UST Solutions Inc.

UST Licensing framework (UstLF)

User Manual

Content

Introduction

UstLF is a licensing framework designed to control the behavior of a software application on the client site. Application behavior is described in a license file(s) issued by the licenser and installed on the target computer. Content of the license file is based on the features that are defined during application design/implementation phase. License file(s) dictates when, where and how these features will behave, will be enabled or disabled. This approach allows to build highly flexible marketing and sales solutions on the target application market. Technically **UstLF** allows to:

- define one or more licensed features (in the simplest case one feature can be associated just with application execution – allow or disallow execution)
- group features in feature sets and set the time periods when each feature set will be enabled (for example it is possible to issue a license that will start on a given day with a trial or a promotional period when all application features are enabled and after its end move to a period where feature set corresponds to the one defined in the sales agreement). The number of features in a feature set, the length of the associated time period (start-end dates) and the number of these feature sets are not limited (at list within reasonable numbers)
- associate additional data with a feature in a license file (this is very useful when there is a need to define at the application design time something that cannot be clearly specified. For example – if licensed application is some kind of a network server and one of the goals is to license a number of network connections, then required number of connections can be passed as a feature data in a license file. Another example – if an application dials with some file processing it is possible to specify "file import" feature and list particular file types in a license file as feature data according to the particular sales agreement).
- create a single license that targets a range of application versions or release dates (for example it is possible to issue a license that will be valid only for application version released on a given day and a year from now. In this case application user will be able to download, install and use not only existing purchased application but all upgrades that will possibly come out during selected period without necessity to update his existing license each time the new release is installed).
- use "node locked" licensing model when licenses are associated with a particular computer and/or user.
- use "floating" licensing model when licenses are stored on a special license server and application acquires license from it from any computer over the network.
- specify the number of concurrent licensed applications running on a single computer or on the network (in case of the license server)
- issue and allocate fixed licenses (that cannot be merged together)
- issue and allocate complementary license files (licenses that can be merged) from one or multiple sources by a single application
- encrypt/decrypt custom strings in the application code.

All above mentioned functionality of **UstLF** is highly and at the same time easily tunable. To make the **UstLF** functioning safe and secure library employs several modern encryption and hashing algorithms.

UstLF is targeting Windows platform and designed generally to be used with Microsoft Visual Studio C/C++, however can be adjusted and used with other programming languages. **UstLF** contains several executables, library and header files. Library file supposed to be linked with the licensed application and increase its footprint by less than 200K. There are several versions of the **UstLF** library file available for different compilation models.

How to use UstLF

(brief overview)

To integrate **UstLF** library with an application it is necessary to complete several simple steps:

1. Using utility **ukg.exe** create an asymmetric encryption key file (see **Key generator**).
2. Create a feature map file containing application specific definitions. This is a C/C++ header file. Format is extremely simple (see **Feature Map File**). Example:

```
---------------------------- Beginning of the file --------------------------

  #ifndef __USTLF_FEATURE_MAP__
  #define __USTLF_FEATURE_MAP__

      #define      MY_FEATURE_1              1
      #define      MY_FEATURE_2              2

  #endif
-------------------------------- End of the file -------------------------------
```

3. Using utility **AppDesGen.exe** create application descriptor C/C++ header file (see **Application descriptor generator**)
4. Add feature map file and application descriptor header file to the MS Visual Studio project for the target application.
5. Add to MS Visual Studio project a reference to appropriate version of the **UstLF** library file.
6. Somewhere in appropriate place of the application code where there is a need to check licensed features add the following code:

```
if ( UstLF::StartUp() )
{
      UstLF::SetEmbeddedData( __aULMPubKey__, __aEncAppDescriptor );

      if ( UstLF::AllocateLicenses( _T("C:\\MyAppLicenseFilesLocation") ) )
      {
            if ( FEATURE_STATUS_ENABLED == UstLF::GetFeatureInfo( MY_FEATURE_1, 0, 0, 0 ) )
            {
                  // feature is enabled by the license file. Do something...
            }
            if ( FEATURE_STATUS_ENABLED == UstLF::GetFeatureInfo( MY_FEATURE_2, 0, 0, 0 ) )
            {
                  // feature is enabled by the license file. Do something...
            }
      }
      UstLF::ShutDown();
}
```

7. Compile application. Done!

This very primitive (but real) code example shows how to check application features status and take some actions. It is possible to lay out license checking code in a different way using additional **UstLF** functions (see **UstLF API**). Also application must be able to generate a license request file (by making in appropriate place a call to the **UstLF API** function `UstLF::CreateLicRequest(…)`).

After **UstLF** is integrated the rest of the process looks as follows (example for the **node locked** license model):

1. User installs licensed application
2. User starts application for the first time and in some way (that should be provided in the code – by pressing some menu item for example) generates the license request file.
3. User submits this license request file to you.

4. Using license **LicGen.exe** utility (see **License Generator**) you open license request file and generate license file that corresponds to your sales agreement with the user.
5. User places the license file in some predefined location on the target computer (where your application can find it). After this application will act according to the rules specified in the license file.

There can be also another scenario with the license server (example for **floating license** model):

1. User installs licensed application on multiple computers and the license server on a dedicated computer.
2. User generates license request from any computer with installed application and generates license request from the license server.
3. User submits to you application license request file and license server request file.
4. Using license **LicGen.exe** utility (see **License Generator**) you open license request files and generate *floating* license file that corresponds to your sales agreement.
5. User places license file in some folder and points license server to this folder. After this licensed application can be started from any computer on the network but the number of concurrent application instances that will have specified features enabled will be limited to the number specified in the license file (for example: if you specified one floating license user will be able to start licensed application from any computer on the network, but only on one computer at a time application features will be enabled. License “floats”).

There can be also a scenario that is a mix between node locked and floating license models. In this case an application allocates node locked license(s) from the local host and floating license(s) from the license server on the network.

This is a brief overview. Details are described below.

Installed files location

By default all **UstLF** files are installed in the following location (32 bit system):

C:\Program Files\UST Solutions\UstLF

On the 64 bit system:

C:\Program Files (x86)\UST Solutions\UstLF

Feature map file

Obviously since your application has a number of licensed features there is a need to declare them in some way for future references. This is the main purpose of feature map file. This is a regular C/C++ header file and the only file that should be created manually (by typing text). However this is very simple exercise. Let see how this can be done on an example.

Let's pretend that we are creating some kind of a graphic image processing application. In our upcoming release we plan to license the following features:

- Ability to export of different file formats
- Ability to import of different file formats
- Ability to capture video stream
- Ability to perform some special network operations

For this imaginary scenario we can create the following feature map file:

```
---------------------------------- beginning of the file ----------------------------------------------------------
   #ifndef __USTLF_FEATURE_MAP__
   #define __USTLF_FEATURE_MAP__

        #define EnableFileExport        1
        #define EnableFileImport        2
        #define EnableVideoCapture      3
        #define EnableNetwork           4

   #endif
------------------------------------------ end of the file ------------------------------------------------------------
```

There are only couple conditions that should be met:

- The identifier used in `#ifndef...#define...#endif` pre-processor directive must have a predefined name **__USTLF_FEATURE_MAP__** . This signals to the software (**AppDesGen.exe** utility) that this is a feature map block.
- The identifiers of the individual features after each `#define` keyword must differ from each other.
- The values associated with the features (`1,2,3,4` in the sample above) must be numeric and also must be different from each other. Actual values and order does not matter (in the sample above you can have `500,123,2,97` instead of `1,2,3,4` or something else…)

You can define any number of features (but certainly not less than one feature).

You can choose any names for your features in the feature map. The only recommendation that these names should be informative enough since you will see them later in the license generating software (also since this is a C/C++ header file you need to follow certain rules).

You can choose any name for the feature map file.

For different version of your application you may have different versions of the feature map file. In other words file content can be modified when necessary.

Key generator

(ukg.exe)

Key generator provides a starting point for entire application licensing process. The licensing information (the license in fact) that you plan distribute to the customers must be safe and secure. An attempt to alter it in any way must be detectable and altered/invalid license must be automatically discarded by the licensing system. To achieve this UstLF employs several modern encryption and hashing algorithms including asymmetric cryptography algorithm[1].

Utility **ukg.exe** is used to generate asymmetric cryptography keys – both "public" and "private". This is a console application.

```
Usage:

ukg.exe -h
ukg.exe [-d <output directory>]

Parameters:
-h                            Show help.
-d <output directory>              specifies the directory where the key file will be
                              generated. This is an optional parameter - if it is omitted the
                              key file will be generated in the current folder.
Examples:  ukg.exe –d C:\Software\GraphicImageProcessor
           ukg.exe –d "C:\My Other Folder"
```

Key generator creates a single file with default hardcoded name **pub_priv.key**. Each time started generator will create a file with a new random key pair. ***Securely store the generated key file. Do not distribute it!*** It is a key to license generating process.

How many key files do you need? Generally you can use only one key file for all applications that you plan to protect. You may choose to have a separate key file for each product line. You may generate a new key file for each version of your application... However having too many key files will create some problems of managing them (keeping them in organized way, making backups, setting up license generator, something else...). In reality the only reason to generate and use a new key file is a suspicion that your current key file somehow left the secure storage and someone unauthorized might be using it to generate license files without your permission. This should be the only consideration for your decisions.

1. In brief asymmetric cryptography algorithm relies on two special tokens – "public" and "private" keys. "Public" keys normally are freely distributed to the clients. Information is encrypted using "public" key and decrypted using "private" key. Also asymmetric cryptography algorithm is used for digital signing of the information. In this case "private" key is used to calculate digital signature of some information block and "public" key is used to validate this signature and detect whether this information was altered or not. In our process "public" key will be embedded in your application and used to validate license file on the customer side while "private" key will be used by you (***and only by you***) to generate and "digitally" sign the license files.

