Intel® Integrated Performance Primitives for Linux* OS

User's Guide

IPP 7.0

Document Number: 320271-005US

Legal Information

Legal Information

What's New

The User's Guide provides quick guidance of using the Intel(R) Integrated Performance Primitives (Intel(R) IPP) for all supported architectures on the Linux* OS.

The User's Guide documents updates of Intel IPP for 7.0 release, including new library structure, directory paths and names, and new names of the library files.

Getting Help and Support

Getting Technical Support

If you did not register your Intel software product during installation, please do so now at the Intel® Software Development Products Registration Center. Registration entitles you to free technical support, product updates and upgrades for the duration of the support term.

For general information about Intel technical support, product updates, user forums, FAQs, tips and tricks and other support questions, please visit http://www.intel.com/software/products/support/.

NOTE. If your distributor provides technical support for this product, please contact them rather than Intel.

For technical information about the Intel IPP library, including FAQ's, tips and tricks, and other support information, please visit the Intel IPP forum: http://software.intel.com/en-us/forums/intel-integrated-performance-primitives/ and browse the Intel IPP knowledge base: http://software.intel.com/en-us/articles/intel-ipp-kb/all/.

Introducing the Intel® Integrated Performance Primitives

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

Intel® Integrated Performance Primitives (Intel® IPP) is a software library that provides a broad range of functionality. This functionality includes general signal and image processing, computer vision, data compression, cryptography, string manipulation, audio processing, video coding, realistic rendering and 3D data processing. It also includes more sophisticated primitives for construction of audio, video and speech codecs such as MP3 (MPEG-1 Audio, Layer 3), MPEG-4, H.264, H.263, JPEG, JPEG2000, JPEG XR, GSM-AMR, G.723.

By supporting a variety of data types and layouts for each function, the Intel IPP library delivers a rich set of options for developers to choose from when designing and optimizing an application.

To speed up performance, Intel IPP functions are optimized to use all benefits of Intel® architecture processors. In addition most of the Intel IPP functions do not use complicated data structures, which helps reduce overall execution overhead.

Notational Conventions

The document uses the following font conventions and symbols:

Notational conventions

Italic	*Italic* is used for emphasis and also indicates document names in body text, for example: see *Intel IPP Reference Manual*
`Monospace lowercase`	Indicates filenames, directory names, and pathnames, for example: `/tools/ia32/perfsys/ps_ippi`
`Monospace lowercase mixed with UPPERCASE`	Indicates commands and command-line options, for example: `ps_ipps -f FIRLMS_32f -r firlms.csv`
`UPPERCASE MONOSPACE`	Indicates system variables, for example, `$PATH`
`monospace italic`	Indicates a parameter in discussions, such as routine parameters, for example, *`pSrc`*; makefile parameters, for example, *`function_list`*, and so forth. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example, <*`ipp directory`*>.
[items]	Square brackets indicate that the items enclosed in brackets are optional.
{ item | item }	Braces indicate that only one of the items listed between braces can be selected. A vertical bar (|) separates the items.

Contents

Intel(R) Integrated Performance Primitives Basics 1

Intel(R) Integrated Performance Primitives (Intel(R) IPP) has the following features:

- Intel IPP provides basic low-level functions for creating applications in several different domains, such as signal processing, audio coding, speech coding, image processing, video coding, operations on small matrices, and realistic rendering functionality and 3D data processing. See detailed information in the section Domains.
- The Intel IPP functions follow the same interface conventions including uniform naming rules and similar composition of prototypes for primitives that refer to different application domains. For information on function naming, see Function Naming.
- The Intel IPP functions use abstraction level which enables you to improve the performance of your application.

Intel IPP is well-suited for cross-platform applications. For example, the functions developed for IA-32 architecture-based platforms can be readily ported to Intel(R) 64 architecture-based platforms (see Cross Architecture Alignment).

Intel IPP for Linux* OS has the following versions:

- Intel IPP for the Linux OS on IA-32 architecture (including low power Intel(R) architecture)
- Intel IPP for the Linux OS on Intel(R) 64 architecture

Cross-Architecture Alignment

Intel IPP is designed to support application development on various Intel(R) architectures. This means that the API definition is common for all processors, while the underlying function implementation takes into account the variations in processor architectures.

By providing a single cross-architecture API, Intel IPP allows software application repurposing and enables you to port features across Intel(R) processor-based desktop, server, and mobile platforms. You can use your code developed for one processor architecture for many processor generations.

See also Dispatching.

Types of Input Data

Intel IPP operations are divided into groups according to the input data on which the operation is performed. Each group has its own prefix in the function name (see Function Naming - Data-Domain). The input data types are:

One-Dimensional Arrays and Signals

This group includes most functions operating on one-dimensional arrays of data. Generally these arrays are signals, and many of the operations are signal-processing operations. Examples of one-dimensional array operations include:

- vectorized scalar arithmetic, logical, statistical operations
- digital signal processing
- data compression
- audio processing and audio coding
- speech coding
- cryptography and data integrity
- string operations

Images

An image is an two-dimensional array of pixels. Their specific features distinguish them from general two-dimensional array. Examples of image operations include:

- arithmetic, logical, statistical operations
- color conversion
- image filtering
- image linear and geometric transformations
- morphological operations
- computer vision
- image compression
- video coding

Matrices

This group includes functions operating on matrices and vectors that are one- and two-dimensional arrays, and on arrays of matrices and vectors. These arrays are treated as linear equations or data vectors and subjected to linear algebra operations. Examples of matrix operations include:

- vector and matrix algebra
- solving systems of linear equations
- solving least squares problem
- computing eigenvalue problem

3D objects

This group includes functions operating with 3D objects. In this case input data depends on the used techniques. Examples of 3D operations include:

- realistic rendering
- resizing and affine transforming

Core Functions

There are several core functions that do not perform operations on one of these input data types. Examples of such operations include getting the type of CPU, aligning pointers to the specified number of bytes, controlling the dispatcher of the merged static libraries. These functions have their own header file, static libraries and SOs.

Code	Header file	Static Libraries	SO	Prefix in Function Name
ippCore	ippcore.h	libippcore_l.a libippcore_t.a	libippcore.so.x.x	ipp

Here x.x refers to the product version number, for example 7.0.

Domains

Intel IPP is divided into subdivisions of related functions. Each subdivision is called *domain* , and has its own header file, static libraries, SOs, and tests. These domains map to the types of input data and the corresponding prefixes. The Intel IPP Manual indicates in which header file each function can be found. The table below lists each domain's code, header and library names, and functional area.

Code of Domain	Header file	Static Libraries	SO	Prefix	Description
ippAC	ippac.h	libippac[_*].a	libippac[**].so.x.x	ipps	audio coding
ippCC	ippcc.h	libippcc[_*].a	libippcc[**].co.x.x	ippi	color conversion
ippCH	ippch.h	libippch[_*].a	libippch[**].so.x.x	ipps	string operations
ippCP	ippcp.h	libippcp[_*].a	libippcp[**].so.x.x	ipps	cryptography
ippCV	ippcv.h	libippcv[_*].a	libippcv[**].so.x.x	ippi	computer vision
ippDC	ippdc.h	libippdc[_*].a	libippdc[**].so.x.x	ipps	data compression
ippDI	ippdi.h	libippdi[_*].a	libippdi[**].so.x.x	ipps	data integrity
ippIP	ippi.h	libippi[_*].a	libippi[**].so.x.x	ippi	image processing

Code of Domain	Header file	Static Libraries	SO	Prefix	Description
ippJP	ippj.h	libippj[_*].a	libippj[**].so.x.x	ippi	image compression
ippMX	ippm.h	libippm[_*].a	libippm[**].so.x.x	ippm	small matrix operations
ippRR	ippr.h	libippr[_*].a	libippr[**].so.x.x	ippr	realistic rendering and 3D data processing
ippSP	ipps.h	libipps[_*].a	libipps[**].so.x.x	ipps	signal processing
ippSC	ippsc.h	libippsc[_*].a	libippsc[**].so.x.x	ipps	speech coding
ippVC	ippvc.h	libippvc[_*].a	libippvc[**].so.x.x	ippi	video coding
ippVM	ippvm.h	libippvm[_*].a	libippvm[**].so.x.x	ipps	vector math

* refers to one of the following: `l`, `t`.

** refers to processor-specific code, for example, s8.

`x.x` refers to the product version number, for example, 7.0

Function Naming and Parameters

Function names in Intel IPP use naming conventions to help you identify the different functions.

Intel IPP function names include a number of fields that indicate the data domain, operation, data type, and execution mode. Each field has a fixed number of pre-defined values.

Function names use the following format:

```
ipp<data-domain><name>[_<datatype>][_<descriptor>](<parameters>);
```

The elements of this format are explained in the sections that follow:

- Data-Domain
- Name
- Data Types
- Descriptor
- Parameters

Data-Domain

The *data-domain* is a single character indicating type of the input data. The current version of Intel IPP supports the following data-domains:

s	for signals (expected data type is a 1D array)

`g`	for signals of the fixed length (expected data type is a 1D array)
`i`	for images and video (expected data type is a 2D array of pixels)
`m`	for vectors and matrices (expected data type is a matrix or vector)
`r`	for realistic rendering functionality and 3D data processing (expected data type depends on supported rendering techniques)

The *core functions* in Intel IPP do not operate on one of these types of the input data (see Core Functions). These functions have `ipp` as a prefix without the data-domain field, for example, `ippGetStatusString`.

