Intel® Platform Innovation Framework for EFI CPU I/O Protocol Specification

Draft for Review

Version 0.9

December 3, 2004

Revision History

Revision	Revision History	Date
0.9	First public release.	12/3/04

Contents

Figures

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the CPU I/O Protocol of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). This protocol provides an I/O abstraction for a system processor. This specification does the following:

- Describes the basic components of the CPU I/O interface
- Provides code definitions for the CPU I/O Protocol and the related data types that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

Data Structure Descriptions

Intel® processors based on 32-bit Intel® architecture (IA-32) are "little endian" machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Intel® Itanium® processor family may be configured for both "little endian" and "big endian" operation. All implementations designed to conform to this specification will use "little endian" operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

The data structures described in this document generally have the following format:

STRUCTURE NAME:	The formal name of the data structure.
Summary:	A brief description of the data structure.
Prototype:	A "C-style" type declaration for the data structure.
Parameters:	A brief description of each field in the data structure prototype.
Description:	A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used only by this data structure.

Protocol Descriptions

The protocols described in this document generally have the following format:

Protocol Name:	The formal name of the protocol interface.
Summary:	A brief description of the protocol interface.
GUID:	The 128-bit Globally Unique Identifier (GUID) for the protocol interface.
Protocol Interface Structure:	A "C-style" data structure definition containing the procedures and data fields produced by this protocol interface.
Parameters:	A brief description of each field in the protocol interface structure.
Description:	A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used in the protocol interface structure or any of its procedures.

Procedure Descriptions

The procedures described in this document generally have the following format:

ProcedureName():	The formal name of the procedure.
Summary:	A brief description of the procedure.
Prototype:	A "C-style" procedure header defining the calling sequence.
Parameters:	A brief description of each field in the procedure prototype.
Description:	A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used only by this procedure.
Status Codes Returned:	A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *Extensible Firmware Interface Specification*.

Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.
Bold	In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
BOLD Monospace	Computer code, example code segments, and all prototype code segments use a **BOLD Monospace** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
Bold Monospace	In the online help version of this specification, words in a **Bold Monospace** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **Bold Monospace** appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.
Italic Monospace	In code or in text, words in *Italic Monospace* indicate placeholder names for variable information that must be supplied (i.e., arguments).
Plain Monospace	In code, words in a Plain Monospace typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

text text text	In the PDF of this specification, text that is highlighted in yellow indicates that a change was made to that text since the previous revision of the PDF. The highlighting indicates only that a change was made since the previous version; it does not specify what changed. If text was deleted and thus cannot be highlighted, a note in red and highlighted in yellow (that looks like *(Note: text text text.)*) appears where the deletion occurred.

See the master Framework glossary in the Framework Interoperability and Component Specifications help system for definitions of terms and abbreviations that are used in this document or that might be useful in understanding the descriptions presented in this document.

See the master Framework references in the Interoperability and Component Specifications help system for a complete list of the additional documents and specifications that are required or suggested for interpreting the information presented in this document.

The Framework Interoperability and Component Specifications help system is available at the following URL:

http://www.intel.com/technology/framework/spec.htm

2 Design Discussion

CPU I/O Protocol Introduction

This document describes the CPU I/O Protocol. This protocol provides an I/O abstraction for a system processor. This protocol is used by a PCI root bridge I/O driver or EFI Runtime drivers to perform memory-mapped I/O and I/O transactions. The I/O or memory primitives can be used by the consumer of the protocol to materialize bus-specific configuration cycles, such as the transitional configuration address and data ports for PCI. Only drivers that require direct access to the entire system should use this protocol.

CPU I/O Protocol Terms

The following are the terms that are used throughout this document to describe the CPU I/O Protocol.

coherency domain

The address resources of a system as seen by a processor. It consists of both system memory and I/O space.

CPU I/O Protocol

A software abstraction that provides access to the I/O and memory regions in a single coherency domain.

