Intel® Platform Innovation Framework for EFI Data Hub Specification

Draft for Review

Version 0.9

September 16, 2003

Intel order number xxxxxx-001

Revision History

Revision	Revision History	Date
0.9	First public release.	9/16/03

Contents

Figures

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the data hub in the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). The data hub is a volatile database that is intended as the major focus for the accumulation of manageability data. This specification does the following:

- Describes the basic components and the usage models of the Data Hub Protocol
- Defines the structure of the data record header and the high-level classes of data records
- Provides code definitions for the Data Hub Protocol and its member functions that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

Data hub subclasses are outside the scope of this document and are defined in other specifications.

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.

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

Data Hub

The data hub is a volatile database that is intended as the major focus for the accumulation of manageability data. The hub is fed by "producers" with chunks of data in a defined format. Consumers may then extract the data in temporal "log" order. As an example, progress codes might be recorded in the data hub for future processing. Other data contributed to the data hub might include, for example, statistics on enumerated items such as memory, add-in buses, and add-in cards and data on errors encountered during boot (for example, the system did not boot off the network because the cable was not plugged in).

Some classes of data have defined formats. For example, the amount of memory in the system is reported in a standard format so that consumers can be written to extract the data. Other data is system specific. For example, additional detail on errors might be specific to the driver that discovered the error. The consumer might be a driver that tabularizes data from the data hub, providing a mechanism for the raw data to be made available to the OS for post-processing by OS-based applications.

The intent of the data hub is for drivers that enumerate and configure parts of the system to report their discoveries to the data hub. This data can then be extracted by other drivers that report those discoveries using standard manageability interfaces such as SMBIOS and Intelligent Platform Management Interface (IPMI). The alternative to a data-hub-like architecture is to require all drivers to be aware of all reporting formats.

Data Hub Protocol

Data Hub Protocol Overview

The **EFI_DATA_HUB_PROTOCOL** defines an abstract memory-based data journal. The protocol can be used for the following:

- To log data
- To recover data that has been logged to the protocol

The memory-based log only persists for the duration of the boot. The **EFI_DATA_HUB_PROTOCOL** also supports the registration of filter driver event handlers that will be signaled every time data is logged. Optionally, an event handler can opt to get signaled only for data classes in which it is interested.

The **EFI_DATA_HUB_PROTOCOL** is well suited to logging errors. Because all data entries are logged to memory, this protocol emulates the basic function of an error log. A filter driver can be added that will save the error log entries to a nonvolatile store. The power of the **EFI_DATA_HUB_PROTOCOL** is that the filter driver can be loaded at any time in the boot process and still have access to all the errors that were logged.

It is also possible to use the **EFI_DATA_HUB_PROTOCOL** for other purposes, such as the following:

- Registering data
- Collecting debug information

In general, any problem that requires production of data over an extended period of the boot process and the consumption of data at some later time lends itself to the data hub.

The global definition of data includes all the classes and should not be mistaken with the data class.

