

SIMPLELPR 2.0

Software Development Kit



Version History

Doc. version	Prod. version	Date	Description
1.0	1.0	28-Nov-09	First version of this document



Liability Disclaimer

Warelogic assumes no responsibility for any errors that may appear in this document nor does it make expressed or implied warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

To allow for design and specification improvements, the information in this document is subject to change at any time, without notice. Reproduction of this document or portions thereof without prior written approval of Warelogic is prohibited.

Warelogic shall not be liable for incidental or consequential damages in connection with, or arising out of the furnishing, performance, or use of this document and the program material that it describes.

Microsoft, MS, MSN, ActiveX, Windows, Windows NT, Visual Basic, Visual C++, and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Microsoft products are licensed to OEMs by Microsoft Licensing, Inc., a wholly owned subsidiary of Microsoft Corporation.

All other product, brand, or trade names used in this publication are the trademarks or registered trademarks of their respective trademark owners.



Table of Contents

1	Introduction	5
2	Supported Countries and Limitations	6
2.1	German License Plates	6
2.2	Spanish License Plates	6
2.3	British License Plates	6
3	Prerequisites, Installation and Deployment	7
3.1	Prerequisites	7
3.2	Installation	7
3.3	Deployment	7
4	License Plate Recognition with <i>SimpleLPR 2.0</i>	8
4.1	<i>SimpleLPR 2.0</i> Instantiation	8
	<i>Native C++ Applications</i>	8
	<i>.NET Applications</i>	8
4.2	Engine Configuration	9
	<i>Native C++ Applications</i>	9
	<i>.NET Applications</i>	10
4.3	License Plate Recognition	10
	<i>Native C++ Applications</i>	10
	<i>.NET Applications</i>	11
4.4	Considerations on Error Handling and Resource Management	12
	<i>Native C++ Applications</i>	12
	<i>.NET Applications</i>	12
5	C++ Interface Reference	13
5.1	Functions	13
5.1.1	Setup	13
5.2	Structures	13
5.2.1	Rect	13
5.2.2	Element	13
5.3	Interfaces	14
5.3.1	IReferenceCounted	14
5.3.2	IErrorInfo	14
5.3.3	ICandidate	15
5.3.4	ICandidates	16



5.3.5	IProcessor.....	16
5.3.6	ISimpleLPR.....	18
6	.NET Interface Reference	21
6.1	SimpleLPR	21
6.1.1	Methods	21
6.2	ISimpleLPR.....	21
6.2.1	Properties.....	21
6.2.2	Methods	21
6.3	IProcessor.....	23
6.3.1	Methods	23
6.4	Candidate.....	24
6.4.1	Fields.....	24
6.5	Element.....	24
6.5.1	Fields.....	25
7	SDK Files	26



I Introduction

At the present time there are many license plate recognition (LPR) solutions in the market, designed to work in areas such as: traffic control and monitoring, parking access, vehicle management, detection of security violations and so forth. The best LPR engines support many countries, all license plate layouts, char sets, and some of them feature a 98% recognition rate or greater at recognition speeds of less than 50 ms per frame. Not surprisingly, this superior performance generally comes at a price; the best engines can cost thousands of dollars per runtime license.

On the other hand, there are consumer oriented applications that would benefit from LPR but the cost of a high end LPR engine is too high for their segment. This is where *SimpleLPR 2.0* comes in. *SimpleLPR 2.0* is a royalty free low end LPR engine directed to price sensitive applications that can cost an order of magnitude less than its high end counterparts, at the expense of a limited number of supported countries and lower accuracy; the typical recognition rate is about 90% although as new releases appear this number is expected to improve. For applications that can trade some recognition accuracy for an affordable price *SimpleLPR 2.0* is an option that should be considered.

From a developer's perspective, *SimpleLPR 2.0* main design goal is simplicity in integration and deployment. Hence, *SimpleLPR 2.0* runtime can be redistributed along with third party applications by just x-copying a pair of *dll* files that do not require COM registration. *SimpleLPR2.dll* is dual, in addition to a C++ interface for native C++ applications it also exports a .NET object model that can be directly used from VB or C#. The current version supports license plates from Germany, Spain and United Kingdom (see [supported countries and limitations](#)). It can read 24 bit RGB or 8 bit grayscale images from *JPEG*, *TIFF*, *BMP* or *PNG* files, or 8 bit grayscale images from a memory buffer.

This guide contains both the product description and documentation for developers. In the following sections all required information on how to integrate *SimpleLPR 2.0* into a third party application is provided.



2 Supported Countries and Limitations

The current version can read license plates from Germany, Spain and United Kingdom provided that they are in good condition; namely the license plates should be crisp, readable, without occlusion, bumps or scratches. In addition, images supplied to the *SimpleLPR 2.0* engine should portray the license plate as viewed from a frontal angle, and the height of the license plate characters should be 20 pixel or taller.