SMP

Symmetric multiprocessing. A collection of processors that share a common view of I/O and memory-mapped I/O.

CPU I/O Protocol Related Information

The following publications and sources of information may be useful to you or are referred to by this specification.

Intel Specifications

See Related Information from Intel in the Framework master help system for the URLs for the specifications listed below.

- *EFI 1.10 Specification,* specifically the following chapters or subsections:
- EFI Driver Model
- PCI Root Bridge I/O Protocol
- *Itanium® Processor Family System Abstraction Layer Specification*

Industry Specifications

See Industry Specifications in the Framework master help system for the URLs for the specifications listed below.

- *PCI Local Bus Specification*, Revision 2.2, PCI Special Interest Group
- *Advanced Configuration and Power Interface Specification*

CPU I/O Protocol

This section describes the CPU I/O Protocol. This protocol is used by code—typically PCI root bridge I/O drivers and drivers that need I/O prior to the loading of the PCI root bridge I/O driver—that is running in the EFI Boot Services environment to access memory and I/O. It is also used by EFI Runtime drivers to access their resources. Finally, this protocol can be used by non-PC-AT* systems to abstract the I/O mechanism published by the processor and/or integrated CPU-I/O complex.

See Code Definitions for the definition of **`EFI_CPU_IO_PROTOCOL`**.

EFI CPU I/O Overview

The interfaces that are provided in the **`EFI_CPU_IO_PROTOCOL`** are for performing basic operations to memory and I/O. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The **`EFI_CPU_IO_PROTOCOL`** allows for future innovation of the platform. It abstracts processor-device-specific code from the system memory map. This abstraction allows system designers to make changes to the system memory map without impacting platform-independent code that is consuming basic system resources.

Systems with one to many processors in a symmetric multiprocessing (SMP) configuration will contain a single instance of the **`EFI_CPU_IO_PROTOCOL`**. This protocol is an abstraction from a software point of view. This protocol is attached to the device handle of a processor driver. The CPU I/O Protocol is the parent to a set of PCI Root Bridge I/O Protocol instances that may contain many PCI segments. A CPU I/O Protocol instance might also be the parent of a series of protocols that abstract host-bus attached devices.

CPU I/O Protocol instances are either produced by the system firmware or an EFI driver. When a CPU I/O Protocol is produced, it is placed on a device handle without an EFI Device Path Protocol instance. The figure below shows a device handle that has the **`EFI_CPU_IO_PROTOCOL`** installed on it.

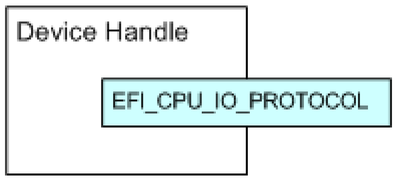

Figure 2-1. EFI CPU I/O Protocol

Other characteristics of the CPU I/O Protocol include the following:

- The protocol uses re-entrancy to enable possible use by a debugger agent that is outside of the generic EFI Task Priority Level (TPL) priority mechanism.
- The protocol will maintain the physical relocations and pointers such that, after a call to **`SetVirtualAddress()`**, the service can be called back in physical mode. This physical-mode usage after a call to **`SetVirtualAddress()`** includes if the EFI firmware needs to perform some I/O transactions as part of the **`ResetSystem()`** runtime implementation. This point has been known as the Afterlife phase, or a post-operating-system execution phase.

See Code Definitions for the definition of **`EFI_CPU_IO_PROTOCOL`**.

3 Code Definitions

Introduction

This section contains the basic definitions of the CPU I/O Protocol. The following protocol is defined in this section:

- `EFI_CPU_IO_PROTOCOL`

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent protocol or function definition:

- `EFI_CPU_IO_PROTOCOL_ACCESS`
- `EFI_CPU_IO_PROTOCOL_WIDTH`

CPU I/O Protocol

EFI_CPU_IO_PROTOCOL

Summary

Provides the basic memory and I/O interfaces that are used to abstract accesses to devices in a system.