The figure below shows a high-level overview of the Data Hub Protocol

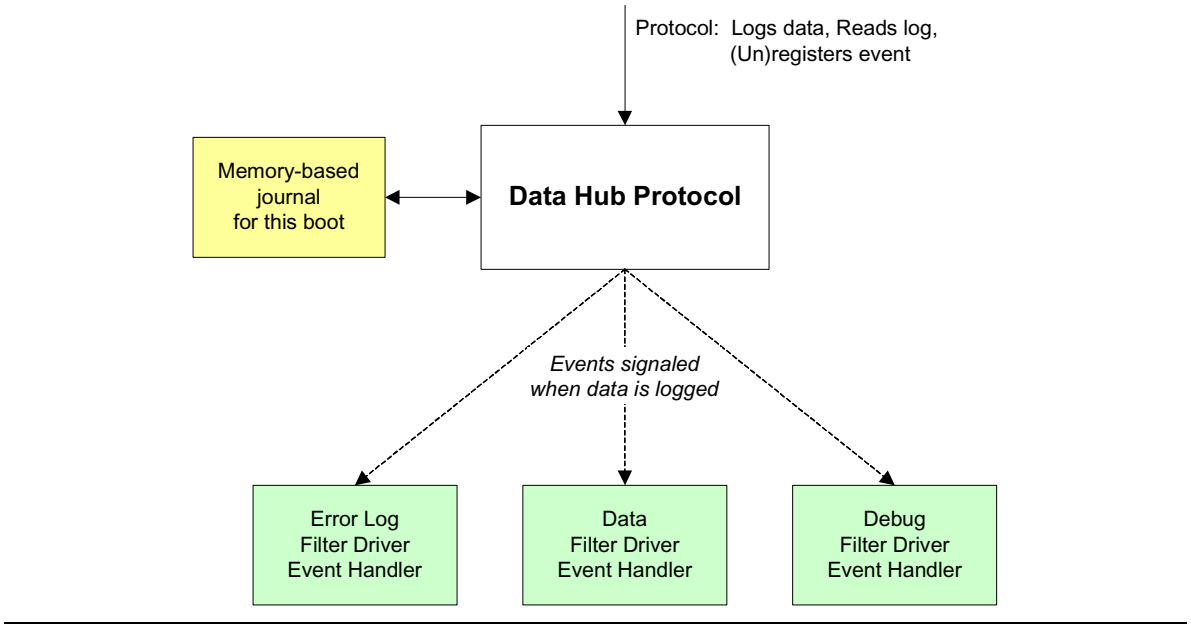

Figure 2-1. Data Hub Protocol Overview

Usage Models

The architecture allows many possible usage models. How you use it is up to you and your needs.

Logging should always be done using this protocol when possible. Use *DataRecord.DataRecordGuid* to allow for the addition of new data log types. Different information can be logged using this framework, including System Management BIOS (SMBIOS) error records or structures, Intelligent Platform Management Interface (IPMI) error records, POST codes, and debug information.

To log to nonvolatile RAM (NVRAM) error logs, an error-logging *FilterEvent* will be required to abstract the specific logging rules.

NOTE

Remember that **EFI_DATA_HUB_PROTOCOL** *only exists in the Boot Services time and cannot be used to log errors from Runtime.*

When defining a new *DataRecord.DataRecordGuid*, it is important to consider what level of abstraction is required. The easy answer may be to pick the error log structure of the standard you are trying to follow. While this approach will work, it will likely have a detrimental impact on the logging code. Does every subsystem that logs errors now need to be changed to support the new error-logging scheme? Making the logging code log format independent is typically a better answer. Another alternative is to consider whether you can convert from a currently supported log type to the type you need to log.

3
Code Definitions

Introduction

This section contains the basic definitions of the data record header and the Data Hub Protocol. The following protocols, functions, and data types are defined in this section:

- **EFI_DATA_RECORD_HEADER** and the definitions for *DataRecordClass*, which are used to filter data types at a very high level
- **EFI_DATA_HUB_PROTOCOL**

Data Record Header

EFI_DATA_RECORD_HEADER

Summary

The standard header that appears at the start of each data record that is logged or read.

Prototype

```
typedef struct {
  UINT16      Version;
  UINT16      HeaderSize;
  UINT32      RecordSize;
  EFI_GUID    DataRecordGuid;
  EFI_GUID    ProducerName;
  UINT64      DataRecordClass;
  EFI_TIME    LogTime;
  UINT64      LogMonotonicCount;
} EFI_DATA_RECORD_HEADER;
```

Parameters

Version

The version of the header. This specification defines the value as 0x0100; see "Related Definitions" below.

HeaderSize

Size of the header in bytes.

RecordSize

Size of the data in the record in bytes.

