The STEMMER IMAGING Guide to writing platform agnostic applications with Common Vision Blox 2011

1.4.3

How to avoid awkward code when targeting Windows, Windows x64 and Linux x64 simultaneously

ABSTRACT: Common Vision Blox 2011 is the first version of Common Vision Blox to be available on more than just the Windows 32bit platform. Obviously this step necessitated some changes to the Common Vision Blox SDK itself, some of which are not completely backward compatible.
This document will show you how to use the adapted Common Vision Blox SDK to write code that will compile under 32 bit windows as well as under 64 bit windows or Linux.

WWW.STEMMER-IMAGING.COM · IMAGING IS OUR PASSION

GERMANY
STEMMER IMAGING GmbH
Gutenbergstr. 9-13
82178 Puchheim
Phone: +49 89 80902-0
Fax: +49 89 80902-116
info@stemmer-imaging.de

UNITED KINGDOM
STEMMER IMAGING Ltd.
The Old Barn, Grange Court
Tongham, Surrey GU10 1DW
Phone: +44 1252 780000
Fax: +44 1252 780001
info@stemmer-imaging.co.uk

FRANCE
STEMMER IMAGING S.A.S.
23 bis, rue Edouard Nieuport
92150 Suresnes
Phone: +33 1 45069560
Fax: +33 1 40991188
info@stemmer-imaging.fr

SWITZERLAND
STEMMER IMAGING AG
Rietbrunnen 48
8808 Pfäffikon SZ
Phone: +41 55 4159090
Fax: +41 55 4159091
info@stemmer-imaging.ch

The STEMMER IMAGING Guide to writing platform agnostic applications with Common Vision Blox 2011 1
How to avoid awkward code when targeting Windows, Windows x64 and Linux x64 simultaneously 1
1. About this guide 3
2. Windows x86 versus Windows x64 4
Layout of the Common Vision Blox Installation on Windows x64 4
The Wow6432 Folder 5
3. Differences between CVB 10.x and CVB 2011 7
C++ 7
A New Type System for CVB's DLL Interface 7
Platform-Polymorphic Types 8
Breaking Changes 8
.Net Languages 9
Shared Objects 9
VPAEntry 10
Node Call-backs and NodeMap Call-backs 10
ActiveX Controls 11
The Type Library Conundrum 11
The C++ Solution 11
The .Net Solution 12
4. How to Write Portable Code 13
Migration of an Existing Application from CVB 10.x on Win32 to CVB 2011 on Win64 13
Starting New Applications From Scratch 14
5. CVB on Lin64 16
Directory Structure 16
Differences Between CVB DLLs and CVB Sos 17
6. Questions and Answers 18
Appendix 19
Version History 19

1. About this guide

To avoid lengthy wordings we will use the following abbreviations throughout this text:

CVB as always is short for Common Vision Blox

Win32 designates the 32bit Windows platforms supported by Common Vision Blox (please refer to the Release Notes for a list)

Win64 designates the 64bit Windows platforms supported by Common Vision Blox (currently Windows Vista x64 and Windows 7 x64; XP x64 is not supported by Common Vision Blox)

Lin64 designates the 64bit Linux platforms supported by Common Vision Blox (currently Ubuntu 9.04 and 9.10)

The chapters of this guide may be read independently or in sequence. Their summarized contents are:

- **Chapter 2** gives a brief description of the differences between the Win32 and the Win64 installation of CVB (Linux will be covered in a later section).
- **Chapter 3** highlights the changes that have been made to the 2011 version of the SDK compared to the CVB 10.x releases.
- **Chapter 4** give guidance for writing code that will compile for Win32 as well as for Win64. This description will be split into the two typical scenarios (migration of an existing application to Win64 versus starting a new application from scratch) and treat C++ and the .Net languages.
- **Chapter 5** will describe the layout of the CVB installation on the Lin64 platform and gives general hints for using CVB on the Linux platform.
- **Chapter 6** answers a few Frequently Asked Questions

Because of the large base of existing CVB installations on Win32, this document focuses strongly on the migration path from Win32 to Win64. Yet, we encourage our customers to give CVB a try on the Lin64 platforms as well.

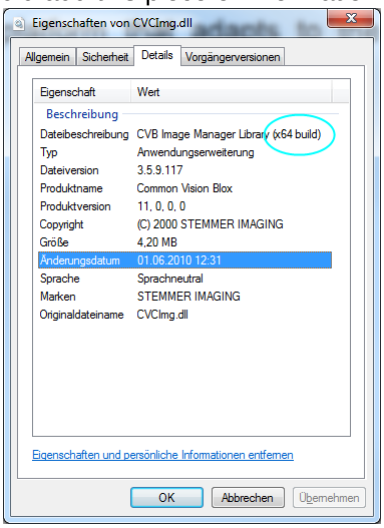
At the time of writing this document, the only programming environment officially supported by CVB on the Win64 platform is Visual Studio (2008 and 2010). Consequently this guide concerns itself exclusively with C++ and C# (with the latter having been selected as a typical representative of the .Net programming languages). Of course the concepts are easily transferable from C# to any other CLR languages.