Name

The *name* identifies the algorithm or operation that the function does. It has the following format:

```
        <name> = <operation>[_modifier]
```

The `operation` field is one or more words, acronyms, and abbreviations that identify the base operation, for example `Set`, `DCTFwd`. If the operation consists of several parts, each part starts with an uppercase character without underscore, for example, `HilbertInitAlloc`.

The `modifier`, if present, denotes a slight modification or variation of the given function. For example, the modifier `CToC` in the function `ippsFFTInv_CToC_32fc` signifies that the inverse fast Fourier transform operates on complex data, performing complex-to-complex (CToC) transform. Functions for matrix operation have and object type description as a modifier, for example, `ippmMul_mv` - multiplication of a matrix by a vector.

Data Types

The `datatype` field indicates data types used by the function in the following format:

```
         <datatype> = <bit_depth><bit_interpretation>
```

where

```
        bit_depth = <1|8|16|32|64>
```

and

```
         bit_interpretation = < u|s|f>[c]
```

Here `u` indicates "unsigned integer", `s` indicates "signed integer", `f` indicates "floating point", and `c` indicates "complex".

For functions that operate on a single data type, the `datatype` field contains only one value.

If a function operates on source and destination objects that have different data types, the respective data type identifiers are listed in the function name in order of source and destination as follows:

```
 <datatype> = <src1Datatype>[src2Datatype][dstDatatype]
```

For example, the function `ippsDotProd_16s16sc` computes the dot product of 16-bit short and 16-bit complex short source vectors and stores the result in a 16-bit complex short destination vector. The `dstDatatype` modifier is not present in the name because the second operand and the result are of the same type.

Descriptor

The optional `descriptor` field describes the data associated with the operation. It can contain implied parameters and/or indicate additional required parameters.

To minimize the number of code branches in the function and thus reduce potentially unnecessary execution overhead, most of the general functions are split into separate primitive functions, with some of their parameters entering the primitive function name as descriptors.

However, general functions with large number of permutations may still have parameters that determine internal operation (for example, `ippiThreshold`).

The following descriptors are used in Intel IPP:

`A`	Image data contains an alpha channel as the last channel, requires C4 descriptor, the alpha channel is not processed.
`A0`	Image data contains an alpha channel as the first channel, requires C4 descriptor, the alpha channel is not processed.
`Axx`	Advanced arithmetic operations with `xx` bits of accuracy.
`C`	The function operates on a specified channel of interest (COI) for each source image.
`Cn`	Image data is made of `n` discrete interleaved channels (`n` = `1`, `2`, `3`, `4`)
`Dx`	Signal is `x` -dimensional (default is `D1`)
`I`	Operation is performed in-place (default is not-in-place).
`L`	For signal processing indicates that one pointer is used for each row in two-dimensional array. For matrix operation: function with layout description of the objects.
`M`	Operation uses a mask to determine the pixels to be processed.
`P`	F or signal processing: specified number of vectors to be processed. For matrix operation: function with pointer description of the objects.
`Pn`	Image data is made of `n` discrete planar (non-interleaved) channels (`n`= `1`, `2`, `3`, `4`) with a separate pointer to each plane.
`R`	Function operates on a defined region of interest (ROI) for each source image.
`S`	Function with standard description of the objects (for matrix operation)
`s`	Saturation and no scaling mode (default mode)
`Sfs`	Saturation and fixed scaling mode.

The descriptors in function names are always presented in alphabetical order.

Some data descriptors are implied as "default" for certain operations. Such descriptors are not added to the function names. For example, the image processing functions always operate on a two-dimensional image and saturate the results without scaling them. In these cases, the implied descriptors D2 (two-dimensional signal) and s (saturation and no scaling) are not included in the function name.

Parameters

The `parameters` field specifies the function parameters (arguments).

The order of parameters is as follows:

1. All source operands. Constants follow arrays
2. All destination operands. Constants follow arrays
3. Other, operation-specific parameters

The parameter's name has the following conventions:

- All arguments defined as pointers start with *p*, for example, *pPhase*, *pSrc*, *pSeed*; arguments defined as double pointers start with *pp*, for example, *ppState*; and arguments defined as values start with a lowercase letter, for example, *val*, *src*, *srcLen*.
- Each new part of an argument name starts with an uppercase character, without underscore, for example, *pSrc*, *lenSrc*, *pDlyLine*.
- Each argument name specifies its functionality. Source arguments are named *pSrc* or *src*, sometimes followed by numbers or names, for example, *pSrc2*, *srcLen*. Output arguments are named *pDst* or *dst*, sometimes followed by numbers or names, for example, *pDst1*, *dstLen*. For in-place operations, the input/output argument contains the name *pSrcDst*.

Examples of function syntax:

- `ippsIIR_32f_I(Ipp32f* pSrcDst, int len, IppsIIRState_32f* pState);`
- `ippiConvert_8u1u_C1R(const Ipp8u* pSrc, int srcStep, Ipp8u* pDst, int dstStep, int dstBitOffset, IppiSize roiSize, Ipp8u threshold);`
- `ippmSub_vac_32f(const Ipp32f* pSrc, int srcStride0, int srcStride2, Ipp32f val, Ipp32f* pDst, int dstStride0, int dstStride2, int len, int count);`

Getting Started with Intel(R) IPP 2

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

This chapter helps you start using Intel(R) Integrated Performance Primitives (Intel(R) IPP) by giving basic information you need to know and describing the necessary steps you need to follow after installation of the product.

Checking Your Installation

After completing the installation of Intel IPP, confirm proper installation and configuration of the library with the following steps:

1. Check the installation directory. The default installation directory is `/opt/intel/ComposerXE-2011`.
2. Check that file `ippvars.sh` is placed in the `bin` directory. You can use this file to set the environment variables `LD_LIBRARY_PATH`, `LIB`, and `INCLUDE` in the user shell.
3. Check that the dispatching and processor-specific libraries are on the path.

If you receive error message "`No shared library was found in the Waterfall procedure`", this means that the operating system is unable to determine the location of the Intel IPP shared object libraries. To solve this issue:

- Ensure that the Intel IPP directory is in the path. Before using the Intel IPP shared libraries, add path to the shared libraries to the system variable `LD_LIBRARY_PATH` as described in Using Intel® IPP Shared Object Libraries (SO).

Obtaining Version Information

To obtain information about the active library version including the version number, package ID, and the licensing information, call the `ippGetLibVersion` function. See the "*Support Functions*" chapter in the *"Intel(R) IPP Reference Manual" (vol.1)* for the function description and calling syntax.

You may also get the version information by running the `ippversion.h` file located in the `/include` directory.

Building Your Application

Follow the procedure described below to build the Linux OS application.

Setting Environment Variables

The shell script `ippvars.sh` in the `ipp/bin` directory sets your `LD_LIBRARY_PATH`, `LIB`, and `INCLUDE` environment variables for Intel IPP on the corresponding architecture. To do this run this file with the one of the following options in the command line:

`ippvars.bat ia32|intel64|lp32`.

Alternatively you can run the architecture specific shell script `ippvars_<arch>.sh` in the `ipp/bin/<arch>` directory to set your `LD_LIBRARY_PATH`, `LIB`, and `INCLUDE` environment variables for the corresponding architecture.

To set environemnt variables manually, add the path to the shared library to the `LD_LIBRARY_PATH` variable as described in Using Intel(R) IPP Shared Object Libraries (SO). You also need to specify the location for the Intel IPP header and library files with the following commands:

`export INCLUDE=$IPPROOT/include:$INCLUDE (bash)`,

`setenv INCLUDE=$IPPROOT/include:${INCLUDE} (csh)` - for header files;

`export LIB=$IPPROOT/lib:$LIB (bash)`,

`setenv LIB=$IPPROOT/lib:${LIB} (csh)` - for library files.

Including Header Files

Intel IPP functions and types are defined in several header files that are organized by the function domains and groups. They are located in the `include` directory. For example, the `ippac.h` file contains declarations for all primitives for audio processing and audio coding.

The file `ipp.h` includes Intel IPP header files with the exception of cryptography and generated functions.

If you do not use cryptography and generated functions, use only `ipp.h` in your application for forward compatibility.

If you want to use cryptography and generated functions, you must directly include their header files `ipcp.h` and `ippgen.h` in your application.

Calling Intel IPP Functions

Due to the shared library dispatcher and merged static library mechanisms described in Linking Your Application with Intel(R) IPP, calling Intel IPP functions is as simple as calling any other C function.

To call an Intel IPP function, do the following:

1. Include the `ipp.h` header file
2. Set up the function parameters
3. Call the function

The multiple versions of optimized code for each function are concealed under a single entry point. Refer to the "*Intel(R) IPP Reference Manual*" for detailed function descriptions, lists of required parameters, return values and so on.

