



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

User's Guide

September 2008

Document Number: 320271-001US

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

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The software described in this document may contain software defects which may cause the product to deviate from published specifications. Current characterized software defects are available on request.

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

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

Developers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Improper use of reserved or undefined features or instructions may cause unpredictable behavior or failure in developer's software code when running on an Intel processor. Intel reserves these features or instructions for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from their unauthorized use.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Inside, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDC-harm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright © 2008, Intel Corporation. All rights reserved.

Contents

Chapter 1 Overview

Technical Support	1-2
About This Document	1-2
Purpose	1-2
Audience	1-2
Document Organization	1-3
Notational Conventions.....	1-4

Chapter 2 Getting Started with Intel® IPP

Checking Your Installation.....	2-1
Obtaining Version Information	2-2
Building Your Application.....	2-2
Setting Environment Variables	2-2
Including Header Files	2-2
Calling IPP Functions	2-3
Before You Begin Using Intel IPP.....	2-4

Chapter 3 Intel® IPP Structure

High-level Directory Structure	3-1
Supplied Libraries	3-2
Using Intel IPP Shared Object Libraries (SO)	3-2
Using Intel IPP Static Libraries.....	3-3
Contents of the Documentation Directory	3-4

Chapter 4 Configuring Your Development Environment

Configuring Eclipse CDT to Link with Intel IPP	4-1
--	-----

	Configuring Eclipse CDT 4.0	4-1
	Configuring Eclipse CDT 3.x	4-2
Chapter 5	Linking Your Application with Intel® IPP	
	Dispatching	5-1
	Detecting Processor Type.....	5-2
	Selecting Between Linking Methods.....	5-3
	Dynamic Linking.....	5-4
	Static Linking (with Dispatching)	5-4
	Static Linking (without Dispatching)	5-6
	Building Custom SO	5-8
	Summary of Intel IPP Linkage Method Comparison.....	5-9
	Selecting Intel IPP Libraries Needed by Your Application	5-9
	Dynamic Linkage.....	5-11
	Static Linkage with Dispatching	5-11
	Library Dependencies by Domain (Static Linkage Only).....	5-12
	Linking Examples	5-13
Chapter 6	Supporting Multithreaded Applications	
	Intel IPP Threading and OpenMP* Support.....	6-1
	Setting Number of Threads	6-1
	Using Shared L2 Cache	6-2
	Nested Parallelization	6-2
	Disabling Multithreading	6-2
Chapter 7	Managing Performance and Memory	
	Memory Alignment	7-1
	Thresholding Data	7-4
	Reusing Buffers.....	7-4
	Using FFT	7-5
	Running Intel IPP Performance Test Tool	7-6
	Examples of Using Performance Test Tool Command Lines.....	7-7
Chapter 8	Using Intel® IPP with Programming Languages	
	Language Support	8-1
	Using Intel IPP in Java* Applications	8-1

Appendix A Performance Test Tool Command Line Options

Appendix B Intel® IPP Samples

Types of Intel IPP Sample Code B-1
Source Files of the Intel IPP Samples B-2
Using Intel IPP Samples B-4
 System Requirements B-4
 Building Source Code B-5
 Running the Software B-5
Known Limitations B-6

Index

Overview

1

Intel® Integrated Performance Primitives (Intel® IPP) is a software library which provides a broad range of functionality including general signal, image, speech, graphics, data compression, cryptography, text strings and audio processing, vector manipulation and matrix math, as well as 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*, and G.723, plus computer vision.

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 while 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 are 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) platforms (Intel IPP for the Linux* OS on Intel® 64 Architecture)
- users who develop on Intel® Itanium® 2 processor family platforms (Intel IPP 6.0 for the Linux* OS on IA-64 architecture)
- users who develop on Intel® Atom™ processor platforms (Intel IPP 6.0 for the Linux* OS on low power Intel® Architecture)

Technical Support

Intel provides a support web site which contains a rich repository of self-help information, including getting started tips, known product issues, product errata, license information, user forums, and more. Visit the Intel IPP support web site at <http://www.intel.com/software/products/support/ipp>.