GUID

```
#define EFI_CPU_IO_PROTOCOL_GUID  \
  {0xB0732526, 0x38C8, 0x4b40, 0x88, 0x77, 0x61, 0xC7, 0xB0,
0x6A, 0xAC, 0x45}
```

Protocol Interface Structure

```
typedef struct _EFI_CPU_IO_PROTOCOL {
  EFI_CPU_IO_PROTOCOL_ACCESS          Mem;
  EFI_CPU_IO_PROTOCOL_ACCESS          Io;
} EFI_CPU_IO_PROTOCOL;
```

Parameters

Mem.Read

Allows reads from memory-mapped I/O space. See the `Mem.Read()` function description. Type `EFI_CPU_IO_PROTOCOL_ACCESS` is defined in "Related Definitions" below.

Mem.Write

Allows writes to memory-mapped I/O space. See the `Mem.Write()` function description.

Io.Read

Allows reads from I/O space. See the **Io.Read()** function description. Type **EFI_CPU_IO_PROTOCOL_ACCESS** is defined in "Related Definitions" below.

Io.Write

Allows writes to I/O space. See the **Io.Write()** function description.

Description

The **EFI_CPU_IO_PROTOCOL** provides the basic memory and I/O interfaces that are used to abstract accesses to platform hardware. This hardware can include PCI- or host-bus-attached peripherals and buses. There is one **EFI_CPU_IO_PROTOCOL** instance for each EFI system. Embedded systems, desktops, and workstations will typically have only one EFI system. Non–symmetric multiprocessing (SMP), high-end servers may have multiple EFI systems. A device driver that wishes to make I/O transactions in a system will have to retrieve the **EFI_CPU_IO_PROTOCOL** instance. A device handle for an EFI system will minimally contain an **EFI_CPU_IO_PROTOCOL** instance.

Related Definitions

```
//*******************************************************
// EFI_CPU_IO_PROTOCOL_ACCESS
//*******************************************************
typedef struct {
  EFI_CPU_IO_PROTOCOL_IO_MEM  Read;
  EFI_CPU_IO_PROTOCOL_IO_MEM  Write;
} EFI_CPU_IO_PROTOCOL_ACCESS;
```

Read

This service provides the various modalities of memory and I/O read.

Write

This service provides the various modalities of memory and I/O write.

EFI_CPU_IO_PROTOCOL.Mem.Read() and Mem.Write()

Summary

Enables a driver to access memory-mapped registers in the EFI system memory space.

Prototype

```
typedef
EFI_STATUS
EFI_RUNTIMESERVICE
(EFIAPI *EFI_CPU_IO_PROTOCOL_IO_MEM) (
  IN      EFI_CPU_IO_PROTOCOL              *This,
  IN      EFI_CPU_IO_PROTOCOL_WIDTH        Width,
  IN      UINT64                           Address,
  IN      UINTN                            Count,
  IN OUT  VOID                             *Buffer
  );
```

Parameters

This

A pointer to the **EFI_CPU_IO_PROTOCOL** instance.

Width

Signifies the width of the memory operation. Type **EFI_CPU_IO_PROTOCOL_WIDTH** is defined in "Related Definitions" below.

Address

The base address of the memory operation. The caller is responsible for aligning the *Address* if required.

Count

The number of memory operations to perform. The number of bytes moved is *Width* size * *Count*, starting at *Address*.

Buffer

For read operations, the destination buffer to store the results. For write operations, the source buffer from which to write data.

Description

The **Mem.Read()** and **Mem.Write()** functions enable a driver to access memory-mapped registers in the EFI system memory space.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that an EFI system on a platform might require. For example, on some platforms, width requests of **EfiCpuIoWidthUint64** do not work. Misaligned buffers, on the other hand, will be handled by the driver.

