EFI Application Toolkit MP Protocol Specification

Revision 0.9

May, 2000

ESG Server Software Technologies (SST)

Revision History

Date	Revision	Modifications
12/99	0.1	Original draft
02/00	0.3	Architectural change from library to protocol Document title updated
03/00	0.7	Updated based on actual implementation
05/00	0.9	Removed restriction on global variables

Disclaimers

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

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The EFI Application Toolkit MP Protocol Specification may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Currently characterized errata are available on request.

*Third-party brands and names are the property of their respective owners.

Table Of Contents

Table Of Figures

This page intentionally left blank

1 Introduction

This document is the specification for the EFI Application Toolkit (EFI-AT) Multiprocessor Protocol (MPP) Specification.

1.1 Scope

This specification defines the content and features of the MPP included in the EFI-AT.

1.2 Target Audience

This specification is for individuals who wish to understand the functionality and implementation details of MPP.

1.3 Reference Documents

The following documents were useful in preparing this specification:

- *Extensible Firmware Interface Specification*. Version 0.99, April 19, 2000.
- *EFI Developer's Guide*. Version 0.2, July 14, 1999.
- *Extensible Firmware Interface Library Specification.* Version 0.2, July 14, 1999.
- *EFI Application Toolkit Product Requirements Document*. Revision 0.97, September 27, 1999.
- *IA-64 System Abstraction Layer (SAL) Specification.* Version 2.6.
- *Advanced Configuration and Power Interface Specification*. Revision 1.0b. Februrary 2, 1999.
- *IA-64 Extensions to the Advanced Configuration and Power Interface Specification.* Revision 0.7, June 14, 1999.

1.4 Product Overview

MPP provides a set of application programming interfaces (APIs) that enable EFI applications to make use of the multiprocessing capabilities of the platform. MPP support is limited to IA64

3 MPP – Architecture

3.1 Overview

The MPP architecture is designed to support development of EFI applications that can be partitioned programmatically onto multiple processors in a shared memory environment. The following sections describe the MPP architecture in detail.

3.2 Architecture Components

The components of the MPP architecture are depicted in Figure 1 - Architecture Components.

