user-specified bits of the payload.

For storing (Personal) Product Keys

HKEY_LOCAL_MACHINE\SOFTWARE\Licenturion GmbH\XXXXXXXX

is used by default. The eight "X" characters represent the Product ID of the product that the stored information belongs to. The end-user entering the (Personal) Product Key requires privileges that enable him or her to create those registry keys ("Licenturion" 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 information stored at this location is visible to all user accounts, this makes the entered (Personal) Product Key available to all users. Thus the (Personal) Product Key has to be entered only once 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 (Personal) Product Key information is stored at the HKEY_CURRENT_USER location instead. As this location is user-specific, this makes the entered (Personal) Product Key available only the the user account of the end-user that has entered the (Personal) Product Key. Other end-users that run the corresponding software application with other user accounts will also have to enter the (Personal) Product Key. Thus the (Personal) Product Key has to be entered separately by each end-user.

On NT-based operating systems The "Licenturion GmbH" and "XXXXXXXX" subkeys are created with "Full Control" permissions for "Everyone". CleanRegistry() will thus successfully remove the (Personal) Product Key from the Windows registry, 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, subsequently entered (Personal) Product Keys 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 WinProductKey

A dialog box is displayed that requests the end-user to enter his or her Product Key. If he or she enters a valid Product Key, it is automatically stored in the Windows registry.

**COM - WinProductKey(SeqNo, Payload)**

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

| | Description |
|---|---|
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the Product Key entered by the end-user. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the Product Key entered by the end-user. |

**DLL - LicWinProductKey(ProductId, SeqNo, Payload)**

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

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the Product Key entered by the end-user. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the Product Key entered by the end-user. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully, i.e. the end-user entered a valid Product Key.
- 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 Product Key could not be created or opened.
    - The Product Key value could not be set in the created or opened registry key.
- 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_INVALID_PRODUCT_KEY
        - The end-user entered an invalid Product Key.

**Remarks**

No remarks.

### 3.1.2 LoadProductKey

Determines whether the Windows registry contains a valid Product Key and, if it does, returns the corresponding sequence number and payload.

**COM - LoadProductKey(SeqNo, Payload)**

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

| | Description |
|---|---|
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the stored Product Key. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the stored Product Key. |

**DLL - LicLoadProductKey(ProductId, SeqNo, Payload)**

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

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the stored Product Key. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the stored Product Key. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully, i.e. the Windows registry contains a valid Product Key.
- LIC_ERROR_CANNOT_READ_FROM_REGISTRY
    - The registry key containing the Product Key could not be opened.
    - The Product Key value could not be read from the opened registry key.
- 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_INVALID_PRODUCT_KEY
    - The Product Key stored in the Windows registry is invalid.

**Remarks**

No remarks.

### 3.1.3 WinPersonalProductKey

A dialog box is displayed that requests the end-user to enter his or her Personal Product Key. If he or she enters a valid Personal Product Key, it is automatically stored in the Windows registry.

**COM - WinPersonalProductKey(UserId, SeqNo, Payload)**

| | Direction | Type (C) | Type (Visual Basic) |
|---|---|---|---|
| UserId | out | BSTR * | String |
| SeqNo | out | int * | Long |
| Payload | out | int * | Long |

| | Description |
|---|---|
| UserId | If the function is successful, this parameter receives the user ID associated with the Personal Product Key entered by the end-user. |
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the Personal Product Key entered by the end-user. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the Personal Product Key entered by the end-user. |

**DLL - LicWinPersonalProductKey(ProductId, UserId, SeqNo, Payload)**

| | Direction | Type (C) |
|---|---|---|
| ProductId | in | const char * |
| UserId | out | char * |
| SeqNo | out | int * |
| Payload | out | int * |

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| UserId | Must point to a buffer of 201 bytes that, if the function is successful, receives the user ID associated with the Personal Product Key entered by the end-user as a null-terminated ASCII string. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the Personal Product Key entered by the end-user. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the Personal Product Key entered by the end-user. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully, i.e. the end-user entered a valid Personal Product Key.
- 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 Personal Product Key could not be created or opened.
    - The Personal Product Key value could not be set in the created or opened registry key.
- 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_INVALID_USER_ID_OR_PRODUCT_KEY
    - The end-user entered an invalid Personal Product Key, i.e. one of its two components - the user ID or the Product Key - was invalid.
- LIC_ERROR_NOT_ENOUGH_MEMORY (COM only)
    - The function could not allocate memory on the process heap.

