Uniform Driver Interface

UDI IA-32/IA-64 ABI Binding Specification Version 1.01

http://www.project-UDI.org/specs.html

UDI IA-32/IA-64 ABI Binding Specification

Abstract

The UDI IA-32/IA-64 ABI Binding Specification defines binary bindings of the UDI Core Specification to the Intel 32-bit and 64-bit processor instruction set architectures referred to as IA-32 and IA-64, respectively, allowing binary portability of UDI drivers across environments supporting these ABIs.

This is an optional extension to the UDI Core Specification, which is defined in a separate book. The intended audience for this book includes environment implementors and build utility implementors.

See the Document Organization chapter in the UDI Core Specification for a description of other books in the UDI Specification collection, as well as references to additional tutorial materials.

Status of This Document

This document has been reviewed by Project UDI Members and other interested parties and has been endorsed as a Final Specification. It is a stable document and may be used as reference material or cited as a normative reference from another document. This version of the specification is intended to be ready for use in product design and implementation. Every attempt has been made to ensure a consistent and implementable specification. Implementations should ensure compliance with this version.

Copyright Notice

Acknowledgements

The authors would like to thank everyone who reviewed working drafts of the specification and submitted suggestions and corrections.

The authors would especially like to thank their significant others for putting up with the many hours of overtime put into the development of this specification over long periods.

Thanks to the following folks who contributed significant amounts of time, ideas, or authoring in support of the development of this specification or in working on the prototype implementations which helped us validate the specification:

Allyn Bowers (Intel)
Steve Bytnar (System Technologies Group)
Mark Evenson (HP)
Kurt Gollhardt (SCO)
Matt Kaufmann (SCO)
Robert Lipe (SCO)
Scott Popp (SCO)
Kevin Quick (Interphase)
John Ronciak (Intel)
Rob Tarte (Pacific CodeWorks)
Linda Wang (Sun)

Finally, thanks to David Roberts (Certek Software Designs) for designing the Project UDI logo.

Table of Contents

Abstract iii
Copyright Notice iv
Acknowledgements v
Table of Contents vii

1 Introduction to the IA-32/IA-64 ABI Bindings 1-1
1.1 Overview 1-1
1.2 General Requirements 1-1
1.3 Normative References 1-2

2 IA-32 ABI Binding 2-1
2.1 Processor Architecture 2-1
2.1.1 Supported processor types 2-1
2.1.2 Endianness 2-1
2.1.3 Driver packaging subdirectories 2-1
2.2 Runtime Architecture 2-2
2.2.1 Binary Bindings to the Source Specifications 2-2
2.2.1.1 Sizes of UDI-specific data types 2-3
2.2.1.2 Implementation-dependent macros 2-3
2.2.1.3 UDI functions implemented as macros 2-4
2.2.2 Building the Driver Object 2-4
2.2.2.1 Object file format 2-4
2.2.2.2 Static driver properties encapsulation 2-5

3 IA-64 ABI Binding 3-1
3.1 Processor Architecture 3-1
3.1.1 Supported processor types 3-1
3.1.2 Endianness 3-1
3.1.3 Driver packaging subdirectories 3-1
3.2 Runtime Architecture 3-2
3.2.1 Binary Bindings to the Source Specifications 3-2
3.2.1.1 Sizes of UDI-specific data types 3-2
3.2.1.2 Implementation-dependent macros 3-3
3.2.1.3 UDI functions implemented as macros 3-4

3.2.2 Building the Driver Object 3-4
3.2.2.1 Object file format 3-4
3.2.2.2 Static driver properties encapsulation 3-4

Introduction to the IA-32/IA-64 ABI Bindings 1

1.1 Overview

The UDI IA-32/IA-64 ABI Binding Specification defines binary bindings of the UDI Core Specification to the Intel 32-bit and 64-bit processor instruction set architectures referred to as IA-32 and IA-64, respectively, allowing binary portability of UDI drivers across environments supporting these ABIs.

This chapter defines general requirements for use of the UDI IA-32/IA-64 ABI Binding Specification. Chapter 2 defines the specifics of the IA-32 ABI Binding. Chapter 3 defines the specifics of the IA-64 ABI Binding.

1.2 General Requirements

Build utilities that build UDI binary modules targeted for these ABIs must create such binary modules in conformance with the definitions and requirements in this Specification. UDI environments that support installing UDI binary modules for these ABIs must interpret such modules in conformance with the definitions and requirements in this Specification.

