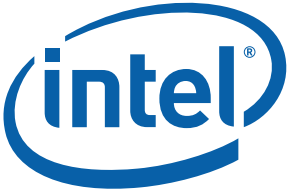


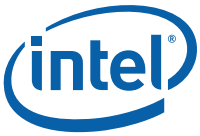
Intel® Integrated Performance Primitives for Linux* OS on IA-32 Architecture

User's Guide

March 2009

Document Number: 320271-003US

World Wide Web: http://developer.intel.com



Version	Version Information	Date
-001	Intel® Integrated Performance Primitives (Intel® IPP) for Linux* OS on IA-32 Architecture User's Guide. Documents Intel IPP 6.0 release.	September 2008
-002	Documents Intel IPP 6.1 beta release	January 2009
-003	Documents Intel IPP 6.1 release	March 2009

Contents

Overview

1

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, speech recognition, 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, GSM-AMR, G.723.

By supporting a variety of data types and layouts for each function and minimizing the number of data structures used, the Intel IPP library delivers a rich set of options for developers to choose from when designing and optimizing an application. A variety of data types and layouts are supported for each function. Intel IPP software minimizes data structures to give the developer the greatest flexibility for building optimized applications, higher level software components, and library functions.

Intel IPP for Linux* OS is delivered in separate packages for:

- Users who develop on 32-bit Intel architecture (Intel IPP for the Linux* OS on IA-32 Intel® Architecture)
- Users who develop on Intel® 64-based (former Intel EM64T) architecture (Intel IPP for the Linux* OS on Intel® 64 Architecture)
- Uusers who develop on Intel® Itanium® 2 processor family (Intel IPP for the Linux* OS on IA-64 architecture)
- Users who develop on Intel® Atom™ processor (Intel IPP for the Linux* OS on low power Intel® Architecture)

Technical Support

Intel IPP provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, see http://developer.intel.com/software/products/.

Intel also provides a support web site that contains a rich repository of self-help information, including getting started tips, known product issues, product errata, license information, and more (visit http://support.intel.com/support/).

Registering your product entitles you to one-year technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing the following services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, or contact Intel, or seek product support, please visit

http://www.intel.com/software/products/support/ipp.

About This Document

This User's Guide provides information about how to make the most of Intel® IPP routines using *Linux** applications running on IA-32 architecture. It describes features specific to this platform, as well as features that do not depend upon a particular architecture.

After installation, you can find this document in the `<install path>/doc` directory (see Contents of the Documentation Directory).

Purpose

This document:

- Helps you start using the library by describing the steps you need to follow after installation of the product.
- Shows how to configure the library and your development environment to use the library.
- Acquaints you with the library structure.
- Explains in detail how to select the best linking method, how to link your application to the library, and it provides simple usage examples.
- Explains how to thread your application using IPP software.
- Describes how to code, compile, and run your application with Intel IPP.

- Provides information about how to accomplish Intel IPP functions performance tests by using Intel IPP Performance Test Tool.
- Describes types of Intel IPP sample code available for developers to learn how to use Intel IPP and it explains how to run the samples.

Audience

This guide is intended for Linux programmers with beginner to advanced software development experience.

Document Organization

The document contains the following chapters and appendices.

Chapter 1	Overview describes the document purpose and organization as well as explains notational conventions.
Chapter 2	Getting Started with Intel® IPP describes necessary steps and gives basic information needed to start using Intel IPP after its installation.
Chapter 3	Intel® IPP Structure describes the structure of the Intel IPP directory after installation and discusses the library types supplied.
Chapter 4	Configuring Your Development Environment explains how to configure Intel IPP and how to configure your environment for use with the library.
Chapter 5	Linking Your Application with Intel® IPP compares linking methods, helps you select a linking method for a particular purpose, describes the general link line syntax to be used for linking with the Intel IPP libraries, and discusses how to build custom dynamic libraries.
Chapter 6	Supporting Multithreaded Applications helps you set the number of threads in multithreaded applications, get information on the number of threads, and disable multithreading.
Chapter 7	Managing Performance and Memory discusses ways of improving Intel IPP performance and tells you how to create Intel IPP functions performance tests by using the Intel IPP Performance Test Tool.
Chapter 8	Using Intel® IPP with Programming Languages discusses some special aspects of using Intel IPP with different programming languages and Linux development environments.
Appendix A	Performance Test Tool Command Line Options gives brief descriptions of possible performance test tool command line options.
Appendix B	Intel® IPP Samples describes types of sample code available to demonstrate how to use Intel IPP, presents the source code example files by categories with links to view the sample code, and explains how to run the samples.

The document also includes an Index.

Notational Conventions

The document usess the following font conventions and symbols:

Table 1-1 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/env/ippvars32.csh`
`Monospace lowercase mixed with uppercase`	Indicates code, commands, and command-line options, for example: `export LIB=$IPPROOT/lib:$LIB`
`UPPERCASE MONOSPACE`	Indicates system variables, for example, `LD_LIBRARY_PATH`
`Monospace italic`	Indicates a parameter in discussions, such as function parameters, for example, *`lda`*; makefile parameters, for example, *`functions_list`*; and so on. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value: *`<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

Getting Started with Intel® IPP

2

This chapter helps you start using Intel® IPP by providing basic information you need to know and describing the necessary steps you need to follow after installation of the product.

Intel IPP Basics

Intel IPP is a collection of high-performance code that provides a broad range of functionality. This functionality includes general signal and image processing, computer vision, speech recognition, data compression, cryptography, string manipulation, audio processing, video coding, realistic rendering and 3D data processing, matrix math. 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, GSM-AMR, G.723.

Based on experience in developing and using the Intel Performance Libraries, Intel IPP has the following major distinctive features:

- Intel IPP provides basic low-level functions for creating applications in several different domains, such as signal processing, audio coding, speech recognition and 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 is best suited to achieve superior performance figures by the application programs.