Before You Begin Using Intel IPP

Before you start using Intel IPP, it is helpful to understand some basic concepts summarized in the table below:

What you need to know before you get started

Function Domain	Identify the Intel IPP functional domain that questions to answer belong to. **Reason**: If you know the functional domain you intend to use, it will narrow the search in the Reference Manuals for specific routines you need. Besides, you may easily find a sample codes you would like to run from http://www.intel.com/software/products/ipp/samples.htm. Refer to the sections "Domains" and "Selecting Intel IPP Libraries Needed for your Application" to understand what function domains are and what libraries are needed, and to the table "Library Dependencies by Domain" to understand what kind of cross-domain dependency is introduced.
Linking model	Decide what linking method is appropriate for linking. **Reason**: If you choose a linking method that suits you, you will get the best linking results. For information on the benefits of each linking method, linking command syntax and examples, see Linking Your Application with Intel(R) IPP
Threading model	Select among the following options to determine how you are going to thread your application: • Your application is already threaded. • You may want to use the Intel(R) threading capability, that is, Compatibility OpenMP* run-time library (`libiomp`), or a threading capability provided by a third-party compiler. • You do not want to thread your application. **Reason**: By default, Intel IPP uses the OpenMP* software to set the number of threads that will be used. If you need a different number, you have to set it yourself using one of the available mechanisms. For more information, see Supporting Multithreaded Applications.

Intel(R) IPP Structure 3

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

This chapter discusses the structure of Intel(R) IPP after installation as well as the library types supplied.

Intel IPP Directory Structure

The table below shows the directory structure of Intel IPP for the given architecture after installation.

Directory	File types
`<`*ipp directory*`>`	Main directory (by default: `/opt/intel/ComposerXE-2011`)
`<`*ipp directory*`>/Documentation/en_US/ipp`	Intel IPP documentation files
`<`*ipp directory*`>/Samples/en_US/IPP`	Intel IPP application-level samples (see Using Intel IPP Samples)
`<`*ipp directory*`>/ipp/include`	Intel IPP header files
`<`*ipp directory*`>/ipp/lib/<`*arch*`>`	Intel IPP static libraries, shared objects and their soft links
`<`*ipp directory*`>/ipp/interfaces`	High-level interfaces for data-compression

Directory	**File types**
`<ipp directory>/ipp/bin/<arch>`	shell scripts to set environment variables
`<ipp directory>/ipp/tools/<arch>/perfsys`	Performance Test tool
`<ipp directory>/ipp/tools/<arch>/staticlib`	tools to create single-processor executable

Here <*arch*> refer to the architecture specific folder.

Supplied Libraries

The following table lists the types of libraries in Intel IPP and shows examples of the library files supplied.

Types of Libraries of Intel IPP

Library types	**Description**	**Folder location**	**Example**
Dynamic	Shared object libraries include both processor dispatchers and function implementations	`ipp/lib/<arch>`	`libipps.so.7.0` `libippsp8.so.7.0`
	Soft links to shared object libraries	`ipp/lib/<arch>`	`libipps.so` `libippsp8.so`
Static merged	Contain function implementations for all supported processor types:		
	libraries with position independent code (PIC)	`ipp/lib/<arch>`	`libipps_l.a`
	(non-PIC libraries*)	`ipp/lib/<arch>/nonpic`	`libipps_l.a`
Threaded static merged	Contain threaded function implementations	`ipp/lib/<arch>`	`libipps_t.a`
Static emerged	Contain dispatchers for the merged libraries:		
	libraries with position independent code (PIC)	`ipp/lib/<arch>`	`libipps_l.a`
	(non-PIC libraries*)	`ipp/lib/<arch>/nonpic`	`libipps_l.a`

Here <*arch*> refer to the architecture specific folder.

* - non-PIC libraries are suitable for kernel-mode and device-driver use.

Using Intel IPP Shared Object (SO) Libraries

Intel IPP includes the shared object (s) libraries () and soft links to them in the ipp/lib/<arch> directory.

Before using the shared object libraries, add a path to the libraries to the system variable LD_LIBRARY_PATH by using the shell script (see Setting Environment Variables).

Alternatively you can set the environment variable LD_LIBRARY_PATH manually. For example, if the libraries are in the /opt/intel/Compiler/ipp/lib/ia32 directory, enter the following command line for bash:

```
export LD_LIBRARY_PATH=/opt/intel/Compiler/ipp/lib/ia32:$LD_LIBRARY_PATH
```

or for csh:

```
setenv LD_LIBRARY_PATH=/opt/intel/Compiler/ipp/lib/ia32:${LD_LIBRARY_PATH}
```

The shared objects libraries libipp*.so.x.x (* denotes the appropriate function domain, x.x - denotes Intel IPP version number) are "dispatcher" dynamic libraries. At run time, they detect the processor and load the correct processor-specific shared object libraries. This enables you to write the code to call the Intel IPP functions without worrying about which processor the code will execute on - the appropriate processor-specific library is automatically used. These processor-specific libraries contain processor specific codes in their names. For example, libippiv8.so.7.0 in the /ipp/lib/ia32 directory reflects the imaging processing libraries optimized for the Intel® Core™ 2 Duo processors.

Include in the project soft links to the shared libraries instead of the shared libraries themselves. These soft links are named as the corresponding shared libraries without version indicator, for example, libippiv8.so.

See also Selecting Intel® IPP Libraries Needed by Your Application.

NOTE. You must include the appropriate libiomp5.so in your PATH environment variable. You can find this file in the compiler/lib/<arch> directory.

Using Intel IPP Static Libraries

The Intel IPP includes the *merged* static library files that contain optimized code for different processor types of each function. These files reside in the /ipp/lib/<arch> directory (see Intel IPP directory structure).

Just as with the dynamic dispatcher, the appropriate version of a function is executed when the function is called. This mechanism is not as convenient as the dynamic mechanism, but it can result in a smaller total code size in spite of the big size of the static libraries.

To use these static libraries, link to the appropriate files libipp*_l.a or libipp*_t.a in the /ipp/lib/<arch> directory (* refers to the functional domain). Follow the directions in the Intel IPP Linkage Samples and create the dispatching stubs for the functions that you need. You need to set your LIB environment variable (see Setting Environment Variables), or refer to these files using their full path.

See also Selecting Intel® IPP Libraries Needed by Your Application.

Contents of the Documentation Directory

The <ipp directory>/Documentation/enUS/ipp directory includes all the documentation related to Intel(R) IPP. See the ipp_documentation.htm file for a listing of all the available document with links or pointers to their location.

Configuring Your Development Environment 4

This chapter explains how to configure your development environment for use with Intel(R) IPP.

Configuring Eclipse CDT to Link with Intel IPP

After linking your CDT with Intel IPP, you can benefit from the Eclipse-provided code assist feature. See *Code/Context Assist* description in *Eclipse Help*.

Configuring Eclipse CDT 4.0

To configure Eclipse CDT 4.0 to link with Intel IPP, follow the instructions below:

1. If the tool-chain/compiler integration supports `include` path options, go to the **Includes** tab of the **C/C++ General** > **Paths and Symbols** property page and set the Intel IPP `include` path, for example, the default value is `/opt/intel/Compiler/ipp/include`.
2. If the tool-chain/compiler integration supports library path options, go to the **Library Paths** tab of the **C/C++ General** > **Paths and Symbols** property page and set a path to the Intel IPP libraries, depending upon the target architecture, for example, with the default installation, `opt/intel/Compiler/ipp/lib/`*`<arch>`*.
3. For a particular build, go to the **Tool Settings** tab of the **C/C++ Build > Settings** property page and specify names of the Intel IPP libraries to link with your application, for example, `ipps_t` or `ippch_t` (as compilers typically require library names rather than library file names, the "`lib`" prefix and "`a`" extension are omitted). See section Selecting Intel® IPP Libraries Needed by Your Application in chapter 5 on the choice of the libraries. The name of the particular setting where libraries are specified depends upon the compiler integration.

NOTE. The compiler/linker automatically picks up the include and library paths settings only in case the automatic makefile generation is turned on, otherwise, you must specify the include and library paths directly in the makefile to be used.

Configuring Eclipse CDT 3.x

To configure Eclipse CDT 3.x to link with Intel IPP, follow the instructions below.

Standard Make projects:

1. Go to **C/C++ Include Paths and Symbols** property page and set the Intel IPP `include` path, for example, the default value is `/opt/intel/Compiler/ipp/include`.

2. Go to the **Libraries** tab of the **C/C++ Project Paths** property page and set the Intel IPP libraries to link with your applications, for example, `opt/intel/Compiler/ipp/lib/ia32/libipps_l.a, or opt/intel/Compiler/ipp/lib/intel64/libippdc_t.a`. See section Selecting Intel® IPP Libraries Needed by Your Application in chapter 5 on the choice of the libraries.

NOTE. With the Standard Make the above settings are needed for the CDT internal functionality only. The compiler/linker does not automatically pick up these settings, and you must specify them directly in the makefile.

Managed Make projects

Specify settings for a particular build:

1. Go to the **Tool Settings** tab of the **C/C++ Build** property page. All the settings you need to specify are on this page. Names of the particular settings depend upon the compiler integration and therefore are not given below.
2. If the compiler integration supports include path options, set the Intel IPP `include` path, for example, the default value is `/opt/intel/Compiler/ipp/include`.
3. If the compiler integration supports library path options, set a path to the Intel IPP libraries, depending upon the target architecture, for example, with the default installation, `opt/intel/Compiler/ipp/lib/<arch>`.
4. Specify names of the Intel IPP libraries to link with your application, for example, `ipps_t` or `ippch_t` (as compilers typically require library names rather than library file names, the "`lib` " prefix and "a" extension are omitted). See section Selecting Intel® IPP Libraries Needed by Your Application in chapter 5 on the choice of the libraries.

