# Software Developer's Guide

# Version 1.0

**LEGAL NOTICE**



This CCIX Software Developer's Guide ("**Guide**") is proprietary to CCIX Consortium, Inc. (sometimes also referred to as "**CCIX**") and/or its successors and assigns.

THIS GUIDE AND ALL CONTENT PROVIDED HEREIN IS PROVIDED ON AN "**AS IS**" BASIS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CCIX CONSORTIUM, INC. (ALONG WITH THE CONTRIBUTORS TO THIS GUIDE) HEREBY DISCLAIM ALL REPRESENTATIONS, WARRANTIES AND/OR COVENANTS, EITHER EXPRESS OR IMPLIED, STATUTORY OR AT COMMON LAW, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, VALIDITY, AND/OR NON-INFRINGEMENT.

Any references or citations to any portions of this Guide must acknowledge CCIX Consortium's Inc.'s copyright ownership of this Guide. The proper copyright citation or reference is as follows: **"© 2019 CCIX CONSORTIUM, INC. ALL RIGHTS RESERVED."** When making any such citation or reference to this Guide you are not permitted to revise, alter, modify, make any derivatives of, or otherwise amend the referenced portion of this Guide in any way without the prior express written permission of CCIX Consortium, Inc.

The CCIX® trademark and the CCIX name (collectively "**CCIX Mark**(s)") are owned solely by CCIX Consortium, Inc. and all rights are reserved therein by CCIX Consortium, Inc. Parties are not permitted to use any CCIX Marks without the prior consent of CCIX Consortium, Inc. or, in the case of CCIX Member, as may be permitted by the CCIX IPR Policy, CCIX Bylaws or any other CCIX policies.

The mere use or mere possession of this Guide by a party does not automatically give such party any separate rights or privileges related to such party's decision to implement or adopt any CCIX specifications, including without limitation any CCIX specifications which may be referenced in this Guide (collectively, "**CCIX Specifications**"). Thus, if a party using this Guide intends to implement or adopt any CCIX Specifications, such party is hereby notified that:

(i) If a party seeks to implement any publicly-available version of any CCIX Specification, such party must first consent to CCIX's Evaluation License Agreement to obtain access to and the right to use the publicly-available version of such CCIX Specification; and

(ii) Access to non-publicly available CCIX Specifications is typically limited to CCIX Members, and therefore only CCIX Members receive the benefits derived from the CCIX IPR Policy, CCIX Bylaws or any other CCIX policies as a result of such CCIX Member's implementation or adoption of any non-publicly available CCIX Specifications.

In the event this Guide makes any references (including without limitation any incorporation by reference) to another party's ("**Third Party**") content or work, including without limitation any specifications or standards of a Third Party ("**Third Party Content**"), you may need to independently obtain a license or other consent from that Third Party in order to have full rights to implement or use that Third Party Content.

# Release History

| Release Version | Date | Description |
|---|---|---|
| 1.0 | October 2019 | Version 1.0 for external release. |

# Table of Contents



# List of Figures



# List of Tables



# Chapter 1. Introduction

## 1.1 Scope

This guide describes CCIX software architecture and provides guidelines for software developers to implement, configure and manage CCIX based systems. This covers the areas of CCIX that are software visible, and where appropriate proposes software architecture solutions.

## 1.2 Reference Documents

**Table 1: Reference Documents**

| Reference | Document | Source |
|---|---|---|
| [1] | Cache-coherent Interconnect for Accelerators CCIX®, Revision 1.0a Version 1.0, July 2019 | https://www.ccixconsortium.com/ |
| [2] | Unified Extensible Firmware Interface Specification, Version 2.8, April 2019 | http://www.uefi.org/specifications |
| [3] | Advanced Configuration and Power Interface Specification, Version 6.3, Feb 2019 | http://www.uefi.org/specifications |
| [4] | UEFI Platform Initialization Specification, Version 1.6, May 2017 | http://www.uefi.org/specifications |
| [5] | Peripheral Component Interconnect, Express Base Specification Revision 4.0 Draft, May 2017 | https://pcisig.com/specifications |
| [6] | PCI Firmware Specification, Revision 3.2, Jan 2015 | https://pcisig.com/specifications |
| [7] | The Turn Model for Adaptive Routing, Glass C.J., Ni L.M., Association for Computing Machinery, 1992 | https://www.acm.org |
| [8] | The DeviceTree Specification | https://www.devicetree.org/ |
| [9] | PCI Bus Power Management Interface Specification Revision 1.2, March 3, 2004 | https://pcisig.com/specifications |
| [10] | Tianocore | https://www.tianocore.org/ |
| [11] | Non-Uniform Memory Access, Description Under ACPI 6.3 | http://www.uefi.org/specifications |

# 1.3 Terms and Acronyms

**Table 2: Terms and Acronyms**

| Term | Description |
|---|---|
| Accelerator | In this document, a CCIX accelerator refers to a processing unit, other than the application processors in the host, that provides accelerated compute for a specific use case. IO masters with compute capabilities are also considered accelerators in this document. CCIX accelerators can be cache coherent with host application processors. |
| AF | Acceleration Function. Logic componentry in a accelerator that provides acceleration of compute. |
| AFC | Acceleration Function Core. The basic interface and programming model for accessing an AF. This will minimally consist of a set of memory-mapped registers for accessing, configuring and controlling the AF, and for managing AFTs (see below). |
| AFT | Accelerator Function Thread. An independent execution thread within an accelerator that can be assigned to a process. |
| ACPI | Advanced Configuration and Power Interface |
| Agent ID | CCIX identifier for an agent. Agent IDs are used to route certain CCIX message types. |
| ATC | Address Translation Cache |
| ATS | Address Translation Services |
| BAT | Base Address Table |
| BDF | Bus, Device, Function |
| BDS | Boot Device Selection |
| BIOS | Basic Input/Output System |
| CCIX | Cache Coherent Interconnect for Accelerators |
| CCIX-SAM | CCIX System Address Map |
| CCSR | Configuration, Control and Status Registers |
| CCSR Space | Address space where the CCIX CCSR structures are located. For example, the Protocol Layer DVSEC structure is an example of a CCSR space when PCIe is used as the underlying transport. |
| CPU | Central Processing Unit |
| DRAM | Dynamic Random Access Memory |
| DVSEC | Designated Vendor-Specific Extended Capability |
| DWORD | Double word |
| DXE | Driver Execution Environment |
| ECAM | Enhanced Configuration Access Mechanism |
| EDK | EFI Development Kit |
| EQ | Equalization |

| | |
|---|---|
| ESM | CCIX Extended Speed Mode |
| FIFO | First In, First Out |
| FLR | Function-level Reset |
| GCD | Global Coherency Domain |
| G-HSAM | Global HA-to-SA System Address Map |
| G-RSAM | Global RA-to-HA System Address Map |
| G-SAM | Global System Address Map |
| GUID | Globally Unique Identifier |
| HA | Home Agent |
| HOB | Hand-Off Block |
| HSAM | Home agent System Address Map |
| IDM | ID Map |
| IOMMU | Input Output Memory Management Unit |
| ME | Management Engine |
| MMU | Memory Management Unit |
| MPU | Memory Protection Unit |
| NUMA | Non-Uniform Memory Access |
| NVRAM | Non-Volatile Random-Access Memory |
| PCIe | Peripheral Component Interconnect Express |
| PEI | Pre-EFI Initialization |
| PASID | Process-Address Space Identifier |
| PSAM | Port System Address Map for Output |
| PMCSR | Power Management Control and Status Register |
| OS | Operating System |
| RA | Request Agent |
| RAS | Reliability, Availability and Serviceability |
| RC | Root Complex |
| RP | Root Port |
| RSAM | Request agent System Address Map |
| RT | Run Time |
| SA | Slave Agent |
| SAM | System Address Map |
| SW | Software |
| TC | PCIe Traffic Class |
| TLP | PCIe Transaction Layer Packet |
| UEFI | Unified Extensible Firmware Interface |
| VC | PCIe Virtual Channel |
| VM | Virtual Machine |
| xSAM | RSAM or HSAM (term used to seamlessly refer to both in a sentence) |

# 1.4 Document Conventions

A CCIX system can be built up in any platform which supports PCIe.

## 1.4.1 Data Structure Descriptions and Types

This document uses "little endian" for data structure where the low-order byte of a multi-byte data item in memory is at the lowest address, while the high-order byte is at the highest address. All numeric values in defined tables, blocks, and structures are always encoded in little endian format.

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

The UEFI common data types are used in the interface definitions [2]. Unless otherwise specified all data types are naturally aligned. Structures are aligned on boundaries equal to the largest internal datum of the structure and internal data are implicitly padded to achieve natural alignment.

## 1.4.2 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 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 UEFI Specification [2].

## 1.4.3 Number Formats

A binary number is presented in this document by any sequence of digits consisting of only the Western-Arabic numerals 0 and 1 immediately followed by a lower-case b (e.g., 0101b).

A hexadecimal number is represented in this document by 0x preceding any sequence of digits consisting of only the Western-Arabic numerals 0 through 9 and/or the upper-case English letters A through F (e.g., 0xFA23), or by any sequence of digits consisting of only the Western-Arabic numerals 0 through 9 and/or the upper-case English letters A through F followed by a lower-case h (e.g. FA23h).

A decimal number is represented in this document by any sequence of digits consisting of only the Arabic numerals 0 through 9 not immediately followed by a lower-case b or lower-case h (e.g., 25).

### 1.4.3.1 Binary Prefixes

In order to be consistent with other CCIX specifications, this document uses the customary prefix symbols Ki, Mi, Gi and Ti defined in JEDEC standards for semiconductor memory, as follows:

**Table 3: Binary Prefixes**

| Value (Base 2) | Name | Symbol |
|---|---|---|
| 1,024 ($2^{10}$) | Kibi | Ki |
| 1,048,576 ($2^{20}$) | Mebi | Mi |
| 1,073,741,824 ($2^{30}$) | Gibi | Gi |
| 1,099,511,627,776 ($2^{40}$) | Tebi | Ti |

For example, 64KiB means 65536 kibibytes or 64 x $2^{10}$ bytes.

# Chapter 2. Architectural Overview

The Cache Coherent Interconnect for Accelerators (CCIX) is an architecture for data sharing between a host, and peripheral-attached memory and accelerators. CCIX provides coherency between accelerators, processors and memory, enabling seamless data movement among them. The CCIX architecture relieves the operating system, or drivers, from having to execute cache management operations in either processor or accelerator caches, when data is being shared between them.

A detailed overview of CCIX architecture can be obtained from [1]. This document focuses on the architecture from a software and firmware perspective. From this perspective, CCIX provides two distinct types of system devices: cache coherent accelerators and expansion memory. This guide outlines software's role in configuring, enabling and managing these devices and their associated use-cases.

## 2.1 Introduction

A CCIX-enabled system is a computer system replete with a host that consists of memory, processor and IO devices, and one or more CCIX devices, an example of such a system is illustrated in Figure 1. For PCIe-based CCIX implementations, the CCIX Root Port depicted is effectively a CCIX-aware PCIe Root Port. It is possible to build CCIX systems that have a mix of CCIX and native PCIe endpoints below a given CCIX Root Port.

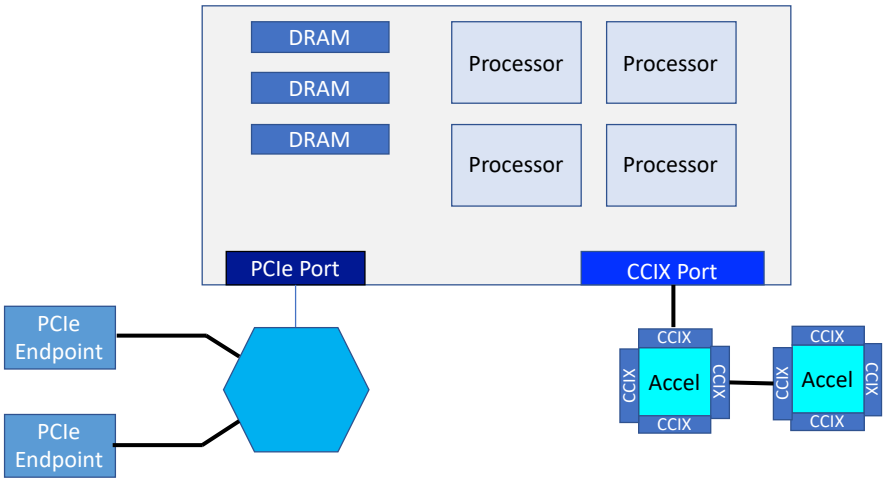



**Figure 1: A CCIX-enabled System**

Figure 2 provides a high-level overview of a typical CCIX device, its internal components and how it interfaces with other CCIX devices in the system.

As depicted in Figure 2, a CCIX device comprises key architectural components – the Acceleration Function (AF), the Request Agent (RA), the Home Agent (HA), the Slave Agent (SA), Ports and Links. These components and their functional properties are elaborated in the following sections in the context of the use-cases that they serve.

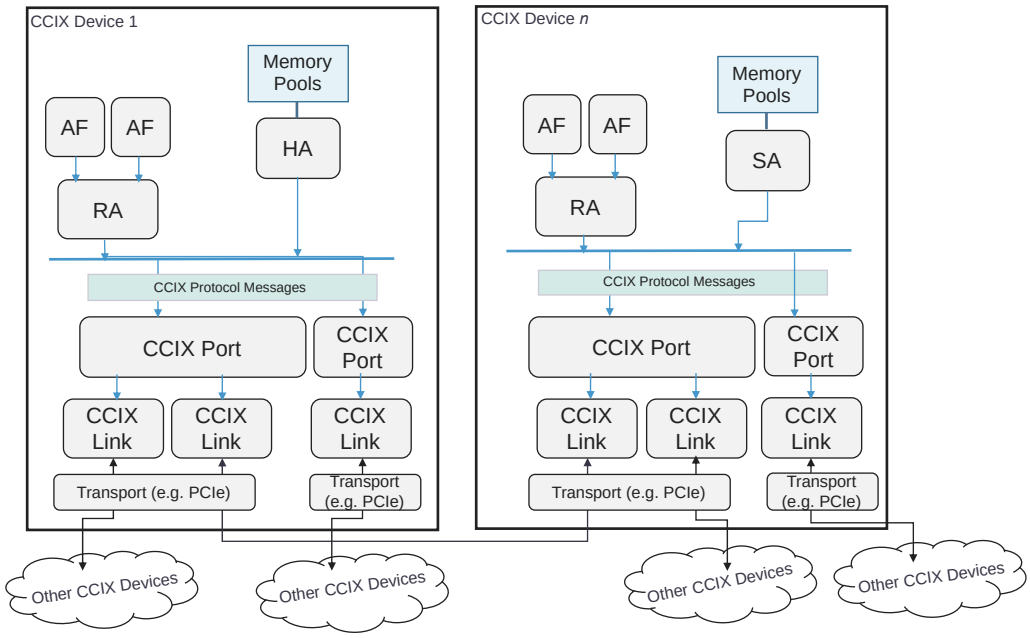



**Figure 2: A CCIX Device Network depicting CCIX components**

# 2.2 Accelerators

In this document, a CCIX accelerator refers to a processing unit, other than the application processors in the host, that provides accelerated computation for a specific use case. IO masters with compute capabilities are also considered accelerators in this document. CCIX provides accelerators that are hardware cache coherent and therefore do not require software support for cache management. From a software perspective, CCIX accelerators are composed of two major components - Acceleration Functions and Request Agents.

## 2.2.1 Acceleration Function (AF)

From a software point of view, a CCIX Acceleration Function (AF) comprises the logic components of the accelerator that provide computation. The software programming interface provided by the AF is called the *Acceleration Function Core* (AFC). The AFC allows software to:

- communicate with the compute engine(s) within the AF
- configure the AF
- query the AF for status and capabilities
- assign the AF to software-based consumers (e.g. a Virtual Machine)

In this document, the term AF will be used to refer to the *generic* notion of a CCIX acceleration function, while the term AFC is used to refer to the *specific* notion of the software-visible programming interface of an AF. This point is illustrated in Figure 3.

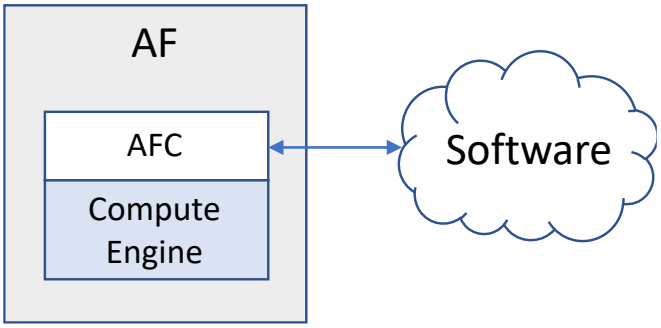



**Figure 3: AF and AFC**

An AF might comprise one or more AFCs. These aspects are illustrated in Figure 4.

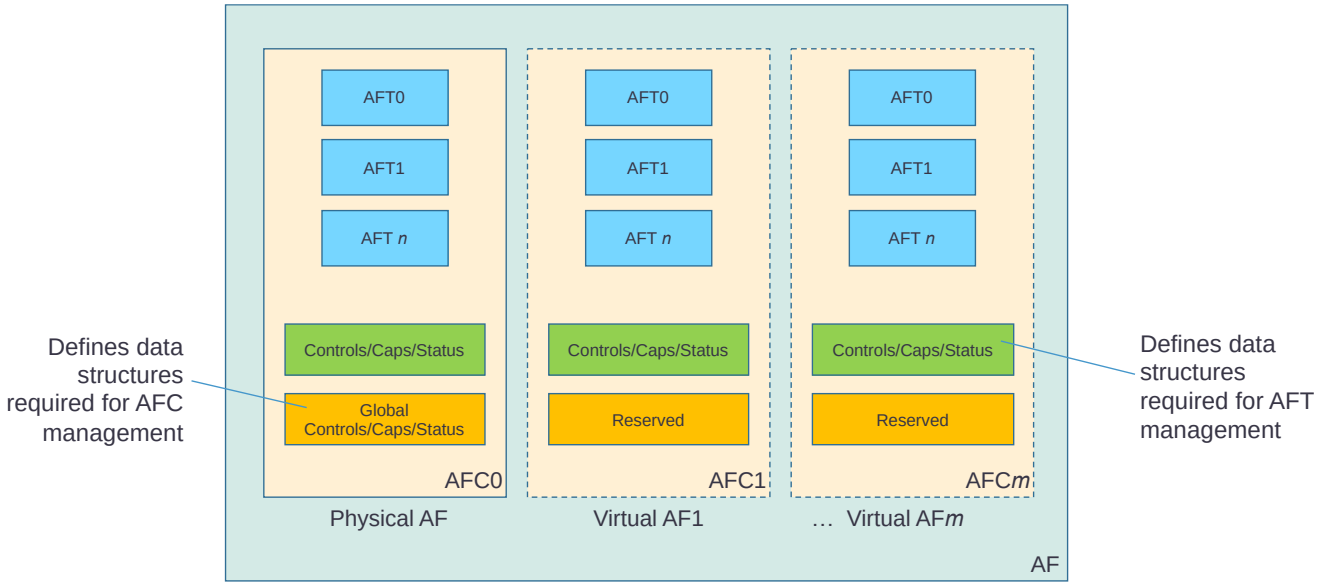



**Figure 4: AFCs and AFTs in an AF**

### 2.2.1.1 Acceleration Function Core

An AFC comprises capabilities and components that are used by supervisory software to manage the operation of the AF. From a virtualization standpoint, an AFC is the fundamental unit that might be assigned by a hypervisor to a Virtual Machine in a virtualized environment. Equivalently, an OS driver may manage the AFC in a bare-metal system.

The AFC itself might comprise the following ingredients in order to provide acceleration services to software:

- One or more contexts belonging to the AFC. Each such context is called an *Acceleration Function Thread* (AFT). AFTs are mutually independent. AFTs are described in detail in the next section.
- A set of controls, capabilities and status registers for managing the AFC, which include functions such as:
  - Enabling/disabling the AFC
  - Configuring event signaling/interrupts

- Setting up virtual addresses for the AFTs
- Assignment/mapping of AFTs to processes (PASID)
- Resetting the AFC
- Power management of the AFC
- Discovering the AFTs within the AFC

AFC0, also called the *Physical Acceleration Function* (PAF), must always be present. AFC0 provides global controls and capability and status for the AF as a whole. A non-zero AFC (AFC*n*, where n ≠ 0), also called a *Virtual Acceleration Function* (VAF), is assignable to a VM in a virtualized environment.

### 2.2.1.2 Acceleration Function Thread

As already described in the preceding section, an AFT is a context that software can use to avail the acceleration capability of the AF. Physically, an AFT is an independent unit of hardware thread that executes an internal acceleration logic. The AFT responds to commands from its user, and operates on data supplied by that user. Both the commands and data might be supplied in external memory.

The internal layout of an AFT in relation to its parent AFC is illustrated in Figure 5.

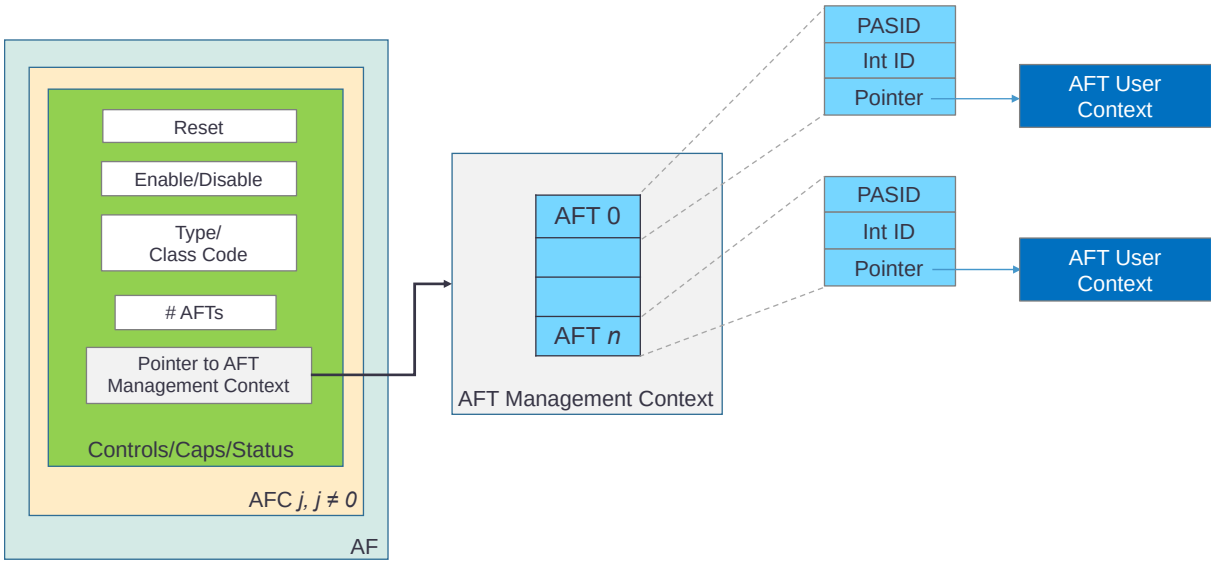



**Figure 5: Internal Composition of an AFT**

The parent AFC of a set of AFTs provides controls, capabilities and status registers for discovery and management of the AFC, which include functions such as:

- Discovering the AFTs
- Configuring the user context for each AFT

The AFT user context is a pointer to an implementation-defined data structure, which is valid if not NULL. The data structure is used for information and data sharing between the AFT and its user process. Examples of data structures could be command/response buffers, communication channels, shared memory.

### 2.2.2 Request Agent

The CCIX Request Agent (RA) is described in [1]. The RA can issue coherent read and write transactions to physical addresses within the system. An RA can also cache the memory locations accessed. From a software perspective, the RA can be thought of as the component of a CCIX device that enables it to participate in the cache coherent network, effectively acting as an accelerator peer of a CPU. An RA might support one or more AFs. Because the RA only operates on system physical addresses, AFs that might issue virtual addresses will require an ATC in order to use the RA.

## 2.3 Expansion Memory

CCIX architecture provides support for memory in three ways:

- It allows an accelerator to have attached memory that the host and other accelerators can access
- It also allows for devices that just provide memory for expanding memory capacity
- It allows accelerators to coherently access host memory and also cache those accesses, just like CPUs do in a multiprocessing system

In all three cases, the memory appears in the system's physical address map. Memory provided by CCIX can thus be considered as an OS-managed pool of memory with non-uniform latency and bandwidth and reliability characteristics. CCIX memory thus conforms to the Non-Uniform Memory Architecture (NUMA) model. Boot firmware is responsible for discovery of CCIX-attached memory, and for mapping this memory in the global system physical address map. Additionally, boot firmware can create necessary system description tables to allow the OS to locate CCIX memory and discover its NUMA properties. NUMA description of CCIX memory is discussed further in Chapter 4.

### 2.3.1 Home Agent

As described in [1], CCIX-attached memory is provided by two types of CCIX agents. The CCIX Home Agent (HA) is responsible for managing coherency and accesses to memory for a range of physical address space. An HA manages coherency by sending snoop transactions to RAs as required. Each HA acts as a Point of Coherency (PoC) for the address ranges that it manages. An HA can have its own physical memory.

### 2.3.2 Slave Agent

The CCIX Slave Agent (SA) also provides physical memory, but as an extension of the memory managed by the HA. The difference is that the SA memory must be homed by a parent HA. The SA enables support for building memory expansion device, allowing the memory associated with an HA in one device, to be physically provided by a separate device. SA memory pools must always be accessed through an associated set of parent HAs. Memory accesses from RAs are never directly sent to SAs, instead these accesses are accepted by the parent HA first, and then the HA redirects them to the SA.

## 2.4 Host

In this document the term host is used to refer to the overall computer system to which CCIX components have been added. Host processors are the main application processors that run the operating system. The host might

include its own memory, which CCIX can augment. An illustration of a host system is depicted in Figure 1. CCIX adds the following requirements to the host system architecture:

1 The host system needs to be able to send and receive CCIX messages, to enable processors to issue memory reads and writes to CCIX agents, and to allow processor caches to respond to snoop requests from CCIX agents.

2 The host system will house a CCIX Error Agent (EA). This agent needs to able to sink CCIX Protocol Error (PER) Messages, and log sufficient information in host-specific error records to aid further diagnosis of the error. Details of the CCIX Error Agent and PER handling support in CCIX are available in [1].

3 The host connects to the CCIX network over at least one CCIX port.

# 2.5 Supported Topologies

The simplest topology highlighted in Figure 6 below pertains a single-node tree, which may be extended to more complex trees with arbitrary number of nodes. The second topology is a mesh, and the third is the most-common complex topology – the fully-connected. Guidelines for discovery and configuration of these topologies is provided in Section 11.1.2. Furthermore, a CCIX node can provide acceleration, or expansion memory or both use-cases, as illustrated below in Figure 7.

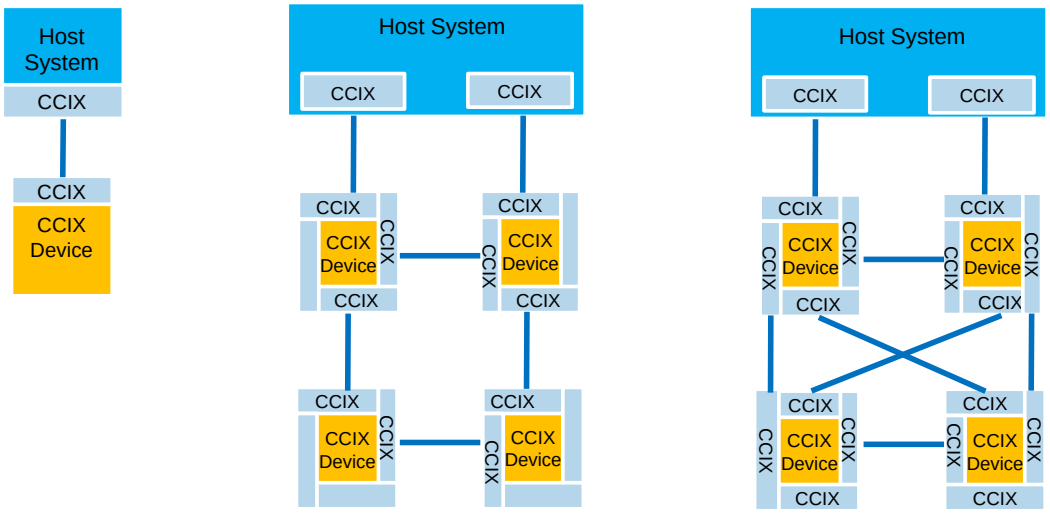



**Figure 6: CCIX topologies supported in CCIX 1.0a [1]**

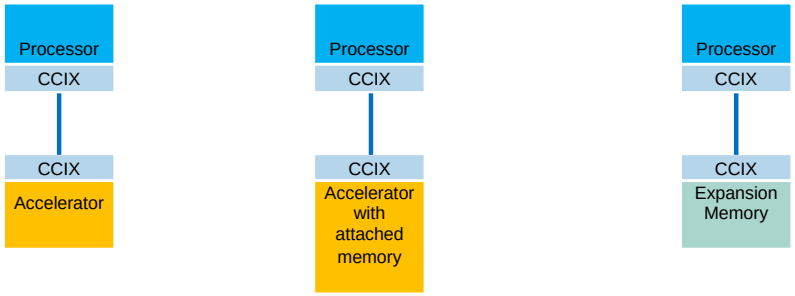


Figure 7: CCIX device classes based on use-cases

# 2.6 Address Spaces and Routing

## 2.6.1 System Address Maps

This document uses different terms to refer to various views of the System Address Map (SAM). Each such view is from the perspective of a specific component in the system:

**Host System Address Map (Host-SAM)**: When a host first boots, prior to CCIX enumeration, the host will have memory and IO devices in its physical address space, which are provided through host architecture specific means. This address map, as seen by a processor in the host, is referred to as the Host-SAM. Host-SAM is created before starting CCIX enumeration.

**Global System Address Map (G-SAM)**: The CCIX enumeration process, described in Chapter 3, extends the Host-SAM by adding to it the memory ranges provided by CCIX components. The overall resulting system memory map is referred to as the G-SAM.

[1] introduces additional definitions of G-RSAM, RSAM, G-HSAM and HSAM.

**Global RA-to-HA System Address Map (G-RSAM)**: is the combined view from all RAs of the G-SAM, which includes host memories accessible by each RA, and all CCIX provided memory that is visible for all RAs. An RSAM describes an RA's configuration structures that govern the portion of system address map that is visible to it. The G-RSAM is constructed by discovery firmware during the boot flow, and is then reflected in the RSAM table of a CCIX device.

**Global HA-to-SA System Address Map (G-HSAM)**: It is the combined view from all HAs of memory pools provided by SAs. An HSAM refers to the individual configuration structures of an HA, that control the portions of the address map where the HA acts as the point of coherency for an SA. CCIX boot firmware generates HSAM entries for each CCIX device based on G-HSAM.

Further details of these various views of memory can be obtained from [1] and Chapter 3.

## 2.6.2 Ports and Links

A CCIX device can have one or more CCIX ports that link it with one or more other CCIX devices. When the transport layer is PCIe, CCIX ports are layered on top of PCIe ports. A sample CCIX network depicting CCIX ports is illustrated in Figure 8.

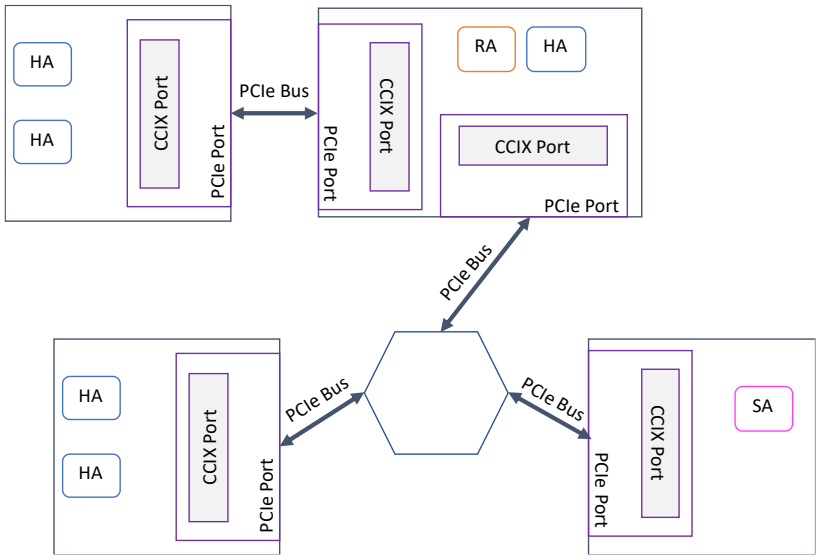



**Figure 8: CCIX Network Built using CCIX Ports**

A CCIX port might offer multiple CCIX links, where a link is a distinct communication channel within a CCIX port. Links might be used for a dedicated pair of source and destination agents on the two devices, or for general-purpose use by all agents. When both RA→HA and HA→SA traffic are required to flow over the same port, it is required for the port to implement two distinct links for each type of traffic respectively, as described in [1]. Links are realized using a pair of transmitting and receiving credit resource pools. Credit buffers are used in order to transfer and receive messages over the link, and the credits themselves are used for flow control and for marshaling the traffic. In particular, when a CCIX protocol unaware switch is used to establish the links between the current port and destination ports, then the link provides details required for routing traffic between peer ports. The concept of a link is illustrated in Figure 9, which also elaborates these above properties of a link.

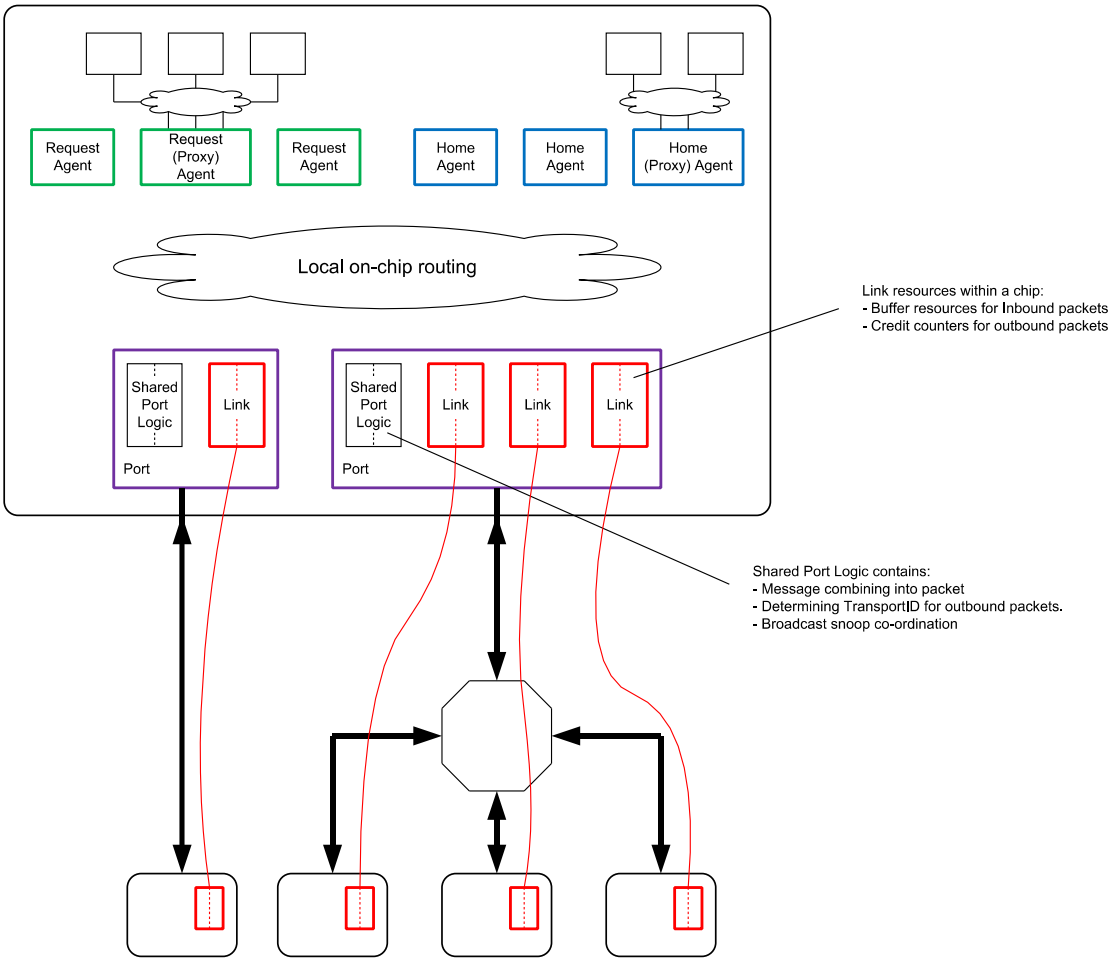



**Figure 9: CCIX Links**

It may be worthwhile noting here that if the PCIe switch in Figure 9 is replaced with a CCIX switch, then the port doesn't need separate links as it only needs to communicate with the peer port on the switch for routing traffic to distant ports beyond the switch.

### 2.6.2.1 Port Aggregation

CCIX can achieve higher bandwidth connectivity between two CCIX devices by optionally aggregating multiple CCIX ports. The CCIX Architecture defines a method to distribute memory access requests and snoops across multiple CCIX ports, where each CCIX Port maps to a PCIe controller when PCIe is used as a transport, to effectively achieve higher bandwidth between CCIX Agents. Port Aggregation is typically used where the throughput available from a single Port is not sufficient to meet the communication needs between two chips.

Software is responsible for detecting and configuring aggregated ports based on methods outlined in [1].

## 2.6.3 Memory Pools

The physical memory managed by an HA is called a memory pool. HAs manage coherent accesses to memory pools that they own. To do so, HA's participate in the coherency domain protocol with RAs. When an HA

receives a memory request from an RA, it must send snoops to other RAs in order to obtain the latest copy of the requested memory that might be in a cache belonging to a different RA. It might then update its own memory and then service the original request, thereby establishing overall memory coherency. An HA might maintain a directory of RAs requests in a Snoop Filter. This follows the directory-based caching model.

An HA might have many memory pools associated with it. Conversely, a memory pool might be managed by multiple HAs, where the exact manner in which the memory ranges within the pool are distributed among the HAs is an implementation choice.

SAs also manage memory pools, which are associated with a parent HA in order to enable coherent accesses to them. An SA might be owned by multiple HAs. Conversely, an HA might own multiple SAs.

Memory pools and their characteristics (i.e. size, type) are described in CCSR structures. For more information, refer [1].

## 2.6.4 CCIX Component Identification

Components in a CCIX system are identified using global identifiers.

### 2.6.4.1 Agent Identifiers

Protocol-specific components (RA, SA and HA) are called agents, and are identified using Agent Identifiers, or Agent IDs. ID-routed CCIX protocol messages that are exchanged between agents carry Agent IDs to allow message carriers and recipients to identify the transmitting agent and the intended recipient of the message. The boot firmware is responsible for assignment of Agent IDs. Agent identification assignment follows the rules of thumb mentioned below:

1. Each agent of a particular type must be assigned an identifier that is globally unique among all agents of that type. This allows that agent to be addressed in a globally unique manner, and messages to that agent to be sent using a distinct route from the sender to that agent.
2. An RA and an HA can be assigned the same identifier only when they reside on the same device.
3. An HA and an SA on the same device cannot share the same identifier.

### 2.6.4.2 Error Agent ID

The Error Agent ID (EAID) represents the recipient or sink of CCIX Protocol Error Reporting (PER) messages, described in detail in [1]. The Error Agent can have a unique identifier or follow a rule of thumb analogous to Rule 2 in Section 2.6.4.1 which states that the Error Agent could be assigned the same identifier as another agent of a different type, only when both Agents reside on the same device. Furthermore, a multi-socket CCIX Host might choose to have an Error Agent per Socket, and program the set of CCIX Devices branching out of each socket's Root Ports with a socket-specific EAID, such that the socket-specific Error Agent is the recipient of CCIX PER messages for the set of CCIX Devices branching out of each socket's Root Ports.

#### 2.6.4.3 Other Identifiers

In addition to agent identifiers described above, a CCIX system has the following identifiers types: Device ID, Port ID, Link ID. The Port ID and Link ID are assigned by the device hardware. The other identifiers are assigned by the boot firmware during initialization.

#### 2.6.4.4 Port ID

Each port on a CCIX device is assigned a unique Port ID by the hardware.

#### 2.6.4.5 Link ID

Each link in a port is assigned a unique Link ID by the hardware. Furthermore, when PCIe is used as the transport layer, the boot firmware must assign the Link a unique PCIe BDF or Requester ID, to identify the peer port at the destination device. The BDF of the peer port is used for routing PCIe TLP packets over the underlying PCIe fabric, and to ensure that the routed packets reach the peer port.

##### 2.6.4.5.1 Device Identifier

The Device Identifier (Device ID) is a unique identifier assigned by boot firmware to a given CCIX device in the system.

##### 2.6.4.5.2 AF Reference Index

AFs do not have a CCIX-defined identifier. However, AFs are located through an indirect reference, called an AF Reference Index. If the AFC of the AF is implemented as a PCIe function, then the AF Reference Index usually is the combination of a CCIX port and a PCIe function on that port.

##### 2.6.4.5.3 Memory Pools

Memory pools are identified by entries in the Base Address Table (BAT) of an HA or SA. There is no explicit CCIX-defined identifier for a memory pool, except its indexed reference in the BAT of that HA or SA.

##### 2.6.4.5.4 Transport ID

The Transport ID specifies a unique slot on the underlying transport that carries the CCIX layers. When PCIe is used as the transport layer, then the Transport ID equates to the BDF or Requester ID of the underlying PCIe port. The Transport ID is programmed in the Link.

### 2.6.5 CCIX Tables

A CCIX device carries multiple tables to describe the logical and physical relationships between components within the CCIX system. Some of the tables contain read-only information set by the device hardware, while others are populated by software at the time of device discovery. Components described in these tables are identified using their identifiers defined in Section 2.6.4.

CCIX devices can carry the following tables:

- The ID Map Table (IDM)
- A Request Agent System Address Map Table (RSAM) or an Home Agent System Address Map Table (HSAM) if they have an RA or an HA, respectively, or both if they have both
    - In the case of a CCIX device that is a switch, or has port to port forwarding, it is necessary for the device to carry xSAM tables even if it has no local agents.
- The Port to System Address Map Table (PSAM) if they have an RA or HA or both and the port has multiple links

A CCIX device might additionally carry the following tables depending on the nature of its components:

- An AF Property Table; if the device implements AFs
- A BAT Table; if the device implements an HA or SA with at least some local memory. An SA might only have local type. An HA is permitted to not have any local memory. HA and SA must have ≥ 1 memory pool.

The CCIX Base specification provides a comprehensive set of rules related to these tables [1].

### 2.6.5.1 The ID Map Table

CCIX messages that are routed to their intended destination based on an Agent ID are called ID-routed messages. An example of an ID-routed message is a snoop response from an RA to a distant HA. The Agent ID in ID routed messages can be the identifier of any Agent type at the destination CCIX Device, including Error Agent.