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

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® Itanium®-based platforms (see Cross-Architecture Alignment).

Cross-Architecture Alignment

Intel IPP is designed to support application development on various Intel® 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 developers to port to unique features across Intel® processor-based desktop, server, and mobile platforms. Developers can write their code once in order to realize the application performance over many processor generations.

Types of Input Data

Intel IPP operations are divided into several groups in dependence on the types of input data on which the operation is performed. The types for these groups are:

One-Dimensional Arrays and Signals

This group includes most functions operating on one-dimensional arrays of data. In many cases these array 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 recognition and speech coding
- cryptography and data integrity
- string operations

Images

An image is an two-dimensional array of pixels. Images have some specific features that distinguishes 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

The Intel IPP functions are primarily grouped according to the input data types listed above. Each group has its own prefix in the function name (see Function Naming).

Core Functions

A few service functions in Intel IPP do not operate on one of these input data type. Such functions are used to detect and set system and Intel IPP configuration. 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 and so on. These functions are called core functions and have its own header file, static libraries and SOs.

Table 2-1

Code	Header File	Static Libraries	SO	Prefix in Function Name
ippCore	ippcore.h	libippcore.a libippcore_t.a	libippcore.so.*.*	ipp

here *.* refers to the product version number, for example `6.1`

Domains

For organizational purposes Intel IPP is internally divided into subdivisions of related functions. Each subdivision is called domain, (or functional domain) and generally has its own header file, static libraries, DLLs, and tests. These domains map easily 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.

Table 2-2

Code	Header file	Static Libraries	SO	Prefix	Description
ippAC	ippac.h	libippac*.a	libippac**.so.***	ipps	audio coding
ippCC	ippcc.h	libippcc*.a	libippcc**.so.***	ippi	color conversion
ippCH	ippch.h	libippch*.a	libippch**.so.***	ipps	string operations
ippCP	ippcp.h	libippcp*.a	libippcp**.so.***	ipps	cryptography
ippCV	ippcv.h	libippcv*.a	libippcv**.so.***	ippi	computer vision
ippDC	ippdc.h	libippdc*.a	libippdc**.so.***	ipps	data compression
ippDI	ippdi.h	libippdi*.a	libippdi**.so.***	ipps	data integrity
ippGEN	ipps.h	libippgen*.a	libippgen**.so.***	ippg	generated functions
ippIP	ippi.h	libippi*.a	libippi**.so.***	ippi	image processing
ippJP	ippj.h	libippj*.a	libippj**.so.***	ippi	image compression
ippMX	ippm.h	libippm*.a	libippm**.so.***	ippm	small matrix operations
ippRR	ippr.h	libippr*.a	libippr**.so.***	ippr	realistic rendering and 3D data processing
ippSC	ippsc.h	libippsc*.a	libippsc**.so.***	ipps	speech coding
ippSP	ipps.h	libipps*.a	libipps**.so.***	ipps	signal processing

Table 2-2

Code	Header file	Static Libraries	SO	Prefix	Description
ippSR	ippsr.h	libippsr*.a	libippsr**.so.***	ipps	speech recognition
ippVC	ippvc.h	libippvc*.a	libippvc**.so.***	ippi	video coding
ippVM	ippvm.h	libippvm*.a	libippvm**.so.***	ipps	vector math

* - refers to one of the following: `emerged`, `merged`, `merged_t`

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

*** - refers to the version number, for example, 6.1

Function Naming and Parameters

Function names in Intel IPP are structured in order to simplify their identification and use. Understanding Intel IPP naming conventions can save you a lot of time and effort in figuring out what the purpose of a specific function is and in many cases you can derive this basic information straight from the function's self-explanatory name.

Naming conventions for the Intel IPP functions are similar for all covered domains.

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

Function names have the following general format:

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

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

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 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, Copy. 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, where the number of permutations of the function becomes large and unreasonable, some functions 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, alpha channel is not processed.
`A0`	Image data contains an alpha channel as the first channel, requires C4, alpha channel is not processed.
`Axx`	Specifies the bits of accuracy of the result for advanced arithmetic operations.
`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`	The operation is performed in-place (default is not-in-place).
`L`	Layout description of the objects for matrix operation, or indicates that one pointer is used for each row in D2 array for signal processing.
`M`	The operation uses a mask to determine pixels to be processed.
`P`	Pointer description of the objects for matrix operation, or specified number of vectors to be processed for signal processing.
`Pn`	Image data is made of n discrete planar (non-interleaved) channels (n= 1, 2, 3, 4) with separate pointer to each plane.
`R`	The function operates on a defined region of interest (ROI) for each source image.
`S`	Standard description of the objects for matrix operation.
`Sfs`	Saturation and fixed scaling mode (default is saturation and no scaling).
`s`	Saturation and no scaling.

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

Some data descriptors are implied when dealing with certain operations. For example, the default for image processing functions is to operate on a two-dimensional image and to saturate the results without scaling them. In these cases, the implied abbreviations `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 parameters name has the following conventions.