Binary modules conforming to this Specification must contain no global external symbol definitions besides those explicitly exported via "provides" declarations or explicitly required by the UDI Core Specification: `udi_init_info` for driver modules, and `udi_meta_info` for metalanguage library modules. Any such module may contain unresolved external symbol references, but if any such symbols not explicitly listed in this Specification are outside the union of all symbol names explicitly exported (via “provides” declaration or similar mechanism) by interfaces listed in the module’s “requires” declarations, the module shall be considered in error, and the behavior of any attempt to use the module in a target environment is implementation-dependent.

As described in Chapter 33, *“Introduction to ABI Bindings”*, of the UDI Core Specification, a UDI ABI binding must specify the following components:

1. A processor architecture, instruction set definition, and endianness. This defines the supported instruction sets, and endianness of the compiled UDI driver, as well as corresponding subdirectory names for the UDI packaging format.
2. Runtime architecture. This defines the procedure calling conventions, register usage, stack conventions, data layouts, etc.
3. Binary bindings to the source-level UDI specifications. This specifies the sizes of fundamental UDI data types, the binary-portability requirements of *implementation-dependent UDI macros* and functional interfaces, and other miscellaneous binary bindings related to the source-level specifications.
4. Building the driver object. This specifies the object file format, and the encapsulation of the static driver properties in the object files.

1.3 Normative References

The UDI IA-32/IA-64 ABI Binding Specification references the following non-UDI specifications, listed below. These specifications contain provisions that, through reference in this document, constitute provisions of the UDI IA-32/IA-64 ABI Binding Specification.

1. *System V Application Binary Interface Specification* (http://www.sco.com/developer/devspecs)
2. *System V Application Binary Interface Intel 386 Architecture Processor Supplement Specification* (http://www.sco.com/developer/devspecs)
3. *IA-64 Software Conventions and Runtime Architecture Guide* (http://developer.intel.com/design/ia64/devinfo.htm)
4. *ELF-64 Object File Format Specification* (http://www.sco.com/developer/gabi/contents.html)
5. *Processor-Specific ELF Supplement for IA-64* (available from Intel Corporation)

The UDI IA-32/IA-64 ABI Binding Specification also references and depends on the UDI Core Specification and the UDI Physical I/O Specification. (Note, though, that conformance to this Specification does not require conformance to the UDI Physical I/O Specification; all references to the UDI Physical I/O Specification are relevant only to drivers and environments that conform to the UDI Physical I/O Specification).

IA-32 ABI Binding 2

The IA-32 ABI binding defines binary bindings for UDI modules that run on IA-32 processors. This allows a UDI driver or library to be compiled once for IA-32 and distributed to any conforming IA-32 platform. (However, this does not preclude source distributions).

2.1 Processor Architecture

2.1.1 Supported processor types

The IA-32 binding works across any processor that conforms to the 32-bit Intel architecture. This includes the Intel® processor types listed in Table 2-1 and all compatible clones by other manufacturers (e.g. AMD, Cyrix, etc.). For maximum binary portability across IA-32 processors, you must compile a UDI module so that it does not depend on any specific IA-32 target processor type. Driver packages must use the subdirectory names specified in Table 2-1 for their IA-32 driver binaries. All binary driver packages for IA-32 must include at least a generic `ia32` binary, but may also include one or more specific binaries.

2.1.2 Endianness

The IA-32 ABI binding supports only little-endian IA-32 binary UDI modules.

2.1.3 Driver packaging subdirectories

Chapter 31, *"Packaging & Distribution Format"*, of the UDI Core Specification defines a directory hierarchy for the contents of UDI driver packages. This hierarchy must contain the various components of a UDI driver "package". The package includes the files that must be provided with a driver to make it installable and useable. In a driver package, one or more "`bin`" directories contain the driver and/or library binary modules. Beneath the "`bin`" directory, `<abi>` subdirectories contain the binaries for a particular ABI. (See Chapter 31, *"Packaging & Distribution Format"*, of the UDI Core Specification for more information.)

The `<abi>` subdirectory names defined for the IA-32 ABI, with their corresponding processor types, are defined in Table 2-1 below.

Table 2-1 `<abi>` subdirectories for IA-32

Directory Name	Processor Type
`ia32`	Any IA-32 Processor
`ia32_p`	Any Pentium Processor

Table 2-1 <abi> subdirectories for IA-32

Directory Name	Processor Type
ia32_p4	Pentium-4 Processor
ia32_p3	Pentium III Processor
ia32_p2	Pentium II Processor
ia32_pmmx	Pentium Processor with MMX™ Technology
ia32_ppro	Pentium Pro Processor
ia32_486	Intel486™ Processor
ia32_386	Intel386™ Processor

Note that the Pentium II Xeon™ and Celeron™ processors share the same instruction set with the Pentium II processor and thus correspond to ia32_p2. Likewise, the Pentium III Xeon processor corresponds to ia32_p3.