ID-routed messaging is realized in the form of the ID Map (IDM). The IDM is a table that resolves a destination Agent ID into an egress link in the device, where that link lies in the route from the device to the destination device. The IDM Table enables local agents on the device to communicate with remote agents. The IDM table on a CCIX switch enables forward communication between two remote Agents on either side of that CCIX Switch. Every CCIX device is required to carry an IDM table. An example is a device with an HA, which requires an IDM table to allow that HA to send snoop requests to remote RAs. Another example is a device with an SA, which needs an IDM table to respond to memory requests from remote HAs.

> **IMPLEMENTATION NOTE**: There could be exceptions to these rules. For example, an SA-only device with a single port might choose to implement no IDM tables, because all memory responses that it generates are always propagated to that lone port. For more details on such nuances, the CCIX Base Specification should be consulted [1].

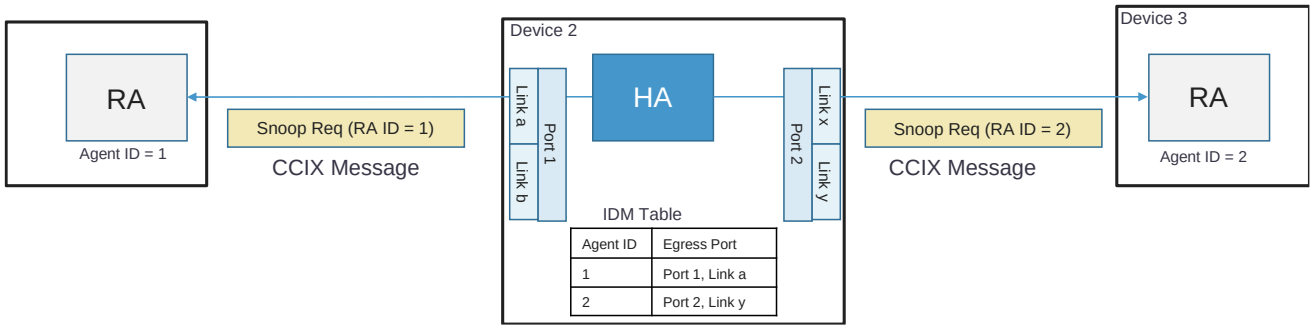



**Figure 10: ID-routed Messaging using IDM Tables**

### 2.6.5.2 The SAM Table

CCIX messages that are routed to their intended destination based on the destination address of the message are called address routed messages. An RA in a CCIX device sends memory requests to an HA in a distant device using address-routed messages. Similarly, an HA on a device sends memory requests to an SA on a distant device using address-routed messages.

The System Address Map (SAM) table enables routing of address-routed messages. A CCIX device must provide an RSAM table if it either has an RA, or an HSAM if it has an HA with memory extension capability, or is a CCIX Switch based on its port-to-port forwarding capability. Locally to the device, the SAM table resolves address ranges into egress ports on the device, such that each port lies on a distinct route between the local device and the destination device where the targeted memory pool resides. Thus, the route to be taken by address routed messages is resolved in part by referencing the SAM Table.

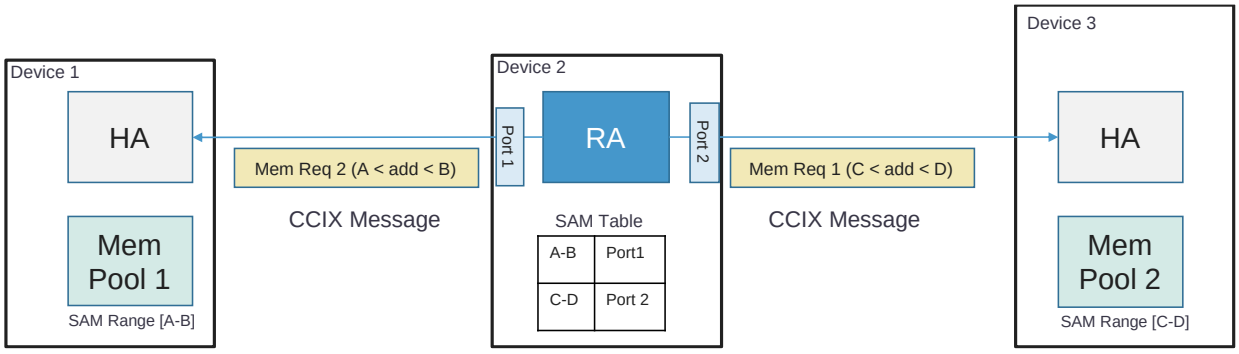



**Figure 11: Address-routed Messaging using SAM Tables**

### 2.6.5.3 The PSAM Table

A port might have multiple links associated with it. Local agents (e.g. RAs) direct memory requests to the port based on the SAM table. When a memory request from a local agent arrives at the port, the port must have some means of determining which link below it is the right path to the destination memory pool. The Port SAM (PSAM) table provides this means.

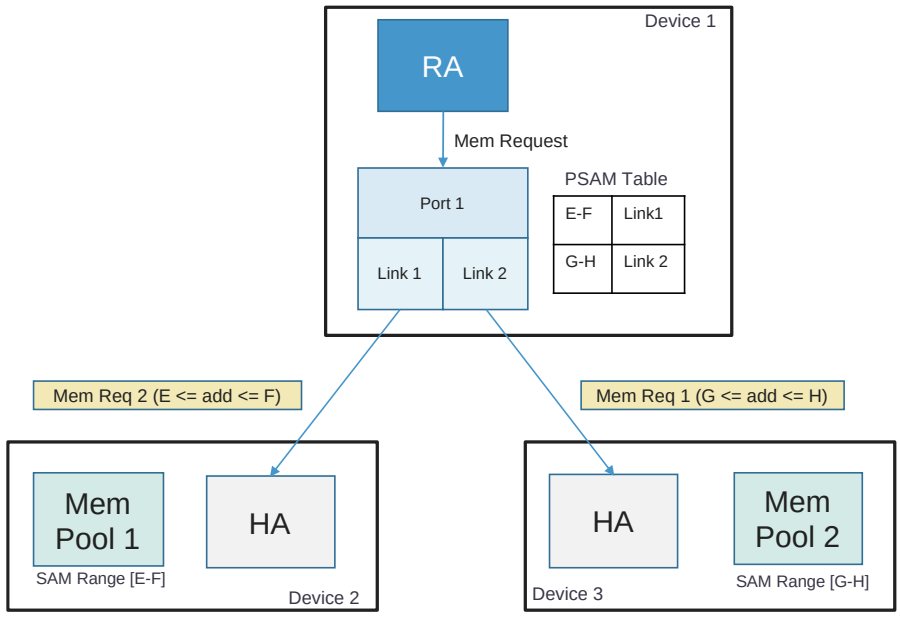



**Figure 12: Address Routing via Port SAM (PSAM) Table**

### 2.6.5.4 The AF Property Table

The AF Property is described as a CCSR structure in [1]. This provides a reference in the form of a {port, function} pair to identify the location of an AFC in the device. Software must consult the AF Property Table to discover presence of AFs in the device, and which RA or RAs provide CCIX messaging service to those AFs.

### 2.6.5.5 The Base Address Table

The BAT table is an indexed table of pointers to the configuration structures for memory pools. An HA or SA must provide a BAT table in order for software to program base addresses, or offsets from a base address, of the memory pools in system address map, for the memory pools discovered. An SA's memory pool may be pointed to by multiple BAT table entries in distinct HAs, thereby permitting the HAs to share the same memory pool.

The relationship between the component identifiers and the CCIX tables is shown in Figure 13. As shown, memory pools and AFs do not have explicit identifiers, and are instead described indirectly via table entries.

More details on how CCIX topologies are overlaid on to PCIe topologies can be found in Section 3.4.5.

### 2.6.5.6 Relationship between Identifiers, Components and Tables

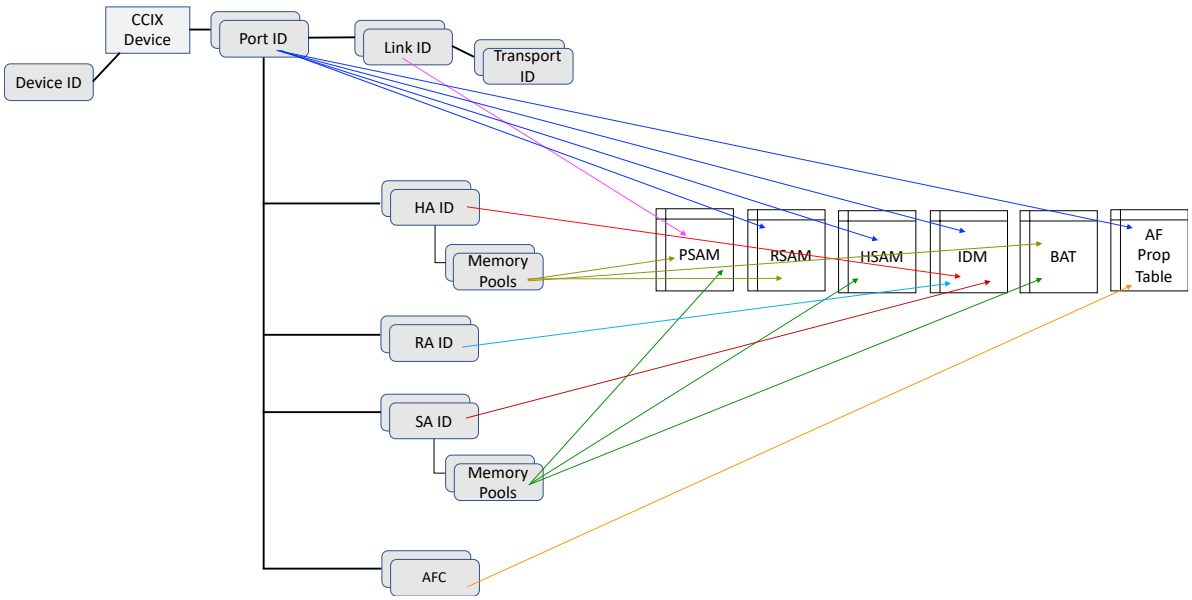



**Figure 13: Relationship between CCIX Component IDs and CCIX Tables**

# 2.7 Reliability, Availability and Serviceability (RAS)

The CCIX Base Specification [1] defines the RAS architecture used by CCIX components to report and log errors. Key ingredients of this architecture are:

**Error messages:** CCIX Protocol Error (PER) are reported through PER messages. The messages are routed by ID to an Error Agent.

**Error Agent:** As described in Section 2.4, the error agent resides in the host. This agent has the role of converting the message into a host processor architecture specific exception or event, and to optionally log the message so that a CCIX generic driver can triage the error.

**Error control and status registers:** CCIX CCSR structures include controls to mask error reporting and logging as well to observe error status.

**Error logs:** CCIX component and device-wide CCSR structures include error logging and reporting sub-structures for CCIX agents.

Further information on CCIX RAS architecture can be obtained from [1]. For further details on the software impacts see Chapter 5.

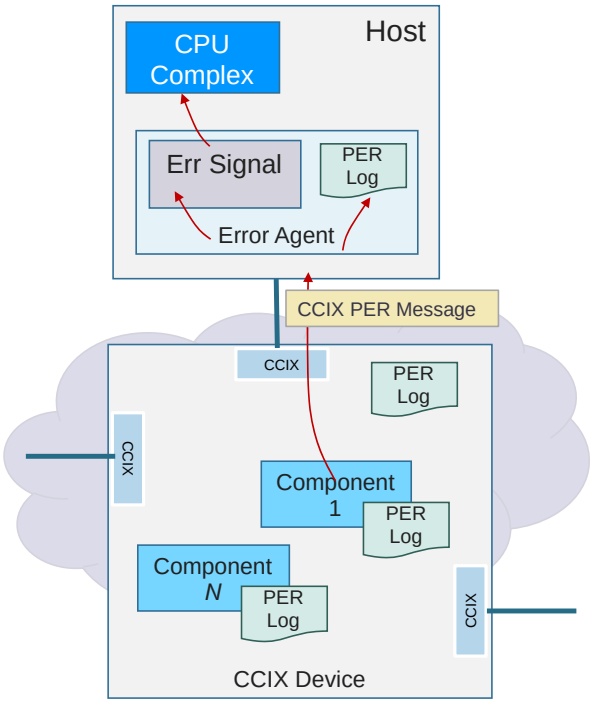



**Figure 14: CCIX Error Logging and PER Message Propagation**

# 2.8 The Role of PCIe in CCIX

## 2.8.1 Discovery

CCIX devices that conform to CCIX 1.0a [1] are discovered and managed as a collection of PCIe endpoints. CCIX components in these devices, and their capabilities, are thus exposed to software as a set of CCIX CCSR structures resident in a CCIX-specific Designated Vendor Specific Enhanced Capability (DVSEC) structure in the PCIe extended capability space of a PCIe function within the device. This allows the CCIX endpoint to be discovered as a PCI function with the CCIX-specific DVSEC [5]. A CCIX device is considered as a collection of CCIX endpoints. The CCIX-specific DVSEC carries a CCIX Consortium Identifier (CCID) that applies globally to all specification compliant CCIX devices. These structures are used by system software to:

- Discover presence of CCIX devices in the system
- Discover which CCIX components are present in each CCIX device, and their properties
- Discover memory pools implemented by each CCIX device, and their properties
- Configure the discovered CCIX devices
- Establish routing and resultant system physical address map (G-SAM)
- Create necessary description tables and provide them to an operating system
- Enable the CCIX coherent network
- Detect and handle errors pertaining to CCIX devices and components

## 2.8.2 Transport

PCIe also functions as the transport layer for CCIX. On devices that are compliant with CCIX 1.0a [1], a CCIX port resides above a PCIe port, such that the PCIe port operates as the transport layer. To this end, a CCIX device must implement an architected transaction layer that provides support for carrying CCIX messages over PCIe in the form of PCIe TLPs, as depicted in Figure 15.

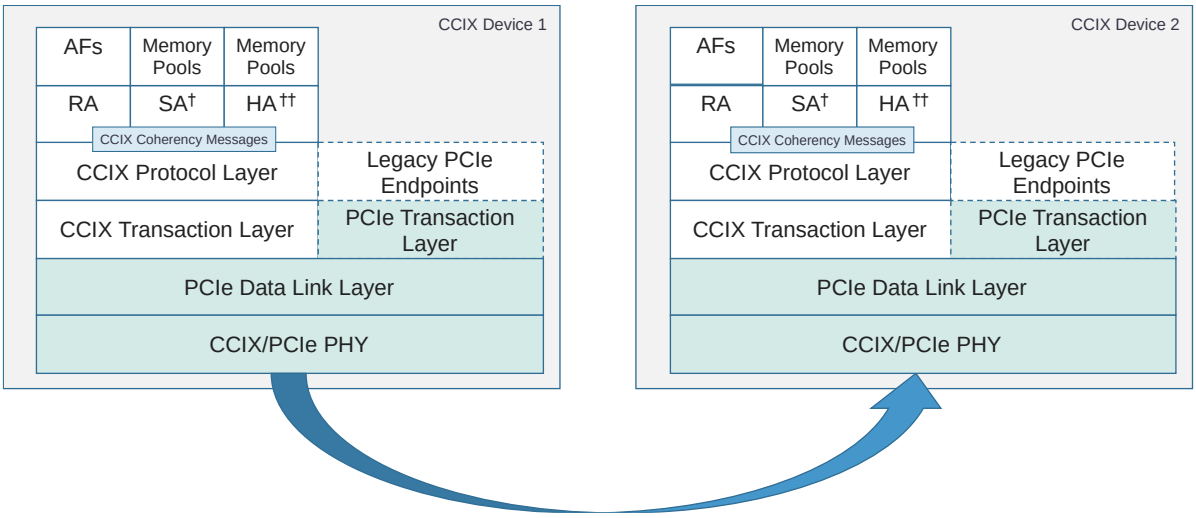



**Figure 15: CCIX over PCIe Transport**

## 2.8.3 AFC and PCIe

AFCs might be mapped to PCIe functions and then managed using a framework driver based on PCIe support in the OS. The framework driver might then be able to manage one or multiple AFCs, again through the same framework. The driver can also assign AFTs within an AFC to user-space processes using PCIe Process Address Space Identifier (PASID).

## 2.8.4 Virtualization

An AF may support virtualization by meeting the following requirements:

1. AFCs within the AF are capable of being assigned to a Virtual Machine (VM), and directly managed by that VM thereafter.
2. AFTs within an AFC mapped to a VM are capable of being mapped into the virtual address space of user-space processes running in that VM.
3. The AFCs must support the ability to be reset across (re)assignments. This may be necessary from a virtualization and security standpoint.

For CCIX systems using PCIe as the transport, the AF may support PCIe SR-IOV. In this scheme, AFC0 is implemented as a Physical Function (PF), while the other AFCs might be implemented as Virtual Functions (VFs) associated with that PF. PCIe SR-IOV, in conjunction with PASID and PCIe FLR, thus suitably equips the AF for virtualization support.

# 2.9 RA to AF Binding

An RA operates as the caching agent for one or more integrated AFs within a CCIX device. RAs and AFs present themselves as endpoints within the device. There could be a *group* of RAs that serve as the caching agents for one or more AFs. Together, the group of RAs, and the AFs that they service, are organized as a logical cluster.

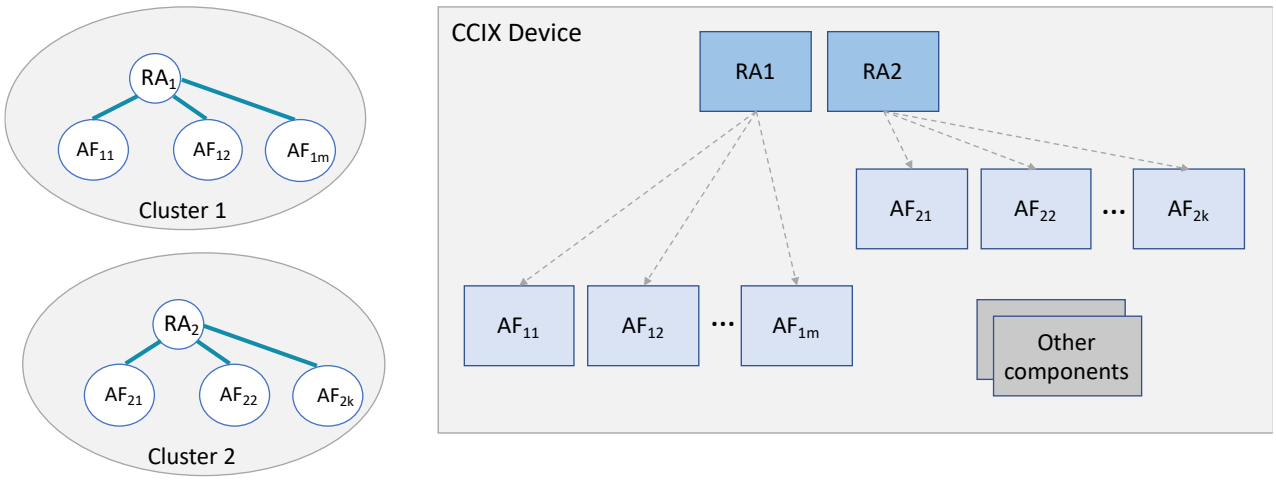
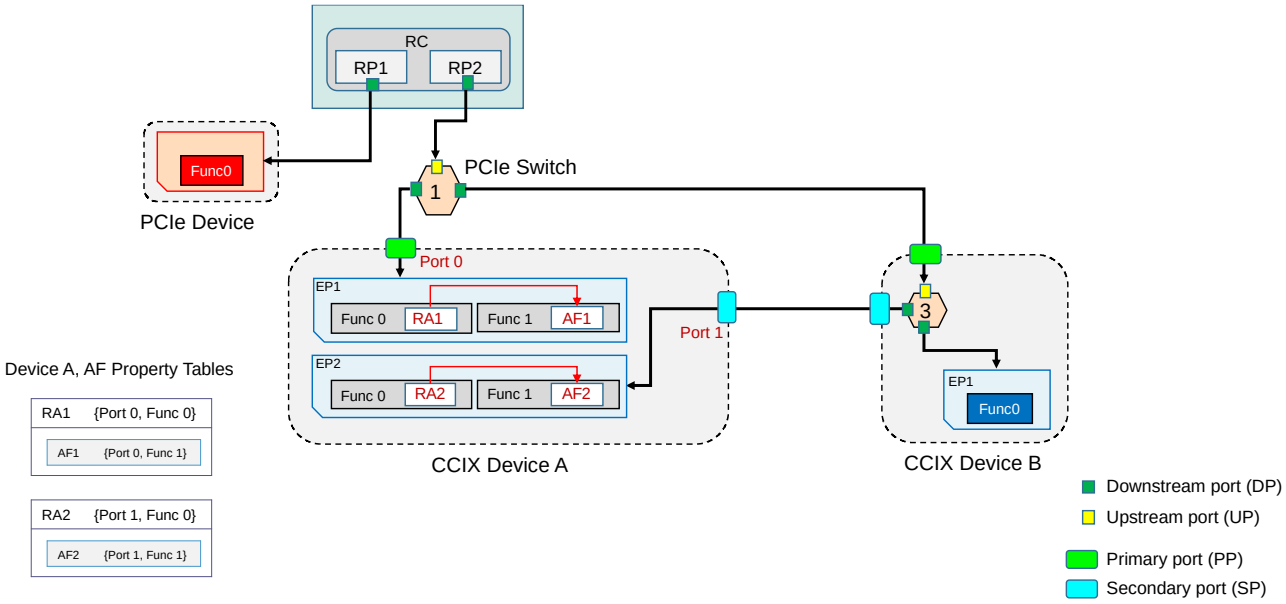



**Figure 16: Example CCIX Device with RAs and AFs and associations thereof**

The RA to AF binding is illustrated in Figure 16. The binding is established and presented as the AF Property Table in the primary port of a CCIX device [1].

## 2.9.1 PCIe-based Designs

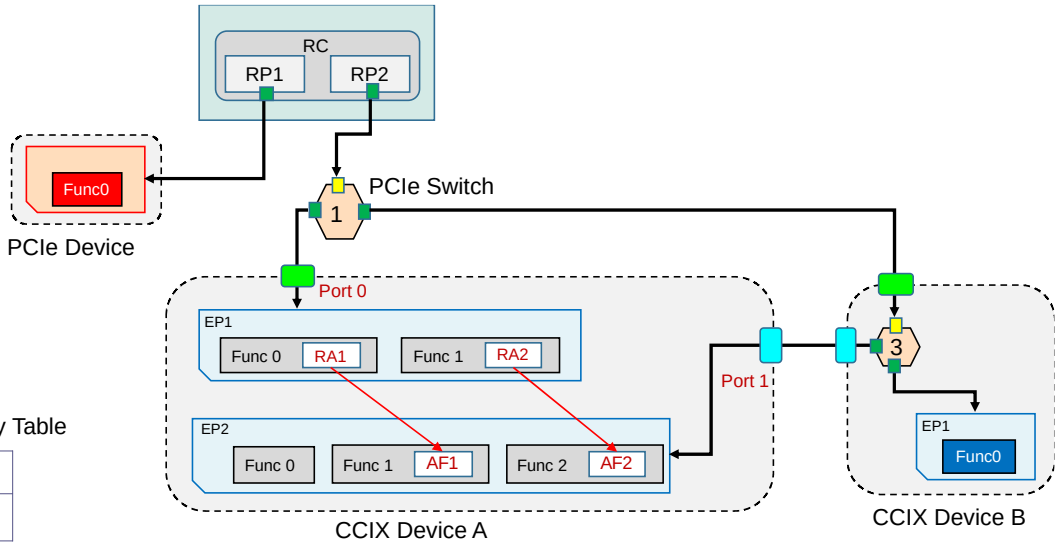
In a physical design using PCIe as the underlying implementation, the RA to AF binding may be established as logical links between PCIe functions on CCIX ports. The tuple {CCIX port, PCIe Function} uniquely identifies an AF in the device. The tuple {CCIX port, PCIe Function, RA CCSR Offset} uniquely identifies an RA in the device. The AF Property CCSR structure provides mapping between such tuples [1]. It is recommended that AFs are located on non-PF0 functions. This allows PF0 to be available for port/device wide properties.




Device A, AF Property Tables

| | |
|---|---|
| RA1 | {Port 0, Func 0} |
| AF1 | {Port 0, Func 1} |

| | |
|---|---|
| RA2 | {Port 1, Func 0} |
| AF2 | {Port 1, Func 1} |




Device A, AF Property Table

| | |
|---|---|
| RA1 | {Port 0, Func 0} |
| AF1 | {Port 1, Func 1} |

| | |
|---|---|
| RA2 | {Port 0, Func 1} |
| AF2 | {Port 1, Func 2} |

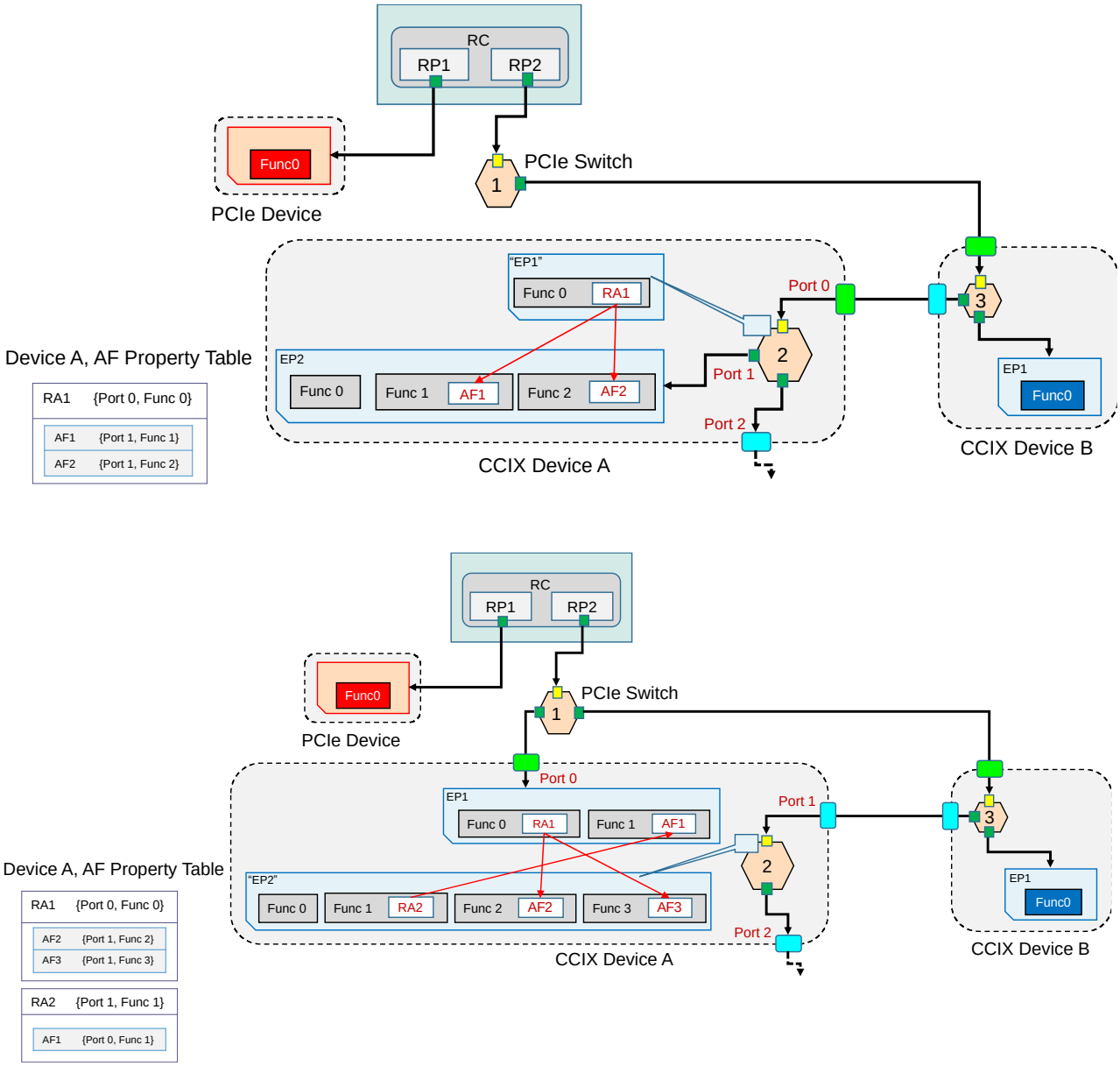



**Figure 17: Examples of RA and AF Bindings on PCI-e based implementations**

As illustrated in Figure 17, a variety of implementations are possible. Software must always consult the AF Property Table to obtain the topological arrangement, and thus be able to comprehend all topologies possible. Software should refrain from making any implicit assumptions regarding the ordering of the RAs and their associated AFs.

## 2.9.2 AF Management Framework using PCIe and SR-IOV

The AF Management framework for PCIe based implementations allow management of AFs using PCIe semantics. This is depicted in Figure 18.

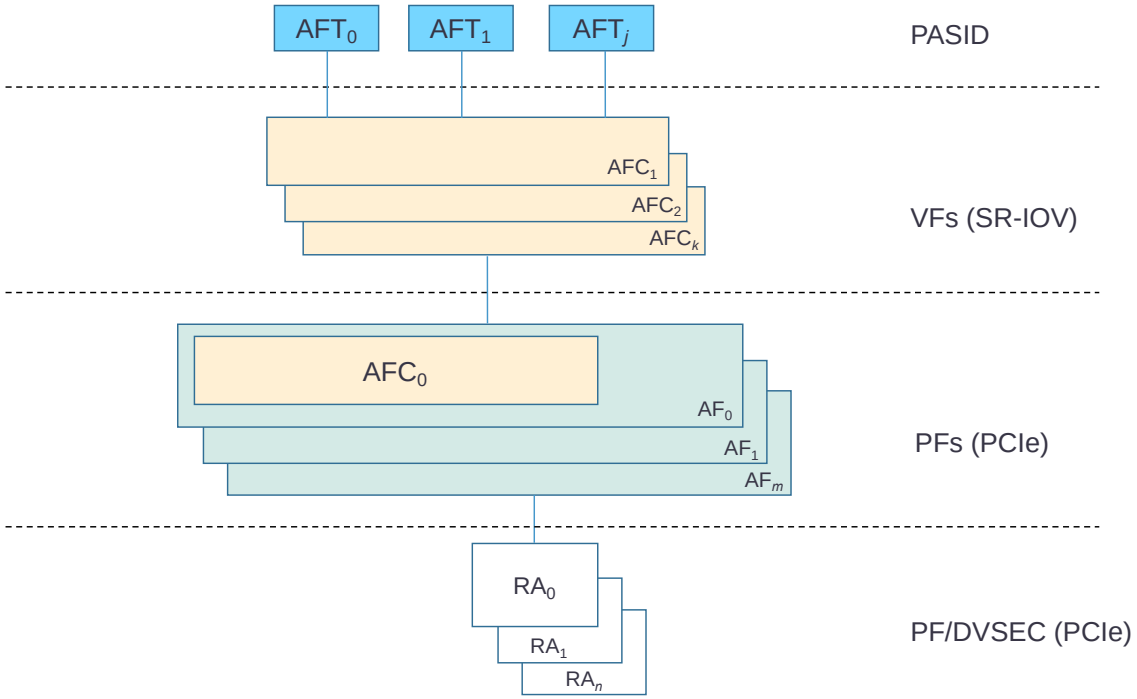



**Figure 18: AF Management Layers on PCIe-based Implementations**

A detailed overview of the AF Management Framework is provided in Section 7.3.

# 2.10 Software Architecture Partitioning

Figure 19 depicts a broad overview of the expected software architecture of a CCIX enlightened operating system kernel, and a CCIX enlightened hypervisor.

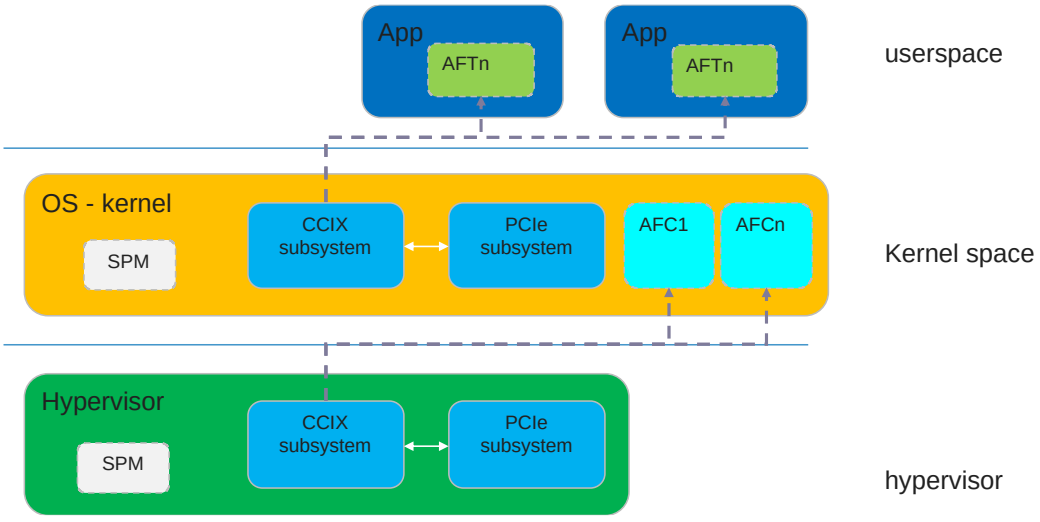



**Figure 19 CCIX OS software architecture overview**

It is expected that an CCIX enlightened OS kernel will contain a generic CCIX subsystem with the following responsibilities:

- AFC assignment to applications
- RAS handling of CCIX specific errors
- Describing CCIX components and their capabilities to user space
- Power management of the CCIX subsystem

The OS CCIX subsystem consists of entirely generic CCIX related code. There are no accelerator-specific drivers or platform specific extensions included in this code. CCIX 1.0a [1] is built on top of PCI Express, and therefore the CCIX generic subsystem is strongly connected to the PCIe subsystem. A hypervisor kernel has much the same architecture, and provides CCIX device assignment to virtual machine guest operating system.

# Chapter 3. Boot Firmware

The boot firmware is responsible for discovery and configuration of CCIX devices and components, and for describing CCIX specific features to the operating system. This section describes the generic CCIX firmware architecture and boot flow.

The boot firmware discovers CCIX devices in the system during PCIe enumeration, as explained in [1], and in Section 2.8.1. Following this explanation, a device that is compliant with CCIX 1.0a [1] advertises itself by incorporating a CCIX-specific DVSEC in the PCIe configuration space of each of its implemented PCIe endpoints. The CCIX-specific DVSEC contains CCIX CCSR structures [1]. The CCIX DVSEC capability is assigned the CCIX-specific CCIX Consortium Identifier (CCID). The physical connection between CCIX devices within the PCIe hierarchy can be a direct PCIe bus, or through an intermediate PCIe/CCIX switch. As a result, CCIX devices appear to the boot firmware as discrete devices interspersed with legacy PCI devices within a legal PCI bus hierarchy, as illustrated in Figure 20. Thus, from a firmware perspective, a CCIX network appears as being effectively overlaid on top of an underlying PCI tree. This arrangement allows the boot firmware to discover the CCIX topology during PCI enumeration.

Because of the overlay nature of the CCIX network, CCIX topologies aren't restricted to the hierarchical tree structure of the underlying PCIe infrastructure. Instead, CCIX devices can be interconnected in various topologies such as meshes and fully-connected. To achieve this, a CCIX device must support an integrated PCIe or CCIX switch to provide at least one upstream port, one or more local CCIX components, and at least one downstream port that serves as a CCIX port to interconnect with a neighboring CCIX device. The integrated PCIe switch and its endpoints are designed to conform to the hierarchical arrangement of a legal topology and yet provide the multi-port, graph-like topology of a CCIX network. CCIX endpoints are discovered as distinct PCIe endpoints. A CCIX endpoint can carry one or more CCIX components. Furthermore, the PCIe endpoint could be a multi-function device, and these functions could in turn house CCIX components. that can be assigned unique BDFs during the PCIe enumeration process. Hardware designers can use pin strapping to configure the internal CCIX switch ports to produce a PCIe compliant topology where each CCIX component now becomes a valid PCIe endpoint. This point is also illustrated in Figure 20.

It may be noted that CCIX is not restricted to PCIe, and can be built on top of *any* underlying transport technology. If different, then the boot firmware must initiate CCIX discovery during the bus initialization process of the specific bus/transport technology on which the CCIX network is overlaid.

The boot firmware can discover CCIX topology based on PCIe device bus number assigned to each CCIX device during the PCIe enumeration process, as already specified. An interesting aspect of CCIX device discovery is that the boot firmware may visit a particular CCIX device with multiple ports multiple times during the PCIe bus enumeration process. This is because each of the CCIX ports of the device represents a unique branch of the PCIe tree that is accessible via an independent path, although the PCIe bus enumeration process itself may be unaware of this fact. The boot firmware must therefore have some means of recognizing a device that has already been visited. Once CCIX devices and their CCIX components are discovered and enumerated, the boot firmware then applies routing algorithms to the topology in which these devices are arranged. The goal of the routing algorithms is to guarantee routing of all CCIX protocol packets within the CCIX subsystem.

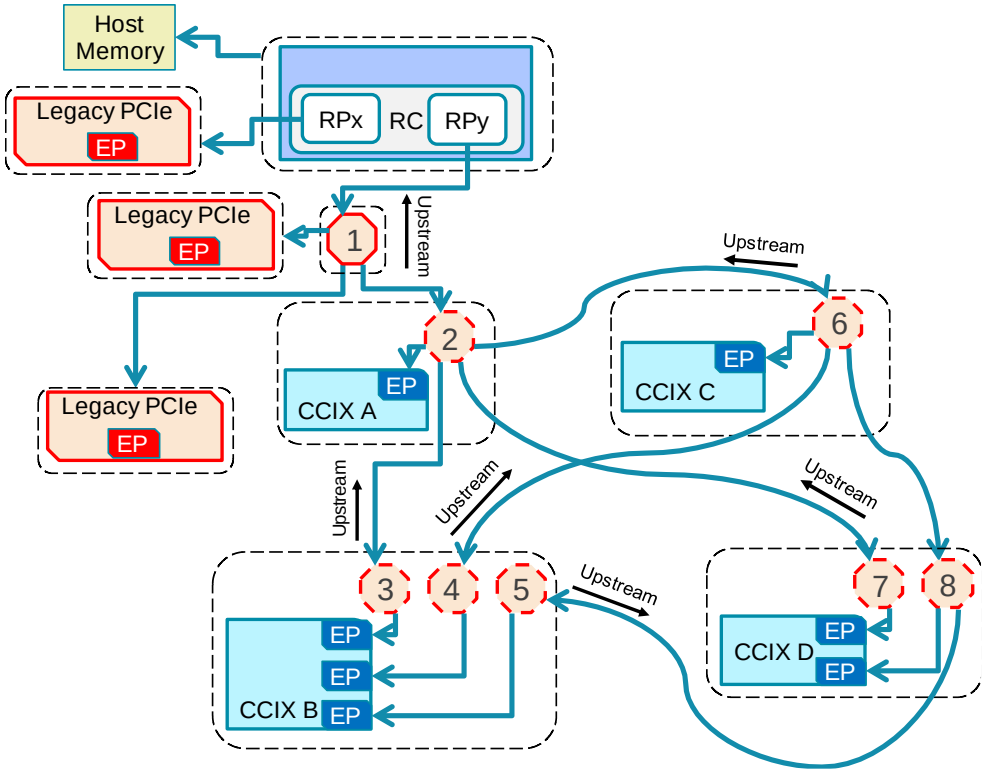



**Figure 20: An example CCIX-enabled System**

The routing algorithms are targeted at accomplishing both goals. The finalized routing paths are then configured in the RSAM, HSAM and IDM tables for each CCIX device.

# 3.1 CCIX Configuration Addressing Scheme

The CCIX CCSR (Control, Capability and Status Register) region, and addressing scheme thereof, is the foundation on which the software-based discovery and configuration process is built. The addressing scheme is intended to serve as an abstraction for the underlying transport technology. The scheme provides a generic, logical addressing scheme that is agnostic of the transport. This transport agnosticism aids in relegating the transport semantics to a thin layer in software, while generalizing the bulk of the software/firmware support for CCIX. This improves overall portability and scalability.

The addressing scheme relies on the fact that the CCIX component structures and registers thereof, generic data structures such as the IDM and *x*SAM tables are all grouped into a distinct configuration address space called the CCIX CCSR region or space. The CCSR region may then be located within the configuration address space of the underlying transport technology when CCIX is based on that transport technology. A given CCIX system may then have multiple such CCSR regions. A CCSR region holds one or more CCSR structures (e.g. IDM tables, RA CCSR structures, HA CCSR structures etc.) as described in [1].

From a software discovery standpoint, these regions may be viewed as logical address spaces. In this logical addressing scheme, the CCSR structures are located within a *CCSR Address Space* (CAS). A group of CAS spaces may be located within a *Local Address Space* (LAS). Local addresses are logically grouped within a *Global Address Space* (GAS).

With such a scheme in place, a CCIX component is discovered using a {GAS, LAS, CAS, CCSR Structure} tuple.

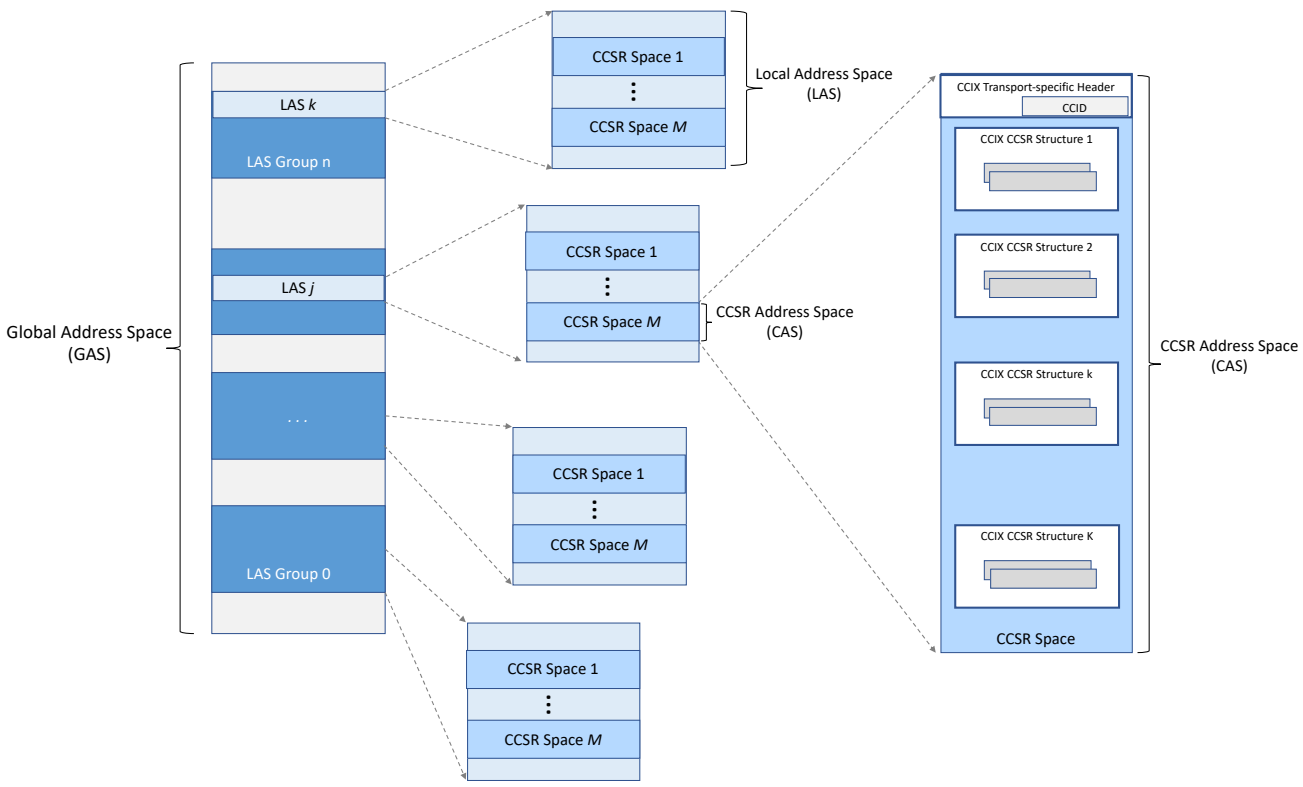



**Figure 21: CCIX Address Spaces**

CCIX-specific identifiers such as the CCID are used to discover CCIX CAS regions within the LAS. The scheme is outlined in Figure 21.

