Multi-processor Startup for ARM Platforms

September 23, 2014

Version 1

Microsoft Corporation

This document describes a protocol for starting multiple ARM processors on a platform. The protocol enables the OS to start processors from a known state without manipulating any voltage or clock controllers. The information in this document describes the boundary between firmware and the OS for starting the processors and is applicable to both firmware and OS writers.

**NOTICE**

The information contained in this document is subject to change without notice. Microsoft irrevocably promises, with respect to the MP Startup for ARM Platforms Specification, not to assert any of their respective Necessary Claims against any Implementers of the MP Startup for ARM Platforms Specification for making, using, selling, offering for sale, importing or distributing any implementation of the MP Startup for ARM Platforms Specification to the extent such implementation conforms to the MP Startup for ARM Platforms Specification, and is compliant with all of the required parts of the mandatory provisions of the MP Startup for ARM Platforms Specification (“Covered Implementation”), except where the Implementer asserts a claim against, or files, maintains or voluntarily participates in a lawsuit, arbitration, administrative action or other judicial or quasi-judicial proceeding against a Microsoft Covered Implementation. Here, “Implementer” means any party to this promise that implements a Covered Implementation. “Necessary Claims” are those claims of Microsoft-owned or Microsoft-controlled patents that are necessary to implement the required portions (which also include the required elements of optional portions) of the Specification that are described in detail and not merely referenced in the Specification. A patent is “controlled” by a party if (a) the patent is owned by an Affiliate of that party or (b) that party or its Affiliate, is the exclusive licensee of the patent; and that party, or its Affiliate, has the right to make this promise regarding the patent without a duty or obligation to pay a royalty or other fee to a third party. “Affiliate” means, in relation to a party, any entity controlled, directly or indirectly, by that party, any entity that controls, directly or indirectly, that party or any entity directly or indirectly under common control with that party.

This promise is not an assurance either (i) that any of Microsoft’s issued patent claims covers a Covered Implementation or are enforceable or (ii) that a Covered Implementation will not infringe the patents or other intellectual property rights of any third party. No other rights except those expressly stated in this promise shall be deemed granted, waived or received by implication, exhaustion, estoppel or otherwise.”

MICROSOFT MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

**Copyright © 2010 by MICROSOFT** **CORPORATION. All rights reserved.**

Table of Contents

[1 Document Terms 3](#_Toc345425047)

[2 Background 3](#_Toc345425048)

[2.1 Challenges 3](#_Toc345425049)

[2.2 Current ARM MP Initialization Methods 3](#_Toc345425050)

[3 Solution for MP Start 5](#_Toc345425051)

[3.1 Application Processor Parked State 6](#_Toc345425052)

[3.2 Application Processor Algorithm to Leave Parked State 7](#_Toc345425053)

[3.3 Starting Application Processors from the OS 7](#_Toc345425054)

[3.4 Mailbox Format 8](#_Toc345425055)

[3.5 Mailbox Access Rules 9](#_Toc345425056)

[3.6 BSP Mailbox 10](#_Toc345425057)

[3.7 Required Tables 10](#_Toc345425058)

[4 Other Considerations 10](#_Toc345425059)

[5 Revision Table 10](#_Toc345425060)

# Document Terms

|  |  |
| --- | --- |
| BSP | Boot strap processor, the processor that completes the firmware to OS transition. |
| AP | Application processor, processors on the system that will run the OS code other than the BSP. |
| Parked State | The state of application processors when the firmware hands control to the OS. |
| Processor ID | For GICv1/v2 implementations, the Processor ID contains the value stored in the CPU Interface Number field of the GICC Structure in the ACPI Multiple APIC Description Table (MADT). See the ACPI specification for more information (<http://www.uefi.org/acpi/specs/>).  GICv3 implementations which support compatibility mode should use GICv1/v2 Processor IDs for the compatible processors (up to eight), and 0 for the remaining ones.  GICv3 implementations which do not support compatibility mode should write 0 for all processors. |
| Version | The revision number of this document, currently “1”. The Parking Protocol Version field of the ACPI MADT should match. |
| Processors Mailbox | Shared memory that the Firmware reserves for each processor to enable the OS to send start messages to them. |

The terms ‘core’ and ‘processor’ will be used interchangeably.