Application descriptor generator

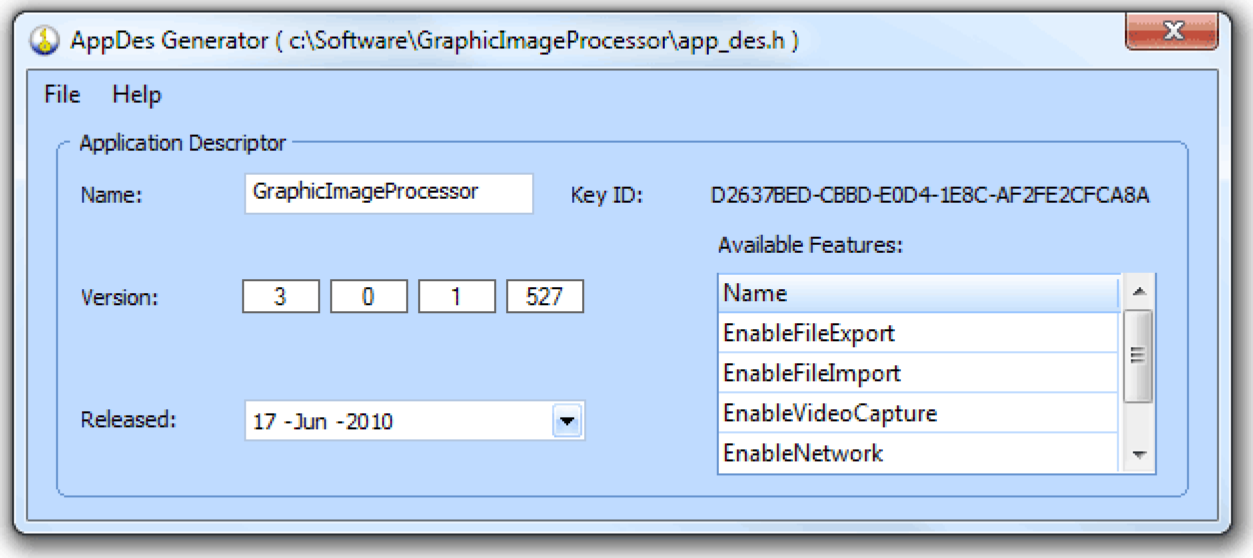
(AppDesGen.exe)

The purpose of this utility is to generate a source code file (application descriptor file) containing some critical data required to securely license and protect your application. First of all this file contains public portion of the asymmetric encryption key file and its ID (ID is provided only for your convenience and future references). Additionally to this application descriptor file contains the following items describing your application:

- Application name
- Application version
- Application release date
- List of licensed features

This information is stored in a secure form (encrypted and signed) and cannot be altered seamlessly. The **AppDesGen.exe** generates application descriptor file in the **current folder** with hardcoded default name: **app_des.h**. You can rename it to any other name you like as long as the content is preserved. ***Never modify application descriptor file content manually!*** Also if renaming - preserve the file extension (application descriptor files are opened and processed by the license generating software as header .**h** files)

The main window of the application descriptor generator:


AppDes Generator (c:\Software\GraphicImageProcessor\app_des.h)
File
Help
Application Descriptor
Available Features:

Name: GraphicImageProcessor

Key ID: D2637BED-CBBD-E0D4-1E8C-AF2FE2CFCA8A

Version: 3 0 1 527

Released: 17 -Jun -2010

Name
EnableFileExport
EnableFileImport
EnableVideoCapture
EnableNetwork

Here it is possible to double check and if necessary modify application name, version and release date. After this application description file should be saved (note: location of the file is shown in the window caption).

On start-up application descriptor generator takes all required information from the command line. If some required information in command line is missing or is invalid a help screen will be displayed.

Here is the **AppDesGen.exe** usage description:

```
Usage:

   AppDesGen.exe -h
   AppDesGen.exe –cfg <filename>
   AppDesGen.exe <Detailed parameters>

Here:

-h                 Show help screen.
-cfg <filename>    Specifies the path to the configuration file with detailed command line

Detailed parameters:
-key <filename>    Specifies the path to the pub_priv.key file. Mandatory parameter.
-map <filename>    Specifies the path to the feature map (.h) file. Mandatory parameter.
-name <appname>    Specifies application name ( Example: -name GraphicImageProcessor ). This
                   parameter is mandatory only if parameter –rc <..> is not used. Otherwise
                   it is optional. It can be used to overwrite the application name
                   retrieved from the resource file ( see below ).
-rc <filename>     Specifies the path of the resource ( .RC ) file. This is a regular MS
                   Visual Studio generated resource file. It is used to extract information
                   about application name and version.
                   This parameter is mandatory if parameter –name <...> is not used.
                   Otherwise it is optional.
-inc <directory>   Specifies the directory with include files referenced in the resource
                   file. This is only necessary if application name and version in the .RC
                   file are referring to some other include file potentially located in
                   different folder. If this is not the case there is no need to use this
                   parameter.
-nogui             Suppresses GUI. This can be used if AppDesGen.exe is activated from the
                   Post-Build Event of the MS Visual Studio project. In this case some
                   execution information will be logged to the MS Visual Studio Output
                   window.
```

As you can see it is possible to list all parameters in the command line or to put them all info the configuration file and use the first form (with `-cfg<...>` parameter). Configuration file can have any name. The content of the configuration file for our imaginary graphic image processing application could be as follows (note: line breaks after each parameter are allowed):

--------------- beginning of the configuration file ---

```
-key pub_priv.key
-map feature_map.h
-name GraphicImageProcessor
-rc GraphicImageProcessor.rc
-nogui
```

------------------------------- end of the file --

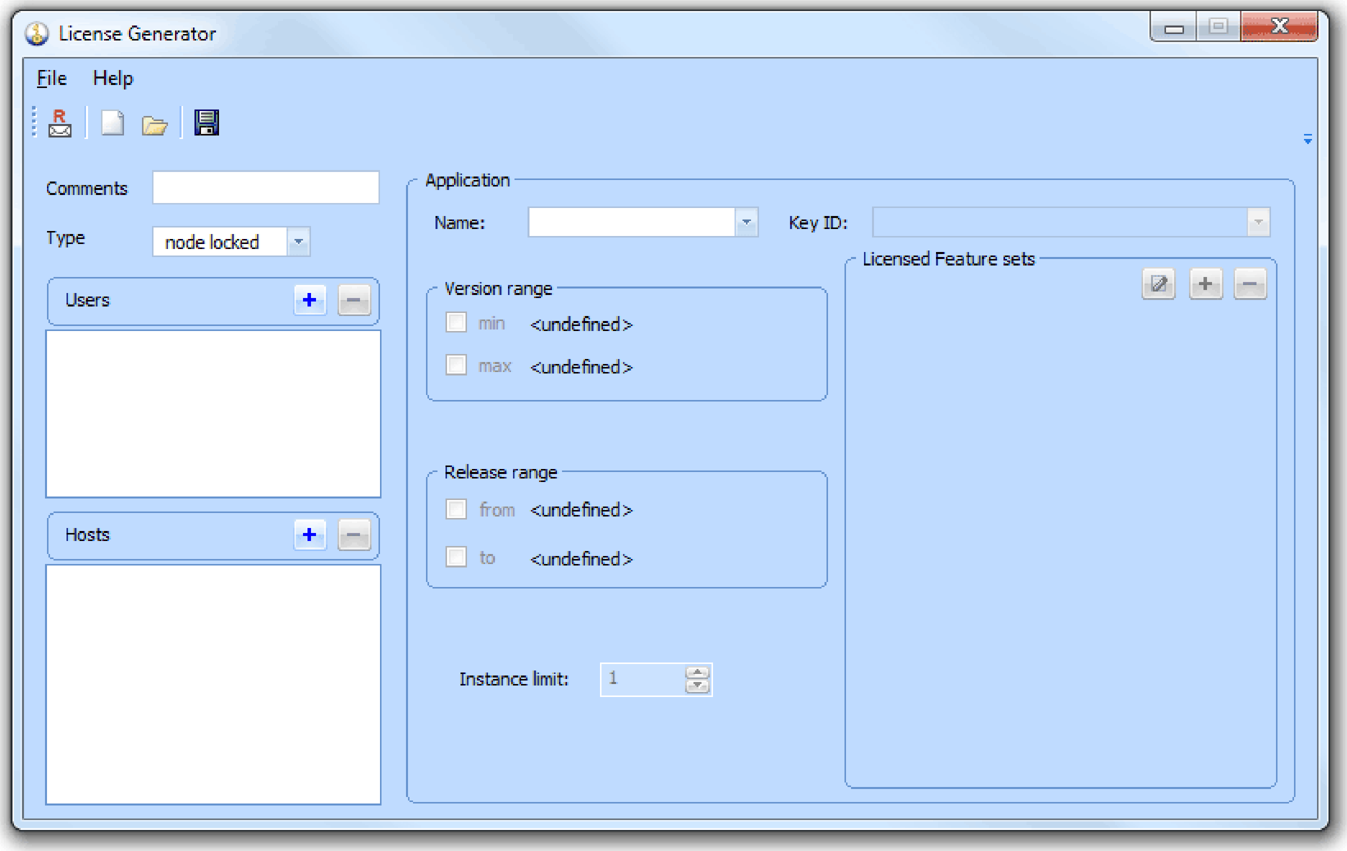
Example: `AppDesGen.exe -cfg appdes.cfg`

Application description file should be regenerated each time when one of the inputs (application name, version, release date, list of features) is changed. This can be automated by starting **AppDesGen.exe** with the configuration file in command line from the Pre-Build event in the MS Visual Studio project.

You may regenerate application descriptor as many times as you need during application development. But as soon as you release the new version of your application to public - store corresponding application descriptor in some repository folder. License generator will need descriptors for released application versions. To avoid name collisions (since you will most likely release multiple versions of your application) you may want to rename application descriptor before storing it. Any file name may be chosen (for example build up the name in the way it reflects released application version like: **MyApp_app_des_ver_1_2_345.h**). But keep the file extension (**.h**) - license generator will be processing all files with this extension from the application descriptor repository folder.

License Generator
(LicGen.exe)

This is the tool for generating license files.
The main screen after start up:



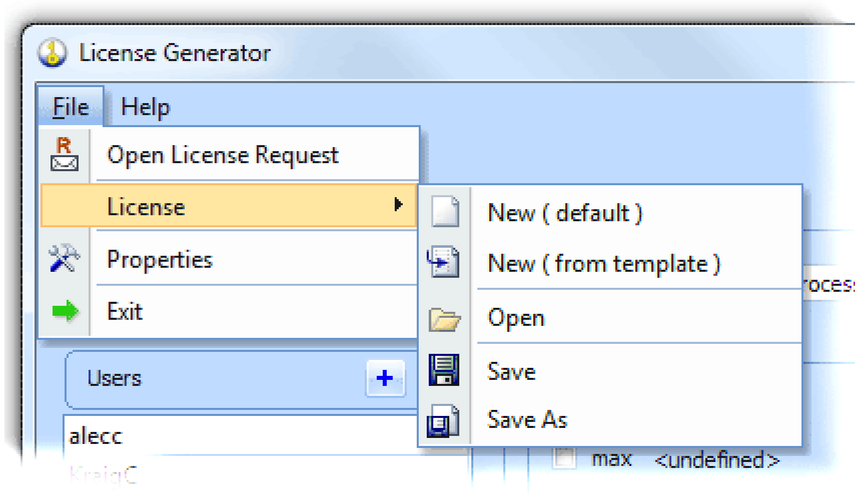
The sequence that should be used to create license files is as follows:

1. Open license request file
2. Set license type and optionally put comments
3. Add user from license request (optional)
4. Add target host(s) from license request (optional but highly recommended)
5. Set application valid version range (may be left undefined)
6. Set application valid release range(may be left undefined)
7. Set application instance count (max allowed concurrent instances)
8. Define and populate feature sets that will be valid during certain periods of time
9. Save license file

Before using license generator it should be properly configured (see **Properties** below).