A group of local address spaces can be associated with a CCIX *endpoint*. The CCIX endpoint is identified using its *CCIX port*. The CCIX port identifies a group of related LAS regions within the GAS space. The physical characteristics of the CCIX port are described in Section 2.6.2.

An individual LAS region within a CCIX endpoint is identified by which *CCIX Function* it resides on. The LAS region of a CCIX function can have multiple CAS regions associated with that CCIX function. Individual CAS regions within a given LAS region are identified via their headers, which are transport-dependent.

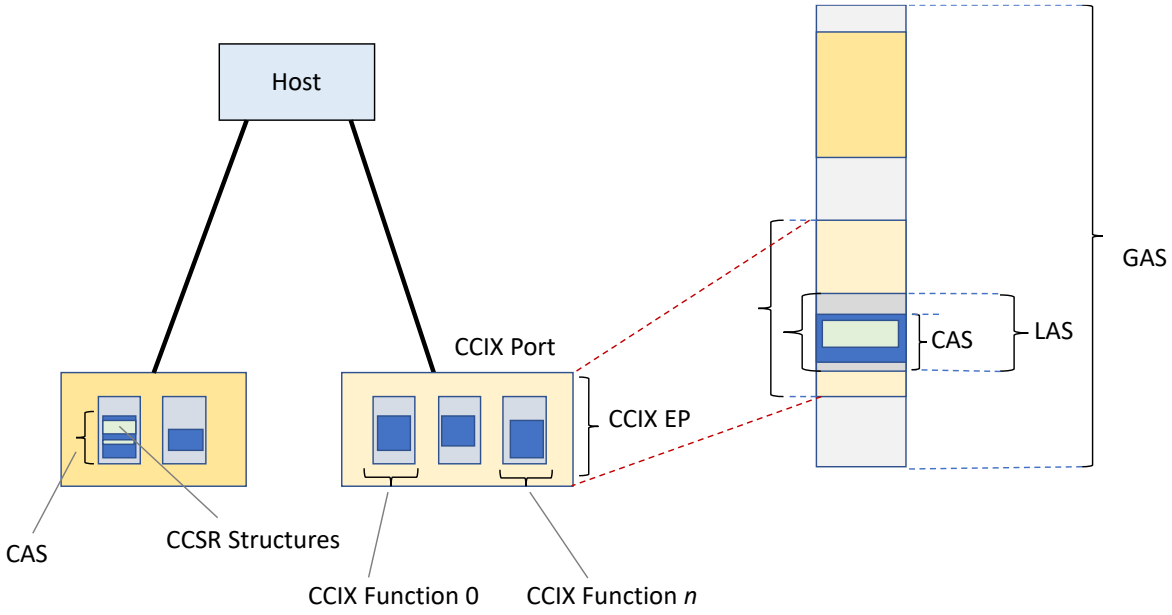



**Figure 22: CCIX Logical Ports and their relation to CCIX Config Addresses**

## 3.1.1 CCIX Configuration Addressing – Mapping to PCIe

When CCIX is implemented above PCIe transport, the PCIe ECAM space in host memory map is representative of the CCIX GAS space. The CCIX endpoint is equivalent to a PCIe endpoint and the CCIX function is the same as a PCI function. The CCIX function's LAS region is mapped to the PCIe configuration space of the equivalent PCI function.

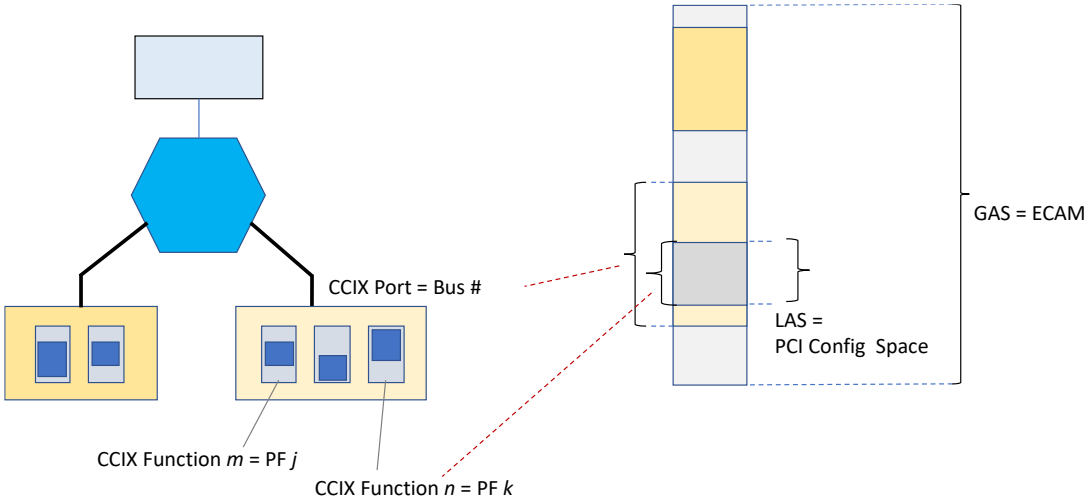



**Figure 23: Mapping CCIX Configuration Addressing Scheme to PCIe**

Figure 23 provides an illustration of these points. The CAS spaces are represented by the PCIe DVSEC structures reserved for CCIX Transport and Protocol Layers, as described in [1]. Since there are two distinct PCIe DVSEC structures – Transport and Protocol, a LAS region might contain up to two CCIX CAS regions.

## 3.1.2 CCIX Configuration Space Addressing – An Example

As explained in Section 3.1, the CCIX configuration space and addressing scheme is fundamental to the discovery and configuration process.

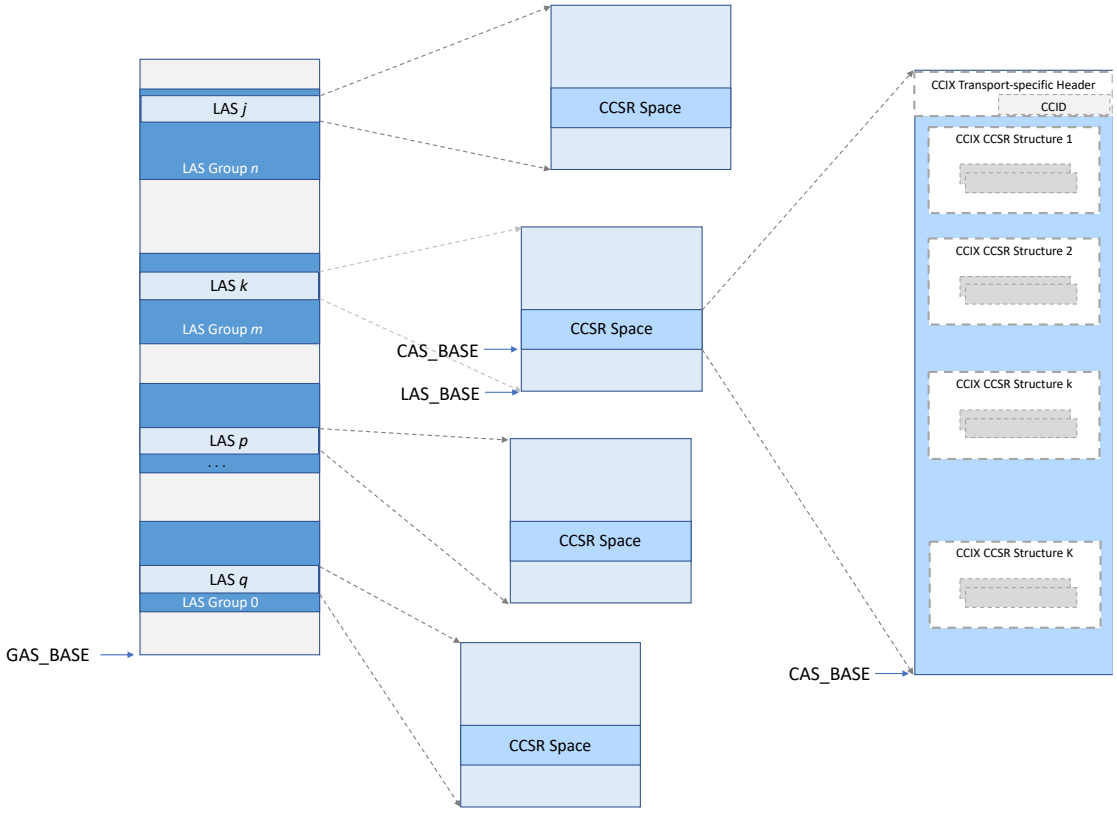



**Figure 24: Discovery and Configuration using CCIX Configuration Space**

As outlined in Figure 24, the discovery of CCIX endpoints requires knowledge of the following information:

- the offset and size of the GAS region in system address map (`GAS_BASE`, `GAS_SIZE`)
- The offset and size of the LAS region within GAS where the specific CCIX endpoint resides (`LAS_BASE`, `LAS_SIZE`)
- The offset of the CAS region with LAS where the CCIX endpoint resides (`CAS_BASE`). CAS size is supplied by the header within the CAS region.

For PCIe As an example, if a system has a CCIX Root Complex that is mapped to the ECAM address range beginning at offset `0x40000000` in system address map, and the ECAM space supports `16` buses, then:

`GAS_BASE` = `0x40000000`

`GAS_SIZE` = `16*32*8*4096 = 1000000h`

If, furthermore, the system has a single-port CCIX device with only one function (PCI function 0), which is an RA, and the CCIX device is located at B12:D0:F0, then,

LAS_BASE = GAS_BASE + 12*32*8*1000h = GAS_BASE + C0000h

LAS_SIZE = 1000h (always the case for PCI functions)

Finally, if the CAS region begins at offset 0x200 from the beginning of the PCI config space of function 0, then,

CAS_BASE = 200h

Finally, if the RA CCSR is located at offset 0x30 within the CAS region, then the actual physical address of the RA CCSR becomes:

CCSR_OFFSET:

= 30h in CAS space

= (CAS_BASE + 30h) in LAS space

= (LAS_BASE + CAS_BASE + 30h) in GAS space

= (GAS_BASE + LAS_BASE + CAS_BASE + 30h) in physical address space

= 0x400C0230

# 3.2 CCIX Bus Scan

The CCIX bus scan involves walking the GAS space in order to locate CAS spaces. Knowledge of the following information is a pre-requisite for this bus scan process:

- GAS_BASE
- CAS header with CCID Identifier

Implementations may choose to relegate the CCIX bus scan process to the underlying native transport.

## 3.2.1 CCIX Bus scan over PCIe Transport

On PCIe-based implementations, the CCID identifier is placed in the CCIX-reserved PCIe DVSEC headers as described in [1], and the LAS regions are a fixed 4KiB in size, to achieve parity with PCIe. The CCIX bus scan is then effectively equivalent to the standard PCIe bus scan.

Each CAS region found during the bus scan constitutes the discovery of a CCIX endpoint. Implementations may choose to either perform a PCIe bus walk on their own, or rely entirely or partially on the underlying native PCIe bus scan support.

# 3.3 Host Discovery

It is strongly recommended that host implementations follow the well-defined CCSR layout that is described in [1]. For such implementations, the host is discovered as just another native CCIX device with CCIX endpoints. Host implementations that deviate from the standard layout require alternate discovery mechanisms. A key motivation of the generalized CCIX configuration and bus scanning abstraction process is to present a standard view of the host implementation to software, as illustrated in Figure 25.

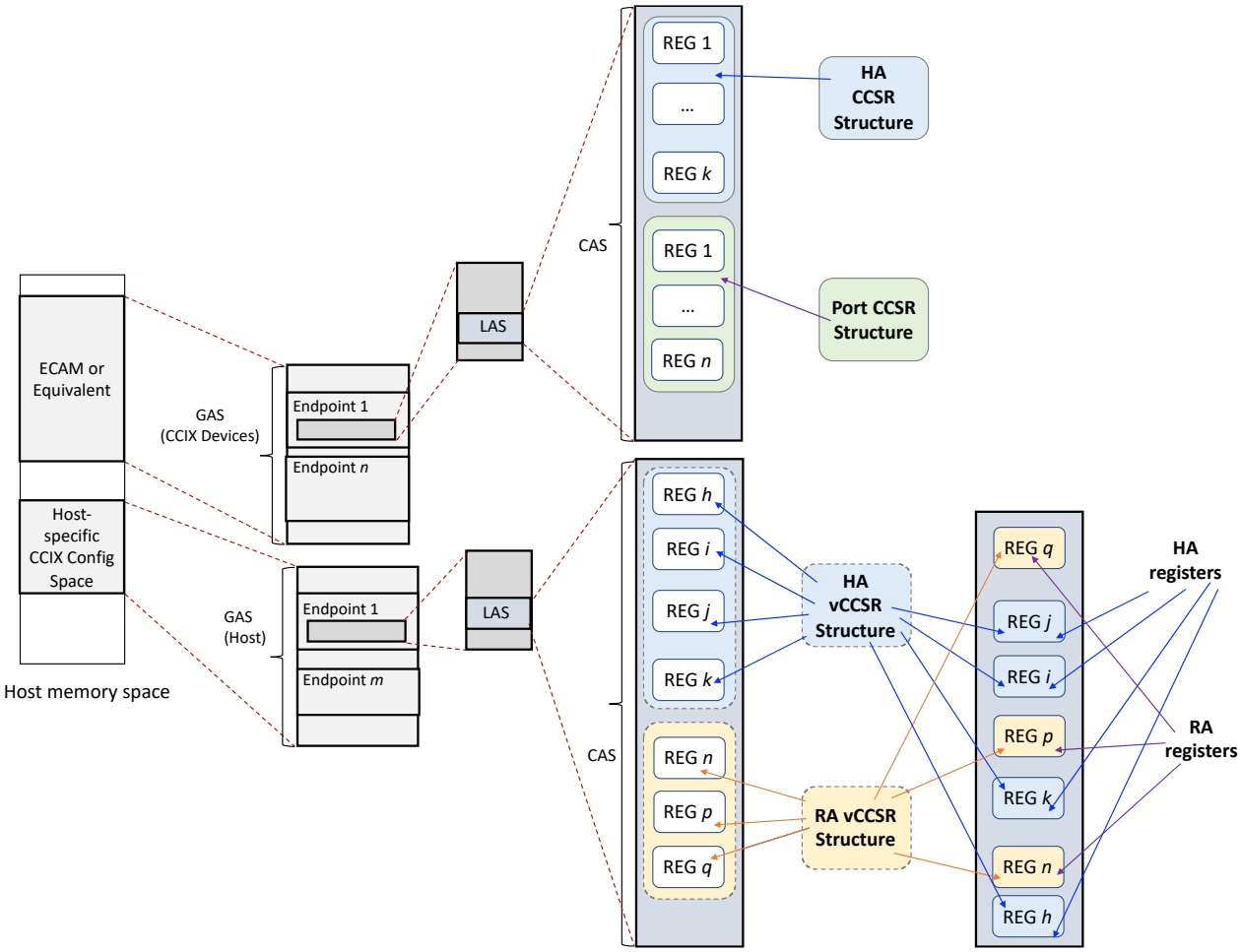



**Figure 25: Abstraction of Host CCIX Configuration Registers as virtual CCSR structures in a virtual CAS space**

To this end, the host may be represented as a native CCIX endpoint or device using a mapper that provides a front-end standard view to software, while abstracting the host implementation details. This front-end may be composed of a number of virtual CCSR structures that are laid out in the generic manner described in this section. Figure 25 provides an example of this mapping functionality. This mapping thus allows a common discovery and configuration process to apply uniformly to both host-resident and native CCIX endpoints. The mapper may be implemented in software or hardware.

# 3.4 Boot Flow

The CCIX boot process involves the following objectives:

- CCIX Device Discovery
- Configuration of CCIX devices and components
- Configuration of CCIX addressing decoding and routing
- Extension of the G-SAM with CCIX memory
- Configuring the host's CCIX specific properties

- Describing properties of the CCIX subsystem to the OS
- Description of the G-SAM to OS

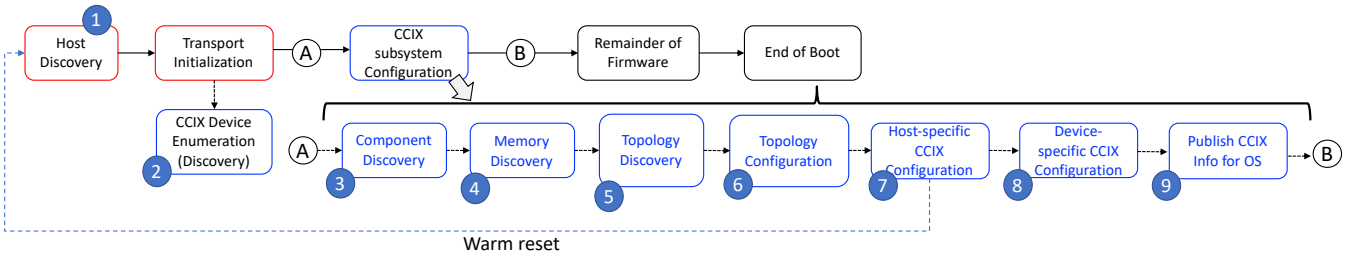



**Figure 26: Boot Firmware Flow (high-level overview with boot stages highlighted)**

As illustrated in Figure 26, the CCIX boot process involves a series of distinct boot stages:

- Host Discovery (Stage 1)
- CCIX Device Enumeration (Stage 2)
- CCIX Component Discovery (Stage 3)
- CCIX Memory Pool Discovery (Stage 4)
- CCIX Topology Discovery (Stage 5)
- CCIX Topology Configuration (Stage 6)
- Host-specific CCIX Configuration (Stage 7)
- CCIX Device-specific Configuration (Stage 8)
- Publishing CCIX Information to OS (Stage 9)

The boot flow highlights the Host Discovery and Transport Initialization steps in red, to point out that these steps are a prerequisite for the CCIX booting process.

There are two distinct paths – cold and warm. In the cold reset path, the host-specific initialization stages may be CCIX-agnostic and configured prior to the discovery of CCIX resources. In certain cases, any host reconfiguration following CCIX resource configuration may require a warm reset path to take full effect. Stage 7 thus includes an optional warm reset, that causes the boot flow to follow the warm reset flow outlined above in Figure 26. Examples where a warm reset may be required for reconfiguration of the host include:

- Updating host-specific decoders to accommodate SAM windows for CCIX memory ranges.
- Enabling cacheability properties in the host for these memory ranges.

Depending on implementation, the boot firmware may choose to skip some of the stages in the warm reset path, e.g. if sufficient configuration has already been performed and preserved during the preceding cold reset path.

Figure 27 provides a more detailed view of the boot process. The darker boxes indicate boot steps involving CCIX firmware, or outputs of the CCIX firmware.

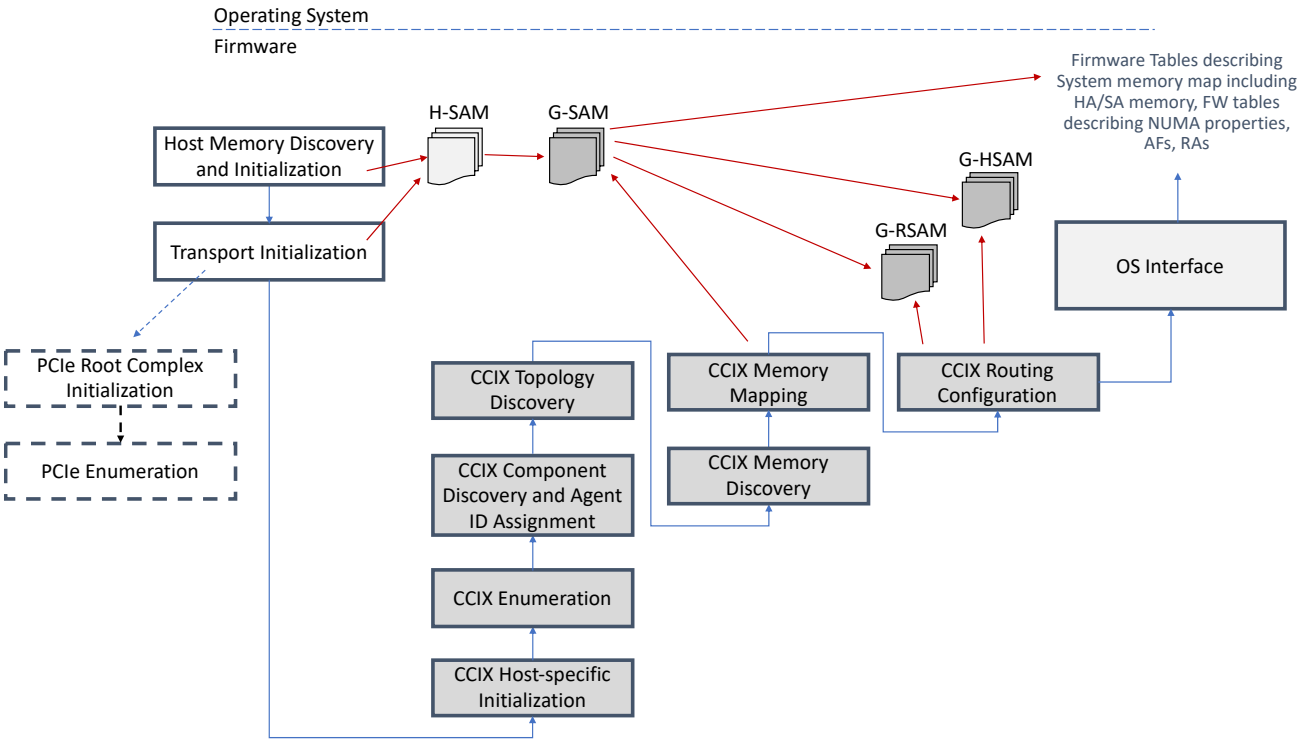



**Figure 27: CCIX 1.0a Firmware Flow (detailed)**

Since CCIX 1.0a [1] implementations are based on PCIe transport, the Transport Initialization step in Figure 27 may include initialization of the PCIe Root Complex and some preliminary bus number assignments (e.g. to Root Ports), as a prerequisite to CCIX discovery and configuration. The exact steps and amount of initialization required are implementation specific.

## 3.4.1 Host Discovery (Stage 1)

All host-specific and implementation-specific initialization is performed during this stage, which is mostly CCIX-agnostic. During this stage, the key steps relevant to CCIX are:

- Configuration of CCIX-aware Root Complexes and Root Ports in the host. For example, programming of a default ECAM for the PCIe hierarchy.
- Creation of the Host-SAM to describe host-specific memory regions.

At the end of this stage, the system is fully prepared for CCIX discovery.

## 3.4.2 CCIX Device Enumeration (Stage 2)

### 3.4.2.1 Discovery of CCIX Endpoints

During PCIe enumeration, the boot firmware discovers a CCIX endpoint via an examination of the PCIe extended capability ID, and DVSEC Vendor ID, in the endpoint's PCIe DVSEC header:

- PCI Express Extended Capability ID = 0x23

- DVSEC Vendor ID = CCID

All CCIX-compliant devices must carry the CCID, as above. This aids in generic discovery of CCIX devices during system boot.

### 3.4.2.2 Device ID Programming

All CCIX endpoints reside on CCIX ports. A CCIX device may have several ports, and, as such, may be discovered from any port. Multiple ports on the CCIX device may thus claim primary port capability. At the same time, some ports may always be secondary. An example of a device with both primary-capable and secondary-always ports is illustrated below in Figure 28. In this example, ports 1 and 2 are primary capable because they are upstream ports that are reachable from an external device through the usual PCI bus walk when this device is a PCIe-based CCIX 1.0a compliant device [1]. Ports 3 and 4 are secondary ports because they can only be visited internally from the device via one of its primary ports.

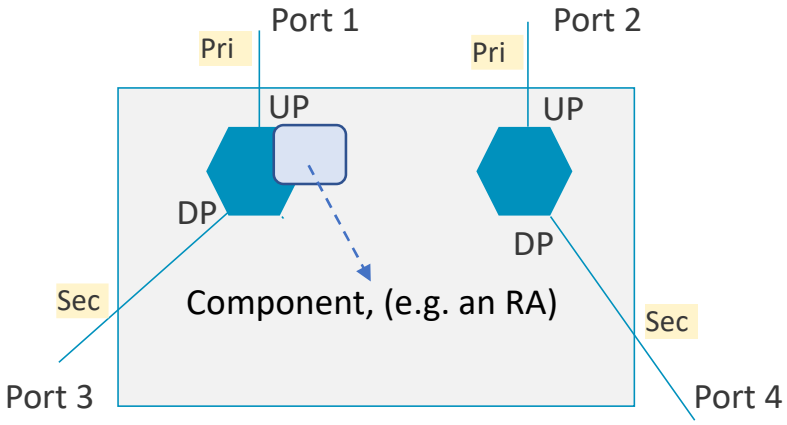



**Figure 28: Primary and Secondary Ports**

The boot firmware must write a non-zero Device ID into the first primary port of a CCIX device that it discovers during bus walk. This prompts the device hardware to reflect that written Device ID on to the remaining CCIX ports of the device, as shown in Figure 29. The first port where the Device ID is programmed must be primary-capable, and the firmware must configure it to be the device-wide primary port. The reflected Device ID on the other ports of the device guarantees that the firmware is able to associate each subsequently visited CCIX port of the device with that device. In other words, the Device ID binds a CCIX device to its children ports. When the firmware encounters a port on the same device subsequently, it must examine the Device ID. If the Device ID matches a previously written port's Device ID, then the parent device was already visited and hence the association of the current port with that device is established. All ports that are discovered subsequent to the primary port must be programmed as secondary ports, even if they are primary-capable. This guarantees that there is one and only one location in the device that is elected to host the device-wide common configuration structures. Common capabilities and control structures are always located on PCIe Function 0 on a primary-capable CCIX port. More details are available in [1].

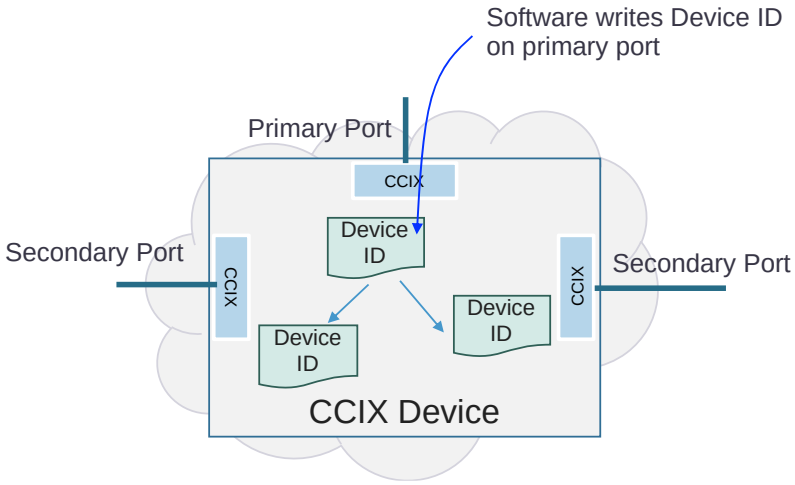



**Figure 29: Programming Device ID on a multi-port CCIX Device**

The high-level flow is illustrated in Figure 30.

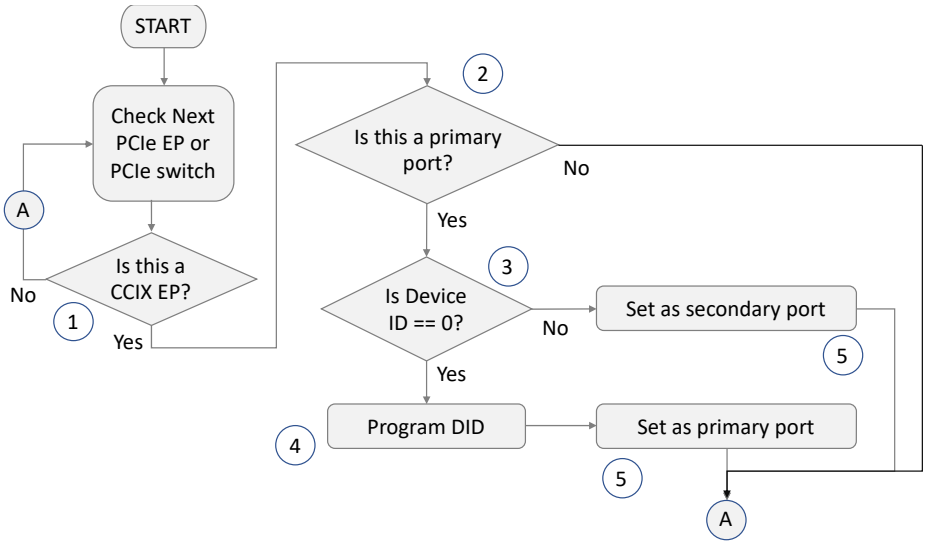



**Figure 30: Device ID Programming**

The EP or switch in Figure 30 refers to a PCIe EP or switch in a CCIX 1.0a [1] compliant implementation which is based on PCI Express. As such, the EP is any local endpoint that provides the LAS space as described in Section 3.1.

The various checkpoints in this flow are explained below:

| Checkpoint | Description |
|---|---|
| 1 | A CCIX EP implemented as a PCIe EP or switch must include a CCIX Protocol DVSEC structure in its extended capability space with DVSEC Vendor ID = CCID |
| 2 | A CCIX EP is a primary port if: |

| | |
|---|---|
| | i) Its on PCIe Function 0 if implemented over PCIe transport<br>ii) It has field MultiPortDevCap Bit 1 set to 1 in its ComnCapStat1 register. |
| 3 | The CCIX Device ID, once programmed, is returned in the DevIDStat field of the ComnCapStat1 register. |
| 4 | The CCIX Device ID is programmed via the DevIDCtrl field in the ComnCtrl1 register. |
| 5 | The CCIX port is set to primary or secondary by setting the PrimaryPortEnable field in the ComnCtrl1 register to 1b or 0b respectively. |

#### 3.4.2.3 Device Readiness

The boot firmware must always check for the device readiness status before determining other properties of the device or attempting any device configuration. The Device Readiness indicator is present on the primary port. If this is not set, then it means the device is still initializing its internal state, e.g. setting up memory pools.

The device readiness interface, as well as the interface for programming the Device ID, is described in detail in [1].

### 3.4.3 Component Discovery (Stage 3)

As described in Chapter 2, agents on a CCIX device communicate with agents on other CCIX devices using the CCIX protocol. The CCIX protocol layer performs routing of CCIX packets from source agents to destination agents using unique Agent IDs. Agent IDs are described in Section 2.6.4.1 . Each device has a dedicated IDM (ID Map) table that maps Agent IDs to unique egress ports on that CCIX device.

The boot firmware is responsible for assignment of Agent IDs for each agent that it discovers. The Agent ID assignment follows principles outlined in [1].

Components can be located on any port (primary, secondary or tertiary), as illustrated in Figure 31.

It is possible for the host to contain one or more components. The Agent ID assignment process should be aware of such host-resident components in addition to regular agents on CCIX devices.

CCIX components: transport, common (device-wide), port and link, have to be located in a primary or secondary port BDF with function 0. A protocol component must always be present on every PCIe function where there is another CCIX component presented. The agent components such as HA, RA and SA, can be located in any BDF of a CCIX device. The port and link components have to be bundled together. The control registers for a common component have to be located on a primary port with function 0.

All components with the same CCIX Device ID on the BDF with function 0 belong to this CCIX device.

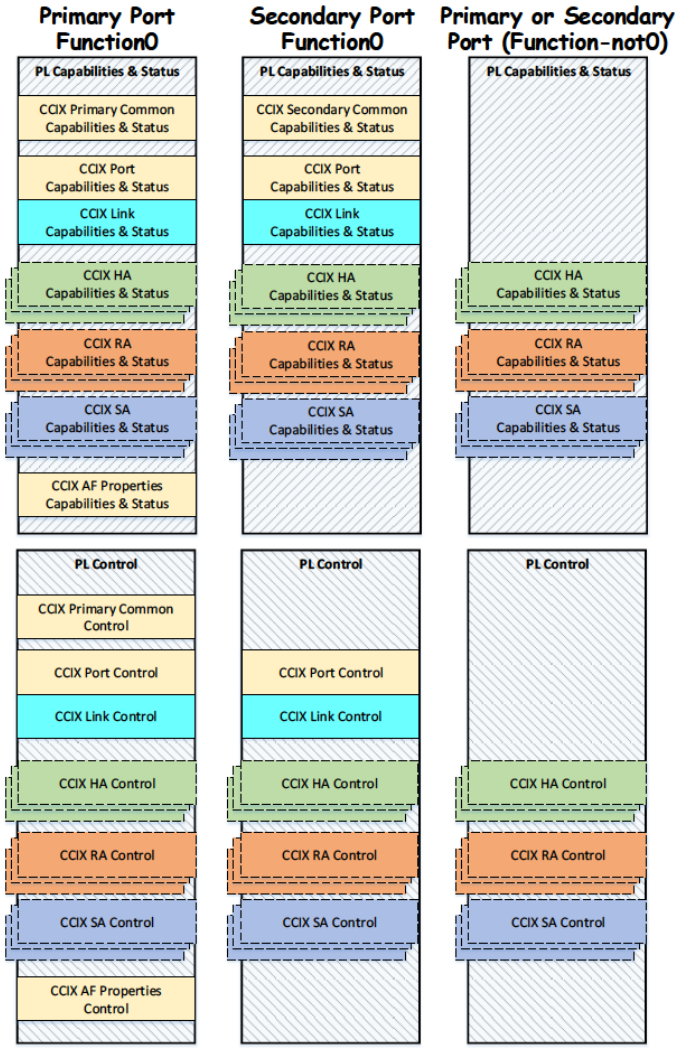



**Figure 31: Location of CCIX Component Structures when LTAS space = PCIe Configuration Space**

## 3.4.4 Memory Pool Discovery (Stage 4)

In this stage, the boot firmware scans through the CCSR structures for each HA and SA discovered in the previous stage, to determine how many memory pools are implemented by that HA or SA. As described in [1], memory pools also advertise properties of the associated memory, such as:

- Memory type (RAM, device, non-volatile, memory hole, private memory etc.)
- Memory attributes (normal/device, cacheable/non-cacheable)
- For an HA, locality of memory (local to HA, extended memory on a remote SA)
- Memory pool size
- Memory pool alignment requirements

The firmware at this point collects all details of the memory pool into a software data structure, referred to as a memory pool descriptor table, for future consumption.

### 3.4.5 Topology Discovery (Stage 5)

After completing CCIX enumeration, the CCIX boot firmware must discover the CCIX topology for building up routing tables and establishing address decoding ranges for the memory pools. Topology discovery refers to the recognition of the manner in which the CCIX devices in the system are networked together. CCIX 1.0a [1] is suited for the following topologies – tree, ring/mesh, and fully-connected, as outlined in Section 2.5. Each of these topologies has well-defined algorithms for deadlock-free routing traffic through them. In this guide the term "optimal routing" is used to refer to deadlock free routing. Guidelines for optimal routing are outlined in Section 11.1.2. An implementation may choose to override the default notion of optimality by incorporating other implementation-defined constraints or conditions.

This guide describes these basic topologies and their discovery and configuration methods. Arbitrary topologies that are decomposable into one or more basic sub-topologies can also be supported by an application of the rules prescribed for the basic set. CCIX doesn't preclude support of yet other topologies, but their discovery and configuration procedure is beyond the scope of the current document.

Illustrative guidelines for routing are provided in Section 11.1.3. The rules are applied to a given system once its topology has been recognized in this stage. This application yields a set of routes within the CCIX subsystem for ID-routed and address-routed messages. These routes are then established by programming the IDM and RSAM/HSAM tables of each device on a route, as outlined in Section 2.6.5.

A CCIX system is constructed as one or more CCIX devices overlaid on a PCIe tree, each of which implements one or more ports. Each port implements one or more links that then represent the interconnection between the device and its peers. During PCIe enumeration, the boot firmware must derive a raw CCIX network from the PCIe subsystem. The raw CCIX network is just a graph of interconnected CCIX devices. The firmware must then recognize the topology of this raw graph based on pattern matching techniques. As an example, the CCIX system in Figure 20 is first reduced to a raw graph, and then recognized as a fully-connected topology using the illustrative example in Figure 32.

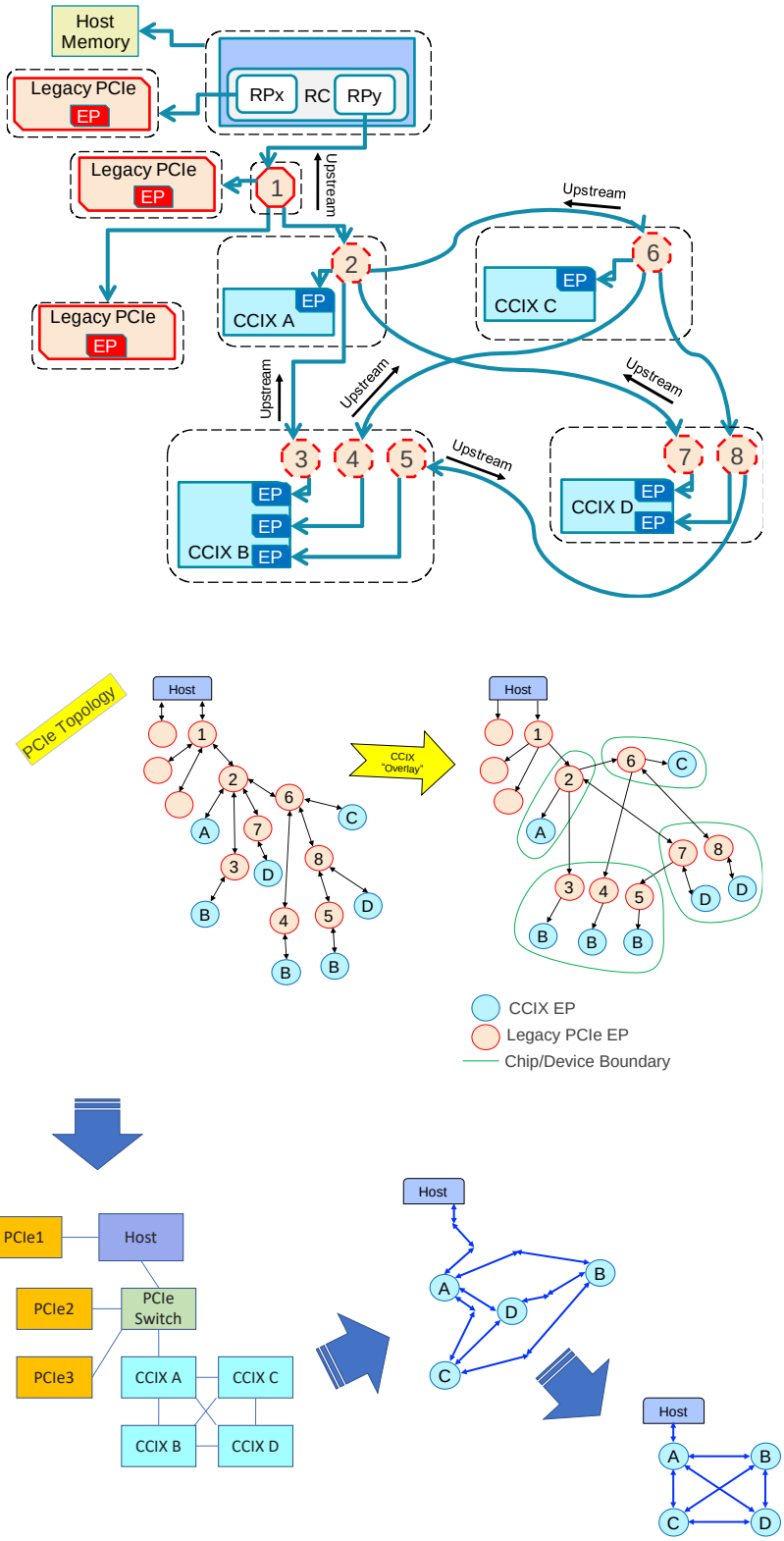



**Figure 32: Topology Discovery**

The boot firmware must perform topology discovery as follows:

1. Organize all discovered ports, links, agents and memory pools of a device into a distinct node of a graph.
2. Reduce all aggregated ports on that device into a single logical port.
3. Set each link of the node to represent an edge in the graph, where the link interconnects that node with a neighboring node.
4. Construct two separate sub-graphs in the above manner. The first sub-graph is derived by joining nodes that have at least one RA with nodes that have at least one HA. This sub-graph represents the RA→HA network. The second sub-graph is derived by joining nodes that have at least one HA with nodes that have at least one SA. This sub-graph represents the HA→SA network.
5. Analyze each of the sub-graphs to perform topology recognition. Section 11.1.3.2.3 provides an example algorithm for recognizing meshes.
6. For the RA→HA sub-topology, coalesce memory pools from each device into a single logical pool associated with the graph node. If an HA has memory extension, include memory from all associated SA devices, as illustrated in Figure 33. HA to SA associativity is established using implementation defined means.
7. For some implementations, optionally, perform a warm reset, if required, to allow for memory windows to be opened in the host's SAM to accommodate CCIX memory. This step might be required to make sure that cache coherency can be enabled for these memory windows at a system-wide level.
8. Apply routing algorithms to find the optimal path between each source node to destination node in the topology.

> **IMPLEMENTATON NOTE**:
> The optimal path may be defined using a variety of parameters. For example, for some systems, the optimal path between two nodes may be determined in terms of the *bandwidth* of each hop on the path instead of the *number* of hops. In this particular example, it could be that the aggregate bandwidth of a path with more hops might be greater than an equivalent path with less hops.

9. Combine logical memory pools on an optimal path from a given source node to a destination node to form a larger logical memory pool, if possible. Some examples of memory pool coalescing are provided in Section 11.1.3.

> **IMPLEMENTATON NOTE**:
> It may not always be possible to coalesce memory pools. For example, if a memory pool in the path pertains to non-coherent memory, then it may be a requirement to place that pool in a separate memory window in system address space, or combine it with a other non-coherent memory in a dedicated address window, resulting in overall improved decoding and handling of memory cacheability.