Irrespective of complying with the above constraints, in general not all valid license plates from a specific country can be recognized. The level of support varies from country to country as it can be seen below.

2.1 German License Plates

The article http://en.wikipedia.org/wiki/Vehicle_registration_plates_of_Germany is the source where the syntax rules for German license plates verification have been obtained from. Specifically, the current 1994 format is supported for private and public vehicles as described in the article. The list of valid district codes has been extracted from http://de.wikipedia.org/wiki/Liste_der_Kfz-Kennzeichen_in_Deutschland. Trailer, diplomatic, motorcycle and military vehicles are not supported.

Square license plates are only partially supported. Due to a limitation in the current engine only license plates with two or three letter district codes can be read. This shortcoming will be addressed in the future.

2.2 Spanish License Plates

Private and public car license plates issued after 1980 are supported, as described in http://en.wikipedia.org/wiki/Vehicle_registration_plates_of_Spain. Trailer, diplomatic, motorcycle and military vehicles are not supported. On the other hand, square license plates are fully supported.

2.3 British License Plates

SimpleLPR 2.0 can read private and public license plates from Great Britain issued after 1883. See http://en.wikipedia.org/wiki/Vehicle_registration_plates_of_the_United_Kingdom. Trailer, diplomatic, motorcycle and military vehicles are not supported. Square license plates issued after 2001 are supported.



3 Prerequisites, Installation and Deployment

3.1 Prerequisites

SimpleLPR 2.0 requires a computer running a *Windows XP* or newer operating system, the *Visual Studio 2008 C++ runtime* and the *MSXML 6.0 XML parser*. From a hardware standpoint, the only requirement is a CPU supporting SSE2 extensions.

Any 5 year old or newer computer kept regularly updated with *Microsoft's* patches and service packs should already meet the above requirements. Nevertheless, if required the above software prerequisites can be downloaded from *Microsoft*.

1. The *Visual Studio 2008 C++ runtime* can be downloaded from <http://www.microsoft.com/downloads/details.aspx?familyid=A5C84275-3B97-4AB7-A40D-3802B2AF5FC2&displaylang=en>
2. *MSXML 6.0* can be installed from <http://www.microsoft.com/downloads/details.aspx?FamilyID=D21C292C-368B-4CE1-9DAB-3E9827B70604&displaylang=en>
3. To check if a specific processor supports SSE2 <http://en.wikipedia.org/wiki/SSE2>

3.2 Installation

The *SimpleLPR 2.0 SDK* is shipped in a *MSI* installer. It creates the [SDK file structure](#) and automatically takes care of the dependencies: if needed it will install the *Visual Studio 2008 C++ runtime*. It will also verify that *MSXML 6.0* is installed and check for SSE2 extensions, prompting the user if any of them are not available.

Upon installation *SimpleLPR 2.0* operates in evaluation mode, which lasts for 30 days. Once the evaluation period terminates *SimpleLPR 2.0* will stop working, unless a valid product key is supplied.

3.3 Deployment

SimpleLPR 2.0 can be deployed by simply copying *bin\SimpleLPR2.dll* and *bin\libguide40.dll* to the same folder as the application executable. Two considerations to be borne in mind are

1. *SimpleLPR 2.0* cannot operate in evaluation mode when redistributed. The application has to supply a product key.
2. The installer of the third party application must ensure that all [prerequisites](#) are met.



4 License Plate Recognition with *SimpleLPR 2.0*

This section describes the major steps to integrate *SimpleLPR 2.0* into an application. In each stage both C++ and .NET flavors are examined individually. .NET sample code is presented in C#.

4.1 *SimpleLPR 2.0* Instantiation

This step comprises the tasks of loading *SimpleLPR2.dll* into the application process space and creating an instance of the *SimpleLPR* engine.

Native C++ Applications

For native C++ applications, *SimpleLPR2.dll* exports a single public function, *Setup*, which works as a factory method of *ISimpleLPR* objects. The prototype of the *Setup* function is defined in *include/SimpleLPR.h*

```
ISimpleLPR * _stdcall Setup() throw ();
```

Since a *.lib* file is not provided for *SimpleLPR2.dll* the binding has to be done programmatically by means of either the *LoadLibrary* or *LoadLibraryEx* calls. Once a pointer to the *Setup* function is acquired, it can be used to create an instance of the *SimpleLPR 2.0* engine, encapsulated in the *ISimpleLPR* interface. The following code snippet illustrates this.

```
// Load the SimpleLPR dll
HMODULE hLib = LoadLibraryEx( L"SimpleLPR2.dll", NULL, 0 );
if ( hLib == NULL )
    ... \\ Handle error condition

// Get the entry point to the Setup function
SimpleLPRSetupFunc pfnSetup = (SimpleLPRSetupFunc)::GetProcAddress( hLib,
"Setup" );
if ( pfnSetup == 0 )
    ... \\ Handle error condition

// Create an instance of the SimpleLPR engine
ISimpleLPR *pLPR = (*pfnSetup)();
if ( pLPR == NULL )
    ... \\ Handle error condition

... \\ Do your stuff

pLPR->release(); // Delete engine
```