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 names or numbers, for example, *pSrc2*, *srcLen*. Output arguments are named *pDst* or *dst* followed by names or numbers, 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).
```

Checking Your Installation

Once you complete the installation of Intel IPP, it is useful to follow these steps that confirm proper installation and configuration of the library.

1. Check that the directory you chose for installation has been created: `<installation path>/intel/ipp/6.1.x.xxx/ia32`. The default installation directory is `opt/intel/ipp/6.1.x.xxx/ia32`.
2. Check that file `ippvars32.sh` is placed in the `tools/env` 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.
4. If you receive the error message "No shared object library was found in the Waterfall procedure", this means that the Linux 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 object libraries, add path to the shared object libraries to the system variable `LD_LIBRARY_PATH` as described in Using Intel IPP Shared Object Libraries (SO) in Chapter 3;

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 IPP Reference Manual" (v.1)* for the function description and calling syntax.

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

Building Your Application

Follow the procedures described below to build the application.

Setting Environment Variables

The shell script `ippvars32.sh` in the `tools/env` directory sets your `LD_LIBRARY_PATH`, `LIB`, and `INCLUDE` environment variables for Intel IPP.

To set environment variables manually, add the path to the shared object libraries to the `LD_LIBRARY_PATH` variable as described in Using Intel IPP Shared Object Libraries (SO) in Chapter 3. You will 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.

For information on how to set up environment variables for threading, refer to Supporting Multithreaded Applications.

Including Header Files

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

The file `ipp.h` includes all Intel IPP header files. For forward compatibility, include only `ipp.h` in your program.

Calling IPP Functions

Due to the shared library dispatcher and merged static library mechanisms described in Linking Your Application with Intel® 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 IPP Reference Manual"* for 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.

Table 2-3 summarizes important things to consider before you start using Intel IPP.

Table 2-3 What you need to know before you get started

Function domains	Identify the Intel IPP function domain that meets your needs. **Reason**: If you know function domain you intend to use will narrow the search in the Reference Manuals for specific routines you need. Besides, you may easily find a sample you would like to run from http://www.intel.com/software/products/ipp/samples.htm. Refer to Table 5-9 to understand what function domains are and what libraries are needed, and to Table 5-10 to understand what kind of cross-domain dependency is introduced.
Linking method	Decide what linking method is appropriate for linking. **Reason:** If you choose a linking method that suits, you will get the best linking results. For information on the benefits of each linking method, linking command syntax and examples, as well as on other linking topics, such as how to create a custom dynamic library, see Linking Your Application with Intel® 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® 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® IPP Structure 3

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

High-level Directory Structure

Table 3-1 shows the high-level directory structure of Intel IPP after installation.

Table 3-1 High-level directory structure

Directory	File types
<ipp directory>	Main directory (by default: /opt/intel/ipp/6.1.x.xxx/ia32)
<ipp directory>/ippEULA.txt	End User License Agreement for Intel IPP
<ipp directory>/doc	Intel IPP documentation files
<ipp directory>/include	Intel IPP header files
<ipp directory>/lib	Intel IPP static libraries
<ipp directory>/sharedlib	Intel IPP shared object libraries
<ipp directory>/tools	Intel IPP Performance Test tool, linkage tools, and tool to set environment variables

Supplied Libraries

Table 3-2 lists the types of libraries in Intel IPP and shows examples of the library files supplied:

Table 3-2 Types of Libraries of Intel IPP

Library types	Description	Folder location	Example
Dynamic	Shared object libraries include both processor dispatchers and function implementations	ia32/sharedlib	libipps.so.6.1, libippst7.so.6.1
	Soft links to the shared object libraries	ia32/sharedlib	libipps.so libippst7.so
Static merged	Contain function implementations for all supported processor types:		
	libraries with position independent code (PIC)	ia32/lib	libippsmerged.a
	non-PIC)* libraries	ia32/lib/nonpic	libippsmerged.a
Threaded static merged	Contain threaded function implementations	ia32/lib	libippsmerged_t.a
Static emerged	Contain dispatchers for the merged libraries:		
	libraries with position independent code (PIC)	ia32/lib	libippsemerged.a
	non-PIC*) libraries	ia32/lib/nonpic	libippsemerged.a

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

Using Intel IPP Shared Object Libraries (SO)

Intel IPP comes with the shared object libraries (SO) and soft links to them in the ia32/sharedlib 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 ippvars32.sh in the tools/env directory.

Alternatively you can set the variable LD_LIBRARY_PATH manually. For example, if the libraries are in the /opt/intel/ipp/6.1.x.xxx/ia32/sharedlib directory, to set the environment variable manually, enter the following command line for bash:

```
export LD_LIBRARY_PATH=
        /opt/intel/ipp/6.1.x.xxx/ia32/sharedlib:$LD_LIBRARY_PATH
```

or for csh:

