Choosing the Best Linkage Model for Your Intel® IPP-based Application for Intel® Pentium and Itanium® architectures

Version 1.0

July 2003

Table of Contents

1. Introduction.....4
2. Organization of Intel IPP Functions.....5
3. Things to Consider When Choosing a Linkage Model.....6
4. Intel IPP Linkage Models and Calling Conventions.....7
   4.1. Intel IPP Dynamic Linkage 8
   4.2. Custom Intel IPP Dynamic Linkage 10
   4.3. Intel IPP Static Linkage with Dispatching (E-Merged) 10
   4.4. Intel IPP Static Linkage Without Dispatching (Merged Static)12
5. Summary 14
Appendix. Custom Dynamic Linkage Example.....14
References 15

1. Introduction

Intel® Integrated Performance Primitives (Intel® IPP) is a software library which provides a broad range of functionality including general signal, image, speech, graphics and audio processing, vector manipulation, matrix math, string processing and cryptography, 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.263, JPEG, GSM-AMR and G723, 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. The library functions are optimized for Intel's latest processor architectures, with automatic run-time dispatching of the best optimizations. For more information about Intel IPP, please visit the Intel IPP website [1].

Intel IPP is optimized for the broad range of Intel® microprocessors: Intel® Pentium® 4 processor, Intel® Pentium® M processor component of Intel® Centrino™ mobile technology, the Intel® Itanium® 2 processor, Intel® Xeon™ processors, and Intel® PCA application processors based on the Intel® XScale™ microarchitecture. With a single API across the range of architectures, developers can have platform compatibility and reduced cost of development. Built-in dispatcher capability of Intel IPP chooses the best optimizations - run-time processor detection automatically dispatches processor-specific code.

To support a broad range of application use, Intel IPP supports several different linkage models for Intel® Pentium and Itanium® architectures that allow the developer to choose one that best suits their application development environment and application deployment constraints:

- Intel IPP Dynamic Linkage – The full set of Intel IPP library functions including the processor-specific optimizations and automatic run-time dispatching for multiple processor types. The DLLs supporting this linkage model are pre-packaged as the Intel IPP RTI for easy redistribution of Intel IPP run-time components with developers' applications.
- Custom Intel IPP Dynamic Linkage – Compact distribution for multiple Intel IPP-based applications, with automatic run-time dispatching for multiple processor types. This model includes the functions and processor-specific optimizations for just those functions/libraries used by the specific application.
- Intel IPP Static Linkage with Dispatching (E-Merged) – Compact, self-contained application (no requirement for Intel IPP run-time DLLs), with automatic dispatching for multiple processor types. Most appropriate for distribution of a single Intel IPP-based application where code sharing enabled by DLLs provides no benefit.
- Intel IPP Static Linkage without Dispatching – Smallest footprint, optimized only for one target processor type, suitable for kernel-mode or driver use.

This document compares and contrasts each of the linkage models available, and discusses the pros and cons for using each model. Aspects covered include 1) relative ease-of-use; 2) typical applications for the model; 3) effects on memory and disk footprint; 4) rebuild requirements for integrating Intel IPP updates; and 5) application installation requirements.

This document begins with background information on the Intel IPP naming convention, and on the organization of Intel IPP functions into libraries. This information plays an important role in understanding how to use each linkage model.

2. Organization of Intel IPP Functions

Intel IPP functions are primarily organized in the following ways (as of Intel IPP version 3.0):

1) **Data-Domain:** A broad category of Intel IPP functions that share a general data model. The data-domain determines the prefix segment of a function name, and also the location of the function's documentation within the Reference Manual(s). The three Intel IPP data-domains are:

a. **Signal Processing**, operations on (1D, one dimensional) vectors of "signal" data. Function-name prefix is "`ipps`".

b. **Image Processing**, operations on (2D) arrays of "image" data. Function-name prefix is "`ippi`".

c. **Matrix**, operations on matrices, specifically optimized for small matrices. Function-name prefix is "`ippm`".