Make sure that your project that uses Intel IPP is open and active.

Linking Your Application with Intel(R) IPP 5

This chapter discusses dispatching of the Intel(R) IPP libraries to specific processors using various models for linking Intel(R) IPP to an application, considers differences between the linking methods regarding development and target environments, installation specifications, run-time conditions, and other application requirements to help you select the best linking method for your application, shows linking procedure for each linking method, and provides examples.

Optimization Notice
The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors. While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Intel recommends that you evaluate other library products to determine which best meets your requirements.

Dispatching

Intel IPP uses codes optimized for various central processing units (CPUs). Dispatching refers to detection of your CPU and selecting the corresponding Intel IPP binary. For example, the `libippiv8.so.7.0` file in the `/ipp/lib/ia32` directory, reflects the imaging processing libraries optimized for the Intel(R) Core™ 2 Duo processors.

A single Intel IPP function, for example `ippsCopy_8u()`, may have many versions, each one optimized to run on a specific Intel(R) processor with specific architecture, for example, the version of this function optimized for the Intel(R) Core™ 2 Duo processor is `v8_ippsCopy_8u()`, and version optimized for 64-bit applications on processors with Intel(R) SSE4.1 is `u8_ippsCopy_8u()`.

The following table Table 5-1 shows processor-specific codes used in Intel IPP:

Table 5-1 Identification of Codes Associated with Processor-Specific Libraries

Abbreviation	Meaning
	IA-32 Intel® architecture

Abbreviation	Meaning
px	C-optimized for all IA-32 processors
v8	Optimized for processors with Intel® Supplemental Streaming SIMD Extensions 3
p8	Optimized for processors with Intel® Streaming SIMD Extensions 4.1
g9	Optimized for processors that support Intel® Advanced Vector Extensions instruction set
s8	Optimized for the Intel® Atom™ processor
Intel® 64 architecture	
mx	C-optimized for processors with Intel® 64 instructions set architecture
u8	Optimized for 64-bit applications on processors with Intel® Supplemental Streaming SIMD Extensions 3
y8	Optimized for 64-bit applications on processors with Intel® Streaming SIMD Extensions 4.1
n8	Optimized for the Intel® Atom™ processor
e9	Optimized for processors that support Intel® Advanced Vector Extensions instruction set

Supporting Intel(R) Xeon(R) Processors with the Intel(R) 64 Architecture

Intel IPP supports Intel(R) Xeon(R) processors with the Intel(R) 64 architecture running 32-bit and 64-bit modes. For the 32-bit mode, download and install Intel IPP for the Pentium(R) processor-based systems. When running the 64-bit mode, install Intel IPP for the Intel(R) 64 architecture.

Detecting Processor Features and Type

To obtain information about the features of the processor used in your computer system, use the function `ippGetCpuFeatures`. To obtain information about the features of the processor that are enabled for the loaded libraries, use the function `ippGetEnabledCpuFeatures`. Both functions are declared in the `ippcore.h` file. These function retrieve main processor features returned by the function `CPUID.1` and store them consecutively in the mask that is returned by the function. The table below lists all CPU features that can be retrieved.

Table 5-3 Processor Features Mask

Mask Value	Bit Name	Supported Feature
1	`ippCPUID_MMX`	MMX™ technology
2	`ippCPUID_SSE`	Intel® Streaming SIMD Extensions
4	`ippCPUID_SSE2`	Intel® Streaming SIMD Extensions 2
8	`ippCPUID_SSE3`	Intel® Streaming SIMD Extensions 3

Mask Value	Bit Name	Supported Feature
16	ippCPUID_SSSE3	Intel® Supplemental Streaming SIMD Extensions 3
32	ippCPUID_MOVBE	MOVBE instruction
64	ippCPUID_SSE41	Intel® Streaming SIMD Extensions 4.1
128	ippCPUID_SSE42	Intel® Streaming SIMD Extensions 4.2
256	ippCPUID_AVX	Intel® Advanced Vector Extensions (Intel® AVX) instruction set
512	ippAVX_ENABLEDBYOS	The operating system supports Intel® AVX
1024	ippCPUID_AES	Intel® Advanced Encryption Standard (AES) instructions set
2048	ippCPUID_CLMUL	PCLMULQDQ instruction

To detect the processor type used in your computer system, use the function `ippGetCpuType`, which is declared in the `ippcore.h` file. It returns an appropriate `IppCpuType` variable value. All of the enumerated values are given in the `ippdefs.h` header file. For example, the return value `ippCpuPII` means that your system uses Intel(R) Core™ 2 Duo processor.

The following table Table 5-4 shows possible return values of `ippGetCpuType` and their meaning.

Table 5-4 Detecting Processor Type. Return Values and Their Meaning

Return Value	Processor Type
ippCpuPP	Intel® Pentium® processor
ippCpuPMX	Pentium® processor with MMX™ technology
ippCpuPPR	Pentium® Pro processor
ippCpuPII	Pentium® II processor
ippCpuPIII	Pentium® III processor and Pentium® III Xeon® processor
ippCpuP4	Pentium® 4 processor and Intel® Xeon® processor
ippCpuP4HT	Pentium® 4 processor with Hyper-Threading Technology
ippCpuP4HT2	Pentium® Processor with Intel® Streaming SIMD Extensions 3
ippCpuCentrino	Intel® Centrino™ mobile Technology
ippCpuCoreSolo	Intel® Core™ Solo processor
ippCpuCoreDuo	Intel® Core™ Duo processor
ippCpuITP	Intel® Itanium® processor

Return Value	Processor Type
`ippCpuITP2`	Intel® Itanium® 2 processor
`ippCpuEM64T`	Intel® 64 Instruction Set Architecture (ISA)
`ippCpuC2D`	Intel® Core™ 2 Duo processor
`ippCpuC2Q`	Intel® Core™ 2 Quad processor
`ippCpuPenryn`	Intel® Core™ 2 processor with Intel® Streaming SIMD Extensions 4.1 instruction set
`ippCpuBonnell`	Intel® Atom™ processor
`ippCpuNehalem`	Intel® Core™ i7 processor
`ippCpuSSE`	Processor with Intel® Streaming SIMD Extensions instruction set
`ippCpuSSE2`	Processor with Intel® Streaming SIMD Extensions 2 instruction set
`ippCpuSSE3`	Processor with Intel® Streaming SIMD Extensions 3 instruction set
`ippCpuSSSE3`	Processor with Intel® Supplemental Streaming SIMD Extensions 3 instruction set
`ippCpuSSE41`	Processor with Intel® Streaming SIMD Extensions 4.1 instruction set
`ippCpuSSE42`	Processor with Intel® Streaming SIMD Extensions 4.2 instruction set
`ippCpuAVX`	Processor supports Intel® Advanced Vector Extensions instruction set
`ippCpuAES`	Processor supports Intel® Advanced Encryption Standard (AES) instructions set
`ippCpuX8664`	Processor supports 64 bit extension
`ippCpuUnknown`	Unknown Processor

Selecting Between Linking Methods

You can use different linking methods for Intel IPP:

- Dynamic linking using the run-time shared object (SO) libraries
- Static linking with dispatching by using emerged and merged static libraries
- Static linking without automatic dispatching using merged static libraries
- Dynamically building your own - custom - SO.

Answering the following questions helps you select the linking method which best suites you:

- Are there limitations on the size of the application executable? Are there limitations on the size of the application installation package?

- Is the Intel IPP-based application a device driver or similar "ring 0" software that executes in the kernel mode at least some of the time?
- Will the application be installed on a range of processor types, or is the application explicitly supported only on a single type of processor? Is the application part of an embedded computer with only one type of processor?
- What resources are available for maintaining and updating customized Intel IPP components? What level of effort is acceptable for incorporating new processor optimizations into the application?
- How often will the application be updated? Will application components be distributed independently or will they always be packaged together?

Dynamic Linking

Dynamic linking is the simplest method and the most commonly used. It takes full advantage of the dynamic dispatching mechanism in the shared object (SO) libraries) (see also Intel® IPP Structure). The following table summarizes the features of dynamic linking to help you understand the benefits and drawbacks of this linking method.

Table 5-5 Summary of Dynamic Linking Features

Benefits	Drawbacks
• Automatic run-time dispatch of processor-specific optimizations • Enables updates with new processor optimizations without recompile/relink • Reduces disk space requirements for applications with multiple Intel IPP-based executables • Enables more efficient shared use of memory at run-time for multiple Intel IPP-based applications	• The application executable requires access to Intel IPP run-time shared object (SO) libraries) • Not appropriate for kernel-mode/device-driver/ring-0 code • Not appropriate for web applets/plug-ins that require very small download • A one-time performance penalty when the Intel IPP SOs are first loaded

To dynamically link with Intel IPP, follow these steps:

1. Include `ipp.h` in your application. This header includes the header files for all Intel IPP functional domains.
2. Call IPP functions using normal, undecorated function names.
3. Link corresponding domain soft links. For example, if you use the function `ippsCopy_8u`, link to `libipps.so`.
4. Make sure that you run `ipp/bin/ippvars.sh` shell script before using Intel IPP libraries in the current session, or set `LD_LIBRARY_PATH` correctly. For example, `export LD_LIBRARY_PATH =$IPPROOT/lib/<arch>:$LD_LIBRARY_PATH` in bash, or `setenv LD_LIBRARY_PATH =$IPPROOT/lib/<arch>:${LD_LIBRARY_PATH}` in csh.