```
setenv LD_LIBRARY_PATH=
        /opt/intel/ipp/6.1.x.xxx/ia32/sharedlib:${LD_LIBRARY_PATH}
```

The shared libraries libipp*.so.6.1 (* denotes the appropriate function domain) are "dispatcher" dynamic libraries. At run time, they detect the processor and load the correct processor-specific shared libraries. This allows you to write code to call the Intel IPP functions without worrying about which processor the code will execute on - the appropriate version is automatically used. These processor-specific libraries are named libipp*px.so.6.1, libipp*w7.so.6.1, libipp*t7.so.6.1, libipp*v8.so.6.1, and libipp*p8.so.6.1 (see Table 5-4). For example, in the ia32/sharedlib directory, libippiv8.so.6.1 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: libipp*-6.1.so, libipp*px-6.1.so, libipp*w7-6.1.so, libipp*t7-6.1.so, libipp*v8-6.1.so, and libipp*p8-6.1.so.

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

NOTE. You must include the appropriate libiomp5.so in your LD_LIBRARY_PATH environment variable. Include the directory sharelib when running on a system with IA-32 architecture.

Using Intel IPP Static Libraries

The Intel IPP comes with "merged" static library files that contain every processor version of each function. These files reside in the ia32/lib directory (see Table 3-1).

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*merged.a` in the `lib` directory. Then follow the directions in Intel IPP Linkage Samples and create the dispatching stubs for just the functions that you need. You will either need to set your LIB environment variable using the shell script file `ippvars32.sh` or refer to these files using their full path.

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

Contents of the Documentation Directory

Table 3-3 shows the content of the `/doc` subdirectory in the Intel IPP installation directory.

Table 3-3 Contents of the `/doc` Directory

File name	Description	Notes
`ipp_documentation.htm`	Documentation index. Lists the principal Intel IPP documents with appropriate links to the documents	
`ReleaseNotes.pdf`	General overview of the product and information about this release.	These files can be viewed prior to the product installation
`README.txt`	Initial User Information	
`INSTALL.htm`	Installation guide	
`ThreadedFunctionsList.txt`	List of all Intel IPP functions threaded with OpenMP*	
`userguide_lnx_ia32.pdf`	Intel® Integrated Performance Primitives User's Guide, this document	
Intel IPP Reference Manual (in four volumes):		
`ippsman.pdf`	*Signal Processing* (vol.1) - contains detailed descriptions of Intel IPP functions and interfaces for signal processing, audio coding, speech recognition and coding, data compression and integrity, string operations and vector arithmetic.	
`ippiman.pdf`	*Image and Video Processing* (vol.2) - contains detailed descriptions of Intel IPP functions and interfaces for image processing and compression, color and format conversion, computer vision, video coding.	

Table 3-3 Contents of the /doc Directory

File name	Description	Notes
ippmman.pdf	*Small Matrices, Realistic Rendering* (vol.3) - contains detailed descriptions of Intel IPP functions and interfaces for vector and matrix algebra, linear system solution, least squares and eigenvalue problems as well as for realistic rendering and 3D data processing.	
ippcpman.pdf	*Cryptography* (vol.4) - contains detailed descriptions of Intel IPP functions and interfaces for cryptography.	

Configuring Your Development Environment

4

This chapter explains how to configure your development environment for the use with Intel® 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/ipp/6.1.x.xxx/include`, where `x.xxx` is the Intel IPP package number.
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/ipp/6.1.x.xxx/lib/ia32`.
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. See section Selecting the 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 that the compiler/linker will automatically pick up the include and library paths settings only in case the automatic makefile generation is turned on, otherwise, you will have to 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:

For 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/ipp/6.1.x.xxx/include` where `x.xxx` is the Intel IPP package number.
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, See section Selecting the Intel IPP Libraries Needed by Your Application in chapter 5 on the choice of the libraries.

Note that with the Standard Make, the above settings are needed for the CDT internal functionality only. The compiler/linker will not automatically pick up these settings and you will still have to specify them directly in the makefile.

For Managed Make projects:

Specify settings for a particular build. To do this,

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/ipp/6.1.x.xxx/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/ipp/6.1.x.xxx/lib/ia32`.
4. Specify names of the Intel IPP libraries to link with your application. See section Selecting the 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® IPP

5

This chapter discusses linking Intel 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 the user select the linking method that suits him best, shows linking procedure for each linking method, and gives linking examples.

Dispatching

Intel IPP uses codes optimized for various central processing units (CPUs). Dispatching refers to detection of your CPU and selecting the Intel IPP binary that corresponds to the hardware that you are using. For example, in the `ia32/sharedlib` directory, file `libippiv8.so.6.1` reflects the optimized imaging processing libraries for Intel® 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® processor with specific architecture, for example: the version of this function optimized for the Pentium® 4 processor is `w7_ippsCopy_8u`().

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
px	C-optimized for all IA-32 processors
w7	Optimized for processors with Intel® Streaming SIMD Extensions 2 (Intel SSE2)
t7	Optimized for processors with Intel® Streaming SIMD Extensions 3 (Intel SSE3)
v8	Optimized for processors with Intel® Supplemental Streaming SIMD Extensions 3 (Intel SSSE3)

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

Abbreviation	Meaning
p8	Optimized for processors with Intel® Streaming SIMD Extensions 4.1 (SSE4.1)
s8	Optimized for the Intel® Atom™ processor.

Processor Type and Features

Processor Features

To obtain information about the features of the processor used in your computer system, use function `ippGetCpuFeatures`, which is declared in the `ippcore.h` file. This function retrieves main CPU features returned by the function `CPUID.1` and stores them consecutively in the mask that is returned by the function. Table 5-2 lists all CPU features that can be retrieved (see more details in the description of the function `ippGetCpuFeatures` in the *Intel IPP Refrence Manual*, vol.1).

Table 5-2 Processor Features

Mask Value	Name	Feature
1	`ippCPUID_MMX`	MMX™ technology
2	`ippCPUID_SSE`	Intel® Streaming SIMD Extensions
4	`ippCPUID_SSE2`	Intel® Streaming SIMD Extensions 2
8	`ippCPUID_SSE3X`	Intel® Streaming SIMD Extensions 3
16	`ippCPUID_SSSE3`	Supplemental Intel® Streaming SIMD Extensions
32	`ippCPUID_MOVBE`	MOVBE instruction is supported
64	`ippCPUID_SSE41`	Intel® Streaming SIMD Extensions 4.1
128	`ippCPUID_SSE42`	Intel® Streaming SIMD Extensions 4.2
256	`ippCPUID_AVXX`	Intel® Advanced Vector Extensions (Intel AVX) instruction set is supported
512	`ippAVX_ENABLEDBYOS`	The operating system supports Intel AVX
1024	`ippCPUID_AES`	AES instruction is supported
2048	`ippCPUID_CLMUL`	PCLMULQDQ instruction is supported

Processor Type

To detect the processor type used in your computer system, use 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 `ippCpuCoreDuo` means that your system uses Intel® Core™ Duo processor.

Table 5-3 shows possible values of `ippGetCpuType` and their meaning.

Table 5-3 Detecting processor type. Returned values and their meaning

Returned Variable 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
`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

Table 5-3 Detecting processor type. Returned values and their meaning (continued)

Returned Variable Value	Processor Type
ippCpuSSE3	Processor with Intel® Streaming SIMD Extensions 3 instruction set
ippCpuSSSE3	Processor with Supplemental Intel® 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
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 libraries (SOs)
- Static linking with dispatching using emerged and merged static libraries
- Static linking without automatic dispatching using merged static libraries
- Dynamic linking with your own - custom - SO.

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

- Are there limitations on how large the application executable can be? Are there limitations on how large the application installation package can be?
- 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 various users install the application 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 where only one type of processor is used?
- 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

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

Table 5-4 Summary of Dynamic Linking Features

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

To dynamically link with Intel IPP, follow these steps:

1. Add `ipp.h`, which includes the header files of all IPP domains.
2. Use the normal IPP function names when calling IPP functions.
3. Link corresponding domain soft links. For example, if you use the `ippsCopy_8u` function, link against `libipps.so`.
4. Make sure that you run `<install path>/tools/env/ippvars32.sh` shell script before using Intel IPP libraries in the current session, or set `LD_LIBRARY_PATH` correctly. For example,
   `export LD_LIBRARY_PATH =$IPPROOT/sharedlib:$LD_LIBRARY_PATH` (bash), or
   `setenv LD_LIBRARY_PATH =$IPPROOT/sharedlib:${LD_LIBRARY_PATH}` (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 via the *emerged* and *merged* libraries offers both the benefits of a small footprint and optimization on multiple processors. The emerged libraries (such as `libippsemerged.a`) provide an entry point for the non-decorated (with normal names) IPP functions, and the jump table to each processor-specific implementation. When linked with your application, the function calls corresponding functions in the merged libraries (such as `libippsmerged.a`) in accordance with the CPU setting detected by functions in `libippcore.a`. The emerged libraries do not contain any implementation code.

The emerged libraries must be initialized before any non-decorated functions can be called. One may choose the function `ippStaticInit()` that initializes the library to use the best optimization available, or the function `ippStaticInitCpu()` that lets you specify the CPU. In any case, one of these functions must be called before any other IPP functions. Otherwise, a "`px`" version of the IPP functions will be called, which can decrease the performance of your application. Example 5-1 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
```

