

**Creating C# Wrappers for
Intel[®] Integrated
Performance Primitives
Using Microsoft .NET*
Interoperability Mechanisms**



This document as well as the software described in it is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

MPEG is an international standard for video compression/decompression promoted by ISO. Implementations of MPEG CODECs, or MPEG enabled platforms may require licenses from various entities, including Intel Corporation.

Celeron, Dialogic, i386, i486, iCOMP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetStructure, Intel Xeon, Intel XScale, Itanium, MMX, MMX logo, Pentium, Pentium II Xeon, Pentium III Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2003 Intel Corporation.

Table of Contents

Executive Summary	4
Intel® IPP Overview	4
Microsoft .NET Framework Overview	4
Hardware and Software Requirements.....	4
Important .NET Terminologies	5
Bridging Managed and Unmanaged Code in .NET.....	7
Using IPP and .NET InteropServices Mechanisms.....	7
COM Interoperability	11
Conclusion	14
Appendix.....	15
References	17

Executive Summary

The audience of this paper are software architects and developers who intend to use the Intel® Integrated Performance Primitives (Intel® IPP) software library in the Microsoft .NET* framework environment.

This document provides information on Microsoft .NET framework interoperability mechanisms to create C# wrapper functions for using Intel IPP C-style libraries from a .NET framework application using the C# language. It is assumed that the reader is familiar with Intel IPP [1] and the Microsoft .NET framework concepts such as common language runtime (CLR), framework class libraries (FCL), assemblies, metadata, custom attributes and tools such as Microsoft Visual Studio* .NET, ILDASM, ILASM, etc.

Intel® IPP Overview

Intel® Integrated Performance Primitives are a cross-platform software library which provides a range of library functions for multimedia, audio codecs, video codecs (for example, H.263, MPEG-4), image processing (JPEG), signal processing, speech compression (G.723, GSM-AMR), computer vision, as well as math support routines for such processing capabilities. Intel IPP is optimized for the broad range of Intel® microprocessors: Intel® Pentium® 4 processor, Intel® Itanium® 2 processor, Intel® Xeon™ processors, and Intel® PCA application processors based on the Intel® XScale™ microarchitecture. This enables ease of porting and migration of software applications between different processor platforms.

Intel IPP functions can be used from the .NET framework managed environment to speed up the performance of applications on Intel® platforms.

Microsoft .NET Framework Overview

Microsoft .NET framework is a managed run time environment for developing applications, by using a common language runtime (CLR) layer and class libraries. This layer consists of runtime execution services such as garbage collection for memory safety, remoting, security, etc. The benefit is to quickly develop various types of software applications like windows forms, XML web services, distributed applications, media applications, imaging applications and others.

A managed application can be written in any language, which has a .NET compliant compiler that compiles the source code into .NET assemblies. The .NET compliant compilers must follow Common Language Specification (CLS) and Common Type System (CTS), which are subsets respectively of common functionality and types, among all languages. These specifications enable type-safety and cross-language interoperability. Simply stated, an object written in one .NET compliant language, for example, VB.NET, can be invoked from another object written in another .NET compliant language, for example, C#.

Hardware and Software Requirements

This section gives a list of the recommended hardware and software configuration for gaining optimal performance on Intel® processor platforms using Intel IPP in the .NET framework environment.

Hardware:

Intel® Pentium® III or Pentium® 4 Processor

Software:

Microsoft Windows* 2000 or Windows* XP

Microsoft Visual Studio* .NET RTM (2002) or "Everett" (2003) or later

Intel IPP Version 3.0 or later [1]

Important .NET Terminologies

Managed code and metadata

Managed code is an image created when the source code is compiled using a .NET framework compliant compiler to create a .NET managed assembly. .NET managed assembly can either be a DLL or a portable executable (PE) file. The managed DLL or PE file contains Microsoft intermediate language (MSIL) code and metadata. Metadata is the information used by the CLR to guarantee security, type-safety, and memory-safety for code execution. Managed code is tightly bound to the execution services provided by CLR and exposes the functionalities for the programmer to quickly develop robust and scalable applications.

Managed data

Managed data is the data under control of the .NET garbage collection service.

Unmanaged code

Unmanaged code is an image created when the source code is compiled using a native compiler to create a native binary for the underlying microprocessor. This code does not utilize any service provided by the CLR, although both managed and unmanaged codes can run together in the same managed runtime environment. Intel IPP library APIs are examples of C-style unmanaged code.