Static Linking with Dispatching

Some applications use only a few Intel IPP functions and require a small memory footprint. Using the static link libraries offers both the benefits of a small footprint and optimization on multiple processors. The static libraries (such as `libipps_l.a`) provide an entry point for the non-decorated (with normal names) Intel IPP functions, and the jump table to each processor-specific implementation. When linked with your application, the function calls corresponding functions in accordance with the CPU setting detected by functions in `libippcore_l.a`.

If you want to use the threaded functions of Intel IPP, you need to link to the threaded versions of the static libraries (such as `libipps_t.a`), and threaded version `libippcore_t.a`.

You can choose one of the following functions to initialize the libraries:

- `ippStaticInit()` to use the best available optimization, or
- `ippStaticInitCpu()` that lets you specify directly the CPU.

One of these functions must be called before any other IPP functions. Otherwise, a C-optimized version of the IPP functions is called. This can decrease the performance of your application. The following example illustrates the performance difference. This example appears in the `t2.cpp` file.

Example 5-1 Performance difference with and without calling StaticInit

```
#include <stdio.h>
#include <ipp.h>

int main() {
   const int N = 20000, loops = 100;
   Ipp32f src[N], dst[N];
   unsigned int seed = 12345678, i;
   Ipp64s t1,t2;
   /// no StaticInit call, means PX code, not optimized
   ippsRandUniform_Direct_32f(src,N,0.0,1.0,&seed);
   t1=ippGetCpuClocks();
   for(i=0; i<loops; i++)
      ippsSqrt_32f(src,dst,N);
   t2=ippGetCpuClocks();
   printf("without StaticInit: %.1f clocks/element\n",
         (float)(t2-t1)/loops/N);
   ippStaticInit();
   t1=ippGetCpuClocks();
   for(i=0; i<loops; i++)
      ippsSqrt_32f(src,dst,N);
   t2=ippGetCpuClocks();
   printf("with StaticInit: %.1f clocks/element\n",
         (float)(t2-t1)/loops/N);
   return 0;
}



t2.cpp
cmdlinetest>t2
without StaticInit: 61.3 clocks/element
with StaticInit: 4.5 clocks/element
```

The following table summarizes the features of static linking to help you understand the benefits and drawbacks of this linking method.

Table 5-6 Summary of features of the static linking (with dispatching)

Benefits	Drawbacks
• Dispatches processor-specific optimizations during run-time • Creates a self-contained application executable • Generates a smaller footprint than the full set of Intel IPP SOs	• Intel IPP code is duplicated for multiple Intel IPP-based applications because of static linking • An additional function call for dispatcher initialization is needed (once) during program initialization

Follow these steps to use static linking with dispatching:

1. Include`ipp.h` in your application. This header includes the header files of all IPP domains.
2. Initialize the static dispatcher using either function `ippStaticInit()` or `ippInitCPU()`, which are declared in the header file `ippcore.h`.
3. Call IPP functions using normal, undecorated function names.
4. Link corresponding static libraries, and then `libippcore_l.a` or `libippcore_t.a`. For example, if you use the function `ippsCopy_8u()`, the linked libraries are `libipps_l.a` and `libippcore_l.a`.

Static Linking without Dispatching

Static linking links directly with the merged static libraries. Use this method to link a self contained application that is supported on a specific processor type. Static linking is useful for embedded applications that are bundled with one type of processor.

The following table summarizes the features of static linking to help you understand the benefits and drawbacks of this linking method.

Table 5-7 Summary of Features of the Static Linking (without dispatching)

Benefits	Drawbacks
• Small executable size with support for only one processor type • Suitable for kernel-mode/device-driver/ring-0 use *) • Suitable for a Web applet or a plug-in requiring very small file download and support for only one processor type • Self-contained application executable that does not require the Intel IPP run-time SOs • Smallest footprint for application package • Smallest installation package	• The executable is optimized for only one processor type • Updates to processor-specific optimizations require rebuild and/or relink

*) for unthreaded non-PIC libraries only

You may want to use your own static dispatcher instead of the provided emerged dispatcher. The IPP sample `mergelib` demonstrates how to do this.

Refer to the latest updated sample from the Intel IPP samples directory: `/ipp-samples/advanced-usage/linkage/mergelib` at http://www.intel.com/software/products/ipp/samples.htm.

The Intel IPP package includes a set of processor-specific header files (such as `ipp_v8.h`). You can use these header files instead of the `IPPCALL` macro. Refer to *Static linking to Intel(R) IPP Functions for One Processor* in `/ipp/tools/<`*`arch`*`>/static.lib/readme.htm`.

Building a Custom SO

A custom shared object (SO) is useful for an application that has few internal modules, and only these modules share Intel IPP code. In this case, you can use dynamic linking with a customized SO containing only those Intel IPP functions that the application uses.

The following table summarizes features of the custom SO.

Table 5-8 Custom SO Features

Benefits	Drawbacks
• Run-time dispatching of processor-specific optimizations • Reduced hard-drive footprint compared with a full set of Intel IPP SOs • Smallest installation package to accommodate use of some of the same Intel IPP functions by multiple applications	• Application executable requires access to the Intel compiler specific run-time libraries that are delivered with Intel IPP • Developer resources are needed to create and maintain the custom SOs • Integration of new processor-specific optimizations requires rebuilding the custom SOs • Not appropriate for kernel-mode/device-driver/ring-0 code

To create a custom SO, you need to create a separate build step or project that generates the SO and stubs. The specially developed sample demonstrates how to do it. Refer to the latest updated `custom so` sample from the Intel IPP samples directory: `/ipp-samples/advanced-usage/linkage/customso` at http://www.intel.com/software/products/ipp/samples.htm.

Comparison of Intel IPP Linking Methods

The following tablegives a quick comparison of the Intel IPP linking methods.

Table 5-9 Intel IPP Linking Method Summary Comparison

Feature	Dynamic Linking	Static Linkingwith Dispatching	Static Linking without Dispatching	Using Custom SO
Processor Updates	Automatic	Recompile & redistribute	Release new processor-specific application	Recompile & redistribute
Optimization	All processors	All processors	One processor	All processors
Build	Link to stub static libraries	Link to static libraries and static dispatchers	Link to merged libraries	Build separate SO
Calling	Regular names	Regular names	Processor-specific names	Regular names

Feature	**Dynamic Linking**	**Static Linkingwith Dispatching**	**Static Linking without Dispatching**	**Using Custom SO**
Total Binary Size	Large	Small	Smallest	Small
Executable Size	Smallest	Small	Small	Smallest
Kernel Mode	No	Yes	Yes	No

Selecting the Intel(R) IPP Libraries Needed by Your Application

The following table shows functional domains and functions groups, and the relevant header files and libraries used for each linking method.

Table 5-10 Libraries Used for Each Linking Method

Description	**Header Files**	**Dynamic Linking**	**Static Linking (with/without Dispatching) and Custom Dynamic Linking**
Audio Coding	`ippac.h`	`libippac.so`	`libippac_l.a` `libippac_t.a`
Color Conversion	`ippcc.h`	`libippcc.so`	`libippcc_l.a` `libippcc_t.a`
String Operation	`ippch.h`	`libippch.so`	`libippch_l.a` `libippch_t.a`
Cryptography	`ippcp.h`	`libippcp.so`	`libippcp_l.a` `libippcp_t.a`
Computer Vision	`ippcv.h`	`libippcv.so`	`libippcv_l.a` `libippcv_t.a`
Data Compression	`ippdc.h`	`libippdc.so`	`libippdc_l.a` `libippdc_t.a`
Data Integrity	`ippdi.h`	`libippdi.so`	`libippdi_l.a` `libippdi_t.a`
Image Processing	`ippi.h`	`libippi.so`	`libippi_l.a` `libippi_t.a`
Image Compression	`ippj.h`	`libippj.so`	`libippj_l.a` `libippj_t.a`

Description	Header Files	Dynamic Linking	Static Linking (with/without Dispatching) and Custom Dynamic Linking
Realistic Rendering and 3D Data Processing	`ippr.h`	`libippr.so`	`libippr_l.a` `libippr_t.a`
Small Matrix Operations	`ippm.h`	`libippm.so`	`libippm_l.a` `libippm_t.a`
Signal Processing	`ipps.h`	`libipps.so`	`libipps_l.a` `libipps_t.a`
Speech Coding	`ippsc.h`	`libippsc.so`	`libippsc_l.a` `libippsc_t.a`
Video Coding	`ippvc.h`	`libippvc.so`	`libippvc_l.a` `libippvc_t.a`
Vector Math	`ippvm.h`	`libippvm.so`	`libippvm_l.a` `libippvm_t.a`
Generated Functions	`ippgen.h`	`libippgen.so`	`libippgen_l.a` `libippgen_t.a`
Core Functions	`ippcore.h`	`libippcore.so`	`libippcore_l.a` `libippcore_t.a`

Libraries for Dynamic Linking

To use the shared objects, you must use the soft link to the domain libraries `libipp*.so` used in your application (here * denotes the appropriate function domain). You can find them in the `/ipp/lib/<arch>` directory. Additionally you must link to the libraries `libipps.so`, `libippcore.so`, and `libiomp.so`.