2. Windows x86 versus Windows x64

Layout of the Common Vision Blox Installation on Windows x64

One of the goals when designing Common Vision Blox 2011 for Win64 was to make it as similar to the Win32 version as reasonably possible. Therefore, the %CVB% folder generated by the Win64 setup looks very similar to that on a x86 machine, and you will recognize many items you might know from the 32bit version of Common Vision Blox in their familiar locations. However, there are a few subtle differences:

- Of course the binaries installed on the Win64 machine have either been compiled for the AMD64/EMT64 architecture or (in case of .Net assemblies) for the "Any CPU" platform that adapts to the CPU architecture during runtime. To make identification of the platform for which our native binaries have been compiled easier, we did add this piece of information to the binary's version resource[1]:



- Several Tool DLLs and ActiveX controls are missing. Because some of our tools have been written in Delphi for which there is no native 64bit compiler available at the moment, we have decided to skip them in the Common Vision Blox 2011 x64 release[2]. This policy will be revisited once a native 64bit Delphi compiler has been released.
- A new folder, `Wow6432`, has been added under %CVB%, containing several ActiveX controls and DLLs and source code files. The contents of this folder help us bridge the gap between the 64bit world and Visual Studio (which is currently still a 32bit process). A detailed description of the purpose of these files will follow.

[1] Otherwise you'd have to use a utility like e.g. the Dependency Walker (http://www.dependencywalker.com) that can parse the binary file in order to identify the CPU architecture that has been targeted.

[2] This refers to the differences between the Common Vision Blox 2011 releases for the Win64 and Win32 platforms. There are in fact also a few files missing when comparing Common Vision Blox 2011 (x86) to Common Vision Blox 10.x - these have been left out after we decided to discontinue some of the tools in Common Vision Blox. Refer to the Common Vision Blox Release Notes for details.

- In the %CVB%\Lib directory only the header/wrapper files for C++ and .Net persist. The header files for Delphi and VB6 have been left out because these languages currently[3] do not support the generation of native 64 bit binaries. The .Net flavours of these languages are supposed to use the wrapper files located in %CVB%\Lib\Net.

The Wow6432 Folder

When developing for Win64 with Visual Studio[4] one has to keep in mind that the entire Visual Studio is still a 32 bit process. This has consequences for development as well as debugging, as 32 bit binaries and 64 bit binaries generally cannot be mixed in the same process.

To work around this 32/64 bit boundary during debugging Microsoft has effectively implemented the debugging of 64 bit binaries as remote debugging, i.e. the process that is being debugged and the debugger itself (inside Visual Studio) are being kept separate from each other.

For development the 32/64 bit boundary becomes important when working with components that are being used in the GUI designers of Visual Studio, namely ActiveX controls or User Controls, that are dependent on 64 bit binaries. This is the case with the CVB ActiveX controls: The .ocx files have been compiled for Win64 and therefore cannot be used directly in Visual Studio[5]. The workaround in this case consists of providing these components in two versions: One that has been compiled for Win32 (that will be visible and available in Visual Studio) and one that has been compiled for Win64 that will be used when running the application.

With CVB, of course the Win64 versions of our ActiveX controls are located in the %CVB% folder of your installation. While the folder %CVB%\Wow6432 folder contains their 32 bit counter parts and the dependencies required to instantiate them. So, whenever a CVB ActiveX control is dropped onto a form in the Visual Studio GUI designer, the control that is instantiated comes from the folder %CVB%\Wow6432.

It is important to note that the ActiveX controls in this folder are *not identical* with those installed on Win32[6]! Their type library has been modified to accommodate cross platform development (see chapters 3 and 4) and is different from that of our Win32 ActiveX controls. To emphasize this, a _wow64 has been appended to the file name. The ActiveX controls thus marked are only intended to be used at design time and are *not suitable to run a 32 bit binary on a Win64 operating system* properly!

In addition to the aforementioned ActiveX controls, the Wow6432 folder also contains a sub folder named "Visual C++ OCX Wrappers". This sub folder contains C++ wrappers for all the ActiveX controls that ship with the Win64 version of CVB. Those files are by and large almost identical to the wrapper files that Visual Studio generates when adding an ActiveX control to a MFC application and generating a member variable for it.