Menu and toolbar

Menu offers all required options to initially set up and control license generator. Here is the expanded menu:



Item "Open License Request". Licenses in **UstLF** are generated for specified protected application that is executed on a specified computer (specified by the customer). Another option is a license generated for license server (that also runs on a computer specified by the customer). As result the starting point of the license generation is the knowledge of target application's specifics and specifics of computer(s) where this application will be executed or where license server is running. All this information contains **license request** (**UstLF** library provides functionality to easily generate this requests. This will be reviewed in **API** section).

Item License – sub item "New (default)". This is self explanatory – create new license file.

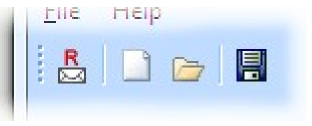
Item License – sub item "New (from template)". License generator allows to use once generated licenses as a templates for a new licenses. After this menu item is selected a **License Template Dialog** will pop up where appropriate template can be chosen (this dialog will be reviewed below).

Items "Open", "Save" and "Save As" are self-explanatory.

Item "Properties" . This item pops up a **Properties** dialog where several necessary entries should be filled up in order to establish correct behavior of the license generator (this dialog will be reviewed below).

Item "Exit" – self-explanatory.

License generator has only one tool bar with several buttons corresponding to some menu items for quick access:

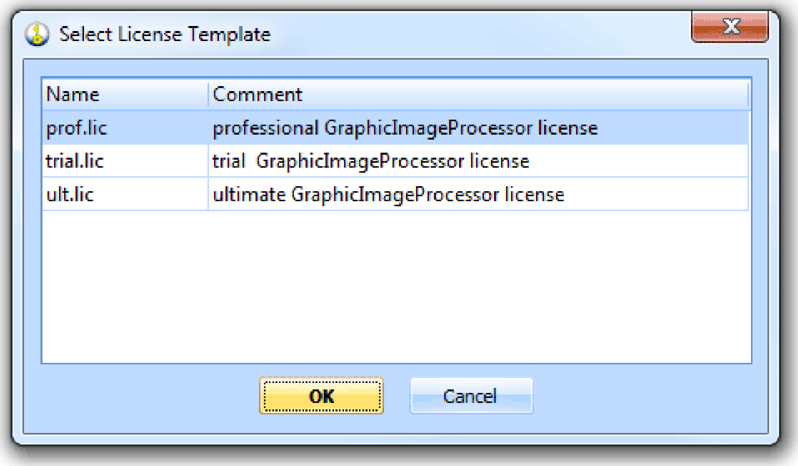


Pop-Up Dialogs

There are only two pop-up dialogs that require some comments: **License Template Dialog** and **Properties** dialog.

License template dialog

This dialog pops up when menu item **"New (from template)"** is selected.



Select License Template

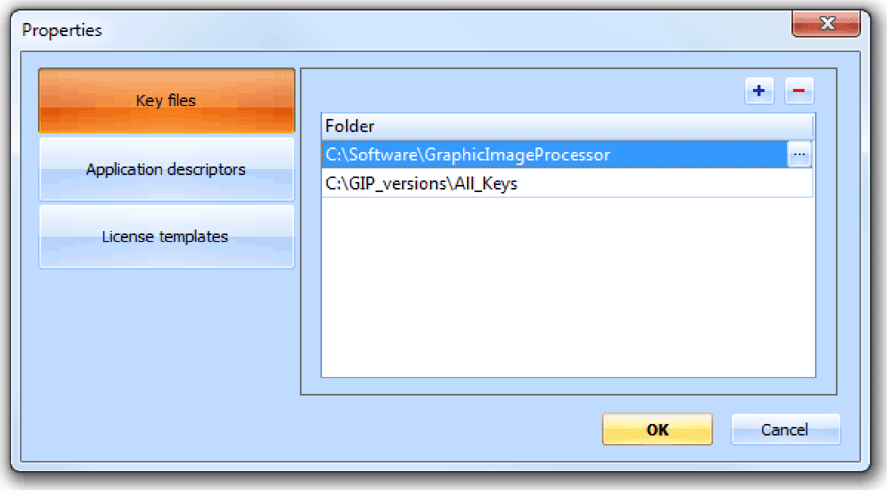
Name	Comment
prof.lic	professional GraphicImageProcessor license
trial.lic	trial GraphicImageProcessor license
ult.lic	ultimate GraphicImageProcessor license

Dialog shows the list of license files that are located in special "templates" folder (see **Properties** dialog below). These are regular license files that were generated sometime earlier and copied to the "templates" folder. When required template is selected and **OK** button is pressed the content of this template/license file populates all fields of the license generator main window. At the same time the values of calendar dates are adjusted so that the license will start from the current date. In other words it pre-sets the new license from the selected old one. This is very useful especially if your application has lots of licensed features and license that you create have more than one **feature set** (this will be explained below).

Properties dialog

This dialog pops up when user selects menu item "**Properties**". Dialog has 3 pages.

Page "Key files"

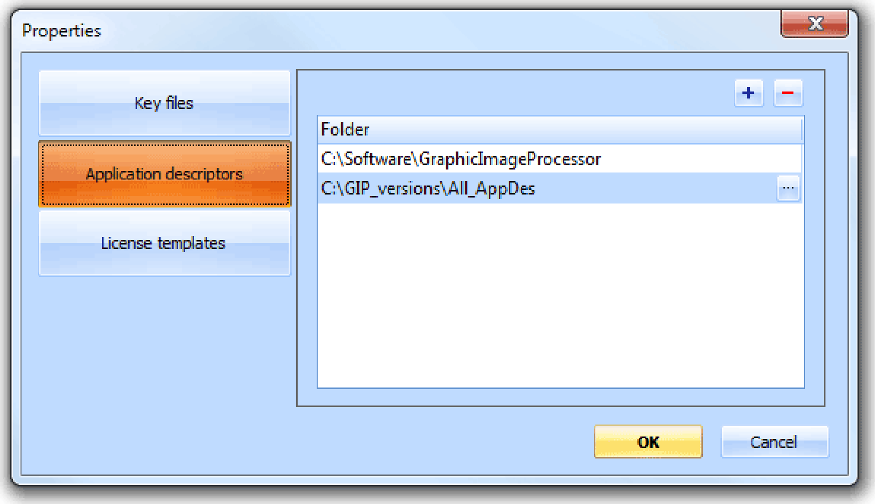


Properties

Folder
C:\Software\GraphicImageProcessor
C:\GIP_versions\All_Keys

As it was mentioned above license generator needs the asymmetric key generated by the **key generator** and used in the process of embedding **UstLF** in the target application. On this dialog page it is possible to define one or more locations where these key files are stored. It does not matter whether these keys were used for one or may different applications – locations of keys for all applications that require licenses should be listed here. These locations may contain other types of files – license generator will pick only files with extension **.key** and valid content. ***Note: it is extremely important to set correct locations with valid key files – without this you will not be able to open license request files and/or create licenses.***

Page "Application descriptors"



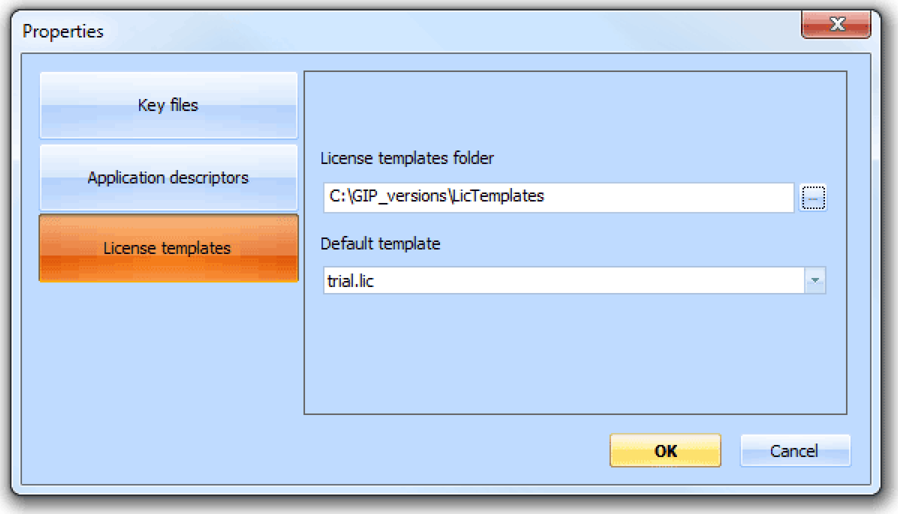
Folder
C:\Software\GraphicImageProcessor
C:\GIP_versions\All_AppDes

Technically it is possible to generate license using only license request file and appropriate asymmetric key file. In many cases this will be enough. However this puts some unnecessary restrictions both on you and on the user.

Imagine the following scenario. Let's say you have some application version 1.0 with features **A** and **B**. Now you release version 2.0 with old features **A**, **B** and a new feature **C**. After this point in time user requests the license for the application **version 1.0** and you will generate it for him with features **A** and **B** enabled. Now user downloads a new release from your web site and discovers new feature **C**. Your support policy makes him eligible for the new stuff. So user again will have to request license from you and you will have to reissue it. This primitive scenario shows how redundant activity is created. In reality there are cases when application features are changed frequently, users move back and forth between application versions and this creates lots of unnecessary activities. For the above imaginary scenario the solution would be to issue a license with features **A, B** and **C** that will work for application version 1.0 and version 2.0. To achieve this it is necessary to have access to combined feature list of all versions of a given application. To have this combined feature list and to make selections on it is important in one more case – when user requests license without submitting license request file (for example providing some required information by phone – this is also possible). In this case there must be a source of information that license generator will use (instead of relying only on the license request).

So, to summarize – on this page you might want to list all locations with application descriptors files used in your software. This will allow you to observe and select (if necessary) application features from all versions. These folders may contain different file types – license generator will consume files only with valid application descriptors. ***Note: application descriptors will be considered valid only if the corresponding asymmetric key file exists in one of the folders on page "Keys files"***.