For example, your application uses three Intel IPP functions `ippiCopy_8u_C1R`, `ippiCanny_16s8u_C1R`, and `ippmMul_mc_32f`. These three functions belong to the image processing, computer vision, and small matrix operations domains, respectively. To include these functions into your application, you must link to the following libraries:

```
libippi.so
libippcv.so
libippm.so
libipps.so
libippcore.so
 libiomp.so
```

Libraries for Static Linking

To use the static linking libraries, you need to link to all required domain libraries `libipp*_l.a`,core library `libippcore_l.a` , and libraries on which domain libraries depend (see next section). The `*` denotes the appropriate function domain.

If you want to use the Intel IPP functions threaded with the OpenMP*, you need to link to the threaded versions of the libraries `libipp*_t.a`, `libippcore_t.a`, and `ibiomp.a`.

All domain-specific and core libraries are located in the `/ipp/lib/<arch>` directory.

For example, your application uses three Intel IPP functions `ippiCopy_8u_C1R`, `ippiCanny_16s8u_C1R`, and `ippmMul_mc_32f`. These three functions belong to the image processing, computer vision, and small matrix operations domains respectively. All these domain libraries depend on signal processing library.

NOTE. The order in which libraries are linked must correspond to the library dependencies by domain (see the next section).

To include these functions into your application, link to the following libraries:

```
libippcv_l.a
libippm_l.a
libippi_l.a
libipps_l.a
libippcore_l.a
```

or if you want to use the threaded functions:

```
libippcv_t.a
libippm_t.a
libippi_t.a
libipps_t.a
libippcore_t.a
libiomp.a
```

Library Dependencies by Domain

The following table lists library dependencies by domain for static linking.

NOTE. Note when you link libraries, the library in the **Library** column must precede the libraries in the **Dependent on** column.

Table 5-11 Library Dependencies by Domain

Domain	Library	Dependent on
Audio Coding	`ippac`	`ippdc`, `ipps`, `ippcore`
Color Conversion	`ippcc`	`ippi`, `ipps`, `ippcore`
Cryptography	`ippcp`	`ippcore`
Computer Vision	`ippcv`	`ippi`, `ipps`, `ippcore`
Data Compression	`ippdc`	`ipps`, `ippcore`
Data Integrity	`ippdi`	`ippcore`
Image Processing	`ippi`	`ipps`, `ippcore`
Image Compression	`ippj`	`ippi`, `ipps`, `ippcore`
Small Matrix Operations	`ippm`	`ippi`, `ipps`, `ippcore`
Realistic Rendering and 3D Data Processing	`ippr`	`ippi`, `ipps`, `ippcore`
Signal Processing	`ipps`	`ippcore`
Speech Coding	`ippsc`	`ipps`, `ippcore`
String Operations	`ippch`	`ipps`, `ippcore`
Video Coding	`ippvc`	`ippi`, `ipps`, `ippcore`
Vector Math	`ippvm`	`ippcore`

Refer to the *Intel IPP Reference Manual* to find which domain your function belongs to.

Generated funcions (`ippgen` library) depends on `ipps` and `ippcore` libraries.

When you use static linking to a certain library (for example, data compression domain `ippdc`), you must link to the libraries on which it depends (in our example, the signal processing `ipps` and core functions `ippcore`).

Linking Examples

For more linking examples, please go to http://www.intel.com/software/products/ipp/samples.htm

For information on using sample code, see Intel(R) IPP Samples.

Supporting Multithreaded Applications 6

This chapter discusses the use of Intel(R) IPP in multithreading applications and helps you:

set number of threads in multithreaded applications

get information on number of threads

use shared L2 cache

avoid nested parallelization

disable multithreading.

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

Intel IPP Threading and OpenMP* Support

All Intel IPP functions are thread-safe. They supports multithreading in both dynamic and static libraries and can be used in multithreaded applications.

Some Intel IPP functions contain OpenMP* code, which increase performance on multi-processor and multi-core systems. These functions include color conversion, filtering, convolution, cryptography, cross correlation, matrix computation, square distance, bit reduction, and so on.

Refer to the *ThreadedFunctionsList.txt* document to see the list of all threaded APIs in the documentation directory of the Intel IPP installation.

See also http://www.intel.com/software/products/support/ipp for more topics related to Intel IPP threading and OpenMP* support, including older Intel IPP versions of threaded API.

Setting Number of Threads

By default, the number of threads for Intel IPP threaded libraries equals the number of processors in the system. If the value of the `OMP_NUM_THREADS` environment variable is less than the number of processors in the system, then the number of threads for Intel IPP threaded libraries equals the value of the `OMP_NUM_THREADS` environment variable.

To configure the number of threads used by Intel IPP internally, call the function `ippSetNumThreads(n)` at the very beginning of an application. `n` is the desired number of threads (1, ...). To disable internal parallelization, call `ippSetNumThreads(1)`.

Getting Information on Number of Threads

To find the number of threads created by the Intel IPP, call function `ippGetNumThreads`.

Using a Shared L2 Cache

Some functions in the signal processing domain are threaded on two threads intended for the Intel(R) Core™ 2 processor family, and make use of the merged L2 cache. These functions (single and double precision `FFT`, `Div`, `Sqrt` and so on) achieve the maximum performance if both two threads are executed on the same die. In this case, these threads work on the same shared L2 cache. For processors with two cores on the die, this condition is satisfied automatically. For processors with more than two cores, set the following OpenMP environmental variable to avoid performance degradation:

```
KMP_AFFINITY=compact
```

Avoiding Nested Parallelization

Nested parallelization may occur if you use a threaded Intel IPP function on a multithreaded application. Nested parallelization may cause performance degradation.

For applications that use OpenMP threading, nested threading is disabled by default, so this is not an issue.

However, if your application uses threading created by a tool other than OpenMP, you must disable multithreading in the threaded Intel IPP function to avoid this issue.

Disabling Multithreading

To disable multithreading, call function `ippSetNumThreads` with parameter `1`, or link your application with IPP unthreaded static libraries.

Managing Performance and Memory

7

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

This chapter describes several methods for managing performance and memory to optimize your Intel(R) IPP application: aligning memory, thresholding denormal data, reusing buffers, using FFT for algorithmic optimization.

Finally, it explains how to test the performance of the Intel IPP functions, using the Intel(R) IPP Performance Test Tool and it gives some examples of using the Performance Tool Command Lines.

Memory Alignment

Access to memory is faster if pointers to the data are aligned, and Intel IPP functions perform better if they process data with aligned pointers.

The following Intel IPP functions can be used for pointer alignment, memory allocation and deallocation:

`void* ippAlignPtr(void* ptr, int alignBytes)`

Aligns a pointer, can align to the powers of 2, that is 2, 4, 8, 16 and so on.

`void* ippMalloc(int length)`

32-byte aligned memory allocation. Memory can be freed only with the function `ippFree`.

`void ippFree(void* ptr)`

frees memory allocated by the function `ippMalloc`.

`Ipp<datatype>* ippsMalloc_<datatype>(int len)`

32-byte aligned memory allocation for signal elements of different data types. Memory can be freed only with the function `ippsFree`.

`void ippsFree(void* ptr)`

Frees memory allocated by the function `ippsMalloc`.

`Ipp<datatype>* ippiMalloc_<mod>(int widthPixels, int heightPixels, int* pStepBytes)` -

32-byte aligned memory allocation for images where every line of the image is padded with zeros. Memory can be freed only with the function `ippiFree`.

`void ippiFree(void* ptr)` -

Frees memory allocated by the function `ippiMalloc`.

The amount of memory that can be allocated is determined by the operating system and system hardware, but it cannot exceed 2GB.

- Intel IPP functions `ippFree`, `ippsFree`, and `ippiFree` can only be used to free memory allocated by the functions `ippMalloc`, `ippsMalloc`, and `ippiMalloc` respectively.
- Intel IPP functions `ippFree`, `ippsFree`, and `ippiFree` cannot be used to free memory allocated by standard functions like `malloc` or `calloc`. The memory allocated by the Intel IPP functions `ippMalloc`, `ippsMalloc`, and `ippiMalloc` cannot be freed by the standard function `free`.

The followint code example shows how to call the function `ippiMalloc`.