**Remarks**

No remarks.

### 3.1.4 LoadPersonalProductKey

Determines whether the Windows registry contains a valid Personal Product Key and, if it does, returns the corresponding user ID, sequence number, and payload.

**COM - LoadPersonalProductKey(UserId, SeqNo, Payload)**

| | Direction | Type (C) | Type (Visual Basic) |
|---|---|---|---|
| UserId | out | BSTR * | String |
| SeqNo | out | int * | Long |
| Payload | out | int * | Long |

| | Description |
|---|---|
| UserId | If the function is successful, this parameter receives the user ID associated with the stored Personal Product Key. |
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the stored Personal Product Key. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the stored Personal Product Key. |

**DLL - LoadPersonalProductKey(ProductId, UserId, SeqNo, Payload)**

| | Direction | Type (C) |
|---|---|---|
| ProductId | in | const char * |
| UserId | out | char * |
| SeqNo | out | int * |
| Payload | out | int * |

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| UserId | Must point to a buffer of 201 bytes that, if the function is successful, receives the user ID associated with the stored Personal Product Key as a null-terminated ASCII string. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the stored Product Key. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the stored Product Key. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CANNOT_READ_FROM_REGISTRY
    - The registry key containing the Personal Product Key could not be opened.
    - The Personal Product Key value could not be read from the opened registry key.
- 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_INVALID_USER_ID_OR_PRODUCT_KEY
    - The Personal Product Key stored in the Windows registry is invalid, i.e. one of its two components - the user ID or the Product Key - is invalid.
- LIC_ERROR_NOT_ENOUGH_MEMORY (COM only)
    - The function could not allocate memory on the process heap.

**Remarks**

No remarks.

### 3.1.5 WinResetDialog

Resets the contents of the dialog boxes displayed by WinProductKey() and WinPersonalProductKey().

**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 lickey.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.

### 3.1.6 CleanRegistry

Removes the Personal Product Key or Product Key stored in the Windows registry by your software application.

**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 lickey.dll file. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully.
- LIC_ERROR_CANNOT_DELETE_FROM_REGISTRY
    - The registry key that contains the (Personal) Product Key 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**

No remarks.

### 3.1.7 SplitPayload

This function is passed a Product Key payload which contains an expiration date. The expiration date is extracted and compared to the current date. If the Product Key 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 Product Key. |

**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 lickey.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 Product Key. |

**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 Product Key has expired.
    - The end-user has tried to fool the expiration mechanism by turning back the system clock.

**Remarks**

No remarks.

### 3.1.8 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 two dialog boxes used by the standard API are defined by two dialog box resources. For the COM object and the DLL default dialog box resources are supplied by lickey.dll. 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 keyres.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 default dialog box resources.

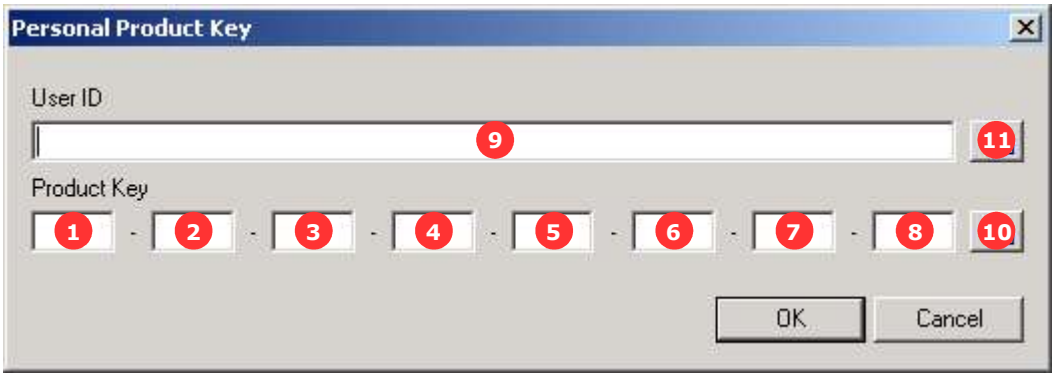
Each standard API function dealing with a dialog box accesses the corresponding dialog box resource using a resource ID as follows.