2.2 Runtime Architecture

The runtime architecture for this IA-32 ABI binding shall be the IA-32 runtime architecture as defined in the *System V Application Binary Interface Specification* (http://www.sco.com/developer/devspecs) and the *System V Application Binary Interface Intel 386 Architecture Processor Supplement Specification* (http://www.sco.com/developer/devspecs).

In particular, a compliant environment must provide an execution environment, implement object file support, and provide linking and loading.

2.2.1 Binary Bindings to the Source Specifications

This section defines the binary bindings to the source-level UDI Specifications. Only a few aspects of the source-level UDI Specifications are implementation-dependent, requiring additional specification for binary portability. These fall into three categories: the sizes of fundamental UDI data types, the binary-portability requirements of implementation-dependent UDI macros, and the specification of UDI functional interfaces, beyond the UDI utility functions, that are allowed to be implemented as macros. Utility functions in UDI can always be implemented as macros without breaking binary portability, but ABI-conforming environments must always provide functional versions, to allow for build environments that choose not to do so. For more information, see Chapter 19, *"Introduction to Utility Functions"*, of the UDI Core Specification.

2.2.1.1 Sizes of UDI-specific data types

As indicated in Chapter 33, *"Introduction to ABI Bindings"*, of the UDI Core Specification, an ABI specification must specify the size of each of the following data types, defined as follows for the IA-32 ABI binding.

Table 2-2 UDI-specific data type sizes for IA-32

Data Type	Size	Description
`void *`	32 bits	Pointer type
`udi_size_t`	32 bits	Abstract type – Size type
`udi_index_t`	8 bits	Abstract type – Index type
`udi_buf_path_t`	32 bits	Opaque type – Buffer path handle
`udi_channel_t`	32 bits	Opaque type – Channel handle
`udi_origin_t`	32 bits	Opaque type – Control block origin handle
`udi_timestamp_t`	32 bits	Opaque type – Timestamp type
`udi_dma_constraints_t`	32 bits	Opaque type – DMA constraints handle
`udi_dma_handle_t`	32 bits	Opaque type – DMA handle
`udi_pio_handle_t`	32 bits	Opaque type – PIO handle

2.2.1.2 Implementation-dependent macros

An implementation-dependent macro is an interface specified to be a macro, other than a utility macro, in a UDI Specification. The expansion of such macros is not specified at the source level, but in order to allow binary portability an ABI must specify the effect of such expansion on code generation, especially including any external symbol references. Further, any environment or platform dependencies must be hidden behind external function calls contained in the macro expansion, so that the compiled UDI module can be installed on different platforms.

Only the UDI Core Specification and the UDI Physical I/O Specification may specify implementation-dependent macros. There are only two such macros in those specifications: the `UDI_HANDLE_ID` macro and the `UDI_HANDLE_IS_NULL` macro.

For the IA-32 ABI, the `UDI_HANDLE_ID` macro must return the exact, non-transformed contents of the specified handle, without producing any external symbol references. The actual C definition of the macro may be provided in an implementation-dependent manner in udi.h. Depending on the actual type definition of the handle type, acceptable definitions include:

```
#define UDI_HANDLE_ID(handle, handle_type) \
                    ((void *)(handle))
```

and:

```
#define UDI_HANDLE_ID(handle, handle_type) \
                    (*(void **)&(handle))
```

The `UDI_HANDLE_IS_NULL` macro must return TRUE if the handle value is zero. For the IA-32 ABI, all handle types are 32 bits, so this can be done using appropriate `(void *)` casts and comparing against NULL. The actual C definition of the macro may be provided in an implementation-dependent manner in `udi.h`. Depending on the actual type definition of the handle type, acceptable definitions include:

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                    ((void *)(handle) == NULL)
```

and:

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                    (*(void **)&(handle) == NULL)
```

2.2.1.3 UDI functions implemented as macros

As previously noted, an ABI binding must specify for each non-utility UDI functional interface whether it must always be implemented as an external function call, must be implemented as a macro, or may be either.

For the IA-32 ABI, all non-utility functional interfaces must be implemented as external function calls, except `udi_assert`, which must be implemented either as a macro that does nothing or as a macro that calls the following function when the assertion condition test fails:

```
void __udi_assert(const char *expr,
                  const char *filename,
                  int lineno);
```

2.2.2 Building the Driver Object

The UDI Core Specification defines the generic aspects of the packaging and distribution of UDI drivers, whether distributed in source or binary format. The ABI binding defines the object file format and the binding of the static driver properties to the driver's object files.

