Overview of Core Components

A core component is the slow-path counterpart of the microblock running on the Intel XScale® core. A core component performs the following functions:

1. Configures its microblock (static configuration by means of imported variables and dynamic configuration through control blocks).
2. Initializes and maintains common data structures that may be updated by other applications.
3. Provides exception as well as control message handler to process packets/messages sent by the microblock. Each core component can have multiple inputs, each of which is associated with a different packet/message handler.

In general, there is a single core component associated with each microblock. The IPv4 Forwarder Core Component services exception packets sent to it by the IPv4 microblock. However, a core component may also manage more than one microblock. In the extreme case there may be a single core component for all the microblocks.

There are two ways to implement a core component. One way is to implement the core component using the IXA Core Component Infrastructure Library. The core component infrastructure provides support for handling messages and packets.

The other way is to implement the core component as a software entity that directly uses the Resource Manager API. The design of this entity (whether it is a shared library, driver, thread, process etc.) and how it processes packets and messages is left entirely to the developer.

1. API Functions

The following listing gives the detailed API specifications provided by the Intel Internet Exchange Architecture Portability Framework. All these should be enough for programming the core components.

1. ix_cci_cc_add_event_handler()

Adds an event handler to a core component. This function should be called within application core component code.

Note: The execution engine only checks for events between the handling of messages and packets. Therefore, if a handler takes a long time to process a packet or message, the event triggers considerably later or less often than requested in the arg_ms parameter. The event function is called in the thread context of the execution engine. Event functions take strict priority over message and packet handlers.

C Syntax

```
ix_error ix_cci_cc_add_event_handler(
                                ix_cc_handle arg_hComponent,
                                ix_uint32 arg_ms,
                                ix_event_func* arg_EventFunc,
                                ix_event_type arg_EventType,
```

ix_uint32 arg_Priority,
ix_event_handle* arg_phEvent);

Input

1) ***arg_hComponent***: A handle specifying the core component or execution engine to which to add an event handler.
2) ***arg_ms:*** The minimum number of milliseconds from the time the operation is invoked to the time at which the event will be triggered. Also the interevent interval when arg_EventType is IX_EVENT_TYPE_PERIODIC.
3) ***arg_EventFunc***: An event handler with the function signature specifying "ix_event_func()."
4) ***arg_EventType:*** Specifies the type of event. This value is one of:
    a) IX_EVENT_TYPE_PERIODIC—for periodic, recurring events
    b) IX_EVENT_TYPE_ONESHOT—for one-shot events
5) ***arg_Priority*** The priority value the event scheduler uses to sequence events if two or more events are scheduled to occur at the same time.
6) ***arg_phEvent:*** A pointer to an event handle for the newly added event handler—used for removing event handlers.

Output/Returns

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:

- IX_CCI_ERR_OS_MEM_ALLOC—not enough memory for internal event structure
- IX_CCI_ERR_CC_ADD_HANDLER_INVALID_HANDLE—invalid handle passed in first argument
- IX_CCI_ERR_EVENT_HANDLER_GET_TIME—error getting system time
- IX_CCI_ASSERTION—invalid function parameter—debug compilations only

2. ix_event_func()

The function prototype specifying the signature for an event-handler function provided by the calling application.

C Syntax

ix_error (* ix_event_func)(void * arg_pContext);

Input

arg_pContext: The core-component context created by the calling-application implementation of the ix_cc_init() function passed to the core component creation function, ix_cci_cc_create().

3. ix_cci_cc_add_event_handler_ex()

Similar to ix_cci_cc_add_event_handler()but instead of specifying the core component or engine context this function allows the calling application to specify the event context.

Note: The execution engine only checks for events between the handling of messages and packets. Therefore, if a handler takes a long time to process a packet or message, the event will be triggered considerably later (or less often) than requested in the arg_ms parameter. The event function is called in the thread context of the execution engine. Event functions take strict priority over message and packet handlers.