---

[3] in the case of Visual Basic 6: "definitely"

[4] Win64 support is available in Visual Studio ever since version 2005. The express Editions are limited in their capabilities to support development for Win64.

[5] In fact pure Win64 ActiveX controls do not even appear in the "Choose Toolbox Items" dialog of Visual Studio, as Visual Studio will only see the (Win32) controls registered under HKEY_CLASSES_ROOT\Wow6432Node, a location that is reserved for 32 bit COM components on the Win64 system.

[6] The DLLs, however, generally are.

However, the Visual Studio wizard responsible for the generation of these files does not generate proper Idispatch callers for properties or functions that use 64 bit numerical types. Therefore all the wrapper files generated by the “Add member variable” wizard will either need to be modified accordingly or be replaced by the files located in the “`Visual C++ OCX Wrappers`” folder. Chapter 4 shows the modifications in more detail.

3. Differences between CVB 10.x and CVB 2011

C++

A New Type System for CVB's DLL Interface

With CVB 2011 a new type system has been introduced for the DLL interface of Common Vision Blox, with the aim of being able to provide and use the same header files for the Win32, the Win64 and the Lin64 platform. Prior to CVB 2011, the header files for the CVB DLLs have mostly been using the compilers' built-in types like `int`, `long` etc. in the functions' signatures. This was working well enough because these built in types always had a fixed size and meaning on the Win32 platform.

With Win64 and Lin64 entering the image, things have gotten slightly more complex:

- Wherever we are looking at memory offsets, we now need a 64 bit value type on Win64, while on Win32 a 32 bit value type was and is still sufficient
- While C++'s `long` on Win64 still has the same meaning as on Win32, the compilers on Lin64 typicall treat `long` as a 64 bit integer value
- Some types that are generally available on the Win32 platform[7] and that have been used in CVB in the past are not usually defined on the Lin64 platform (e.g. `HRESULT`, `GUID`)

To compensate for these effects, we have come up with a type system that now replaces the use of the built-in types in the CVB header files. The significant definitions are all located at the beginning of `iCVCImg.h`, where preprocessor statements branch into the Win32/Win64[8] and Lin64 sections. The basic concept was to define and use the new types according to what they stand for instead of simply fixing the bit size of those types for the new old and new platforms:

Type	sizeof Win32	sizeof Win64	sizeof Lin64	Rationale
`cvbbool_t`	4	4	1	Used for irecto values and results. For backward compatibility this maps to `BOOL` on Win32 and Win64.
`Cvbdatatype_t`	4	4	4	Represents a CVB image plane's datatype identifier bit field as returned by the `ImageDatatype` function.
`Cvbdensity_t`	4	4	8	Represents a search density as used by the ScanImage/ScanPlane functions. Valid values range from 0 to 1000.
`Cvbdim_t`	4	4	8	Represents an x, y or z coordinate in pixels/planes.
`Cvblicres_t`	4	4	4	Result type for the functions `GetLicenseInfo` and `GetSerialNumber`.

[7] at least once <`windows.h`> has been included

[8] As a matter of fact thanks to the way the Visual C++ compiler uses its built-in types, these two operating systems can still use the same set of `typedef` statements.

Cvbres_t	4	4	4	Return type used by most CVB functions outside the CVCImg.dll. This type is interpreted similar to the Windows HRESULT: The most significant bit indicates an error, the remaining bits indicate the nature of the error (which is usually one of the values defined in CVCError.h).
cvbval_t	4	4	8	Generic integer value type where the actual amount of bits in the parameter is insignificant for the purposes where this type is being used.
Cvbint64_t	8	8	8	Integer value type that is guaranteed to be 64 bits wide.
Cvbuint64_t	8	8	8	Unsigned integer value type that is guaranteed to be 64 bits wide.
Cvbint32_t	4	4	4	Integer value type that is guaranteed to be 32 bits wide.
Cvbuint32_t	4	4	4	Unsigned integer value type that is guaranteed to be 32 bits wide.
Cvbint16_t	2	2	2	Integer value type that is guaranteed to be 16 bits wide.
Cvbuint16_t	2	2	2	Unsigned integer value type that is guaranteed to be 16 bits wide.
Cvbint8_t	1	1	1	Integer value type that is guaranteed to be 8 bits wide.
Cvbuint8_t	1	1	1	Unsigned integer value type that is guaranteed to be 8 bits wide.
Cvbguid_t	16	16	16	Regular Universally Unique Identifier.