2.2.2.1 Object file format

The object file format for this IA-32 binding shall be the ELF-32 object file format as defined in the *System V Application Binary Interface Specification* and the *System V Application Binary Interface Intel 386 Architecture Processor Supplement Specification*.

Any driver package that includes IA-32 binary modules in a particular alternate version within the package must include IA-32 binary modules for all modules in that alternate.

2.2.2.2 Static driver properties encapsulation

A driver's static driver properties shall be encapsulated in the ELF-32 object file of the driver's primary module, in a special no-load section of that object file called, "`.udiprops`". The contents of this section are ASCII-encoded, null-terminated strings from the "`udiprops.txt`" file. The only differences between the contents of the "`udiprops.txt`" file and the "`.udiprops`" section are that:

- Comments are removed.
- Extraneous white space is removed.
- All line terminator characters are removed, and each declaration is appended with a null character ('\0'), so that each declaration is stored as a single, null terminated string.
- Blank lines may be present (i.e. a null character may appear without any preceeding non-null text, even at the beginning and end of the properties section).

IA-64 ABI Binding 3

The IA-64 ABI binding defines binary bindings for UDI modules that run on IA-64 processors. This allows a UDI driver or library to be compiled once for IA-64 (for a particular endianness) and distributed to any conforming IA-64 platform that supports the same endianness. (However, this does not preclude source distributions).

3.1 Processor Architecture

3.1.1 Supported processor types

The IA-64 binding is designed to work across any processor that conforms to the 64-bit Intel Architecture. This includes the Intel® processor types listed in Table 3-1. For maximum binary portability across IA-64 processors, you must compile a UDI module so that it does not depend on any specific IA-64 target processor type. Driver packages must use the subdirectory names specified in Table 3-1 for their IA-64 binaries. All binary driver packages for IA-64 must include at least a generic `ia64_le` or `ia64_be` binary, but may also include one or more specific binaries.

3.1.2 Endianness

The IA-64 ABI binding supports both big- and little-endian IA-64 binary UDI modules, but conformant build tools and environments are only required to support one endianness or the other.

It is recommended, but not required, that little-endian binaries be supported.

3.1.3 Driver packaging subdirectories

Chapter 31, *"Packaging & Distribution Format"*, of the UDI Core Specification defines a directory hierarchy for the contents of UDI driver packages. This hierarchy must contain the various components of a UDI driver "package". The package includes the files that must be provided with a driver to make it installable and useable. In a driver package, one or more "`bin`" directories contain the driver and/or library binary modules. Beneath the "`bin`" directory, `<abi>` subdirectories contain the binaries for a particular ABI. (See Chapter 31, *"Packaging & Distribution Format"*, of the UDI Core Specification for more information.)

The <abi> subdirectory names defined for the IA-64 ABI, with their corresponding processor types, are defined in Table 3-1 below.

Table 3-1 <abi> subdirectories for IA-64

Directory Name	Processor Type
ia64_le	Any IA-64 Processor in little-endian mode
ia64_be	Any IA-64 Processor in big-endian mode
ia64_le_itanium	Intel Itanium™ Processor in little-endian mode
ia64_be_itanium	Intel Itanium™ Processor in big-endian mode

3.2 Runtime Architecture

The runtime architecture for this IA-64 ABI binding shall be the IA-64 runtime architecture as defined in the *IA-64 Software Conventions and Runtime Architecture Guide* (http://developer.intel.com/design/ia64/devinfo.htm) from Intel and Hewlett-Packard. This is the definition of the runtime architecture that is common to all operating environments on the IA-64 architecture.

3.2.1 Binary Bindings to the Source Specifications

This section defines the binary bindings to the source-level UDI Specifications. Only a few aspects of the source-level UDI Specifications are implementation-dependent, requiring additional specification for binary portability. These fall into three categories: the sizes of fundamental UDI data types, the binary-portability requirements of implementation-dependent UDI macros, and the specification of UDI functional interfaces, beyond the UDI utility functions, that are allowed to be implemented as macros. Utility functions in UDI can always be implemented as macros without breaking binary portability, but ABI-conforming environments must always provide functional versions, to allow for build environments that choose not to do so. For more information, see Chapter 19, *"Introduction to Utility Functions"*, of the UDI Core Specification.

3.2.1.1 Sizes of UDI-specific data types

As indicated in Chapter 33, *"Introduction to ABI Bindings"*, of the UDI Core Specification, an ABI specification must specify the size of each of the following data types, defined as follows for the IA-64 ABI binding.

