WHITE PAPER

Choosing the Best Intel® Integrated Performance Primitives Linkage Model for your Application for Intel® Architectures

Version 1.0
December 2003

Introduction

Intel® Integrated Performance Primitives (Intel® IPP) is a software library that provides a broad range of functionality including general signal, image, speech, graphics and audio processing, vector manipulation, matrix math, string processing and cryptography. It also provides more sophisticated primitives for construction of audio, video and speech codecs such as MP3, MPEG-1, MPEG-2, MPEG-4, H.264, JPEG, JPEG2000, G.722, G.723.1, G.726, G.728, and G.729, plus algorithms for speech recognition and computer vision.

By supporting a variety of data types and layouts for each function, and by 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, and some functions offer automatic run-time dispatching of the best optimizations. For more information about Intel IPP, visit www.intel.com/software/products/ipp/

Intel IPP is optimized for the broad range of Intel® microprocessors: the Intel® Pentium® 4 processor with Streaming SIMD Extensions 3; the Intel® Pentium® 4 processor; the Intel® Pentium M processor, component of Intel® Centrino™ mobile technology[1]; the Intel® Itanium® 2 processor; Intel® Xeon™ processors; and Intel® Personal Internet Client Architecture (Intel® PCA) application processors based on the Intel XScale® microarchitecture. With a single application programming interface (API) across the range of architectures, developers can have platform compatibility and reduced cost of development. The built-in dispatcher capability in Intel IPP chooses the best optimizations, and run-time processor detection automatically dispatches processor-specific code.

For Intel PCA, Intel IPP provides static-linked libraries for embedded applications. For Pentium and Itanium architectures, Intel IPP supports several different linkage models, allowing the developer to choose one that best suits their application development environment and application deployment constraints. Table 1 summarizes the choices.

Table 1. The Four Intel® Integrated Performance Primitives (Intel® IPP) Linkage Models

Linkage Model	Description
Intel® IPP Dynamic Linkage	This is the full set of Intel IPP library functions, including all functions, optimizations for all processors, and automatic run-time dispatching for multiple processor types. The Dynamic Link Libraries (DLLs) supporting this linkage model are pre-packaged as the Intel IPP Runtime Installer (RTI) for easy redistribution with developers' applications.
Custom Intel® IPP Dynamic Linkage	This is a 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 only those functions/libraries used by the specific application.
Intel® IPP Static Linkage with Dispatching (E-Merged)	This is a compact, self-contained application (no requirement for Intel IPP run-time DLLs), with automatic dispatching for multiple processor types. This model is 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	This model has the smallest footprint, is optimized for only one target processor type, and is best suited for kernel-mode or driver use.

This document compares and contrasts each of the linkage models and discusses the pros and cons of each. The discussions include these considerations:

- Relative ease-of-use
- Applications for which each model is best suited
- Memory and disk footprints
- Requirements for rebuilding when integrating Intel IPP updates
- Requirements for application installation

This document begins with 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.

Organization of Intel Integrated Performance Primitives Functions

Intel IPP functions are primarily organized in the following ways (as of Version 4.0):

Data Domain

This is 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 Manuals. The three Intel IPP Data Domains are:

- **Signal Processing:** Operations on one dimensional (1D) vectors of "signal" data. The function-name prefix is "`ipps`"
- **Image Processing:** Operations on (2D) arrays of "image" data. The function-name prefix is "`ippi`"
- **Matrix:** Operations on matrices, specifically optimized for small matrices. The function-name prefix is "`ippm`"

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 contains a set of static and dynamic libraries that use consistent naming conventions. 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 4.0 of Intel IPP are currently:

- `ipps` signal processing functions
- `ippi` image processing functions
- `ippj` JPEG/JPEG2000 functions
- `ippsr` speech recognition functions
- `ippsc` speech coding functions
- `ippac` audio coding functions
- `ippvc` video coding functions
- `ippm` small matrix operation functions
- `ippvm` vector math functions
- `ippcv` computer vision functions
- `ippcore` Intel IPP common and generic functions
- `ippch` string functions
- `ippcp` cryptographic functions
- `ippalign` Intel PCA compatibility functions (depreciated)

The Application-Domain grouping (or the library name) is reflected in the chapter names in the *Intel IPP Reference Manuals* at www.intel.com/software/products/ipp/docs/manuals.htm. 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:

The Application-Domain grouping is distinct from the Intel IPP Data Domain function prefixes, which denote only the data dimension – `ipps` for one-dimensional and `ippi` for two-dimensional data.