Platform-Polymorphic Types

Of course there are occasions where it is necessary to have a type that is precisely 32 bits wide on the 32 bit platform and 64 bits wide on the 64 bit platforms – think e.g. of the VPAT, where the offset entries can continue to be 32 bits wide on Win32 but where they should rather have 64 bits on Win64 and Lin64.

Luckily, the C++ standard already provides us with these types once the header `stddef.h` has been included in the form of the types `size_t`, `uintptr_t` and `intptr_t`. Where necessary, we have substituted `long/int` by these standard types rather than defined our own version of them.

Breaking Changes

While we did manage to achieve binary compatibility[9] of the CVB DLLs between CVB 10.x and CVB 2011 on Win32, we did have to introduce some changes to function signatures that will break backward compatibility on

[9] That means the amount of bytes passed to and from the functions and their interpretations were preserved on the win32 platform. Consequently, an application built against CVB 10.x will continue to work with CVB 2011 if it does not use discontinued functionality.

the compiler level. In other words: Some applications will need to be modified slightly before they will compile without errors with the headers shipped with CVB 2011[10].

The affected functions are:

- `AnalyseXVPAT`, `AnalyseYVPAT` and `GetLinearAccess` (`iCVCUtilities.h`)

  These functions return memory increments by reference, and in response to the requirements of the 64 bit platforms the output parameters' types have changed from `long*` to `intptr_t*`. Please update the types used in your source code accordingly.

- Callbacks `TFLine`, `TFPixelUnary`, `TFPixelBinary` (`iCVCImg.h`)

  The return type of these callbacks has changed from `long` to `cvbbool_t` because the original use of `long` here was ill-considered as, internally, the return type was evaluated in terms of a irecto result. In order to pass the compiler's type checks, the callbacks defined in your code need to be switched to `cvbbool_t`.

.Net Languages

All the .Net DllImport wrappers for CVB have been recompiled for the CVB 2011 release using the "Any CPU" setting of the C# compiler. The aforementioned changes to the C++ DLL interface have been taken into account, therefore the newly built wrappers[11] will be identical on the Win32 and the Win64 release of CVB.

Furthermore, if you build a .Net application based on the CVB managed wrappers and built with the "Any CPU" setting, it will be possible to run the same assembly with the same wrapper DLLs on the Win32 as well as the Win64 platforms.

Shared Objects

One of the concepts customers frequently seem to struggle with when programming with CVB is that of reference counted objects. In CVB, objects that may potentially be used in several locations of an application and/or in several threads have been given a reference count that – when used properly – allows the object to determine its own lifetime and manage object disposal. Ill use of that reference count however will easily and invariably sooner or later lead to either a memory leak or to access violations.

The .Net runtime provides two features that allow us for the first time to offer objects that automatically handle their reference count properly:

- The CLR itself maintains a reference count of the managed reference objects (which is required for the garbage collector to do a reliable job)
- The CLR allows for the use of objects across programming language boundaries

[10] Note that if you want to stay on the Win32 platform it is not even a requirement to change type usage in your code to the new type system: The new types have been designed to be compatible with the type usage in CVB 10.x with the few exceptions mentioned here.

[11] The new wrappers may be easily identified by their major version number which has been promoted to "2".

We have made use of that and implemented several new objects that can manage their lifetime autonomously rather than relying on the user to call `ReleaseObject`/`ShareObject`. These objects may be used as a substitute for their "classic" counterparts as they are implicitly convertible. Where CVB functions take one of the old types (see table) as a `ref` or `out` parameter, an overload has been added to take its shared counterpart as a `ref` or `out` parameter.

Old "blunt" type	**new shared type**
`Cvb.Image.IMG`	`Cvb.SharedImg`
`Cvb.Image.OBJ`	`Cvb.SharedObj`
`Cvb.Image.PIXELLIST`	`Cvb.SharedPixelList`
`Cvb.GenApi.NODE`	`Cvb.SharedNode`
`Cvb.GenApi.NODEMAP`	`Cvb.SharedNodeMap`

We strongly recommend using the shared object types when starting a new application.

Note that the old types have changed as well: Prior to CVB 2011 they were designed as value types and have now be changed to be reference types. Where necessary, update your code accordingly.

VPAEntry[12]

As with C++, the components of the VPAEntry structure had to change from pure 32 bit values to platform-polymorphic IntPtr values. As you cannot calculate with IntPtr values directly, be prepared for using lots of typecasts where VPAT operations are involved and make sure to prefer `System.Int64` over `System.Int32` when doing so, as you might otherwise accidentally cast away the uppermost 32 bits of a 64 bit value.