Page "License templates"



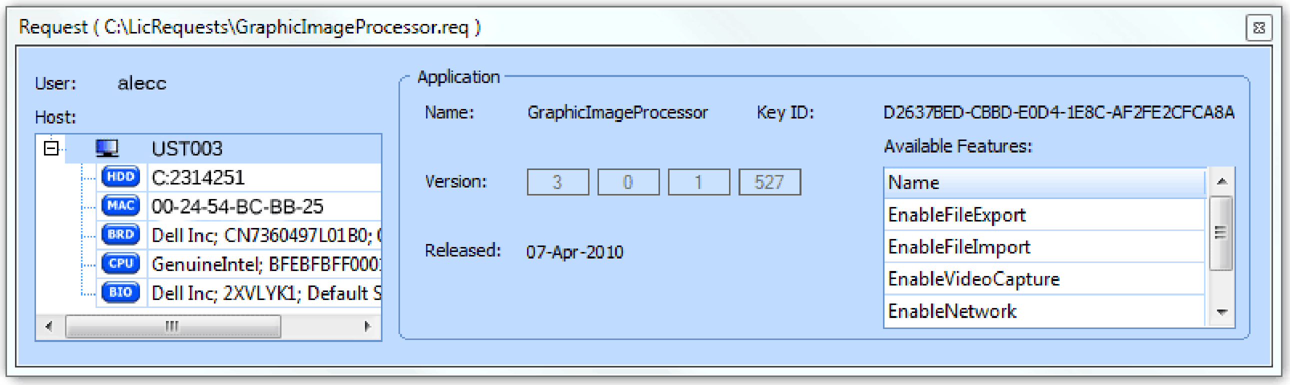
License templates folder: C:\GIP_versions\LicTemplates

Default template: trial.lic

As it was mentioned above it is possible to create a library from previously generated licenses and use it to preset the new license that you are about to create.
License templates folder identifies location of these template license files.
Default Template field identifies template that will be used when menu item **"New (default)"** is selected.

License request window

This window gets opened in response to **"Open license request"** command from the menu or the toolbar.



Request (C:\LicRequests\GraphicImageProcessor.req)

User: alecc

Host:

```
UST003
  HDD C:2314251
  MAC 00-24-54-BC-BB-25
  BRD Dell Inc; CN7360497L01B0; (
  CPU GenuineIntel; BFEBFBFF000
  BIO Dell Inc; 2XVLYK1; Default S
```

Application

Name: GraphicImageProcessor

Key ID: D2637BED-CBBD-E0D4-1E8C-AF2FE2CFCA8A

Version: 3 0 1 527

Released: 07-Apr-2010

Available Features:

Name
EnableFileExport
EnableFileImport
EnableVideoCapture
EnableNetwork

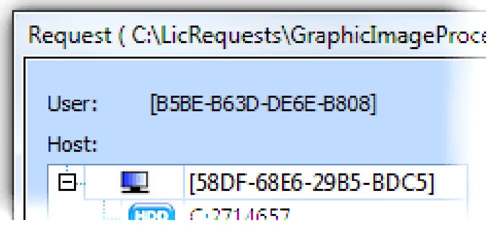
Information in this window contains user login name, computer name where application was started, a list of computer hardware parameters and specific application details.

UstLF collects information about the following hardware parameters:

HDD	– hard drive serial number
MAC	– MAC address
BRD	– mother-board related information
CPU	– processor related information
BIO	– BIOS related information

Specific application details contain application name, version, release date, list of features and a cryptography key ID (ID of the public key embedded into the application and used to encode license request file).

Information in the **User** and **Hosts** boxes may be presented as a clear and meaningful text or as a clear but meaningless text surrounded by brackets:



Request (C:\LicRequests\GraphicImageProc

User: [B5BE-B63D-DE6E-B808]

Host:

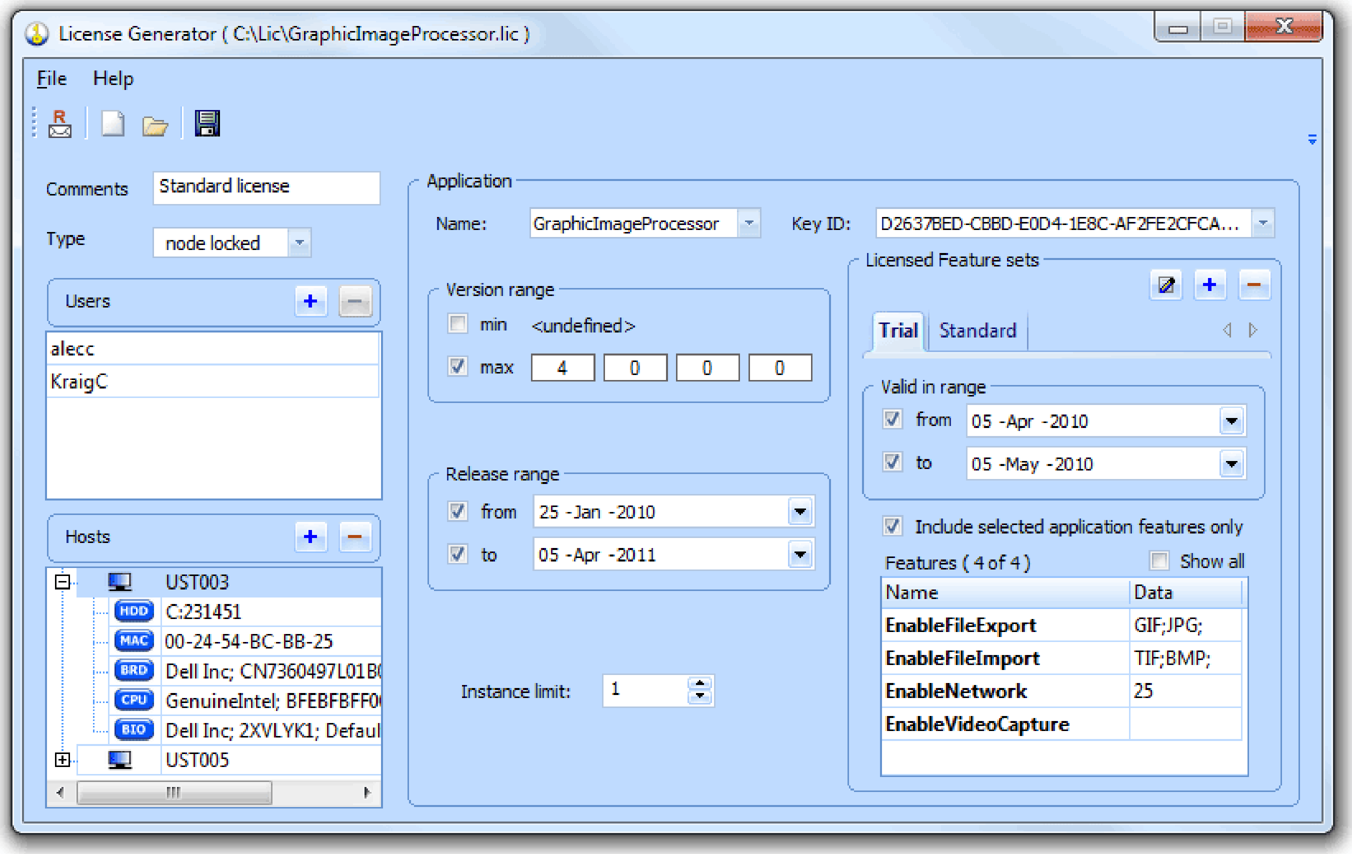
[58DF-68E6-29B5-BDC5]

The second form (meaningless text surrounded by brackets) is a result of selecting particular options when **UstLF** is embedded in the application (see **API** description). This form can be chosen to add extra privacy to the collected data when request is generated. However extra privacy normally is not required for hardware related information.

Note, that if key file that was used to create application descriptor is missing in the keys folder (**see Page "Key files"**) then it is not possible to open license request and you will get an error **"Failed to load request file".**

Main window

After populating main license generator window with data it might look as follows:



License Generator (C:\Lic\GraphicImageProcessor.lic)

Comments: Standard license
Type: node locked

Users
- alecc
- KraigC

Hosts
- UST003
  - HDD C:231451
  - MAC 00-24-54-BC-BB-25
  - BRD Dell Inc; CN7360497L01B
  - CPU GenuineIntel; BFEBFBFF0
  - BIO Dell Inc; 2XVLYK1; Defaul
- UST005

Application
Name: GraphicImageProcessor
Key ID: D2637BED-CBBD-E0D4-1E8C-AF2FE2CFCA...

Version range
- [ ] min <undefined>
- [✓] max 4 0 0 0

Release range
- [✓] from 25 -Jan -2010
- [✓] to 05 -Apr -2011

Instance limit: 1

Licensed Feature sets: Trial | Standard

Valid in range
- [✓] from 05 -Apr -2010
- [✓] to 05 -May -2010

[✓] Include selected application features only

Features (4 of 4) [ ] Show all

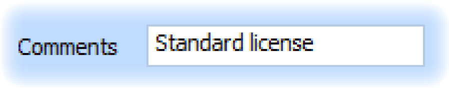
Name	Data
EnableFileExport	GIF;JPG;
EnableFileImport	TIF;BMP;
EnableNetwork	25
EnableVideoCapture	

The main window has two sections: general section and Application section.

General section

This section contains following elements and element groups:

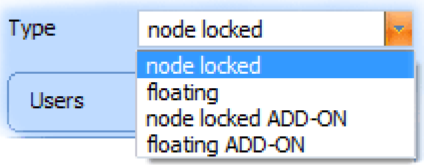
Comments



Comments: Standard license

In this field any license related comments can be entered. UstLF does not process them in any way. This is for your reference only.

Type



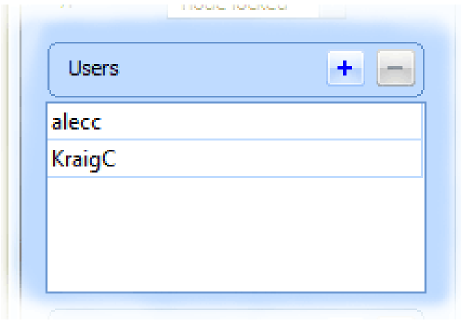
This drop list box should be used to select one of available license types.

The **node locked** license enables execution (or application features) on a given computer(s) identified by some hardware specific information. This can be accompanied with the user(s) information (user login).

Floating license (placed on the license server) enables execution (or application features) on any computer on the network. In this case license server controls the number of concurrent licensed application instances.

The **node locked ADD-ON** and **floating ADD-ON** are similar to **node locked** and **floating** with one exception: licenses with these types bam be mixed with previously issued licenses (for details see "**Mixing Licenses**" below).

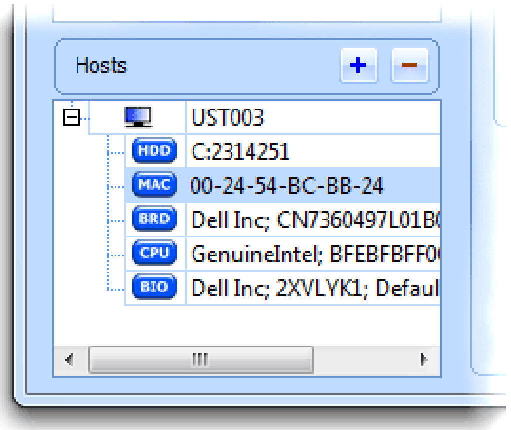
Users



This group of elements allows to specify one or more users that will be using **licensed application**. ***If users are not specified then the license will be valid for any user***. User can be specified for node locked and for floating licenses. Users can be added or removed using + or − buttons. If **License request window** is opened then user from request will be added otherwise you will have to type in user name manually. Note, that there is no reason to add user that is logged on to the license server unless this user will be using application from some other computer on the network.