Unsafe code

C# code that uses pointers is called **unsafe code**. It is required to use the `unsafe` keyword as a modifier for the callable members such as properties, methods, constructors, classes or any block or code. Unsafe code is a C# feature for performing memory manipulation using pointers. It is typically used when managed code calls into unmanaged code like C or C++ code or a COM interface that requires pointers as arguments. Although the unsafe code consists of pointers, it still executes under the hood of the CLR, but it is not managed by the garbage collection (GC) memory management service. C# allows casting of pointers from one type to another. VB.NET does not support pointers, hence unsafe code cannot be written using VB.NET.

Note: Unsafe code should be compiled using the `/unsafe` compiler switch.

Note: When garbage collection sweeps through the memory, it moves objects around for memory re-compaction. If managed code calls into unmanaged code, use the `fixed` keyword to pin the memory to avoid moving the managed object being used when unmanaged code is being called.

Marshalling

The .NET framework has defined a common type system (CTS), which supports a set of data types used in most modern programming languages. Explicit marshalling should be performed when arguments and return values exchanged between managed and unmanaged code have a non-value type data representation [11]. The .NET framework provides an interoperability marshaller to convert data between managed and unmanaged environments.

Isomorphic data types

Data types that have similar data representation in both managed and unmanaged code are called isomorphic data types. The marshalling is automatic and does not need any special handling or conversion by the programmer when the data is passed between managed and unmanaged code.

Non-isomorphic data types

Data types that have different data representations in managed and unmanaged code are called non-isomorphic data types. If non-isomorphic data types are used in unmanaged code, marshalling is required to exchange data in the correct formats that can be understood by managed and unmanaged code. Some of the most commonly marshaled data types are listed in the following table.

Managed	Unmanaged
Boolean	BOOL, Win32 BOOL or Variant
Char	CHAR or Win32 WCHAR
Object	Variant or Interface
String	LPStr, LPWStr or BStr
Array	SafeArray

`MarshalAs` attribute and `Marshal` Class in `System.Runtime.InteropServices` namespace can be used for marshalling data between managed and unmanaged code. COM supports three types of marshalling – automatic marshalling, standard marshalling and custom marshalling. Details of COM marshalling are beyond the scope of this document [12].

Bridging Managed and Unmanaged Code in .NET

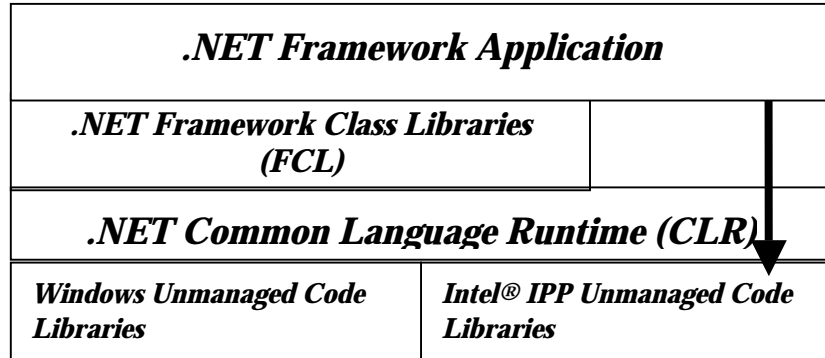


Figure 1

Interoperability between managed and unmanaged code

.NET supports communication between managed and existing unmanaged code through interoperability services as supported in .NET's `System.Runtime.InteropServices` namespace. Interoperability can be done from both .NET to unmanaged environment and from unmanaged to .NET environment.

Using Intel IPP and .NET InteropServices Mechanisms

The Intel IPP library is a C-style unmanaged code library. The following sections provides information on different .NET interoperability mechanisms with simple examples to use Intel IPP APIs.

Platform Invoke (P/Invoke)

The .Net Platform Invocation Service, commonly known as PInvoke or P/Invoke, allows managed code to call C-style unmanaged code functions in DLLs. P/Invoke can be used in any .NET compliant language. It takes time and effort to use P/Invoke effectively. Hence it is important to become familiar with usage of `DllImportAttribute`, `MarshalAsAttribute`, `StructLayoutAttribute`, and their enumerations.

Tips:

- For better usability, combine frequently used Intel IPP functions or a set of logical functions together into a class. Every function in the class should be declared as static extern.
- If groups of data can be combined to make one chunky call to Intel IPP API, you can get better performance rather than making many chatty calls to Intel IPP API [6].