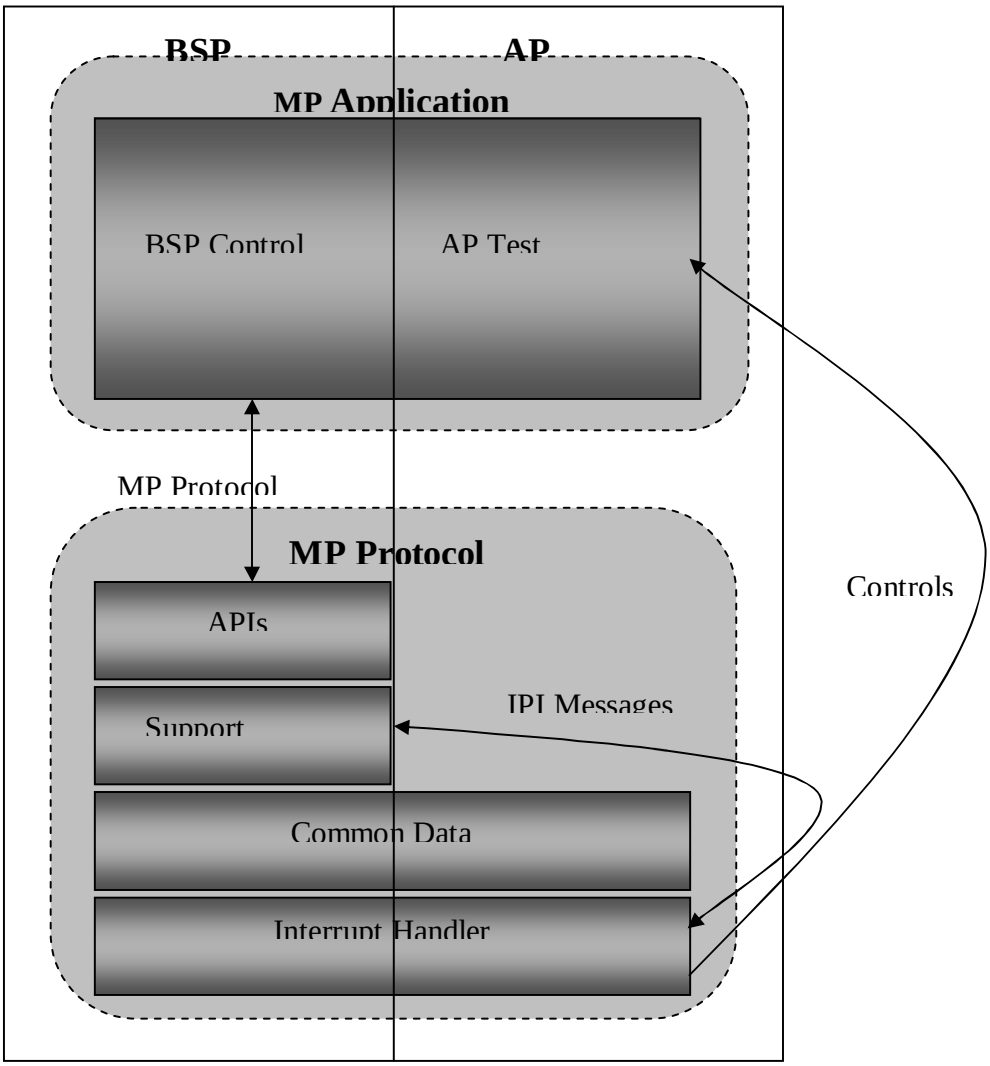

Figure 1 - Architecture Components

In the MPP architecture, the boot processor (BSP) is responsible for the overall control of the MP application. The BSP is responsible for distributing functions, starting functions, and stopping

functions on each of the non-boot processors (APs). The BSP uses a set of APIs from the MPP to facilitate these actions. The APs perform tasks assigned by the BSP. APs cannot distribute, start, or stop functions.

3.3 Application Programming Interface

The MPP provides a set of programming interfaces that are available to the BSP. APIs are included to initialize/deinitialize the protocol, register a function for execution on an AP, and start/stop a function on an AP. The programming interfaces are not available to APs and are described in detail in the section MPP - Programming Interfaces.

3.4 Support Component

The APIs included in the MPP interact with a support component. The support component maintains the internal data structures within the protocol and provides a mechanism to send IPI messages to other processors. Additionally, the support component includes the initialization sequence for the protocol. The initialization sequence is described further in the following paragraphs.

After the protocol is loaded into memory, control is transferred to the entry point, which contains the initialization sequence for the protocol. On success, the protocol is left to reside in memory to provide services for other EFI applications. On failure, the driver is removed from memory and is not available to provide any services to other EFI applications.

The initialization sequence performs the following operations:

- Initializes the EFI and C libraries linked to the MPP
- Enumerates the processors in the system
- Wakes the APs from the SAL boot rendezvous state to an idle loop
- Establishes internal data structures to enable communication between the BSP and the APs
- Chains an interrupt handler to the external interrupt vector shared between the BSP and the APs to enable handling of inter-processor interrupts (IPIs) sent from the BSP to the APs

See Appendix B: Protocol Initialization Return Codes for a list of error codes that may be returned during protocol initialization.

3.5 Common Data Component

Several data structures within the protocol are shared between the BSP and the APs. The shared data structures include a message array, an address array, a global pointer array, and an argument array. Each array is allocated during the initialization sequence in the support component. The length of each array is determined by the number of processors in the system. Arrays are indexed by the processor local identifiers.

The BSP uses the message array to pass start and stop messages to the APs. APs use the message array to send completion messages to the BSP. The message array provides a very simple

communication mechanism between the BSP and the APs. With the message array, the BSP initiates communication to an AP by first writing a message into the message array, then sends an IPI to the AP to retrieve the message. The BSP then blocks until the AP has processed the message. The AP signals completion to the BSP by writing a completion message into the message array.

The address array is used to maintain the address of the function where an AP should begin execution.

The global pointer array is used to maintain the relevant global pointer for the function where an AP should begin execution.

The argument array is used to maintain the argument to be passed to a function to begin execution on AP.

3.6 Interrupt Handler Component

The interrupt handler is installed during the initialization sequence found in the support component and is shared between the BSP and the APs. The handler is chained immediately before the default external interrupt handler provided by the system and is used to capture and process IPI messages sent from the BSP to the APs. Processing of IPI messages includes both starting and stopping a function on an AP. The interrupt handler includes support for context save and restoration so that APs can be returned to an idle loop after a stop message has been processed.

4 MPP - Programming Interfaces

The following sections describe the programming interfaces provided by MPP.

4.1 InitMpProtocol()

The **InitMpProtocol()** function initializes the MPP.

#include <atk_mp.h>

EFI_STATUS
InitMpProtocol (

IN **EFI_MP_INTERFACE** **This*

) ;

Parameters

This The EFI_MP_INTERFACE instance.

Description

An EFI application must call **InitMpProtocol()** to initialize the protocol prior to using any other function provided.

After completion of a successful call to **InitMpProtocol()**, successive calls to this function will fail until **DeInitMpProtocol()** is called.

Status Codes Returned

EFI_SUCCESS	Protocol was successfully initialized
EFI_MP_FAILURE	Protocol was not successfully initialized
EFI_INVALID_PARAMETER	Invalid parameter passed to function

4.2 DeInitMpProtocol()

The **DeInitMpProtocol()** function deinitializes the MPP.