# Background

## Challenges

Starting ARM-based platforms with multiple processors requires one processor to execute the firmware; this processor is typically the first processor that starts on the system. Additional processors are kept from interfering with the firmware start process. Starting these processors when the OS is up and ready requires manipulation of the voltage and clock controllers to release these processors from reset. In addition, the reset vector is not architecturally defined and any non-architectural state, which the OS may not manage, needs to be configured.

To abstract the variations in hardware and the details of non-architectural state, this document describes a protocol for the firmware to handoff the application processors in an architecturally-defined state called the Parked State. The advantage of having the firmware address MP configurations is that it can do initial platform-specific setup, increase boot performance through parallelism, and ensure that all processors are in a safe and defined state.

## Current ARM MP Initialization Methods

An increasing number of devices implement multiple ARM processors. Multiprocessing capability was added in ARMv6 and further refined in later architectural revisions and extensions. The ARM architecture does not provide a specific mechanism to gain control of a processor. There is no designated IPI or hardware mechanism that an OS can use to transition cores from a Parked State to executing OS code.

Existing systems have used conventions and protocols shared between the firmware and the OS. Typically, these protocols have consisted of leaving the core in a well-defined state, and defining a doorbell mechanism as well as some means for specifying what code the core will execute when it wakes up. In terms of interrupts, the interrupt controller used for today’s ARM MP implementations is the industry-standard ARM GIC.

# Solution for MP Start

This protocol does not require any additional firmware runtime services. The protocol consists of the following:

* An ACPI table (the MADT) describing:
  + The hardware cores in the system and their CPU identifiers.
  + A mailbox address associated with each of the processors in the system.
* A convention for the Parked State in which the firmware will leave the application processor.
* A (shared) mail box for each processor, which is a 4K memory block allocated by the firmware. Note that each processor must be allocated a separate physical page of the memory.
* An algorithm defining how an application processor determines that it has received a valid start condition from the OS and how to decode the address where it should begin executing OS code.
* An algorithm which the OS can use to signal a start condition to any application processor in order to transition it from the Parked State to an OS entry point.



Figure 1 State diagram for MP start protocol

## Application Processor Parked State

In the Parked State, application processors are not continuously executing code; they check their mailbox for start conditions and then go into a wait for interrupt (WFI) state if the start condition is not valid. In this state, all bits in the Processor ID must be set, and the jump address must be zero. From the WFI state, an interrupt will cause the processor to recheck the mailbox before going back into the WFI state.

*Note: While it is generally expected that all CPUs will be in the Parked State before OS entry, this is not a firm requirement and may not be true in certain scenarios. In particular, a processor may be powered off (and thus not be in the Parked State) before OS entry. However, so long as the processor can autonomously power on upon receipt of a software-generated interrupt and follow the steps described in this section, it will remain compatible with the outlined approach for OS entry. For ease of composition, in the rest of this document we refer to all such processors as being parked upon OS entry.*

The Parked State relies on the ARM architectural requirement that a core exit from WFI if there is a pending interrupt. When interrupt handling is disabled in the core, execution will resume at the next instruction.

The application processor acknowledges that it received a valid start condition by writing zero to the jump address mailbox field. This acknowledgement also serves as a means of invalidating the mailbox contents and returning control of the jump address to the OS.

In order to use this startup protocol the following conditions must be met:

1. All processors must be part of the same inner sharable domain
2. Firmware must initialize non-architectural core state (i.e. any state that is not defined by the ARM Architecture)
3. Before a processor enters coherency with any running processors in the system, all caches that will enter coherency must be invalidated
4. Before jumping to the OS, the processor must conform to the state described in Table 1

|  |  |  |
| --- | --- | --- |
| State | Execution state of FW and OS | |
| AArch32 (ARMv7 systems) | AArch64 |
| Instruction Set | ARM | A64 |
| Exception or privilege level | PL1 | EL2 if virtualization extensions are present, else EL1. |
| Endianness | little endian  CPSR.E must be 0 | little endian  SCTLR\_ELx.EE must be clear  (ELx being exception level that the firmware and OS reside in, EL2 or non-secure EL1) |
| Exceptions | Must be masked  CPSR. {A,I,F} bits must be set to {1,1,1} | Must be masked DAIF. {D,A,I,F} bits must be set to {1, 1, 1, 1} |
| GIC CPU Interface | Enabled | Enabled |
| MMU, Cache, Branch Prediction | Disabled  SCTLR.{I, C, M} bits must be set to {0, 0, 0}. In addition the Z bit must set in a way that is consistent with the reset behavior of the platform. | Disabled  SCTLR\_ELx.{ I, C, M} bits must be set to {0, 0, 0}  (ELx being exception level that the firmware and OS reside in, EL2 or non-secure EL1) |