About This Document

This document, *Intel® IPP for the Linux* OS on the IA-32 Architecture User's Guide*, focuses on the *usage* information needed to make the most of Intel IPP routines using the Linux applications running on the IA-32 architecture. This usage of Intel IPP has its particular features. The User's Guide describes these features along with those that do not depend upon a particular architecture.

After installation, you may 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 perform after the 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 linking method that suits you best, how to link your application to the library and provides simple usage scenarios.
- Explains how to thread your application using IPP software.
- Describes how to code, compile, and run your application with Intel IPP.
- Gives information 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 explains how to run the samples.

Audience

This guide is intended for Linux programmers whose software development experience may vary from beginner to advanced.

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 library types supplied.
- Chapter 4 [Configuring Your Development Environment](#) explains how to configure Intel IPP and your development environment for the use with the library.
- Chapter 5 [Linking Your Application with Intel® IPP](#) compares linking methods; helps selecting a linking method for particular purpose, describes the general link line syntax to be used for linking with the Intel IPP libraries; discusses how to build custom dynamic libraries.
- Chapter 6 [Supporting Multithreaded Applications](#) helps you set number of threads in multithreaded applications, get information on number of threads, disable multithreading.
- Chapter 7 [Managing Performance and Memory](#) discusses ways of improving Intel IPP performance; gives information how to accomplish Intel IPP functions performance tests by using 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 the Intel IPP sample code available for developers to learn how to use the Intel IPP, presents the source code example files by categories with links to view the sample code, and explains how to run the samples using the example of simple media player application.

The document also includes an [Index](#).

Notational Conventions

The document employs the following font conventions and symbols:

Table 1-1 Notational conventions

<i>Italic</i>	<i>Italic</i> is used for emphasis and also indicates document names in body text, for example, see <i>Intel IPP Reference Manual</i>
Monospace lowercase	Indicates filenames, directory names and pathnames: <code>tools/env/ippvars32.csh</code>
Monospace lowercase mixed with uppercase	Indicates code, commands and command-line options: <code>export LIB=\$IPPROOT/lib:\$LIB</code>
UPPERCASE MONOSPACE	Indicates system variables, for example, <code>LD_LIBRARY_PATH</code>
<i>Monospace italic</i>	Indicates a parameter in discussions: function parameters, for example, <code>lda</code> ; makefile parameters, for example, <code>functions_list</code> ; and so on. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value: <code><ipp directory></code> .
[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 giving basic information you need to know and describing the necessary steps you need to perform after the installation of the product.

Checking Your Installation

Once you complete the installation of the Intel IPP, it is useful to perform a basic verification task that confirms proper installation and configuration of the library.

1. Check that the directory you chose for installation has been created: `<installation path>/intel/ipp/6.0.x.xxx/ia32`. The default installation directory is `opt/intel/ipp/6.0.x.xxx/ia32`.
2. Check that the 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 the processor-specific libraries are on the path.

If you receive 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 from 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 get started using Intel IPP, sorting out some important basic concepts will help you get off to a good start.

[Table 2-1](#) summarizes some important things to consider before you start using Intel IPP.

Table 2-1 **What you need to know before you get started**

Function domains	<p>Identify the Intel IPP function domain that meets your needs.</p> <p>Reason: The knowing of the function domain you intend to use will narrow the search in the Reference Manuals for specific routines you need.</p> <p>Besides, you may easily find a sample you would like to run from http://www.intel.com/software/products/ipp/samples.htm.</p> <p>Refer to Table 5-8 to understand what function domains are and what libraries are needed, and to Table 5-9 to understand what kind of cross-domain dependency is introduced.</p>
Linking method	<p>Decide what linking method is appropriate for linking.</p> <p>Reason: If you choose a linking method that suits you most of all, 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, like how to create a custom dynamic library, see Linking Your Application with Intel® IPP</p>
Threading model	<p>Select among the following options how you are going to thread your application:</p> <ul style="list-style-type: none">• Your application is already threaded• You may want to use the Intel® threading capability, that is, Compatibility OpenMP* run-time library (<code>libiomp</code>), or a threading capability provided by a third-party compiler.• You do not want to thread your application. <p>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.</p>

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 a high-level directory structure of Intel IPP after installation.