| **API function** | **Resource ID** |
|---|---|
| WinPersonalProductKey() | 1972 |
| WinProductKey() | 1973 |

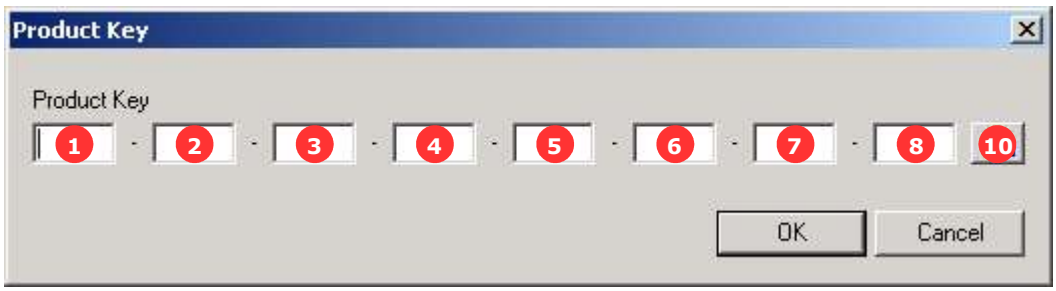
The main application icon must have a resource ID of 1974, the icon on the user ID paste button needs to have a resource ID of 1975, and the icon on the Product Key paste button is required to have a resource ID of 1976. If you do not use an API function then you do not need to care about the corresponding dialog box resource.

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

- **WinPersonalProductKey() dialog**




- **WinProductKey() dialog**




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

| # | Type | Control ID |
|---|---|---|
| 1 | edit control | 1001 |
| 2 | edit control | 1002 |
| 3 | edit control | 1003 |
| 4 | edit control | 1004 |
| 5 | edit control | 1005 |
| 6 | edit control | 1006 |
| 7 | edit control | 1007 |
| 8 | edit control | 1008 |
| 9 | edit control | 1009 |
| 11 | button control | 1010 |
| 12 | button control | 1011 |

Note that the paste buttons must have the BS_ICON style. Otherwise the standard API will be unable to assign the icon to the button.

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 11 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 the resource IDs 1972 through 1976 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 plain paste buttons without the BS_ICON style or to disable support for the paste buttons altogether.

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 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 Product Key, look 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 (Personal) Product Key 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 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 WinPersonalProductKey() and WinProductKey().

**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 lickey.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 lickey.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 WinPersonalProductKey() and WinProductKey() 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 | int |

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.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 WinPersonalProductKey() and WinProductKey() 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 16 values and an index range from 0 to 15 to the function. The first two elements (index 0 through index 1) in this array specify the resource IDs of the two dialog boxes. The third, fourth, and fifth element (index 2 through index 4) specify the resource IDs of the main application icon and the paste button icons. The remaining 11 elements (index 5 through index 15) specify the 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, 1001, 1002, 1003, ..., 1010, 1011). |

**DLL - LicWinMapIdValues(ProductId, Values)**

| | 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 lickey.dll file. |
| Values | Must point to an array of 16 integers. The first two elements (index 0 through index 1) in this array specify the resource IDs of the two dialog boxes. The third, fourth, and fifth element (index 2 through index 4) specify the resource IDs of the main application icon and the paste button icons. The remaining 11 elements (index 5 through index 15) specify the 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, 1001, 1002, 1003, ..., 1010, 1011). |
| Length | Must be set to the number of elements in the Values array, i.e. to 16. |

**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 16 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 one of the paste icons (index 3 and index 4) is set to zero, the standard API does not try to assign any icon to the respective paste button. Set these array elements to zero if you use standard paste buttons with a text caption.

If the control ID of one of the paste buttons (index 15 and index 16) is set to zero, the standard API skips the handling of messages related to the corresponding paste button. Set these array elements to zero if you use dialog boxes without paste buttons.

If the resource ID of the main application icon (index 2) is set to zero, the standard API does not try to set the application icon with WM_SETICON.

### 3.2.5 SetRegistryInfo

Specifies the two registry keys to be used for storing the (Personal) Product Key.