In majority of the cases with node locked licenses there is no need to specify users. This makes sense to do only sometimes in cases with floating licenses.

Hosts



This group of elements allows to specify one or more hosts where generated license will be valid. ***If hosts are not specified then license will be valid on any computer*** (this is not recommended unless you are creating some kind of a demo license that should be valid everywhere for a short period of time). Hosts can be specified for node locked and for floating licenses. **In case of floating license host is the computer where the license server will be running**. Hosts can be added or removed using + or − buttons. If License request window is opened then the host from request will be added otherwise you will have to type in host name and host parameters manually.

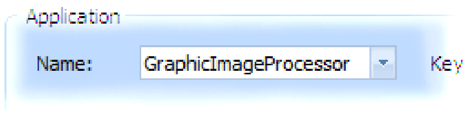
Each host entry consists of the main entry representing the host name and a set of specific hardware related parameters (see **License Request window**). All parameters are optional and may be removed (blank line). If parameter is removed it will not be checked by the **UstLF** system.

Each hardware parameter has its own “weight”. When **UstLF** system checks the host for validity it takes into account weights of matching and not matching parameters. If the total weight of matching parameters complies with the predefined tolerance then the host considered to be valid. This approach allows user to reasonable modify (upgrade) the host hardware without necessity to reissue license for the new hardware configuration.

Application section

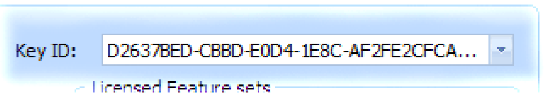
All application related details are grouped in this section.

Application name

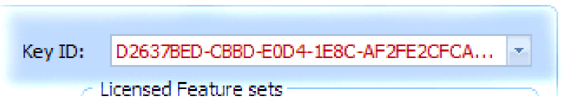


This drop list box allows to select application you create license for (note: license request that you opened may not correspond to the license that you are about to create – for example you can use license request from some other application to retrieve just user name and/or parameters of the target computer). The list is populated from the application descriptor files residing in referenced folders (see **Page “Application descriptors”**). If application descriptor folders are not specified you will have only one application name coming from license request.

Key ID

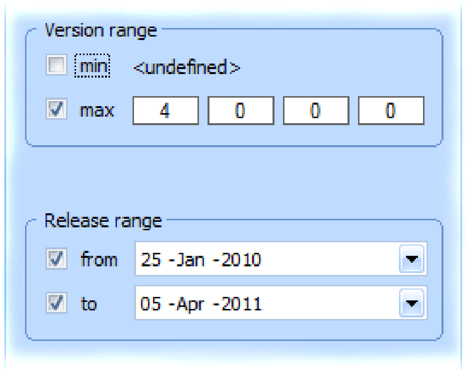


This drop list box allows to select encryption key that was used for application descriptor. The list is populated from the key files residing in referenced folders (see **Page “Key files”**). Normally license generator will automatically select key that corresponds to the license request. If key cannot be found the value will be displayed in **red**:



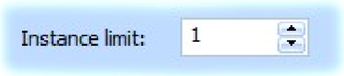
In this case you will not be able to generate license file. If a wrong key is used (does not match license request) then license will be generated however will not work – application will ignore it.

Version range and Release Range



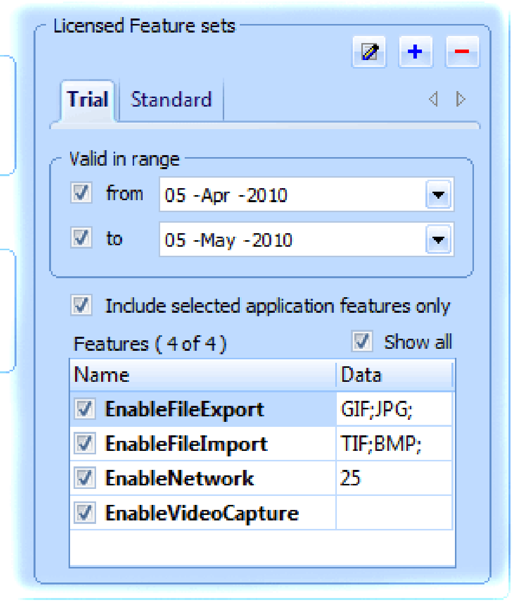
Here you specify target application boundaries for which license will be valid. All parameters are optional (if parameter is **undefined** then any value will be accepted). Issuing licenses with specified version range and/or release range will allow your customer to use certain application versions without necessity to regenerate the license file. For example if your customer purchased a license today, say 05-Apr-2010 and you are providing him with 1 year of free upgrades then simply set the end of release range to 05-Apr-2011 – license will be valid for all application versions released within this period. Note that after the end date that you specified license will not stop working – it will work as usual with application versions that were released within specified period. Also note, that **"min"** and **"from"** values must not be greater than the **version and release date of the application** that you license.

Instance limit


Instance limit: 1

This allows you to specify the number of concurrent licensed applications running on a single computer (in case of a **node locked** license) or on a network (in case of the **floating** license). In case when your application can access multiple valid licenses **UstLF** framework will try to allocate one instance from each valid license file. This means that if you issued a license file with instance counter equal to one and after this issued identical license (note : this is not a file copy process) and place it in the same license folder, a single application with **UstLF** will allocate both of them. Do not expect that you will be able to launch 2 applications each of which will allocate one license file. The rule: ***Application with UstLF will try to allocate one instance from each valid license file.***

Licensed Feature Sets


Licensed Feature sets

Trial | Standard

Valid in range
- ✓ from 05 -Apr -2010
- ✓ to 05 -May -2010

✓ Include selected application features only

Features (4 of 4) ✓ Show all

Name	Data
✓ **EnableFileExport**	GIF;JPG;
✓ **EnableFileImport**	TIF;BMP;
✓ **EnableNetwork**	25
✓ **EnableVideoCapture**	

This group allows to specify feature sets with required parameters.

Feature sets are added or removed using + and − buttons. To rename feature set use ✎ button. License must contain at list one feature set. Each feature set can have a name. It is only for your convenience and do not affect license functioning in any way.

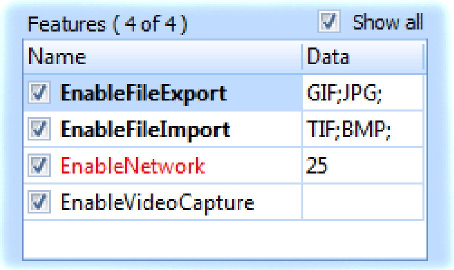
Each feature set can be associated with a certain time period. This is done by specifying values in the **"Valid in range"** section. During this period license will refer to the selected features in the "**Features**" table. If one of the period values is undefined then feature set in not limited from the corresponding side. It is allowed to have both period valued undefined. If more than one feature set is defined then feature set ranges must not overlap.

Each feature set must have at list one feature associated with it. Features are selected by placing a check mark in the **"Features"** table. Note, that feature table will contain a **combined array** of features from all versions of your application (License generator will examine all application descriptors that you specified. See **Page "Application descriptors"**). An alternative way of selecting ***all existing and possible*** application features is to uncheck the check box **"Include selected application features only".**

Each feature can have some data associated with it. Data is a plain text that should be typed into the **Data** column. **UstLF** does not make any assumptions about feature data. Data is treated as a regular text and can be retrieved using appropriate **API** function. It is your responsibility to define individual features data and process this data in the application. The same license can have different data in different feature sets.

Last element **"Show all"** will just toggle the **"Features"** table between listing all (selected and not) application features and listing only selected features. This is for your convenience only.

If you have several version of your application and each version have slightly different features then you might see some color coding in the "**Features**" table (certainly the **Page "Application descriptors"** should be properly initialized). For example:



Features (4 of 4) ✓ Show all

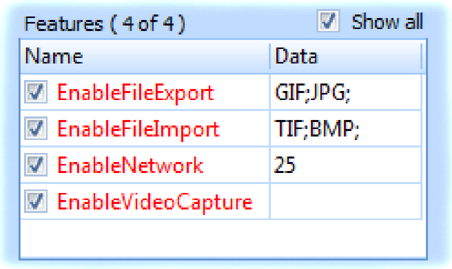
Name	Data
✓ **EnableFileExport**	GIF;JPG;
✓ **EnableFileImport**	TIF;BMP;
✓ EnableNetwork	25
✓ EnableVideoCapture	

This is a response to the fact that some or all applications that fall under the **Version range** and **Release Range** do not have features listed in the table. So, in this case:

- If feature name is printed with "**black bold**" font - this means that all application versions within defined **Version range** and **Release Range** have this feature.
- If feature name is printed with "black regular" font – this means that some applications within defined range have and some applications do not have this feature.
- If feature name is printed with "red regular" font – this means that none of applications within defined range have this feature.

So, in other words this color coding indicates whether a feature exists fully, partially or does not exist at all in application versions within the specified **Version range** and **Release Range**. Selecting feature printed in red makes sense only if you think that you did not place all application descriptors in specified folders (the **Page "Application descriptors"**) and user might have an application version with this feature.

There can be a case when you don't have any references to application descriptors (the **Page "Application descriptors"** is not set) but key file(s) exist and you opened existing license file. In this case license generator cannot make any assumptions about release date and version of your application so feature names will be printed in with "red regular" font. For example:



Features (4 of 4) ✓ Show all

Name	Data
✓ EnableFileExport	GIF;JPG;
✓ EnableFileImport	TIF;BMP;
✓ EnableNetwork	25
✓ EnableVideoCapture	

You will be able however to modify this license file and save it.

Mixing licenses

If application has more than one licensed features then users once in a while tend to upgrade their licenses and enable some feature(s) that were previously disabled. There are two ways of doing this:

1. Issue a fixed license (**node locked** or **floating**) that will have all previously enabled features (like an old license) and additionally to this new features requested by user. In this case old license is not needed any more and user can safely remove it from his system. New license will have all requested stuff.
2. Issue a complementary license (**node locked ADD-ON** or **floating ADD-ON**) that will enable only newly requested features. In this case user will keep the old license file and add the new one. **UstLF** will process both licenses, internally merge them and create combined set of enabled features.

The second case is quite common due to many reasons.

License merging is based on two rules:

1. If at list one of two licenses has some feature enabled – then **UstLF** will enable this feature.
2. If two licenses have some feature enabled – then **UstLF** will associate with this feature data from the license that was issued last (the newest license. Note, that each license has a time stamp with the date and time when it was issued. For example of you have an old license **A** with a feature **EnableFileExport** and data **"BMP"** and newly issued license **B** with the same feature but with feature data **"GIF;JPG"** then in result feature **EnableFileExport** will have associated data **"GIF;JPG"**. Note that the data from the license A feature is not used).