Example 5-1 Performance difference with and without calling StaticInit

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

When you perform static linking via the emerged libraries, there are things you should consider. Table 5-5 summarizes the pros and cons of this type of static linking.

Table 5-5 Summary of Features of the Static Linking (with Dispatching)

Benefits	Considerations
• 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 code.
2. Before calling any Intel IPP functions, initialize the static dispatcher using either the function `ippStaticInit()` or `ippInitCpu()`, which are declared in the header file `ippcore.h`.
3. Use the normal IPP function names to call IPP functions.
4. Link corresponding emerged libraries followed by merged libraries, and then `libippcore.a`. For example, if the function `ippsCopy_8u()` is used, the linked libraries are `libippsemerged.a`, `libippsmerged.a`, and `libippcore.a`.

Static Linking (without Dispatching)

This method uses linking directly with the merged static libraries. You may want to use your own static dispatcher instead of the provided emerged dispatcher. The IPP sample `mergelib` demonstrates how to do this.

Please 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.

When a self-contained application is needed, only one processor type is supported and there are tight constraints on the executable size. One common use for embedded applications is when the application is bundled with only one type of processor.

Table 5-6 summarizes basic features of this method of linking.

Table 5-6 Summary of Features of the Static Linking (without dispatching)

Benefits	Considerations
• Small executable size with support for only one processor type • An executable suitable for kernel-mode/device-driver/ring-0 use*) • An executable 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 to run • 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 not-threaded non-PIC libraries only

You can use alternatives to the above procedure. The Intel IPP package includes a set of processor-specific header files (such as `ipp_w7.h`). You can use these header files instead of the `IPPCALL` macro. Refer to *Static linking to Intel® IPP Functions for One Processor* in `ia32/tools/staticlib/readme.htm`.

Building a Custom SO

Some applications have few internal modules and the Intel IPP code needs to be shared by these modules only. In this case, you can use dynamic linking with the customized shared object library (SO) containing only those Intel IPP functions that the application uses. Table 5-7 summarizes features of the custom SOs.

Table 5-7 Custom SO Features

Benefits	Considerations
• 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 OSs • 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 it can be done. Please 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 Linkage Methods

Table 5-8 gives a quick comparison of the IPP linkage methods.

Table 5-8 Intel IPP Linkage Method Summary Comparison

Feature	Dynamic Linkage	Static Linkage with Dispatching	Static Linkage 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 or threaded merged libraries	Build separate SO
Calling	Regular names	Regular names	Processor-specific names	Regular names
Total Binary Size	Large	Small	Smallest	Small
Executable Size	Smallest	Small	Small	Smallest
Kernel Mode	No	Yes	Yes	No

Selecting the Intel IPP Libraries Needed by Your Application

Table 5-9 shows functional domains and the relevant header files and libraries used for each linkage method.

Table 5-9 Libraries Used for Each Linkage Method

Domain Description	Header Files	Dynamic Linking	Static Linking with Dispatching and Custom Dynamic Linking	Static Linking without Dispatching
Audio Coding	`ippac.h`	`libippac.so`	`libippacemerged.a`	`libippacmerged.a` `libippacmerged_t.a`

Table 5-9 Libraries Used for Each Linkage Method (continued)

Domain Description	Header Files	Dynamic Linking	Static Linking with Dispatching and Custom Dynamic Linking	Static Linking without Dispatching
Color Conversion	ippcc.h	libippcc.so	libippccemerged.a	libippccmerged.a libippccmerged_t.a
String Processing	ippch.h	libippch.so	libippchemerged.a	libippchmerged.a libippchmerged_t.a
Cryptography	ippcp.h	libippcp.so	libippcpemerged.a	libippcpmerged.a libippcpmerged_t.a
Computer Vision	ippcv.h	libippcv.so	libippcvemerged.a	libippcvmerged.a libippcvmerged_t.a
Data Compression	ippdc.h	libippdc.so	libippdcemerged.a	libippdcpmerged.a libippdcmerged_t.a
Data Integrity	ippdi.h	libippdi.so	libippdipemerged.a	libippdimerged.a libippdimerged_t.a
Generated Functions	ipps.h	libippgen.so	libippgenemerged.a	libippgenmerged.a libippgenmerged_t.a
Image Processing	ippi.h	libippi.so	libippiemerged.a	libippimerged.a libippimerged_t.a
Image Compression	ippj.h	libippj.so	libippjemerged.a	libippjmerged.a libippjmerged_t.a
Realistic Rendering and 3D Data Processing	ippr.h	libippr.so	libippremerged.a	libipprmerged.a libipprmerged_t.a
Small Matrix Operations	ippm.h	libippm.so	libippmemerged.a	libippmmerged.a libippmmerged_t.a
Signal Processing	ipps.h	libipps.so	libippsemerged.a	libippsmerged.a libippsmerged_t.a
Speech Coding	ippsc.h	libippsc.so	libippscpemerged.a	libippscmerged.a libippscpmerged_t.a
Speech Recognition	ippsr.h	libippsr.so	libippsremerged.a	libippsrmerged.a libippsrmerged_t.a