Table 3-1 High-level directory structure

Directory	File types
<code><ipp directory></code>	Main directory (by default: <code>/opt/intel/ipp/6.0.x.xxx/ia32</code>)
<code><ipp directory>/ippEULA.txt</code>	End User License Agreement for Intel IPP
<code><ipp directory>/doc</code>	Intel IPP documentation files
<code><ipp directory>/include</code>	Intel IPP header files
<code><ipp directory>/lib</code>	Intel IPP static libraries
<code><ipp directory>/sharedlib</code>	Intel IPP shared object libraries
<code><ipp directory>/tools</code>	Intel IPP Performance Test tool, linkage tools, and tool to set environment variables

Supplied Libraries

The following [Table 3-2](#) lists types of libraries of Intel IPP and examples of 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.6.0, libippst7.so.6.0
	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	Dispatchers for the merged libraries:		
	libraries with position independent code (PIC)	ia32/lib	libippsemmerged.a
	non-PIC)* libraries	ia32/lib/nonpic	libippsemmerged.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/6.0.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/6.0.x.xxx/ia32/sharedlib:$LD_LIBRARY_PATH
```

or for csh:

```
setenv LD_LIBRARY_PATH=
/opt/intel/ipp/6.0.x.xxx/ia32/sharedlib:${LD_LIBRARY_PATH}
```

The shared libraries `libipp*.so.6.0` (* 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 your 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.6.0`, `libipp*w7.so.6.0`, `libipp*t7.so.6.0`, `libipp*v8.so.6.0`, and `libipp*p8.so.6.0` (see [Table 5-3](#)). For example, in the `ia32/sharedlib` directory, `libippiv8.so.6.0` 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*.so`, `libipp*px.so`, `libipp*w7.so`, `libipp*t7.so`, `libipp*v8.so`, and `libipp*p8.so`.

See also [Selecting Intel IPP Libraries Needed by Your Application](#).



NOTE. It is required that the appropriate `libiomp5.so` is also included 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 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 Intel IPP Libraries Needed by Your Application](#).

Contents of the Documentation Directory

[Table 3-3](#) lists the main documents places in the `/doc` subdirectory in the Intel IPP installation directory.

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

File name	Description	Notes
<code>ipp_documentation.htm</code>	Documentation index. Lists the principal Intel IPP documents with appropriate links thereto	
<code>ReleaseNotes.htm</code>	General overview of the product and information about this release.	This files can be viewed prior to the product installation
<code>README.txt</code>	Initial User Information	
<code>INSTALL.htm</code>	Installation guide	
<code>ThreadedFunctionsList.txt</code>	List of all Intel IPP functions threaded with the OpenMP*	
<code>userguide_lin_ia32.pdf</code>	Intel® Integrated Performance Primitives User's Guide, this document	
Intel IPP Reference Manual (in four volumes):		
<code>ippsman.pdf</code>	<i>Signal Processing</i> (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.	
<code>ippiman.pdf</code>	<i>Image and Video Processing</i> (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	<i>Small Matrices, Realistic Rendering</i> (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	<i>Cryptography</i> (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/6.0.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/6.0.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 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/6.0.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 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/6.0.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/6.0.x.xxx/lib/ia32`.