If *Width* is **EfiCpuIoWidthUint8**, **EfiCpuIoWidthUint16**, **EfiCpuIoWidthUint32**, or **EfiCpuIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations that is performed.

If *Width* is **EfiCpuIoWidthFifoUint8**, **EfiCpuIoWidthFifoUint16**, **EfiCpuIoWidthFifoUint32**, or **EfiCpuIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiCpuIoWidthFillUint8**, **EfiCpuIoWidthFillUint16**, **EfiCpuIoWidthFillUint32**, or **EfiCpuIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

Related Definitions

```
//*******************************************************
// EFI_CPU_IO_PROTOCOL_WIDTH
//*******************************************************
typedef enum {
  EfiCpuIoWidthUint8,
  EfiCpuIoWidthUint16,
  EfiCpuIoWidthUint32,
  EfiCpuIoWidthUint64,
  EfiCpuIoWidthFifoUint8,
  EfiCpuIoWidthFifoUint16,
  EfiCpuIoWidthFifoUint32,
  EfiCpuIoWidthFifoUint64,
  EfiCpuIoWidthFillUint8,
  EfiCpuIoWidthFillUint16,
  EfiCpuIoWidthFillUint32,
  EfiCpuIoWidthFillUint64,
  EfiCpuIoWidthMaximum
} EFI_CPU_IO_PROTOCOL_WIDTH;
```

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the EFI system.
EFI_INVALID_PARAMETER	*Width* is invalid for this EFI system.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	The *Buffer* is not aligned for the given *Width*.
EFI_UNSUPPORTED	The address range specified by *Address*, *Width*, and *Count* is not valid for this EFI system.

EFI_CPU_IO_PROTOCOL.Io.Read() and Io.Write()

Summary

Enables a driver to access registers in the EFI CPU I/O space.

Prototype

```
typedef
EFI_STATUS
EFI_RUNTIMESERVICE
(EFIAPI *EFI_CPU_IO_PROTOCOL_IO_MEM) (
  IN     EFI_CPU_IO_PROTOCOL                    *This,
  IN     EFI_CPU_IO_PROTOCOL_WIDTH              Width,
  IN     UINT64                                 Address,
  IN     UINTN                                  Count,
  IN OUT VOID                                   *Buffer
  );
```

Parameters

This

A pointer to the **EFI_CPU_IO_PROTOCOL** instance.

Width

Signifies the width of the I/O operation. Type **EFI_CPU_IO_PROTOCOL_WIDTH** is defined in **EFI_CPU_IO_PROTOCOL.Mem()**.

Address

The base address of the I/O operation. The caller is responsible for aligning the *Address* if required.

Count

The number of I/O operations to perform. The number of bytes moved is *Width* size * *Count*, starting at *Address*.

Buffer

For read operations, the destination buffer to store the results. For write operations, the source buffer from which to write data.

Description

The **Io.Read()** and **Io.Write()** functions enable a driver to access PCI controller registers in the EFI CPU I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and I/O width restrictions that an EFI system on a platform might require. For example on some platforms, width requests of **EfiCpuIoWidthUint64** do not work. Misaligned buffers, on the other hand, will be handled by the driver.

If *Width* is **EfiCpuIoWidthUint8**, **EfiCpuIoWidthUint16**, **EfiCpuIoWidthUint32**, or **EfiCpuIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations that is performed.

If *Width* is **EfiCpuIoWidthFifoUint8**, **EfiCpuIoWidthFifoUint16**, **EfiCpuIoWidthFifoUint32**, or **EfiCpuIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiCpuIoWidthFillUint8**, **EfiCpuIoWidthFillUint16**, **EfiCpuIoWidthFillUint32**, or **EfiCpuIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the EFI system.
EFI_INVALID_PARAMETER	*Width* is invalid for this EFI system.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	The *Buffer* is not aligned for the given *Width*.
EFI_UNSUPPORTED	The address range specified by *Address*, *Width*, and *Count* is not valid for this EFI system.