**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.<br>• LIC_HKEY_CLASSES_ROOT<br>• LIC_HKEY_CURRENT_USER<br>• LIC_HKEY_LOCAL_MACHINE<br>• LIC_HKEY_USERS<br>• LIC_HKEY_PERFORMANCE_DATA<br>• LIC_HKEY_CURRENT_CONFIG<br>• LIC_HKEY_DYN_DATA |
| Path1 | Passes the path of the registry key to be used for storing the (Personal) Product Key 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 lickey.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.<br>• LIC_HKEY_CLASSES_ROOT<br>• LIC_HKEY_CURRENT_USER<br>• LIC_HKEY_LOCAL_MACHINE<br>• LIC_HKEY_USERS<br>• LIC_HKEY_PERFORMANCE_DATA<br>• LIC_HKEY_CURRENT_CONFIG<br>• 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 (Personal) Product Key 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
    - The specified path 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 a (Personal) Product Key (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 (Personal) Product Key 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\Product Key

as the fallback location for storing the (Personal) Product Key, pass

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

to this function.

**IMPORTANT:** Remember that in this example calling CleanRegistry will delete the "Product Key" subkey!

### 3.2.6 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 lickey.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.

### 3.2.7 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 lickey.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 consists of a mere four functions.

- **VerifyProductKey()** and **VerifyPersonalProductKey()** verify whether the (Personal) Product Key passed to them is valid and, if it is, return the corresponding sequence number and payload.
- **StoreProductKey()** and **StorePersonalProductKey()** are the complement to LoadProductKey() and LoadPersonalProductKey() contained in the standard API. They verify whether the given (Personal) Product Key is valid and, if it is, store it in the Windows registry and return the corresponding sequence number and payload.

To determine whether the Windows registry contains a valid (Personal) Product Key, i.e. whether the end-user has already entered a valid (Personal) Product Key, call the **LoadProductKey()** function or the **LoadPersonalProductKey()** function included in the standard API.

The **CleanRegistry()** function supplied by the standard API discards any information stored in the Windows registry. It is typically called by the uninstallation routine of your software product.

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 (Personal) Product Key for the software application.

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

**IMPORTANT:** Verifying the (Personal) Product Key only once, then setting a "(Personal) Product Key is valid" flag, e.g. in the Windows registry, and just verifying whether this flag is set to subsequently determine whether a valid (Personal) Product Key has already been entered is 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 seen a valid (Personal) Product Key. Always store and verify the complete (Personal) Product Key. In contrast to a simple flag, it is **computationally infeasible to fake (Personal) Product Keys**.

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 (Personal) Product Key.

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

| **Subdirectory** | **Contents** |
|---|---|
| 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). |
| DialogBox | 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 (DialogBox.exe). |

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

### 4.1 VerifyProductKey

Verifies the specified Product Key and, if it is valid, returns the contained sequence number and payload.

**COM - VerifyProductKey(SeqNo, Payload, ProdKey)**

| | Direction | Type (C) | Type (Visual Basic) |
|---|---|---|---|
| SeqNo | out | int * | Long |
| Payload | out | int * | Long |
| ProdKey | in | BSTR | String |

| | Description |
|---|---|
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the passed Product Key. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the passed Product Key. |
| ProdKey | Passes the Product Key to be verified. |

**DLL - LicVerifyProductKey(ProductId, SeqNo, Payload, ProdKey)**

| | Direction | Type (C) |
|---|---|---|
| ProductId | in | const char * |
| SeqNo | out | int * |
| Payload | out | int * |
| ProdKey | in | const char * |

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the given Product Key. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the given Product Key. |
| ProdKey | Must point to a null-terminated ASCII string specifying the Product Key to be verified. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully, i.e. the specified Product Key is valid.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_INVALID_PRODUCT_KEY
    - The specified Product Key is invalid.

**Remarks**

No remarks.

### 4.2 VerifyPersonalProductKey

Verifies the specified Personal Product Key and, if it is valid, returns the contained sequence number and payload.