2) **Application-Domain:** A focused category of Intel IPP functions that share a specific use. Sometimes associated with a set of closely-related industry standards, each application-domain is represented by a corresponding set of static and dynamic libraries that use the same naming convention. In other words, the application-domain determines which Intel IPP libraries to link with for any given Intel IPP function. The application-domains for version 3.0 of Intel IPP are currently:

1. `ipps` -- signal processing functions
2. `ippi` -- image processing functions
3. `ippj` -- JPEG/JPEG2000 functions
4. `ippsr` -- speech recognition functions
5. `ippsc` -- speech coding functions
6. `ippac` -- audio coding functions
7. `ippvc` -- video coding functions
8. `ippmp` -- media processing functions
9. `ippm` -- small matrix operation functions
10. `ippvm` -- vector math functions
11. `ippcv` -- computer vision functions
12. `ippcore` -- Intel IPP common and generic functions

The Application-Domain grouping (or the library name) is reflected in the chapter names in the Intel IPP reference manuals [2]. For example, the function `ippsCopy_8u` (memory copy of byte arrays) is one of the signal processing functions and is located in library `ipps20.lib` (an import library for dynamic linking, `libipps.so` for Linux*).
Note that the Application-Domain grouping is distinct from the Intel IPP Data-Domain function prefixes, which only denote the data dimension – `ipps` *for one-dimensional and ippi for two-dimensional data.*

3) **Target Processor:** The libraries are also grouped by processor-specific optimizations using the following naming convention (as of version 3.0 of Intel IPP):

`px` -- generic C optimized code

`m6` -- code optimized for Intel® Pentium® II processor

`a6`	--	code optimized for Intel® Pentium® III processor
`w7`	--	code optimized for Intel® Pentium® 4 processor
`i7`	--	code optimized for Intel® Itanium® and Itanium® 2 processors

Developers need to be mindful of the processor identifiers whenever creating Intel IPP-based applications *without* the automatic processor-specific dispatching – to determine the *name-decoration* for processor-specific versions of Intel IPP functions, and to aid in confirming correct installation of processor-specific run-time DLLs.

These processor identifiers are used as a prefix for processor-specific function names. For example, `ippsCopy_8u` has many versions, each optimized to run on a different Intel processor type. The Pentium 4 processor-specific function name for this function is `w7_ippsCopy_8u()`.

The processor identifiers are also used as a suffix for the processor-specific dynamic link library files containing functions optimized for that processor. For example, the Pentium 4 processor-specific functions for signal processing are located in `ippsw7.dll` (dynamic link library) and `ippsw7.so` (shared object).

For static linkage, all processor-specific versions of functions in the same application-domain are combined into one "merged library" file for the application-domain, with the Intel IPP function names for each processor-specific version prefixed by the processor abbreviation. This is often referred to as the "decorated name". For example, `ippsmerged`.lib (`libippsmerged.a` for Linux*) contains `px_ippsCopy_8u`, `m6_ippsCopy_8u`, `a6_ippsCopy_8`, and `w7_ippsCopy_8u`. *Note that the code optimized for Itanium processors is in a separate library due to the different 64-bit format.*

3. Things to Consider When Choosing a Linkage Model

When using a library of functions in the development of an application, the developer must decide whether the functions will be statically linked or dynamically linked with the application. The choices are relatively simple, and based on a trade-off of complexity vs. size vs. flexibility. Dynamic linkage allows multiple applications to share the same code and can reduce the overall size of the applications and their supporting dynamic libraries. In general, using the Intel IPP dynamic link libraries (DLLs on Windows*, shared objects on Linux*) is the simplest linkage model to develop with and also the simplest in terms of application distribution. By using the full set of Intel IPP DLLs for run-time support, developers need not focus on which Intel IPP functions are used in their application when determining what components to redistribute. And the run-time components come pre-packaged for redistribution with the developer's application. However, there are a number of constraints that may apply to an application, its development environment, and its target installation environment. Since each application has its own unique set of constraints, it is important to consider the specific install-time and run-time constraints and requirements of the application, as well as development resources available and the release / update / distribution plans for the application:

1. Executable and/or application installation package size. Are there limitations on how large the application executable can be? Are there limitations on how large the application installation package can be?
2. File locations at installation time. When installing the application, are there restrictions on placing files in system folders?

3. Number of co-existing Intel IPP-based applications. Does the application include multiple Intel IPP-based executables? Are there other Intel IPP-based applications that often co-exist on the users' systems?
4. Memory available at run-time. What are the memory restrictions on the users' systems at run-time?
5. Kernel-mode vs. User-mode operation of application. Is the Intel IPP-based application a device driver or similar "ring 0" software that executes in Kernel mode at least some of the time?
6. Processor types supported by application. Will various users install the application on a range of processor types, or is the application explicitly only supported on a single type of processor? Is the application part of an embedded system or similar where only one type of processor is used?
7. Development resources. 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?
8. Release / Update / Distribution plans. How often will the application be updated? Will application components be distributed independently or always packaged together?