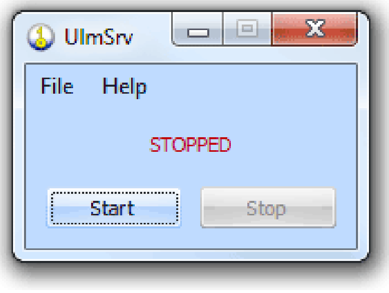
These rules are applicable not only for two but for any number of licenses. Also it does not matter what is the type of the licenses (**node-locked ADD-ON** or **floating ADD-ON**). As result it is possible to make all sorts of mixes – for example issue a floating license and later on issue a complimentary node-locked license with some additional features enabled. Or issue a license that will overwrite just feature data for a given feature. There can be many scenarios, but for your own benefit we recommend to follow some simple and logical licensing pattern.

Important thing to understand – fixed licenses (**node locked** or **floating,** not … ADD-ONs) will overwrite each other in relation to the issue date-time. **UstLF** system will allocate all valid licenses but will get features and feature data **only from the most resent** fixed license and all …ADD-ON licenses that were issued **after** this fixed license. Features and feature data from **all** licenses issued before will be ignored. So, in other words each issued fixed license sets some sort of a reference point - everything before it is ignored, everything after it is consumed. It is like physical deletion of all license files issued before most resent fixed license.

Note: it is not necessary to issue fixed licenses. You may provide customer only with …ADD-ON licenses. They will be merged together according to the rules above. This really depends on the policy you choose and current situation.

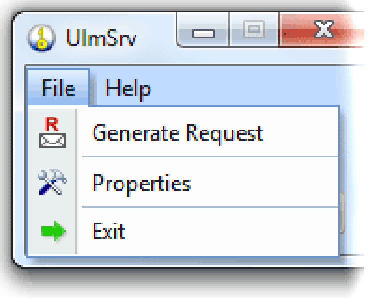
License Server

License server (**UlmSrv.exe**) is an application used to manage and access floating licenses on the network. Here is the main screen of the license server:



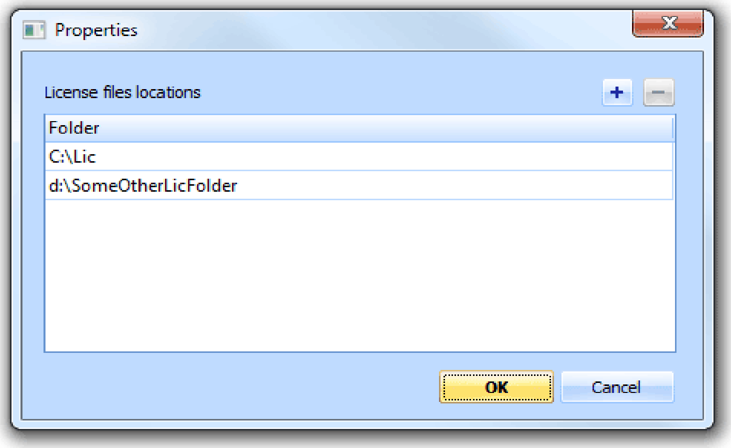
Minimized window parks itself in the task bar.

Server UI has simple controls accessible through the main menu:



Menu item **"Generate Request"** is used to create a license request file. This license request can be opened in the license generator (see **LicGen.exe**) however information from it that should be used (in the license) is information related to server host. Including host in the license file makes this license valid only on this host.

Before starting license server should be configured by specifying license locations (license folders). This is done by selecting menu item **"Properties"**. Here is the Properties window:



Folder
C:\Lic
d:\SomeOtherLicFolder

Note, that without this you will not be able to start license server.

One running license server can serve floating licenses for multiple **UstLF** protected applications. There can be one or more license servers started on the same network however each license server must be started on its own host. License server requires communication DLL **ulmc.dll** that must be placed in the same folder with **UlmSrv.exe**

In communication between license server and client application TCP and Multicast sockets are used. Used IP addresses and port may be changed if necessary.

Some additional server tuning may be done by modifying registry :

```
   HKEY_LOCAL_MACHINE\SOFTWARE\UST\UstLF\UlmSrv
```

Double word value “Options” is described in **API** (see function `SetOptions(...)`)

String encryption utility

(sfe.exe)

Sometimes there is a need to use in an application strings that carry specific and secure information. These strings should be available only at certain periods at runtime and should not be visible or editable in the binary (exe or dll) file directly. Utility **sfe.exe** provides functionality to create encrypted strings that can be decrypted at runtime using API function `DecriptEmbeddedStringA(...)`.

```
Usage:
sfe.exe –s <filename> -k <filename> [-d <output directory>]

Parameters:
-s <filename>             specifies the path to the source code file with plain strings.
-k <filename>             specifies the path to the key file (see Key generator) that
                          will be used to encrypt strings.
-d <output directory>          specifies the directory where the encrypted strings files
                          will be generated. This is an optional parameter - if it is
                          omitted the generated file will be placed in the folder with
                          the input file source.

Example:   sfe.exe –d C:\MyFolder\estr.str –key C:\MyFolder\pub_priv.key
```

Source file with strings that should be encrypted can have any name. Here is an example of such file:

--------------------------------- beginning of the file ---

```
#ifndef __STR_SOURCE_H__
#define __STR_SOURCE_H__

     /* Strings to be encrypted: */
     CHAR * sFirstEncriptedStrA = "This is my encripted string of CHARs";
     WCHAR * sSecondEncriptedStrW = L"This is my encripted string of WCHARs";

#endif
```

-- end of the file ---

You can have as many different string declarations as you like. Declarations should follow regular C/C++ conventions, however empty/blank strings are not acceptable. After you run **sfe.exe** you will get 2 files (.h and .cpp). Include these files in your project. After this you will be able to decrypt these strings during runtime in the following way:

```
     int nResultStrLenInTChar = 0;
     CHAR sCharBuf[100] = {0};
     WCHAR sWCharBuf[100] = {0};

     BOOL bDecript = FALSE;
     bDecript = UstLF::DecriptEmbeddedStringA( UstLF::sFirstEncriptedStrA,
                                              sCharBuf, 100, &nResultStrLenInTChar);
     bDecript = UstLF::DecriptEmbeddedStringW(UstLF::sSecondEncriptedStrW,
                                             sWCharBuf, 100, &nResultStrLenInTChar);
```

UstLF API

Library API is presented in the header file **ustlf_api.h**. Additional file **ustlf_def.h** contains value definitions (regular #define... statements) for functions input parameters and return values. Here are function descriptions.

```
BOOL CALL_CONV StartUp();
```

Starts the **UstLF** subsystem.

Returns

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters: none.

Remarks

This is the first function that should be called in order to use **UstLF** library. Normally it should be called once when application starts up.

```
BOOL CALL_CONV ShutDown();
```

Stops the **UstLF** subsystem.

Returns

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters: none.

Remarks

This function stops **UstLF** subsystem and releases all allocated resources. Normally it should be called once when application shuts down. It is critical to call this function because it will automatically release all allocated licenses.

```
BOOL CALL_CONV SetEmbeddedData( const UCHAR * pCryptoKeyData,const UCHAR * pAppDesData );
```

Initializes UstLF library with application dependant data containing in the application descriptor file (see **Application Descriptor generator**).

Returns

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters:

`const UCHAR * pCryptoKeyData`

Pointer to the public key defined in the application descriptor file. Normally it is the variable with the name `__aULMPubKey__`.

`const UCHAR * pAppDesData`

Pointer to the application descriptor data defined in the application descriptor file. Normally it is the variable with the name **`__aEncAppDescriptor`**.

Remarks

This function should be called only once after UstLF subsystem is started (see `StartUp()`) and before any other API call. Example:

```
SetEmbeddedData( __aULMPubKey__, __aEncAppDescriptor );
```

```
int CALL_CONV SetOptions( int nNewOptionsBitMask );
```

Sets different option flags allowing to tune the functioning of the UstLF library.

Returns

combined bitmap of current option flags (before new options are applied).

Parameters:

`int nNewOptionsBitMask`

combined bitmap of desired new option flags.

Remarks

This function takes and returns a bitmap of flags based on the following tables:

License request flags table

OPT_LREQ_HOST_NAME_MASK	Hide computer name
OPT_LREQ_USER_NAME_MASK	Hide user name
OPT_LREQ_HD_MASK	Hide hard drive serial number
OPT_LREQ_MAC_MASK	Hide MAC address
OPT_LREQ_BOARD_MASK	Hide computer board related info
OPT_LREQ_PROC_MASK	Hide computer processor info
OPT_LREQ_BIOS_MASK	Hide BIOS info

Flags in this table dictate how information will be pre-processed before it will be stored in the license request. If none of these flags are set then computer related information will be in its original form meaningful for a human (for example user name will be exactly as on the Windows Log In screen). If any of these flags are set then corresponding information will be pre-processed to a special hashed form so that it is readable however meaningless for a human. Use this flags to control the level of privacy that you provide to the users.

Additional clarification: information in the license request file in encoded any way, but when the file if opened in the **License Generator** then the fields of the license request will be visible to you in original or hashed form according to the applied flags.

Behavior flags table

OPT_BH_WATCH_LOCAL_FOLDERS	Enable UstLF automatically watch for changes in folder(s) with licenses
OPT_BH_AUTO_LIC_REALLOC	Enables automatic license reallocation when a change in a license pool is detected and when current date changes (at midnight). Note: change in a license pool can be a result of a change in local license folders and a change on the license server(s) side (if licenses are allocated from license server)

Event flags table

`OPT_EV_LIC_POOL_CHANGED`	Enables firing of an event/call-back when a change in a license pool is detected
`OPT_EV_CONNECTION_OPENED`	Enables firing of an event/call-back when a network connection with the license server is opened
`OPT_EV_CONNECTION_CLOSED`	Enables firing of an event/call-back when a network connection with the license server is closed
`OPT_EV_CURRENT_DATE_CHANGED`	Enables firing of an event/call-back at midnight when license features might change status
`OPT_EV_FEATURE_SET_CHANGED`	Enables firing of an event/call-back when combined feature set is changed.

See `MessageCallbackFn(...)` for detailed events description.

Default options on UstLF start up are :

```
OPT_EV_LIC_POOL_CHANGED |
OPT_EV_CONNECTION_OPENED |
OPT_EV_CONNECTION_CLOSED |
OPT_EV_FEATURE_SET_CHANGED |
OPT_BH_WATCH_LOCAL_FOLDERS |
OPT_EV_CURRENT_DATE_CHANGED
```

(all events are enabled and **UstLF** will watch local license folders for changes)

```
int CALL_CONV GetOptions();
```

Returns the bitmap with current option flags.

Returns

option flags bitmap

Parameters: none.

Remarks

Option flags define the functioning of the UstLF library. All available flags are grouped in tables: License request flags, Behavior flags, Event flags. See description of the function `SetOptions(...)`.

```
BOOL CALL_CONV CreateLicRequest( LPCTSTR szDstFilePath,
                                 LPTSTR szErrorBuffer,
                                 int nErrorBifSizeInTChar );
```

Creates license request file.