.NET Applications

The .NET case is more straightforward since *SimpleLPR2.dll* is a .NET assembly and therefore it is early bound. The user only has to add it to the project as an assembly reference. Next, an instance of the *SimpleLPR 2.0* engine can be created using the *Setup* static factory method in the *SimpleLPR* class.

```
ISimpleLPR lpr = SimpleLPR.Setup();
```



4.2 Engine Configuration

Firstly, depending on whether *SimpleLPR 2.0* is to be used in evaluation mode or not, a valid product key must be provided. The product key can be supplied either from a file or from a memory buffer. As it contains personal data from the purchaser in readable form the second method is preferred.

Then, the user must decide what countries are to be enabled and what are their relative weights. These weights are used to break ties in the case that two or more candidates from different countries are equally feasible. The *SimpleLPR 2.0* engine exports methods to assign each country a weight equal or greater than 0. Setting a weight equal to 0 effectively disables a country. Those weights are later normalized to the [0...1] range. As license plate recognition goes on, chances are that discovered groups of text and numbers match the syntax verification rules of more than one country. The 'goodness' index of each license plate candidate is multiplied by the normalized country weight and the best candidate is kept.

Enabling multiple countries is discouraged as it has an impact on reliability. In case this cannot be avoided, the relative country weights should be set equal to the expected ratio of vehicles from each country.

The samples below show how to set the product key and configure country weights in each development platform.

Native C++ Applications

```
// Set the product key
bool bOk = pLPR->productKey_set( L"productkey.xml" );
if ( ! bOk )
    ... \\ Handle error condition

// Enable Germany, disable Spain and United Kingdom
bOK = pLPR->countryWeight_set( L"Germany", 1.f )
if ( ! bOk )
    ... \\ Handle error condition
bOK = pLPR->countryWeight_set( L"Spain", 0.f )
if ( ! bOk )
    ... \\ Handle error condition
bOK = pLPR->countryWeight_set( L"UK-GreatBritain", 0.f )
if ( ! bOk )
    ... \\ Handle error condition

// Apply changes
bOK = pLPR->realizeCountryWeights();
if ( ! bOk )
    ... \\ Handle error condition
```



.NET Applications

```
// Set the product key
lpr.set_productKey("productkey.xml");

// Enable Germany, disable Spain and United Kingdom
lpr.set_countryWeight("Germany", 0.0f);
lpr.set_countryWeight("Spain", 0.0f);
lpr.set_countryWeight("UK-GreatBritain", 0.0f);

// Apply changes
lpr.realizeCountryWeights();
```

4.3 License Plate Recognition

Once the *SimpleLPR 2.0* engine has been properly configured the remaining step is creating an instance of an *IProcessor* object to perform license plate recognition. Since *IProcessor* is not thread safe, each working thread involved in LPR should keep its own instance of *IProcessor*.

An *IProcessor* can read and process images from a file, in either *RGB* or *grayscale JPEG, TIFF, PNG* and *BMP* formats or, alternatively, it can process images directly from a memory buffer. In the later case the image must be *grayscale 8-bit*. The output of the LPR methods is a list of license plate candidate objects. Each object having the license plate text in Unicode string form, the global confidence index, and the bounding box and confidence value of all individual elements in the license plate.

The license plate text is formatted according to the rules of each specific country so spaces or hyphens can be added to separate groups of text and numbers. The returned confidence index is the direct output from the OCR classifier. It ranges from 0 to 1, but it should not be regarded as a probability. Depending on the license plate font and the specific glyph its operating range can vary significantly. As a result, care should be taken when setting thresholds to discard weak candidates. The rule of thumb is that very low values almost always point to an unreliable detection, and values close to one imply a safe candidate. Unfortunately, in most cases the confidence values lay in no one's land and nothing can be decided. Future versions of the product will try to address this problem. Finally, the global candidate confidence index is taken as the lowest value of all elements in the license plate.