#include <atk_mp.h>

EFI_STATUS
DeInitMpProtocol (

IN	**EFI_MP_INTERFACE**	**This*

) ;

Parameters

This	The EFI_MP_INTERFACE instance.

Description

An EFI application must call **DeInitMpProtocol()** to deinitialize the protocol.

Prior to calling **DeInitMpProtocol()** the MPP must be initialized using the function **InitMpProtocol()**.

Status Codes Returned

EFI_SUCCESS	Protocol was successfully deinitialized
EFI_MP_FAILURE	Protocol was not successfully deinitialized
EFI_INVALID_PARAMETER	Invalid parameter passed to function

4.3 GetNumEnabledProcessors()

The **GetNumEnabledProcessors()** function returns the number of enabled processors on the platform.

#include <atk_mp.h>

EFI_STATUS
GetNumEnabledProcessors (

IN	**EFI_MP_INTERFACE**	**This,*
OUT	**UINT8**	**NumProcessors*

) ;

Parameters

This	The EFI_MP_INTERFACE instance.
NumProcessors	The address to return the number of enabled processors on the platform.

Description

GetNumEnabledProcessors() returns the number of enabled processors in the parameter *NumProcessors*.

On an IA64 platform, the number of processors is determined by the Processor Local SAPIC data structures maintained in the platform ACPI table. If the ACPI table does not contain any Processor Local SAPIC data structures or if the ACPI table is not valid, the call will return an error code and *NumProcessors* will be set to zero.

Prior to calling **GetNumEnabledProcessors()** the MPP must be initialized using the function **InitMpProtocol()**.

Status Codes Returned

EFI_SUCCESS	Number of enabled processors successfully returned
EFI_MP_FAILURE	Number of enabled processors not successfully returned.
EFI_INVALID_PARAMETER	Invalid parameter passed to function

4.4 GetEnabledProcessorsInfo()

The **GetEnabledProcessorsInfo()** function returns information about the enabled processors on the platform.

#include <atk_mp.h>

EFI_STATUS
GetEnabledProcessorsInfo (

IN	**EFI_MP_INTERFACE**	**This,*
IN OUT	**EFI_MP_PROC_INFO**	**Buffer,*
IN OUT	**UINTN**	*BufferSize*

) ;

typedef struct {
UINT16 *ACPIProcessorID ;*
UINT32 *Flags ;*
} EFI_MP_PROC_INFO ;

Parameters

This	The EFI_MP_INTERFACE instance.
Buffer	A pointer to the data buffer to return.
BufferSize	Size of *Buffer* in EFI_MP_PROC_INFO records.

Description

The function fills in *Buffer* with the processor information for each enabled processor on the platform. The information is returned in a series of *EFI_MP_PROC_INFO* structures.

Prior to calling **GetEnabledProcessorInfo()**, *Buffer* should be allocated large enough to hold an *EFI_MP_PROC_INFO* structure for each enabled processor. If *Buffer* is not large enough, *BufferSize* is updated to the number of *EFI_MP_PROC_INFO* structures required to return processor information for each enabled processor on the platform. If *Buffer* is large enough, *BufferSize* is updated to the number of *EFI_MP_PROC_INFO* structures returned.

The number of enabled processors can be determined using this function with a *BufferSize* of zero.

On an IA64 platform, the number of processors is determined by the Processor Local SAPIC data structures maintained in the platform ACPI table. If the ACPI table does not contain any Processor Local SAPIC data structures or if the ACPI table is not valid, the call will return an error code and *BufferSize* will be set to zero.

Prior to calling **GetEnabledProcessorsInfo()** the MPP must be initialized using the function **InitMpProtocol()**.

The *Flags* field of the *EFI_MP_PROC_INFO* structure has the following format:

Bits	Name	Description
31:1	Reserved	Reserved for future use
0	Processor Type	0 – Processor is BSP 1 – Processor is an AP

Status Codes Returned

EFI_SUCCESS	Enabled processor information successfully returned
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small to read the enabled processor information for the platform.
EFI_MP_FAILURE	Enabled processor information not successfully returned.
EFI_INVALID_PARAMETER	Invalid parameter passed to function

4.5 StartProcessor()

The **StartProcessor()** function starts execution of an AP at the address previously specified by **StartProcessorAddress()**.

#include <atk_mp.h>

EFI_STATUS
StartProcessor (
IN **EFI_MP_INTERFACE** **This,*
IN **UINT16** *ACPIProcessorID*
) ;

Parameters

This	The EFI_MP_INTERFACE instance.
ACPIProcessorID	The ACPI processor identifier.

Description

StartProcessor() starts execution of an AP at the address specified by the function **StartProcessorAddress()**.

StartProcessor() fails if the starting address has not been previously specified for the AP, the ACPI processor identifier is invalid, or if the AP is not currently stopped.