Node Call-backs and NodeMap Call-backs

Not only have the new shared objects been introduced (see table previously in this chapter), but the underlying "blunt" types also changed: Previously they were value types, and starting with CVB 2011 they have become reference types. Generally this does not really affect customer's code because all the calls into unmanaged libraries have been adapted accordingly. However there is one situation for which no workaround exists that does not break backward compatibility and that is when the two call-back types `Cvb.GenApi.TFNode` and `Cvb.GenApi.TFNodeProc` are involved: To reproduce the correct behaviour we had on the value type versions of `Cvb.GenApi.NODE` and `Cvb.GenApi.NODEMAP` it was necessary to change the signature of those two call-backs to accept `System.IntPtr` parameters instead of `Cvb.GenApi.NODE` and `Cvb.GenApi.NODEMAP`.

Inside the callbacks, of course the respective objects are easily created with `new Cvb.SharedNode(IntPtr)` or `new Cvb.SharedNodeMap(IntPtr)` calls.

[12] Note that in the managed wrappers there is still a struct entry called `tagVPAEntry`. This type is only a remnant from the original C definition of the `VPAEntry` structure and is not to be used any longer.

ActiveX Controls

The Type Library Conundrum

When programming for Win64 and Win32 with the CVB ActiveX controls, you will stumble across one obstacle that will affect C++ and .Net programmers alike: Practically all CVB ActiveX controls at some point map handles to properties[13] or function parameters of type `long`. On the Win32 platforms this did not hurt, as handles and `long` are both 32 bits wide. For the Win64 platforms, however, handles are supposed to be 64 bits wide.

The best workaround for this was to use the `__int3264` type for these properties. When compiling the CVB ActiveX controls for Win32 this type maps to `long`, when compiling for Win64 it becomes an `__int64` in the ActiveX control's type library[14]. That way, the compiler generates ActiveX controls that have appropriate type libraries for their target systems, but unfortunately the type library needs to be different between the Win32 and the Win64 version of the ActiveX control.

This has consequences for developers using the ActiveX control because usually a development environment uses the ActiveX control's type library to generate a wrapper class for the object. If the type library differs between the Win32 and the Win64 then this means that the wrapper classes will differ as well – which makes it difficult to write portable code. To make things worse, the wizard that generates the C++ ActiveX wrapper classes cannot handle `__int64` properties correctly[15].

The C++ Solution

To alleviate the situation we have provided modified wrapper classes for C++, based on what the Visual Studio wizard generates. These files are located in the folder `%CVB%\Wow6432\Visual C++ OCX Wrappers`[16] (see also chapter 3). Simply replace the files that the Visual Studio wizard generated with these modified files and you will be able to use the same source for the Win32 and Win64 version of your application. You might need to modify the class names as the Visual Studio wizard by default appends an ordinal number to the class name, which has been omitted in our modified wrappers.

[13] The most widely used example for these is probably the `Image` property of the CVB ActiveX controls.

[14] One might be inclined to use a type like `LPUNKNOWN` for the handles, however the MIDL-supported pointer types work under the assumption that they are pointing to an `IUnknown`-derived COM object, which is not the case with the CVB objects.

[15] The Wizard that generates the C++ ActiveX wrapper classes incorrectly tries to marshal `__int64` values as `VT_EMPTY` instead of `VT_I8` through the `IDispatch` interface. This will compile, but as soon as one of these properties is being accessed, it will raise a runtime error.

[16] These wrappers have been modified to use `intptr_t` wherever handles are involved. Conditional compilation is being used to make sure that the `IDispatch` marshalling is carried out correctly on Win32 and Win64.

The .Net Solution

When creating a .Net application, Visual Studio frequently re-generates the ActiveX wrappers that have been generated by `aximp` and `tlbimp`. Manually modifying these files is partly possible, but with Visual Studio redoing them on each project build this is a tedium.

Therefore we suggest another approach: When building your application on either Win32 or Win64 simply use the ActiveX wrappers that have been generated by Visual Studio as they are. This means that on Win32 the `Image` property will have type `System.Int32` and on Win64 the Image property will have type `System.Int64`. Of course this means that when you ship your application for Win32, you will need to ship it with the ActiveX wrappers generated on the Win32 platform – and likewise with Win64.

In your application you will typically convert between the Image property and variables of either `Cvb.Image.IMG` or (recommended) `Cvb.SharedImg`. If you route the Image property through a variable of type `System.IntPtr` you're on the safe side: You can write platform independent code and your assembly will be usable on Win32 as well as on Win64 – only the wrapper DLLs (`AxInterop.<Control Name>.dll` and `Interop.<Control Name>.dll`) will need to match the operating system. The easiest way to get a matching wrapper DLL for the system on which you are deploying is probably to copy them from any of the .Net Tutorials that come with CVB.