Native C++ Applications

```
// Create processor
IProcessor *pProc = pLPR->createProcessor();
if( pProc == NULL )
    ... \\ Handle error condition

// Process source file
ICandidates *pCds = pProc->analyze( L"vehicle.jpg", 120 /* max char height
*/ );
if( pCds == NULL )
    ... \\ Handle error condition

if ( pCds->numCandidates_get() == 0 )    std::wcout << L"No license plate
found" << std::endl;
else
{
    std::wcout << pCds->numCandidates_get() << L" license plates found:" <<
std::endl;
```



```

// Iterate over all candidates
for ( _SIZE_T i = 0; i < pCds->numCandidates_get(); ++i )
{
    ICandidate *pCd = pCds->candidate_get( i );
    if( pCds == NULL )
        ... \\ Handle error condition

    std::wcout << L"Candidate " << i + 1 << std::endl;
    std::wcout << L"Text: " << pCd->text_get() << L" , confidence: ";

    std::wcout << pCd->confidence_get() << L", Light background " << pCd-
>brightBackground_get() << std::endl;
    std::wcout << L"Elements: " << std::endl;

    // Iterate over all elements
    for ( _SIZE_T j = 0; j < pCd->numElements_get(); ++j )
    {
        Element e;
        pCd->element_get( j, e );

        std::wcout << L"Glyph: " << e.glyph << L", confidence: " <<
e.fConfidence;
        std::wcout << L", left: " << e.boundingBox.left << L", top: " <<
e.boundingBox.top;
        std::wcout << L", width: " << e.boundingBox.width << L", height: " <<
e.boundingBox.height;
        std::wcout << std::endl;
    }
    pCd->release();
}
}
// Cleanup
pCds->release();
pProc->release();

```

.NET Applications

```

// Create Processor
IProcessor proc = lpr.createProcessor();

// Process source file
List<Candidate> cds = proc.analyze( "vehicle.jpg", 120 120 /* max char
height */ );
if (cds.Count > 0)
{
    // Iterate over all candidates
    foreach ( Candidate cd in cds )
        Console.WriteLine(" [{0}, {1}]", cd.text, cd.confidence);
    Console.WriteLine();
}
else
    Console.WriteLine("Nothing detected");

```



4.4 Considerations on Error Handling and Resource Management

For the sake of simplicity, the subject of error handling has been deliberately omitted in the sample code above. In addition, the reference counting scheme implemented in the C++ interface deserves some discussion.

Native C++ Applications

Most C++ developers consider that exceptions are the best way of dealing with error conditions in C++. However, exceptions are compiler vendor dependant and are not appropriate for a public interface exported from a *DLL*. For this reason, *SimpleLPR 2.0* relies on the old technique of returning error codes and maintaining a per-thread error state. Every method in the C++ interface returns either a boolean (true meaning success) or a NULL value in case that the method is expected to return a pointer.

Once an error condition has been determined, further error information can be queried by means of the [lastError_get\(\)](#) method. Each thread keeps its own copy of the *last_error* variable. The following code snippet shows how the error state can be obtained.

```
SimpleLPR2_Native::IErrorInfo *pErr = pLPR->lastError_get();
if ( pErr != NULL )
{
    std::wcerr << L"Error occurred. Error code: " << pErr->errorCode << L",
description: " << pErr->description << std::endl;
    pErr->release(); // Free object
}
```

On the subject of resource management, the standard way of de-allocating objects in C++ is using the *delete* operator. On the other hand, the golden rule of any *DLL* returning C++ objects is that resources must be allocated and de-allocated inside the *DLL*. A way to accomplish it is making objects delete themselves. The reference counting idiom does this; each object keeps track of how many references point to it. When this counter reaches 0 the object deletes itself. Thus, all objects exported in the C++ interface derive from *IReferenceCounted*. This interface has two methods, [addRef](#), which increments objects reference counter by one, and [release](#), which decrements it. All objects returned by the *SimpleLPR 2.0* factory methods have their reference counter set to 1 and a call to their *release* method actually destroys them.

.NET Applications

As usual, *.NET* makes life easier for developers. Error conditions are dealt with the use exceptions, which is the standard way in *.NET*. Likewise, resource management is not an issue here as the garbage collector takes care of this subject.



5 C++ Interface Reference

The file `include\SimpleLPR.h` contains all class declarations and function prototypes described in this section. All members are declared under the `SimpleLPR2_Native` namespace.

5.1 Functions

5.1.1 Setup

```
SIMPLELPR_API ISimpleLPR * SIMPLELPR_CALL Setup() throw ();
```

Description: This is the only function exported by SimpleLPR2.dll. It is a factory method for [ISimpleLPR](#) objects.

Returns: A pointer to a [ISimpleLPR](#) object. The returned [ISimpleLPR](#) object must be de-allocated by calling its [release](#) method.

Remarks: Since no .lib file is provided in the SDK, `SimpleLPR2.dll` must be loaded dynamically.

5.2 Structures

5.2.1 Rect

This structure represents a rectangle.

```
struct Rect
{
    __int32 left;
    __int32 top;
    __int32 width;
    __int32 height;
};
```

Members

left Leftmost coordinate of the rectangle.

top Topmost coordinate of the rectangle.

width Rectangle width.

height Rectangle height.