StartProcessor() is a blocking call. On success, it will not return until *after* the AP has been started.

Prior to calling **StartProcessor()** the MPP must be initialized using **InitMpProtocol()**.

Status Codes Returned

EFI_SUCCESS	Processor successfully started.
EFI_MP_ACPI_ID_FAILURE	ACPI processor identifier is invalid or is BSP.
EFI_MP_AP_NOT_STOP	Application processor is running.
EFI_MP_ADDRESS_FAILURE	Execution address has not been specified
EFI_MP_FAILURE	Processor not successfully started.
EFI_INVALID_PARAMETER	Invalid parameter passed to function

4.6 StopProcessor()

The **StopProcessor()** function stops execution of an AP previously started by **StartProcessor()**.

#include <atk_mp.h>

EFI_STATUS
StopProcessor (

IN	**EFI_MP_INTERFACE**	**This,*
IN	**UINT16**	*ACPIProcessorID*

) ;

Parameters

This	The EFI_MP_INTERFACE instance.
ACPIProcessorID	The ACPI processor identifier.

Description

StopProcessor() stops execution of an AP previously started by **StartProcessor()**.

StopProcessor() fails if the ACPI processor identifier is invalid or if the AP is not currently running.

StopProcessor() is a blocking call. On success, it will not return until *after* the AP has been stopped.

Prior to calling **StopProcessor()** the MPP must be initialized using **InitMpProtocol()**.

Status Codes Returned

EFI_SUCCESS	Processor successfully stopped.
EFI_MP_ACPI_ID_FAILURE	ACPI processor identifier is invalid or is BSP.
EFI_MP_AP_NOT_START	Application processor is not running.
EFI_MP_FAILURE	Processor not successfully stopped.
EFI_INVALID_PARAMETER	Invalid parameter passed to function

4.7 StartProcessorAddress()

StartProcessorAddress() assigns the address of the function where an AP begins execution. The function also specifies an argument to be passed to the function where an AP begins execution.

#include <atk_mp.h>

EFI_STATUS
StartProcessorAddress (

IN	**EFI_MP_INTERFACE**	**This,*
IN	**UINT16**	*ACPIProcessorID,*
IN	**VOID**	**Address,*
IN	**VOID**	*Argument*

) ;

Parameters

This	The EFI_MP_INTERFACE instance.
ACPIProcessorID	The ACPI processor identifier.
Address	Address of function to begin execution on AP.
Argument	Argument to be passed to function to begin execution on AP.

Description

StartProcessorAddress() assigns the address of the function where an AP begins execution and specifies an argument to be passed to the function when the AP is started.

StartProcessorAddress() fails if the ACPI processor identifier is invalid.

Prior to calling **StartProcessorAddress()** the MPP must be initialized using **InitMpProtocol()**.

Status Codes Returned

EFI_SUCCESS	Starting address for processor successfully assigned.
EFI_MP_ACPI_ID_FAILURE	ACPI processor identifier is invalid or is BSP.
EFI_MP_FAILURE	Starting address for processor not successfully assigned.
EFI_INVALID_PARAMETER	Invalid parameter passed to function

5 MPP – Usage Considerations

5.1 Seamless Promotion to MP OS

After loading the MPP in the IA64 environment, non-boot processors cannot be returned to the SAL boot rendezvous state. This limitation may cause the subsequent booting of an MP OS to fail. A reboot is required after using MPP to return the non-boot processors to the SAL boot rendezvous state.

5.2 Available Stack and Backing Store Resources

The MPP relies on the SAL to provide adequate stack and backing store resources for the non-boot processors in the IA64 environment. As such, the amount of stack and backing store resources are platform dependent.

5.3 Data Coherency Between Processors

Data coherence is guaranteed for all processors residing in the same coherence domain.

Appendix A: Glossary

AP	Application Processor. A processor not responsible for system initialization. A system may include multiple APs.
BSP	Bootstrap Processor. The processor responsible for system initialization. A system may include only one BSP.
IA	Intel Architecture.
IPI	Inter-processor Interrupt. Interrupt message sent between two processors.
MPP	Multiprocessor Protocol.
SAL	System Abstraction Layer.

Appendix B: Protocol Initialization Return Codes

The following table lists the protocol initialization return codes as found in the file atk_mp.h.

Error Code	Description
EFI_SUCCESS	Successful initialization
EFI_MP_FAILURE	Unsuccessful initialization
EFI_MP_ENUM_FAILURE	Unable to enumerate processors
EFI_MP_WAKEUP_FAILURE	Unable to wake application processors
EFI_MP_INTHANDLER_FAILURE	Unable to chain interrupt handler
EFI_MP_COMM_FAILURE	Unable to create communication buffers

Table 1 - Protocol Initialization Return Codes