Table 5-9 Libraries Used for Each Linkage Method (continued)

Domain Description	Header Files	Dynamic Linking	Static Linking with Dispatching and Custom Dynamic Linking	Static Linking without Dispatching
Video Coding	ippvc.h	libippvc.so	libippvcemerged.a	libippvcmerged.a libippvcmerged_t.a
Vector Math	ippvm.h	libippvm.so	libippvmemerged.a	libippvmmerged.a libippvmmerged_t.a
Core Functions	ippcore.h	libippcore.so	libippcore.a	libippcore.a libippcore_t.a

Dynamic Linkage

To use the shared objects, you must use the soft link `libipp*.so` files in the `sharedlib` directory, where * denotes the appropriate function domain. You must also link to all corresponding domain libraries used in your applications plus the libraries `libipps.so`, `libippcore.so`, and `libiomp.so`.

For example, consider that 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 Intel IPP libraries:

`libippi.so`

`libippcv.so`

`libippm.so`

`libippcore.so`

`libiomp.so`

Static Linkage with Dispatching

To use the static linking libraries, you need to link to `lib*emerged.a`, `lib*merged.a`, `libsemerged.a`, `libsmerged.a`, and `libcore.a`. The * denotes the appropriate function domain.

If you want to use the Intel IPP functions threaded with the OpenMP*, you need to link to `lib*emerged.a`, `lib*merged_t.a`, `libsemerged.a`, `libsmerged_t.a`, `libcore_t.a`, and `libiomp5.a`.

All these libraries are located in the `lib` directory containing domain-specific functions. Note that both merged and emerged libraries for all domains plus the signal processing domain must be linked to your application.

For example, consider that 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. Note the order in which libraries are linked must correspond to the library dependencies by domain (see below). If you want to use the threaded functions, you must link the following libraries to your application:

`libcvemerged.a` and `libcvmerged_t.a`

`libmemerged.a` and `libmmerged_t.a`

`libiemerged.a` and `libimerged_t.a`

`libsemerged.a` and `libsmerged_t.a`

`libcore_t.a`

`libiomp5.a`

Library Dependencies by Domain (Static Linkage Only)

Table 5-10 lists library dependencies by domain. When you link to a certain library (for example, data compression domain), you must link to the libraries on which it depends (in our example, the signal processing and core functions).

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

Table 5-10 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`
Generated Functions	`ippgen`	`ipps`, `ippcore`
Image Processing	`ippi`	`ipps`, `ippcore`
Image Compression	`ippj`	`ippi`, `ipps`, `ippcore`

Table 5-10 Library Dependencies by Domain

Domain	Library	Dependent on
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
Speech Recognition	ippsr	ipps, ippcore
String Processing	ippch	ipps, ippcore
Video Coding	ippvc	ippi, ipps, ippcore
Vector Math	ippvm	ippcore

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

Linking Examples

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

For information on using sample code, please see "Intel® IPP Samples".

Supporting Multithreaded Applications

6

This chapter discusses the use of Intel® IPP in multithreading applications.

Intel IPP Threading and OpenMP* Support

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

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

Refer to the *ThreadedFunctionsList.txt* document to see the list of all threaded functions in the `doc` 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

The default number of threads for Intel IPP threaded libraries is equal to the number of processors in the system and does not depend on the value of the `OMP_NUM_THREADS` environment variable.

To set another number of threads used by Intel IPP internally, call the function `ippSetNumThreads(n)` at the very beginning of an application. Here `n` is the desired number of threads (1,...). If internal parallelization is not desired, call `ippSetNumThreads(1)`.

Using Shared L2 Cache

Some functions in the signal processing domain are threaded on two threads intended for the Intel® Core™2 processor family, and exploit the advantage of a 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, a special OpenMP environmental variable must be set:

```
KMP_AFFINITY=compact
```

Otherwise, the performance may degrade significantly.

Nested Parallelization

If the multithreaded application created with OpenMP uses the threaded Intel IPP function, this function will operate in a single thread because the nested parallelization is disabled in OpenMP by default.

If the multithreaded application created with other tools uses the threaded Intel IPP function, it is recommended that you disable multithreading in Intel IPP to avoid nested parallelization and to avoid possible performance degradation.

Disabling Multithreading

To disable multi-threading, call function `ippSetNumThreads` with parameter 1, or link your application with IPP non-threaded static libraries.

Managing Performance and Memory

7

This chapter describes ways you can get the most out of the Intel® IPP software such as aligning memory, thresholding denormal data, reusing buffers, and using Fast Fourier Transform (FFT) for algorithmic optimization (where appropriate). Finally, it gives information on how to accomplish the Intel IPP functions performance tests by using the Intel IPP Performance Test Tool and it gives some examples of using the Performance Tool Command Lines.

Memory Alignment

The performance of Intel IPP functions can be significantly different when operating on aligned or misaligned data. Access to memory is faster if pointers to the data are aligned.

Use the following Intel IPP functions for pointer alignment, memory allocation and deallocation:

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

Aligns a pointer, can align to 2/4/8/16/…

`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 `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 `ippiMalloc`.

Example 7-1 demonstrates how the function `ippiMalloc` can be used. The amount of memory that can be allocated is determined by the operating system and system hardware, but it cannot exceed 2GB.