5.2.2 Element

It stores the LPR results of an individual character in a license plate.

```
struct Element
{
    wchar_t glyph;
    _REAL_T fConfidence;
    Rect boundingBox;
};
```

Members

glyph Unicode representation of the character.



fConfidence 'Goodness' of the recognition. Its range is 0 to 1 and can be used to rank candidates although it should not be regarded as a probability. In general, a 2x goodness value is not twice as good as x. See the section on [LPR](#) for more detailed information on confidence values.

boundingBox Bounding rectangle of the character, in pixel coordinates..

5.3 Interfaces

5.3.1 IReferenceCounted

This interface constitutes the base for all interfaces. It manages object life cycles by means of reference counting. All factory methods in this library return pointers to objects with their reference count set to 1 or more.

It is advisable to call the [addRef](#) method every time a pointer alias is created. Before an object pointer goes out of scope its [release](#) method should be called. When an object reference count reaches zero the object is destroyed.

```
struct IReferenceCounted
{
    virtual void addRef( void ) throw () = 0;
    virtual void release( void ) throw () = 0;
};
```

Methods

```
virtual void addRef( void ) throw () = 0;
```

Description: Increments an object reference count by one.

```
virtual void release( void ) throw () = 0;
```

Description: Decrements an object reference count by one. When it reaches 0 the object is destroyed.

5.3.2 IErrorInfo

This class conveys an error code and a description. Every time an error occurs an *IErrorInfo* object is created and kept in thread local storage (TLS). To retrieve it use the [ISimpleLPR:lastError_get](#) method.

```
struct IErrorInfo : public IReferenceCounted
{
    virtual _HRESULT errorCode_get() const throw () = 0;
    virtual const wchar_t * description_get() const throw () = 0;
};
```

Methods

```
virtual _HRESULT errorCode_get() const throw () = 0;
```



Description: Returns a COM like *HRESULT* error code.

```
virtual const wchar_t * description_get() const throw () = 0;
```

Description: Returns a textual description of the error.

5.3.3 ICandidate

Encapsulates a license plate candidate.

```
struct ICandidate : public IReferenceCounted
{
    virtual const wchar_t *text_get() const throw () = 0;
    virtual const wchar_t *country_get() const throw () = 0;
    virtual _REAL_T confidence_get() const throw () = 0;
    virtual bool brightBackground_get() const throw () = 0;
    virtual _SIZE_T numElements_get() const throw () = 0;
    virtual bool element_get( _SIZE_T id, /*[out]*/Element &rElem ) const
    throw () = 0;
};
```

Methods

```
virtual const wchar_t *text_get() const throw () = 0;
```

Description: Returns the Unicode representation of the license plate string. Separators are represented as white space.

```
virtual const wchar_t *country_get() const throw () = 0;
```

Description: Returns the country code in string form.

```
virtual _REAL_T confidence_get() const throw () = 0;
```

Description: Returns the overall 'goodness' of the recognition. Currently it is calculated as the minimum goodness value of all individual characters in the license plate.

```
virtual bool brightBackground_get() const throw () = 0;
```

Description: *true* if the license plate features dark text on a light background. *false* if otherwise.

```
virtual _SIZE_T numElements_get() const throw () = 0;
```

Description: Number of components in the license plate..

```
virtual bool element_get( _SIZE_T id, /*[out]*/Element &rElem
) const throw () = 0;
```



Description: Information about the individual chars that make up the license plate. They are listed in the same order as they appear in the text string. To know the physical layout of the license plates use the [Element::bbox](#) field.

5.3.4 ICandidates

Encapsulates a collection of license plate candidates.

```
struct ICandidates : public IReferenceCounted
{
    virtual _SIZE_T numCandidates_get() const throw () = 0;
    virtual ICandidate *candidate_get( _SIZE_T id ) const throw () = 0;
};
```

Methods

```
virtual _SIZE_T numCandidates_get() const throw () = 0;
```

Description: Returns the number of elements in the collection.

```
virtual ICandidate *candidate_get( _SIZE_T id ) const throw ()
= 0;
```

Description: Returns a candidate object given its index.

Parameters

Input

Id: Candidate identifier. Must be \leq [numCandidates_get\(\)](#)

Returns: A pointer to the selected candidate object.

Remarks: The returned [ICandidate](#) object must be de-allocated by calling its [release](#) method.

5.3.5 IProcessor

Encapsulates the LPR functionality of *SimpleLPR 2.0*. This class is not multi-threaded and, therefore, each thread should use a different *IProcessor* instance.

```
struct IProcessor : public IReferenceCounted
{
    virtual ICandidates *analyze( const void *pcvImgData,
                                  _SIZE_T widthStep,
                                  _SIZE_T width,
                                  _SIZE_T height,
                                  _SIZE_T maxCharHeight ) throw () = 0;
    virtual ICandidates *analyze( const wchar_t * pcwsImgPath,
                                  _SIZE_T maxCharHeight ) throw () = 0;
};
```

Methods



```
virtual ICandidates *analyze( const void *pcvImgData,
                             _SIZE_T widthStep,
                             _SIZE_T width,
                             _SIZE_T height,
                             _SIZE_T maxCharHeight ) throw ()
= 0;
```

Description: Looks for license plate candidates in a memory buffer containing an 8 bit gray scale image.