Table 3-2 UDI-specific data type sizes for IA-64

Data Type	Size	Description
void *	64 bits	Pointer type
udi_size_t	64 bits	Abstract type – Size type
udi_index_t	8 bits	Abstract type – Index type
udi_buf_path_t	64 bits	Opaque type – Buffer path handle

Table 3-2 UDI-specific data type sizes for IA-64

Data Type	Size	Description
udi_channel_t	64 bits	Opaque type – Channel handle
udi_origin_t	64 bits	Opaque type – Control block origin handle
udi_timestamp_t	64 bits	Opaque type – Timestamp type
udi_dma_constraints_t	64 bits	Opaque type – DMA constraints handle
udi_dma_handle_t	64 bits	Opaque type – DMA handle
udi_pio_handle_t	64 bits	Opaque type – PIO handle

3.2.1.2 Implementation-dependent macros

An implementation-dependent macro is an interface specified to be a macro, other than a utility macro, in a UDI Specification. The expansion of such macros is not specified at the source level, but in order to allow binary portability an ABI must specify the effect of such expansion on code generation, especially including any external symbol references. Further, any environment or platform dependencies must be hidden behind external function calls contained in the macro expansion, so that the compiled UDI module can be installed on different platforms.

Only the UDI Core Specification and the UDI Physical I/O Specification may specify implementation-dependent macros. There are only two such macros in those specifications: the `UDI_HANDLE_ID` macro and the `UDI_HANDLE_IS_NULL` macro.

For the IA-64 ABI, the `UDI_HANDLE_ID` macro must return the exact, non-transformed contents of the specified handle, without producing any external symbol references. The actual C definition of the macro may be provided in an implementation-dependent manner in udi.h. Depending on the actual type definition of the handle type, and the target programming model (e.g. LP64 vs P64), acceptable definitions include:

```
#define UDI_HANDLE_ID(handle, handle_type) \
                    ((void *)(handle))
```

and:

```
#define UDI_HANDLE_ID(handle, handle_type) \
                    (*(void **)&(handle))
```

The `UDI_HANDLE_IS_NULL` macro must return TRUE if the handle value is zero. For the IA-64 ABI, all handle types are 64 bits, so this can be done using appropriate `(void *)` casts and comparing against NULL. The actual C definition of the macro may be provided in an implementation-dependent manner in `udi.h`. Depending on the actual type definition of the handle type, acceptable definitions include:

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                    ((void *)(handle) == NULL)
```

and:

```
#define UDI_HANDLE_IS_NULL(handle, handle_type) \
                    (*(void **)&(handle) == NULL)
```

3.2.1.3 UDI functions implemented as macros

As previously noted, an ABI binding must specify for each non-utility UDI functional interface whether it must always be implemented as an external function call, must be implemented as a macro, or may be either.

For the IA-64 ABI, all non-utility functional interfaces must be implemented as external function calls, except `udi_assert`, which must be implemented either as a macro that does nothing or as a macro that calls the following function when the assertion condition test fails:

```
void __udi_assert(const char *expr,
                  const char *filename,
                  int lineno);
```

3.2.2 Building the Driver Object

The UDI Core Specification defines the generic aspects of the packaging and distribution of UDI drivers, whether distributed in source or binary format. The ABI binding defines the object file format and the binding of the static driver properties to the driver's object files.

3.2.2.1 Object file format

The object file format for this IA-64 binding shall be the ELF-64 object file format as defined in the *ELF-64 Object File Format Specification* (http://www.sco.com/developer/gabi/contents.html) and in the *Processor-Specific ELF Supplement for IA-64* (available from Intel Corporation).

The appropriate endian bit indicating the target endianness of the driver module must be set in the ELF-64 file header, as defined in the *ELF-64 Object File Format Specification.*

Any driver package that includes little-endian IA-64 binary modules in a particular alternate version within the package must include little-endian IA-64 binary modules for all modules in that alternate.

Any driver package that includes big-endian IA-64 binary modules in a particular alternate version within the package must include big-endian IA-64 binary modules for all modules in that alternate.

3.2.2.2 Static driver properties encapsulation

A driver's static driver properties shall be encapsulated in the ELF-64 object file of the driver's primary module, in a special no-load section of that object file called, "`.udiprops`". The contents of this section are ASCII-encoded, null-terminated strings from the "`udiprops.txt`" file. The only differences between the contents of the "`udiprops.txt`" file and the "`.udiprops`" section are that:

- Comments are removed.
- Extraneous white space is removed.
- All line terminator characters are removed, and each declaration is appended with a null character ('\0'), so that each declaration is stored as a single, null terminated string.

- Blank lines may be present (i.e. a null character may appear without any preceeding non-null text, even at the beginning and end of the properties section).