To sum it up: Rather than writing e.g.

```
public void ProcessImage(Cvb.Image.IMG img)
{
  …
}
private void cvImg_ImageUpdated(object sender, EventArgs e)
{
  ProcessImage(Cvb.Image.ToCvbIMG(cvImg.Image));
}
```

you should code like this:

```
public void ProcessImage(Cvb.SharedImg img)
{
  …
}
private void cvImg_ImageUpdated(object sender, EventArgs e)
{
  ProcessImage(new Cvb.SharedImg(cvImg.Image));
}
```

This short sample assumes that `cvImg` is an object of type `AxCVIMAGELib.AxCVimage`.

4. How to Write Portable Code

With the differences between the Win32 and Win64 outlined in chapter 3, the guidelines to writing portable code are fairly straightforward (in fact some of them boil down to a repetition of what has been said in chapter 3).

Migration of an Existing Application from CVB 10.x on Win32 to CVB 2011 on Win64

C++

The types used in the DLL interface are practically identical between Win32 and Win64 (of course with the pointer types being 64 bits wide, but that is handled implicitly as long as no conversion between pointer types and integer types happens in your code)[17]. So for the DLL interface all that needs to be done is to adapt your code to the breaking changes that have been made to the DLL interface (see section "Breaking Changes" in chapter 3).

With the ActiveX controls, there is the added complication of the differing type library (and therefore the need for different wrapper classes on the different architectures). *You should, at any rate, replace the C++ wrappers for ActiveX controls* that have been generated for Win32 with the ones found in the "`%CVB%\Wow6432\Visual C++ OCX Wrappers`" folder: If you don't, the old Win32 wrappers will only marshal 4 bytes for each Image pointer through the OCX's `Idispatch` interface without even raising an error[18]. This will work for quite a while (while memory consumption in a process is still low, the upper 32 bits of pointers on Win64 tend to be zero…), but sooner or later this will lead to intermittent problems that will be very challenging to debug.

Once these two changes have been made to a project, it is ready for Win64 compilation. Transfer it to your Win64 development machine if you have not already done so and add "x64" debug and release configurations to the Visual Studio project (you can simply base all the settings on the corresponding "Win32" platform configurations). Note that even though you can run the Win32 build result of some applications on the Win64 platform this is not a supported configuration (see chapter 6)!

It is not strictly necessary to adapt your project to the new type system outlined in chapter 3 for it to compile and run on Win64. If, however, you want to target Lin64 at some point, a switch to the new type system will be inevitable.

.Net

With the .Net languages, migrating a project to CVB 2011 and the Win64 platform is no more difficult than with C++. The first step will be to add the newly built DllImport wrappers to your project references[19]. Doing so will

[17] See table in the C++ section of chapter 3.

[18] The only hint at the perilous situation are a few level 4 compiler warnings and 64 bit portability warnings - level 4 warnings are by default disabled and in the latest Visual Studio also the 64 bit portability warnings are no longer enabled by default.

[19] Without this step, compilation will not be possible anyway because CVB 2011 no longer ships with the wrapper DLLs of the 10.x releases.

make the compiler point out the locations of breaking changes to you, so fixing those will be fairly straightforward.

What requires more attention is the handling of the ActiveX controls. Make sure you do the port on a Win64 development system – otherwise the ActiveX wrappers will be built versus the wrong (Win32) type library. Remove your projects' `obj` and `bin` folders – this will force Visual Studio to rebuild the ActiveX wrapper DLLs. Note that ActiveX Control usage will not work when using .Net 4.0 as the targeted framework version (for details see the "starting from scratch" section)! When using C#, once the ActiveX wrappers have been recreated all problematic property assignments from 64 bit integer values to 32 bit integer values will be pointed out by the compiler (C# does not support automatic conversion from `System.Int64` to `System.Int32`). VB.Net unfortunately is less strict here, and it is highly recommended to search your code for such assignments and correct them[20].

Consider upgrading your code to use the new shared object types introduced in CVB 2011. For portability this is not a requirement, but these types go a long way to helping you make sure the CVB reference count is being used properly.