10. Repeat steps 4-9 until optimal paths for all source, destination node pairs have been determined.
11. Create SAM windows in the G-SAM for each logical memory pool obtained during step 9.
12. Program RSAM, HSAM and IDM tables according to the optimal paths determined in steps 7-9.

- NOTE: the optimum routing applies equivalently to both IDM and RSAM tables. The mesh topology is an exception. See 11.1.3.2 for more details on the mesh topology.

### 3.4.5.1 Memory Coalescing of Slave Memory

If an HA supports memory extension and is interconnected to a remote SA, then the memory belonging to the SA may be coalesced with the local memory of the HA and presented as a single large memory pool to RAs, via an RSAM table entry. This coalescing process is illustrated in Figure 33, where device D has an HA that has memory extension enabled. The memory extension is provided by device E, which incorporates a Slave Agent that is bound to the HA on device D. Thus, memory pool E on device E can be coalesced with memory pool D on device D to form a larger equivalent memory pool, D'. For RAs in devices A, B and C, routing tables (RSAMs) are programmed to reflect the coalesced memory, D', on a logical device D' with an effective HA that has the larger memory pool D'.

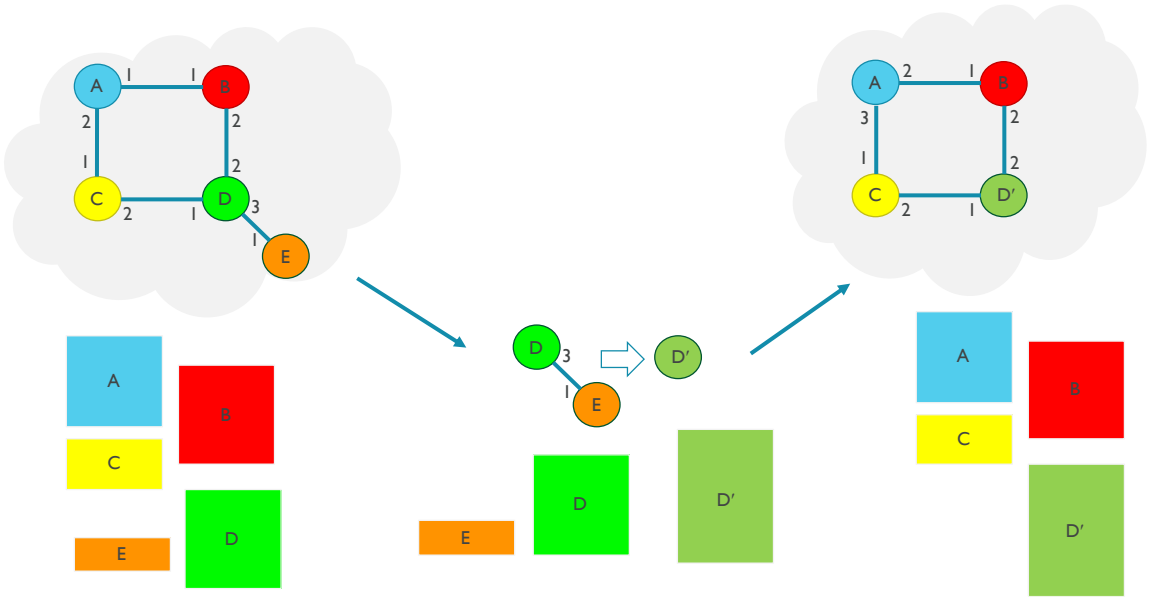



**Figure 33: Memory Coalescing of Slave Memory Pools**

## 3.4.6 Topology configuration (Stage 6)

Stage 5 results in two outputs:

- An offset in global system address map for each memory pool
- A set of routing entries for addressing each memory pool from each device in the CCIX graph

During this stage, the above information is programmed into the RSAM, HSAM, PSAM and IDM tables of each device in the CCIX system, effectively reflecting the optimum routing paths and memory layout established at the end of Stage 5. These tables are present in the common and component CCSRs of the device, as described in [1].

#### 3.4.6.1 SA to HA Binding for Extended Memory

In a complex system with multiple HAs and SAs, the boot firmware is responsible for associating an SA with a parent HA, and establishing the required routes thereof. In order to accomplish this, the firmware may use information such as:

1. *A priori* information such as hardware straps, supplied by the platform hardware, which could include information such as number of SAs to be paired with the HA and the interleave logic to be deployed.

2. Affinization between an SA and an HA, based on the following criteria:

   i. Nearest First – SA is allocated to the HA that is the least number of hops away
   ii. Parity of memory pools – the SA is associated with the nearest HA (based on criteria i), that has a memory expansion pool that best matches the memory pool that the SA implements, and drives the same format that matches the size of the memory pool)
   iii. Load Balancing – it is recommended that once the above two are established, the firmware should perform a reexamination of the loads on individual HAs and rebalance memory pools based on equity.

### 3.4.7 Host-specific CCIX Configuration (Stage 7)

From a host perspective, all CCIX devices appear in late boot as hot-plugged sockets with memory and processing elements. To accommodate these devices and their resources into its own view of the system, the host must provide implementation-defined configuration mechanisms, to enable it to become a participant in the CCIX subsystem. For example, the host must provide equivalents of the SAM and IDM tables, that the boot firmware must then program based on resources discovered in stages 1 through 6.

#### 3.4.7.1 Warm Reset

In certain implementations, a warm reset may be required to allow the host to reconfigure its internal state and other inherent properties to accommodate CCIX resources. As an example, the host's cacheability attributes may only be defined very early in boot phase (for example, Stage 1), and a soft/warm reset may be required in order to reconfigure these attributes so that they can now apply to CCIX memory regions. In this document, a warm reset is defined as a special reset that allows at least some context to be preserved. For the host, this minimally requires preservation of the SAM and IDM table entries programmed during the preceding cold reset path.

### 3.4.8 Device Configuration (Stage 8)

In this stage, the remainder of device initialization is performed, and the device is brought online. Care must be taken to ensure that there are no inadvertent conflicts. For example, an RA must not issue memory requests until all HAs have been activated. Where applicable, a warm reset must be issued prior to this stage, as explained in Section 3.4.7.

The generic flow is as follows:

- Activate each memory pool by setting its Valid bit.
- Activate each IDM entry by setting its Valid bit.
- Activate each SR-IDM entry, if present, by setting its Valid bit.

- Activate each SAM (RSAM and HSAM) Entry by setting its Valid bit.
- Activate each PSAM table of each port entry by setting its Valid bit.
- Activate each SA by setting its Valid bit. At this point, each slave memory is reachable by its HA.
- Activate each HA by setting its Valid bit. At this point, all peripheral memory is functional.
- Enable each port in each device by setting its Valid bit.
- Enable every link under a port by setting its Valid bit.
- Activate each IDM table.
- Activate every SR-IDM table if it is present, and if the current system topology is organized as a mesh. The SR-IDM is a special-purpose routing table meant for mesh topologies and addresses the routing requirements. Guidelines for mesh topology routing are provided in Section 11.1.3.2.1.
- Activate every HSAM table.
- Activate every RSAM table.
- Activate each RA in the system. At this point, memory requests can begin, and RAs can respond to snoops.
- Initialize the Transport Layer. Minimally, this requires the following operations:
    - Follow initialization step outlined in Section 3.4.8.1 .
    - On the host, perform any implementation-defined steps required to enable VDM routing for CCIX messages on the VC outlined in Section 3.4.8.1 .
- Enable every CCIX device. At this point, the CCIX system is brought online. Coherent traffic can now been issued from the host, and from device to device.

#### 3.4.8.1 Transport Layer Initialization

For each port in the device, the transport layer must be enabled in order for CCIX messaging to flow. Minimally, for PCIe-based implementations, this involves enabling the PCIe VC for the port that is reserved for CCIX. This VC can be discovered by querying the Transport Layer CCSR structures [1].

### 3.4.9 Publishing CCIX Information to OS (Stage 9)

This is the concluding stage related to CCIX initialization in the boot firmware flow. At this stage, the global SAM updated with CCIX memory regions, and NUMA domains are identified. Other data structures to support creation of system description tables such as ACPI and Device Trees, may be also created.

## 3.5 Transport Layer Initialization

### 3.5.1 Extended Speed Mode

CCIX supports an Extended Speed Mode (ESM) when two CCIX ports are directly connected through a CCIX interconnect. Detailed description of ESM is available in [1]. Following CCIX topology discovery, the firmware can identify each pair of interconnected CCIX ports, verify their ESM capabilities and configure ESM through their transport components. Figure 34 illustrates the flowchart on CCIX transport configuration process.

Before performing ESM configuration, the firmware must ensure that both ports on a link can support ESM. After initiating the process, ESM status also needs to be verified to ensure ESM calibration has completed.

The firmware must attempt to default to regular PCIe speeds if ESM cannot be supported.

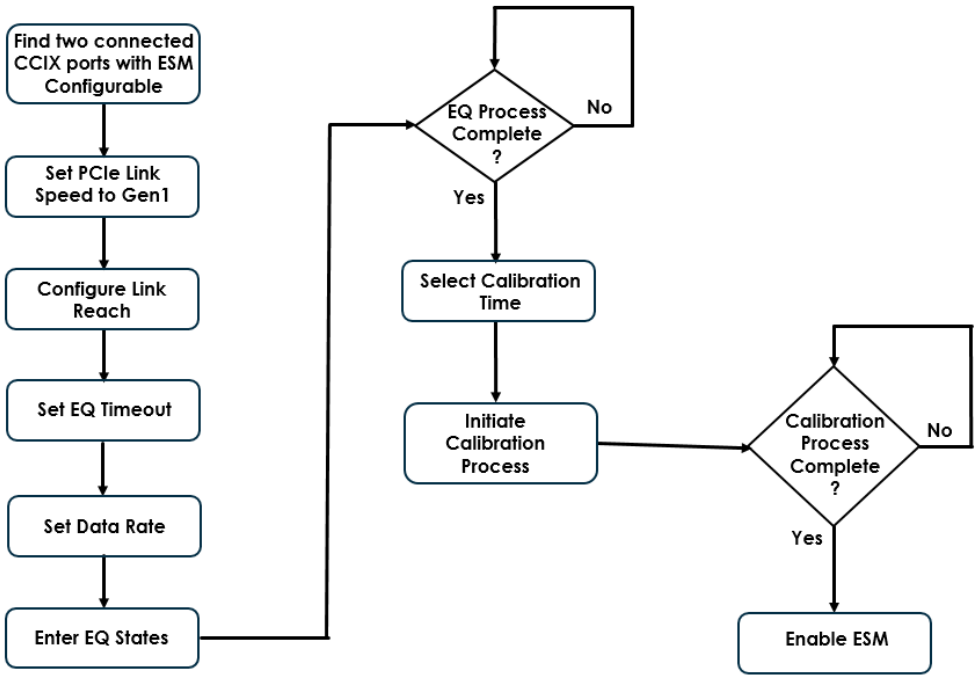



**Figure 34: CCIX Transport Configuration Flow**

ESM, if present, must be applied for each link to enable highest possible communication speed between peer devices. ESM is enabled based on an assessment of supported rates by each peer on a link. The negotiated maximum common denominator is then chosen as the link rate.

The following tables must be used in order to obtain the maximum common denominator:

**Table 4: ESM Programming for CCIX implementations above PCIe Gen 4**

| Component Advertised Rates | Link Partner Advertised Rates | Firmware Action |
|---|---|---|
| 25, 20, 16 | 25, 20, 16 | Choose CCIX mode and enable ESM. Set ESMDataRate1 to 25 GT/s and ESMDataRate0 to 16 GT/s. Programmed rate will be 25 GT/s. |
| 16 | 25, 20, 16 | Choose PCIe mode and do not enable ESM. Programmed rate will be 16 GT/s (highest PCIe data rate possible). |
| 25, 20, 16 | 16 | Choose PCIe mode and do not enable ESM. Programmed rate will be 16 GT/s (highest PCIe data rate possible) |
| No ESM | n/a | Choose PCIe mode. ESM remains not enabled. Programmed rate will be the |

| | | |
|---|---|---|
| | | highest common data rate as defined by PCIe. |

**Table 5: ESM Programming for PCIe Gen 5**

| Component Advertised Rates | Link Partner Advertised Rates | Firmware Action |
|---|---|---|
| 32, 25, 20, 16 | 32, 25, 20, 16 | Choose PCIe mode, and do not enable ESM. Programmed rate will be the highest common data rate as defined by PCIe (32 GT/s). |
| 32, 25, 20, 16 | 25, 20, 16 | Choose CCIX mode and enable ESM. Set ESMDataRate1 to 25 GT/s and ESMDataRate0 to 16 GT/s. Programmed rate will be 25 GT/s. (Same as 1.0 spec) |
| 25, 20, 16 | 32, 25, 20, 16 | Choose CCIX mode and enable ESM. Set ESMDataRate1 to 25 GT/s and ESMDataRate0 to 16 GT/s. Programmed rate will be 25 GT/s. |
| 25, 20, 16 | 25, 20, 16 | Choose CCIX mode and enable ESM. Set ESMDataRate1 to 25 GT/s and ESMDataRate0 to 16 GT/s. Programmed rate will be 25 GT/s. |
| No ESM | n/a | Choose PCIe mode. ESM remains not enabled. Programmed rate will be the highest common data rate as defined by PCIe. |

### 3.5.2 Configuring Optimized TLP Mode

The CCIX Optimized TLP format is described in [1]. Boot firmware should enable this mode early in boot phase, before any traffic can commence on the CCIX links in the system. Boot firmware may use the following assessment to configure this mode on a link:

- If the topology has PCIe switches, then optimized header mode cannot be enabled.
- If both ports on a link support optimized header, then enable optimize header format for that link.

## 3.6 Credit Allocation

Each port of a CCIX device communicates with its peers using credits. There are two key properties of the port associated with credit management:

1. The port can send only a certain maximum number of credits to its peers. This maximum is based on how many outstanding incoming messages can be queued at the port as a receiver, one credit per receive slot.

2. The port might only be able to accumulate a certain maximum number of credits from its peers.

Additionally, a port may have multiple links, and credits may be shared among these links. Ports advertise their credit management capabilities, and provide controls to allocate credits, via the link CCSRs of their links [1]. Figure 35 summaries the CCSR registers that are used for credit pool assignment.

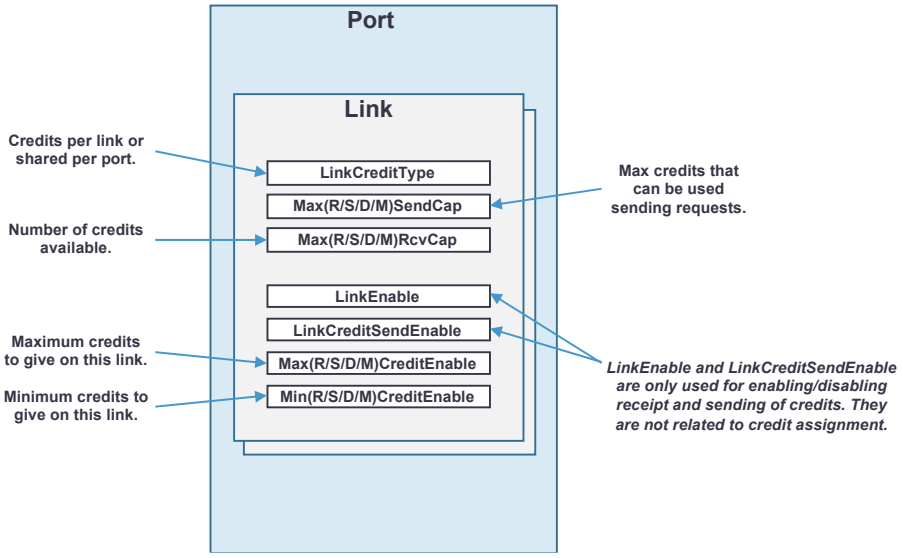



**Figure 35: Link CCSR Registers/Fields used for Credit Assignment**

Software is required to allocate credits based on an examination of the credit handling capacities of peer ports on each link in a device, in both directions. Some example methods are outlined below.

## 3.6.1 Dedicated Link Credit Pools

Ports with dedicated credit pools per link advertise this capability in the LinkCreditType field of the CCSR of each link [1]. Each link of the port then indicates this capability via the LinkCreditType field in its Link Capabilities CCSR, as outlined in Figure 35. For ports with links that have dedicated credit pools per link, then software may use the following scheme:

```
If MaxSendCap[peer] < MaxRcvCap[self] then
        set credit pool of self to MaxSendCap[peer]
else
        set credit pool of self to MaxRcvCap[self]
```

This avoids sending more credits than can be accumulated.

In the example system depicted in Figure 36, the maximum number of credits that Link 0 on the transmitting port (Port on Device 1) can accumulate is less than the maximum number of credits available on the receiver (Link 0 on Device 0). In other words:

**MaxRcvCap[00]** > **MaxSendCap[10]**

Thus, the credits pool allocated to Link 0, Device 0 is set to **MaxSendCap[10]**.

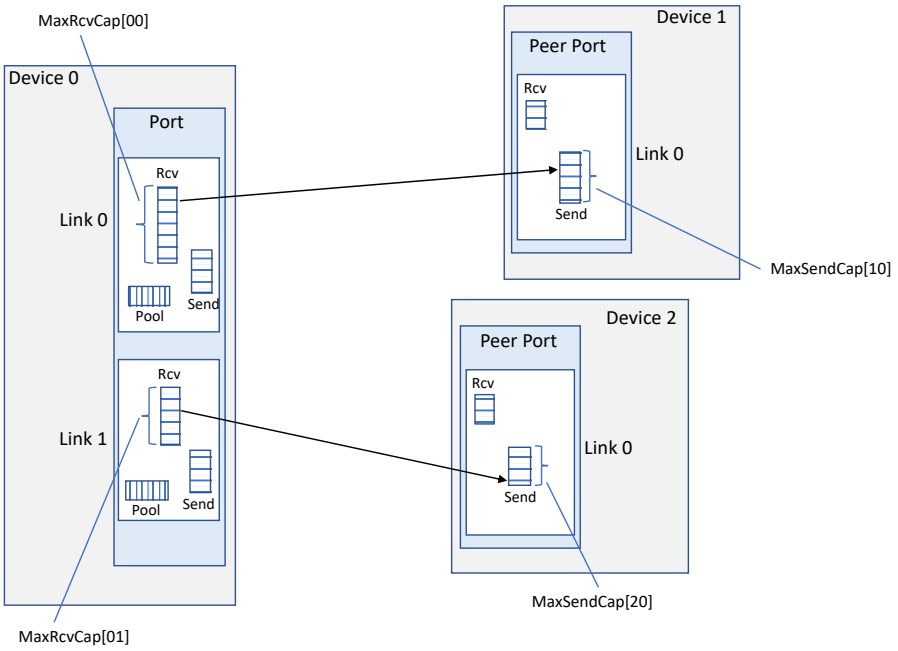



**Figure 36: Receiver-capability based Credit Allocation**

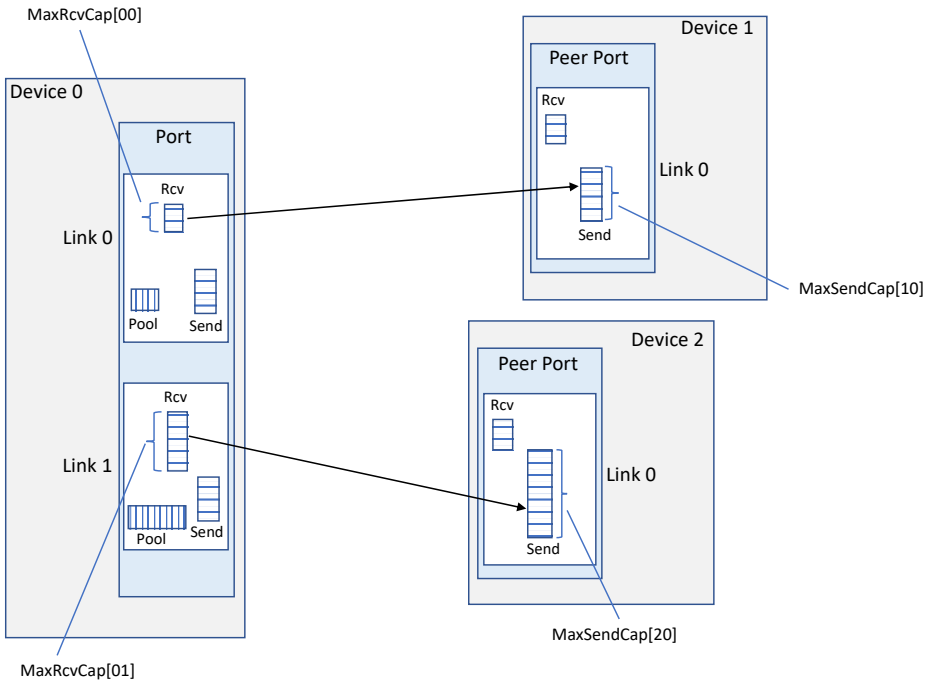



**Figure 37: Transmitter-capability based Credit Allocation**

In the example system depicted in Figure 37, the port on Device 2 can use more credits than the port/Link1 on Device 0. Hence the credit pool on Device 0 is set to **MaxRcvCap[01]**.

## 3.6.2 Shared Link Credit Pools

If all the links of a port share a common global pool of credits, then software will have to assign credits to each port such that the requirements of each link of that port are met simultaneously. Ports with dedicated pools per link advertise this capability via the LinkCreditType field of the CCSR of each link [1].

Two schemes are possible:

- Fixed allocation
- Flexible allocation

Once the overall credit pool for the port is determined, software is then responsible for dividing this pool among the links of the port.

> **IMPLEMENTATION NOTE**:
>
> These are illustrative schemes only. Other schemes are also possible.

## 3.6.3 Undersubscription

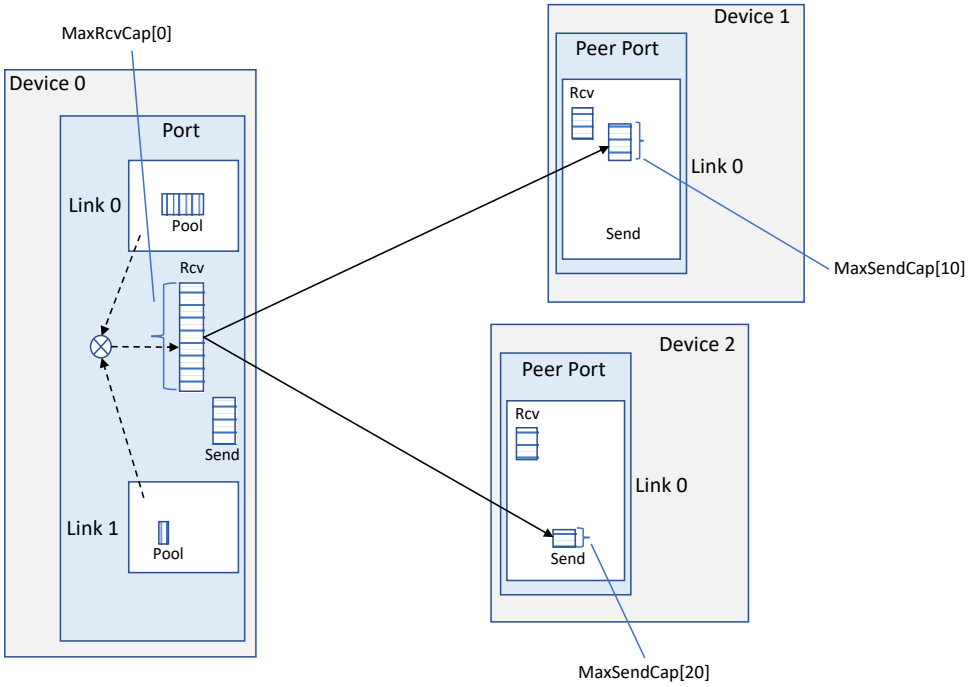



**Figure 38: Shared Credit Pool Assignment for Undersubscribed links**

This scheme works in the case when the receiving port has enough credits to allocate to all of its links. Such a case pertains to undersubscription and is depicted in Figure 38. Here,

**MaxRcvCap[0]** = $\sum_{j=0}^{N} \mathbf{MaxSendCap[j0]}$ (N = Number of partners, assuming that each partner has only one link)

Thus, each link is allocated a credit pool to match the number of credits that its partner can use.

### 3.6.4 Oversubscription

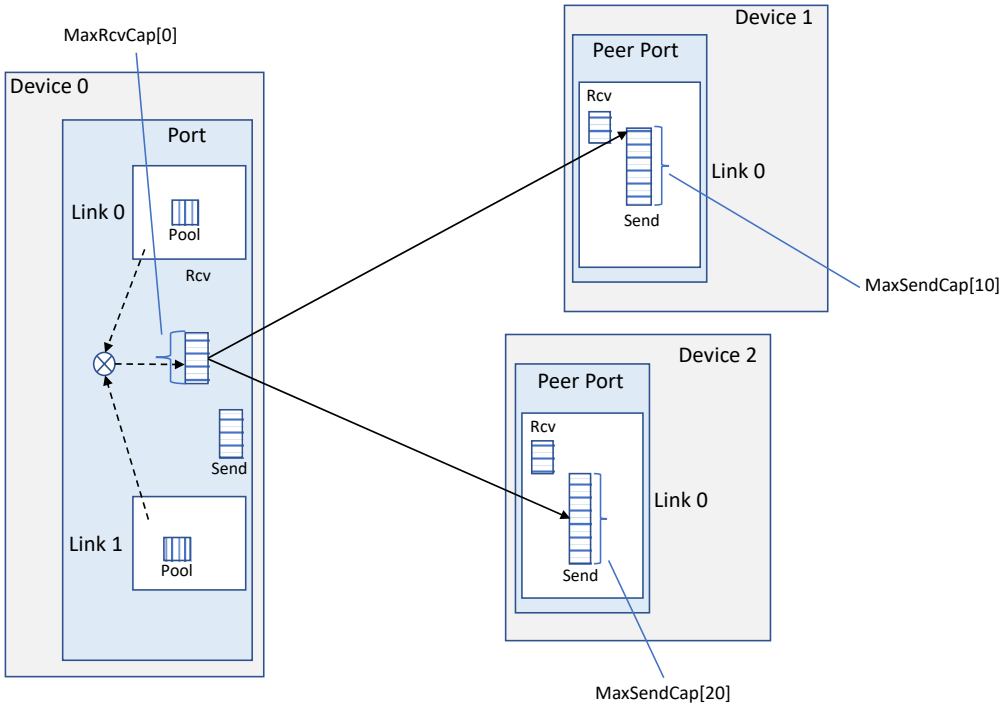



**Figure 39: Credit Pool Assignment for Oversubscribed Links**

When the port has fewer credits to divide among its links than the total number of credits that its link partners can use, then software will be required to come up with an apportioning algorithm. In the simplest case, which is called the *Fixed Allocation* method, the available credit pool may be divided equally among all links, as is depicted in Figure 39.

## 3.7 UEFI Firmware Requirements for CCIX Systems

The UEFI standard is widely adopted for boot firmware architecture on Arm and x86 platforms [2]. Most popular operating systems such as Windows, iOS and various Linux distributions support UEFI.

The following sections provide detailed specifications on CCIX FW services, protocols and CCIX FW drivers under UEFI development environment, and described in [2, 4].

Figure 40 describes the CCIX UEFI components integrated into UEFI architecture at different phases.

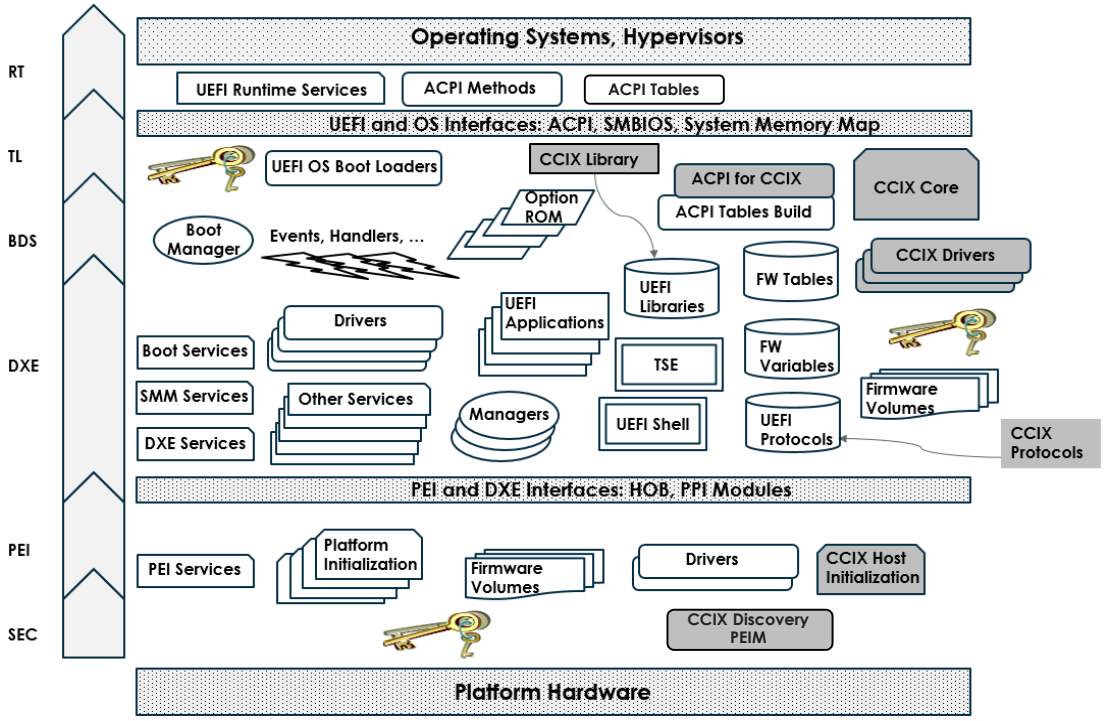



**Figure 40: CCIX UEFI Architecture**

## 3.7.1 Boot Flow

Section 3.4 described the generic boot flow for systems that incorporate CCIX, performed in several stages (Stage 1 to Stage 9). Stages 1 through 8 are largely generic and apply to all boot firmware technologies. Stage 9 is unique to the boot firmware technology deployed. For UEFI-based firmware, Stages 1-8 are performed by the CCIX UEFI driver described in Section 0. Stage 9 pertains to transforming discovered CCIX resources (memory, AFs) into UEFI or ACPI data structures that can then be consumed by other UEFI drivers or the OS during and after the boot process respectively. For example, EDK2-based ACPI platform drivers can refer to these data structures to construct ACPI tables such as the SRAT [3].

## 3.7.2 UEFI Firmware Architecture

Figure 41 provides the high-level overview of the stack recommended for a UEFI-compatible firmware for a CCIX 1.0a system based on PCIe [1]. The various boot stages (Stage 1-9) described in Section 3.4 and their interfacing with other parts of the generic UEFI boot process are also highlighted. The CCIX Core driver, the CCIX Transport driver and the CCIX Platform driver provide the complete CCIX boot process. These components, as well as other participants in the CCIX boot process, are outlined in the following sections of this chapter.

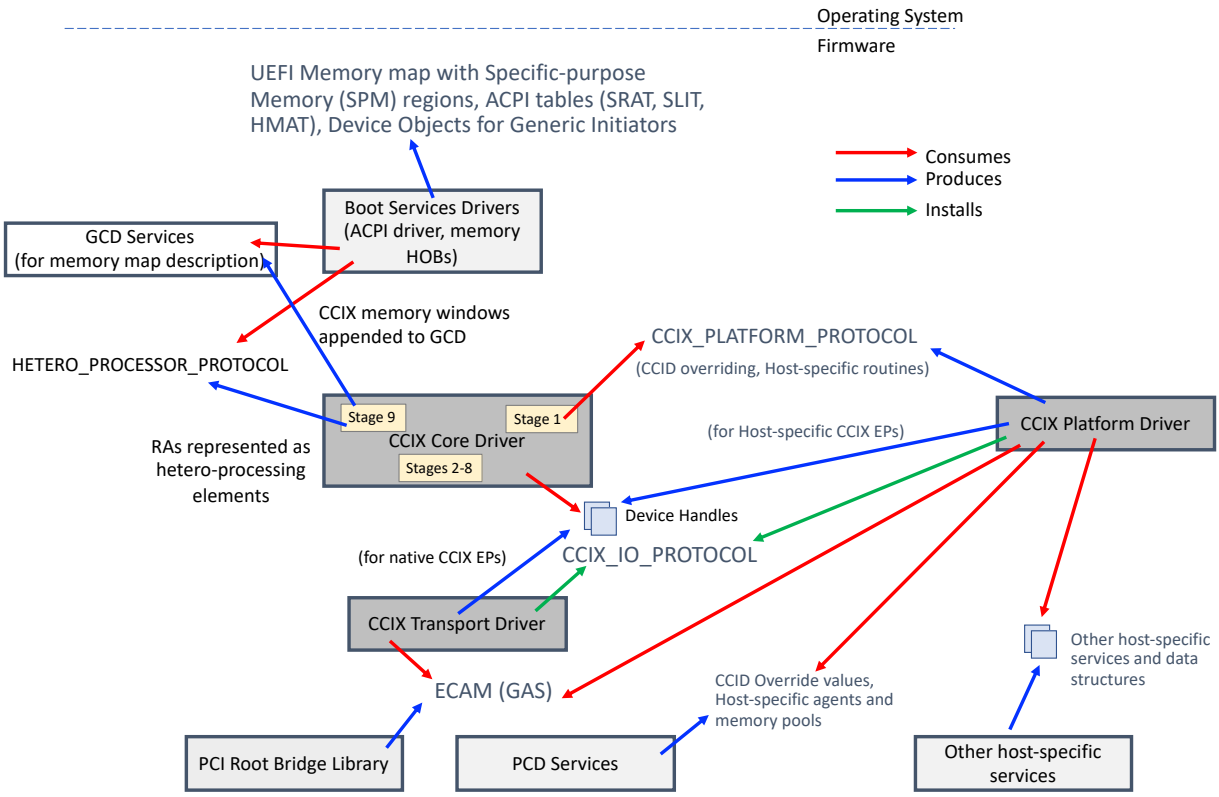



**Figure 41: UEFI Firmware Stack**

## 3.7.3 Discovery and Configuration Flow

If CCIX is implemented over PCIe, then the discovery and configuration flow may leverage protocol interfaces provided by the UEFI PCI Root Bridge support in UEFI. A high-level overview of this scheme is provided in Figure 41.

The boot sequence follows the chronological flow depicted in Figure 42. Noteworthy points are as follows:

- A device handle with one or more instances of the **CCIX_IO_PROTOCOL** protocol is created for every CCIX function discovered. See section 3.7.8 for more details on this protocol.
    - Similar device handles are also created for CCIX functions in the host.
- For functions that pertain to an RA or group of RAs, the **GENERIC_INITIATOR_PROTOCOL** protocol is installed on the function's device handle. This allows RAs in the system to be identified post CCIX discovery. See section 3.7.11.1 for more details on this protocol.

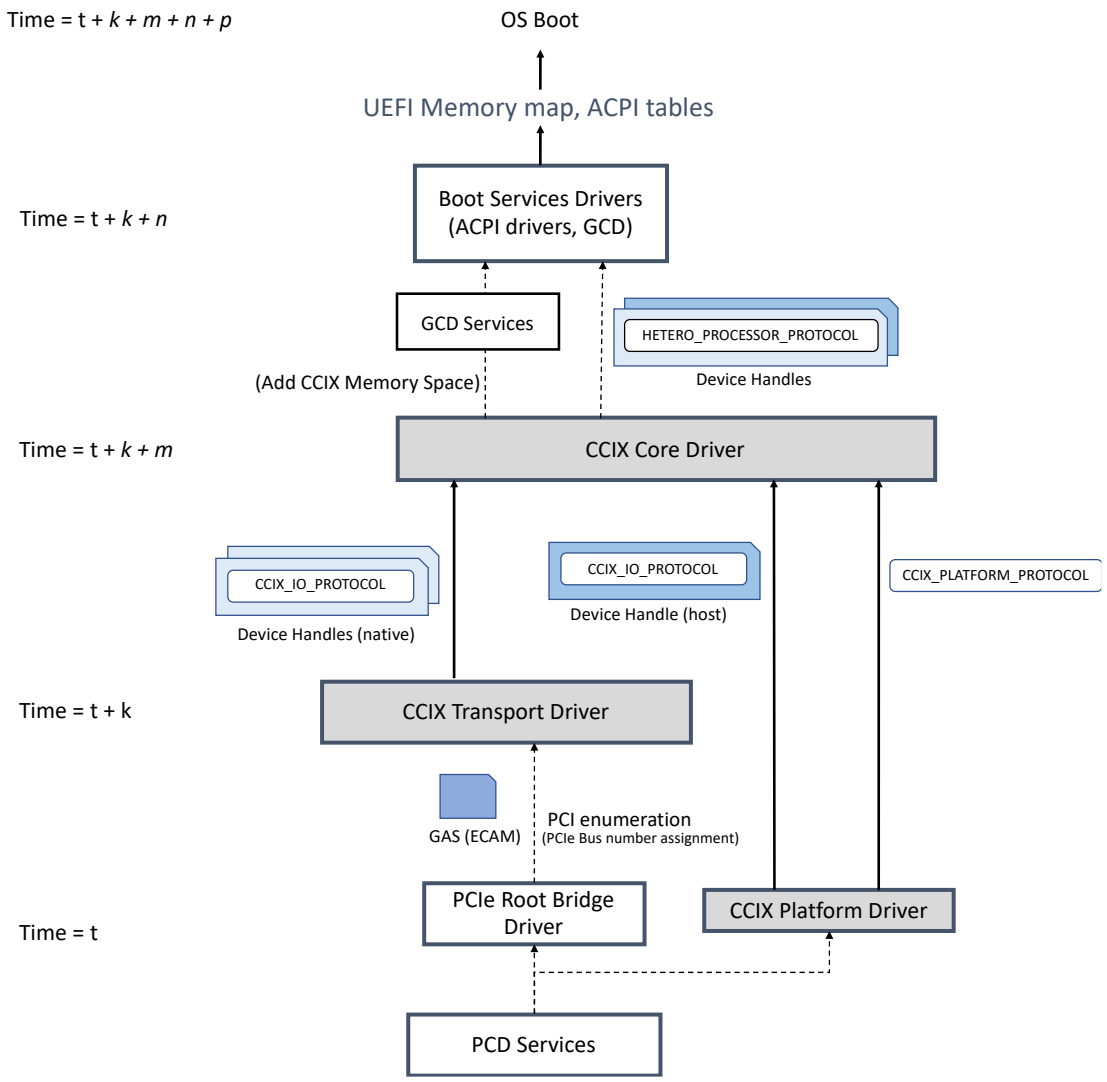



**Figure 42: Chronological Sequence of CCIX Boot Flow**

## 3.7.4 CCIX Bus Scan

The CCIX bus scan process will be based on the generic process outlined in Section 3.2.

### 3.7.4.1 CCIX over PCIe

For CCIX over PCIe implementations, the CCIX bus scan process relies on *a priori* PCIe bus enumeration to ensure that the underlying PCIe ports of CCIX endpoints have been assigned bus numbers and all intermediate PCIe switches forming the overall bus hierarchy have been configured to decode bus numbers accordingly.

## 3.7.5 CCIX Transport Driver

This driver is responsible for discovery of CCIX endpoints and functions and is implemented as a boot services driver. The discovery is based on knowledge of the GAS, LAS and CAS spaces as outlined in Section 3.2. For CCIX 1.0a based implementations [1], the discovery is based on the presence of the Protocol Layer DVSEC structure in LAS space, where the LAS space is equivalent to the PCIe config space of the PCI function that represents the CCIX function.

The discovery process involves performing a bus walk on GAS spaces to access LAS spaces belonging to CCIX endpoints, and then searching for CCIX functions, and examining each CCIX function to locate CAS spaces.

The CCIX Transport driver creates a device handle for each CCIX function thus located. It then installs an instance of the **CCIX_IO_PROTOCOL** protocol for each CAS space present in the function.

## 3.7.6 CCIX Core Driver

The CCIX Core driver is implemented as a boot services driver. It performs generic CCIX discovery and configuration flow, which involves CCIX agent discovery, CCIX topology recognition, CCIX memory mapping and configuration. The flow maps to Stages 1-9 of the CCIX boot flow outlined in Section 3.4.

The CCIX Core driver involves the following steps:

- It locates and memorizes the instance of the **CCIX_PLATFORM_PROTOCOL** protocol created for the current system.
- It then searches the handle database to locate device handles that have the **CCIX_IO_PROTOCOL** installed on them.
- It next performs Stages 1-9 of the generic boot flow outlined in Section 3.4, where it communicates with CCIX functions using the device handles obtained in the previous step.
- In stage 1, it invokes the *PreEnumeration* interface of the **CCIX_PLATFORM_PROTOCOL** protocol in order to pre-configure the host prior to the commencement of the CCIX boot flow.
- In stage 7, it invokes the *PostEnumeration* interface of the **CCIX_PLATFORM_PROTOCOL** protocol in order to configure the host following the completion of CCIX boot flow.
- During the agent discovery process (Stage 4), it installs the **GENERIC_INITIATOR_PROTOCOL** protocol on the device handle created for the current CCIX function, if this function holds an RA. Details of the **GENERIC_INITIATOR_PROTOCOL** protocol are available in Section 3.7.11.1 .
- The driver is then responsible for assigning mutually unique EAIDs to error agents reported through the **CCIX_PLATFORM_PROTOCOL** instance, and then programming these identifiers in all CCIX endpoints during the CCIX device configuration stage (Stage 8). This involves programming the EAID register in the Common CCSR of CCIX devices [1].

The CCIX Core driver's role in the overall UEFI boot process is outlined in Figure 41.

### 3.7.6.1 Resolving PCIe Dependencies

For CCIX implementations using PCIe transport, EDK2 implementations of the firmware necessitate the bringing up of the PCIe transport prior to CCIX discovery, owing to the following reasons:

- The CCIX bus scan process may require bus numbers to be assigned to the PCI endpoints *prior* to the CCIX bus scan. This ensures that PCIe config space decoding within the ECAM space is operational. For example, switches and Root Ports are configured for bus decoding prior to the enumeration process.
- The CCIX bus scan requires *a priori* knowledge of the PCIe ECAM space in order to begin.

This dependency is resolved using the UEFI dependency (depex) expression, and is based on the gEfiPciEnumerationCompleteProtocolGuid GUID. More details on this GUID can be obtained from [10].

## 3.7.7 CCIX_ADDRESS

The addressing scheme for CCIX is closely related to the physical layout of CCIX functions and endpoints within a device and of the device itself within the overall system. The physical address of a CCIX component determines where its CAS is located, as described in Section 3.1.

Thus, a component is uniquely identified using the following information:

1. The physical address of the component.
2. The offset of the endpoint's CCSR structure in the endpoint's CAS region.

The physical address of a component is the combination of the parent CCIX port and function where the component is located.

```
typedef struct _CCIX_ADDRESS {
        UINT32      CcixPort;
        UINT32      CcixFunction;
} CCIX_ADDRESS;
```

## 3.7.8 CCIX_IO_PROTOCOL

This protocol provides interfaces for reading and writing to a CCIX CAS region within a CCIX function, as illustrated below in Figure 43.