4. Specify names of the Intel IPP libraries to link with your application. 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® 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 unit (CPU). 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, `libippiv8.so.6.0` 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 (SSE2)
t7	Optimized for processors with Intel® Streaming SIMD Extensions 3 (SSE3)
v8	Optimized for processors with Intel® Supplemental Streaming SIMD Extensions 3 (SSSE3)
p8	Optimized for processors with Intel® Streaming SIMD Extensions 4.1 (SSE4.1)

Detecting Processor Type

To detect the processor type used in your computer system, use function `ippGetCpuType` 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 5-2](#) shows possible values of `ippGetCpuType` and their meaning.

Table 5-2 Detecting processor type. Returned values and their meaning

Returned Variable Value	Processor Type
<code>ippCpuPP</code>	Intel® Pentium® processor
<code>ippCpuPMX</code>	Pentium® processor with MMX™ technology
<code>ippCpuPPR</code>	Pentium® Pro processor
<code>ippCpuPII</code>	Pentium® II processor
<code>ippCpuPIII</code>	Pentium® III processor and Pentium® III Xeon® processor
<code>ippCpuP4</code>	Pentium® 4 processor and Intel® Xeon® processor
<code>ippCpuP4HT</code>	Pentium® 4 processor with Hyper-Threading Technology
<code>ippCpuP4HT2</code>	Pentium® Processor with Intel® Streaming SIMD Extensions 3
<code>ippCpuCentrino</code>	Intel® Centrino™ mobile Technology
<code>ippCpuCoreSolo</code>	Intel® Core™ Solo processor
<code>ippCpuCoreDuo</code>	Intel® Core™ Duo processor
<code>ippCpuITP</code>	Intel® Itanium® processor
<code>ippCpuITP2</code>	Intel® Itanium® 2 processor
<code>ippCpuEM64T</code>	Intel® 64 Instruction Set Architecture (ISA)
<code>ippCpuC2D</code>	Intel® Core™ 2 Duo Processor
<code>ippCpuC2Q</code>	Intel® Core™ 2 Quad processor
<code>ippCpuBonnell</code>	Intel® Atom™ processor
<code>ippCpuPenryn</code>	Intel® Core™ 2 processor with Intel® Streaming SIMD Extensions 4.1 instruction set
<code>ippCpuSSE</code>	Processor supports Intel® Streaming SIMD Extensions instruction set
<code>ippCpuSSE2</code>	Processor with Intel® Streaming SIMD Extensions 2 instruction set

Table 5-2 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 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
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 always packaged together?

Dynamic Linking

The dynamic linking is the simplest 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 5-3 Summary of Dynamic Linking Features

Benefits	Considerations
<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • 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/plugin-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, do the following:

1. Add `ipp.h` that 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 `libippmerged.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. Without this, (most often issues) performance of your application is not the best because a “px” version of the IPP functions will be called. To illustrate the performance difference, you can use the following [Example 5-1](#) 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;
}

cmdlinetest>cl /nologo /Fet2.exe -I "C:\Program
Files\Intel\IPP\6.0.x.xxx\ia32\include\" t2.cpp /link /LIBPATH:"C:\Program
Files\Intel\IPP\6.0.x.xxx\ia32\lib\" ippcore1.lib ippmergedem64t.lib
ippmerged.lib
t2.cpp

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

[Table 5-4](#) summarizes pros and cons of the static linking via the emerged libraries you should consider before using this particular linking method.

Table 5-4 Summary of Features of the Static Linking (with Dispatching)

Benefits	Considerations
<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • 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 `ippStaticInitCpu()` 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 can prefer 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 Intel IPP samples directory: `ipp-samples/advanced-usage/linkage/mergelib` at <http://www.intel.com/software/products/ipp/samples.htm>.