Parameters

Input

plmgData: Pointer to the first image row. The image must be 8 bit gray scale and top down. The top row of the image is the first row in memory, followed by the next row down.

widthStep: Distance in bytes between starts of consecutive rows in the source image.

width: Image width in pixels.

height: Image height in pixels.

maxCharHeight:: Maximum height in pixels of the characters in the license plate

Returns: A pointer to an [ICandidates](#) collection containing all license plate candidates. If something goes wrong it returns a NULL pointer, use [ISimpleLPR::lastError_get](#) to get the error information.

Remarks: This method is not multi-threaded. The returned [ICandidates](#) object must be de-allocated by calling its [release](#) method.

```
virtual ICandidates *analyze( const wchar_t * pcwslmgPath,
                             _SIZE_T maxCharHeight ) throw ()
= 0;
```

Description: Looks for license plate candidates in an image in a .jpg, .png, .tif or .bmp file. The images can be either 24 bit RGB or 8 bit gray scale.

Parameters

Input

pcwslmgPath: Path to a file containing a 24 bit RGB or 8 bit gray scale image.

maxCharHeight:: Maximum height in pixels of the characters in the license plate

Returns: A pointer to a [ICandidates](#) collection containing all license plate candidates. If something goes wrong it returns a NULL pointer, use [ISimpleLPR::lastError_get](#) to get the error information.

Remarks: This method is not multi-threaded. The returned [ICandidates](#) object must be de-allocated by calling its [release](#) method.



5.3.6 ISimpleLPR

Encapsulates the *SimpleLPR 2.0* engine.

```
struct ISimpleLPR : public IReferenceCounted
{
    virtual _SIZE_T numSupportedCountries_get() const throw () = 0;
    virtual bool countryCode_get( _SIZE_T id,
                                  /*[out]*/const wchar_t * &rpcwsCode
                                ) const throw () = 0;
    virtual bool countryWeight_get( _SIZE_T id,
                                     /*[out]*/_REAL_T &rfWeight
                                   ) const throw () = 0;
    virtual bool countryWeight_get( const wchar_t *id,
                                     /*[out]*/_REAL_T &rfWeight
                                   ) const throw () = 0;
    virtual bool countryWeight_set( _SIZE_T id,
                                     _REAL_T fWeight ) throw () = 0;
    virtual bool countryWeight_set( const wchar_t *id,
                                     _REAL_T fWeight ) throw () = 0;
    virtual bool realizeCountryWeights() throw () = 0;
    virtual IErrorInfo *lastError_get( bool bClear = true ) throw () = 0;
    virtual IProcessor *createProcessor() throw () = 0;
    virtual bool productKey_set( const wchar_t *productKeyPath ) throw ()
= 0;
    virtual bool productKey_set( const void *key, _SIZE_T keySize ) throw
() = 0;
};
```

Methods

```
virtual _SIZE_T numSupportedCountries_get() const throw () =
0;
```

Description: Returns the number of supported countries.

```
virtual bool countryCode_get( _SIZE_T id, /*[out]*/const
                               wchar_t * &rpcwsCode ) const
throw () = 0;
```

Description: Given a country index it returns its string identifier.

Parameters

Input

id: The country index. $0 \leq id \leq \text{numSupportedCountries_get()} - 1$.

Output:

rpcwsCode:: The country Unicode string identifier..

Returns: *true* if the method succeeded, *false* otherwise. In the later case use

[ISimpleLPR:lastError_get](#) to obtain the error information.

```
virtual bool countryWeight_get( _SIZE_T id,
                                 /*[out]*/_REAL_T &rfWeight )
const throw () = 0;
```



Description: Given a country index it returns the relative country weight. Weights are used to break ties when a candidate can belong to multiple countries.

Parameters

Input

Id: The country index. $0 \leq id \leq \text{numSupportedCountries_get()} - 1$.

Output

rfWeight:: The relative weight of the country.

Returns: *true* if the method succeeded, *false* otherwise. In the later case use

[ISimpleLPR::lastError_get](#) to obtain the error information.

```
virtual bool countryWeight_get( const wchar_t *id,
                               /*[out]*/_REAL_T  &rfWeight  )
const throw () = 0;
```

Description Given a country string identifier it returns the relative country weight. Weights are used to break ties when a candidate can belong to multiple countries.

Parameters

Input

Id: The country string identifier. See [countryCode_get](#).

Output

rfWeight:: The relative weight of the country.