DataRecordGuid

A GUID that defines the semantic contents of the data that follows the header. This specification does not define specific *DataRecordGuid* types. Because the *DataRecordGuid* is a GUID, there is no need for a centralized allocation of *DataRecordGuid* values. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

ProducerName

A GUID that identifies the component that produced this header and its associated data. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DataRecordClass

Used to tag the general class of records being logged. See "Related Definitions" below.

LogTime

Represents the time the data was logged. If time services are not available at the time the data is registered, this field will be set to all zeros. Type **EFI_TIME** is defined in **GetTime()** in the *EFI 1.10 Specification.*

LogMonotonicCount

Used to uniquely identify each data record inside the data hub.

Description

Each data record that is logged or read starts with a standard header of type **EFI_DATA_RECORD_HEADER**.

The *DataRecord.DataRecordClass* is used to tag the general class of records being logged. The class can be used to filter out a *DataRecord.DataRecordGuid* that is unknown to a consumer. The class is high-level information such as whether this record is a debug, error, or data record. All possible values of *DataRecordClass* are defined or reserved by this specification; see "Related Definitions" below.

Each data record header contains a *LogMonotonicCount* that is guaranteed to be unique for the duration of a boot. A monotonic count is simply a value that is guaranteed to increase over time. Thus the *LogMonotonicCount* is used to uniquely identify each data record inside the data hub.

Related Definitions

```
//*******************************************************
// Version value
//*******************************************************
#define EFI_DATA_RECORD_HEADER_VERSION    0x0100


//*******************************************************
// DataRecordClass values
//*******************************************************
//
//  Definition of DataRecordClass. These are used to filter
//  out data types at a very high level. The
//  DataRecord.DataRecordGuid still defines the format
//  of the data.
//

#define EFI_DATA_CLASS_DEBUG         0x0000000000000001
#define EFI_DATA_CLASS_ERROR         0x0000000000000002
#define EFI_DATA_CLASS_DATA          0x0000000000000004
#define EFI_DATA_CLASS_PROGRESS_CODE 0x0000000000000008
```

Following is a description of the fields in the above definition.

EFI_DATA_CLASS_DEBUG	This class is used to signify debug information. It is not intended to be logged to error logs and it does not contain data for normal system operation.
EFI_DATA_CLASS_ERROR	This class is used to signify error information. This information is destined for nonvolatile error logs. It does not contain data needed for the normal operation of the system.
EFI_DATA_CLASS_DATA	This class is used to signify data that can be used to boot the system or for informational purposes. It is not intended to be logged to nonvolatile error logs.
EFI_DATA_CLASS_PROGRESS_CODE	This class is used to signify data that was logged via the `ReportStatusCode()` API. See the DXE CIS for the definition.

Data Hub Protocol

EFI_DATA_HUB_PROTOCOL

Summary

This protocol is used to log information and register filter drivers to receive data records.

GUID

```
#define EFI_DATA_HUB_PROTOCOL_GUID \
  { 0xae80d021, 0x618e, 0x11d4, 0xbc, 0xd7, 0x0, 0x80, 0xc7,
    0x3c, 0x88, 0x81 }
```

Protocol Interface Structure

```
typedef struct _EFI_DATA_HUB_PROTOCOL {
  EFI_DATA_HUB_LOG_DATA                 LogData;
  EFI_DATA_HUB_GET_NEXT_DATA_RECORD     GetNextDataRecord;
  EFI_DATA_HUB_REGISTER_DATA_FILTER_DRIVER
                                        RegisterFilterDriver;
  EFI_DATA_HUB_UNREGISTER_DATA_FILTER_DRIVER
                                        UnregisterFilterDriver;
} EFI_DATA_HUB_PROTOCOL;
```

Parameters

LogData

Logs a data record. See the **LogData()** function description.

GetNextDataRecord

Gets a data record. Used both to view the memory-based log and to get information about which data records have been consumed by a filter driver. See the **GetNextDataRecord()** function description.

RegisterFilterDriver

Allows the registration of an EFI event to act as a filter driver for all data records that are logged. See the **RegisterFilterDriver()** function description.