Example 7-1. Calling the Function ippiMalloc

```
#include "stdafx.h"
#include "ipp.h"
#include "tools.h"

int main(int argc, char *argv[])
{
   IppiSize size = {320, 240};

   int stride;
   Ipp8u* pSrc = ippiMalloc_8u_C3(size.width, size.height, &stride);
   ippiImageJaehne_8u_C3R(pSrc, stride, size);
   ipView_8u_C3R(pSrc, stride, size, "Source image", 0);

   int dstStride;
   Ipp8u* pDst = ippiMalloc_8u_C3(size.width, size.height, &dstStride);
   ippiCopy_8u_C3R(pSrc, stride, pDst, dstStride, size);
   ipView_8u_C3R(pDst, dstStride, size, "Destination image 1", 0);

   IppiSize ROISize = { size.width/2, size.height/2 };
   ippiCopy_8u_C3R(pSrc, stride, pDst, dstStride, ROISize);
   ipView_8u_C3R(pDst, dstStride, ROISize, "Destination image, small", 0);

   IppiPoint srcOffset = { size.width/4, size.height/4 };
   ippiCopy_8u_C3R(pSrc + srcOffset.x*3 + srcOffset.y*stride, stride,
      pDst, dstStride, ROISize);
   ipView_8u_C3R(pDst, dstStride, ROISize, "Destination image, small & shifted", 1);

   return 0;
}
```

Thresholding Data

Denormal numbers are the border values in the floating-point format and special case values for the processor. Denormal data occurs, for example, in filtering by Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters of the signal captured in fixed-point format and converted to the floating-point format. Operations on denormal data make processing slow, even if corresponding interrupts are disabled. To avoid the slowdown effect in denormal data processing, the Intel IPP threshold functions can be applied to the input signal before filtering. For example:

```
if (denormal_data)
 ippsThreshold_LT_32f_I( src, len, 1e-6f );
ippsFIR_32f( src, dst, len, st );
```

The `1e-6f` value is the threshold level; the input data below that level are set to zero. Because the Intel IPP threshold function is very fast, the execution of these two functions is faster than execution of filter if denormal numbers meet in the source data. Of course, if the denormal data occurs while using the filtering procedure, the threshold functions do not help.

For Intel(R) Pentium(R) 4 processor and later processors, you can set special computation modes - *flush-to-zero* (FTZ) and the *denormals-are-zero* (DAZ). Use the functions `ippSetFlushToZero` and `ippSetDenormAreZeros` to enable these modes. Note that this setting takes effect only when computing is done with the Intel(R) Streaming SIMD Extensions (Intel(R) SSE) and Intel(R) Streaming SIMD Extensions 2 (Intel(R) SSE2) instructions.

The following table illustrates how denormal data may affect performance and shows the effect of thresholding denormal data. As you can see, thresholding takes only three clocks more. On the other hand, denormal data can cause the application performance to drop x250.

Table 7-1 Performance Resulting from Thresholding Denormal Data

Data/Method	Normal	Denormal	Denormal + Threshold
CPU cycles per element	46	11467	49

Reusing Buffers

Some Intel IPP functions require internal memory for various optimization strategies. However, memory allocation inside of the function may have a negative impact on performance in some situations, for example, cache misses. To avoid or minimize memory allocation and keep your data in a warm cache, some functions, for example, Fourier transform functions (FFT), can use or reuse memory given as a parameter to the function.

If you call such a function, many times, you can reuse of an external buffer and get better performance.

The following example reuses memory buffers to compute FFT as two FFTs in two separate threads:

```
ippsFFTInitAlloc_C_32fc(&ctxN2, order-1, IPP_FFT_DIV_INV_BY_N, ippAlgHintAccurate);
ippsFFTGetBufSize_C_32fc( ctxN2, &sz );

buffer = sz > 0 ? ippsMalloc_8u( sz ) : 0;

/// prepare source data for two FFTs
int phase = 0;
ippsSampleDown_32fc( x, fftlen, xleft, &fftlen2, 2, &phase );
phase = 1;
```

```
ippsSampleDown_32fc( x, fftlen, xrght, &fftlen2, 2, &phase );

ippsFFTFwd_CToC_32fc( xleft, Xleft, ctxN2, buffer );
ippsFFTFwd_CToC_32fc( xrght, Xrght, ctxN2, buffer );
```

The external buffer is not necessary. If the pointer to the buffer is 0, the function allocates memory inside.

Using Fast Fourier Transform

Fast Fourier Transform (FFT) is a universal method to increase performance of data processing, especially in the field of digital signal processing where filtering is essential.

The convolution theorem states that filtering of two signals in the spatial domain can be computed as point-wise multiplication in the frequency domain. The data transformation to and from the frequency domain is usually performed using the Fourier transform. You can apply the Finite Impulse Response (FIR) filter to the input signal by using Intel IPP FFT functions, which are optimized for Intel® processors. You can also increase the data array length to the next greater power of two by padding the array with zeroes and then applying the forward FFT function to the input signal and the FIR filter coefficients. Fourier coefficients obtained in this way are multiplied point-wise and the result can easily be transformed back to the spatial domain. The performance gain achieved by using FFT may be very significant.

If the applied filter is the same for several processing iterations, then the once calculated filter coefficients can be reused in each iteration. The twiddle tables and the bit reverse tables are created in the initialization function for the forward and inverse transforms at the same time. The following example presents the main operations of this kind of filtering:

```
ippsFFTInitAlloc_R_32f( &pFFTSpec, fftord, IPP_FFT_DIV_INV_BY_N, ippAlgHintNone );

/// perform forward FFT to put source data xx to frequency domain
ippsFFTFwd_RToPack_32f( xx, XX, pFFTSpec, 0 );

/// perform forward FFT to put filter coefficients hh to frequency domain
ippsFFTFwd_RToPack_32f( hh, HH, pFFTSpec, 0 );

/// point-wise multiplication in freq-domain is convolution
ippsMulPack_32f_I( HH, XX, fftlen );

/// perform inverse FFT to get result in time-domain
ippsFFTInv_PackToR_32f( XX, yy, pFFTSpec, 0 );

/// free FFT tables
ippsFFTFree_R_32f( pFFTSpec );
```

The zeros in the example above could be pointers to the external memory, which is another way to increase performance. Another way to significantly improve performance is using FFT and multiplication for processing large size data.

The signal processing FIR filter in Intel IPP is implemented using FFT, and you do not need to create a special implementation of the FIR functions.

Using the Intel IPP Performance Test Tool

Intel IPP installation includes a command-line tool for performance testing - The Intel IPP Performance Test Tool (PTT). It does performance testing for each Intel IPP functions on the same hardware platforms that are valid for the related Intel IPP libraries.

The performance test executable files `ps_ipp*` files (`*` - functional domain suffix) are placed in the `/ipp/tools/<arch>/perfsys` directory. For example, `ps_ipps` is a tool to measure performance of all Intel IPP signal processing functions.

With the command-line options you can create a list of functions to test and set required parameters with which the function is called during the performance test. You can define the functions and parameters in the `.ini` file, or enter them directly from the console.

The results are saved in a `.csv` file. The course of test is displayed on the console, you can be save it in the `.txt` file.

In the enumeration mode, the Intel IPP PTT creates only a list of the tested functions on the console and can store it in the `.txt` or `.csv` files.

The command-line format is:

```
<ps_FileName> [option_1] [option_2]... [option_n]
```

A short reference for the command-line options can be displayed on the console. To invoke the reference, just enter `-?` or `-h` in the command-line:

```
ps_ipps -h
```

The command-line options are divided into several groups by functionality. You can enter options in arbitrary order with at least one space between each option name. Some options (like `-r`, `-R`, `-o`, `-O`) may be entered several times with different file names, and option `-f` may be entered several times with different function patterns. For detailed descriptions of the performance test tool command-line options, see Performance Test Tool Command-Line Options.

Examples of Performance Test Tool Command Lines

The following are examples of Intel IPP performance test tool command lines.

Example 1. Running in the standard mode:

```
ps_ippch -B -r
```

This command tests all Intel IPP string functions by the default timing method on standard data (`-B` option) and stores results in file `ps_ippch.csv` (`-r` option).

Example 2. Testing selected functions:

```
ps_ipps -f FIRLMS_32f -r firlms.csv
```

This command tests the signal processing function `FIRLMS_32f`(`-f` option), and stores the results in `firlms.csv` (`-r` option).

Example 3. Retrieving function lists:

```
ps_ippvc -e -o vc_list.txt
```

This command lists all video coding functions (`-e` option) in the output file `vc_list.txt` (`-o` option).

```
ps_ippvc -e -r H264.csv -f H264
```

This command displays the list of functions with names containing `H264`(`-f` option) that can be tested (`-e` option) on the console and stores the list in `H264.csv` (`-r` option).

Example 4. Launching performance test tool with the .ini file:

```
ps_ipps -B -I
```

This command creates `ps_ipps.ini` file after the first run (`-I` option) to test all signal processing functions using the default timing method on standard data (`-B` option).

```
ps_ippi -i -r
```

This command performs the second run to test all functions with parameters specified in the `ps_ipps.ini` file (`-i` option), and generates the output file `ps_ipps.csv` (`-r` option).

For detailed descriptions of performance test tool command line options, see Performance Test Tool Command Line Options.

Performance Test Tool Command Line Options

A

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

The following table A-1 lists the Intel(R) IPP Performance Test tool (PTT) command-line options by functional groups, and briefly describes each option.