Returns: *true* if the method succeeded, *false* otherwise. In the later case use

[ISimpleLPR::lastError_get](#) to obtain the error information.

```
virtual bool countryWeight_set( _SIZE_T id,
                               _REAL_T fWeight) throw () = 0;
```

Description: Given a country index it sets the country relative weight. Weights are used to break ties when a candidate can belong to multiple countries.

Parameters

Input

id: The country index. $0 \leq id \leq \text{numSupportedCountries_get()} - 1$.

fWeight:: The desired country weight. $fWeight \geq 0$.

Returns: *true* if the method succeeded, *false* otherwise. In the later case use

[ISimpleLPR::lastError_get](#) to obtain the error information.

Remarks: This method is not multi-threaded. Weight must be ≥ 0 . Use a zero weight to effectively disable a specific country.

```
virtual bool realizeCountryWeights() throw () = 0
```

Description: Rebuilds the internal country verification lookup tables based on which countries are enabled and their relative weights. Call it once you have finished configuring country weights.

Returns: *true* if the method succeeded, *false* otherwise. In the later case use

[ISimpleLPR::lastError_get](#) to obtain the error information.

Remarks: This method is not multi-threaded. Depending on the countries selected this method can be time consuming. After this method execution all existing and new



[IProcessor](#) instances will start using the new weights. Avoid calling this method when another thread is executing [IProcessor::analyze](#).

```
virtual IErrorInfo *lastError_get( bool bClear = true ) throw  
( ) = 0;
```

Description Returns an [IErrorInfo](#) object that describes the latest error occurred.

Parameters

Input

bClear: If *true* the error state will be cleared after this call.

Returns: An [IErrorInfo](#) object that describes the latest error or NULL if no error has occurred since application startup or the last call to [lastError_get](#) with *bClear* set to *true*.

Remarks: This method is multi-threaded. In particular each thread maintains a TLS slot with error state so threads can be independent from each other.

```
virtual IProcessor *createProcessor() throw ( ) = 0;
```

Description: Creates a new [IProcessor](#) object.

Returns: The newly created [IProcessor](#) instance, or NULL if the method failed. Use [ISimpleLPR::lastError_get](#) to obtain the error information.

Remarks: This method is multi-threaded. For this method to succeed, either the product must be within the evaluation period or a valid product key must be supplied using [productKey_set](#). The returned [IProcessor](#) object must be de-allocated by calling its release method.

```
virtual bool productKey_set( const wchar_t *productKeyPath )  
throw ( ) = 0;
```

Description: Sets the product key from a license file.

Parameters

Input

productKeyPath: Path to the product key file..

Returns: *true* if the method succeeded, *false* otherwise. In the later case use [ISimpleLPR::lastError_get](#) to obtain the error information.

```
virtual bool productKey_set( const void *key, _SIZE_T keySize  
) throw ( ) = 0;
```

Description: Sets the product key from a memory buffer.

Parameters

Input

key: Pointer to the memory buffer.

keySize: *keySize* in bytes.

Returns: *true* if the method succeeded, *false* otherwise. In the later case use [ISimpleLPR::lastError_get](#) to obtain the error information.



6 .NET Interface Reference

6.1 SimpleLPR

SimpleLPR 2.0 factory class.

6.1.1 Methods

6.1.1.1 Setup

Creates a [ISimpleLPR](#) object.

6.1.1.1.1 Return Value

Return Value: A [ISimpleLPR](#) instance.

6.2 ISimpleLPR

Encapsulates the *SimpleLPR 2.0* engine.

6.2.1 Properties

6.2.1.1 numSupportedCountries

Return Value: Number of supported countries.

6.2.2 Methods

6.2.2.1 set_productKey(System.Byte[])

Sets the product key from a memory buffer.

6.2.2.1.1 Parameters

productKey: Byte array containing the product key.

6.2.2.2 set_productKey(System.String)

Sets the product key from a license file.

6.2.2.2.1 Parameters

productKeyPath: Path to the product key file.

6.2.2.3 createProcessor

Creates a new [IProcessor](#) object.



6.2.2.3.1 Return Value

Return Value: The newly created [IProcessor](#) instance.

6.2.2.4 realizeCountryWeights

Rebuilds the internal country verification lookup tables based on which countries are enabled and their relative weights. Call it once you have finished configuring country weights.

6.2.2.5 set_countryWeight(System.String,System.Single)

Sets the country relative weight. Weights are used to break ties when a candidate can belong to multiple countries.

6.2.2.5.1 Parameters

id: The country string identifier. See [get_countryCode\(System.UInt32\)](#).

val: The desired country weight. *val* ≥ 0

6.2.2.6 get_countryWeight(System.String)

Given a country string identifier it gets the country relative weight. Weights are used to break ties when a candidate can belong to multiple countries.

6.2.2.6.1 Parameters

id: The country string identifier. See [get_countryCode\(System.UInt32\)](#).