If you migrated your application to Visual Studio 2010 and want to target the .Net Framework 4.0 there is yet another pitfall waiting for you: By default, the ActiveX wrappers generated by Visual Studio 2010 with the .Net Framework 4.0 toolset will be specific to the platform of the ActiveX control for which they have been generated (in our case, the "_wow" versions of the CVB ActiveX controls). In other words: The thus generated wrappers will be targeting the Win32 platform and when trying to load them into a Win64 application, the .Net Runtime will throw a `System.BadImageFormatException`. In order to coerce Visual Studio 2010 into generating "Any CPU" wrappers with the .Net Framework 4.0 toolset (namely `aximp` and `tlbimp`) it is necessary to manually modify the csproj or vbproj file[21]:

- Locate all the `PropertyGroup` nodes containing a `Condition` entry of the form `'$(Configuration)|$(Platform)'=='Release|AnyCPU'` (in a typical project there would be 2 of these nodes: One for `Debug|AnyCPU` and one for `Release|AnyCPU`).
- For all those nodes where the condition states `AnyCPU` add an entry `<PlatformTarget>AnyCPU</PlatformTarget>`

With these changes applied, the automatically generated ActiveX wrappers will be loadable into an AnyCPU application running on Win64. The same change is also required for C++/CLI projects, which, by default, always target the .Net Framework 4.0.

Starting New Applications From Scratch

C++

When starting a new application, using the newly introduced type system in the DLL interface is recommendable as this will make ports to other platforms a lot easier. When using ActiveX controls, do not forget to replace the

[20] Runtime errors arising from a 64 bit to 32 bit assignments can be very hard to identify later on...

[21] We recommend using a decent xml editor like "XML Notepad 2007" for the changes described here.

faulty C++ wrappers generated by Visual Studio with the ones provided in "%CVB%\Wow6432\Visual C++ OCX Wrappers".

In case you are planning on targeting Lin64 later on it is a good idea to either use a GUI toolkit that is available for both platforms right from the beginning or to keep your user interface code as separate from your image processing code as possible to make the necessary switch to a different GUI toolkit easier. ActiveX controls are a technology that is not available on the Lin64 platform and should therefore not be used at all in applications that are intended to be portable. Note that for C++/CLI projects in Visual Studio 2010 the AnyCPU tweak for aximp and tlbimp described in the migration section must also be applied.

.Net

With newly created .Net applications there is actually not much attention to be paid to Win32/Win64 portability. The main issue here is to keep track of which platform the ActiveX wrappers have been created on[22] (if your application uses ActiveX controls); a mismatch however will immediately lead to an exception, is therefore fairly easy to identify and will probably never go unnoticed. Keeping the coding hints from chapter 3 in mind and using the new shared objects will help avoid improper reference count handling and (for VB.Net users) perilous assignments of 64 bit values to 32 bit variables.

If you use Visual Studio 2010 please note that the AnyCPU tweak described in the migration section must be applied to a project file before it will be possible to generate and run applications targeted to the .Net 4.0 framework.

[22] The development environment actually tries to point out this particular pitfall to you with warnings like "TlbImp : warning TI0000: Importing a type library into a platform agnostic assembly. This can cause errors if the type library is not truly platform agnostic."

5. CVB on Lin64

Directory Structure

When porting CVB to the Lin64 platform, one of the goals was to integrate CVB as harmonically as possible into this operating system, that looks and feels a lot different from Windows. So instead of re-using the directory tree that is built on the Windows platforms we decided to come up with a new structure that follows the File Hierarchy Standard v. 2.3[23].

Therefore, on a Lin64 system the CVB 2011 installation is located in the folder `/opt/cvb-11.0` and its subfolders:

Folder	Content
`/opt/cvb-11.0`	base folder of the CVB installation; empty by default except for the subdirectories listed below
`/opt/cvb-11.0/bin`	folder for standalone applications (e.g. configuration utilities) and compiled tutorials
`/opt/cvb-11.0/doc`	contains the documentation for CVB in compiled HTML format (requires a tool like kchmviewer, gnochm or xchm to read)
`/opt/cvb-11.0/drivers`	location for CVB's video interface drivers
`/opt/cvb-11.0/include`	contains the C++ header files for CVB (only C++ is supported on the Lin64 platform); add this location to your compiler's include paths
`/opt/cvb-11.0/lib`	folder holding the shared object files for CVB; this location has been added to the system's linker cache through the file `/etc/ld.so.conf/libcvb.conf` and should be added to your compiler's linker paths
`/opt/cvb-2011/tutorial`	holds the source code of the tutorial applications that ship with CVB for Lin64

To make version switches easier in the future, a symbolic link `/opt/cvb` pointing to `/opt/cvb-11.0` is added to the `/opt` irectory. All environment variables and references like the dynamic linker cache use this symlink instead of `/opt/cvb-11.0` (or whatever the names will be in the future). Whenever CVB is updated in the future, you may simply bend the symbolic link to the newer version.