It is most appropriate 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 is for embedded applications where the application is bundled with only one type of processor.

[Table 5-5](#) summarizes basic features of this method of linking.

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

Benefits	Considerations
<ul style="list-style-type: none"> • 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 Web applet or plug-in requiring very small file download and support for only one processor type • Self-contained application executable that does not require Intel IPP run-time SOs to run • Smallest footprint for application package • Smallest installation package 	<ul style="list-style-type: none"> • 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 alternative to 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 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 5-6](#) summarizes features of the custom SOs.

Table 5-6 Custom SO Features

Benefits	Considerations
<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • 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 Intel IPP samples directory: `ipp-samples/advanced-usage/linkage/customso` at <http://www.intel.com/software/products/ipp/samples.htm>.

Summary of Intel IPP Linkage Method Comparison

[Table 5-7](#) gives quick comparison of the IPP linkage methods.

Table 5-7 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 Intel IPP Libraries Needed by Your Application

The libraries used for each linkage method with regard to particular functional domain are listed in [Table 5-8](#).

Table 5-8 Libraries Used for Each Linkage Method

Domain Description	Header Files	Dynamic Linking	Static Linking with Dispatching & Custom Dynamic Linking	Static Linking without Dispatching
Audio Coding	ippac.h	libippac.so	libippacmerged.a	libippacmerged.a libippacmerged_t.a
Color Conversion	ippcc.h	libippcc.so	libippccmerged.a	libippccmerged.a libippccmerged_t.a

Table 5-8 Libraries Used for Each Linkage Method (continued)

Domain Description	Header Files	Dynamic Linking	Static Linking with Dispatching & Custom Dynamic Linking	Static Linking without Dispatching
String Processing	ippch.h	libippch.so	libippchemerged.a	libippchmerged.a libippchmerged_t.a
Cryptography	ippcp.h	libippcp.so	libippcpmerged.a	libippcpmerged.a libippcpmerged_t.a
Computer Vision	ippcv.h	libippcv.so	libippcvmerged.a	libippcvmerged.a libippcvmerged_t.a
Data Compression	ippdc.h	libippdc.so	libippdcmerged.a	libippdcmerged.a libippdcmerged_t.a
Data Integrity	ippdi.h	libippdi.so	libippdipmerged.a	libippdimerged.a libippdimerged_t.a
Generated Functions	ipps.h	libippgen.so	libippgenmerged.a	libippgenmerged.a libippgenmerged_t.a
Image Processing	ippi.h	libippi.so	libippimerged.a	libippimerged.a libippimerged_t.a
Image Compression	ippj.h	libippj.so	libippjmerged.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	libippmmerged.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	libippscmerged.a	libippscmerged.a libippscmerged_t.a
Speech Recognition	ippsr.h	libippsr.so	libippsremerged.a	libippsrmerged.a libippsrmerged_t.a
Video Coding	ippvc.h	libippvc.so	libippvcmerged.a	libippvcmerged.a libippvcmerged_t.a

Table 5-8 Libraries Used for Each Linkage Method (continued)

Domain Description	Header Files	Dynamic Linking	Static Linking with Dispatching & Custom Dynamic Linking	
			Static Linking with Dispatching	Static Linking without Dispatching
Vector Math	ippvm.h	libippvm.so	libippvmmerged.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. All corresponding domain libraries used in your applications plus the libraries `libipps.so`, `libippcore.so` and `libiomp.so` must be linked to.

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 Intel IPP libraries:

```
libippi.so
libippcv.so
libippm.so
libippcore.so
libiomp5.so
```

Static Linkage with Dispatching

To use the static linking libraries, you need link to `lib*merged.a`, `lib*emerged.a`, `libsemerged.a`, `libsmmerged.a`, and `libcore.a`.

If you want to use the Intel IPP functions threaded with the OpenMP*, you need link to `lib*merged.a`, `lib*merged_t.a`, `libsemerged.a`, `libsmmerged_t.a`, `libcore_t.a`, and additionally `libiomp.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, 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:

```
libcvmerged.a and libcvmerged_t.a
libmemerged.a and libmmerged_t.a
libiemerged.a and libimerged_t.a
libsemerged.a and libserged_t.a
libcore_t.a
libiomp5.a
```

Library Dependencies by Domain (Static Linkage Only)

[Table 5-9](#) lists library dependencies by domain. When you link to the certain library (for example, data compression domain) you must link to the libraries on which it depends on (in our example, 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.

lists library dependencies by domain.

Table 5-9 Library Dependencies by Domain

Domain	Library	Dependent on
Audio Coding	<code>ippac</code>	<code>ippdc</code> , <code>ipps</code> , <code>ippcore</code>
Color Conversion	<code>ippcc</code>	<code>ippi</code> , <code>ipps</code> , <code>ippcore</code>
Cryptography	<code>ippcp</code>	<code>ippcore</code>
Computer Vision	<code>ippcv</code>	<code>ippi</code> , <code>ipps</code> , <code>ippcore</code>
Data Compression	<code>ippdc</code>	<code>ipps</code> , <code>ippcore</code>
Data Integrity	<code>ippdi</code>	<code>ippcore</code>
Generated Functions	<code>ippgen</code>	<code>ipps</code> , <code>ippcore</code>
Image Processing	<code>ippi</code>	<code>ipps</code> , <code>ippcore</code>
Image Compression	<code>ippj</code>	<code>ippi</code> , <code>ipps</code> , <code>ippcore</code>
Small Matrix Operations	<code>ippm</code>	<code>ippi</code> , <code>ipps</code> , <code>ippcore</code>

Table 5-9 Library Dependencies by Domain

Domain	Library	Dependent on
Realistic Rendering and 3D Data Processing	ipprr	ippi, ippss, ippcore
Signal Processing	ippss	ippcore
Speech Coding	ippssc	ippss, ippcore
Speech Recognition	ippssr	ippss, ippcore
String Processing	ippch	ippss, ippcore
Video Coding	ippvc	ippi, ippss, 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 gives significant performance gain 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 desired number of threads (1,...). If internal parallelization is not desired, call `ippSetNumThreads(1)`.

Using Shared L2 Cache

Some functions in signal processing domain are threaded on 2 threads intended for the Intel® Core™2 processor family, and exploit advantage of merged L2 cache. These functions (single and double precision FFT, Div, Sqrt, etc.) 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 the OpenMP uses the threaded Intel IPP function, this function will operate in the single thread because the nested parallelization is disabled by default in the OpenMP.

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

Disabling Multithreading

To disable multi-threading link your application with IPP non-threaded static libraries, or build the custom SO using the non-threaded static libraries.

Managing Performance and Memory

7

This chapter describes ways you can follow to get the most out of the Intel® IPP software: aligning memory, thresholding denormal data, reusing buffers, using 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 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 be only 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 occur, 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 occur 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 a special computation modes - flush-to-zero (FTZ) and the denormals-are-zero (DAZ). You can use functions `ippsSetFlushToZero` and `ippsSetDenormAreZeros` to enable these modes. Note that this setting takes effect only when computing is done with the Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2) instructions.

[Table 7-1](#) illustrates how denormal data may affect performance and 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 well aware that memory allocation inside of the function may have negative impact on performance in some situations 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 for example, an FFT function many times, the reuse of an external buffer results in better performance. A simple examples of this kind of processing are to perform filtering using FFT, or computing FFT as two FFT 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

Using 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 transform 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 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 FFT forward transform 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 could be very significant.

If the applied filter is the same for several processing iterations, then the FFT transform of the filter coefficients need to be done only once. The twiddle tables and the bit reverse tables are created in the initialization function for the forward and for inverse transforms at the same time. The main operations in this kind of filtering are presented below:

```
    ippFFTInitAlloc_R_32f( &pFFTSpec, fftord, IPP_FFT_DIV_INV_BY_N,
        ippAlgHintNone );

    // perform forward FFT to put source data xx to frequency domain
    ippFFTFwd_RTToPack_32f( xx, XX, pFFTSpec, 0 );