NOTE. Intel IPP memory functions are wrappers of the standard `malloc` and `free` functions that align the memory to a 32-byte boundary for optimal performance on the Intel architecture.

NOTE. The Intel IPP functions `ippFree`, `ippsFree`, and `ippiFree` can only be used to free memory allocated by the functions `ippMalloc`, `ippsMalloc` and `ippiMalloc`, respectively.

NOTE. The Intel IPP functions `ippFree`, `ippsFree`, and `ippiFree` cannot be used to free memory allocated by standard functions like `malloc` or `calloc`; nor can the memory allocated by the Intel IPP functions `ippMalloc`, `ippsMalloc`, and `ippiMalloc` be freed by the standard function `free`.

Example 7-1 Calling the ippiMalloc function

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


void ipView(Ipp8u* img, int stride, char* str)
{
   int w, h;
   printf("%s:\n", str);
   Ipp8u* p = img;
   for( h=0; h<8; h++ ) {
      for( w=0; w<8*3; w++ ) {
         printf(" %02X", *(p+w));
      }
      p += stride;
      printf("\n");
   }
}

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

   int stride;
   Ipp8u* pSrc = ippiMalloc_8u_C3(size.width, size.height, &stride);
   printf("pSrc=%p, stride=%d\n", pSrc, stride);
   ippiImageJaehne_8u_C3R(pSrc, stride, size);
   ipView(pSrc, stride, "Source image");

   int dstStride;
   Ipp8u* pDst = ippiMalloc_8u_C3(size.width, size.height, &dstStride);
   printf("pDst=%p, dstStride=%d\n", pDst, dstStride);
   ippiCopy_8u_C3R(pSrc, stride, pDst, dstStride, size);
   ipView(pDst, dstStride, "Destination image 1");

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

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

   ippiFree(pDst);
   ippiFree(pSrc);

   return 0;
}
```

Thresholding Data

Denormal numbers are the border values in the floating-point format and special case values for the processor. Operations on denormal data make processing slow, even if corresponding interrupts are disabled. 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. 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-6` 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 two functions is faster than execution of one 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.

In this case, for Intel processors beginning with the Intel® Pentium® 4 processor, it is possible to set special computation modes - flush-to-zero (FTZ) and the denormals-are-zero (DAZ). You can use functions `ippSetFlushToZero` and `ippSetDenormAreZeros` to enable these modes. Note that this setting takes effect only when computing is done with the Intel® Streaming SIMD Extensions (Intel® SSE) and Intel Streaming SIMD Extensions 2 (Intel SSE2) instructions.

Table 7-1 illustrates how denormal data may affect performance and it 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 250 times.

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. At the same time, you should be aware that memory allocation inside of the function may have a negative impact on performance in some situations, such as in the case of cache

misses. To avoid or minimize memory allocation and keep your data in warm cache, some functions, for example, Fourier transform functions, can use or reuse memory given as a parameter to the function.

If you have to call a function, for example, an FFT function, many times, the reuse of an external buffer results in better performance. A common example of this kind of processing is to perform filtering using FFT, or 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;


int phase = 0;
/// prepare source data for two FFTs


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 FFT

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 very fast on 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 FFT of the filter coefficients needs to be done only once. 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 main operations in this kind of filtering are presented below:

```
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 frequency domain is convolution
ippsMulPack_32f_I( HH, XX, fftlen );

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

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

Another way to significantly improve performance is by using FFT and multiplication for processing large size data. Note that the zeros in the example above could be pointers to the external memory, which is another way to increase performance. Note that the Intel IPP signal processing FIR filter is implemented using FFT and you do not need to create a special implementation of the FIR functions.

Running Intel IPP Performance Test Tool

The Intel IPP Performance Test Tool is available for Windows* operating systems based on Intel® Pentium® processors and Intel® Itanium® processors. It is a fully-functioned timing system designed to do performance testing for Intel IPP functions on the same hardware platforms that are valid for the related Intel IPP libraries. It contains command line programs for testing the performance of each Intel IPP function in various ways.

You can use comand line options to control the course of tests and generate the results in a desirable format. The results are saved in a `.csv` file. The course of timing is displayed on the console and can be saved in a `.txt` file. You can create a list of functions to be tested and set required parameters with which the function should be called during the performance test. The list of functions to be tested and their parameters can either be defined in the `.ini` file, or entered directly from the console.

In the enumeration mode, the Intel IPP performance test tool creates a list of the timed functions on the console and in the `.txt` or `.csv` files.

Additionally, this performance test tool provides all performance test data in the `.csv` format. It contains data covering all domains and CPU types supported in Intel IPP. For example, you can read that reference data in sub-directory `tools/perfsys/data`.

Once the Intel IPP package is installed, you can find the performance test files located in the `ia32/tools/perfsys` directory. For example, `ps_ipps` is a tool to measure performance of the Intel IPP signal processing functions. Similarly, there are the appropriate executable files for each Intel IPP functional domain.

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 can be divided into six groups by their functionality. You can enter options in an 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 Appendix A, "Performance Test Tool Command Line Options".

Examples of Using Performance Test Tool Command Lines

The following examples illustrate how you can use common command lines for the performance test tool to generate IPP function performance data.

Example 1. Running in the standard mode:

```
ps_ippch –B –r
```

This command causes all Intel IPP string functions to be tested by the default timing method on standard data (`-B` option). The results will be generated in file `ps_ippch.csv` (`-r` option).

Example 2. Testing selected functions:

`ps_ipps -fFIRLMS_32f -r firlms.csv`

This command tests the FIR filter function `FIRLMS_32f` (`-f` option), and generates a `.csv` file named `firlms.csv` (`-r` option).

Example 3. Retrieving function lists:

`ps_ippvc -e –o vc_list.txt`

This comand causes the output file `vc_list.txt` (`-o` option) to list all Intel IPP video coding functions (`-e` option).