In addition to the core CVB installation directory there are a few more folders that contain dependencies or configuration data:

Folder	Content
`/etc/opt/cvb`	location for driver configuration files and license information files
`/opt/genicam-1.1.2` `/opt/genicam` (symlink)	folder hierarchy of the GenICam library that is being used by CVB

[23] http://www.pathname.com/fhs/

Differences Between CVB DLLs and CVB Sos

The shared object files shipped with CVB for Lin64 contain almost the same set of functions and (more importantly) use the same function interface as the DLLs on the Windows platforms. Where functions have been left out, this happened simply because the necessary underlying infrastructure was not available on Lin64 (for example the function `ImageToDC` expects a Windows device context handle and is therefore not available).

One difference that immediately hits the eye of anyone coming from the Windows platform is the lack of a standardized resource format for the shared object files. As there are no resource fields in the shared objects to store version information in, versioning is handled differently on the Lin64 platform: The version information simply goes into the file name, so `libCVCImg.so.3.3.0` is version 3.3.0 of the `CVCImg.dll`. Usually, the thus fully versioned shared object file is accompanied by a set of symbolic links that drop the version information step by step: `libCVCImg.so.3.3`, `libCVCImg.so.3`, `libCVCImg.so`. That way, an application can decide how much version information should be used when linking against a shared object (if e.g. an application links against the symlink `libCVCImg.so.3` and this link is later modified to refer to `libCVCImg.so.3.5.0` the application will automatically use the newer shared object file). With CVB's shared objects it should by the way typically not be necessary to preserve more than the major version number as the function interface does not change frequently and usually not in a code breaking way.

This Linux-like mechanism of preserving version information is of course not very robust against inadverted modification. To provide a more robust approach as a backup, all the CVB shared objects export a hidden `GetVersion` function that returns the version the shared object had at compile time. This information may be accessed using the `GetFileVersion` function from `libCVCUtilities.so`.

6. Questions and Answers

Following below is a selection of questions and answers concerning the Win64 and Lin64 port of CVB. The questions are in no particular order and the list will grow over time. If you have additional questions you would like to see answered please feel free to mail them to support@stemmer-imaging.de.

Q: Is it possible to install CVB for Win32 on a Win64 platform, possibly in parallel to the Win64 version of CVB so that it is possible to run a 32bit and 64bit build of a CVB-based application on the same operating system?

A: This constellation is not officially supported. The Win32 version of CVB uses several device drivers (the CVB.sys for enumerating PCI devices, the siFilterIP.sys that implements the GigE Vision Filter Driver for CVB, the device drivers that are being used by each of the .vin drivers…) that are not compatible with the Win64 environment. Therefore, while it is generally possible to run all the Win32 binaries of CVB on a Win64 operating system, this would be of limited use because a Win32 CVB application running on a Win64 operating system will not be able to speak to hardware[24].

Also to accommodate the Win32 and the Win64 version of CVB being present on the same operating system we would have had to make several changes to the way CVB is installed on a computer in order to prevent clashes between the Win32 and Win64 binaries of CVB in the DLL-Loader. Given the limited usefulness of such a parallel installation we decided against these changes and instead kept CVB 2011 as similar as close to CVB 10.x.

[24] Currently the only exceptions being the GigE Vision Socket Driver and the IDS Camera Drivers.

Appendix

Version History

Rev. & Date	Who	Comments
1.0.0 (08.06.2010)	VGi	Initial draft. Chapters 4 and 5 to be done.
1.0.1 (09.06.2010)	SDo	Proof-reading and minor corrections.
1.1.0 (21.06.2010)	VGi	Revision of existing material. Added Chapters 4 and 5.
1.2.0 (24.06.2010)	VGi	Added comments about C++/CLI and the Visual Studio 2010 toolkit.
1.3.0 (25.06.2010)	VGi	Corrected the C++/CLI on Visual Studio 2010 section and extended the tweak to C# and VB.Net.
1.3.1 (01.07.2010)	VGi	Modified FAQ entry.
1.4.0 (12.07.2010)	VGi	Added section about VPAEntry in .Net.
1.4.1 (30.07.2010)	VGi	Added comment about changes to callback signatures in iCVGenApi.dll.
1.4.2 (28.03.2011)	VGi	Updated references to Common Vision Blox 2010 to Common Vision Blox 2011 for consistency.
1.4.3 (29.03.2011)	VGi	Changed “CVB User Guide” to Release Notes.

We hope that these notes were useful for you and look forward to your feedback. In case of further questions, please do not hesitate to contact our technical support.

Phone: +49 089 80902-200
E-Mail: support@imaging.de.

Additional information as well as frequently asked questions and a lot of valuable details regarding image processing, can also be found on our website: www.stemmer-imaging.de or www.commonvisionblox.com (menu *Service -> FAQ).*

Yours sincerely - STEMMER IMAGING Technical Team