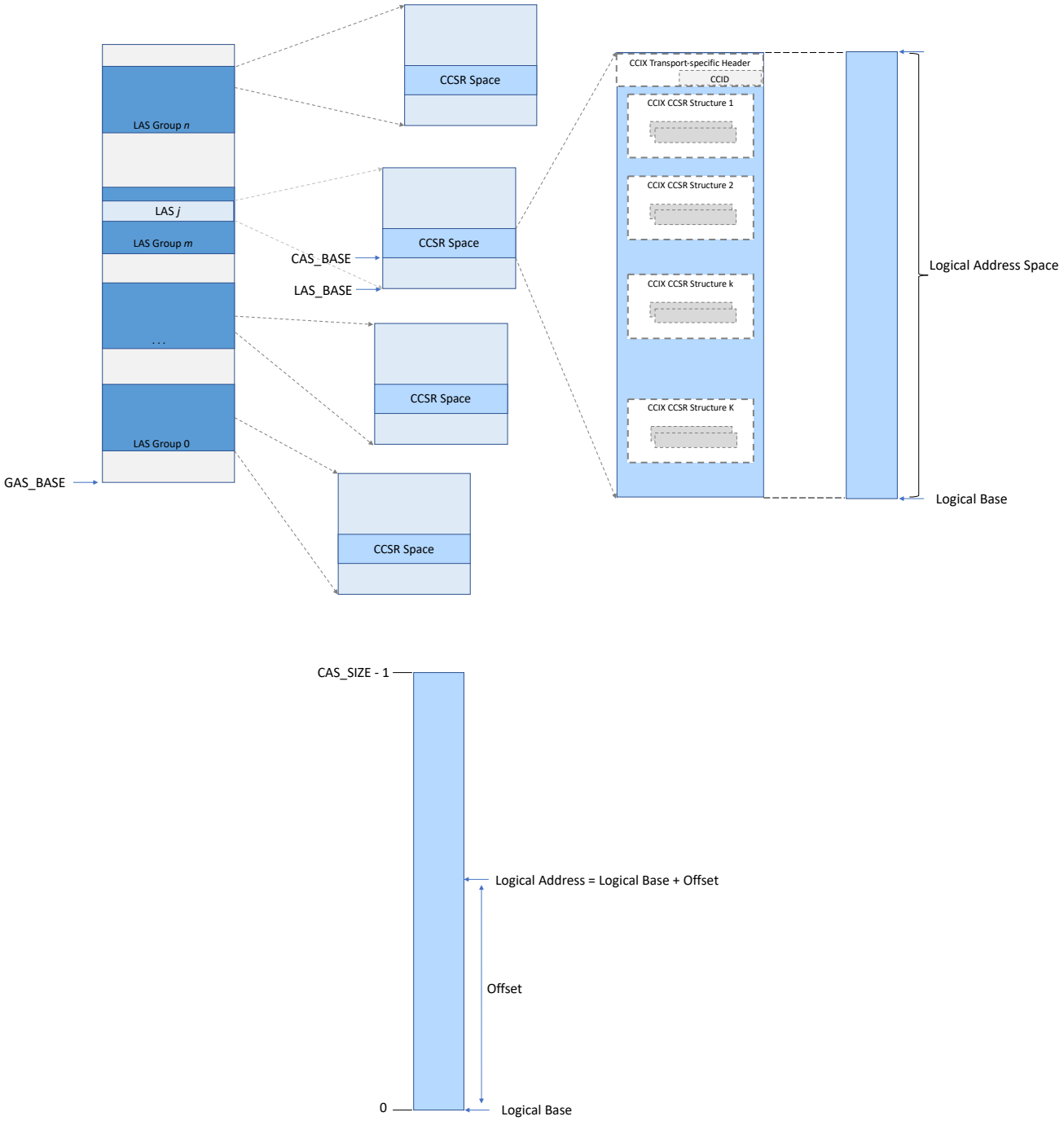



**Figure 43: Accessing the CAS region as a Logical Address Space**

The **CCIX_IO_PROTOCOL** provides an abstraction for accessing arbitrary addresses within the CAS region, where the CAS region is conceptualized as a logical address space. Offsets passed to the protocol interface are relative to the base of this logical address space, and the base is at logical address 0 in this logical space, as depicted in Figure 44.

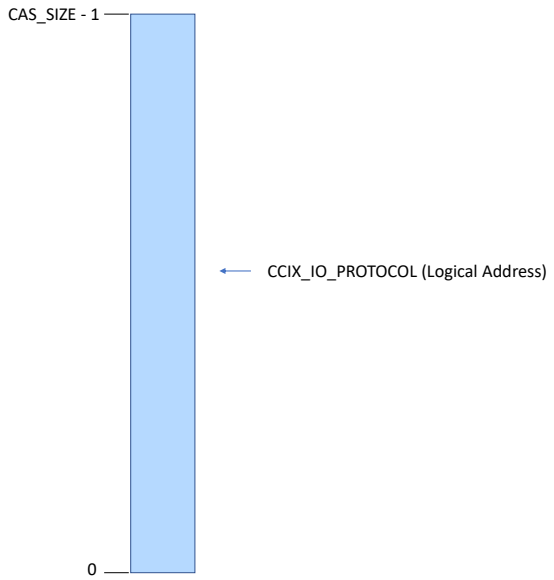



**Figure 44: Using CCIX_IO_PROTOCOL to access logical addresses in the CAS region**

### 3.7.8.1 Host-specific CCIX Discovery and Configuration

In Section 3.3, the need for a mapper for presenting the host's implementation-defined register layout as a standard layout to software was described. UEFI based firmware may follow the same guidelines for host implementations that do not follow the standard layout.

The **CCIX_IO_PROTOCOL** allows UEFI-based firmware implementations to create the same logical address space view of the host's implemented registers as defined for native CCIX endpoints in Section 3.7.8.

As illustrated in Figure 25, this involves organizing the host-specific CCIX registers into a virtual GAS space, and thenceforth to virtual LAS and CAS spaces. The firmware then should bind the **CCIX_IO_PROTOCOL** protocol to this virtual CAS space, as illustrated in Figure 45.

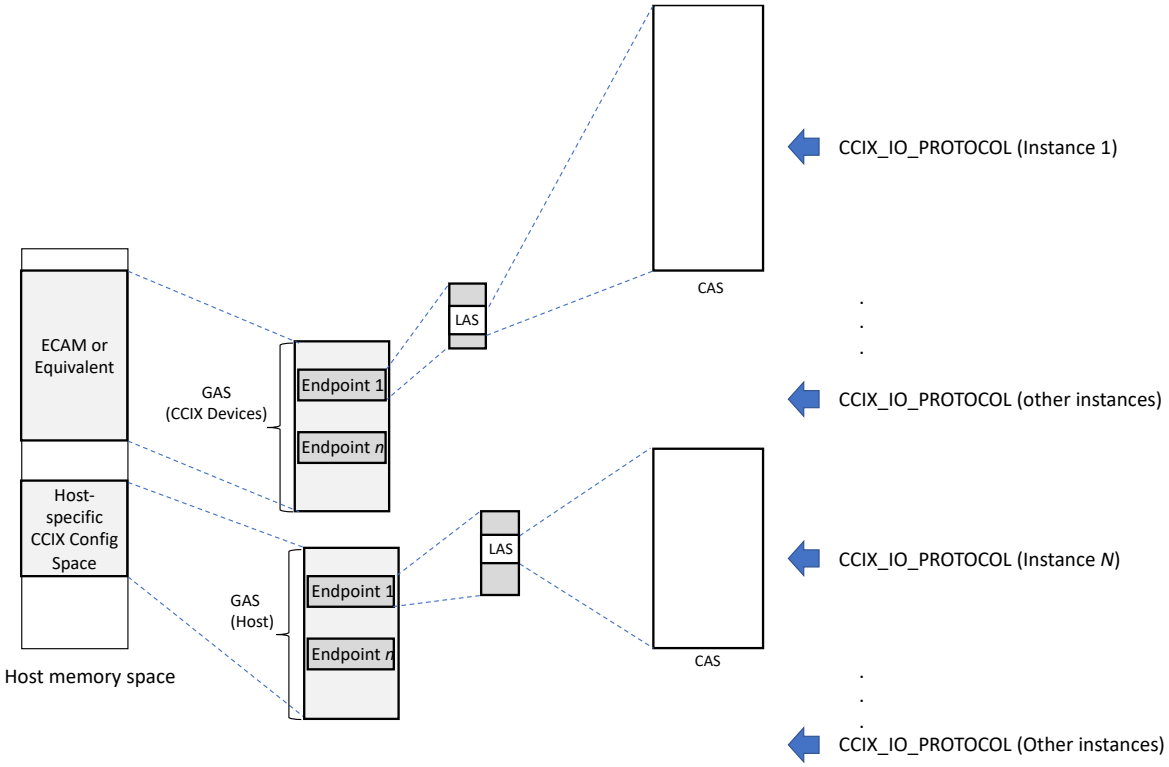



**Figure 45: Installing CCIX_IO_PROTOCOL on Host and Device Endpoints based on CCIX GAS/LAS/CAS addressing scheme**

### 3.7.8.2 Description

## CCIX_IO_PROTOCOL

### Summary

Provides interfaces for I/O operations on CCIX CAS regions.

### GUID

```
#define CCIX_IO_PROTOCOL_GUID \
 { \
   0x76ac4eb3, 0xbaa8, 0x43d4, {0x8c, 0xb8, 0x16, 0x66, 0x8b, 0xb6, 0xfa, 0xad} \
 }
```

### Protocol Interface Structure

```
typedef struct _CCIX_IO_PROTOCOL {
        CCIX_IO_PROTOCOL_READ      Read;
        CCIX_IO_PROTOCOL_WRITE     Write;
```

```
} CCIX_IO_PROTOCOL;
```

### Parameters

| | |
|---|---|
| *Read* | Reads a logical address in CCIX CAS space. |
| *Write* | Writes a logical address in CCIX CAS space. |

### Related Definitions

```
typedef enum {
        CcixTransportCcsrType = 0x01,
        CcixProtocolCcsrType
} CCIX_CCSR_REGION_TYPE;

typedef enum {
        CcixWidthUint8,
        CcixWidthUint16,
        CcixWidthUint32,
        CcixWidthUint64,
        CcixWidthMaximum
} CCIX_IO_PROTOCOL_WIDTH;
```

## CCIX_IO_PROTOCOL.Read ()

### Summary

Reads a logical address in the specified CCIX CAS region.

### Prototype

```
typedef
EFI_STATUS
(EFIAPI *CCIX_IO_PROTOCOL_READ)  (
        CCIX_IO_PROTOCOL            *This,
        CCIX_CCSR_REGION_TYPE       RegionType,
        UINTN                       Address,
        CCIX_IO_PROTOCOL_WIDTH      Width,
        UINTN                       *Val
);
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **CCIX_IO_PROTOCOL** instance. |
| *RegionType* | Specifies the type of CAS region. |
| *Address* | Offset from the beginning of the CAS region. |
| *Width* | Width of the read operation. |
| *Val* | Returns the read data. |

### Description

This interface is used for reading a logical address in CCIX CAS space that is specified by *RegionType*. The interface internally converts the specified logical CAS address to a physical address in G-SAM using the conversion formula outlined in 3.1.2.

## CCIX_IO_PROTOCOL.Write ()

### Summary

Writes to a logical address in the specified CCIX CAS region.

### Prototype

```
typedef
EFI_STATUS
(EFIAPI *CCIX_IO_PROTOCOL_WRITE) (
        CCIX_IO_PROTOCOL            *This,
        CCIX_CCSR_REGION_TYPE       RegionType,
        UINTN                       Address,
        CCIX_IO_PROTOCOL_WIDTH      Size,
        UINTN                       Val
);
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **CCIX_IO_PROTOCOL** instance. |
| *RegionType* | The type of CAS region. |
| *Address* | Offset from the beginning of the CAS region. |
| *Width* | Width of the write operation. |
| *Val* | Data to be written. |

### Description

This interface is used for writing a logical address in CCIX CAS region that is specified by *RegionType*. The interface internally converts the specified logical address to a physical address using the conversion formula outlined in Section 3.1.2.

## 3.7.9 CCIX_PLATFORM_PROTOCOL

This protocol provides interfaces for obtaining and setting platform-specific configuration related to the CCIX subsystem:

- Overriding the bootstrap DVSEC VID.
- Locating the CCIX GAS space.
- Performing host discovery.
- Performing host configuration.

### 3.7.9.1 Description

**CCIX_PLATFORM_PROTOCOL**

**Summary**

Provides interfaces for platform-specific operations related to CCIX.

**GUID**

```
#define CCIX_PLATFORM_PROTOCOL_GUID \
{0xaf6ac321, 0xa4c3, 0x11d2, {0x8e, 0x3c, 0x00, 0xa0, 0xc9, 0x69, 0x72, 0x3c}}
```

**Protocol Interface Structure**

```
typedef struct  _CCIX_PLATFORM_PROTOCOL {
CCIX_PLATFORM_PROTOCOL_GETCCIDOVERRIDELIST      GetCCIDOverrideList;
CCIX_PLATFORM_PROTOCOL_SETCCID                  SetCCID;
CCIX_PLATFORM_PROTOCOL_PRE_ENUMERATION          PreEnumeration;
CCIX_PLATFORM_PROTOCOL_POST_ENUMERATION         PostEnumeration;
CCIX_PLATFORM_PROTOCOL_GET_ERROR_AGENTS         GetErrorAgents;
} CCIX_PLATFORM_PROTOCOL;
```

**Parameters**

| | |
|---|---|
| *GetCCIDOverrideList* | Returns a list of alternate CCIDs that may be used for overriding the default or bootstrap CCID. |
| *SetCCID* | Selects and configures the selected CCID into the CCIX host, effectively overriding the default value of CCID. |

| | |
|---|---|
| *PreEnumeration* | Placeholder for all host-specific operations prior to generic CCIX discovery and configuration. |
| *PostEnumeration* | Placeholder for all host-specific operations on completion of generic CCIX discovery and configuration. |
| *GetErrorAgents* | Returns a list of error agents residing in the host. |

## CCIX_PLATFORM_PROTOCOL.GetCCIDOverrideList()

### Summary

Provides a list of CCID values for the current platform, that are available for overriding the bootstrap CCID on CCIX devices. By default, CCIX devices have CCID set to CCUV (CCIX Consortium Unique Value) on power on. However, the default CCUV can be overridden by an alternative value. This interface allows the host vendor to supply a list of override values. This interface only allows the override to be programmed into the host. CCID values on CCIX devices and endpoints thereof are overridden by explicitly writing to the COV (CCIX Override Value) register of the device or endpoint [1].

### Prototype

typedef

```
EFI_STATUS
(EFIAPI *CCIX_PLATFORM_PROTOCOL_GETCCIDOVERRIDELIST) (
IN CCIX_PLATFORM_PROTOCOL      *This,
OUT UINT16                     *List,
OUT UINTN                      *NumCCIDs
);
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **CCIX_PLATFORM_PROTOCOL** instance. |
| *List* | List of CCIDs that are available for overriding the CCID. |
| *NumCCIDs* | Number of entries in the CCID list. |

### Description

This interface can be called during boot services to retrieve the list of DVSEC CCIDs that can be programmed into the platform-specific placeholder in the host which holds the CCID of the CCIX Root Complex in the host.

## CCIX_PLATFORM_PROTOCOL.SetCCID ()

### Summary

Configures the selected CCID into the platform-specific placeholder for the host. Upon conclusion of this configuration, the default CCID (CCUV) in the host system is overridden with the specified CCID. The host will use the CCID value programmed in this placeholder for all CCIX messages (PCIe VDMs) that it generates, as well as for examination of all incoming CCIX messages that it receives from the CCIX fabric. Section 3.7.12 has more details on CCID override procedure for CCIX endpoints.

### Prototype

```
EFI_STATUS
(EFIAPI *CCIX_PLATFORM_PROTOCOL_SETCCID) (
    IN CCIX_PLATFORM_PROTOCOL    *This,
    IN UINT16                    *Ccid
    );
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **CCIX_PLATFORM_PROTOCOL** instance. |
| *Ccid* | CCID value to be programmed into the host's CCID field. |

### Description

This interface can be called during boot services to set a new DVSEC CCID into an implementation-defined placeholder for the host system. The new CCID becomes effective for all VDM filtering, as well as for any CCIX-defined CCSR structures in the host. Note that the boot firmware must program the new CCID into the COV register of every CCIX device in the system in order to get the system to globally work with the new CCID.

## CCIX_PLATFORM_PROTOCOL.PreEnumeration ()

### Summary

This interface serves as a callback for performing platform-specific initialization before generic CCIX discovery and configuration is begun.

### Prototype

```
EFI_STATUS
(EFIAPI *CCIX_PLATFORM_PROTOCOL_PRE_ENUMERATION) (
    IN CCIX_PLATFORM_PROTOCOL    *This
    );
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **CCIX_PLATFORM_PROTOCOL** instance. |

## CCIX_PLATFORM_PROTOCOL.PostEnumeration ()

### Summary

This interface serves as a callback for performing platform-specific initialization upon the completion of generic CCIX discovery and configuration.

### Prototype

```
EFI_STATUS
(EFIAPI *CCIX_PLATFORM_PROTOCOL_POST_ENUMERATION) (
    IN CCIX_PLATFORM_PROTOCOL        *This
);
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **CCIX_PLATFORM_PROTOCOL** instance. |

## CCIX_PLATFORM_PROTOCOL.GetErrorAgents ()

### Summary

Returns a list of descriptors describing error agents residing in the host.

### Prototype

```
EFI_STATUS
(EFIAPI *CCIX_PLATFORM_PROTOCOL_GET_ERROR_AGENTS) (
    IN CCIX_PLATFORM_PROTOCOL        *This,
    IN UINTN                         *Size,
    CCIX_ADDRESS                     *ErrorAgentList
);
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **CCIX_PLATFORM_PROTOCOL** instance. |
| *Size* | Returns the size of the *ErrorAgentList* list. |
| *ErrorAgentList* | Returns a list of CCIX addresses specifying the locations of error agents. |

### Description

An error agent is located using its **CCIX_ADDRESS**. For example, a CCIX Root Complex with two Root Ports with one error agent each will require a list of two **CCIX_ADDRESS** addresses to be returned by this interface. A CCIX Root Complex that implements an error agent as a RCIE would require the **CCIX_ADDRESS** returned by this interface to be set to reflect the internal bus of the Root Complex, and the PCI function number of the RCIE.

## 3.7.10 CCIX Platform Driver

The CCIX platform driver is a boot services driver that is responsible for supporting the CCIX Core driver and providing host or platform specific services to aid generic CCIX discovery and configuration. It performs the following operations:

- It instantiates and publishes the **CCIX_PLATFORM_PROTOCOL** that provides basic host or platform specific information to the CCIX Core driver.
- It creates device handles for host-resident CCIX functions, and installs the **CCIX_IO_PROTOCOL** protocol on these handles. This enables host components to be discovered natively.
- It maintains a mapper that translates **CCIX_IO_PROTOCOL** based accesses to a host-resident CCIX to the actual host-implemented methods for accessing the component.
- It provides the backend implementation of host or platform specific operations required to support *PreEnumeration* and *PostEnumeration* phases, servicing Stages 1 and 7 of the generic boot process outlined in Section 3.4.

The CCIX Platform driver's role in the UEFI boot process is outlined in Figure 41.

## 3.7.11 Publishing CCIX Information

### 3.7.11.1 Publishing CCIX Accelerator Information

The **GENERIC_INITIATOR_PROTOCOL** describes a heterogeneous processor in the CCIX system. A heterogeneous processor may be defined to represent any of the following:

- An RA that acts as the front-end cache for a group of AFs.
- A collection of RAs that act as the front-end cache for a group of AFs.

A heterogeneous processor may be associated with a region of SPM that is dedicated for its private use, and may have additional generic properties.

This protocol may be consumed by boot-service drivers. For example, a platform-specific ACPI driver might consume this protocol in order to construct ACPI tables (SRAT, HMAT) to describe CCIX-based Generic Initiators. The mechanisms required for identifying Generic Initiators in a CCIX system are identified in section 4.7.3.

The CCIX Core driver is responsible for publishing this protocol at the end of CCIX discovery and enumeration process.

**GENERIC_INITIATOR_PROTOCOL**

### Summary

This protocol describes a generic initiator and its properties.

### Prototype

```
typedef struct _GENERIC_INITIATOR_PROTOCOL {
    _ GENERIC_INITIATOR_PROTOCOL    *This;
    UINT8                           *MemoryRegion;
    UINTN                           *MemoryRegionSize;
    GENERIC_INITIATOR_PROPERTIES    Properties;
    EFI_DEVICE_PATH                 Path;
} GENERIC_INITIATOR_PROTOCOL;
```

### Related Definitions

```
typedef enum {
    HpPropertyAtomics,
    HpPropertyCacheCoherency
} GENERIC_INITIATOR_PROPERTIES;
```

### Parameters

| | |
|---|---|
| *This* | A pointer to the **GENERIC_INITIATOR_PROTOCOL** instance. |
| *MemoryRegion* | Offset of the SPM memory region that is associated with this generic initiator device. |
| *MemoryRegionSize* | Size of the SPM memory region associated with this generic initiator device. |
| *Properties* | Generic properties of this generic initiator device. |
| *Path* | UEFI device path for this generic initiator device that serves as a location identifier for the device for a given underlying transport or bus. |

## 3.7.11.2 Publishing CCIX Memory Pools

At Stage 9, the CCIX Core driver is required to update the UEFI memory map with descriptions for the newly discovered CCIX memory ranges. The CCIX Core driver can invoke the GCD services for a PI implementation of the UEFI firmware. The GCD services include two specific interfaces for this purpose:

- **AddMemorySpace**
- **SetMemorySpaceAttributes**

The following snippet illustrates how the GCD services can be used for registering a CCIX memory pool with the GCD, and informing the UEFI core that this memory pool needs to be marked as SPM:

```
CCIX_Discover_Next_Memory_Pool (&MemoryPoolOffset, MemoryPoolSize);

...


// Stage 9 begins here.
...