Target Processor

The libraries are also grouped by processor-specific optimizations using the following naming convention (in Version 4.0 of Intel IPP):

- `px` generic C-optimized code
- `a6` code optimized for Pentium III processor
- `w7` code optimized for Pentium 4 processor
- `t7` code optimized for the Intel Pentium 4 processor with Streaming SIMD Extensions 3
- `i7` code optimized for Itanium and Itanium 2 processors

Be mindful of the processor identifiers whenever creating Intel IPP-based applications without the automatic processor-specific dispatching. Be careful specifying the processor identifier for processor-specific versions of Intel IPP functions. Choosing the wrong identifier could lead to installation of the wrong 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 identification. This is often referred to as the "decorated name." For example, `ippsmerged.lib` (`libippsmerged.a` for Linux) contains `px_ippsCopy_8u`, `a6_ippsCopy_8`, `w7_ippsCopy_8u`, and `t7_ippsCopy_8u`.

The code optimized for Itanium processors is in a separate library due to the different 64-bit format.

Things to Consider When Choosing a Linkage Model

When using a library of functions in the development of an application, developers must decide whether the functions will be statically linked or dynamically linked with the application. The choices are relatively simple. They are based on a trade-off of complexity versus size versus 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 an 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, developers must 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. Before choosing a linkage model, developers should answer these questions:

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' computers?
4. Memory available at run-time. What are the memory restrictions on the users' computers 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 supported only on a single type of processor? Is the application part of an embedded computer 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.

Linkage Models and Calling Conventions

In addition to the usual choices of static versus 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). For summary descriptions of the models, see Table 1. The descriptions that follow provide more information about how and when to use each model.

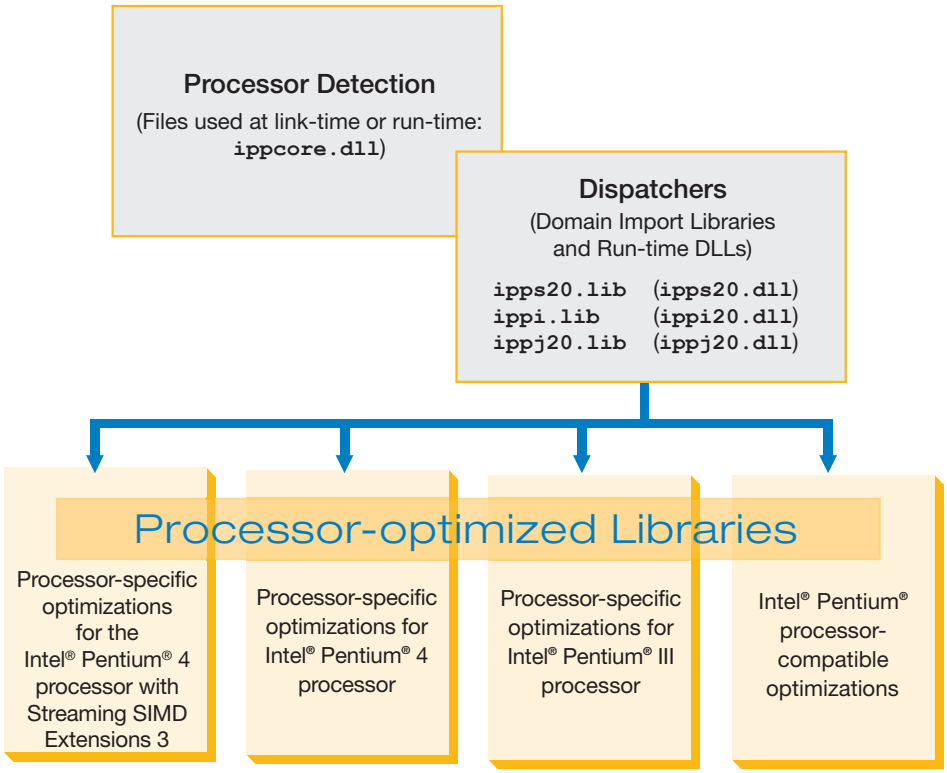
Intel Integrated Performance Primitives Dynamic Linkage

The Intel® IPP Dynamic Link libraries (or Shared Objects on Linux*) is the model that is simplest to use and probably the most commonly used. 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 outweigh most other concerns. This model offers the best way to ensure that the end-users of the application experience great performance on their PCs.

By detecting the processor type at runtime during the DLL initialization, this model 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 Figure 1.

The import libraries `ipps20.lib`, `ippi20.lib`, etc. are used at link time. The runtime library `ippcore.dll` features the processor detection mechanism, and `ipps20.dll`, `ippi20.dll`, etc. direct 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 processor identification, and then a "waterfall" procedure is carried out to determine the best available optimization (`t7`, `w7`, `a6`, 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. For Linux, the library names are slightly different, but the same mechanism applies.