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

This comand causes the list of functions with names containing `H264` (`-f` option) that can be tested (`-e` option) to be displayed on the console and stored in file `H264.csv` (`-r` option).

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

`ps_ipps –B –I`

This comand causes the `.ini` file `ps_ipps.ini` to be created 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 comand causes the second run to test all functions usung the timing procedure and all function parameters values specified in the `ps_ipps.ini` file (`-i` option) and genarates the output file `ps_ipps.csv` (`-r` option).

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

Using Intel® IPP with Programming Languages

8

This chapter describes how to use Intel IPP with different programming languages in the Windows*OS development environments, and gives information on relevant samples.

Language Support

In addition to the C programming language, Intel IPP functions are compatible with the following languages (download the samples from http://www.intel.com/software/products/ipp/samples.htm):

Table 8-1 Language support

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

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 will help improve the overall performance.

Performance Test Tool Command Line Options

A

Table A-1 gives brief descriptions of possible command line options for the performance test tool (PTT).

Table A-1 Performance Test Tool Command Line Options

Groups	Options	Descriptions
1. Adjusting Console Input	-A	Ask parameters before every test from console
	-B	Batch mode
	-r[<file-name>]	Create .csv file and write PS results
	-R[<file-name>]	Add test results to .csv file
	-H[ONLY]	Add 'Interest' column to table file [and run only hot tests]
2. Managing Output	-o[<file-name>]	Create .txt file and write console output
	-O[<file-name>]	Add console output to .txt file
	-L<ERR|WARN|PARM|INFO|TRACE>	Set detail level of the console output
	-u[<file-name>]	Create .csv file and write summary table ('_sum' is added to default title name)
	-U[<file-name>]	Add summary table to .csv file ('_sum' is added to default title name)
	-e	Enumerate tests and exit
	-g[<file-name>]	Signal file is created 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 of functions with pattern in name, case sensitive
	-f-<not-pattern>	Do not test functions with pattern in name, case sensitive

Table A-1 Performance Test Tool Command Line Options (continued)

Groups	Options	Descriptions
	-f+<and-pattern>	Run only tests of functions with pattern in name, case sensitive
	-f=<eq-pattern>	Run tests of functions with this full name, case sensitive
	-F<func-name>	Start testing from function with this full name, case sensitive
4. Operation with .ini Files	-i[<file-name>]	Read PTT parameters from .ini file
	-I[<file-name>]	Write PTT parameters to .ini file and exit
	-P	Read tested function names from .ini file
5. Adjust default directories and file names for input & 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 priorit Y (normal is default)
8. Setting environment	-N<num-threads>	Call `ippSetNumThreads`(<num-treads>)
9. Getting help	-h	Type short help and exit
	-hh	Type extended help and exit
	-h<option>	Type extended help for the specified option and exit

Intel® IPP Samples

B

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 the sample applications.

Types of Intel IPP Sample Code

There are three types of Intel IPP sample code available for developers to learn how to use the Intel Integrated Performance Primitives. Each type is designed to demonstrate how to build software with the Intel IPP functions. All types are listed in Table B-1.

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.
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`).
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 Manual* (`.pdf`) as part of the function descriptions.

NOTE. Intel IPP samples are intended only to demonstrate how to use the APIs and how to build applications in different development environments.

Source Files of the Intel IPP Samples

Table B-2 presents the list of files with source code for the Intel IPP samples. All these samples are created for Windows* OS, but they can be easily adapted for Linux* OS.

Table B-2 Source Files of the Intel IPP Sample Code

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: 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

Table B-2 Source Files of the Intel IPP Sample Code (continued)

Category	Summary	Description and Links
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 • Performance comparison, vector vs. scalar: perform.cpp • Performance comparison, buffered vs. unbuffered: perform2.cpp

Table B-2 Source Files of the Intel IPP Sample Code (continued)

Category	Summary	Description and Links
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 samples from http://www.intel.com/software/products/ipp/samples.htm.

These samples are updated in each version of Intel IPP. It is strongly recommended that you upgrade the Intel IPP Samples when a new version of Intel IPP is available.

System Requirements

Refer to the *readme.htm* document in the root directory of each sample to learn the system requirements for the specific 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

Software requirements:

- Intel® IPP for the Linux* OS, version 6.1
- Red Hat Enterprise Linux* operating system version 3.0 or higher
- Intel® C++ Compiler for Linux* OS: versions 11.1, 11.0 or 10.1 ; GNU C/C++ Compiler 3.2 or higher
- Qt* library runtime and development environment

•

Building Source Code

The building procedure is described in the *readme.htm* document for 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: `/opt/Intel/IPP/6.1.x.xxx/ia32/.`

To build the sample, change your current folder to the root sample folder and run shell script `build32.sh` [`option`].

By default, the script searches the compilers step by step according to the table below (assuming that compiler is installed in the default directory). If you wish to use a specific version of the Intel C/C++ compiler or the GCC software, set an option for the script from the table below.

Table B-3 Options for Script

Compiler	Option
Intel C++ Compiler 11.1 for Linux OS	`icc111`
Intel C++ Compiler 11.0 for Linux OS	`icc110`
Intel C++ Compiler 10.1 for Linux OS	`icc101`
GCC 4.x.x	`gcc4`
GCC 3.4.x	`gcc3`

After the successful build, the result file or files are placed in the corresponding sample directory: `<install_dir>/ipp-samples/<sample-name>/bin/linux32_<compiler>`,

where `compiler = icc111|icc110|icc101|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.

Refer to the *readme.htm* document in the directory of the specific sample for detailed instructions on how to run the application, the list of command line options, or menu commands.

Known Limitations

The applications created with the Intel IPP Samples are intended to demonstrate how to use the Intel IPP functions and help developers to create their own software. These sample applications have some limitations that are described in the section "Known Limitations" in the *readme.htm* document for each sample.

Index

P

R

S

T

U

V