The answers to these questions will help determine which linkage model is best for a given application.

4. Intel IPP Linkage Models and Calling Conventions

In addition to the usual choices of static vs. dynamic linkage, the processor-specific dispatching aspect of the Intel IPP library creates another dimension, yielding four distinct linkage models (two dynamic and two static). They are:

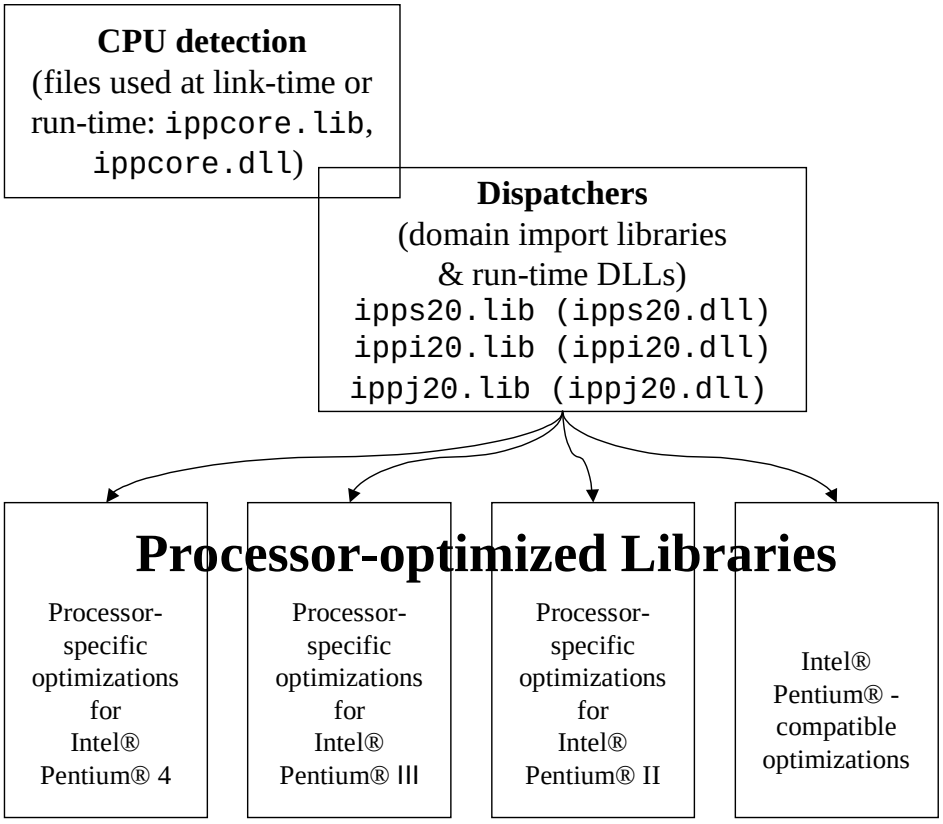
- Intel IPP Dynamic Linkage – Automatic dispatching of processor-specific optimizations, includes threaded optimizations. This model includes the full Intel IPP library, all functions, all optimized libraries. The DLLs supporting this linkage model are pre-packaged as the Intel IPP RTI for easy redistribution of Intel IPP run-time components with developers' applications.
- Custom Intel IPP Dynamic Linkage – Compact distribution for multiple Intel IPP-based applications, dispatching of processor-specific optimizations. This model includes the functions and processor-specific optimizations for just those functions/libraries used by the specific application.
- Intel IPP Static Linkage with Dispatching (E-Merged) – Compact, self-contained application (no requirement for Intel IPP run-time DLLs), with automatic dispatching for multiple processor types. Most appropriate for distribution of a single Intel IPP-based application where code sharing enabled by DLLs provides no benefit.
- Intel IPP Static Linkage without Dispatching – Smallest footprint, optimized only for one target processor type, suitable for kernel-mode or driver use.

Each of these linkage models is described in the sections below, along with the corresponding calling conventions and the procedures for implementing the model.