Note:

- P/Invoke does not support importing functions from a .lib static library. The unmanaged functions must be exposed by a DLL.

When a P/Invoke call is initiated in the managed code to call an unmanaged function in an unmanaged DLL, the steps taken by P/Invoke service are as follows:

1. Locates the DLL specified by `DllImportAttribute` either by searching in the working directory or in the directories and sub-directories specified in the `PATH` variable and loads the DLL into the process memory.
2. Finds the static function in DLL loaded in memory.
3. Pushes the arguments on the stack by performing marshalling, if required, using the `MarshalAsAttribute` and `StructLayoutAttribute`.
4. Enables pre-emptive garbage collection.
5. Gives control to the static unmanaged function and passes any data or pointer to it.

Example:

```
DllImportAttribute is used to call unmanaged C-style IPP functions in DLLs
[DllImport("ipps20.dll", EntryPoint="ippsCopy_16s")]
unsafe public static extern IppStatus ippsCopy_16s (short *pSrc, short *pDst,
int len);
```

Code Sample 1

```
[DllImport("ipps20.dll")] unsafe public static extern
    Ipp.IppStatus ippsCopy_16s( char* src, char* dst, int len );

    unsafe public static string Copy(string str)
    {
        if (str == null)
            throw new System.ArgumentNullException();

        char[] temp = new char[str.Length];
        fixed( char*pt=temp, ps=str )
        {
            ippsCopy_16s( ps, pt, str.Length );
        }
        return new System.String(temp);
    }
```

Working with structures

Many Intel IPP functions use structures as parameters. To pass the structures as parameters from the managed to the unmanaged environment, it is necessary to match the physical layout of managed code structure with the one in unmanaged code. Usually the structure is represented as a class in managed code.

1. **StructLayoutAttribute** is used to describe the structure as a class. A `LayoutKind` enumeration must be specified depending on what is being marshaled from the managed to unmanaged environment.

Example of an Intel IPP structure in unmanaged code:

`IppiPoint` is an Intel IPP structure used for storing the geometric position of an object.

```
typedef struct
{
    int x;
    int y;
} IppiPoint;
```

a. [StructLayout(LayoutKind.Sequential)] – This enumeration is used when a structure is being marshaled. The fields in the structure are laid out sequentially in the same order as they appear in unmanaged code.

Code Sample 2

```
using System;
using System.Runtime.InteropServices;

class IppiStructLayoutSequentialSample
{
    void IppiStructLayoutSequentialSample();
    [StructLayout(LayoutKind.Sequential, CharSet=CharSet.Ansi)]
    public struct IppiPoint {
        public int x;
        public int y;
        public IppiPoint ( int x, int y ) {
            this.x = x;
            this.y = y;
        }
    }; //struct IppiPoint
}

// class IppiStructLayoutSequentialSample
```

Constructor has been added to the structure to make creation of an `IppiPoint` variable easier.

The hidden Intel IPP structures are declared as follows:

```
[StructLayout(LayoutKind.Sequential)] public struct IppToneState_16s {};
```

b. [StructLayout(LayoutKind.Explicit)] – This enumeration is used when a structure or a union is being marshaled. The fields in the structure are laid out using the byte offsets specified by the user using the `[FieldOffset(n)]` attribute, where `n` is the offset from the object's starting memory location. Hence the user has the control to precisely position the fields in the structure.

Notes:

- It is required to define the `[FieldOffset(n)]` attribute when the `Explicit` enumeration is used.

- For a union, offset for every field should be zero

Code Sample 3

```
using System;
using System.Runtime.InteropServices;

class ippStructLayoutExplicitSample
{
    void ippStructLayoutExplicitSample();

    [StructLayout(LayoutKind.Explicit, CharSet=CharSet.Ansi)]
    public struct IppiRect
    {
        [FieldOffset(0)] public int x;
        [FieldOffset(4)] public int y;
        [FieldOffset(8)] public int width;
        [FieldOffset(12)] public int height;
    };
} // ippStructLayoutExplicitSample
```

2. MarshalAsAttribute

The .NET Common Type System (CTS) supports only a subset of the types supported by many languages. Unmanaged code supports many primitive data types as well as user-defined types that .NET do not support.

This attribute is used to marshal unmanaged type that is not supported by the .NET data types. The attribute converts physical structure from a .NET type to the unmanaged type. The .NET Value types such as `char`, `int`, `float` and `Boolean` do not require marshaling since most of the unmanaged compilers support these primitive data types.