Table A-1 Performance Test Tool Command Line Options

Groups	Options	Descriptions
1. Adjusting Console Input	`-A`	Prompt for the parameters before every test from console
	`-B`	Batch mode
2. Managing Output	`-r[<file-name>]`	Create `<file-name>.csv` file and write PTT results to it
	`-R[<file-name>]`	Add test results to the file `<file-name>.csv`
	`-H[ONLY]`	Add 'Interest' column to `.csv` file [and run only hot tests]
	`-o[<file-name>]`	Create `<file-name>.txt` file and write console output to it
	`-O[<file-name>]`	Add console output to the file `<file-name>.txt`

Groups	Options	Descriptions
	`-L` *<ERR|WARN|PARM|INFO|TRACE>*	Set detail level of the console output
	`-u[<file-name>]`	Create `<file-name>.csv` file and write summary table ('_sum' is added to the file name)
	`-U[<file-name>]`	Add summary table to the file `<file-name>.csv` ('_sum' is added to the file name)
	`-e`	Enumerate tests and exit
	`-g[<file-name>]`	Create signal file just at the end of the whole testing
	`-s[-]`	Sort or don't sort functions (sort mode is default)
3. Selecting Functions for Testing	`-f <or-pattern>`	Run tests for functions with `pattern` in their names, case sensitive
	`-f-<not-pattern>`	Do not test functions with `pattern` in their names, case sensitive
	`-f+<and-pattern>`	Run tests only for functions with `pattern` in their names, case sensitive
	`-f=<eq-pattern>`	Run tests of functions with name `eq-pattern`, case sensitive
	`-F<func-name>`	Start testing from function with the full name `func-name`, case sensitive
4. Operation with .ini Files	`-i[<file-name>]`	Read PTT parameters from the file `<file-name>.ini`
	`-I[<file-name>]`	Write PTT parameters to the file `<file-name>.ini` and exit
	`-P`	Read tested function names from the `.ini` file
5. Adjust default directories and file names for input and output	`-n<title-name>`	Set default title name for `.ini` file and output files
	`-p<dir-name>`	Set default directory for `.ini` file and input test data files
	`-l<dir-name>`	Set default directory for output files
6. Direct Data Input	`-d<name>=<value>`	Set PTT parameter value
7. Process priority	`-Y<HIGH/NORMAL>`	Set high or normal process priority (normal is default)

Groups	Options	Descriptions
8. Setting environment	`-N<`*`num-threads`*`>`	Call `ippSetNumThreads(<`*`num-treads`*`>)`
9. Getting help	`-h`	Display short help and exit
	`-hh`	Display extended help and exit
	`-h<`*`option`*`>`	Display extended help for the specified option and exit

Intel(R) IPP Samples

B

Optimization Notice
The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors. While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Intel recommends that you evaluate other library products to determine which best meets your requirements.

This appendix describes the types of Intel IPP sample code available for developers to learn how to use Intel IPP, gives the source code example files by categories with links to view the sample code, and explains how to build and run sample applications.

Types of Intel IPP Sample Code

Intel IPP sample code are provided to help you to learn how to use the Intel Integrated Performance Primitives. They are intended only to demonstrate how to use the APIs and how to build applications in different development environments.

Table B-1 Types of Intel IPP Sample Code

Type	Description
Application-level samples	These samples illustrate how to build a wide variety of applications such as encoders, decoders, viewers, and players using the Intel IPP APIs (see Using Intel IPP Samples).
Source Code Samples	These platform independent examples show basic techniques for using Intel IPP functions to perform such operations as performance measurement, time-domain filtering, affine transformation, canny edge detection, and more. Each example consists of 1-3 source code files (`.cpp`) (see Source Code Samples).

Type	Description
Code Examples	These code examples (or code *snippets*) are very short programs demonstrating how to call a particular Intel IPP function. Numerous code examples are contained in the *Intel IPP Reference Manual* as part of the function descriptions.

Source Code Samples

The following table presents the files with the source code for the Intel IPP samples by categories and links to them.

Category	Summary	Description and Links
Basic Techniques	Introduction to programming with Intel IPP functions	Performance measurement: GetClocks.cpp Copying data: Copy.cpp Optimizing table-based Functions: LUT.cpp
Digital Filtering	Fundamentals of signal processing	Executing the: DFT.cpp Filtering with FFT: FFTFilter.cpp Time-domain filtering: FIR.cpp
Audio Processing	Audio signal generation and manipulation	Generating DTMF tones: DTMF.cpp Using IIR to create an echo: IIR.cpp Using FIRMR to resample a signal: Resample.cpp
Image Processing	Creating and processing a whole image or part of an image	Allocating, initializing, and copying an image: Copy.cpp Rectangle of interest sample wrapper: ROI.h ROI.cpp ROITest.cpp Mask image sample wrapper: Mask.h Mask.cpp MaskTest.cpp
Image Filtering and Manipulation	General image affine transformations	Wrapper for resizing an image: Resize.h Resize.cpp ResizeTest.cpp Wrapper for rotating an image: Rotate.h Rotate.cpp RotateTest.cpp Wrapper for doing an affine transform on an image: Affine.h Affine.cpp AffineTest.cpp
Graphics and Physics	Vector and small matrix arithmetic functions	ObjectViewer application: ObjectViewerDoc.cpp ObjectViewerDoc.h ObjectViewerView.cpp ObjectViewerView.h Transforming vertices and normals: `CTestView::OnMutateModel` Projecting an object onto a plane: `CTestView::OnProjectPlane` Drawing a triangle under the cursor: `CTestView::Draw`

Category	Summary	Description and Links
		Performance comparison, vector vs. scalar: perform.cpp Performance comparison, buffered vs. unbuffered: perform2.cpp
Special-Purpose Domains	Cryptography and computer vision usage	RSA key generation and encryption: rsa.cpp rsa.h rsatest.cpp bignum.h bignum.cpp Canny edge detection class: canny.cpp canny.h cannytest.cpp filter.h filter.cpp Gaussian pyramids class: pyramid.cpp pyramid.h pyramidtest.cpp

Using Intel IPP Samples

Download the Intel IPP application-level samples from http://www.intel.com/software/products/ipp/samples.htm.

These samples are updated in each version of Intel IPP. For best results upgrade the Intel IPP samples when a new version of Intel IPP is available.

Several common samples are included with the product. You can find them in the `<`*`ipp directory`*`>/samples/en_US/IPP`.

System Requirements

The specific system requirements for each sample are listed in the `readme.htm` document in the root directory of each sample. Most common requirements are listed below.

Hardware requirements:

- A system based on an Intel® Pentium® processor, Intel® Xeon® processor, or a subsequent IA-32 architecture-based processor
- A system based on processors supported the Intel® 64 architecture

Software requirements:

- Intel(R) IPP for the Linux* OS, version 7.0 or higher
- Red Hat Enterprise Linux* operating system version 4.0 or higher
- Intel(R) C++ Compiler for Linux OS: versions 12.0, 11.1; GNU C/C++ Compiler 3.2 or higher
- Qt* library runtime and development environment
- GNU Binutils version 2.15 or higher

Building Source Code

The specific building procedure is described in the `readme.htm` document in the root directory of each sample. Most common steps are described below.

Set up your build environment by creating an environment variable named `IPPROOT` that points to the root directory of your Intel IPP installation. For example, default `opt/intel/compiler` .

To build all samples, change your current directory to the root media samples directory and use shell script `build*.sh[option]`.

By default, the shell script searches the last version of compiler (assuming that compiler is installed in the default directory). If you wish to use specific version of the Intel C/C++ compiler or GCC software, set an option for the shell script in accordance with the following table:

Table B-2 Options for Shell Scripts

Compiler	Option
Intel C++ Composer XE 2011 for Linux OS	`icc120`
Intel C++ Compiler 11.1 for Linux OS	`icc111`
GCC 4.x.x	`gcc4`
GCC 3.4.x	`gcc3`

After the successful build, the executable file is placed in the corresponding sample directory:

`<`*`install_dir`*`>/ipp-samples/<sample-name>/bin/linux*_compiler` ,

where `compiler = icc111|icc120|gcc3|gcc4` .

Running the Software

To run each sample application, the Intel IPP shared object libraries must be on the system's path. See "Setting Environment Variables" for more details.

Detailed instructions on how to run the application, the list of command line options or menu commands are given in the `readme.htm` document in the root directory of each sample.

Known Limitations

The applications created with the Intel IPP Samples are intended to demonstrate how to use the Intel IPP functions and help you to create your own software. These sample applications have some limitations. The specific limitations for each sample are described in the section *"Known Limitations"* in the `readme.htm` document in the root directory of each sample.

Language Support

C

The following table lists the different languages that Intel(R) IPP supports. It also provides information on the Intel IPP samples available for each language.

Language Support

Language	Environment	The Sample Description
C++	Makefile Intel C++ compiler, GNU C/C++ compiler	Overloads the Intel IPP C-library functions to create classes for easy signal and image manipulation.
Fortran	Makefile	N/A
Java*	Java Development Kit 1.5.0	Uses the Intel IPP image processing functions in a Java wrapper class.

Optimization Notice

The Intel® Integrated Performance Primitives (Intel® IPP) library contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in the Intel® IPP library offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for the Intel® IPP library as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

See Also

Download the samples from http://www.intel.com/software/products/ipp/samples.htm

Using Intel IPP in Java* Applications

You can call Intel IPP functions in your Java application by using the Java* Native Interface (JNI*). There is some overhead associated with JNI use, especially when the input data size is small. Combining several functions into one JNI call and using managed memory can improve the overall performance.

Index