4.1. Intel IPP Dynamic Linkage

The simplest and probably the most commonly used linkage model is using the Intel IPP Dynamic Link libraries (or Shared Objects on Linux). The benefits of run-time code sharing among multiple Intel IPP-based applications, automatic dispatching of processor-specific optimizations, and the ability to provide updated processor-specific optimizations without relinking or redistributing the application executable out-weigh most other concerns. This is by far the best way to ensure that the end-user of the application is experiencing the great performance on his/her PC.

By detecting the CPU type at runtime during the DLL initialization, Intel IPP dispatches the processor-specific optimizations automatically. This means that the optimizations of `ippsCopy_8u` for Pentium 4 processors will be used on Pentium 4 processor-based systems, and the optimizations for Pentium III processors will be used on Pentium III processor-based systems. This mechanism is illustrated in the figure below:



The import libraries `ipps20.lib`, `ippi20.lib`, etc. are used at link time. The runtime library `ippcore.dll` features the CPU detection mechanism and `ipps20.dll`, `ippi20.dll`, etc. directs execution to the lower level processor specific optimizations for the Intel IPP functions, a process also known as dispatching. During the DLL initialization, functions in `ippcore.dll` are called to obtain the CPU identification, and then a "waterfall" procedure is carried out to determine the best *available* optimization (`w7`, a6, `m6`, or `px`). Finally, all functions in `ipps20.dll`, `ippi20.dll`, etc. are redirected to corresponding processor-specific optimized libraries. This detection and

dispatching process is automatically handled for the application.
Note that for Linux, the library names are slightly different but the same mechanism applies.

The distribution of applications based on this linkage model is further simplified by the presence of pre-packaged Intel IPP run-time libraries, which may be redistributed with Intel IPP-based applications. The Run-Time Installer, or RTI package, automatically installs a full set of Intel IPP runtime libraries in the system or specified directory. Most applications are good candidates for using this linkage model. This is the recommended linkage model for Intel IPP.

Benefits:

- ☑ Automatic run-time dispatching of processor-specific optimizations
- ☑ Enables updates with new processor optimizations without recompile / relink
- ☑ May reduce disk space requirements for applications with multiple Intel IPP-based executables
- ☑ Enables more efficient shared use of memory at run-time for multiple Intel IPP-based applications
- ☑ Simple redistribution of Intel IPP run-time libraries via RTI package
- ☑ Additional performance gains from threaded implementations in some*) Intel IPP functions (only available via Intel IPP DLLs)

*) See www.intel.com/support/performancetools/libraries/ipp/ia/win for details on the use of threading in Intel IPP functions. [3]

Considerations:

- ☐ Installation package size: application + ~28MB RTI package for Intel IPP v3.0
- ☐ Application executable requires access to Intel IPP run-time DLLs or 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 penalty when the Intel IPP DLLs (or SOs) are first loaded

To dynamically link with Intel IPP:

1. Include ipp.h (and/or corresponding domain include files) in your code
2. Use the non-decorated Intel IPP function names (for example, ippsCopy_8u) in the application code.
3. Link corresponding domain import libraries. For example, if ippsCopy_8u is used, link against ipps20.lib (ipps.so on Linux).
4. Run-time libraries, for example `ipps20.dll` (`ipps.so` on Linux), must be on the executable search path at run time. The Run-Time Installer package is the simplest way to ensure this. For further details on the Intel IPP RTI, please consult the `readme.htm` located in the runtime folder where the Intel IPP product has been installed (typically `c:\Program Files\Intel\IPP\tools\runtime`, on Windows).

4.2. Custom Intel IPP Dynamic Linkage

If a smaller installation package is required, it is possible to accomplish this while retaining the key benefit of full processor range coverage as well as most of the other benefits of standard Intel IPP Dynamic Linkage model. This is done by creating and linking with a Custom Intel IPP Dynamic Link library, which contains only the Intel IPP functions needed by the application(s) being linked. This model is often used in application designs where Intel IPP becomes part of some "core technology" utilized by multiple applications / products by the development company.