//gDS is a pointer to the DXE Services as per PI implementation
gDS->AddMemorySpace ( EfiGcdMemoryTypeSystemMemory,
                  MemoryPoolOffset,
                  MemoryPoolSize,
                  EFI_MEMORY_SP );
```

## 3.7.12 Programming COV

All CCIX devices are hardwired to carry the CCUV value in their CCID field. This is the bootstrap or default CCID value. The firmware may optionally incorporate the ability to program an alternate CCID if there are CCIX devices that do not have the standard CCUV programmed in the CCID field as above [1]. Such devices must, however, always carry the CCIX GUID, which is present at the CCIX GUID Offset in CCSR space as defined in [1]. This allows such devices to be discovered by software as CCIX devices, and then enabled to communicate with other standard CCIX devices using CCIX VDM messages. As defined in [1], the CCIX GUID is the value:

**{0xC3CB993B, 0x02C4, 0x436F, 0x9B68, {0xD2, 0x71, 0xF2, 0xE8, 0xCA, 0x31}}**

The boot firmware may opt for selecting a given CCID value based on a platform policy. The boot firmware can install and consume the **CCIX_PLATFORM_PROTOCOL** protocol to return a list of candidate CCIDs, and to program one of these candidates into all CCIX devices, based on another platform policy. The CCIX Base Specification provides details on how the COV field in a CCIX device can be programmed in order to override the bootstrap CCID value [1].

### 3.7.12.1 CCID Override Algorithm

If there is a requirement or a policy that the CCID value must be overridden, or if there are CCIX devices that do not carry the default CCUV in their CCID fields, then the boot firmware is required to program a globally common CCID value for all CCIX devices the platform. This programming requires overriding the CCID by writing to the COV field, as already outlined in Section 3.7.12.

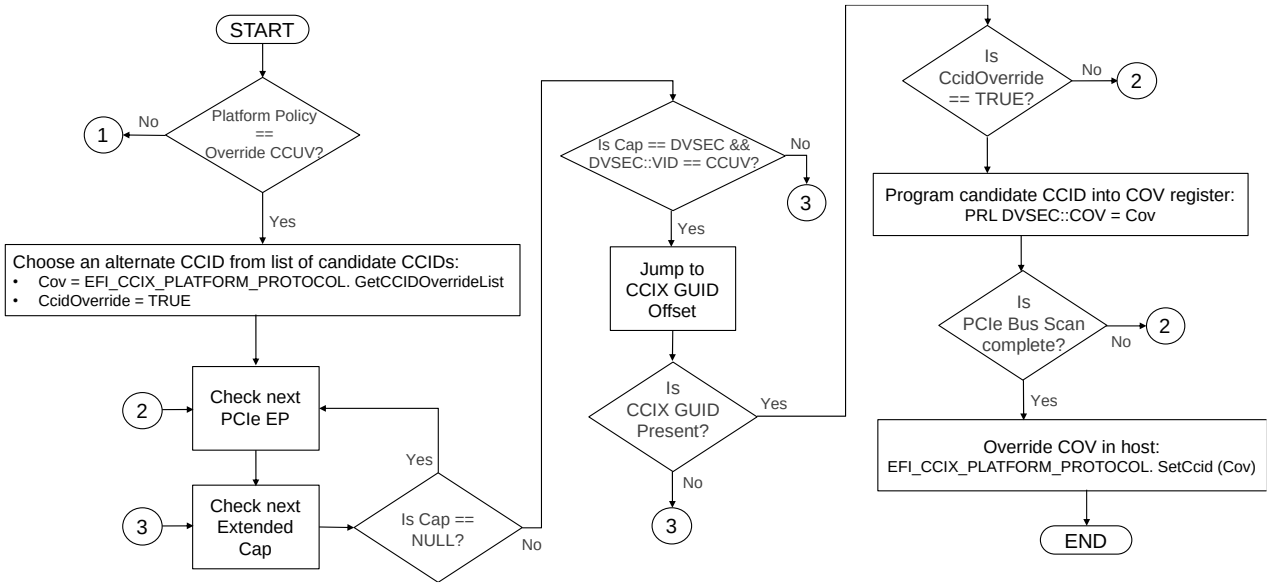



**Figure 46: CCID Override Flow**

The algorithm in Figure 46 provides a high-level overview of the CCID overriding procedure.

**IMPLEMENTATION NOTE:**

If a conflict exists between another feature and CCIX, such that they implement DVSEC structures with the CCUV value programmed in the DVSEC VID field, then there is a likelihood of a collision. To resolve this, the boot firmware must:

a) always examine the *CCIXGUIDOffset* field in the DVSEC structure for the presence of the CCIX GUID. If a CCIX GUID is present, then that confirms that the current endpoint is a CCIX endpoint.
b) redefine the DVSEC VID using the method outlined in 3.7.12.1 . The new CCID must be chosen from the list of candidate CCIDs only, and then applied universally to all CCIX devices in the system.

OS-based runtime re-enumeration or hot-plug may require firmware support in order to accomplish this. An ACPI _DSM method will be defined for the purpose. See Chapter 4 for details.

# Chapter 4. System Description

This section describes data structures and functions required for describing a CCIX-enabled system to a general-purpose operating system.

## 4.1 Memory Types

CCIX devices with one or more integrated HA's or SA's may have physical memory installed on them. This physical memory can have properties such as Type, cacheability and technology, as specified by the CCIX Base Specification [1].

**Table 6: CCIX Memory Pool Description to Generic Memory Class Mapping**

| CCIX CCSR MPGMTC[3:0] | CCIX CCSR MPSMTC[3:0] | CCIX CCSR Memory Attribute (MPATTR[2:0]) | CCIX CCSR Memory Extended Attribute (MPEATTR[2:0] | Generic Memory Class |
|---|---|---|---|---|
| ROM | Other | Normal cacheable | Don't care | ROM |
| Non-volatile | Flash | Normal cacheable | Don't care | Non-volatile RW (block writable) |
| Non-volatile | NVDIMM-F or NVDIMM-N | Normal cacheable | Don't care | Non-volatile RW (byte writable) |
| Volatile | SRAM, DRAM or HBM | Normal cacheable | Don't care | Volatile coherent |
| Volatile | SRAM, DRAM or HBM | Normal non-cacheable | Don't care | Volatile non-coherent |
| Device/Register | Don't care | Device | Don't care | Device |
| Don't care | Don't care | Don't care | Memory Hole | Hole |
| Other/ non-specified | Other/non-specified | Don't care | Don't care | Non-specified |
| Don't care | Don't care | Don't care | Private | Private |

The boot firmware is responsible for translating the memory pool descriptors in memory pool CCSRs into firmware tables for the operating system to consume. A device could also indicate holes in its memory pools, which could be internal memory that is consumed by the device and is either invisible to the operating system or appears to the latter as reserved memory.

The memory type mapping is a two-stage process. In the first stage, the memory pool capabilities structures are scanned to obtain information such as the Generic Memory Type, the specific memory attributes and extended attributes pertinent to the memory pool. These are then combined together and then translated into a generic memory class according to Table 6. In the second stage, the generic memory class is then mapped to firmware table descriptors in preparation to OS boot, for OS consumption.

# 4.2 NUMA

A CCIX-based system is essentially a NUMA system with at least one memory region, a host system with processors, and a set of AFCs. Memory regions managed by HAs, SAs and the host system are targets in this NUMA system, while the processors and AFCs are the initiators.

The boot firmware may describe the NUMA characteristics of the system using industry-standard firmware tables, such as ACPI [3] and device trees [8].

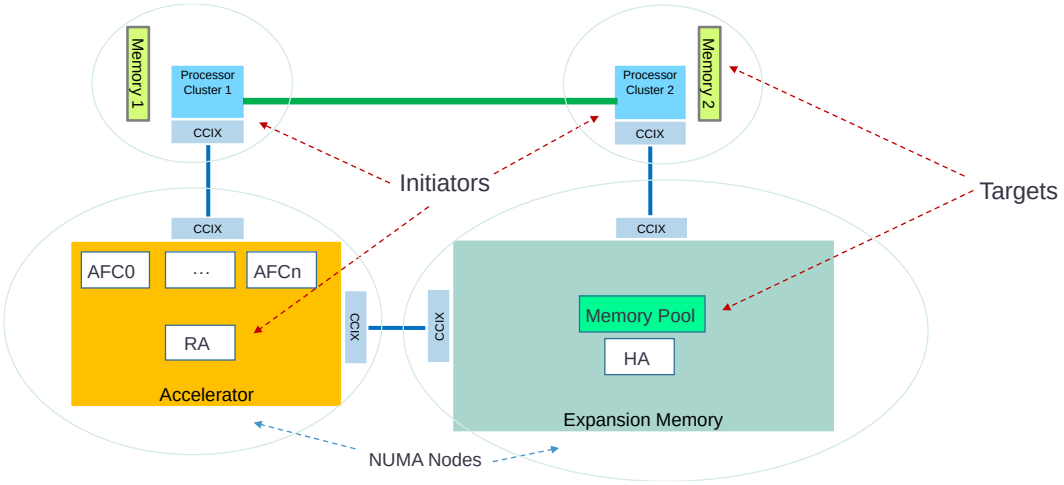



**Figure 47: An example CCIX System and its NUMA Classification**

## 4.2.1 Generic Initiators

RAs on CCIX devices may be considered as initiators in a NUMA system since they generate memory requests. There are two system NUMA designs possible:

1. An RA may be physically co-located with processors and memory (e.g. on a CPU socket), such that the memory is equidistant from the RA and the processors.
2. An RA may exist on a standalone accelerator device with or without associated local memory.

In both NUMA designs, firmware may identify the RA as a distinct Generic Initiator and assign an appropriate device handle to it. In the second case, the RA may warrant a dedicated NUMA domain, to distinguish it from standard NUMA domains that only pertain to processors. The decision in favor of the dedicated NUMA domain is made based on assessment of the interconnect between the device with the RA, presence of local memory, and memory bandwidth and latency characteristics. For example, an RA with a local memory may be combined into a single NUMA domain, such that memory requests from the RA are satisfied from that local memory. Another example would be a device with RA, and a second device with expansion memory directly interconnected with the first device. In this case, the interconnect between the two devices, as well as the expansion memory on the second device, may be deemed to be the best NUMA path to satisfy memory allocations from the RA. It may be noted that these are general rules of thumb that apply to any traditional NUMA system, and as such are not unique to CCIX. The key difference here is the creation of a dedicated NUMA domain for RAs as GIs.

### 4.2.2 NUMA Domains

#### 4.2.2.1 Creating NUMA Domains for CCIX Systems

As described in Chapter 2, a CCIX-enabled system may include:

- Type A (Accelerators only): CCIX devices with AFCs and RAs only
- Type B (Expansion memory only): CCIX devices with only HAs (and attached memory)
- Type C (Expansion memory only): CCIX devices with only SAs (and attached memory)
- Type D (Hybrid): CCIX devices with a mix of the above components

In each of the above cases, the NUMA properties of the system may be established based on the combinations present. In particular, the following rules might be applied:

1. If a device or group of interconnected devices of Type A exist within the topology, then they may be described as Generic Initiators as long as they lie on the same PCIe segment below the host.
2. If a device or group of interconnected devices of Types B or C or both exist within the topology, then the memory ranges that pertain to them may be affinized into memory-only NUMA domains, where:
   i. All memory ranges that possess a common set of properties (bandwidth and latency) and are reachable from the rest of the system via a common interconnect or interconnects with identical properties (bandwidth and latency), may be combined together into a distinct memory-only NUMA domain.
   ii. For each initiator (Generic Initiator or IO device or processor), if two distinct memory ranges have different bandwidth and/or latency characteristics, then they may be separated into distinct memory-only NUMA domains.
3. If a device or group of devices of Type D exist within the topology, then they may be affinized to each other and placed within the same NUMA domain, unless one of the conditions in 2.i. or 2.ii. holds true.

## 4.3 Specific-purpose Memory (SPM)

Specific-purpose Memory, or SPM, is memory that is preferred for use by one or more accelerators in the system. CCIX devices may include SPM memory regions hosted by HAs or SAs. The general rule of thumb for SPM usage is that:

- SPM memory is generally not favorable for storage of non-relocatable kernel data structures such as page tables, and as such, it is recommended that the OS may use SPM for general-purpose kernel and application allocations only under memory pressure.

The boot firmware is responsible for flagging SPM regions with a special memory attribute that enables the OS to distinguish these regions from regular conventional memory.

## 4.4 Error Agent

The Error Agent (EA) is described in Section 2.4. Boot firmware is responsible for identifying the EA in the system and assigning an appropriate EAID to it. The EAID, described in Section 2.6.4.2 , is globally known within a device, and firmware is responsible for programming it in the common CCSR of the primary port of the device. The IDM table on the device must carry routing information for the EAID.

Once thus programmed, the device can route PER messages to the EA. Any intermediate devices in the path to the EA must then also carry this IDM entry on its ports, to allow routing of PER messages.

**IMPLEMENTATION NOTE**:

A typical implementation of an EA is a host-resident sink. This sink allows PER messages to be routed to the CPU complex within the host system, thereby enabling system software to take control of the CCIX system on errors and triage the reported error as appropriate. Host systems then are required to provide host-defined mechanisms for error routing and signaling. Additionally, the Root Complex may provide implementation defined methods for error logging and error signaling for errors occurring in the CCIX interface within the Root Complex.

# 4.5 Host Description

The minimal set of CCIX-specific features of interest to system software is:

- Location of the Error Agent within the host, if implemented, for sinking PER signals and processing PER messages from devices.
- Location of the host-specific CAS regions for any runtime/dynamic (re)configuration, for each host-specific CCIX endpoint implemented.
- Location of error logs that pertain to the CCIX core within the host (e.g. CCIX Root Port errors).

System software may use the above information for:

- Dynamic re-enumeration of the CCIX part of the host.
- Dynamic programming of the port and link characteristics of the CCIX interconnect between the host and downstream CCIX devices, including:
  - Dynamic credit pool redistribution for flow control
  - Port aggregation
- Retrieving host-specific CCIX error logs.

System software may require the ability to discover these features and associated means of configuring or using them. Two methods are outlined below.

In this method, the firmware may provide a runtime service or method that the OS or OS drivers can use to discover CCIX features in the host, and to configure these features at runtime.

The boot firmware might provide a firmware table that lists the above features. This table may then be read by the OS to discover host-specific CCIX capabilities.

# 4.6 AFC (Acceleration Function Core)

For CCIX 1.0a based devices [1], AFCs are rooted in PCI PFs or VFs and are thus discoverable by software. Software needs to consult the Acceleration Function Property structure in its primary port to locate individual AFCs within a device. The AF Property structure is described in [1].

### 4.6.1 AFC to RA Associativity

The associativity between the dependent AFCs and their parent RA is also defined in the AF Property structure in the primary port of the device, as outlined in [1]. Software drivers may use this information to manage AFCs as a group or set of assignable resources, and to perform power management of the AFCs and RAs, again as a set of related resources.

## 4.7 Description in a UEFI and ACPI-aware system

For a CCIX system that is based on the UEFI [2] and ACPI [3] specifications, certain key aspects of the UEFI/ACPI support must be enhanced to enable discovery, configuration and use of CCIX resources using UEFI and ACPI-based interfaces. Such enhancements include programming data structures such as ACPI tables, with information relevant to CCIX. This section details provides an overview of the key areas that need to be considered to this end.

### 4.7.1 UEFI Memory Map

The UEFI boot firmware is responsible for translating the memory pool descriptors in component CCSRs into UEFI memory map for a UEFI-aware OS to consume. The UEFI memory map must accommodate all G-SAM windows that pertain to CCIX memory. The mapping of CCIX memory regions is performed in DXE, during the second stage of memory type classification, as explained in Section 4.1 and outlined in Table 7.

Additionally, the firmware must declare CCIX memory that is earmarked for exclusive use by CCIX AFCs as specific-purpose memory using an appropriate UEFI memory attribute encoding, as outlined in Table 8.

A device could also indicate holes via its memory pools, which are then marked as **EfiReservedMemoryType** memory to prevent inadvertent OS usage.

The ACPI defined memory types are defined in [3].

**Table 7: Mapping Memory Pool Description to UEFI and ACPI Descriptors**

| Generic Memory Class | CCIX CCSR Memory Attribute (MPATTR[2:0]) | UEFI Memory Type and Attribute (UEFI) | ACPI Memory Type (OS) |
|---|---|---|---|
| Non-volatile RO | Normal cacheable | **EfiConventionalMemory** EFI_MEMORY_RO \| EFI_MEMORY_SP | AddressRangeMemory |
| Non-volatile R/W | Normal cacheable | **EfiPersistentMemory** EFI_MEMORY_NV \| EFI_MEMORY_SP | AddressRangePersistentMemory |
| Volatile Coherent | Normal cacheable | **EfiConventionalMemory** | AddressRangeMemory |

| | | | |
|---|---|---|---|
| | | EFI_MEMORY_WB \| EFI_MEMORY_SP | |
| Volatile Non-coherent | Normal non-cacheable | **EfiConventionalMemory** EFI_MEMORY_WC \| EFI_MEMORY_UC \| EFI_MEMORY_SP | AddressRangeReserved |
| Device | Device | **EfiMemoryMappedIO** EFI_MEMORY_UC \| EFI_MEMORY_SP | AddressRangeReserved |
| Hole/Private/ non-specified | Don't care | **EfiReservedMemoryType** EFI_MEMORY_UC \| EFI_MEMORY_SP | AddressRangeReserved |

**Table 8: Recommended UEFI memory map based on Usage**

| OS | Usage | EFI Memory Type | EFI Memory Attribute | Description |
|---|---|---|---|---|
| Legacy OS | Expansion Memory | **EfiReservedMemoryType** | EFI_MEMORY_XX \| EFI_MEMORY_SP | OS avoids to this memory for allocation to kernel data/code. The memory is managed by an OS driver. |
| | Acceleration -reserved | **EfiReservedMemoryType** | EFI_MEMORY_XX \| EFI_MEMORY_SP | OS avoids using this memory for allocation to kernel data/code. The memory is managed by an OS driver. |
| Enlightened OS | Expansion Memory | **EfiConventionalMemory** | EFI_MEMORY_XX \| EFI_MEMORY_SP | OS treats memory based on specified attributes. The memory is managed by the kernel memory manager. However, the OS avoids using this memory for allocation to kernel data/code. |
| | Acceleration -reserved | Depends on memory pool mapping specified in Table 7. | EFI_MEMORY_XX \| EFI_MEMORY_SP | OS treats memory based on specified attributes. The memory is managed by the kernel memory manager. However, the OS avoids using this memory for allocation to kernel data/code. |

EFI_MEMORY_XX: Depends on memory pool mapping specified in Table 7.

**IMPLEMENTATION NOTE**:

The boot firmware requires *á priori* knowledge of the Operating System that will run on the system in order to correctly apply the mapping recommended in Table 8. This information may be supplied in boot phase in the form of a firmware policy.

## 4.7.2 SPM

The UEFI specification version 2.8 defines a new memory attribute called EFI_MEMORY_SP, for describing specific purpose memory [2]. The UEFI boot firmware should set this attribute on all CCIX SPM memory discovered during the boot phase.

The FW should declare SPM memory as conventional memory types (**EfiConventionalMemory**) to allow OS memory managers to treat SPM as regular system memory but with the caveats that are described in Table 8.

The firmware flow required for SPM is illustrated in Figure 48.

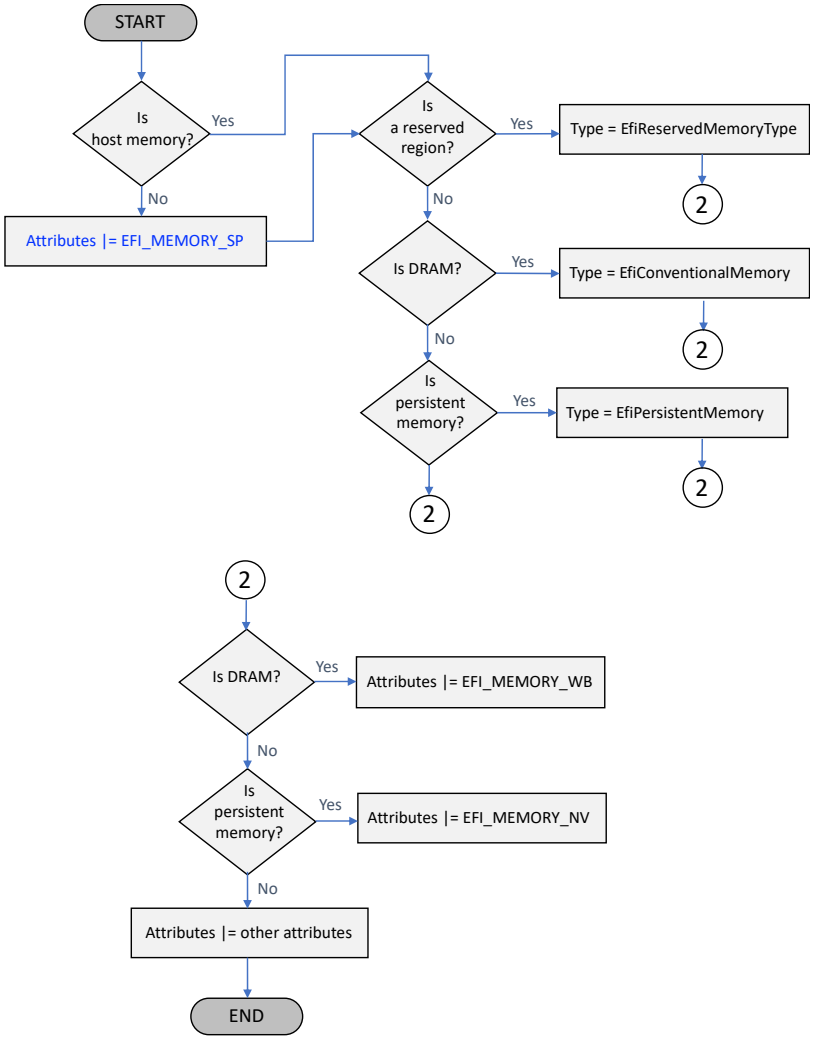



**Figure 48: UEFI Boot Flow for SPM Initialization**

### 4.7.3 NUMA Characteristics

The boot firmware may describe the NUMA properties of a CCIX-enabled system in accordance with principles laid out in section 4.2. These properties are described by the ACPI SRAT and SLIT tables. For CCIX systems with HA-based memory, it is strongly recommended to provide the HMAT table. The HMAT table may be used to describe additional heterogeneous characteristics pertinent to CCIX expansion memory. This is because the expansion memory provided by the HAs inherently has different NUMA properties than regular/conventional memory provided by the host system. The difference in NUMA properties arises as a result of the properties of the CCIX links that interconnect the expansion memory to the rest of the system, as well as the inherent proximity that the expansion memory has to CCIX-based accelerators.

The SRAT and HMAT may together also be useful for describing CCIX RAs as Generic Initiators (GIs) and describing the NUMA properties of paths between GIs and the SPM memory regions that they are associated with.

The description must include the following properties:

- The overall NUMA characteristics of the system, and the NUMA domains that CCIX components belong to.
- The mapping of memory subranges in the system to a distinct NUMA domain. This leverages UEFI system memory map definitions – the EFI system memory map, the EFI memory types and EFI memory attribute.

**IMPLEMENTATION NOTE**

An ACPI 6.3 aware firmware might choose to place Generic Initiators in dedicated proximity domains that have no other entity (initiator or target) [3]. However, this is a problem for legacy operating systems (i.e. Operating Systems that do not support ACPI 6.3) if the GI is the only device associated with the proximity domain. Operating systems are designed to only accept and work with proximity domains that are associated with "well-known" devices.

The boot firmware must be able to dynamically assign GIs to alternate proximity domains if a legacy OS is running. This can be accomplished through the use of the _PXM method to "migrate" the GI to the closest proximity domain that has at least one of the well-known devices (memory or processors) associated with it. The decision to steer the decision taken within the _PXM method is based on the _OSC bit described in 4.7.4.1.1.

For a comprehensive overview of programming guidelines and related recommendations specific to NUMA, please refer [11].

#### 4.7.3.1 Other NUMA Descriptors

In addition to the definition of CCIX NUMA domains, the firmware must also supply information such as memory address translation units (IOMMUs), types and characteristics of memory (bandwidth and latency, access sizes) and distances between memory nodes and initiators.

**Table 9: Common ACPI tables for CCIX**

| Table | Details |
|---|---|
| SRAT | SRAT must be populated to describe association of CCIX memory ranges to NUMA domains (proximity domains). |
| SLIT | SLIT must be populated to describe the relative NUMA distances between proximity domains defined for CCIX memory regions as well as CCIX-based Generic Initiators (RAs). |
| HMAT | HMAT sub-structures must be defined to describe bandwidth, latency, cacheability (memory-side caches only) of CCIX memory regions. |

### 4.7.4 OS and Firmware Negotiation

Certain CCIX-specific system configurations mandate a negotiation between the firmware and the OS. The _OSC method is intended for such negotiations.

#### 4.7.4.1 Platform-wide _OSC

The platform-wide _OSC bit specifies bitmaps in DWORD 2 to allow the OS and the platform to perform a handshake to mutually communicate support for specific features. In this section, _OSC bits specific to CCIX and their intended usage models are described.

##### 4.7.4.1.1 Generic Initiator Support

CCIX systems that have Generic Initiators may require the use of the following _OSC bit to allow the OS and the boot firmware to exchange GI awareness [3]:

| Bits | Field Name | Definition |
|---|---|---|
| … | … | … |
| 17 | Generic Initiator Support | This bit is set if OSPM supports the Generic Initiator Affinity Structure in SRAT. |

#### 4.7.4.2 CCIX-specific _OSC

| Arg | Name | Description |
|---|---|---|

| | | |
|---|---|---|
| Arg 0 | UUID | 4c619395-e8dd-48fa-94fe-178498d2b98b |
| Arg 1 | Revision | Set to value of 1 for 1.0. |
| Arg 2 | Number of DWORDs | Set to 3. The first DWORD is as per the ACPI specification [3], the second DWORD is a set of Support fields, and the third DWORD is a set of Control fields. |
| Arg3 | OSC Capabilities | |

#### 4.7.4.2.1 Interpretation of _OSC Support Field

| Control Field bit Offset | Interpretation |
|---|---|
| 0 | CCIX Native PER Handling – the OS sets this bit to 1 to indicate that it can handle CCIX PER errors natively. |
| 1 | Re-enumeration – the OS sets this bit to 1 to indicate that it can perform CCIX topology discovery and configuration. |
| 2-31 | Reserved |

#### 4.7.4.2.2 Interpretation of _OSC Control Field, Passed in via Arg 3

| Control Field bit Offset | Interpretation |
|---|---|
| 0 | The OS sets this bit to 1 to request native CCIX PER handling. |
| 1 | The OS sets this bit to 1 to be allowed to discover and reconfigure CCIX topology. The OS may also be required to re-enumerate the underlying PCI buses as a result. OS should request re-enumeration only if it knows how to configure CCIX topology and routing. |
| 2-31 | Reserved |

#### 4.7.4.2.3 Interpretation of _OSC Control Field, Returned Value

| Control Field bit Offset | Interpretation |
|---|---|
| 0 | The firmware sets this bit to 1 to allow the OS to natively handle CCIX PER errors. |
| 1 | The firmware sets this bit to 1 to allow OS to rearrange and reconfigure the CCIX topology, including re-enumeration of the underlying PCIe buses. |
| 2-31 | Reserved |

#### 4.7.4.3 Re-enumeration

A CCIX-aware boot firmware may perform certain configurations in PCI space that are pertain to CCIX (e.g. VC capability, ATS). More importantly, bus numbers allocated to PCIe ports reflect in CCIX link control structures in CCIX CAS space. In future, CCIX-specific configurations may exist in PCI BAR space of the CCIX endpoint. Such CCIX-specific configuration in PCIe space may require that the boot firmware be able to prevent the OS from re-enumerating the PCI subsystem. The boot firmware may enforce this requirement by supporting the Ignore PCI Boot Configurations _DSM method (Function 5), which should return a 0 to request preservation of boot settings. More detailed information regarding this _DSM method can be obtained from [6].

The _DSM method must be placed on the CCIX Root Ports to which the CCIX device networks are connected.

### 4.7.5 ACPI Identifiers

If CCIX components (agents, AFCs) are not implemented as PCI functions, then they may be identified using standard ACPI identifiers instead, so that operating systems can enumerate them, and device drivers can bind themselves to the appropriate components in a standardized manner.

This section provides high-level guidelines for this identification process.

#### 4.7.5.1 _HID

The _HID is optional if the CCIX components leverage any existing transport technology like PCIe that allows for bus-based enumeration. If the underlying transport implementation does not support auto-enumeration, then the _UID must be defined and used to allow ACPI-based enumeration as recommended in the rest of this section.

CCIX components may be declared with a hardware identifier, _HID, in ACPI namespace and ACPI tables. The _HID may be a vendor-specific ID or the standard ACPI CCIX Vendor ID string, "CCIXnnnn". This allows a vendor to provide their own dedicated driver for a CCIX device (e.g. for special or proprietary handling), or default to standard CCIX drivers.

#### 4.7.5.2 _CID

The _CID is optional if the CCIX components leverage any existing transport technology like PCIe that allows for bus-based enumeration. If the underlying transport implementation does not support auto-enumeration, then the _CID must be defined and used to allow ACPI-based enumeration as recommended in the rest of this section.

CCIX components may be declared with a compatibility identifier, _CID in ACPI namespace and ACPI tables. The _CID must always be the standard CCIX Vendor ID, "CCIXnnnn".

#### 4.7.5.3 _UID

The _UID is optional if the CCIX components leverage any existing transport technology like PCIe that allows for bus-based enumeration. If the underlying transport implementation does not support auto-enumeration, then the _UID must be defined and used to allow ACPI-based enumeration as recommended in the rest of this section.

For AFCs within a device, the _UID represents the AF instance number associated with an RA. For RAs, SAs and HAs within a device, the _UID may be equated to the Agent ID assigned to these agents during topology discovery.

# Chapter 5. Reliability, Availability & Serviceability

This section describes the details of how software is expected to handle CCIX errors.

When host software is notified of a CCIX protocol error, software must capture the syndrome to determine how to walk the CCIX PER logs in component structure space.

NOTE: The details of how the syndrome is captured by host software/firmware is implementation defined and outside the scope of this specification.

## 5.1 RAS Framework

The details of the CCIX RAS framework is implementation defined. This section describes high-level examples of how the framework ties together, namely the interaction between AER and PER handling.

Figure 49 describes an example flow for AER and PER errors. While AER error logging is standardized on both device and host sides, the PER log is only standardized on the device side. How the host hardware captures the PER message syndrome and notifies host software/firmware is completely implementation defined.

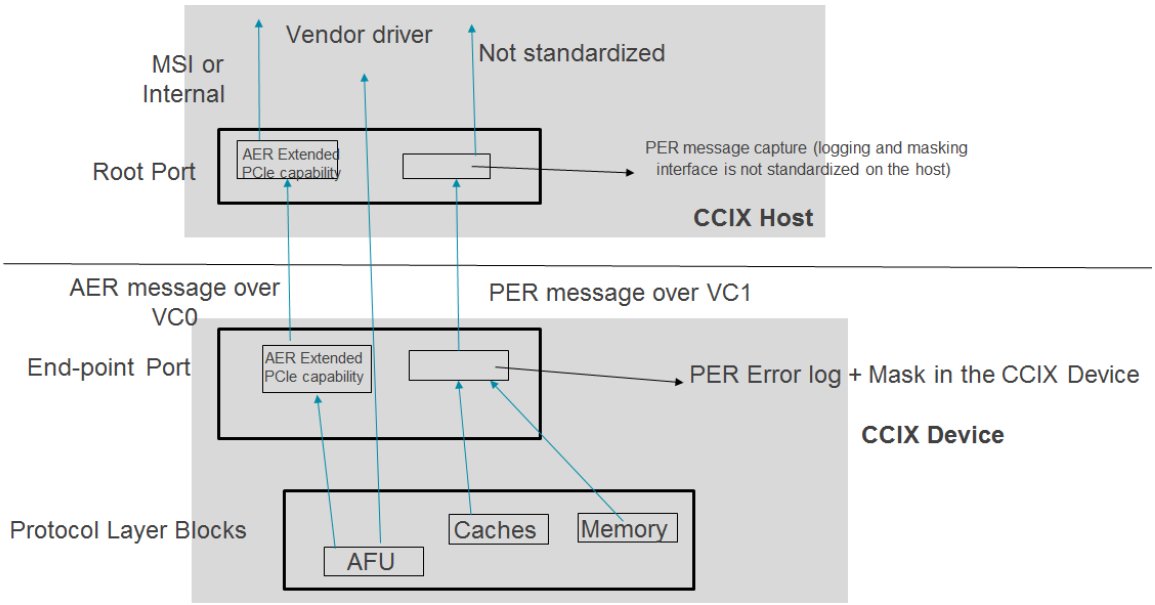



**Figure 49 Example Logging and Signaling Flow**

## 5.2 Interactions with PCIe RAS

When CCIX is used over a port that supports Downstream Port Containment (DPC), care must be taken in configuring DPC, so that the CCIX port does not bring the link down before software is able to process the necessary information from the CCIX device on the other side of the link. It is recommended that DPC is by

default disabled in a CCIX enabled system and the RAS software could enable DPC based on vendor specific solution.

When handling AER for a CCIX device, it is possible that this may result in a protocol error or may have been due to a protocol error. In this situation, it is recommended that the AER driver call back into the vendor device driver and/or call back to the standard CCIX PER driver to check for (and handle) any pending PER log. It may also be possible that there will be pending AER when doing PER handling as well.

Figure 50 illustrates an example of a host software AER handler calling back into the vendor device driver for special handling of device errors (e.g. for device recovery), as well as back to the standard CCIX PER driver to address needed system actions (e.g. for example offline relevant pages and kill impacted applications and/or virtual machines).

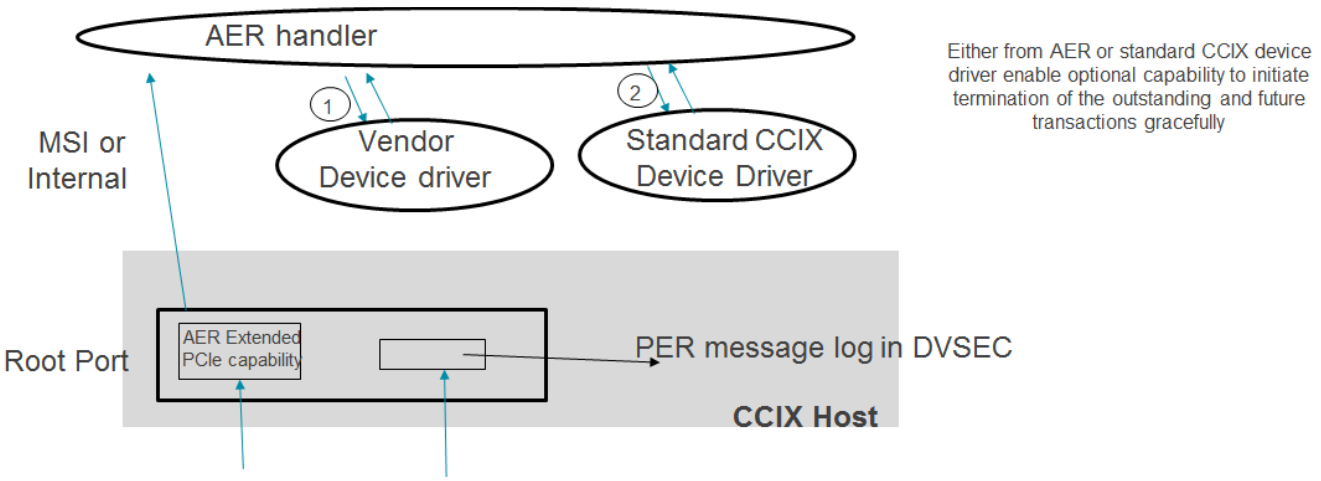



**Figure 50 Example AER Handling Flow**

The decision to notify the CCIX PER driver will typically depend on the severity of the AER error.

For example:

- In the case of a **non-fatal AER without PER pending**: everything may be handled per the traditional AER handler.
- In the case of a **fatal AER without a PER pending**, software needs to notify the CCIX handler before the AER handler resets the link in hopes of gracefully quiescing all CCIX transactions.
- In the case of a **fatal AER with a PER pending**, the AER handler should hold the link hot-reset till CCIX device has completed its recovery.
- In the case of **PER with an AER pending,** actions may vary depending on both the AER and PER error syndrome information from both AER and PER.
- In the case of PER without an AER pending, actions may vary depending on the captured PER log details (e.g. severity and address).

Figure 51 illustrates two possible implementation options for handling PER messages on the host side:

- Firmware-first error handling leveraging ACPI Platform Error Interface (APEI), described in the next Section.
- Vendor specific exception handler in software, described further in Section 5.3.

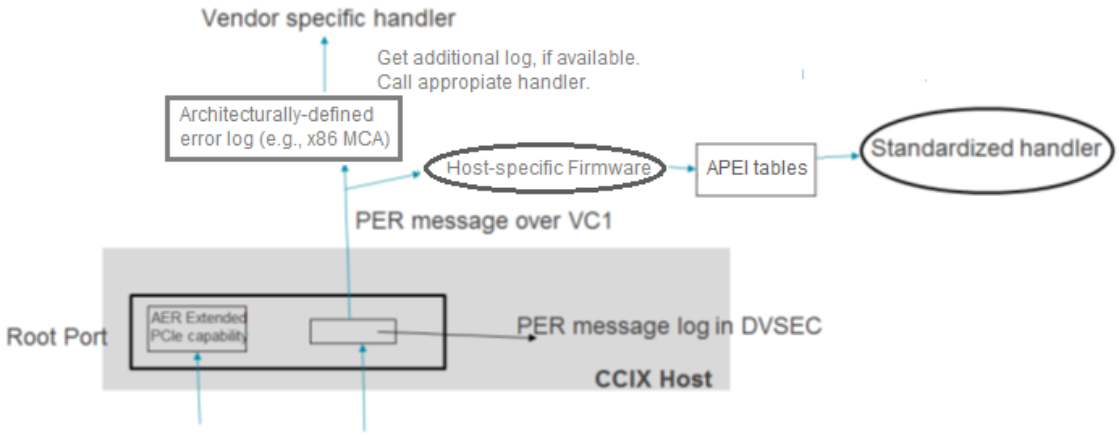



**Figure 51 Example PER Handling Flow**

# 5.3 CCIX Firmware-First Error Handling

This section focuses on how to leverage ACPI Platform Error Interrupt (APEI) based firmware-first error handling for notifying software that a CCIX protocol error has occurred. APEI is described in ACPI specification.

Figure 52 provides a high-level flow of how CCIX errors could be handled by firmware and passed on to software via APEI. In this specific example, firmware advertises a Generic Hardware Error Source (GHES) to software, which provides an error status block for communicating error syndrome info to the operating system per the UEFI Common Platform Error Record (CPER) specification.

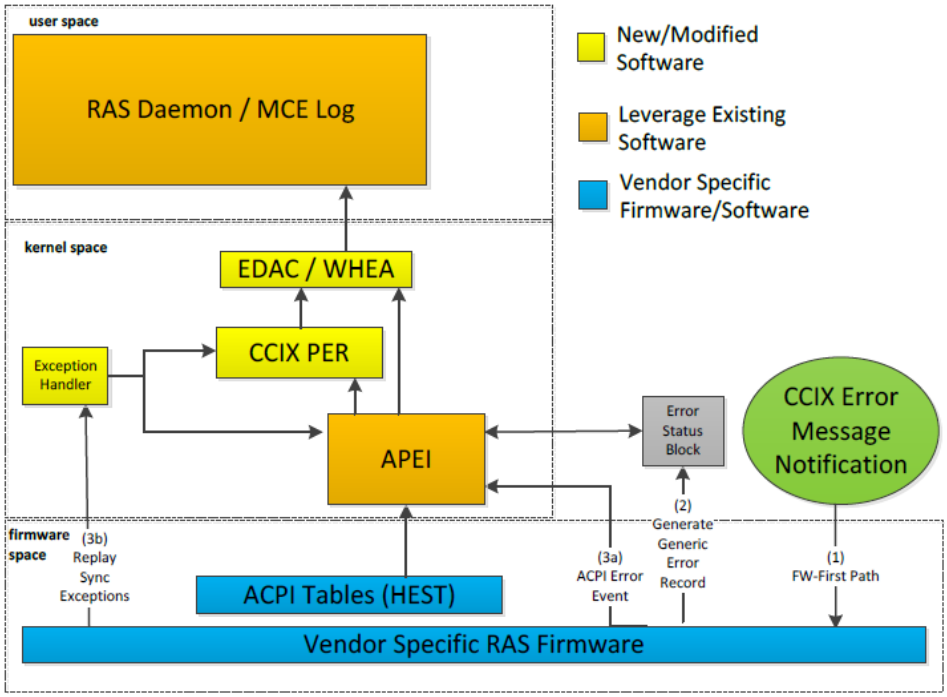



**Figure 52 Example Firmware First Error Handling Flow in Linux**

The Vendor specific firmware details are outside the scope of this specification, but the normal expected flow:

1. At boot, firmware and software will go through a handshake procedure via the ACPI _OSC method (to determine OS capabilities)
   - Boot firmware indicates that it supports CCIX PER Firmware-First handling
   - If OS acknowledges that it supports the CCIX PER driver:
     - Firmware will report CCIX PER errors as CCIX error records to be handled by the CCIX PER driver
   - If OS does not acknowledge that it supports the CCIX PER driver:
     - Firmware will translate CCIX PER errors into one of memory, cache, or bus error records to be handled by the legacy EDAC driver.
2. CCIX error message triggers implementation defined exception or interrupt that is trapped by RAS firmware
3. RAS firmware uses component structures to locate PER Log details. Generate an APEI error record per the UEFI CPER specification (Appendix N of the UEFI Spec)
4. The next action depends on whether the error was
   - **Asynchronous**: Raise an ACPI event to indicate to the APEI driver
   - **Synchronous**: May replay the exception to software, the exception handler will notify APEI driver if the type of exception is configured for "Firmware-First"
5. APEI driver queries error status block
   - If general memory/cache error: firmware pass memory/cache error record directly to EDAC
   - If CCIX PER:
     - If OS supports CCIX PER (i.e. PER _OSC bit is acked): firmware has passed info as a CCIX error record, which will be handled by the CCIX PER driver.
       - If GHES assist is indicated via the GHES_ASSIST flag, then the CCIX PER driver should consult the firmware for further information related to the PER, as outlined in [3].

- If OS does not support CCIX PER (i.e. PER _OSC bit is nacked): firmware will translate the CCIX PER message into a memory/cache/bus error record, which will be handled by the legacy EDAC driver.

6. CCIX PER driver will handle the error as described in this specification.
   - CCIX PER driver will retrieve the log based on the PER message details passed into the error status block (i.e. determine BDF based on Error Source ID and Error Port ID).
   - CCIX PER driver will acknowledge the error and clear the log via error control mechanisms described in [1].
   - CCIX PER driver will take appropriate action (e.g. log, attempt to recover the device, or crash and reboot the system in fatal scenarios).

## 5.3.1 CCIX Common Platform Error Record (CPER)

The CCIX Error Record is defined in Appendix N of the UEFI specification, version 2.8 [2].

**NOTE**: The intent of this proposed structure is to provide a means to report any PER Log errors that may otherwise not be reported by other implementation-specific means.

**Non-Standard Section: CCIX PER Log**

**Section Type GUID:** {0x91335ef6, 0xebfb, 0x4478, 0xa6a6, {0x88, 0xb7, 0x28, 0xcf, 0x75, 0xd7}}

**Table 10 CCIX Error Record Format for Firmware-First Error Handling**

| Mnemonic | Byte Offset | Byte Length | Description |
|---|---|---|---|
| Validation Bits | 0 | 8 | Indicates which of the following fields is valid:<br>Bit 0 – CCIX Source ID Valid<br>Bit 1 – CCIX Port ID Valid<br>Bit 2 – CCIX PER Log Valid<br>Bit 3-63 – Reserved |
| CCIX Source ID | 8 | 1 | If the agent type is an HA, SA, or RA: This field indicates the CCIX Agent ID of the component that reported this error. In this case bits 7:6 must be zero, since Agent ID is only 6 bits.<br><br>Otherwise, this field this specifies the CCIX Device ID (i.e. in the case of Port, CCIX Link, or device errors). |
| CCIX Port ID | 9 | 1 | This field indicates the CCIX Port ID that reported this error. Bits 7:5 must be zero, since CCIX Port ID is only 5 bits. |
| Reserved | 10 | 2 | Must be zero. |
| CCIX PER Log | 12 | 20..n | CCIX PER Log Structure DWORDs, as described in the CCIX Base Specification[1]. |

# 5.4 CCIX OS-Direct Error Handling

Operating system or hypervisor software directly handling CCIX protocol errors will be permitted, but is implementation defined, and outside the scope of this specification.

## 5.4.1 CCIX PER Handling Driver

As shown in Figure 51, the CCIX PER driver would be a new driver that is intended to handle and parse CCIX protocol errors. The driver may be invoked by the APEI (in the case of Firmware-First handling) or may be invoked by the architecture or vendor specific exception handler (in the case of OS-Direct handling).

When invoked by APEI, the PER Log structures may be retrieved directly from the error record, which would reside in the GHES "error status block", unless the error record indicates via the CCIX PER Log Valid bit in its Validation Bits field that the PER driver should retrieve the error logs directly from the specified component structures in the CCIX endpoint that is indicated using the Source ID and Port ID fields of the error record, as specified in Section 5.3.1.

The CCIX PER will parse the component address space (e.g. PRL DVSEC) to determine the error log details and pass this information up to the software/hardware error handling framework (e.g., EDAC driver for Linux).

# Chapter 6. Virtualization

A CCIX-based AF supports virtualization, where AFCs can be assigned to virtual machines in a virtualized system. AFCs are fully hardware isolatable to the owning VM. Within a VM, the AFTs implemented within an AFC may be assigned to multiple processes running within that VM.

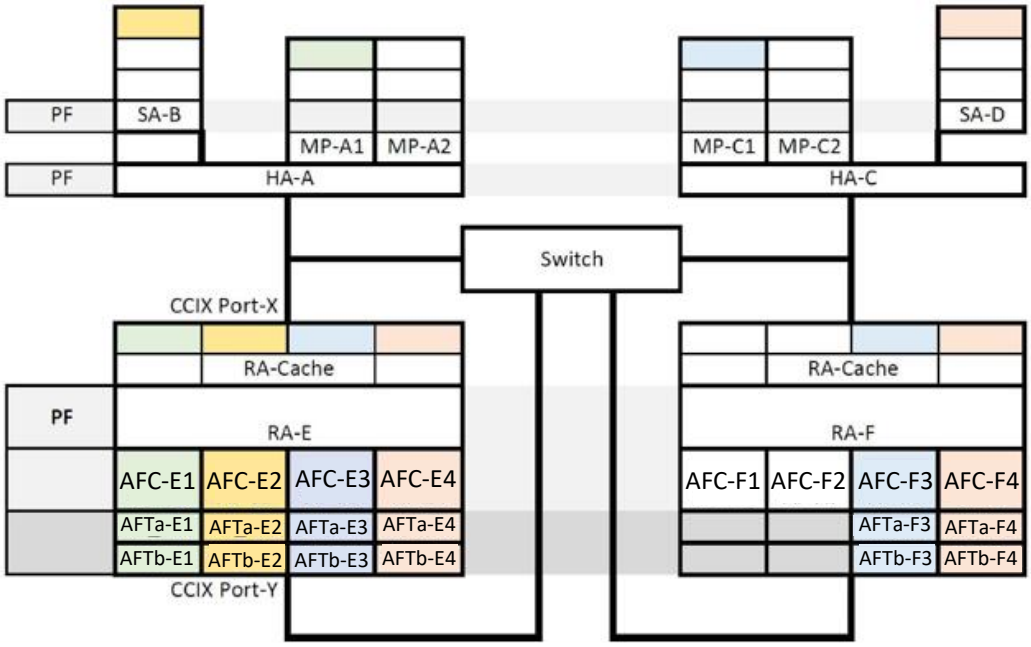



**Figure 53: Internal Layout of CCIX Components**

As illustrated in Figure 53, and elaborated in Section 7.3, the AFC/AFT architecture and the AF Property Table are used to enable discovery of AFs in a CCIX device. The discovery of AFCs within an AF is dependent on the underlying virtualization framework. AFTs within an AFC are discovered using a CCSR support.

An appropriate software virtualization framework is required for providing support for AFC drivers to bind to AFCs.

## 6.1 CCIX AF Virtualization Framework

The CCIX Generic AFV (AF Virtualization) framework provides the groundwork for enabling virtualization of AFCs. This is based on the principles outlined in Section 2.2.1, and allows assignment of AFCs and AFTs in a virtualized environment. The high-level partitioning of the AF components is depicted in Figure 54.

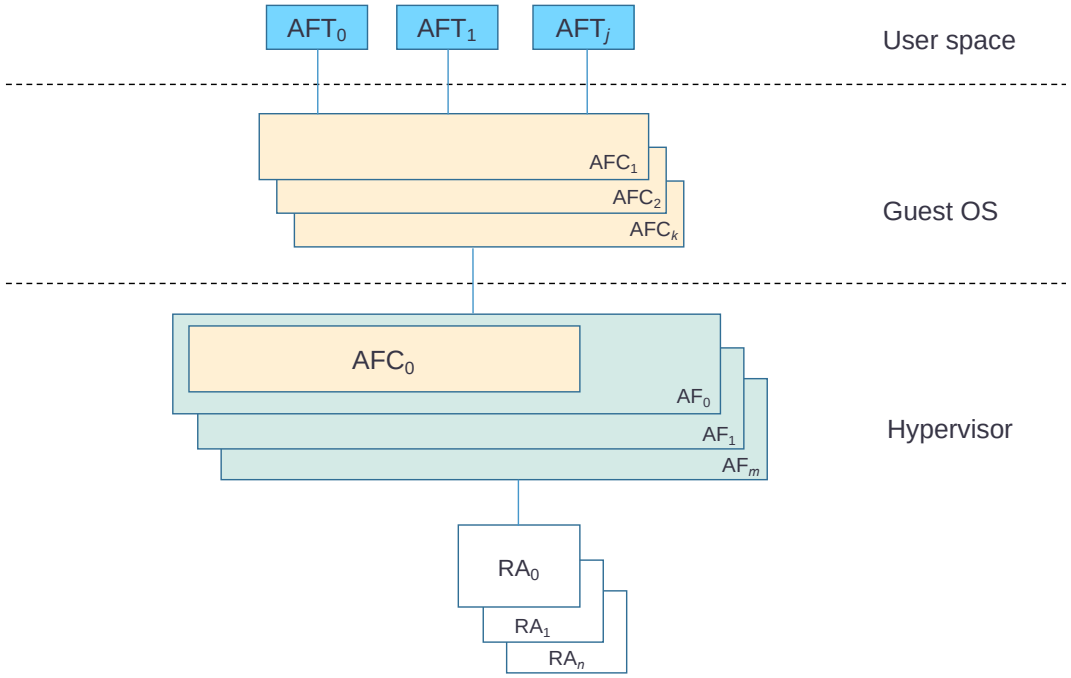



**Figure 54: AF View in a Virtualized System**

The acceleration software framework consists of a set of libraries and drivers as depicted in Figure 55.

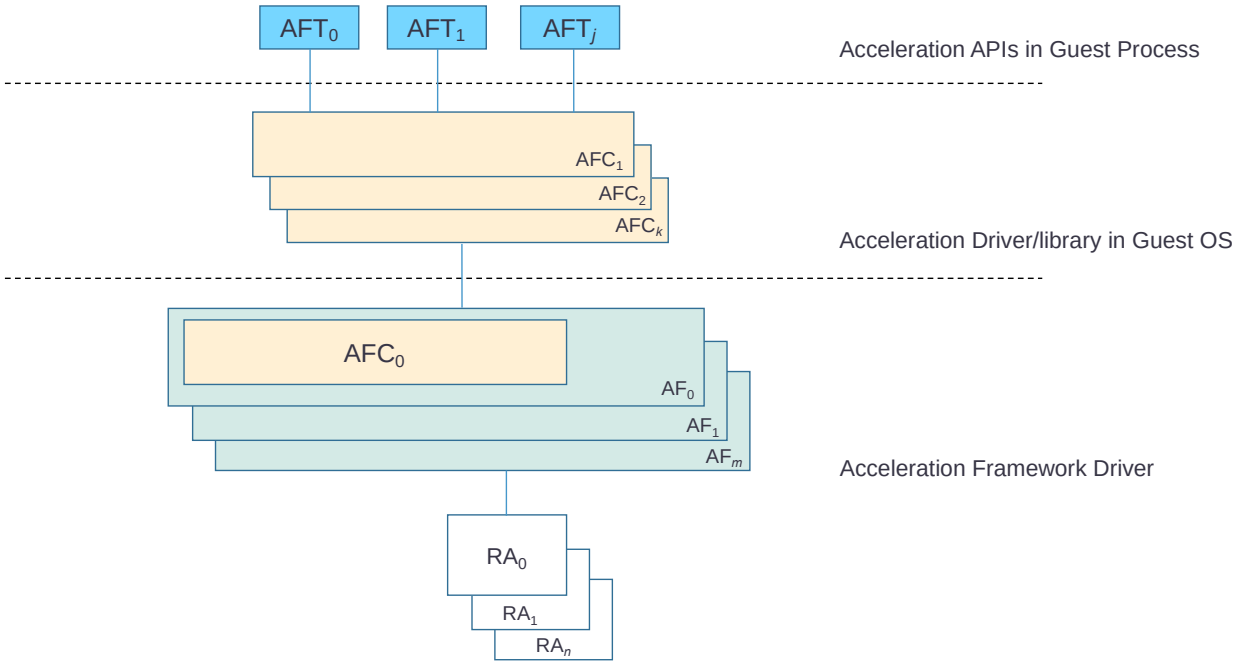



**Figure 55: Software Management of AFs, AFCs and AFTs in a virtualized environment**

The AF Management Framework driver in the hypervisor discovers AFs through the AF Property Table CCSR [1], and binds itself to the AFs. The hypervisor then assigns AFCs within an AF to guest operating systems running in VMs. Each guest OS then discovers AFTs using capability structures in the parent AFC. An AFT management library or driver in the guest OS can then assign AFTs to processes running in the guest OS.

## 6.2 AF Virtualization using PCIe and Para-virtualization

If CCIX is implemented using PCIe as transport, AFCs may be implemented as independent PFs that are para-virtualized by the VMM. The AF framework driver in the VMM is then responsible for managing allocation/deallocation of the AFCs, and for managing traffic between the VMs and the AFCs. A separate AFC driver may exist in each VM, that is para-virtualization aware, and thus works in concert with the AF framework driver in the hypervisor. This model is similar to para-virtualization of a PF driver in the hypervisor to manage the PF.

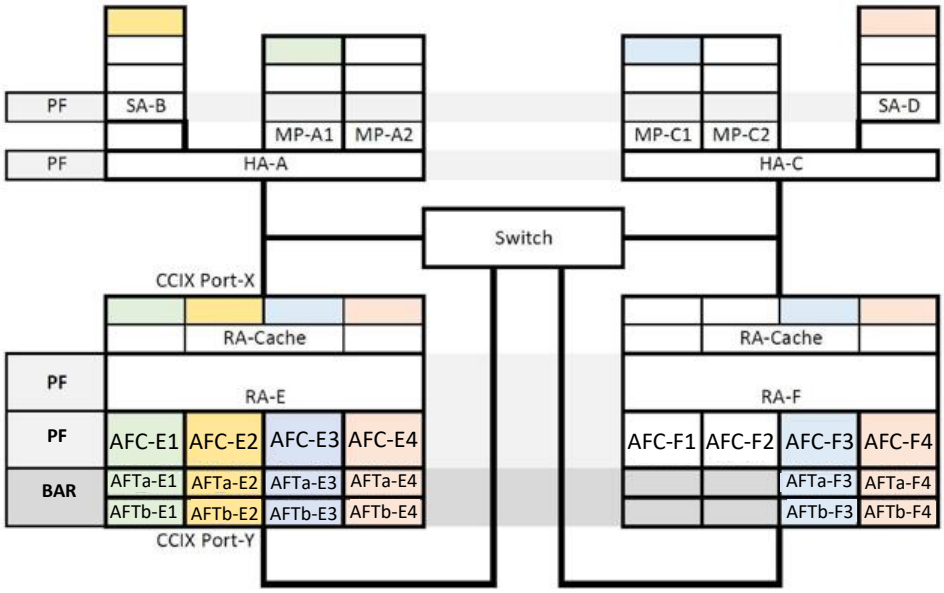



**Figure 56: CCIX Virtualization Support using PCIe**

This is illustrated in Figure 56, where AFCs are implemented as PCIe functions, and AFTs are implemented as memory-mapped device regions within those functions. AFTs may also be implemented as device contexts in PCIe configuration spaces.

## 6.3 AF Virtualization using SR-IOV

If CCIX is implemented using PCIe as transport, AFCs are implemented as PFs, and AFCs are rooted in PCIe VFs in accordance with SR-IOV. When SR-IOV is supported, the first AFC, AFC0, must always be rooted in the PF, while the subsequent AFCs may be in VFs, as illustrated in Figure 57. Software can then manage AFCs using native capabilities provided by SR-IOV. In this model, a PF driver in VMM performs allocations/deallocations of AFCs in VFs. The PF driver also performs the task of allocations/deallocations of AFCs to AF drivers in VMs. In this model, the AFCs are directly assigned to VMs.

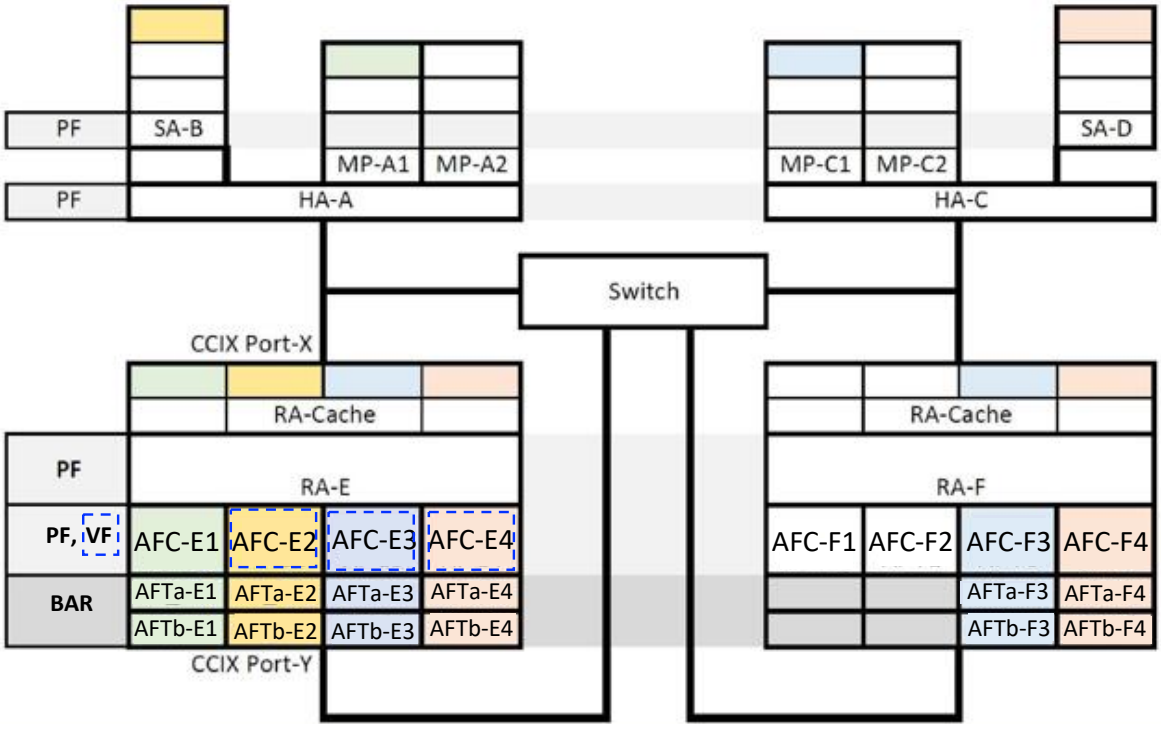



**Figure 57: CCIX Virtualization Support using PCIe SRIOV**

# Chapter 7. System Software

This chapter is dedicated to features that are accessible, initialized by or used by system software. The term system software refers to any supervisory or control software, including the Operating System kernel or Hypervisor, and OS drivers.

## 7.1 Generic Boot-time Requirements

System software is required to initialize core CCIX features during its boot-process. The specific requirements that must be met during the boot-process are outlined below:

1. PCIe ATS services must be enabled. If not supported, virtual addressing for AFs must be disabled.
2. The IOMMU must be enabled. If PRI is supported, it must also be enabled. If PRI is not supported, HA power management must be disabled.
3. PCIe AER, if supported, must be enabled based on _OSC negotiation.
4. CCIX CPER error logs reported through APEI BERT must be logged and then cleared.
5. SPM Memory must be recognized and accommodated into the OS memory manager.
6. OS re-enumeration of PCI sub-system must be disabled if _DSM support Function 5 is enforced.
7. ACPI support for HMAT and SRAT tables must be provided. This support includes awareness of GIs and creation of NUMA nodes for each GI specific proximity domain discovered.

## 7.2 Handling CCIX Specific-purpose Memory (SPM)

As explained in Section 4.7.2, CCIX memory is advertised to the OS as SPM, which means that it is conventional memory, but using this memory for kernel or general-purpose user space applications can impact performance of accelerators for whom the memory is meant for. As such, the kernel must respond to the SPM hint by earmarking SPM ranges principally for accelerator-specific usage.

Early kernel boot process must recognize SPM memory in the system memory map and accommodate it in its memory manager module, as illustrated below in Figure 58.

The boot firmware must indicate presence of CCIX memory intended to be used as SPM with a special marker. This facilitates the kernel boot process outlined above.

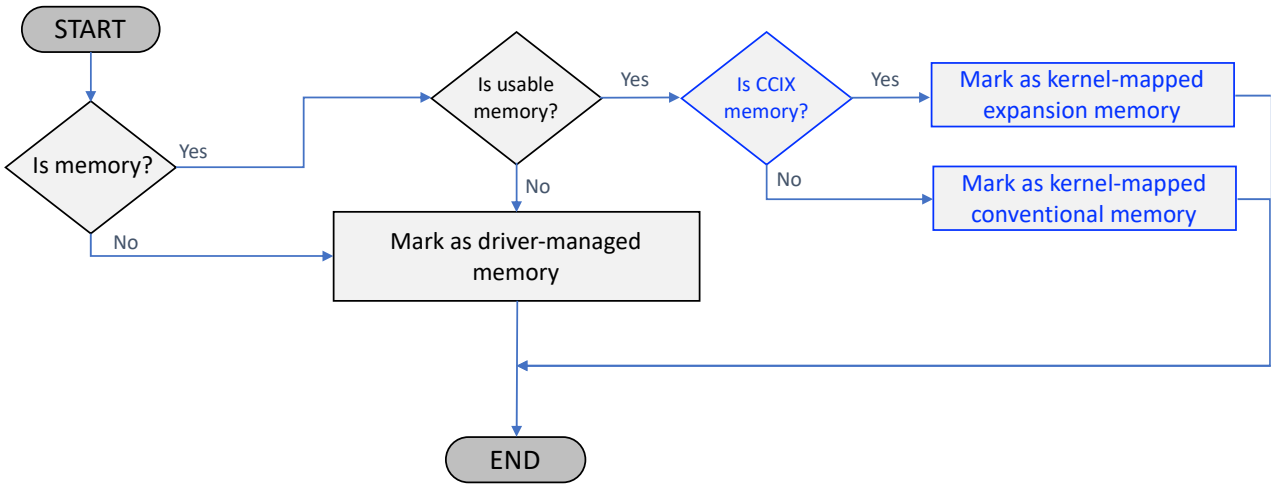



**Figure 58: Recognition and Management of CCIX Memory**

## 7.2.1 UEFI-aware Kernel Boot

UEFI Specification 2.8 [2] describes the Specific-purpose memory attribute (EFI_MEMORY_SP) as the hint that enables the OS to distinguish between regular conventional memory and SPM.

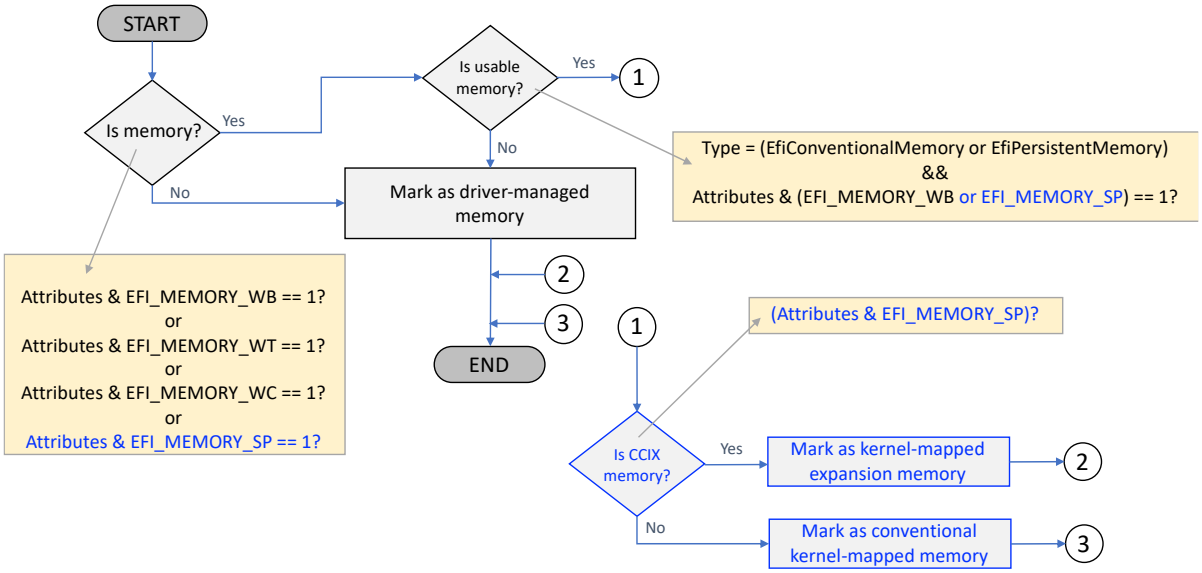



**Figure 59: Handling SPM during OS Kernel Boot**

Note that if the kernel is CCIX-unaware, then it will be unable to recognize CCIX memory (i.e. memory marked with EFI_MEMORY_SP). The boot firmware must thus mark all of CCIX memory with UEFI memory type **EfiReservedMemoryType**. The firmware decision is governed by the OS that is expected to run on the given

CCIX-based platform. When a legacy OS runs, it treats CCIX memory as reserved memory based on the UEFI memory type, and management of CCIX memory is then relegated to a device driver.

## 7.2.2 Segregation of SPM Ranges

Following SPM detection during boot, the kernel should mark pages mapped to SPM regions with an indicator that helps to steer regular kernel and user-space allocations away from this memory. This allows the CCIX accelerator drivers to claim the SPM ranges for their dedicated use. An example kernel memory management support required to enable this segregation, and memory request flows thereof, is illustrated in Figure 60 below.

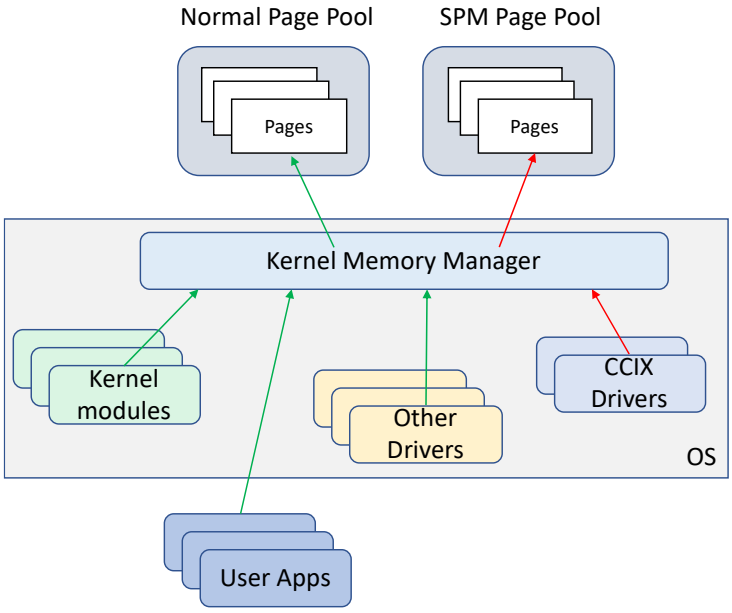



**Figure 60: Segregation of SPM Page Pools**

## 7.2.3 Claiming SPM Ranges

CCIX Device drivers, such as CCIX accelerator drivers, may bind to their dedicated SPM ranges through an assessment of the NUMA properties of the SPM ranges. To this effect, the driver may consult the ACPI NUMA tables, SRAT and HMAT, that are outlined in Section 4.7.3.

## 7.2.4 CCIX Driver Framework

In a CCIX device, the most common use-case that requires a driver is acceleration, where there are one or more AFs present in the device. The driver is responsible for the runtime management of AFCs (and AFTs) within the AFs. In CCIX 1.0a [1], AFCs present themselves as PCI functions to software. This enables a PCI-based device driver model to be implemented. The driver binds itself to the AFCs and manages them using standard PCI-based support. AFCs are discovered using the AF Property Table. The driver is required to bind to all AFCs described by the AF Property table.

The driver performs the following tasks:

- Managing SPM range allocations for the AFCs if there is local SPM in the device.
- Registering callbacks for power management events so that the AFCs, as well as their parent RAs can be power managed.
- Additionally, extend power management support for HAs if the device has local SPM.
- Registers callbacks for error events for agents local to the device.
- Configures AFCs.
- Registers driver interfaces or APIs for user-space applications (e.g. accelerator libraries) to request or relinquish access to the AFs. CCIX APIs are described in Chapter 8.
- Performs runtime power management of the AFCs in response to user-space requests, as well as device-wide power management, which could include RAs associated with the AFCs, as well as HAs that are homing SPM on the device. Power management flows are described in Chapter 9.

#### 7.2.4.1 Claiming Generic Initiators

A CCIX accelerator driver that is managing a Generic Initiator (GI) may explicitly request ownership of SPM ranges through an examination of NUMA properties of each distinct SPM and comparing them with those of the GI. The driver may bind the GI to the SPM that is closest to the GI's own NUMA domain.

System software (OS kernel) may support the CCIX accelerator driver by providing a generic system call that binds initiator and target NUMA domains.

**IMPLEMENTATION NOTE**

A sample CCIX accelerator driver initialization routine is presented below, that illustrates the steps that the driver may take to bind an accelerator to an SPM region that is closest to it.

Note that this is an illustrative example where the driver is responsible for AF to SPM binding, and existing NUMA allocation schemes can be used equivalently. For example, the kernel core may already have mapped SPM memory and "knows" which SPM range needs to be bound to which AFC.