6.2.2.6.2 Return Value

Return Value: The relative weight of the country.

6.2.2.7 set_countryWeight(System.UInt32,System.Single)

Sets the country relative weight. Weights are used to break ties when a candidate can belong to multiple countries.

6.2.2.7.1 Parameters

id: The country index. $0 \leq id \leq \text{numSupportedCountries} - 1$

val: The desired country weight. *val* ≥ 0

6.2.2.8 get_countryWeight(System.UInt32)

Given a country index it gets the country relative weight. Weights are used to break ties when a candidate can belong to multiple countries.



6.2.2.8.1 Parameters

id: The country index. $0 \leq id \leq \text{numSupportedCountries} - 1$

6.2.2.8.2 Return Value

Return Value: The relative weight of the country.

6.2.2.9 get_countryCode(System.UInt32)

Given a country index it returns its string identifier.

6.2.2.9.1 Parameters

id: The country index. $0 \leq id \leq \text{numSupportedCountries} - 1$

6.2.2.9.2 Return Value

Return Value: The country string identifier.

6.3 IProcessor

Provides access to the license plate recognition functionality of SimpleLPR.

This class is not multi-threaded and, therefore, each thread should use a different [IProcessor](#) instance.

6.3.1 Methods

6.3.1.1 analyze(System.String, System.UInt32)

Looks for license plate candidates in an image loaded from a .jpg, .png, tif or .bmp file. The images can be either 24 bit RGB or 8 bit gray scale.

imgPath: Path to a file containing a 24 bit RGB or 8 bit gray scale image.

maxCharHeight: Maximum height in pixels of the characters in the license plate.

Return Value: List of [Candidate](#) containing all license plate candidates.

6.3.1.2 analyze(System.IntPtr, System.UInt32, System.UInt32, System.UInt32, System.UInt32)

Looks for license plate candidates in a memory buffer containing a 8 bit gray scale image.

6.3.1.2.1 Parameters

pImgData: Pointer to the first image row. The image must be 8 bit gray scale and top down: the top row of the image is the first row in memory, followed by the next row down.



widthStep: Distance in bytes between starts of consecutive rows in the source image.

width: Image width in pixels.

height: Image height in pixels.

maxCharHeight: Maximum height in pixels of the characters in the license plate.

6.3.1.2.2 Return Value

Return Value: List of [Candidate](#) containing all license plate candidates.

6.4 Candidate

Encapsulates a license plate candidate.

6.4.1 Fields

6.4.1.1 elements

Information about the individual chars that make up the license plate. They are listed in the same order as they appear in the [text](#) string. To know the physical layout use the [bbox](#) field.

6.4.1.2 brightBackground

True if the license plate features dark text on a light background. False if otherwise.

6.4.1.3 confidence

Overall 'goodness' of the recognition. Currently it is calculated as the minimum goodness value of all individual characters in the license plate. See [confidence](#).

6.4.1.4 country

Country code string.

6.4.1.5 text

Unicode representation of the license plate string. Separators are represented as white space.

6.5 Element

Encapsulates a candidate character in a license plate.



6.5.1 Fields

6.5.1.1 bbox

Bounding box of the character, in pixel coordinates.

6.5.1.2 confidence

'Goodness' of the recognition. Its range is 0 to 1 and can be used to rank candidates although it should not be regarded as a probability. In general, a 2x goodness value is not twice as good as x.

6.5.1.3 glyph

Unicode representation of the character.



7 SDK Files

Upon installation the *SimpleLPR 2.0 SDK* is organized in the following folder structure

```
SimpleLPR 2.0    % Root folder
├── EULA.rtf      % End user license agreement
├── bin           % SDK redistributable binaries
│   ├── libguide40.dll % Redistributable .dll
│   ├── SimpleLPR2.dll % Redistributable .dll
│   └── SimpleLPR2.xml % .NET documentation; do not redistribute
├── doc          % SDK documentation
│   └── SimpleLPR.pdf % This guide
├── include      % C++ interface
│   └── SimpleLPR.h % C++ interface include file
├── sample       % Sample applications
│   ├── SimpleLPR_Samples.sln % Visual Studio 2008 solution
│   ├── bin        % Samples in executable form
│   │   ├── libguide40.dll % Redistributable .dll
│   │   ├── SimpleLPR2.dll % SimpleLPR.dll
│   │   ├── SimpleLPR2.xml % .NET documentation; do not redistribute
│   │   ├── SimpleLPR_CPP.exe % C++ sample
│   │   ├── SimpleLPR_CSharp.exe % C# sample
│   │   └── SimpleLPR_UI.exe % C# sample with user interface
│   ├── SimpleLPR_CPP % C++ sample
│   ├── SimpleLPR_CSharp % C# sample
│   └── SimpleLPR_UI % User interface sample
```