Figure 1. Intel® Integrated Performance Primitives Dynamic Linkage



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

This model offers:

- 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
- 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/thread.htm for details on the use of threading in Intel IPP functions

Considerations

Before using this model, consider these implications:

- Installation package size: application + ~27-MB RTI package for Intel IPP 4.0
- Application executable requires access to Intel IPP run-time dynamic link libraries (DLLs) or Shared Objects (SOs) to run
- Not appropriate for kernel-mode/device-driver/ring-0 code
- Not appropriate for web applets/plug-ins that require very small download
- There is a one-time performance penalty when the Intel IPP DLLs (or SOs) are first loaded

To dynamically link with Intel Integrated Performance Primitives

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. Ensure run-time libraries, for example `ipps20.dll` (`ipps.so` on Linux), are 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, 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).

Custom Dynamic Linkage in Intel Integrated Performance Primitives

If a smaller installation package is required, developers can accomplish this while retaining the key benefit of full processor range coverage and most of the other benefits of the standard Intel IPP Dynamic Linkage model. The Custom Intel IPP Dynamic Link library, which provides only the Intel IPP functions needed by the application(s) being linked, offers this smaller solution. For companies that develop core technologies, code that ships with many of their products must run on any Intel processor while still having a small footprint. The Custom Intel IPP Dynamic Link library is the right model.

Benefits

This model offers:

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

Considerations

Before using this model, consider these implications:

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

To build a Custom 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; replace it with your own).
2. In `mydll.c`, write a DLL initialization function called `DllMain()`, and from it call `ippStaticInitBest()` 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 the Appendix for an example of `mydll.h` and `mydll.c`.

Once this process has been mastered, developers can apply more advanced techniques, such as merging other shared code into this DLL or writing function wrappers to link only a subset of the processor-specific optimizations.

Static Linkage with Dispatching (E-Merged)

If a dynamic linkage is not desired, developers can use a static linkage model while retaining the key benefits of full processor range coverage and some of the other benefits of the 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 pre-determined dispatching mechanism for application developers. If not all processor optimizations are required, it is possible to create customized E-merged libraries that only contain a subset of processor dispatching. 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.

Figure 2 illustrates the relationship 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.

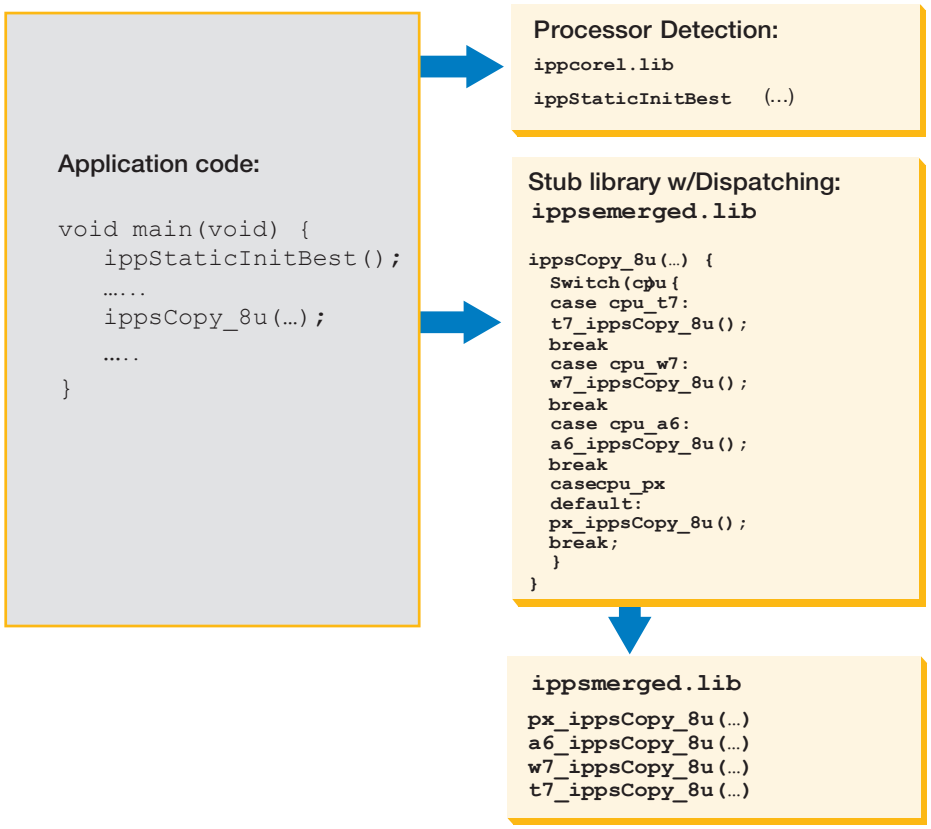
The E-merged libraries (such as `ippsemerged.lib`) provide the non-decorated entry point for the Intel IPP functions, along with the dispatching mechanism to each processor-specific implementation. The corresponding processor-specific functions in the merged libraries are called by the functions in the E-merged libraries, which do not contain any implementation code. *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 the "Organization of Intel IPP Functions" discussion in this document.