C Syntax

```
ix_error ix_cci_cc_add_event_handler(
                                    ix_cc_handle arg_hComponent,
                                        ix_uint32 arg_ms,
                                    ix_event_func* arg_EventFunc,
                                     ix_event_type arg_EventType,
                                        ix_uint32 arg_Priority,
                                       void* arg_EventContext,
                                  ix_event_handle* arg_phEvent );
```

Input

arg_hComponent: A handle to the core component.

arg_ms: The minimum number of milliseconds from the time the operation is invoked to the time at which the event will be triggered. Also the interevent interval when arg_EventType is IX_EVENT_TYPE_PERIODIC.

arg_EventFunc: An event handler with function signature specified for "ix_event_func()."

arg_EventType: Specifies the type of event. This value is one of:

a. IX_EVENT_TYPE_PERIODIC—for periodic, recurring events
b. IX_EVENT_TYPE_ONESHOT—for one-shot events

arg_Priority: The priority value the event scheduler uses to sequence events if two or more events are scheduled to occur at the same time.

arg_EventContext: Context to be passed to the event handler specified by arg_EventFunc when the event occurs.

arg_phEvent: A pointer to the event handle used for removing event handlers.

Output/Returns

Return Value:

Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:

• IX_CCI_ERR_OS_MEM_ALLOC—there was not enough memory for the internal event structure
• IX_CCI_ERR_CC_ADD_HANDLER_INVALID_HANDLE—an invalid handle was passed in as the first argument
• IX_CCI_ERR_EVENT_HANDLER_GET_TIME—there was an error getting the system time
• IX_CCI_ASSERTION—an invalid function parameter was specified—applies to debug compilations only

4. ix_cci_change_event()

Changes the period of a periodic event.

C Syntax

```
ix_error ix_cci_change_event(
        ix_event_handle arg_hEvent, ix_uint32 arg_ms );
```

Input

arg_hEvent: A handle to the periodic event whose period is to be changed.
arg_ms: The new event period for the event specified by arg_hEvent.

Output/Returns

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating the following error code:
• IX_CCI_ERROR_ASSERTION—an invalid handle was passed in as the first argument or the event is not periodic—applies to debug compilations only

5. ix_cci_cc_add_message_handler()

Adds a message handler to a core component and associates it with an input ID. This function should be called in the ix_exe_init() function of the core component's execution engine. Attempting to associate a handler to an ID that already is associated with a handler returns an error.
Note: This function should be called in the ix_cci_init()function for the core component or the ix_exe_init()function for the core component's execution engine.

C Syntax

```
ix_error ix_cci_cc_add_message_handler(
        ix_cc_handle arg_hComponent,
        ix_uint32 arg_InputID,
        ix_msg_handler* arg_Handler,
        ix_input_type arg_SourceType);
```

Input

arg_hComponent: A handle to the execution engine or a core component—used to gain access to the execution engine's default policy. If the execution engine is using its default policy tree, the message handler is automatically added to this policy tree. Otherwise, if a custom policy tree has been connected to the execution engine—using ix_cci_exe_add_policy()—it is the responsibility of the calling application to add the handler to the appropriate policy using the function ix_cci_policy_add_leaf().
arg_InputID: Input ID to be associated with the handler.
arg_Handler: A pointer to a message handler provided by the calling application. The callback function prototype is specified as in "ix_msg_handler()."
arg_SourceType: Indicates whether the input receives messages from one source or multiple sources. Valid values are:

a. IX_INPUT_TYPE_SINGLE_SRC—messages are received from a single source
b. IX_INPUT_TYPE_MULTI_SRC—messages are received from multiple sources

NOTE: The framework uses this type value to determine whether or not to employ locking when writing data into the queue corresponding to the input ID associated with this handler.

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ERR_OS_MEM_ALLOC—could not allocate enough memory for internal handler structure
• IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL—attempted to add the handler to a full policy that cannot grow
• IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC—a memory allocation error occurred when attempting to add the handler to a full policy
• IX_CCI_ERR_MSG_SET_MODE—an error was reported by the Resource Manager when setting the handler's mode of operation
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only

6. ix_msg_handler()

The function prototype for message-handler callback functions provided by the calling application.

C Syntax

```
ix_error (* ix_msg_handler) (
                ix_buffer_handle arg_hDataToken,
                ix_uint32 arg_UserData,
                void* arg_pComponentContext);
```

Input
arg_hDataToken: Calling application specific data.
arg_UserData: Calling application specific data.
arg_pComponentContext: The core component context created by the application-defined core component ix_cc_init()function passed to the core component's creation function, ix_cci_cc_create().

Output/Returns
Return Value:
Returns IX_SUCCESS or a valid ix_error.