UnregisterFilterDriver

Used to remove a filter driver that was added with **RegisterFilterDriver()**. See the **UnregisterFilterDriver()** function description.

Description

The **EFI_DATA_HUB_PROTOCOL** is used by any agent in the system that wishes to log data or to be notified whenever something is being logged on the system.

EFI_DATA_HUB_PROTOCOL.LogData()

Summary

Logs a data record to the system event log.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DATA_HUB_LOG_DATA) (
  IN EFI_DATA_HUB_PROTOCOL          *This,
  IN EFI_GUID                       *DataRecordGuid,
  IN EFI_GUID                       *ProducerName,
  IN UINT64                         DataRecordClass,
  IN VOID                           *RawData,
  IN UINT32                         RawDataSize
  );
```

Parameters

This

The **EFI_DATA_HUB_PROTOCOL** instance.

DataRecordGuid

A GUID that indicates the format of the data passed into *RawData*. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

ProducerName

A GUID that indicates the identity of the caller to this API. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

DataRecordClass

This class indicates the generic type of the data record. This generic nature enables filtering without having to know every possible *DataRecordGuid*. See "Related Definitions" in **EFI_DATA_RECORD_HEADER** for defined high-level classes of records.

RawData

The *DataRecordGuid*-defined data to be logged.

RawDataSize

The size in bytes of *RawData*.

Description

This function allows any agent to log data. This member function takes the input arguments (*DataRecordGuid*, *ProducerName*, *DataRecordClass*, *RawData*, and *RawDataSize*) and creates an **EFI_DATA_RECORD_HEADER** followed by the record-specific data. **LogData()** is responsible for adding *Version*, *HeaderSize*, *LogTime*, and *LogMonotonicCount* to the **EFI_DATA_RECORD_HEADER** and inserting it into the memory-based log.

All currently registered filter driver events are signaled after the data is logged.

Status Codes Returned

EFI_SUCCESS	Data was logged.
EFI_OUT_OF_RESOURCES	Data was not logged due to lack of system resources.

EFI_DATA_HUB_PROTOCOL.GetNextDataRecord()

Summary

Allows the system data log to be searched.

Prototype

```
EFI_STATUS
(EFIAPI * EFI_DATA_HUB_GET_NEXT_DATA_RECORD) (
  IN EFI_DATA_HUB_PROTOCOL              *This,
  IN OUT  UINT64                        *MonotonicCount,
  IN  EFI_EVENT                         *FilterDriver,  OPTIONAL
  OUT EFI_DATA_RECORD_HEADER            **Record
  );
```

Parameters

This

The **EFI_DATA_HUB_PROTOCOL** instance.

MonotonicCount

On input, it specifies the *Record* to return. An input of zero means to return the first record.

FilterDriver

If *FilterDriver* is not passed in a *MonotonicCount* of zero, it means to return the first data record. If *FilterDriver* is passed in, then a *MonotonicCount* of zero means to return the first data not yet read by *FilterDriver*. Type **EFI_EVENT** is defined in **CreateEvent()** in the *EFI 1.10 Specification.*

Record

Returns a dynamically allocated memory buffer with a data record that matches *MonotonicCount*.

Description

This function gets a data record. It is used both to view the memory-based log and to get information about which data records have been consumed by a filter driver.

An **EFI_DATA_RECORD_HEADER** is returned that matches the *MonotonicCount*. *MonotonicCount* also returns the value for the next *MonotonicCount* or zero if no more data records exist. If the *MonotonicCount* is nonzero, the data record that matches *MonotonicCount* will be returned regardless of *FilterDriver* .

If *FilterDriver* is **NULL** and the *MonotonicCount* is zero on input, then the first data record in the memory log is returned. On output, *MonotonicCount* will contain the monotonic count of the next data record.

If *FilterDriver* is valid and the *MonotonicCount* is zero on input, then the first data record that has not yet been read by the *FilterDriver* is returned. On output, *MonotonicCount* will contain the monotonic count of the next data record that matches the criteria defined by *FilterDriver*.

Status Codes Returned

EFI_SUCCESS	Data was returned in *Record*.
EFI_INVALID_PARAMETER	*FilterDriver* was passed in but does not exist.
EFI_NOT_FOUND	*MonotonicCount* does not match any data record in the system. If a *MonotonicCount* of zero was passed in, then no data records exist in the system.
EFI_OUT_OF_RESOURCES	*Record* was not returned due to lack of system resources.

EFI_DATA_HUB_PROTOCOL.RegisterFilterDriver()

Summary

Registers an event to be signaled every time a data record is logged in the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_DATA_HUB_REGISTER_DATA_FILTER_DRIVER) (
  IN EFI_DATA_HUB_PROTOCOL              *This,
  IN EFI_EVENT                          FilterEvent,
  IN EFI_TPL                            FilterTpl,
  IN UINT64                             FilterClass
  IN EFI_GUID                           *FilterDataRecordGuid
OPTIONAL
  );
```