    // perform forward FFT to put filter coefficients hh to frequency domain
    ippFFTFwd_RTToPack_32f( hh, HH, pFFTSpec, 0 );

    // point-wise multiplication in frequency domain is convolution
    ippMulPack_32f_I( HH, XX, fftlen );

    // perform inverse FFT to get result yy in time domain
    ippFFTInv_PackToR_32f( XX, yy, pFFTSpec, 0 );

    // free FFT tables
    ippFFTFree_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 for the Linux* OS based on Intel® Pentium® processors is a very functional timing system designed to accomplish Intel IPP functions performance tests on the same hardware platforms as the related Intel IPP libraries run. It contains command line programs for testing the performance of each Intel IPP function in various ways.

You can control the course of tests and generate the results in the desirable format by using command line options. 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_ipp` 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> [switch_1] [switch_2] ... [switch_n]
```

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

```
ps_ipp -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. Some options (like `-v`, `-V`, `-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 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 performance tool common command lines to generate IPP function performance data.

Example 1. Running in the standard mode:

```
ps_ippch -B -v
```

All Intel IPP string functions are tested by the default timing method on standard data (`-B` option). The results are generated in file `ps_ippch.csv` (`-v` option).

Example 2. Testing selected functions:

```
ps_ippf -fFIRLMS_32f -v firllms.csv
```

Tests the FIR filter function `FIRLMS_32f` (-f option), and generates a `.csv` file named `firllms.csv` (-v option).

Example 3. Retrieving function lists:

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

The output file `vc_list.txt` (-o option) lists all Intel IPP video coding functions (-e option).

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

The list of functions with names containing `H264` (-f option) that may be tested (-e option) is displayed on the console and stored in file `H264.csv` (-v option).