Benefits:

- ☑ Run-time dispatching of processor-specific optimizations
- ☑ Reduced hard-drive footprint compared with full set of Intel IPP DLLs / SOs
- ☑ Smallest installation package when multiple applications use some of the same Intel IPP functions

Considerations:

- ❑ Application executable requires access to Intel IPP run-time DLLs or SOs to run
- ❑ Developer resources needed to create and maintain the Custom DLL
- ❑ Integration of new processor-specific optimizations require rebuilding of Custom DLL
- ❑ Not appropriate for kernel-mode / device-driver / ring-0 code

To build a Custom Intel IPP DLL:

1. List all Intel IPP functions that will be used. Copy and paste the function prototypes from the corresponding include files (for example, `ipps.h`) into `mydll.h` (where 'mydll' is an example name, defined by you.)
2. In `mydll.c`, write a DLL initialization function called `DllMain()`, and call `ippStaticInitBest()` in it to initialize the Intel IPP dispatching mechanism.
3. Also in `mydll.c`, write a wrapper function for each function in `mydll.h`. The wrapper function serves as a direct jump to the correct processor-specific optimized version of the Intel IPP function.
4. Compile `mydll.c` as a dynamic link library and link it against `ippsemerged.lib`, `ippsmerged.lib` and `ippcorel.lib`. The import library `mydll.lib` will be generated automatically.

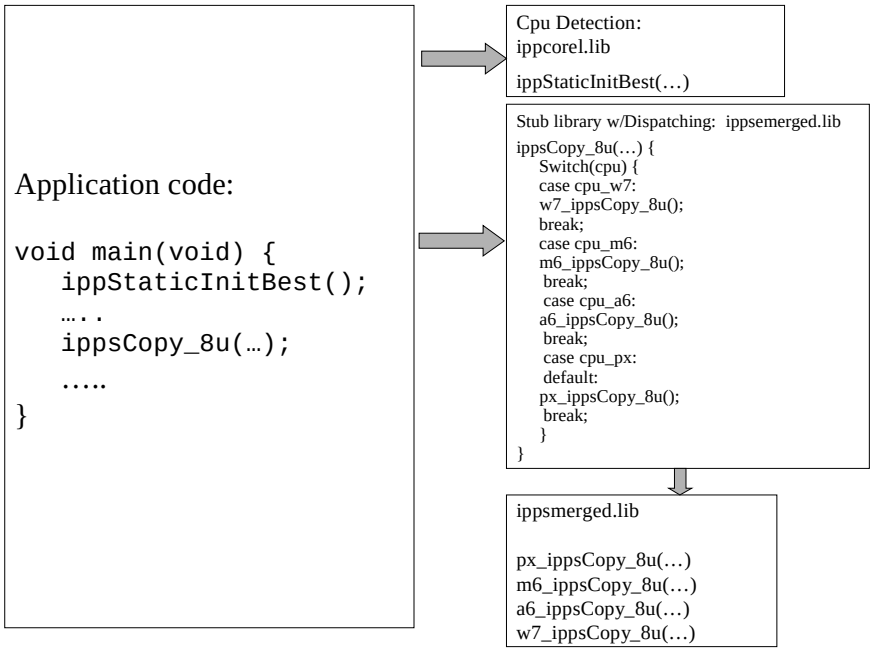
See Appendix for an example of `mydll.h` and `mydll.c`.

Once this process has been mastered, one can apply more advanced techniques such as: (1) merge other shared code into this DLL; (2) write function wrappers to link only a subset of the processor-specific optimizations.

4.3. Intel IPP Static Linkage with Dispatching (E-Merged)

If a dynamic linkage is not desired, it is possible to use a static linkage model while retaining the key benefit of full processor range coverage as well as some of the other benefits of standard Intel IPP Dynamic Link library model. This is done by linking with the Intel IPP "E-Merged" static libraries combined with the Intel IPP merged static libraries. The E-Merged libraries provide a static linkage model with processor-specific dispatching included. The distribution of applications based on this linkage model is quite simple since the application `.EXE` is self-contained.

The following figure illustrates the relations between the different static libraries used with the E-merged static dispatching model:



The library `ippcorel.lib` (`ippcorel.a` on Linux) contains the runtime processor detection functions for initializing the dispatch mechanism.

The e-merged libraries (such as `ippsemerged.lib`) provide the non-decorated entry point for the Intel IPP functions, and the dispatching mechanism to each processor-specific implementation. The corresponding processor-specific functions in the merged libraries are called by the functions in the e-merged libraries, which do not contain any implementation code.
Note that the overhead of this redirection is only one jump instruction.

The merged libraries (such as `ippsmerged.lib`) contain all processor-specific implementations. Each function entry point is prefixed by a processor identification string as outlined in section 2 of this document.

The e-merged libraries require initialization before any non-decorated function names can be called. One may choose `ippStaticInitBest()`, which initializes the library to use the best optimization available (the same waterfall procedure as in the dynamic linkage model), or function `ippStaticInitCpu()`, which lets the user designate one specific CPU type only.