The E-merged libraries require initialization before any non-decorated function names can be called. To initialize first, developers 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 developers designate only one specific processor type.

E-Merged libraries provide a pre-determined dispatching mechanism for application developers. If some processor optimizations are not required, developers can create customized E-merged libraries that contain only a subset of processor dispatching. This model is often the best choice when static linkage is required along with support for multiple processor types.

Figure 2. Static Linkage with E-merged Dispatching Model



Benefits

This model offers the following features:

- Dispatches processor-specific optimizations during run-time
- Enables updates with new processor optimizations without recompile/relink
- Creates a self-contained application executable
- Generates a smaller footprint than the full set of Intel IPP DLLs/SOs
- Creates the smallest installation package when multiple applications use some of the same Intel IPP functions

Considerations

Before using this model, consider these implications:

- 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
- This model is not appropriate for kernel-mode/ device-driver/ring-0 code

To link with 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`. Alternatively, use `ippStaticInitCpu` instead to force dispatching of optimizations for a specified processor-type. Declare these functions 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).

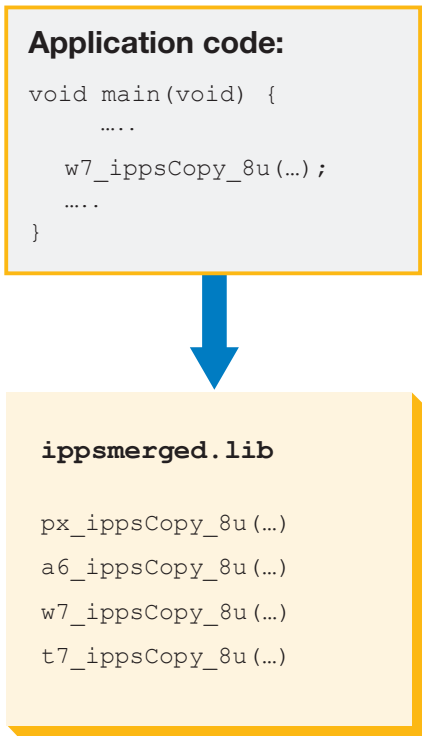
Static Linkage Without Dispatching (Merged Static)

To obtain the smallest footprint for Intel IPP-based applications, link 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. Linking with the Merged Static Libraries yields an executable containing no unneeded Intel IPP functions. It also contains code only for the chosen processor. The result is a footprint even smaller than the E-Merged Static Linkage, which includes the dispatcher and processor-specific functions for multiple target processors. *While this model achieves the smallest footprint, it does so by restricting the optimizations to one specific processor type.*

Figure 3 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 earlier in 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.

Figure 3. Merged Static Linkage



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 the application is bundled with only one type of processor.

Benefits

This model offers:

- 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 DLLs or SOs to run
- Smallest footprint for application package
- Smallest installation package

Considerations

Before using this model, consider these implications:

- The executable is optimized for only one processor type
- Updates to processor-specific optimizations require rebuild and/or relink

To link with 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 (for example, `ippsmerged.lib`).

Summary

Intel IPP provides several linkage models to fit a wide range of developer needs. With two dynamic linkage models and two static linkage models to choose from,

developers can exercise a high degree of control over the disk space and memory requirements of their applications, as well as the installation complexity. Correct implementation of a chosen linkage model allows the developer to 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.

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.

Intel® Integrated Performance Primitives are described at: www.intel.com/software/products/ipp/

Intel® Integrated Performance Primitives for Intel® Architecture Reference Manual, is located at: www.intel.com/software/products/ipp/docs/manuals.htm

Details on the use of threading in Intel IPP functions are provided at: www.intel.com/support/performancetools/libraries/ipp/ia/thread.htm

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;
}
```

[1] Wireless connectivity requires additional software, services or external hardware that may need to be purchased separately. Availability of public wireless access points is limited. System performance, battery life and functionality will vary depending on your specific hardware and software.

Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052-8119
USA

For product and purchase information visit:

www.intel.com/software/products

Intel, the Intel Logo, Itanium, Pentium, Intel Xeon, Intel XScale and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

Copyright © 2003, Intel Corporation. All Rights Reserved. 1203/JXP/ITF/XX/PDF

300350-001