The commonly used `UnmanagedType` values used with `MarshalAsAttribute` for marshalling are as follows:

- LPWSTR, LPStr, LPCTSTR, BStr, AnsiBStr, TBStr** – `System.String` [5] is a commonly used .NET reference type. It can be marshaled to any of the above commonly used unmanaged code types. For COM objects, string is marshaled to `BStr` by default.
- ByValTStr** – This is typically used for fixed length character arrays in a structure. It is very useful for Intel IPP functions.

Example:

```
[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)] public string
targetCpu;

[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Ansi)] public class
IppLibraryVersion {
    public int major;
    public int minor;
```

```
public int majorBuild;
public int build;
[MarshalAs(UnmanagedType.ByValTStr, SizeConst=4)] public string
targetCpu;
```

c. **ByValArray** – Just like `System.String`, `System.Array` is a reference type also. This value must be used to pass an array to unmanaged code. Precise length of the array must be also specified using the `SizeConst` parameter.

Example:

```
[MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)] public string targetCpu;
```

d. **Interface** – This is typically used when a `struct` or a class has to be marshaled as a COM interface.

COM Interoperability

COM Interoperability, commonly called as COM Interop, is another way of calling unmanaged code from managed code in .NET. Customers who are familiar with Microsoft COM, COM+ or MTS technologies can benefit from the information in this section [12].

Note: COM Interop consumes more processor cycles compared to using regular P/Invoke mechanism as explained in the earlier section.

This section gives an introduction to .NET COM Interop and tools provided by Microsoft for COM Interop. Introduction to COM and creating COM object wrappers for Intel IPP functions are beyond the scope of this document.

Some differences between a COM object and the .NET assembly that are relevant to this document are as follows:

1. **Managed and unmanaged:** COM object is unmanaged code, whereas .NET assembly is managed code. .NET assembly consists of Microsoft intermediate language code (MSIL) and metadata.
2. **Type information:** COM object holds its type information in the registry, whereas the .NET assembly holds the type information in the form of **metadata**. Hence COM object has to be registered on the local machine.
3. **IUnknown interface:** COM object consists of `IUnknown` interface, whereas the .NET assembly does not have the concept of `IUnknown` interface.

Calling Intel IPP wrapped COM object from managed code in .NET

It is necessary to create a bridge between COM objects and .NET managed environment for the managed code to call into a COM object. This bridge is called a **Runtime Callable Wrapper** (RCW).

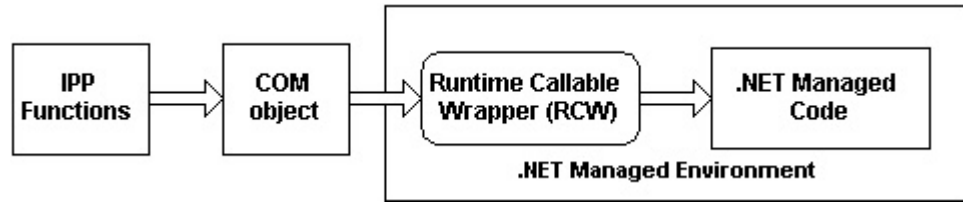


Figure 2: .NET to COM Runtime Callable Wrapper (RCW)

Runtime Callable Wrapper is a .NET assembly (DLL) that contains a proxy wrapper to the COM object in the .NET environment. An RCW can be created from a COM DLL either by using the Visual Studio .NET IDE, by using the *tlbimp.exe* command line tool, or programmatically using the classes provided in `System.Runtime.InteropServices` namespace in the .NET framework. This new DLL will contain the metadata that encapsulates all the information such as the COM type libraries and argument marshalling required by .NET environment. Unlike P/Invoke mechanism, there is no need to create any more function prototypes for using the RCW. The .NET application can use this DLL just like any other .NET assembly.

The sub-sections below show multiple ways to create RCW .NET assembly (DLL) from a COM DLL. It is assumed that COM object has already been created for Intel IPP functions and is available on the local machine.

Creating .NET RCW DLL using Visual Studio .NET IDE

The steps to create a .NET RCW DLL from a COM DLL are as follows:

1. Select the project in the Solution Explorer bar.
2. Right click on the References and select Add Reference or select Add Reference menu item in the Project pull down menu in the IDE.
3. Select the COM tab in the Add Reference window.
4. Press the Browse button and select the COM DLL for which you want to create a .NET RCW DLL.
5. Make sure that the COM DLL is in the Selected Components and press the OK button.