Table 1 Processor State when jumping into OS

## Application Processor Algorithm to Leave Parked State

The algorithm described in this section defines two necessary conditions for an application processor to leave the Parked State. The Processor ID portion of the mailbox must match the Processor ID for the core executing the check (as represented in the CPU Interface Number field of the GICC Structure in the ACPI MADT), and the jump address portion of the mailbox must be non-zero. When the application processor observes these conditions it will do the following:

* It reads the jump address field in the mailbox and stores the value elsewhere.
* It writes 0 to the jump address field in the mailbox to acknowledge that it received a valid start condition.
* It executes a branch (BX for AArch32 / BR for AArch64) to the jump address it read from the mailbox.

*Note: If these two conditions are not met, the processor will assume it was a false wake and go return to the WFI state.*

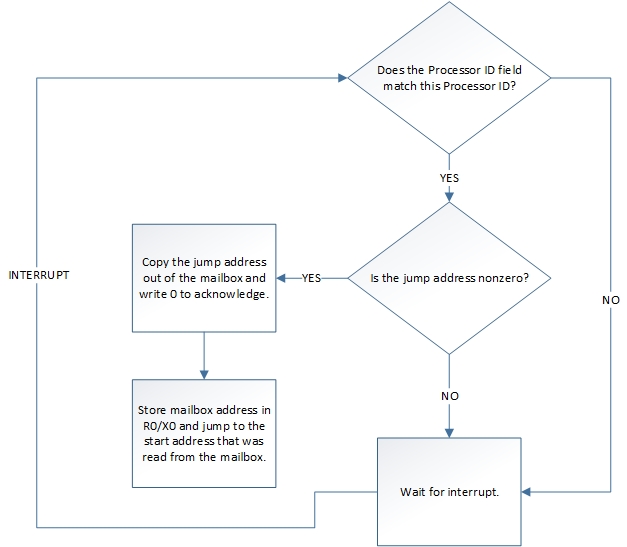


Figure 2. Parked Algorithm.

## Starting Application Processors from the OS

The OS will perform the following tasks to enumerate and start application processors:

* The OS learns the Processor ID and mailbox physical address for each processor by consuming the GICC structures in the ACPI MADT.
* The OS will map the mailbox area as strongly-ordered memory (On ARMv8 Device-nGnRnE).
* The OS checks if the processor is properly parked, ID is set to all ones.
* The OS configures the mailbox for each processor it decides to start by configuring the jump address and Processor ID portions of the mailbox with the ID of the target processor and the OS entry point respectively.
* The OS will execute a data synchronization barrier after each write to ensure that the data writes are completed in program order, all writes will be reflected in external memory because the mailbox is not cached.
* The OS will not attempt to start application processors before it calls ExitBootServices().
* The OS will send an interrupt to the target application processor to signal that the mailbox is ready. The purpose of the interrupt is to transition the processor from the Parked State to running state.

## Mailbox Format

The processor mailbox is 4KB block of memory and its address is 4K aligned. This memory is reported to the OS as permanently in use by the FW. The firmware is not allowed to relocate the mailbox when the OS starts running. The mailbox is broken down into two 2KB sections, an OS section and a firmware section. All data shares between the OS and FW as part of this protocol must be in little endian format.

The OS section contains a Processor ID slot and a Jump Address slot; these two slots are the only parts that the firmware is allowed to write. The rest of the OS section is reserved for OS use and should not be written by the firmware. The firmware section of the mailbox is reserved for firmware use exclusively.

For GICv1/v2 implementations, the Processor ID contains the value stored in the CPU Interface Number field of the GICC Structure in the ACPI Multiple APIC Description Table (MADT). See the ACPI specification for more information (<http://www.uefi.org/acpi/specs/>).