Returns

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters:

`LPCTSTR szDstFilePath`

Destination license request file path. Zero-terminated string.

`LPTSTR szErrorBuffer`

User allocated error buffer. Can be NULL

`int nErrorBifSizeInTChar`

Size of the error buffer in characters. Can be 0.

Remarks

Function will create a license request file in the specified destination (`szDstFilePath`). In case of error will store zero-terminated error string in the error buffer.

```
BOOL CALL_CONV GetLicRequestData( void * pReqDataBuffer,
                                  int nReqDataBufLenInBytes,
                                  int * pnReqDataLenInBytes )
```

Creates license request data and stores it into the user defined buffer.

Returns

`TRUE`	if function call succeeded,
`FALSE`	if failed.

Parameters:

`void * pReqDataBuffer`

Destination buffer where license request data will be stored.

`int nReqDataBufLenInBytes`

Size of the destination buffer in bytes.

`int * pnReqDataLenInBytes`

Pointer to the integer variable to store the actual size of the data (length of the data string) associated with the feature. Value in the variable is updated on function return. Can be NULL.

Remarks

User is responsible for allocating and releasing destination buffer. The minimal size of the buffer can be calculated by calling this function with buffer length set to 0. In this case variable pointed by `pnReqDataLenInBytes` will receive the length of the license request data.

```
void CALL_CONV Reset();
```

Full UstLF system reset

Returns: none

Parameters: none.

Remarks

Releases all allocated licenses, resets all entered parameters to default values.

```
void CALL_CONV SetHeartBeatParams( int nIntervalInMs, int nMaxMissingPulsesCount );
```

Returns: none

Parameters:

`int nIntervalInMs`

Interval between “heart beat” packets between client and server in milliseconds

`int nMaxMissingPulsesCount`

Max number of missing "heart beat" packets in a row after which network client-server connection is considered invalid.

Remarks

When a network connection between client application and license server has been established a "heart beat" packets are used in order to monitor connection state. When something abnormal happens system starts to count lost "heart beats" and after the specified max number is reached connection is considered invalid, allocated licenses on the server side are released and appropriate event/callback is fired on the client application side. Default value of the "heart beat" interval is 4000 milliseconds and the max missing pulses is 30. Note: this function should be called ***before*** allocating licenses from the server (before a call to `AllocateLicenses(...)`).

```
BOOL CALL_CONV AllocateLicenses( LPCTSTR szSourcesStr );
```

Allocates licenses from specified sources.

Returns

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters:

`LPCTSTR szSourcesStr`

Semicolon separated list of sources. Zero-terminated string.

Remarks

Allocates licenses from each source listed in the input parameter `szSourcesStr`. Source string is a semicolon separated list or sources. Each source can be:

- a local folder
- a license server selected by IP (optionally enclosed in brackets)
- a license server selected by name
- an asterisk `"*"` (reference to all license servers available at the moment on the network)

The number of sources in the source string is not limited. Examples of the source string:

`"C:\\MyApp\\Lic;[192.168.2.57]"`	licenses will be allocated from specified folder and from license server located on computer with IP `192.168.2.57`
`"C:\\MyApp\\Lic;Gizmo;192.168.1.107"`	licenses will be allocated from specified folder and from license server located on computer with the name `Gizmo` and on computer with IP `192.168.1.107`
`"C:\\MyApp\\Lic;C:\\Licenses;*"`	licenses will be allocated from two local folders and from all servers available on the network
`"*"`	licenses will be allocated only from license servers available on the network

Each call to the function will overwrite the previous call, will **release** licenses from sources that do not correspond to the current source string and try to allocate licenses from new sources. In other words all sources that are planned to use should be listed in a single source string. Note, that each time a license is allocated UstLF increments system-wide instance counter associated with the license until it reaches the limit specified in the license file. After the limit is reached corresponding license cannot be allocated. When license is released (see functions `ShutDown`, `ReleaseLicenses`) system-wide instance counter associated with this license will be decremented.

```
BOOL CALL_CONV ReAllocateLicenses();
```

Re-allocates licenses from the sources specified in a `AllocateLicenses(...)` function call.

Returns

`TRUE`	if the function call succeeded
`FALSE`	if failed.

Parameters: none

Remarks

This function should be called when the license pool (specified by the sources string in a `AllocateLicenses(...)` function call) is changed. License pool considered to be changed when license files are added/deleted/renamed in the local license folders, in the folders monitored by license servers and in cases when license servers are stopped/started.

```
void CALL_CONV ReleaseLicenses();
Releases allocated licenses.
```

Returns: **none**

Parameters: none

Remarks

Call this function to release all allocated licenses. This function also will decrement system-wide instance counters for all released licenses. Normally there is no need to call this function and licenses should be release by a call to the function `ShutDown` when application is shutting down.

```
void CALL_CONV GetLicenseCount(int * pnNodeLockedCount, int * pnFloatingCount);
Returns the number of currently allocated licenses.
```

Returns: number of allocated licenses

Parameters:

`int * pnNodeLockedCount`

Pointer to the variable that will receive the number of allocated node-locked licenses

`int * pnFloatingCount`

Pointer to the variable that will receive the number of allocated floating licenses

Remarks:

Function returns total number of allocated licenses (node-locked and floating). Each parameter points to the user defined variable that will receive the number of corresponding licenses. Parameters may be set to NULL.

```
int CALL_CONV GetFeatureInfo( int nFeatureKey,
                              TCHAR * pDataBuffer,
                              int nDataBufLenInTChar,
                              int * pnDataStrLenInTChar );
```

Get the feature status and the data that is associated with the feature.

Returns:

`FEATURE_STATUS_UNKNOWN`	Feature does not exist in the feature map
`FEATURE_STATUS_DISABLED`	Feature is disabled
`FEATURE_STATUS_ENABLED`	Feature is enabled

Parameters:

`int nFeatureKey`

Feature key as it is defined in the feature map.

`TCHAR * pDataBuffer`

User allocated buffer for the data associated (in the license file) with the feature. Can be NULL

`int nDataBufLenInTChar`

Size of the data buffer. Can be 0

`int * pnDataStrLenInTChar`

Pointer to the integer variable to store the actual size of the data (length of the data string) associated with the feature. Value in the variable is updated on function return. Can be NULL.

Remarks

The size of the data assigned to the variable pointed by `pnDataStrLenInTChar` will be valid even if paramethers `pDataBuffer` and/or `nDataBufLenInTChar` are set to 0. This may be used to determine required data buffer size. If the size of the data buffer allows then the returned data will be terminated with zero character. If size is equal to the length of the data string then terminator is not applied. Is size of the buffer is less than the actual data size then buffer will be filled up to its limit and the rest of the data will be lost.

```
BOOL CALL_CONV QueryNetworkServers();
```

Query license servers on the network.

Returns:

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters: none

Remarks

Queries license servers on the network and builds internal table with server parameters. This function along with `GetServerCount` and `GetServerInfo` can be used to examine the network. However there is no need to call these functions explicitly in the process of license allocation – UstLF subsystem will collect all required information itself when required.

```
int CALL_CONV GetServerCount();
```

Returns the number of license servers detected on the network.

Returns:

the number of license servers

Parameters: none

Remarks:

Returned number is based on the information collected during last call to the `QueryNetworkServers` function.

```
BOOL CALL_CONV GetServerInfo( int nSrvIndex,
                              TCHAR * pHostNameBuffer,
                              int nHostNameBufLenInTChar,
                              int * pnHostNameStrLenInTChar,
                              TCHAR * pIpBuffer,
                              int nIpBufLenInTChar,
                              int * pnIpStrLenInTChar,
                              USHORT * pnPort );
```

Retrieves information about specified license server.

Returns:

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters:

`int nSrvIndex`

Index of the server to retrieve information for.

`TCHAR * pHostNameBuffer`

User allocated buffer to store host name where server is running. Can be NULL

`int nHostNameBufLenInTChar`

Size of the buffer to store host name. Can be 0.

`int * pnHostNameStrLenInTChar`

Pointer to user variable that will receive the length of the host name. Can be NULL.

`TCHAR * pIpBuffer`

User allocated buffer to store server IP. Can be NULL.

`int nIpBufLenInTChar`

Size of the buffer to store server IP. Can be 0.

`int * pnIpStrLenInTChar`

Pointer to the user variable that will receive the length of the server IP string. Can be NULL.

`USHORT * pnPort`

Pointer to user variable that will receive server port. Can be NULL.

Remarks

Function retrieves information about license server specified by index. Index is zero based and should be less then the value returned by the function `GetServerCount(...)`.

```
void CALL_CONV SetMessageCallback( MessageCallbackFn fn, void * pUserParam );
```

Sets the user callback function.

Returns: **none**

Parameters:

`MessageCallbackFn fn`

Pointer to the user callback function

`void * pUserParam`

Pointer to any kind of user object.

Remarks:

User callback function will be called only when appropriate events are enabled. See function `SetOptions(...)` description. In the callback function it is possible to allocate/reallocate licenses and/or check feature statuses or perform other activities. Callback function is called on a separate thread.

```
typedef void ( __stdcall * MessageCallbackFn )( void * pUserParam,
                                                ServerInfo * pSrvInfo,
                                                int nCode ) ;
```

User callback function prototype (see `SetMessageCallback(...)`)

Returns: **none**

Parameters:

`void * pUserParam`

Pointer to the user object passed in the call to the `SetMessageCallback(...)` function.

`ServerInfo * pSrvInfo`

Pointer to the structure identifying the source of the event/callback

`int nCode`

Code of the event/call-back.

Remarks:

The structure `ServerInfo` identifies the primary source of the event. This structure is declared in the header file **ustlf_api.h** as follows:

```
struct ServerInfo
{
  TCHAR*     m_szHostName;
  TCHAR*     m_szIP;
  USHORT     m_nPort;
};
```

Here:

`m_szHostName`	points to the buffer with the zero-terminated host name

`m_szIP`	points to the buffer with the zero-terminated host IP string.
`m_nPort`	contains server port.

Parameter `nCode` in the call-back prototype is a bitmap that will be populated with **combinations** of the following bit flags:

`CONNECTION_OPENED`	Network connection to the license server was opened
`CONNECTION_CLOSED`	Network connection to the license server was closed
`LIC_POOL_CHANGED`	License pool (local or on the server) was changed
`DATE_CHANGED`	Current date changed (it is midnight)
`FEATURE_SET_CHANGED`	Combined feature set was changed

The presence of the `LIC_POOL_CHANGED` bit indicates that the set of licenses (in one of the local folders or on the server side) was changed due to some reason (this can happen when licenses are added/removed/renamed, network servers stopped/started, network connection is lost). The presence of `DATE_CHANGED` bit indicates that current date changed (it is midnight). ***Both bit flags indicates that a call to license-allocation functions (`AllocateLicenses/ReAllocateLicenses`) might result in changes in allocated licenses, features statuses and/or features data.***

The presence of the `FEATURE_SET_CHANGED` bit indicates ***that features statuses and/or feature data are changed***. It is a good time to check features by making calls to the `GetFeatureInfo(...)` function.