These steps create a .NET assembly in the `obj` sub-directory under the project directory by default. The default naming convention followed by .NET for the newly created Interop .NET assembly is `Interop-<COM Object>.DLL`.

Creating .NET RCW DLL using *tlbimp.exe* tool

The *tlbimp.exe* is a command line tool provided by the .NET framework to create a .NET assembly from a COM object. It creates an equivalent metadata for COM type library in the .NET assembly.

Example: If you have an Intel IPP wrapped COM DLL called `ippsCopyCOM.DLL`, you can create a .NET RCW DLL using VS .NET Command Prompt as follows:

```
> tlbimp ippsCopyCOM.DLL /out:Interop-ippsCopyCOM.DLL
```

Creating .NET RCW DLL programmatically

The `System.Runtime.InteropServices` namespace in .NET framework provides classes that can be used for accessing COM objects from .NET by creating .NET RCW DLLs. The

`System.Runtime.InteropServices.TypeLibConverter` class and its method `ConvertTypeLibToAssembly` can be used to develop a customized utility or tool to create .NET RCW DLLs from COM DLLs.

Note: The garbage collection in .NET does not manage the COM objects in RCW assemblies. Hence the COM object will continue to reside in memory after its reference is set to null. The COM object can be released by calling the `Marshal.ReleaseComObject` on the RCW. Another way to release the COM object is to force the garbage collector by calling the `System.GC.Collect` method.

Early binding

Early binding is a process that takes place at compile time when an object is assigned to an object variable and the COM object's type definitions can be made available to the client.

After a .NET RCW DLL is created from a COM DLL, early bound activation is similar to any other .NET assembly in a C# program. The steps for using early binding are as follows:

1. Add Reference to the .NET RCW DLL in VS .NET project or if the code is compiled on command line, then reference the RCW DLL.
2. Import the DLL into the C# file using the **using** keyword.

Example: **using** Interop-ippCopyCOM

3. Instantiate the COM object like any other .NET class in a .NET assembly

Example: `ippCopyCOM.myIPPClass myIPPObj;`
`myIPPObj = new ippCopyCOM.myIPPClass();`

Late binding

Late binding is used in scenarios such as a web application on shared hosts where the interface is not known during compile time and will be made available at runtime.

Reflection namespace should be used for late binding to enable programmatic access to the types in any .NET assembly. The steps for using late binding are as follows:

4. Import the metadata to get the type information from the COM object using the `Type` class.

Example:

```
Type IppCopyType = Type.GetTypeFromProgID("IppProject.IPPCopyCOMObject");
```

This returns a `Type` object based on the COM object's unique ProgID.

1. Create an instance using the obtained `Type` object using the `System.Activator` class.

Example: `System.Activator.CreateInstance(IppCopyType)`

2. Invoke the method using the `InvokeMember` method in `Type` class.

Example: `IppCopyType.InvokeMember("IppsCopy", ...)`

Note: It may be necessary to create arguments before step 3 above and pass it as arguments to `InvokeMember` function for the function being invoked.

Tip: Although late binding provides flexibility to bind with the interface during runtime, there is a performance loss. Early binding provides better performance than late binding because it allows the compiler to optimize the code for faster execution.

Conclusion

Intel IPP functions can be used from a managed runtime environment like Microsoft .NET framework. This paper shows an example of how to use Intel IPP functions from a managed environment.