GICv3 implementations which support compatibility mode should use GICv1/v2 Processor IDs

for the compatible processors (up to eight), and 0 for the remaining ones.

GICv3 implementations which do not support compatibility mode should write 0 for all processors.

*Note: In any SMP OS, all processors must be in the same coherency domain (even if they are in multiple clusters).*

The jump address has been provisioned as a 64-bit address with future 64-bit compatibility in mind. On 32-bit Arm systems, the jump address field must be below 4GB.

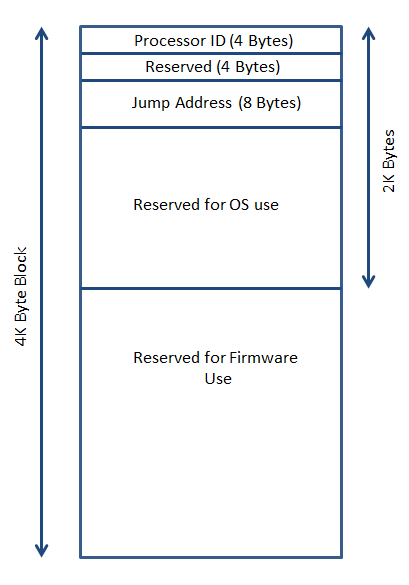


Figure 3. Mailbox Format.

## Mailbox Access Rules

The Processor ID and Jump Address parts of the mailbox can be written by the OS and the firmware. The firmware will initialize these parts of the mailbox to all ones for the Processor ID to indicate that it is parked and ready and all zeros for the jump address before executing the OS code. The firmware must perform this initialization when the platform is starting up with uninitialized memory contents. If memory contents were preserved, the firmware shouldn’t change the contents of the mailbox.

When the OS is running it will not access or change the firmware reserved sections of the mailbox. The OS may choose to change the jump address across power transitions that maintain the contents of the memory. In this case the OS must follow this sequence:

* Change the Processor ID to all ones
* Execute a data synchronization barrier
* Change the jump address to the required value
* Execute a data synchronization barrier
* Program the correct Processor ID in the mailbox

If the OS may change the jump address at which an application processor starts, it must invalidate (by changing the Processor ID to all ones as outlined above) the mailbox before changing the jump address. This invalidation avoids any race conditions if the application processor received any false wake from the WFI state while the OS is configuring the Jump address.

***Note: All accesses to the mailbox memory should occur through strongly-ordered memory mappings. On ARMv8 systems, the OS must map the memory as Device-nGnRnE.***

## BSP Mailbox

Even though it is not required to power up the platform from a cold boot, the BSP will have its dedicated mailbox. The OS and firmware will use this mailbox to start the BSP when the platform transitions to and from power states where memory contents is maintained.

This condition arises when the firmware is resuming the OS from a previous OS stored state that is maintained in memory. The OS will use the BSP mailbox to save its resume address. In this case, the firmware must avoid writing to the contents of the mailbox.

## Required Tables

The GIC CPU Interface for each processor is described in the GICC Structure as part of the ACPI Multiple APIC Description Table (MADT). This table also contains a descriptor for the distributor interface of the GIC. This descriptor will provide the OS with the following information:

* CPU Interface Number of application processors, which maps to the Processor ID in this document.
* Mailbox address of all processors

Systems that use ACPI must describe the processors through the ACPI-defined MADT table.

# Other Considerations

* The memory for the mailboxes has to be reserved by firmware.
* All access to mailbox memory is strongly-ordered (ARMv7) or Device-nGnRnE (ARMv8)
* After the firmware initializes the mailbox memory, it must avoid writing to that memory unless the contents of that memory are lost. (Exception: acknowledging a valid jump address).
* All cores must be configured in one coherency domain.

# Revision Table

|  |  |
| --- | --- |
| Date | Change |
| 9/30/2010 | First publication of the document |
| 1/12/2011 | Added note on cache invalidation for parked processor. |
| 10/13/2011 | Initial Release |
| 11/14/2011 | Public Release |
| 12/20/2012 | Small changes, including clarification on Parked States and non-cached memory access (changed to strongly-ordered), as well as a correction on the mailbox format. |
| 9/23/2014 | Changes surrounding the Processor ID to reflect support for ARMv8 and an updated MADT table in ACPI. |