**COM - VerifyPersonalProductKey(SeqNo, Payload, UserId, ProdKey)**

| | Direction | Type (C) | Type (Visual Basic) |
|---|---|---|---|
| SeqNo | out | int * | Long |
| Payload | out | int * | Long |
| UserId | in | BSTR | String |
| ProdKey | in | BSTR | String |

| | Description |
|---|---|
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the passed Personal Product Key. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the passed Personal Product Key. |
| UserId | Passes the user ID component of the Personal Product Key to be verified. |
| ProdKey | Passes the Product Key component of the Personal Product Key to be verified. |

**DLL - LicVerifyPersonalProductKey(ProductId, SeqNo, Payload, UserId, ProdKey)**

| | Direction | Type (C) |
|---|---|---|
| ProductId | in | const char * |
| SeqNo | out | int * |
| Payload | out | int * |
| UserId | in | const char * |
| ProdKey | in | const char * |

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the passed Personal Product Key. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the passed Personal Product Key. |
| UserId | Must point to a null-terminated ASCII string specifying the user ID component of the Personal Product Key to be verified. |
| ProdKey | Must point to a null-terminated ASCII string specifying the Product Key component of the Personal Product Key to be verified. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully, i.e. the specified Personal Product Key is valid.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)

    - An invalid Product ID was specified.
- LIC_ERROR_INVALID_PRODUCT_KEY (COM only)
    - The specified Product Key was too long.
- LIC_ERROR_INVALID_USER_ID (COM only)
    - The specified user ID was too long.
- LIC_ERROR_INVALID_USER_ID_OR_PRODUCT_KEY
    - The specified Personal Product Key, i.e. one of its two components - the user ID or the Product Key -, was invalid.

**Remarks**

No remarks.

### 4.3 StoreProductKey

Verifies the specified Product Key and, if it is valid, stores it in the Windows Registry and returns the contained sequence number and payload.

**COM - StoreProductKey(SeqNo, Payload, ProdKey)**

| | Direction | Type (C) | Type (Visual Basic) |
|---|---|---|---|
| SeqNo | out | int * | Long |
| Payload | out | int * | Long |
| ProdKey | in | BSTR | String |

| | Description |
|---|---|
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the passed Product Key. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the passed Product Key. |
| ProdKey | Passes the Product Key to be verified and stored. |

**DLL - LicStoreProductKey(ProductId, SeqNo, Payload, ProdKey)**

| | Direction | Type (C) |
|---|---|---|
| ProductId | in | const char * |
| SeqNo | out | int * |
| Payload | out | int * |
| ProdKey | in | const char * |

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the given Product Key. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the given Product Key. |
| ProdKey | Must point to a null-terminated ASCII string specifying the Product Key to be verified and stored. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully, i.e. the specified Product Key is valid.
- LIC_ERROR_CANNOT_WRITE_TO_REGISTRY
    - The registry key to store the Product Key could not be created or opened.
    - The Product Key value could not be set in the created or opened registry key.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_INVALID_PRODUCT_KEY

- The specified Product Key is invalid.

**Remarks**

No remarks.

### 4.4 StorePersonalProductKey

Verifies the specified Personal Product Key and, if it is valid, stores it in the Windows registry and returns the contained sequence number and payload.

**COM - VerifyPersonalProductKey(SeqNo, Payload, UserId, ProdKey)**

| | Direction | Type (C) | Type (Visual Basic) |
|---|---|---|---|
| SeqNo | out | int * | Long |
| Payload | out | int * | Long |
| UserId | in | BSTR | String |
| ProdKey | in | BSTR | String |

| | Description |
|---|---|
| SeqNo | If the function is successful,.this parameter receives the sequence number corresponding to the passed Personal Product Key. |
| Payload | If the function is successful, this parameter receives the payload corresponding to the passed Personal Product Key. |
| UserId | Passes the user ID component of the Personal Product Key to be verified and stored. |
| ProdKey | Passes the Product Key component of the Personal Product Key to be verified and stored. |

**DLL - LicVerifyPersonalProductKey(ProductId, SeqNo, Payload, UserId, ProdKey)**

| | Direction | Type (C) |
|---|---|---|
| ProductId | in | const char * |
| SeqNo | out | int * |
| Payload | out | int * |
| UserId | in | const char * |
| ProdKey | in | const char * |

| | Description |
|---|---|
| ProductId | Must point to a null-terminated ASCII string specifying the Product ID of your lickey.dll file. |
| SeqNo | Must point to an integer that, if the function is successful, is set to the sequence number corresponding to the passed Personal Product Key. |
| Payload | Must point to an integer that, if the function is successful, is set to the payload corresponding to the passed Personal Product Key. |
| UserId | Must point to a null-terminated ASCII string specifying the user ID component of the Personal Product Key to be verified and stored. |
| ProdKey | Must point to a null-terminated ASCII string specifying the Product Key component of the Personal Product Key to be verified and stored. |

**Result codes**

- LIC_NO_ERROR
    - The function completed successfully, i.e. the specified Personal Product Key is valid.