7. ix_cci_cc_add_packet_handler()

Adds a packet handler to a core component and associates it with an input ID. This function should be called in the ix_exe_init() function of the core component's execution engine. Attempting to associate a handler to an ID that is already associated with a handler returns an error.
Note: This function should be called in the ix_cci_init()for the core component or the ix_exe_init()function of the core component's execution engine.

C Syntax

```
ix_error ix_cci_cc_add_packet_handler(
                                    ix_cc_handle arg_hComponent,
                                    ix_uint32 arg_InputID,
                                    ix_pkt_handler arg_Handler,
                                    ix_input_type arg_SourceType );
```

Input

arg_hComponent: A handle to the execution engine or a core component—used to gain access to the execution engine's default policy. If the execution engine is using its default policy tree, the packet handler is automatically added to this policy tree. Otherwise, if a custom policy tree has been connected to the execution engine using ix_cci_exe_add_policy() it is the responsibility of the calling application to add the handler to the appropriate policy using the function ix_cci_policy_add_leaf().
arg_Handler: Pointer to the packet handler with the signature specified for "ix_pkt_handler()."
arg_InputID: The input ID to be associated with the handler.
arg_SourceType: Indicates whether the input receives messages from one source or multiple sources. Valid values are:

c. IX_INPUT_TYPE_SINGLE_SRC and
d. IX_INPUT_TYPE_MULTI_SRC.

NOTE: The framework uses this type value to determine whether or not to employ locking when writing data into the queue corresponding to the input ID associated with this handler.

Output

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ERR_OS_MEM_ALLOC—could not allocate enough memory for the internal handler structure
• IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL—attempted to add a handler to a full policy that cannot grow
• IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC—there was a memory allocation error when attempting to add handler to a full policy
• IX_CCI_ERR_MSG_SET_MODE—there was an error reported by the Resource Manager when setting the handler's mode of operation
• IX_CCI_ASSERTION—an invalid function parameter was passed in—applies to debug compilations only

8. ix_pkt_handler()

The function prototype for packet handler callback functions provided by the calling application.
C Syntax

```
ix_error (* ix_pkt_handler) (
                                    ix_buffer_handle arg_hDataToken,
                                        ix_uint32 arg_ExceptionCode,
                                    void* arg_pComponentContext);
```

Input
arg_hDataToken: A handler to calling-application defined data.
arg_ExceptionCode: A calling-application defined exception code.
arg_pComponentContext: The core component context created by the application-defined core component ix_cc_init()function passed to the core component's creation function, ix_cci_cc_create()

Output/Returns
Return Value:
Returns IX_SUCCESS or a valid ix_error.

9. ix_cci_cc_create()

Creates a core component. This function should be called from the application-defined ix_exe_init() function in the core component's execution engine.

C Syntax

```
ix_error ix_cci_cc_create(
                                    ix_exe_handle arg_EngHandle,
                                        ix_cc_init arg_InitFunc,
                                        ix_cc_fini arg_FiniFunc,
                                        void* arg_pContextIn,
                                    ix_cc_handle* arg_phComponent);
```

Input
arg_EngHandle: A handle to the execution engine where the core component code is to be run.
arg_InitFunc: A pointer to the core component initialization function. Application code may use this function to create resources shared between the core and any microblocks with which the component interacts. It may also patch symbols to those microblocks. Implementation of this function is an application responsibility. The function signature is described as in, "ix_cc_init()." The ppContext parameter points to an input/output context. The context pointer is stored in the component and passed to the function specified by arg_FiniFunc when the component is destroyed. The context is also passed to any events, message handlers, or packet handlers added to the core component.
arg_FiniFunc: A pointer to the component termination function. The function signature as for "ix_cc_fini()."
arg_pContextIn: A pointer to a context that is passed into arg_InitFunc.

Output
Return Value:

Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ERR_OS_MEM_ALLOC—there was not enough memory to complete the operation
• IX_CCI_ASSERTION—an invalid function parameter was passed in—applies to debug compilations only
• Any errors returned by the ix_cc_init()callback function arg_phComponent A pointer to the location for return of the core-component handle.

10. ix_cc_init()

This is the function prototype for a component initialization function.

C Syntax

```
ix_error (*ix_cc_init)( void** ppContext );
```

Input/Output
ppContext: A pointer to any context convenient for use by the core component. As an input, it may be some global context; as an output it may be a pointer to resources allocated within the initialization function.

Output/Returns
Return Value:
Returns IX_SUCCESS if successful or an ix_error token.

11. ix_cc_fini()

This is the function prototype for a component termination function.