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

```
ps_ippf -B -I
```

The `.ini` file `ps_ippf.ini` is 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 -v
```

The second run tests all functions, reading timing procedure and all function parameters values from the `ps_ippf.ini` file (-i option) and the output file `ps_ippi.csv` (-v option) is generated.

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 the usage of Intel IPP with different programming languages in the Windows*OS development environments, and gives information on the 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 demonstrates using the Intel IPP image processing functions in a Java wrapper class.

Using Intel IPP in Java* Applications

You may 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



[Table A-1](#) gives brief descriptions of possible performance test tool command line options.

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
2. Managing Output	-H[ONLY]	Add 'Interest' column to table file [and run only hot tests]
	-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)	

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

Groups	Options	Descriptions
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
	-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 PS parameters from .ini file
	-I[<file-name >]	Write PS 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 input files that is .ini and test data files
	-l<dir-name >	Set default directory for output files
6. Direct Data Input	-d<name >= <value >	Set PS parameter value
7. Process priority	-Y<HIGH/NORMAL >	Set high or normal process priority (normal is default)
8. Setting environment	-T<cpu-name >	Call <code>ippInitStaticCpu(ippCpu<cpu-name >)</code>
	-N<num-threads >	Call <code>ippSetNumThreads(<num-treads >)</code>

Intel® IPP Samples

B

This appendix describes 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 the 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	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 <i>Intel IPP Manual</i> (.pdf) as part of the function descriptions.



NOTE. Intel IPP samples are intended only to demonstrate how to use the APIs and 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	<ul style="list-style-type: none"> Performance measurement GetClocks.cpp Copying data: Copy.cpp Optimizing table-based Functions: LUT.cpp
Digital Filtering	Fundamentals of signal processing	<ul style="list-style-type: none"> Executing the DFT: DFT.cpp Filtering with FFT: FFTFilter.cpp Time-domain filtering: FIR.cpp
Audio Processing	Audio signal generation and manipulation	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> <li data-bbox="911 270 1367 366">• Wrapper for resizing an image: Resize.h Resize.cpp ResizeTest.cpp <li data-bbox="911 366 1367 461">• Wrapper for rotating an image: Rotate.h Rotate.cpp RotateTest.cpp <li data-bbox="911 461 1367 557">• Wrapper for doing an affine transform on an image: Affine.h Affine.cpp AffineTest.cpp
Graphics and Physics	Vector and small matrix arithmetic functions	<ul style="list-style-type: none"> <li data-bbox="911 557 1367 713">• ObjectViewer application: ObjectViewerDoc.cpp ObjectViewerDoc.h ObjectViewerView.cpp ObjectViewerView.h <ul style="list-style-type: none"> <li data-bbox="1073 713 1367 835">Transforming vertices and normals: <code>CTestView::OnMutateModel</code> <li data-bbox="1073 835 1367 956">Projecting an object onto a plane: <code>CTestView::OnProjectPlane</code> <li data-bbox="1073 956 1367 1060">Drawing a triangle under the cursor: <code>CTestView::Draw</code> <li data-bbox="911 1060 1367 1121">• Performance comparison, vector vs. scalar: perform.cpp <li data-bbox="911 1121 1367 1213">• 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	<ul style="list-style-type: none">• 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 to upgrade Intel IPP Sample 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 6.0
- Red Hat Enterprise Linux* operating system version 3.0 or higher
- Intel® C++ Compiler for Linux* OS: versions 10.1, 10.0 or 9.1; 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/6.0.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 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 Intel C/C++ compilers or GCC software, set an option for script according to the table below.

Table B-3 Options for Batch Files

Compiler	Script Option
Intel C++ Compiler 10.1 for Linux OS	<code>icc101</code>
Intel C++ Compiler 10.0 for Linux OS	<code>icc10</code>
Intel C++ Compiler 9.1 for Linux OS	<code>icc91</code>
GCC 4.x.x	<code>gcc4</code>
GCC 3.4.x	<code>gcc3</code>

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 = icc101|icc10|icc91|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, list of command line options or menu commands.

Known Limitations

The applications created with the Intel IPP Samples are examples 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

B

building application, 2-2
building samples, B-5

C

calling functions, 2-3
checking your installation, 2-1
configuring Eclipse, 4-1
controlling number of threads, 6-1

D

detecting processor type, 5-2
Disabling Multithreading, 6-2
dispatching, 5-1
document

- audience, 1-2
- filename, 3-4
- location, 3-1
- organisation, 1-3
- purpose, 1-2

H

header files, 2-2

I

Intel® IPP, 1-1

J

java applications, 8-1

L

language support, 8-1
library dependencies by domain, 5-12
linking

- custom dynamic, 5-8
- dynamic, 5-4
- static, with dispatching, 5-4
- static, without dispatching, 5-6

linking examples, 5-13
linking models, 5-3
linking models comparison, 5-9

M

managing performance, 7-1

- memory alignment, 7-1
- reusing buffers, 7-4
- thresholding data, 7-4
- using FFT, 7-5

memory alignment, 7-1

N

notational conventions, 1-4

O

OpenMP support, 6-1

P

- performance test tool, 7-6
 - command line examples, 7-7
 - command line options, A-1
- processor-specific codes, 5-1

R

- reusing buffers, 7-4
- running samples, B-4
 - known limitations, B-6

S

- Sample, B-1
- samples
 - types, B-1
- selecting
 - libraries, 5-9
 - linking models, 5-3
- setting environment variables, 2-2
- source code samples, B-2
- structure
 - by library types, 3-2
 - documentation directory, 3-4
 - high-level directory, 3-1
- supplied libraries, 3-2

T

- technical support, 1-2
- threading, 6-1
- thresholding data, 7-4

U

- usage, 1-2
- using
 - DLLs, 3-2
 - static libraries, 3-3
- using FFT, 7-5
- using Intel IPP
 - with Java, 8-1
 - with programming languages, 8-1

- with Visual C++ .NET, 4-2

V

- version information, 2-2