Appendix

P/Invoke Code Samples

```
using System;
using System.Runtime.InteropServices;

namespace ipp {

    public enum IppStatus {
        ippStsNoErr = 0,
    };

    // some IPP functions, for example FFT, have option to be faster or accurate
    public enum IppHintAlgorithm {
        ippAlgHintNone = 0,
        ippAlgHintFast = 1,
        ippAlgHintAccurate = 2,
    };

    // hidden IPP structure
    [StructLayout (LayoutKind.Sequential)] public struct IppsFFTSpec_R_32f{};

    // class of signal processing sample
    unsafe public class sp {
        // print data to be sure the results are ok
        internal static void print(float[] buf) {
            for(int i=0; i<buf.Length; i++) Console.Write("{0:F1} ",buf[i]);
            Console.WriteLine();
        }

        // name of IPP SP dispatcher DLL
        internal const string lib = "ipps20.dll";
        // declare the functions the example uses
        [DllImport(lib)] public static extern
        IppStatus ippsTone_Direct_32f(float*pDst,int len,float magn,
float rfreq,float*pPhase,IppHintAlgorithm hint);
        [DllImport(lib)] public static extern
        IppStatus ippsFFTFree_R_32f(IppsFFTSpec_R_32f*pFFTSpec);
        [DllImport(lib)] public static extern
        IppStatus ippsFFTFwd_RToCCS_32f (float*pSrc,float*pDst,
IppsFFTSpec_R_32f*pFFTSpec,byte*pBuffer);
        [DllImport(lib)] public static extern
        IppStatus ippsFFTInitAlloc_R_32f(IppsFFTSpec_R_32f**pFFTSpec,
int order,int flag,IppHintAlgorithm hint);
        [DllImport(lib)] public static extern
        string ippGetStatusString(IppStatus st);
        // to process bad argument exceptions
        internal class IppException : Exception {
            string funcName;
            IppStatus status;
        }
    }
}
```

```

        public IppException(string fname,IppStatus st) {
funcName = fname; status = st; }
        override public string Message { get {
return funcName+": "+ippGetStatusString(status); }}
    }
    // signal processing sample
    public static void Main()
    {
        const int ord = 2, len = 1<<ord;
        float[] pPhase = {0};
        float[] pSrc = new float[len], pDst = new float[len+2];
        IppStatus st;
        try {
            // generate input data
            fixed(float*pbuf=pSrc,pphase=pPhase) {
                st = ippsTone_Direct_32f (pbuf,len,1,1.0f/len,
pphase,IppHintAlgorithm.ippAlgHintNone);
                if(IppStatus.ippStsNoErr!=st)
throw(new IppException("Tone",st));
            }
            print(pSrc);
            // FFT transform
            IppsFFTSpec_R_32f*spec;
            st = ippsFFTInitAlloc_R_32f (&spec,ord,2,
IppHintAlgorithm.ippAlgHintFast);
            if(IppStatus.ippStsNoErr!=st)
throw(new IppException("FFTInit_R",st));
            fixed(float*psrc=pSrc,pdst=pDst) {
                st = ippsFFTFwd_RToCCS_32f (psrc,pdst,spec,null);
                if(IppStatus.ippStsNoErr!=st)
throw(new IppException("FFTFwd_R",st));
            }
            print(pDst);
            ippsFFTFree_R_32f(spec);
        }
        catch(IppException e) { Console.WriteLine(e.Message); }
    }
}
}

```

Note: in the above code, managed data and unmanaged data is used. We do not free the input and output arrays, and we free the memory allocated by the InitAlloc function for the FFT context structure. Usually, a special interface file is used to create C# interface to all Intel IPP functions, in particular, ipps.cs for the signal processing domain. In this example the prototypes of Intel IPP types are declared explicitly to make the example self-contained. Most of the special things a developer could encounter or needs to know are given in the example – declaration of Intel IPP types, hidden Intel IPP structures and functions; test of the status returned by Intel IPP function; pinning pointers to data in the fixed section.

References

- [1] Intel IPP:
<http://www.intel.com/software/products/perflib/>
- [2] IPP Overview
<http://www.intel.com/software/products/ipp/ipp30/overview.htm>
- [3] Intel IPP Download:
<http://www.intel.com/software/products/ipp>
- [4] Intel IPP Installation help:
<http://support.intel.com/support/performance/tools/libraries/ipp/ia/win/install.htm>
- [5] Default Marshalling for Strings (MSDN)
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpcondefaultmarshalingforstrings.asp>
- [6] Performance Tips and Tricks in .NET Applications
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dotnetperftips.asp>
- [7] .NET Interop: Get Ready for Microsoft .NET by Using Wrappers to Interact with COM-based Applications
<http://msdn.microsoft.com/msdnmag/issues/01/08/interop/default.aspx>
- [8] .NET Platform Invoke Services
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmxspec/html/vcmg_part2start.asp
- [9] Accessing COM objects from the Runtime
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmxspec/html/vcmg_AccessingCOMObjectsfromtheRuntime.asp
- [10] Creating Prototypes in Managed Code
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcreatingprototypesinmanagedcode.asp>
- [11] Interop Marshalling
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconinteropmarshaling.asp>
- [12] COM Interop
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconadvancedcominterop.asp>
- [13] Recommended tool: Dependency Walker
<http://www.dependencywalker.com/>
- [14] Natham, Adam. *.NET and COM – The Complete Interoperability Guide*.