Parameters

This

The **EFI_DATA_HUB_PROTOCOL** instance.

FilterEvent

The **EFI_EVENT** to signal whenever data that matches *FilterClass* is logged in the system. Type **EFI_EVENT** is defined in **CreateEvent()** in the *EFI 1.10 Specification*.

FilterTpl

The maximum **EFI_TPL** at which *FilterEvent* can be signaled. It is strongly recommended that you use the lowest **EFI_TPL** possible. Type **EFI_TPL** is defined in **RaiseTPL()** in the *EFI 1.10 Specification*.

FilterClass

FilterEvent will be signaled whenever a bit in **EFI_DATA_RECORD_HEADER.***DataRecordClass* is also set in *FilterClass*. If *FilterClass* is zero, no class-based filtering will be performed.

FilterDataRecordGuid

FilterEvent will be signaled whenever *FilterDataRecordGuid* matches **EFI_DATA_RECORD_HEADER.***DataRecordGuid*. If *FilterDataRecordGuid* is **NULL**, then no GUID-based filtering will be performed.

Description

This function registers the data hub filter driver that is represented by *FilterEvent*. Only one instance of each *FilterEvent* can be registered. After the *FilterEvent* is registered, it will be signaled so it can sync with data records that have been recorded prior to the *FilterEvent* being registered.

FilterClass and *FilterDataRecordGuid* can be optionally used to restrict which events will cause *FilterEvent* to be signaled. *FilterClass* and *FilterDataRecordGuid* have an "**AND**" relationship because each argument must be matched to signal an event.

Status Codes Returned

EFI_SUCCESS	The filter driver event was registered
EFI_ALREADY_STARTED	*FilterEvent* was previously registered and cannot be registered again.
EFI_OUT_OF_RESOURCES	The filter driver event was not registered due to lack of system resources.

EFI_DATA_HUB_PROTOCOL.UnregisterFilterDriver()

Summary

Stops a filter driver from being notified when data records are logged.

Prototype

```
EFI_STATUS
(EFIAPI * EFI_DATA_HUB_REGISTER_DATA_FILTER_DRIVER) (
  IN EFI_DATA_HUB_PROTOCOL              *This,
  IN EFI_EVENT                          FilterEvent
  );
```

Parameters

This

The **EFI_DATA_HUB_PROTOCOL** instance.

FilterEvent

The **EFI_EVENT** to remove from the list of events to be signaled every time errors are logged. Type **EFI_EVENT** is defined in **CreateEvent()** in the *EFI 1.10 Specification.*

Description

This function allows a filter driver to stop being notified when data records are logged. **UnregisterFilterDriver()** can be performed only on a *FilterEvent* that has been previously registered with **RegisterFilterDriver()**.

Status Codes Returned

EFI_SUCCESS	The filter driver represented by *FilterEvent* was shut off.
EFI_NOT_FOUND	*FilterEvent* did not exist.