This model is often the best choice when static linkage is required while the benefits of support for multiple processor-types is still clearly desired.

Benefits:

- ☑ Run-time dispatching of processor-specific optimizations
- ☑ Enables updates with new processor optimizations without recompile / relink
- ☑ Self-contained application executable
- ☑ Reduced footprint compared with full set of Intel IPP DLLs / SOs
- ☑ Smallest installation package when multiple applications use some of the same Intel IPP functions

Considerations:

- ☐ Intel IPP code duplicated for multiple Intel IPP-based applications because of static linking
- ☐ Additional function call for dispatcher initialization needed (once) during program initialization
- ☐ Not appropriate for kernel-mode / device-driver / ring-0 code

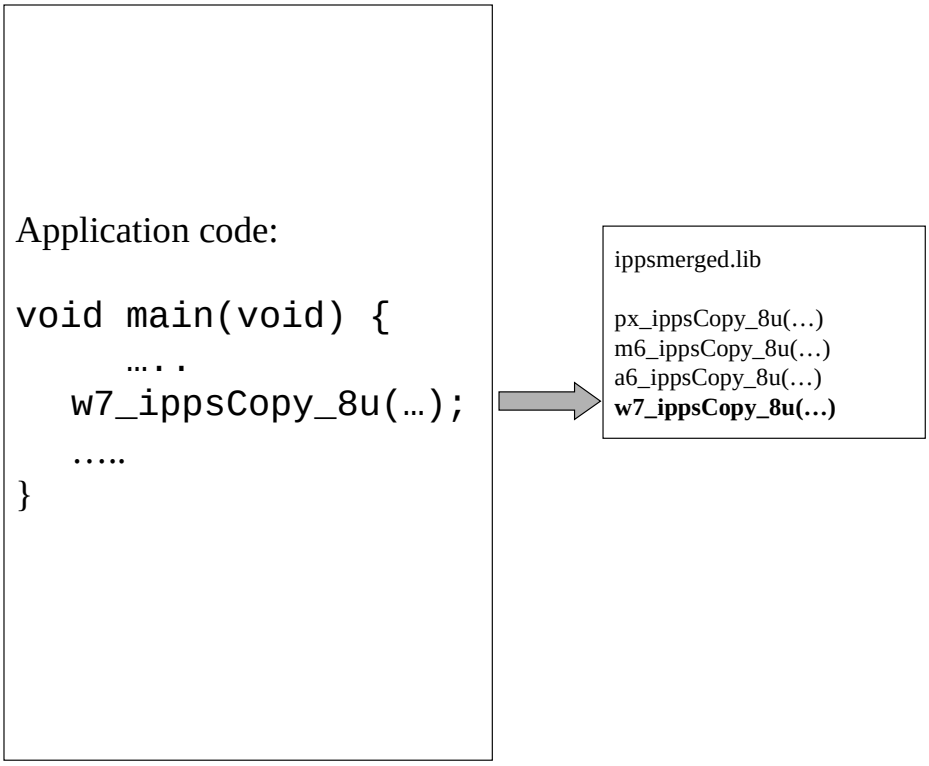
To link with Intel IPP static libraries with dispatching (E-Merged):

1. Include ippcore.h and ipp.h (and/or corresponding domain include files.)
2. Before calling any Intel IPP functions, initialize the static dispatcher by calling ippStaticInitBest(), or ippStaticInitCpu() may be used instead to force dispatching of optimizations for a specified processor-type. These functions are declared in ippcore.h.
3. Use the non-decorated Intel IPP function names (for example, ippsCopy_8u) in the application code.
4. Link corresponding e-merged libraries followed by merged libraries, and then ippscorel.lib. For example, if ippsCopy_8u is used, the linked libraries are ippsemerged.lib, ippsmerged.lib, and ippcorel.lib (order is important).