Messages with the above mentioned flags can be enabled or disabled by setting options value (see `SetOptions(...)`) with corresponding bit flags (`OPT_EV_...`) set.

```
BOOL DecriptEmbeddedString( const UCHAR * pEncriptedData,
                             TCHAR * pResultStrBuffer,
                             int nResultStrBufferLenInTChar,
                             int * pnResultStrLenInTChar );
```

Decrypts embedded user string.

Returns:

`TRUE`	if the function call succeeded,
`FALSE`	if failed.

Parameters:

`const UCHAR * pEncriptedData`

Pointer to the encrypted user string

`TCHAR * pResultStrBuffer`

Pointer to the buffer where decrypted string will be stored

`int nResultStrBufferLenInTChar`

The size of the buffer where decrypted string will be stored in TCHARs

`int * pnResultStrLenInTChar`

Pointer to the user variable that will receive the length of the decrypted string.

Remarks:

Function takes a pointer to the encrypted string, decrypts it and places result into the user allocated buffer. The size of the buffer should be not less than the length of the decrypted string including termination zero. The actual length of decrypted string will be returned in the variable pointed by `pnResultStrLenInTChar`. The size of the string will be also returned if `pResultStrBuffer` and/or `nResultStrBufferLenInTChar` are set to NULL. This can be used to determine the expected length of the decrypted string.

To encrypt strings the **sfe.exe** utility should be used. Note that strings are decrypted with the key file used to create application descriptor. So, in other words the same key file should be used for creating application descriptor and for encrypting strings. Also note that the function `SetEmbeddedData(...)` should be called **before** any calls to the `DecriptEmbeddedString(...)`.

Library Files

UstLF library is comes in the following versions :

ustlfmmt.lib	- Multi-Byte character set, Multi-Threaded
ustlfmmd.lib	- Multi-Byte character set, Multi-Threaded DLL
ustlfumt.lib	- Unicode character set, Multi-Threaded
ustlfumd.lib	- Unicode character set, Multi-Threaded DLL

Functionality exposed in these libraries is described in the API section.

There is also a communication **DLL** ulmc.dll that should be used in scenarios with floating network licenses. In this case ulmc.dll should be placed in the same folder were the UstLF protected application resides. Also this DLL should be placed in the folder with the license server (**UlmSrv.exe**) executable.

Appendix (code samples)

The first code sample (Sample 1) shows the simplest case when license features are checked right after application startup. Things happen in the following sequence:

- application starts
- UstLF system is activated
- licenses are allocated
- features are examined
- application performs its target activities
- UstLF system shuts down
- application closes

No use of events is made in this sample. Feature data are not extracted.

The second code sample (Sample 2) shows the similar case with only one difference – feature data are extracted for further processing. No use of events is made in this sample.

The third code sample (Sample 3) shows one of the possible scenarios when feature status is examined dynamically – at any moment when license allocations changes. This sample makes use of events/callbacks.

For simplicity feature data are not extracted. This is the most recommended approach especially with floating license scenarios. Events will be fired when server is down or network connection is lost by some reason and application can react right away. This is not possible in Sample1 and Sample2 approaches.

All code samples reference (in `#include` statements) API header files **(ustlf_api.h, ustlf_def.h**), feature map file ("feature_map.h") and derived from it application descriptor file (see **Application descriptor generator**). Feature map file is the one that was used in previous samples and has the following content:

--- beginning of the file ---

```
#ifndef __USTLF_FEATURE_MAP__
#define __USTLF_FEATURE_MAP__

#define    EnableFileExport        1
#define    EnableFileImport        2
#define    EnableVideoCapture      3
#define    EnableNetwork           4

#endif
```

--- end of the file ---

When compiling samples in MS Visual Studio appropriate version of the **UstLF** library should be referenced in order to keep linker happy.

Sample 1

```
#include "stdafx.h"

#include "ustlf_def.h"
#include "ustlf_api.h"
// include application descriptor file
#include "app_des.h"
// include feature map file
#include "feature_map.h"

int _tmain(int argc, _TCHAR* argv[])
{
       // start and initialize UstLF system when application starts
  if ( UstLF::StartUp() )
  {
       UstLF::SetEmbeddedData( (LPCSTR)UstLF::__aULMPubKey__,
                                    (LPCSTR)UstLF::__aEncAppDescriptor );
  }

  // set appropriate options ( mask user name and host name in the line below )
  UstLF::SetOptions( OPT_LREQ_USER_NAME_MASK | OPT_LREQ_HOST_NAME_MASK );

  /*
  // use this code to create license request file
  #define MAX_ERR 256
  TCHAR sErrBuf[ MAX_ERR ] = {0};
  if ( !UstLF::CreateLicRequest( _T("C:\\Tmp\\GIP.req"), sErrBuf, MAX_ERR ) )
  {
       // failed to create lic. request file
       // process error here...
  }
  */

  // allocate licenses from the local folder and from any lic server on the network
  if ( UstLF::AllocateLicenses(_T("C:\\MyAppLicFolder;*")) )
  {
       // check status of the feature "EnableFileExport"
       // ( feature definitions are in "feature_map.h" file )
       int nFeatureStatus = UstLF::GetFeatureInfo( EnableFileExport, NULL, 0, NULL );
       if ( nFeatureStatus == FEATURE_STATUS_ENABLED ) // <- is feature is enabled ?
       {
             // Feature is enabled. Do something here...
       }
       else
       {
             // Feature is disabled. Do something...
       }
       // TODO: check other features statuses here in the similar way ...
  }
  // TODO: perform main application activities here...

  // shut down UstLF system before application is closed
  UstLF::ShutDown();

  return 0;
}
```

Sample 2

```
#include "stdafx.h"
#include "ustlf_def.h"
#include "ustlf_api.h"   // include application descriptor file
#include "app_des.h"     // include feature map file
#include "feature_map.h"
int _tmain(int argc, _TCHAR* argv[])
{
  // start and initialize UstLF system when application starts
  if ( UstLF::StartUp() )
  {
        UstLF::SetEmbeddedData( (LPCSTR)UstLF::__aULMPubKey__,
                                (LPCSTR)UstLF::__aEncAppDescriptor );
  }
  // set appropriate options ( mask user name in the line below )
  UstLF::SetOptions( OPT_LREQ_USER_NAME_MASK );
  /*
  // use this code to create license request file
  #define MAX_ERR 256
  TCHAR sErrBuf[ MAX_ERR ] = {0};
  if ( !UstLF::CreateLicRequest( _T("C:\\Tmp\\Request.req"), sErrBuf, MAX_ERR ) )
  {
        // failed to create lic. request file. Process error here...
  }
  */
  // allocate licenses from the local folder and from any lic server on the network
  if ( UstLF::AllocateLicenses( _T("C:\\MyAppLicFolder;*")) )
  {
        int nDataBufSize = 100;  /* <- some default feature data size */
        // allocate buffer for the feature data
        TCHAR * sFeatureData = (TCHAR*)malloc( nDataBufSize * sizeof(TCHAR) );
        int nFeatureDataSize = 0;
        int nFeatureStatus = FEATURE_STATUS_UNKNOWN;
        // check status of the feature "EnableFileExport" and get feature data
        // ( feature definitions are in "feature_map.h" file )
        nFeatureStatus = UstLF::GetFeatureInfo(EnableFileExport,
                                          sFeatureData,
                                          nDataBufSize,
                                          &nFeatureDataSize );
        if ( nFeatureStatus == FEATURE_STATUS_ENABLED ) // <- is feature is enabled ?
        {
              if ( nFeatureDataSize > nDataBufSize ) // <- is buffer large enough ?
              {
                    // reallocate buffer and get feature data one more time
                    nDataBufSize = nFeatureDataSize;
                    sFeatureData = (TCHAR*)realloc( sFeatureData, nDataBufSize * sizeof(TCHAR) );
                    UstLF::GetFeatureInfo(  EnableFileExport,
                                sFeatureData,
                                nDataBufSize,
                                &nFeatureDataSize );
              }
              // Feature is enabled. Feature data extracted. Do something here...
        }
        else
        {
              // Feature is disabled. Do something...
        }
        // TODO: check other features statuses here in the similar way ...
        free( sFeatureData ); //<- release memory
  }
  // TODO: perform main application activities here...
  UstLF::ShutDown(); // shut down UstLF system before application is closed
  return 0;
}
```

Sample 3

```
#include "stdafx.h"
#include <conio.h>
#include "ustlf_def.h"
#include "ustlf_api.h"   // include application descriptor file
#include "app_des.h"     // include feature map file
#include "feature_map.h"

void __stdcall MyAppMessageCallback( void * pUserParam, UstLF::ServerInfo * pSrvInfo, int nCode )
{
  if (( nCode & FEATURE_SET_CHANGED ) == FEATURE_SET_CHANGED )
  {
        // feature set changed. Get features statuses:

        // check status of the feature "EnableFileExport"
        // ( feature definitions are in "feature_map.h" file )
        int nFeatureStatus = UstLF::GetFeatureInfo( EnableFileExport, NULL, 0, NULL );
        if ( nFeatureStatus == FEATURE_STATUS_ENABLED ) // <- is feature is enabled ?
        {
              _tprintf( _T("+++ status ENABLED\n")) ;// Feature is enabled. Do something here...
        }
        else
        {
              _tprintf( _T("--- status DISABLED\n")) ;// Feature is disabled. Do something...
        }
        // TODO: check other features statuses here in the similar way ...
  }
}

int _tmain(int argc, _TCHAR* argv[])
{
        // start and initialize UstLF system when application starts
  if ( UstLF::StartUp() )
  {
        UstLF::SetEmbeddedData( (LPCSTR)UstLF::__aULMPubKey__,
                                            (LPCSTR)UstLF::__aEncAppDescriptor );
  }
  // set option to automatically reallocate licenses when license pool chenges
  UstLF::SetOptions( UstLF::GetOptions() | OPT_BH_AUTO_LIC_REALLOC );
        /*
  // use this code to create license request file
  #define MAX_ERR 256
  TCHAR sErrBuf[ MAX_ERR ] = {0};
  if ( !UstLF::CreateLicRequest( _T("C:\\Tmp\\GIP.req"), sErrBuf, MAX_ERR ) )
  {
        // failed to create lic. request file. Process error here...
  }
  */
  // set the callback function to handle events
  UstLF::SetMessageCallback( MyAppMessageCallback, NULL );

  // allocate licenses from the local folder and from any lic server on the network
  UstLF::AllocateLicenses(_T("C:\\MyAppLicFolder;*")) ;

  // TODO: perform main application activities...
  _tprintf( _T("press any key to exit\n\n")) ;
  _getch();
  // shut down UstLF system before application is closed
  UstLF::ShutDown();

  return 0;
}
```