```
ccix_afc_driver_init ()
{
    AFC_INSTANCE    *afc;
    UINT32          num_afcs, i, domain;
    NUMA_DOMAIN     *cur_numa_domain, nearest_target_domain;
    UINT32          afc_numa_domain, spm_numa_domains, total_numa_domains;
    NUMA_DISTANCE   shortest_distance = INFINITY, cur_distance;

    num_afcs = bind_self_to_ccix_afcs ();

    for (i = 0; i < num_afcs; i++) {
        afc = get_next_afc ();
        afc_numa_domain = get_numa_domain (afc);
```

```
        while (TRUE) {
            cur_numa_domain = get_next_numa_domain_from_numa_table ();
            if (cur_numa_domain == NULL) break;

            // Look for all NUMA domains that pertain to memory targets...
            if (cur_numa_domain->domain_type == NUMA_DOMAIN_TYPE_TARGET) {
                cur_distance = get_numa_distance (
                                    cur_numa_domain->domain_id,
                                    afc_numa_domain);
                if (cur_distance < shortest_distance) {
                    shortest_distance = cur_distance;
                    nearest_target_domain = cur_numa_domain;
                }
            }
        }

        bind_numa_domains (afc_numa_domain, nearest_target_domain);
    }

    // Other AFC initialization stuff here...
}
```

## 7.3 Acceleration Function Management Framework

The Acceleration Function (AF) Management Framework is a software framework that enables an OS to discover, configure and manage AFs in a CCIX-based system. As explained in Section 2.2, CCIX-based AFs include one or more AFCs per AF, and one or more AFTs per AFC. AFTs are acceleration contexts or engines that are assigned to user-space processes.

### 7.3.1 AF Discovery

AF discovery relies on a combination of standard CCIX CCSRs, virtualization framework to locate AFs in CCIX devices, and their associated AFCs and AFTs.

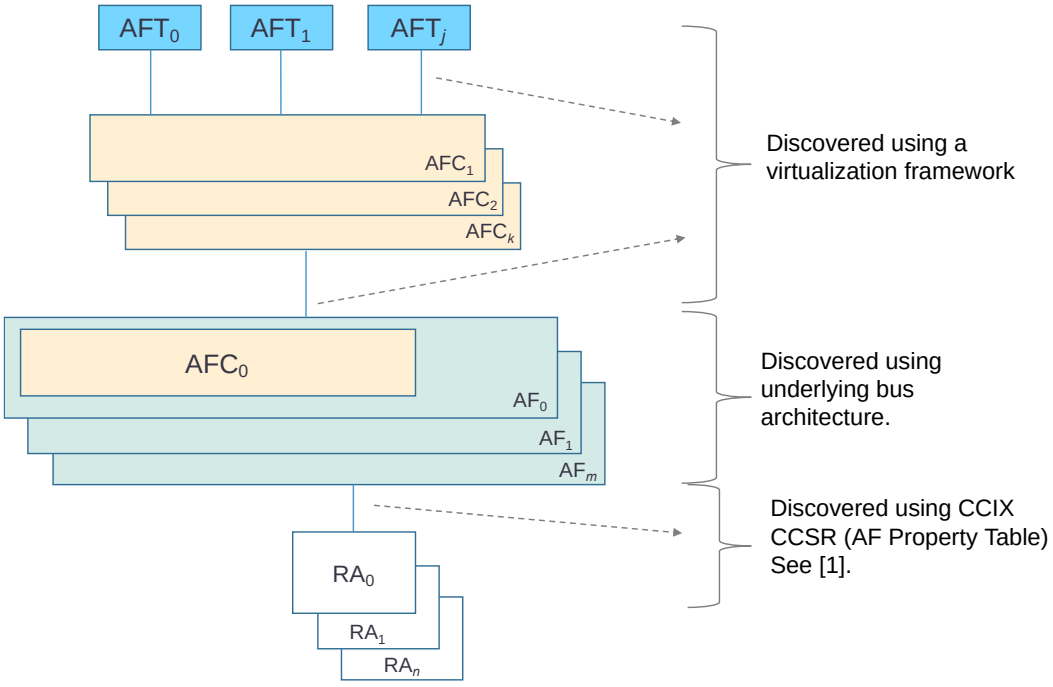



**Figure 61: AF Discovery**

### 7.3.2 AFC Management

The AFCs require an OS driver for management. Chapter 6 provides more details on AF virtualization.

### 7.3.3 AFT Management

AFTs are managed by the AFC driver. Chapter 8 provides an example of how an AFC driver might expose interfaces to allow user-processes to use AFTs for acceleration.

The AFC driver manages AFTs in an AFC. This involves the following operations:

- Enabling/disabling the AFTs
- Configuring event signaling/interrupts
- Setting up virtual addresses for the AFTs
- Assignment/mapping of AFTs to processes (PASID)

# Chapter 8. Application Programming Interface

This chapter provides an example API that applications such as acceleration libraries may use to access CCIX resources in a CCIX-based system. The example is intended to serve as an illustration of how a CCIX-aware operating system may support application layers and is thus by no means an exhaustive representation.

# 8.1 CCIX AFC Driver Interfaces

The example system is a simple CCIX accelerator device that comprises the following ingredients:

1. A single RA.
2. A single HA that homes SPM.
3. An AF with the following properties:
   - A single AFC, which contains an integrated accelerator context. The accelerator context is in the form of a bitstream that may be loaded at runtime.
   - A set of controls associated with the AFC that are available as part of its Context Management. The context management layer may be presented as a register structure in BAR0 of the AFC PF.
   - A pointer to the offset of a memory region homed by the HA in the device. The AFC accelerator context will access this memory. The pointer may be specified using the context management layer.
4. The RA functions as the front-end cache for the AFC. There is a 1:1 binding between the AFC and the RA.

An AFC driver for managing this example CCIX accelerator device will minimally need to expose the following APIs to user-space:

1. *get* – API for requesting use of the AF.
2. *alloc* – API for the user-space entity to request a region of SPM to be mapped to the AF. This allows can read and write data structures pertaining to the current application.
3. *program* – API for the user-space entity to program a bitstream into the AF and begin its execution.
4. *put* – API for relinquishing use of the AF.

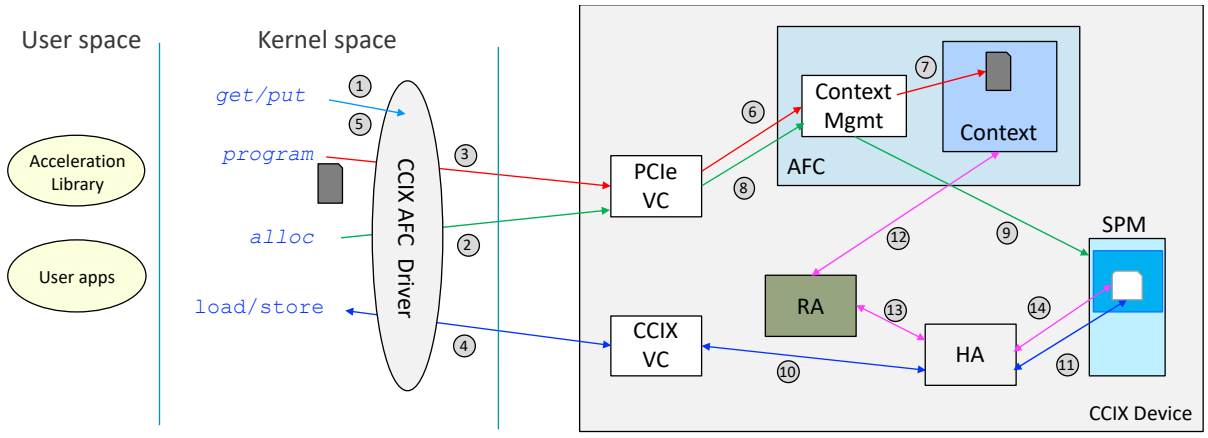



**Figure 62: Minimal set of Driver APIs for CCIX User-space Applications**

Once the driver has bound a region of SPM to the AFC, the AFC and the user-space entity can perform direct load/store operations on the memory to exchange on data that they can both operate on. Both may support virtual memory addressing, enabled through Shared Virtual Address/Memory (SVM) support in the kernel. The load/store semantics require copying data back and forth between local (i.e. user-space local) memory and SPM memory, and this happens transparently through hardware coherency support that CCIX provides.

In Figure 62, the following points are highlighted in a typical flow:

1. User-space entity requests an AFC using the *get* call. The driver reserves the AFC for the entity.

2. User-space entity calls *alloc* to request a region of memory for the AFC. The driver returns a VA pointing to the SPM behind the HA on the device.
3. User-space entity calls *program* to program a bitstream into the AFC context.
4. User-space entity uses native load/store instructions to directly read to the VA in order to communicate with the AF.
5. User space entity completes its current operation and calls *put* to relinquish use of the AF.

The following additional flows occur internally:

6. This flow is triggered by the *program* call. It involves DMA over the PCIe VC to program the bitstream. The DMA operation may be triggered by a control bit in the context management layer in BAR0. The CCIX framework driver may explicitly write this bit to control the DMA operation.
7. The DMA targets the context area within the AFC. When done, the programmed bitstream code is loaded in the accelerator context. On completion, the CCIX framework driver may set a context management bit so that the bitstream may begin executing on the accelerator.
8. This flow is triggered in response to the *alloc* call. Here, the context management layer may obtain a region of memory behind the HA as an SPM memory for use by the loaded accelerator code.
9. The context management layer memorizes the allocated SPM memory region for the currently loaded accelerator code. This information may then be latched by the running bitstream code so that it can begin operating on data passed in SPM memory.
10. This flow occurs in response to load/store operations on the SPM memory by the host (accelerator library). The operations generate CCIX memory read/write requests that flow over the CCIX VC to eventually terminate at the HA on the device.
11. The HA fulfils the CCIX memory requests issued in step 10.
12. In the interim, the bitstream code may continue to operate on the SPM memory region. These generate memory read/write requests to the RA. These requests are first translated by an ATC internal to the AF, and the AF may use PCIe ATS services to achieve the translation.
13. The RA issues CCIX memory read/write requests on behalf of the AF. These flow over the internal CCIX fabric to the HA. On completion, the RA may cache the read SPM data.
14. The HA fulfils the memory requests from the RA.

As explained earlier, this example driver is representative of the minimal support that CCIX AF driver frameworks may provide. More sophisticated interfaces or APIs may be built on top, depending on the application.

# Chapter 9. Power Management

## 9.1 Overview

CCIX device power management involves managing power state transitions of each CCIX component within CCIX devices, as well as overall device-level power management, with an eye to reducing overall system power consumption.

### 9.1.1 Power Management Use Cases

| Usage Model | Power Management Function |
|---|---|
| System events | CCIX devices and agents must be powered down in response to global, system-level power events such as hibernate, suspend and wakeup. Likewise, CCIX devices and agents must be woken up and return to full running state when the system resumes. |
| Idle detection | CCIX devices and components thereof should be moved into low power state when an idle condition is detected, viz., when a component (e.g. an AF, or an RA) is not allocated to any software consumer, or has been explicitly transitioned into an idle state. For example, AFs are dynamically assigned to a VM in a virtualized environment when direct assigned, and reclaimed when the VM is torn down or when the VM relinquishes the AF. The Virtualization Framework is responsible for informing CCIX device drivers for idle detection and power management thereof. |

CCIX power management could be hierarchical or non-hierarchical. For example, since RAs service memory requests from AFs, an RA can be transitioned to a low-power state if all dependent AFs are idle. An entire CCIX device including all its components can be power transitioned to a low-power state if all components within it are idle or unused. These are examples of hierarchical power management.

Since CCIX topologies could be non-hierarchical in nature, care is required in orchestrating the device and system-level power transitions.

### 9.1.2 Power Management Framework

The Operating System performs power management of the entire system using a hierarchical model. In this document, such a model is referred to as the *Power Management (PM) framework*. CCIX devices and agents must operate under this framework and comply with its semantics. Device drivers usually register callbacks with the PM framework to participate in system-level power management. CCIX drivers may adopt the same approach.

The PM framework handles the following two use-cases:

- Idle/Inactivity (Runtime Power Management)

- System-level power transition event (e.g. system suspend or shutdown)

The third use-case, power management associated with software allocation/deallocation of AFCs, is handled by OS drivers. Additionally, the device may perform autonomous power management. Autonomous power management is implementation specific and thus outside the scope of this guide.

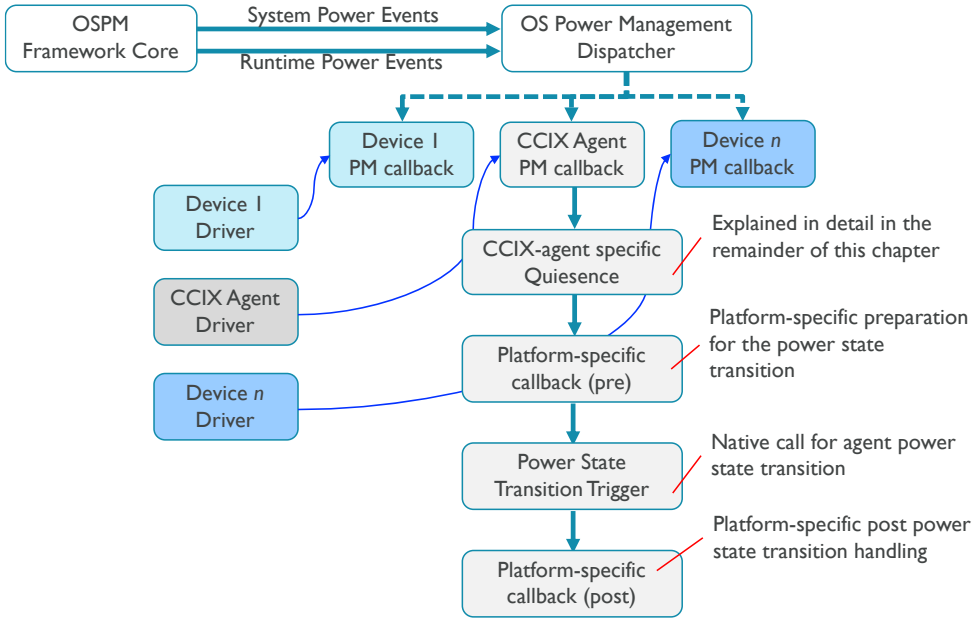



**Figure 63: Generic Power Management Framework with CCIX-awareness**

### 9.1.2.1 System-level Power Management

System-level power management involves software controlled transition of a CCIX agent to low power state to enable system-level power state transition such as system suspend. Software PM framework involves the following actions:

1. Power Management core calls agent's driver to commence transition to low-power.
2. Agent's PM callback routine in driver begins quiescence sequencing. This follows the set of operations described in the rest of this chapter.
3. Agent's device driver or the PM core may then invoke platform-specific firmware (e.g. to turn power resources off for the agent or device, as required.)
4. PM software executes power state transition. This causes the agent to enter the specified low-power state.

### 9.1.2.2 Runtime Power Management

Runtime power management involves software controlled transition of an idle component to low power state. CCIX device drivers should register themselves with the PM framework for receiving notifications, such as idle timer expiry. If a CCIX device is idle for greater than the idle time, the driver can transition the device to low power to minimize runtime power consumption. The chosen power state must satisfy wake latency criteria.

Software PM framework involves the following actions:

1. Software (e.g. driver or a generic runtime power management framework) detects that the CCIX agent (e.g. RA) is idling and determines that the agent should be put into low power.
2. Software then executes a quiescence sequence on the agent until it reaches quiescence. Since agent is idling, it would normally not be generating any transactions at this point. However, there may be buffered/enqueued transactions that would need to be retired. There could also be users of the agent that would need to be notified of the impending power transition.
3. Software next sets up exit latency requirements and/or latency tolerance limits for the agent.
4. Software executes power state transition using an architected means. This causes the agent to enter the specified low power state.
5. Wake sequences follow the reverse path. When there is a transaction targeting the agent, software needs to wake up the agent using architected means.

#### 9.1.2.3 Mapping to PCIe Power Management Framework

When mapping to PCIe, the PCI PMCSR register [9] of the PCIe PF on which the agents reside, may be used to execute power state transition of the agents (i.e. step 5 in section 9.1.2.2 , and step 4 in section 9.1.2.1 )

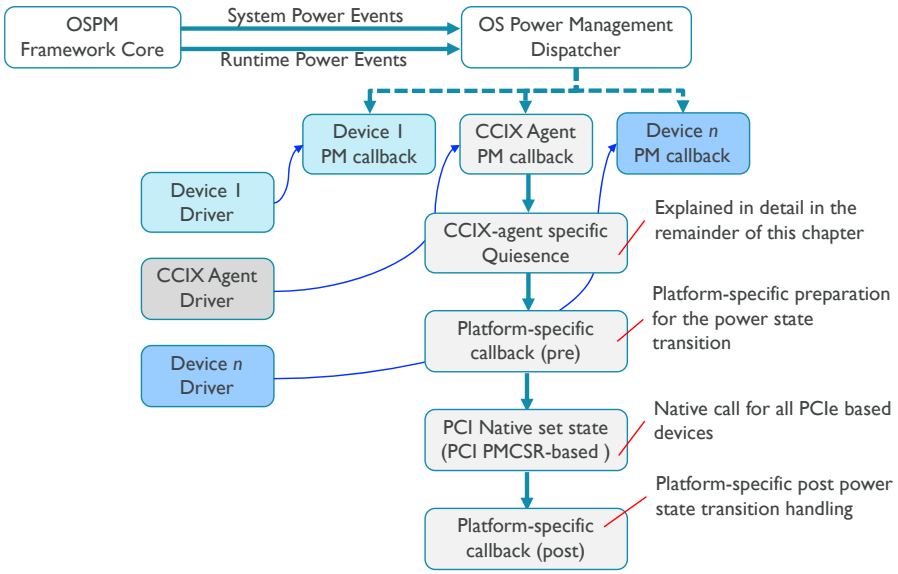



**Figure 64: Power Management Framework for CCIX Power Management based on PCIe**

### 9.1.3 Power State Dependencies

CCIX drivers are required to inform the OS PM framework of the hierarchical relationships between devices. The PM Framework then ensures that the order of power state transitions follow the reported hierarchical ordering.

For CCIX, this translates to the following implicit hierarchical sequences:

**Table 11: Power State Dependency Table**

| Dependency | Low-power Sequencing | Wakeup Sequencing |
|---|---|---|
| AFCs and RAs | AFCs first, then RAs | RAs first, then AFCs |
| RAs and HAs | RAs first, then HAs | HAs first, then RAs |
| Devices | Deepest to shallowest level, as illustrated in Figure 65. | Shallowest to deepest level, as illustrated in Figure 65. |

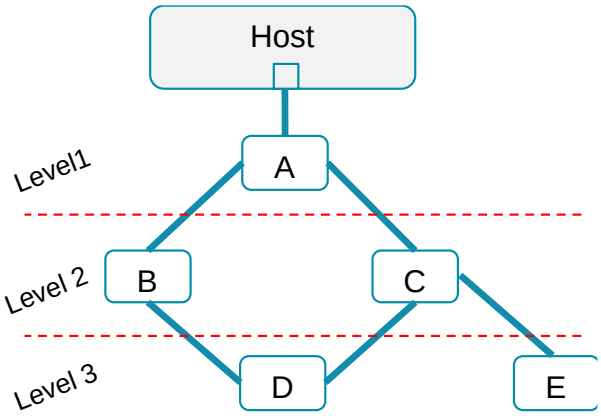



**Figure 65: Implicit hierarchy of CCIX devices in a given topology**

# 9.2 RA Power Management

An RA can be placed in specific power states to achieve power savings, based on its specific usage by software, as well as by hardware components that communicate with it during runtime operation.

## 9.2.1 RA Power States

An RA can be in one of the following logical power states:

**R0** – The RA is fully operational:

- Caches that form part of the RA can field snoop requests.
- Caches that form part of the RA can also field memory requests from AFCs associated with the RA.
- AFCs associated with the RA are enabled and can generate memory requests.

**R1** – The RA is in a low-power state and caching is fully disabled:

- A small part of the caches that form part of the RA stay operational in order to field snoop requests from HAs.
- AFCs associated with the RA are disabled or in a quiescent state.

**R2** - The RA is in a low-power state, and caching is disabled. However, snoops are still serviced:

- Caches that form part of the RA are disabled, such that they can no longer field memory requests from associated AFs.
- Caches that form part of the RA must field snoop requests with a miss response. This requires a small part of the cache logic to remain on in an internal low-power state, such that it can provide the miss response.
- AFCs associated with the RA are disabled or in a quiescent state.

**R3** - The RA and caches that form part of it are powered off:

- Caches that form part of the RA are off and do not field snoop requests or memory requests.
- Functional units in the RA are disabled or in a quiescent state.

## 9.2.2 RA Power State Transitions

### 9.2.2.1 Transitions to R1

Transitions to R1 may come from the R0 state. How this is achieved is implementation defined. There are no component structures or control registers associated with this state. The RA function might provide an RA function specific driver interface to enable entry. Alternatively, the RA hardware might also enter this state autonomously.

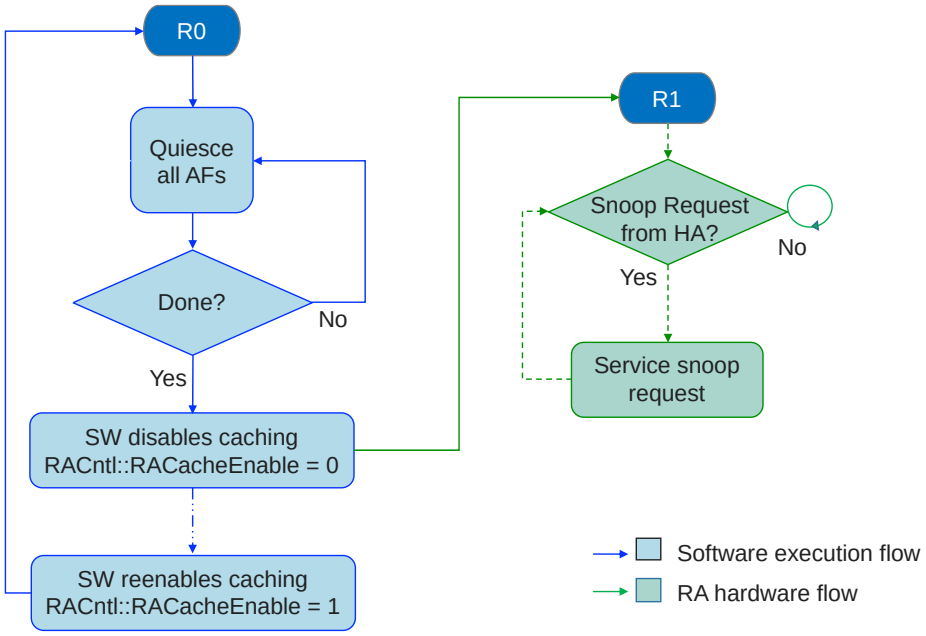



**Figure 66: Transition to R1**

### 9.2.2.2 Transitions to R2

Transitions to R2 may be architectural or implementation defined. Support for architectural transitions is mandatory and governed by RA CCSR structures.

#### 9.2.2.2.1 Architectural Entry into R2

Architectural entry is gained by using architected fields within RA CCSR structure, as follows:

1 Software disables cache allocation using the cache enable control (RACE). This stops operation in all functional units in the RA regardless of their current state.

2 Software performs a cache flush through the cache flush enable control (RAFE).

Architectural entry is guaranteed to work regardless of the initial state of the RA. This allows an operating system or hypervisor to take control of an RAs power state. This can be required if an application or VM crashes.

Note that whilst RACE is clear, and following a cache flush, a CCIX device might power down the cache, so long as the as following are true:

- Steps 1 and 2 above have been completed for all RAs that share the cache.
- The CCIX device has mechanism to respond to snoops. E.g. a port on the device may act as a proxy for the RA, generating dummy snoops on behalf of the RA.

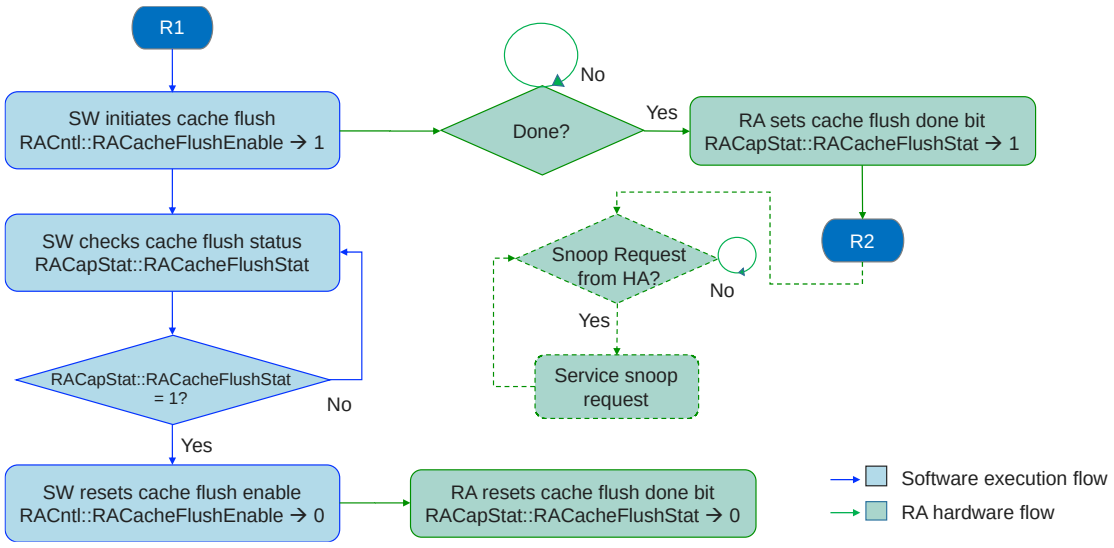



**Figure 67: Transition to R2**

#### 9.2.2.2.2 Implementation Defined Entry into R2

Power state entry mechanisms are implementation defined. The RA function might provide an RA function specific driver interface to enable this entry. Alternatively, the RA hardware might also enter this kind of state autonomously. In these cases, the entry is not reflected in the state of the RA CCSR controls or capabilities of the RA.

### 9.2.2.3 Transitions to R3

Only architected entry to R3 is supported. The R3 state is entered through an architectural transition into R2 state first. R2 state transition is described in Section 9.2.2.2.1.

In R3, caches that form part of an RA are fully powered down, including the cache miss logic that fields snoop responses in R1 and R2 states. As a cache may be shared by more than one RA, it is necessary to complete steps 1 to 2 above for all RAs that share the cache. If a cache is shared, all RAs that share the cache must transition to R3 in order for the cache to be powered down.

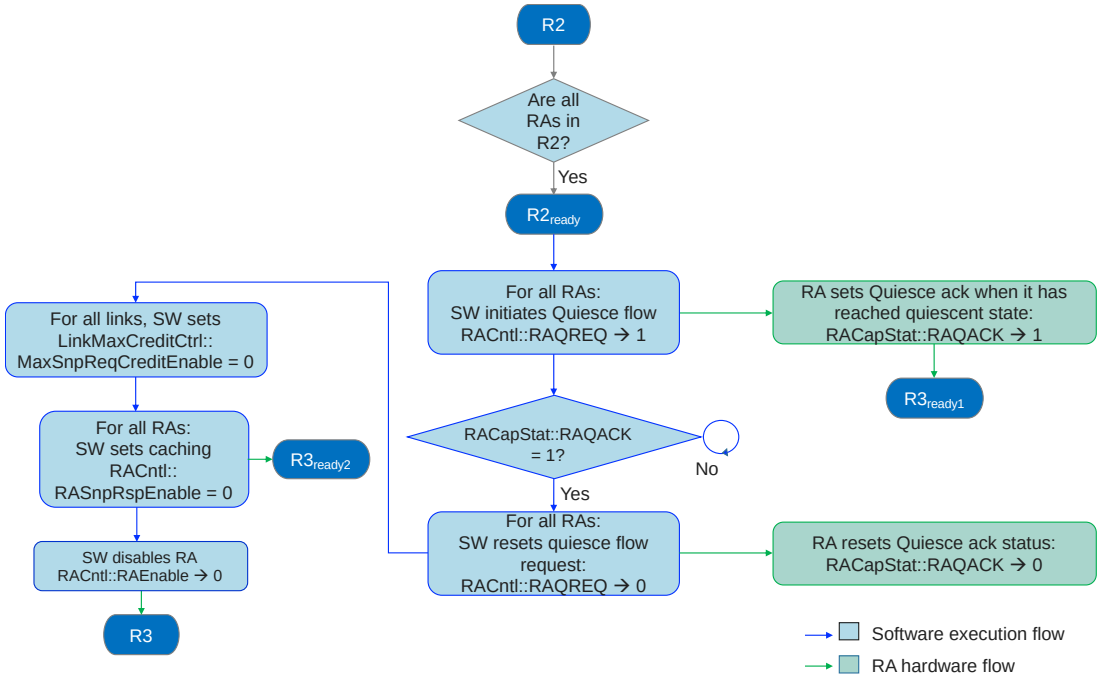



**Figure 68: Transition to R3**

Transition to R3 requires that no snoop requests arrive at an RA cache. This can be achieved by setting the max snoop request credits of all links into the CCIX device to zero, thereby ensuring that the device can no longer field snoop requests from remote HAs. This also means that if a CCIX device has multiple RAs which do not share caches, then all the RAs are required to enter R3 together. In such a device, it is not possible for a single RA to enter R3.

As illustrated in Figure 68, there are three temporary pseudo sub-states that the RA attains during its transition from R2 to R3. These sub-states, $R2_{ready}$ , $R3_{ready1}$ and $R3_{ready2}$, are preparatory to the transition to the eventual software-visible R3 state, and are not software visible.

## 9.2.3 RA Power State Dependencies

A fundamental requirement for an RA to be transitioned to a low-power state is that its child AFCs must all have all reached quiescence first. This point is illustrated in Figure 69. This When a RA powers down, it presents an opportunity for further power management events to take place "downstream" of the RA. For example, if a CCIX port has only RAs, then that port is a candidate for transitioning to low-power state. Likewise, if all RAs in a system have been transitioned to a low-power state, software may opportunistically explore transitioning the HAs in the system to a low-power state. The exact power management policies are governed by software choice and flexibility, and the application use-cases. These are highlighted in Table 12 below. An additional CCIX protocol requirement is that an RA must continue to respond to incoming snoop requests even if it doesn't have

a copy of the requested cache line in its cache. This dependency imposes restrictions on what power states the RA can transition to, and when. This key point is also captured in Table 12 below.

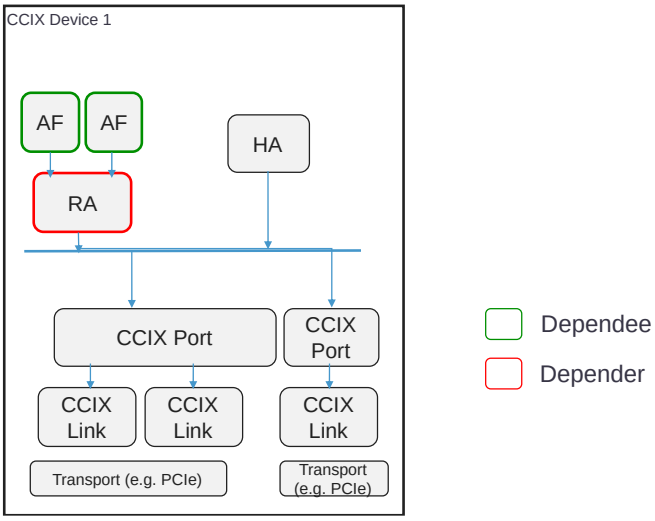



**Figure 69: RA Power State Dependencies**

The following table provides a mapping between RA power states and permissible states of other components.

**Table 12: RA Power State Dependency Table**

| Power State | AFC | RA | HA | Link |
|---|---|---|---|---|
| **R0** | **Don't care**<br>• Can generate requests. | **Active**<br>• Services all requests from AFCs.<br>• Services all snoops from remote and local HAs. | **Don't care**<br>• Can send snoop requests to the RA. | **Active**<br>• Must carry snoop responses from RA.<br>• Must service snoop requests from remote HAs.<br>• May provide early CopyBack write acks to the RA. |
| **R1** | **Quiescent**<br>• Cannot generate requests. | **Partially Quiescent**<br>• Caching is disabled.<br>• Caches belonging to the RA may or may not be flushed.<br>• Must continue to service snoops from | **Don't care**<br>• Can send snoop requests to the RA. | **Active**<br>• Must continue to service snoop requests/responses. |

| | | | | |
|---|---|---|---|---|
| | | remote and local HAs. | | |
| R2 | **Quiescent**<br>• Cannot generate requests. | **Partially Quiescent**<br>• Caching is disabled.<br>• All caches have been flushed.<br>• Must continue to service snoops from remote and local HAs. However, the responses will be a "nack" to indicate that the RA is no longer participating in the coherency domain. | **Don't care**<br>• Can send snoop requests to the RA. | **Active**<br>• Must continue to service snoop requests/responses. |
| **R3** | **Quiescent**<br>• Cannot generate requests. | **Quiescent**<br>• Caching is disabled.<br>• Snoop response is disabled.<br>• RA is off. | **Don't care**<br>• May send snoop requests to the RA. | **Don't care**<br>• May continue to service snoop request/responses from other RAs on the same CCIX device.<br>• May send a nack for the current RA |

#### 9.2.3.1.1 Quiescence

RA power state transitions require that dependent components have reached a quiescent state before the RA can be transitioned into a state in which it becomes completely dormant or off. To achieve this quiescence, software must follow the order below:

- All children AFCs of the RA must first be transitioned to a quiescent state. The software driver managing the AFCs may achieve this by retiring all pending transactions from the AFCs, and then prohibiting the AFCs from accepting any further requests.
- Software must then prepare the RA to enter low power. This involves preventing any requests from AFCs and then flushing all cache lines from the RA.
- Cache CopyBacks from the RA arising from the flush operation are transmitted as memory write requests. It is possible for intermediate ports to enqueue these memory requests and send an early ack to the RA. For example, the port may do so for performance reasons during normal operations. For RA power transitions, thus, software must program the ports to begin retirement of all such enqueued memory requests to ensure that they are committed to the destination memory. The port then sends an acknowledgement on completion.

At this point, the RA can begin transition to low power state.

## 9.2.4 Mapping to PCIe

### 9.2.4.1 Power States

When implemented as a standalone PCIe function, an RA's architectural power states are mapped to PCIe power states D0 through D3.

## 9.2.5 Resets

An RA can be reset to bring it to a default state. Following the reset, the RA will:

1. Retain its pre-reset configuration
2. Have all cache lines in the invalid state
3. Clear all stale status fields

### 9.2.5.1 Mapping to PCIe

When implemented as a standalone PCIe function, RA reset is mapped to PCIe Function-level Reset (FLR).

# 9.3 AFC Power Management

AFCs are a key component in a CCIX device from a power management standpoint, since they are the only resources that are assigned as a unit to software entities at runtime.

## 9.3.1 AFC Power States

AFCs must minimally support two states – AF0 and AF3, that map to the ON and OFF states respectively. Other intermediate states may also be defined in an implementation-specific manner.

## 9.3.2 AFC Power State Transitions

### 9.3.2.1 Transitioning to AF0

The AFC design is implementation-specific, and hence there are no architected means of putting an AFC in the AF0 state. Software will have to follow any implementation-defined methods specified by the vendor to initiate state transitions.

### 9.3.2.2 Transitioning to AF3

The AFC design is implementation-specific, and hence there are no architected means of putting an AFC in the AF3 state. Software will have to follow any implementation-defined methods specified by the vendor to initiate state transitions.

### 9.3.3 AFC Power State Dependencies

Software is responsible for making sure that any software entity that is using an AFC must be quiesced prior to moving the AFC to a low-power state.

An RA cannot be transitioned to a low-power state unless all its associated AFCs have first been transitioned to a low-power state. In the reverse path, the AFC cannot be transitioned to the AF0 state until all RAs that field its memory requests have first been transitioned to the R0 state first.

### 9.3.4 Mapping to PCIe

When implemented as a PCIe physical function, the AFC is power managed using standard PCIe power management mechanisms. For example, the AF0 and AF3 states may be equated to the PCI/ACPI D0 and D3 respectively. Additional implementation-specific states, if any, may be mapped to D1 and D2 states. These states may be initiated by software writing the desired state to the PCI PMCSR structure of the AFC [9].

## 9.4 HA Power Management

PM software may opportunistically move an HA into a low-power state if there are no users of the memory it is homing. HAs usually provide memory that is used system-wide (e.g. expansion memory in use by one or more CCIX and other kernel drivers). As a result, power management of HAs may be only performed for system-wide events.

### 9.4.1 HA Power States

HA must minimally support two power states – H0 and H3, translating to the ON and OFF states respectively. In the H0 state, the HA is in an active state, which means that it is participating in the coherency domain – sending snoop requests and servicing memory requests. In the H3 state, the HA is in a low-power state. It is no longer participating in the coherency domain. Memory behind the HA must be quiesced before the HA can transition to this state, and any allocated memory behind the HA must be migrated to an alternate location prior to the quiescence.

### 9.4.2 HA Power State Transitions

#### 9.4.2.1 Transitions to H0

The HA enters H0 state when software writes to the HA Enable bit in the HA CCSR structure.

#### 9.4.2.2 Transitions to H3

The first step involved in HA low-power state transition is a page migration process, where all pages homed by the HA are migrated to an alternate destination HA. Upon the completion of the page migration process, the HA is deemed to be in an idle state. At this point, PM software is required to transition the HA to a quiescent state by flushing out any pending transactions in the fabric that the HA may have issued. At this point, the HA is deemed ready to be transitioned to a low-power state. These steps are outlined in Figure 70.

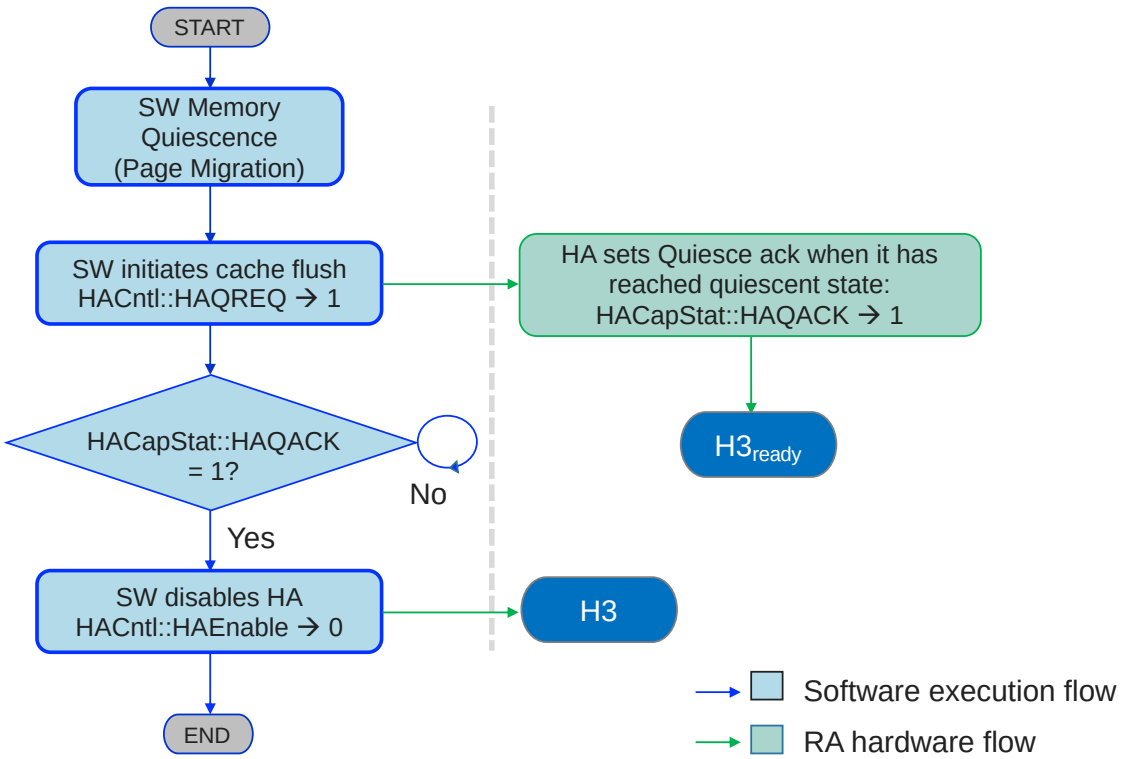



**Figure 70: HA Power Management Flow**

**IMPLEMENTATION NOTE**:

An example page migration process is outlined below:

- Software identifies a destination HA or memory region for migration. The destination region must be at least as large as the memory being homed by the HA.
- Software then marks all pages that are homed by the HA as "pages under migration".
- Software next clears the TLB to force a TLB miss on translation requests.
- Software then begins migration of the pages marked "under migration". For this, software must perform the following steps in order of appearance:
  - Invalidate page table entries for each of these pages.
  - Next, remap the pages to the destination physical memory (on the selected destination HA)
  - During this process, PRI support may be leveraged to provide migration on demand. Otherwise, devices requesting memory from the current HA may be stalled until device pages have been migrated to the new destination.
  - The HA must remain in H0 state (active) throughout this migration process.
- Once migration is complete (i.e. there are no more pages that are marked "under migration"), software can begin transitioning the HA to low-power.

The above steps ensure that there exist no users of memory behind the HA. Exact implementations, and steps thereof, will vary from platform to platform, and from OS to OS.

### 9.4.3 HA Power State Dependencies

Software must consider the following dependencies when managing HA power states:

- An HA cannot enter a low-power state unless all RAs are also transitioned to low-power state first. This point is illustrated in Figure 71.
- A CCIX device with HAs cannot transition to low-power state until all the HAs have reached H3 state first.

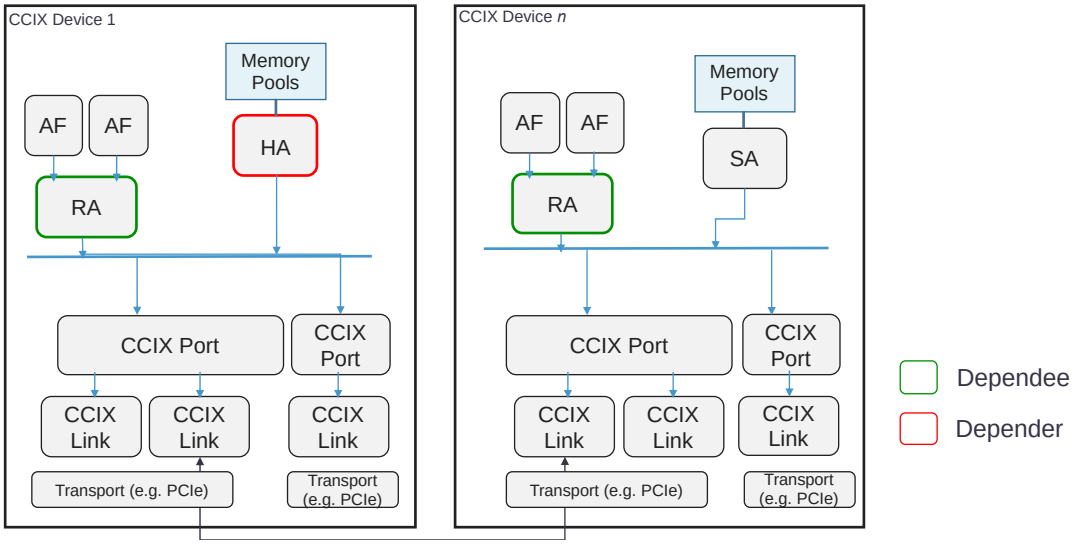



**Figure 71: HA Power State Dependencies**

### 9.4.4 Mapping to PCIe

When the HA is implemented as a standalone PCIe function, the H0 and H3 states map to PCI D0 and D3 states respectively.

## 9.5 SA Power Management

PM Software may move memory ranges into low-power state based on two conditions:

1. Implementation-defined power management capabilities advertised by the SA. For example, self-refresh.
2. Architected entry based on low-power state transition of the parent HAs.

Condition 2 is more common and can occur when the parent HAs of the SA transition to low-power state. SA power state transitions are thus a by-product of the memory migration flows pertaining to HAs.

### 9.5.1 SA Power States

An SA can be in one of two power states, SA0 and SA3, corresponding to the ON and OFF states respectively. Other intermediate states are also possible but are relegated to implementation choice.

### 9.5.2 SA Power State Dependencies

- An SA cannot transition to a low -power state until all its parent HAs have first been transitioned to low-power states. This point is illustrated in Figure 72, where an example system is shown that has two HAs, HA1 and HA2, that share the memory pool on the dependent SA.

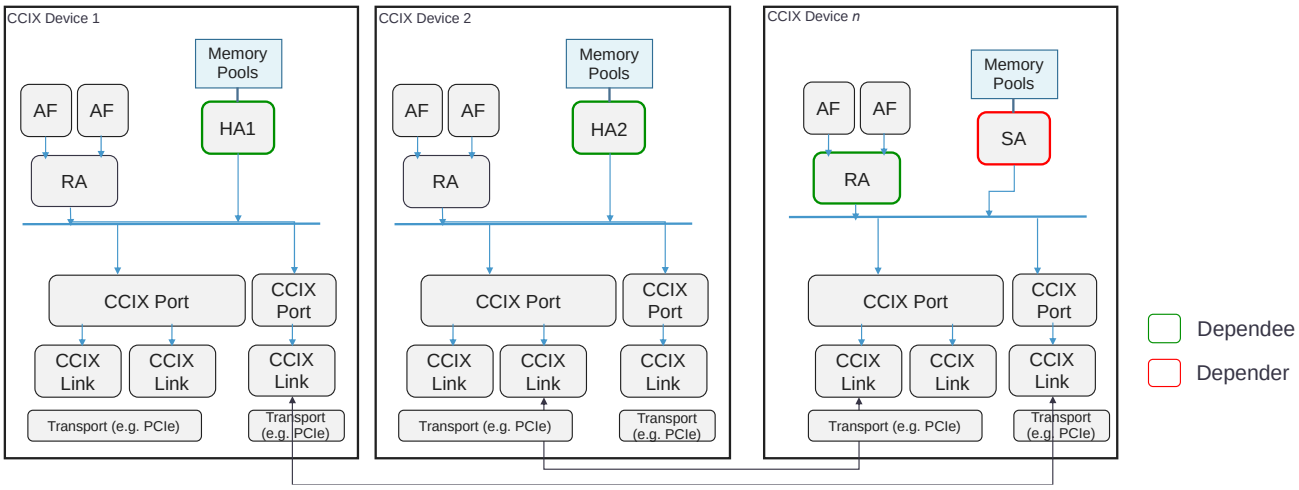



**Figure 72: SA Power Management Dependencies**

- A CCIX device with only SAs cannot transition to low-power state until all the SAs have first reached SA3 state.

### 9.5.3 Mapping to PCIe

SA power states SA0 and SA3 map to PCI/ACPI power states D0 and D3 respectively. Other intermediate states can also be defined, but they are implementation defined and managed as such.

## 9.6 Device Power Management

Device power management involves transitions of a CCIX device to low-power states. In a low power state, the entire device is in a low power quiescent state where it neither generates nor receives any transaction. Device power management must be carefully orchestrated by PM software because an intermediate device in a CCIX network provides routing for other devices.

## 9.6.1 Links, Ports and Device Power States

Ports must minimally support two distinct power states, P0 (Active or On) and P3 (Quiescent or Off). Link power management commences as soon as the port enters P3.

### 9.6.1.1 Link Quiescence

Link quiescence involves draining all pending/enqueued transactions from the link, and then ensuring that there are no more credits at the peer for any communication to ensue. At this point, the link is said to be in a quiescent state. Software must trigger quiescence sequencing on the link prior to transitioning the device or port or link into a low power state. The Link CCSR structure provides a control bit for software to request quiescence. The link sets a corresponding status bit as an acknowledgement, when it has reached quiescence.

## 9.6.2 Mapping to PCIe

When PCIe is used as the transport, a CCIX port resides above a PCIe port. The link in such a case is the collection of logic that pertains to a given link below the CCIX port, which establishes the link between the current port and a remote port. A CCIX link's power management leverages the PCIe link power management machinery. Like in the case of PCIe, therefore, a CCIX link autonomously enters a low power state when the port's PF0 transitions to a non-D0 state.

## 9.6.3 Device Quiescence

At the device level, there is a dependency on agent quiescence and port quiescence. The sequence is as follows:

- Agents within the device are quiesced in the order of their power state dependencies, in accordance with Table 11.
- When the last agent is quiesced, the port must be quiesced. This follows the port quiescence flow outlined in section 9.1.3.
- Once the port is quiesced, the device quiescence begins. This follows the inter-device power state dependency rules outlined in Table 11. The device drivers must work in concert with the PM framework in order to ensure the correct hierarchical sequencing of devices within the CCIX network. An example power state sequencing is outlined below, where devices within a mesh topology are quiesced in the order of their depth or level (number of hops) from the perspective of the host in the CCIX graph, as depicted in Figure 65.

# 9.7 Hot-plug

CCIX 1.0a [1] supports memory and accelerator on-lining or hot-add. ACPI-aware operating systems can use ACPI methods to achieve hot-add.

## 9.7.1 Memory On-lining

Memory on-line process caters to two distinct use-cases:

- Capacity on demand: where expansion memory is brought online at runtime in response to a demand for increased capacity.
- AF on-line/offline: where SPM memory that is dedicated to an accelerator is brought online or offline along with its paired AF.

These are achieved using existing means available to standard operating systems. Standard memory on-lining flows are followed for both use-cases. A brief outline is provided below:

- Boot firmware discovers memory as outlined in Chapter 3.
- For Capacity on demand, boot firmware might declare ACPI memory device objects (PNP0C80) for SPM ranges that are intended for capacity addition. This decision might be based on a platform policy.
- The memory device objects shall carry two methods:
    - _PS0, to power up the device.
    - _PXM, to associate the memory device with a memory range described by SRAT.
- The memory ranges are then declared in SRAT with the hot-pluggable bit set. The SRAT entries are then marked as disabled to indicate that these ranges are offline.
- During and after OS boot, the OS refrains from allocating from the offlined SPM.
- To online the SPM range(s), a special CCIX memory driver can be loaded in the system. The driver executes _PS0 on the memory device object associated with the SPM range of interest. This internally causes the memory to come online.
    - NOTE: In response to the onlining process triggered by the driver, the OS must perform initialization of all its kernel data structures required to bring the memory fully online. This is a standard OS flow, but is beyond the scope of this CCIX guide.

The above flow may vary from OS to OS. Also, the flow depends on use of ACPI methods (_PSx) to achieve online/offline. Other methods are also possible.

### 9.7.2 Accelerator (AF) On-lining

AF on-lining might involve two distinct use-cases:

- AF Activation/Deactivation - involves bringing an AF online at runtime when software uses it, and then taking the AF offline when software no longer needs to use it.
- Dynamic Reconfiguration – a typical use-case involving AFs is FPGA dynamic reconfiguration, where the bitstream of an FPGA behind the AF is updated during runtime.

In both cases, any SPM memory regions that have been earmarked for use by the AF might also be brought online or taken offline to support the AF, as already pointed out in Section 9.7.1.

# Chapter 10. Security

There are three specific security related use-cases that apply to CCIX systems:

- Providing a secure execution environment – creation of secure realms within the CCIX network.
- Memory Protection – protecting memory regions from illegal accesses.
- Providing device attestation – trusting a CCIX device or endpoint.

CCIX 1.0a [1] does not cover device attestation. Implementors are free to design their own device attestation schemes, but these are beyond the scope of this guide.

## 10.1 Secure Execution Environment

CCIX secure execution environment involves enabling realms that are only visible or accessible to authenticated entities. In the context of CCIX, a secure execution environment may involve:

1. Enabling secure memory regions that can be reached only via secure routes.
2. Providing a trusted compute base in which one or more CCIX devices may be grouped.
3. Terminating non-secure accesses to secure regions with a NOP or equivalent response.

## 10.2 Memory Protection

CCIX messages exchanged between CCIX agents, such as CCIX memory requests, always carry physical memory addresses. The SAM tables also carry physical memory addresses.

At the same time, it is desirable that AFCs are capable of working with virtual addresses (VA), owing to the following reasons:

- Programming paradigms require that accelerators work with virtual addresses, just like their CPU counterparts. For instance, SVM enables CPUs and accelerators to share a virtual address space, reducing programming complexity and improving data sharing.
- Memory protection can be supported through virtual addressing. In a virtualized environment, for instance, a VM is guaranteed to have exclusive access to its own memory. At the same time, the VM can be prevented from accessing memory belonging to other VMs.

Memory protection thus is a fundamental security requirement, where the CCIX endpoint hardware must guarantee that:

- AFCs cannot bypass RAs and generate PAs directly to destination home agents.
- AFCs only submit PAs to RAs.

These requirements are illustrated in Figure 73.

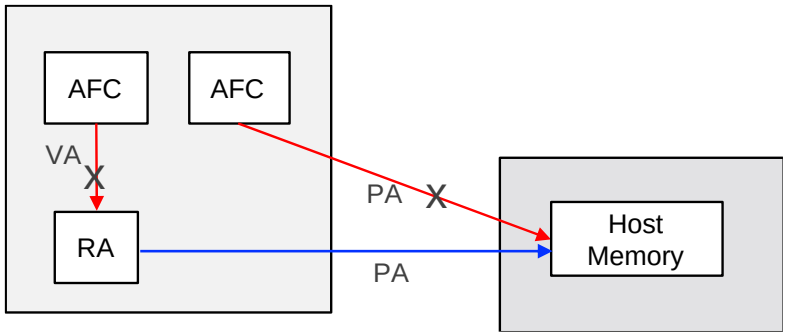



**Figure 73: Illegal Memory transactions**

When an AFC generates a request to access a VA, it must first be translated to a physical address (PA). This translation can be performed any underlying machinery that the implementation may provide, such as:

1. PCIe Address Translation Services (ATS)
2. An in-built Memory Protection Unit (MPU)

## 10.2.1 ATS-based Memory Protection

This scheme relies on PCIe ATS, which is realized using a combination of a system-wide IOMMU, and PCIe ATS capability in the CCIX RA. In this scheme, when an AFC generates a request to read host memory, it emits a Virtual Addresses (VA) pertaining to that memory. The AFC then uses PCIe ATS to translate the VA into a PA. The ATS services may in turn consult a remote IOMMU, which in turn performs the translation. The AFC may also choose to first consult a local ATC, if implemented, to optimize the translation. The resultant PA is then forwarded to the RA that front-ends the AFC. The RA then transmits a CCIX message directly to the destination, bypassing the IOMMU. This scheme is outlined in Figure 74.

CCIX software must follow the logic in Table 13 in order to enable ATS-based memory protection. Note that this means that, for PCIe- based implementations of CCIX, both ATS services and an IOMMU are mandatory for supporting memory protection.

**Table 13: IOMMU and ATS Requirements for CCIX Endpoints**

| IOMMU Present? | ATS Supported? | Supported Addressing |
|---|---|---|
| No | x | PA |
| Yes | Yes | VA |
| Yes | No | PA |

Accordingly, software must enable the IOMMU and ATS services prior to use of virtual addressing.

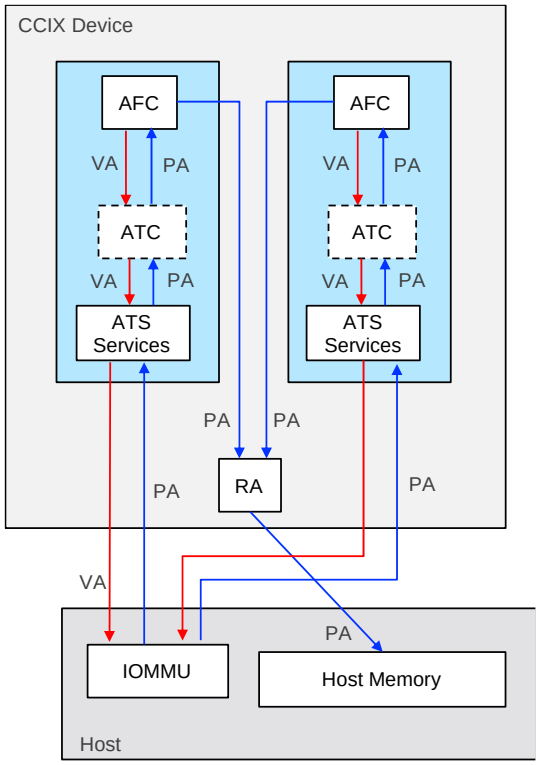



**Figure 74: PCIe ATS-based Memory Protection**

## 10.2.2 MPU-based Memory Protection

Some systems may not support an IOMMU. At the same time, some CCIX devices in the system may carry a local MPU that can translate VAs to PAs. In such cases, software can choose to enable virtual addressing and leverage the device-specific MPU. However, this is performed in an implementation defined manner and is hence outside of the scope of this guide.

# Chapter 11. Appendix

This chapter is devoted to providing miscellaneous information specific to software enablement of CCIX systems, such as:

- Reference implementations to showcase how a CCIX enabled system may be realized in practice, and how it can be supported in software.
- Guidelines that may be followed in actual practice for certain aspects of a CCIX system

## 11.1 Routing and Memory Addressing

This section provides guidelines for configuring CCIX memory maps and topology-specific information in a CCIX-enabled system and programming routing logic accordingly. The section serves as a useful guide for system designers and firmware developers to understand how to optimize addressing and routing in CCIX system.

### 11.1.1 Guidelines for Memory Addressing

Address mapping follows from the optimized routing paths. This ensures that all memory in the system is addressed and decoded as efficiently as possible. For a given CCIX device, all memory pools on the device are combined into a larger pool, and then mapped to a unique region in system memory map. This guarantees a unique SAM window in the G-SAM for the device.

When a routing path involves one or more intermediate devices, memory pools on that path may logically be combined into a single contiguous logical pool that can effectively be reached using a single routing entry from the origin of the path. This arrangement thus minimizes routing table entries, and reducing design costs.

In the example topology depicted in Figure 75, it is easy to see that device A will require only one routing entry in its routing table on port $P_1$ in order to address all of memory belonging to devices B, C and D together, if they are coalesced into a single pool. This is because all the memory below is coalesced and arranged as a single large contiguous logical pool in the system address map.

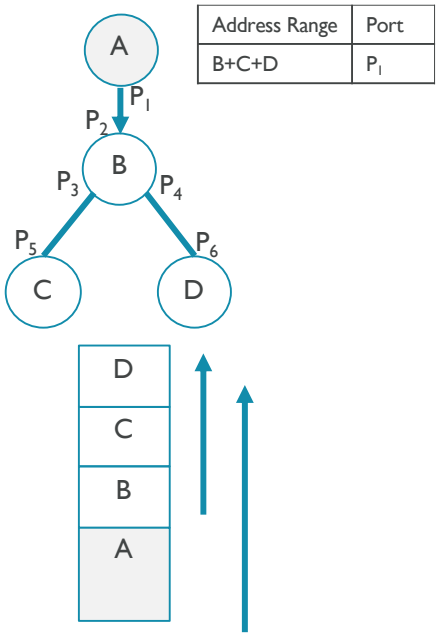



**Figure 75: Example Topology Depicting Address Coalescing Requirement**

If the same topology involves a non-contiguous, arbitrary arrangement of memory pools in the system address map, as depicted in Figure 76, then this results in a non-optimal configuration where device A must now have two entries in its routing table to be able to reach all other devices in the system.

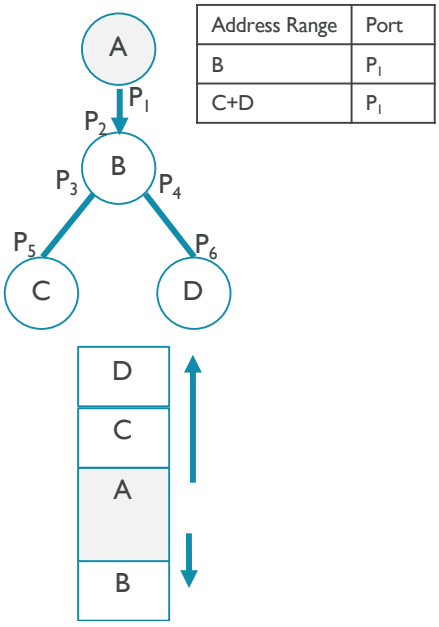



**Figure 76: Memory map with non-optimized address decoding**

## 11.1.2 Guidelines for Routing and Addressing

The routing paths from each device in the system are required to following a set of basic rules:

- The routing paths must guarantee absence of loops and deadlocks.
- For a typical topology, a destination address or agent must be reached using the least number of hops in the CCIX network. The firmware may choose an alternate path with more hops, if it determines that the net cost of that path is lower. In this context, a cost is an abstract value that represents an interconnect property, such as round-trip latency, bandwidth or other parameters of interest. The exact definition of cost, and the associated decision-making is implementation defined.
- Once the cost of each hop in a CCIX network is established, the firmware may choose to deploy routing cost optimization algorithms to the network, such as the Dijkstra's algorithm.
- The best-case configuration would be one address range per port on each device.

### 11.1.2.1 Generic Routing Rules

The general rule of thumb for routing messages within a CCIX network is based on the types of messages that agents can generate, and the nature of the initiating agent. The following rules are particularly important:

1. Memory requests are generated by RAs or HAs and are routed on the basis of the SAM table.
2. Memory responses are always ID routed. HAs (if the requester is an RA) or SAs (if the requester is an HA) refer to the IDM table to route memory responses to the RA or HA that generated the memory request, as the case may be.
3. Snoop requests are only generated by HAs, and they are ID-routed.
4. Snoop responses are only generated by RAs, and they are ID-routed. A snoop response is always returned over the same path that the snoop request took.

#### 11.1.2.1.1 Routing Considerations for ID-based Messages

As already described in Section 2.6.5, ID-based messages (snoop request, snoop response and memory response) are routed by specifying a destination Agent IDs instead of a destination memory address. The CCIX protocol mandates that:

1. Memory requests are issued by RAs targeting HAs. Memory requests are also issued by HAs targeting SAs.
2. Memory responses are generated by HAs that own the memory requested by the RA. Memory responses must be sent to the RA along the same path as the memory request.
3. Memory responses are also generated by SAs in response to memory requests from HAs. The memory responses must be returned along the same path as the request from the HA.
4. Snoop requests are generated by HAs and are sent to destination RAs. Snoop requests must follow the same path as a writeback from the RA to that HA.
5. A snoop response from an RA must be transmitted along the same path that the snoop request from the HA.

These rules are laid out to ensure proper credit management along a message path, and to satisfy certain coherency protocol requirements such as hazarding [1]. For certain topologies, the rules can cause ID routing and address routing tables to be dissimilar.

## 11.1.3 Topologies and Routing Requirements

The boot firmware must satisfy routing and address mapping requirements as described in the previous sections. These requirements vary from system topology to system topology. It is recommended that the firmware be designed to be general-purpose and apply to at least a basic set of typical topologies:

- Tree
- Mesh
- Fully-connected

In this section, these common topologies and techniques for recognizing them in firmware are described in detail.

**IMPLEMENTATION NOTE**

This section provides a few example topology recognition algorithms. However, these are by no means exhaustive, and system designers can choose or develop other techniques as they deem fit.

### 11.1.3.1 The Tree Topology

The tree topology is similar to the PCIe hierarchy. CCIX devices are organized in the same manner as PCIe devices.

Address routing and memory decoding in the tree topology is optimized by performing a depth-first walk of the tree, and then placing the logical memory pool on each node visited on top of its predecessor's memory pool.

Figure 77 provides an example of the application of the above method to a sample tree topology and highlights how and why method works. Nodes in this example tree are numbered in an increasing order by visiting them in the depth-first manner, as illustrated. The logical memory pool in each node is also assigned the same identifier as its parent node. The logical memory pools are then arranged in the system memory map linearly in increasing order of their identifiers. This allows memory addressing to closely match the hierarchical arrangement of the nodes in the topology. For example, for node 1, all memory on downstream nodes are in a single contiguous range {2-10} in the G-SAM. Likewise node 2, which has downstream nodes 3-6, sees the corresponding memory pools 3-6 in the contiguous range {3-6}.

Performing this coalescing of memory pools on downstream or upstream nodes into contiguous ranges reduces the number of entries required on each port to address each memory pool. In this example, port P1 on node 1 can now be programmed to route all memory requests in the contiguous range {2-6}, while port P2 programmed to route all memory requests in the contiguous range {7-10}. This effectively means that node 1 needs only one routing table entry per port to address all of system memory in this particular system topology. Comparing this with the worst-case brute-force approach where one routing table entry per destination memory pool is

required, and it is easy to see that the worst-case number of table entries for port P1 on node 1 would have been five entries (one each for memory pools on node 2 to node 6 respectively).

The minimum number of routing entries with the depth-first approach is one entry per port per device, and the maximum is two entries per port. In the example in Figure 77, it can be seen that node 4 requires two routing entries on port P1.

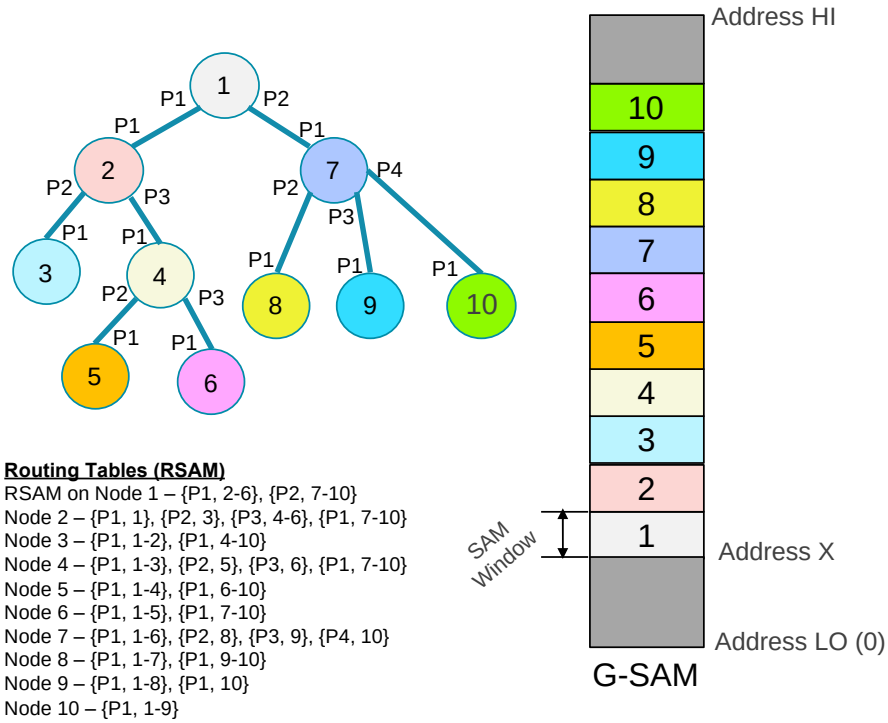



**Figure 77: Routing and Addressing in the Tree Topology**

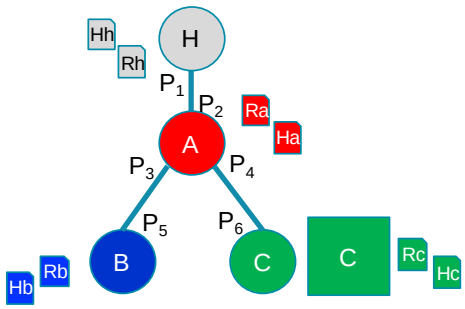
It can be seen that other methods, for example, breadth-first, will necessitate more routing entries per port per node than the depth-first approach.

#### 11.1.3.1.1 Recognition of Tree Topologies

A Tree topology is recognized by the presence of a hierarchical arrangement of devices at different levels, where the following must hold true:

1. Every device at a given level N has one and only one parent device at level N-1.
2. There exists no direct link between two devices at the same level.

#### 11.1.3.1.2 Routing Configuration for ID-based Messages




A (IDM)

| ID | Port |
|---|---|
| Ra, Ha | Local |
| Rh, Hh | $P_2$ |
| Rb, Hb | $P_3$ |
| Rc, Hc | $P_4$ |

B (IDM)

| ID | Port |
|---|---|
| Ra, Ha | $P_5$ |
| Rh, Hh | $P_5$ |
| Rb, Hb | Local |
| Rc, Hc | $P_5$ |

C (IDM)

| ID | Port |
|---|---|
| Ra, Ha | $P_6$ |
| Rh, Hh | $P_6$ |
| Rb, Hb | $P_6$ |
| Rc Hc | Local |

**Figure 78: ID Routing in Tree Topologies**

The optimal paths for ID-based messages are identical to the memory address paths.

### 11.1.3.2 The Mesh Topology

The mesh is a non-hierarchical topology that has at least one loop and at least one pair of nodes that are multiple hops apart.

Mesh topologies are optimized using the well-known dimension-ordered routing algorithms. Typical examples are X-then-Y routing (XY Turn Model) and Y-then-X routing (YX Turn Model).

More details on these algorithms are available in [7].

Memory pools on each multi-hop path along a dimension-routed path are combined to form a larger logical memory pool.

SAM windows in G-SAM are then assigned to the final set of logical memory pools. Best-case number of RSAM table entries per device is N, where N is the number of ports on the device, and the best case number of RSAM table entries per port per device is 1.

The resultant configuration steps are illustrated in Figure 79.

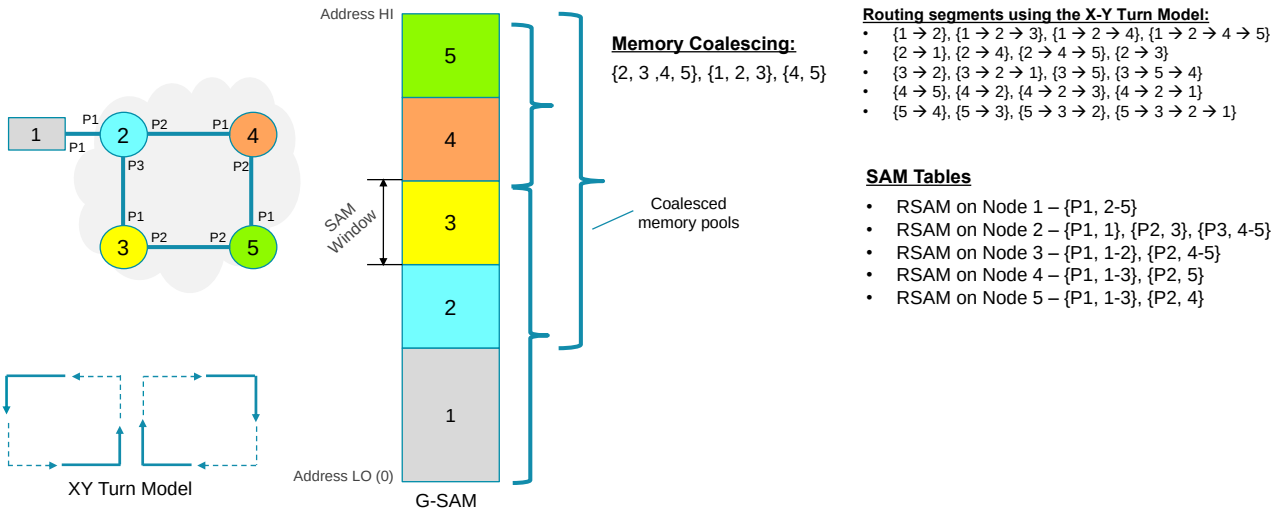



**Figure 79: Routing and Addressing in the Mesh Topology**

### 11.1.3.2.1 Special Considerations

When dimension-ordered routing is applies to meshes, some paths become illegal. This in turn can affect the routing rules of ID-based messages. For example, if XY Turn model is used, then a snoop response cannot retrace the snoop request because it violates the turn model, as depicted in Figure 80.

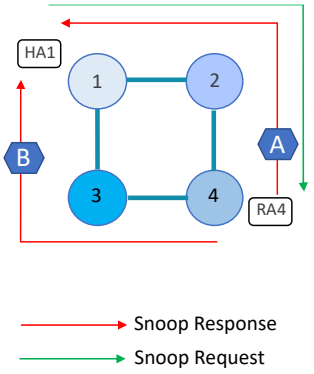



**Figure 80: Dimension-ordered Routing and ID-routed messages**

In this particular example, if the XY Turn Model is chosen, then a snoop request from HA1 reaches RA4 via the 1-2-4 path (Path A). According to the CCIX protocol rules, the snoop response from RA4 is required to retrace the original path of the memory request, i.e. the reverse path 4-2-1. However, this violates the XY Turn Model of routing. To adhere to the dimension routing rule, however, the snoop response should have followed the 4-3-1 path, or path B. This, however, violates the snoop response routing rule.

To resolve this routing anomaly, snoop responses (HA to RA messages) within mesh topologies must follow the converse of the routing rule employed for snoop requests. In other words, if snoop requests are XY routed, then snoop responses must be YX routed.

#### 11.1.3.2.2 Mesh Topology Recognition

Before the dimension-ordered algorithms can be applied to a given system, the firmware must first detect the existence of a mesh topology. An approach to recognizing a mesh topology is described below and illustrated in Figure 81. The firmware may deploy this step in Stage 5 to recognize the presence of a mesh in the CCIX subsystem, and then follow the methodology outlined in this section to determine optimized routing.

#### 11.1.3.2.3 Mesh Recognition Algorithm

A suggested algorithm is described below.

- Find a minimum set of 4 nodes that are chained together in a ring format.
- This set is called a minimum mesh.
- A minimum mesh is now treated as a second-level node, which may be chained to other second-level nodes.
- When four such second-level nodes form a ring structure themselves, a third-level node is said to be present.
- The above is repeated until all (first-level) nodes have been visited.

The resultant graph is an *n*th order mesh, depending on the number of levels discovered. The firmware can now apply the routing rules specific to mesh topologies to the result.

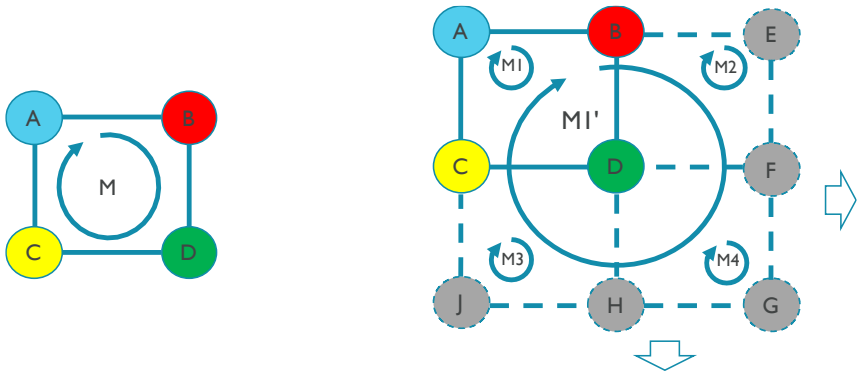



**Figure 81: Mesh Recognition Algorithm**

**IMPLEMENTATION NOTE:**

This algorithm is meant as to illustrate the basic approach and is by no means the only recommended approach to topology recognition. Firmware implementors are free to devise other, potentially better solutions as they deem fit.

### 11.1.3.3 The Fully-connected Topology

The fully-connected topology is non-hierarchical in nature and involves an edge between every pair of nodes in the graph. In the simplest case, the boot firmware can choose the direct path (edge) between each pair in order to achieve optimize routing.

There is no restriction on the logical memory pool placement in the G-SAM, and the SAM windows can be created anywhere in G-SAM, even with intermediate gaps between successive SAM windows, as is depicted in Figure 82.

> **IMPLEMENTATION NOTE:**
>
> When referring to specific topologies such as the mesh and the full-connected, the topology refers to the arrangement of the CCIX nodes and excludes the host. In a typical system, the CCIX subsystem begins as a spur from the host, as depicted in Figure 79 and Figure 82.

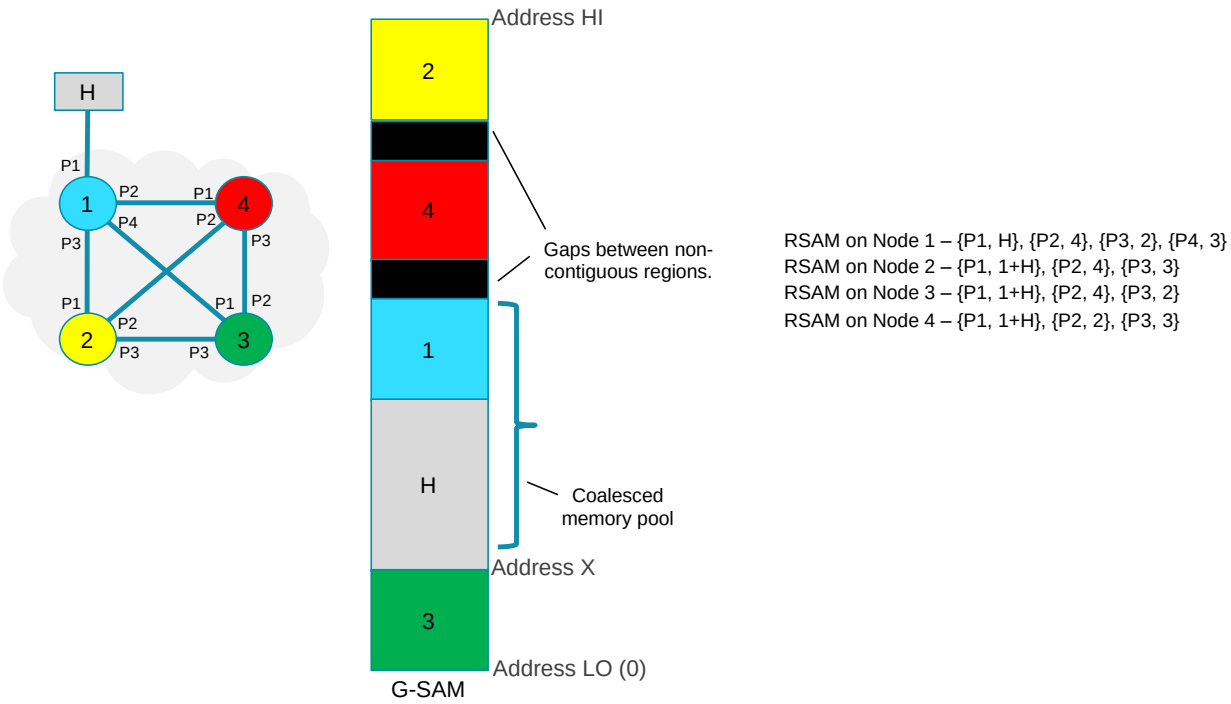



**Figure 82: Routing and Addressing in the Fully-connected Topology**

#### 11.1.3.3.1 Recognition of the Fully-connected Topology

The fully-connected topology is recognizable by the presence of direct links between every pair of devices.

# Acknowledgements

The CCIX® Consortium acknowledges the invaluable contributions of the following persons to this guide:

Harb Abdulhamid
Ard Biesheuval
Eric Biscondi
Jonathan Cameron
Gord Caruk
Jaideep Dastidar
Jeff Defilippi
Leo Duran
Matt Evans
Charles García-Tobin
Veronique Guerre
Kenneth Ma
Phanindra Mannava
Jon Masters
Bruce Mathewson
Millind Mittal
Martin Pickett
Lorenzo Pieralisi
Nariman Poushin
Thanunathan Rangarajan
Vikram Sethi
Ariel Shahar
Kangkang Shen
Vilas Sridharan
Dong Wei