- LIC_ERROR_CANNOT_WRITE_TO_REGISTRY
    - The registry key to store the Personal Product Key could not be created or opened.
    - The Personal Product Key value could not be set in the created or opened registry key.
- LIC_ERROR_INVALID_PRODUCT_ID (DLL only)
    - An invalid Product ID was specified.
- LIC_ERROR_INVALID_PRODUCT_KEY (COM only)
    - The specified Product Key was too long.
- LIC_ERROR_INVALID_USER_ID (COM only)
    - The specified user ID was too long.
- LIC_ERROR_INVALID_USER_ID_OR_PRODUCT_KEY
    - The specified Personal Product Key, i.e. one of its two components - the user ID or the Product Key -, was invalid.

**Remarks**

No remarks.

# 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 (Personal) Product Keys 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 (Personal) Product Keys reliably protects us from key generators, an attacker is still able to modify the executable that uses (Personal) Product Keys. Suppose that we have written a simple function that implements Product Keys by means of the standard API and that looks as follows.

```
void CheckProductKey(void)
{
  int Res;
  int SeqNo;
  int Payload;

  Res = LicLoadProductKey(PRODUCT_ID, &SeqNo, &Payload);

  /*
   *  no Product Key in the registry
   */

  if (Res != LIC_NO_ERROR)
  {
    Res = LicWinProductKey(PRODUCT_ID, &SeqNo, &Payload);

    /*
     *  no valid Product Key entered
     */

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

The function first checks whether the end-user has entered a valid Product Key before by searching the registry for a valid Product Key with LicLoadProductKey(). If the registry does not contain a valid Product Key, the end-user is asked to enter his or her Product Key by LicWinProductKey(). If the end-user fails to supply a valid Product Key, 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 = LicLoadProductKey(PRODUCT_ID, &SeqNo, &Payload);
```

into something like the following line.

```
Res = LIC_NO_ERROR;
```

So, instead of actually searching the registry for a valid Product Key the function would unconditionally assume that the registry already contains a valid Product Key. Our piece of software would thus never ask the end-user for his or her Product Key but nevertheless work as if he or she had entered 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 (Personal) Product Keys. 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 (Personal) Product Keys 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 a valid (Personal) Product Key. 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 a valid (Personal) Product Key, the end-user would be requested to enter his or her (Personal) Product Key. If the end-user entered a valid (Personal) Product Key, 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 Personal Product Keys 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? |
| 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. |

| Error code | Meaning |
|---|---|
| 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:\LicKeySDK>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 Personal Product Keys. 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:\LicKeySDK>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, a standard API window will appear that asks us to enter a Personal Product Key. If this window does not appear, then your registry probably already contains a valid Personal Product Key. Otherwise, after we have entered a valid Personal Product Key, it 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 a valid Personal Product Key has been entered or 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:\LicKeySDK>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 Constants

## 6.1 Result codes

| Constant | Value |
|---|---|
| LIC_NO_ERROR | 0 |
| LIC_ERROR_INTERNAL | 2 |
| LIC_ERROR_NOT_ENOUGH_MEMORY | 3 |
| LIC_ERROR_CANNOT_READ_FROM_REGISTRY | 5 |
| LIC_ERROR_CANNOT_WRITE_TO_REGISTRY | 6 |
| LIC_ERROR_CANNOT_DELETE_FROM_REGISTRY | 9 |
| LIC_ERROR_INVALID_PRODUCT_KEY | 10 |
| LIC_ERROR_INVALID_PRODUCT_ID | 11 |
| LIC_ERROR_CANCEL | 12 |
| LIC_ERROR_LIBRARY_NOT_FOUND | 13 |
| LIC_ERROR_NO_LIBRARY_LOADED | 14 |
| LIC_ERROR_INVALID_USER_ID_OR_PRODUCT_KEY | 15 |
| LIC_ERROR_CANNOT_CREATE_DIALOG | 16 |
| LIC_ERROR_INVALID_ARRAY_FORMAT | 17 |
| LIC_ERROR_STRING_TOO_LONG | 18 |
| LIC_ERROR_EXPIRED | 23 |
| LIC_ERROR_INVALID_USER_ID | 24 |

## 6.2 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 |