4.4. Intel IPP Static Linkage Without Dispatching (Merged Static)

The absolute smallest footprint for Intel IPP-based applications is obtained by linking with the Intel IPP Merged Static libraries. These static libraries contain copies of each processor-specific version of the Intel IPP library functions, merged into a library file specific to each Intel IPP Application-Domain. By linking with the Merged Static Libraries, only the Intel IPP functions needed by the application being linked are included in the executable, and only for the specific processor target chosen. This yields an even smaller footprint than the E-Merged Static Linkage, which includes the dispatcher and processor-specific functions for multiple target processors. *Note that while this model achieves the smallest footprint, it does so by restricting the optimizations to just one specific processor type.*

The following figure illustrates the Merged Static linkage. Note that only the processor-specific function referenced in the application code is linked into the application executable from the Intel IPP merged static library:



Each function entry point in the merged static libraries is prefixed by a processor identification string as outlined in section 2 of this document, allowing an application to directly reference the Intel IPP function for one specific (target) processor type. This eliminates the need for the initialization and dispatching code contained in `ippcorel.lib` and the e-merged libraries.

This linkage model 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. It is also the linkage model to use for kernel-mode / device-driver / ring-0 code. One common use is for embedded applications where there is no need to support multiple types of processors (i.e., the application is bundled with only one type of processor).

Benefits:

- ☑ Small executable size with support for only one processor type
- ☑ Suitable for kernel-mode / device-driver / ring-0 use
- ☑ Suitable for web applet / plug-ins requiring very small download and only one processor type of concern
- ☑ Self-contained application executable – no requirement for Intel IPP run-time DLLs or SOs to run
- ☑ Smallest footprint for application package
- ☑ Smallest installation package

Considerations:

- ☐ Only optimized for one processor type
- ☐ Updates to processor-specific optimizations require rebuild / relink

To link with Intel IPP static libraries without dispatching:

1. Prior to including `ipp.h` in your code, define the following preprocessor macros:
   ```
   #define IPPAPI(type,name,arg) extern type __STDCALL w7_##name arg;
   #define IPPCALL(name) w7_##name
   ```
2. Include `ipp.h` in your code
3. Wrap each Intel IPP function call in your application with an `IPPCALL` macro as shown below. For example, a call to `ippsCopy_8u(…)` would be expressed as:
   `IPPCALL(ippsCopy_8u)(…)`
4. Link against the appropriate merged static library (i.e., `ippsmerged.lib`).

5. Summary

Intel IPP provides various linkage models to fit a wide range of developer needs. With two dynamic linkage models and two static linkage models to choose from, a developer can exercise a high degree of control over the impact on disk space and memory requirements of their application, as well as the installation complexity. Correct implementation of a chosen linkage model allows the developer to easily change to another linkage model with relatively little effort. Choosing the best linkage model is an important step in getting the most from Intel IPP processor-specific optimizations.

Appendix. Custom Dynamic Linkage Example

```
========= mydll.h ==========
#ifndef __MYDLL_H__
#define __MYDLL_H__

#ifndef IPPAPI
#define IPPAPI(type,name,args)
#include "ipps.h"
#undef  IPPAPI
#define IPPAPI(type,name,args) extern type __STDCALL my_##name args;
#endif

#ifdef __cplusplus
extern "C" {
#endif

/* List Function Prototypes Here */
IPPAPI(IppStatus, ippsCopy_8u,( const Ipp8u* pSrc, Ipp8u* pDst, int len ))
IPPAPI(IppStatus, ippsCopy_16s,( const Ipp16s* pSrc, Ipp16s* pDst, int len ))
#ifdef __cplusplus
}
#endif
#endif // __MYDLL_H__
```

```
======== mydll.cpp =========
#define STRICT
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "ippcore.h"
#include "ipps.h"

#undef  IPPAPI
#define IPPAPI(type,name,args) __declspec(naked dllexport) \
void __STDCALL my_##name args { __asm {jmp name } }
#include "mydll.h"

BOOL WINAPI DllMain(HINSTANCE hinstDLL,DWORD fdwReason,LPVOID lpvReserved) {
switch( fdwReason ) {
case DLL_PROCESS_ATTACH:
if (ippStaticInitBest()!=ippStsNoErr) return FALSE;
default:
hinstDLL;
lpvReserved;
break;
}
return TRUE;
}
```

References

The following documents are referenced in this application note, and provide background and/or supporting information for understanding the topics presented in this document.

[1] Intel® Integrated Performance Primitives, http://www.intel.com/software/products/ipp/ipp30/.

[2] Intel® Integrated Performance Primitives for Intel® Architecture Reference Manual, under http://www.intel.com/software/products/ipp/docs/manuals.htm.

[3] Details on the use of threading in Intel IPP functions: http://www.intel.com/support/performancetools/libraries/ipp/ia/thread.htm