C Syntax

```
ix_error (*ix_cc_fini)( void* pContext );
```

Input
pContext: The context output by core-component initialization function—see “ix_cc_init().”

Output/Returns
Return Value Returns IX_SUCCESS if successful or an ix_error token.

12. ix_cci_cc_destroy()

Destroys the component. The function performs internal cleanup and invokes the application defined ix_cc_fini() function associated with the component. This function should be called from the application-defined ix_exe_fini()function for the core component's execution engine.

C Syntax

ix_error ix_cci_cc_destroy(ix_cc_handle arg_hComponent);

Input

arg_hComponent: A handle to the core component originally initialized using ix_cci_cc_create().

Output

Return Value Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:

• Any errors returned by the ix_cc_fini()callback function
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only

13. ix_cci_cc_remove_event_handler()

Removes an event handler from a core component. This function should be called within application core-component code and is only required for removing periodic events and unexpired one-shot events.

Note: It is not necessary to remove a one-shot event handler after it has expired. Although this function checks that the input parameters are valid, attempting to remove an expired event does not return an error.

C Syntax

ix_error ix_cci_cc_remove_event_handler(ix_event_handle arg_hEvent);

Input

arg_hEvent: A handle to the event to be removed.

Output

Return Value:

Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:

• IX_CCI_ASSERTION—an invalid function parameter was specified—applies to debug compilations only
• IX_CCI_ERR_EVENT_NOT_FOUND—no trace of event found, even on expired event list—debug compilations only

14. ix_cci_cc_remove_message_handler()

Removes a message handler previously added using ix_cci_cc_add_message_handler(). This function may be called from the application-defined termination function of the core component's execution engine, or from the core component's own termination function.

C Syntax

ix_error ix_cci_cc_remove_message_handler(ix_uint32 arg_InputID);

Input

arg_InputID: The input ID associated with the message handler to be removed. After calling this function, any messages arriving on this input ID are dropped.

Output

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:

• IX_CCI_ERR_DISABLE_MSG_HANDLER—error disabling message handler in Resource Manager
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only

15. ix_cci_cc_remove_packet_handler()

Removes a packet handler previously added using ix_cci_cc_add_packet_handler(). This function may be called from the application-defined termination function of the core component's execution engine, or from the core component's own termination function.

C Syntax

ix_error ix_cci_cc_remove_packet_handler(ix_uint32 arg_InputID);

Input

arg_InputID: The input ID where packet handler is to be removed. After calling this function, any packets arriving on this input ID are dropped.

Output/Returns

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:

• IX_CCI_ERR_DISABLE_MSG_HANDLER—error disabling message handler in Resource Manager
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only

16. ix_cci_exe_add_policy()

Adds a branch node—a scheduling policy—to an execution engine. This function should be called from the application-defined ix_exe_init()function for the execution engine in order to replace the execution engine's default policy tree with a different scheduling policy or policy tree.

Note: This function, in conjunction with ix_cci_policy_add_branch() and ix_cci_policy_add_leaf(), should only be called to replace the default policy tree with a different policy or policy tree. All message and packet handlers must be added to the execution engine's core components before calling this function. Failure to do so results in an error being returned from the add-handler function. Care should be taken that all IDs are added to the new policy tree using ix_cci_policy_add_leaf() - otherwise the handlers do not run when packets or messages arrive on those IDs.

C Syntax

```
ix_error ix_cci_exe_add_policy(
                                    ix_exe_handle arg_hParent,
                                  ix_policy_handle arg_hChild,
                                    ix_uint32 arg_NumCredits);
```

Input

arg_hParent: The execution engine to which to add the policy.
arg_hChild: A handle to the policy to add.
arg_NumCredits: The number of times a handler is called before another handler is selected by the policy tree—this assumes the handler's queue has enough entries and that no new messages or packets are received with a higher priority.

Output

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating the following error code:
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only

17. ix_cci_exe_get_info()

Returns the execution-engine handle, engine number, and a context pointer associated with the execution engine in which the caller is running. If this function is called from a non-engine thread, it returns information about the default engine. The default engine is set by the function ix_cci_exe_set_default()—if this function has not been called, the default engine is engine zero.

C Syntax

```
ix_error ix_cci_get_info(
                                    ix_exe_handle* arg_pExeHandle,
                                  ix_uint32* arg_pEngineNumber,
                                    void* arg_ppContext);
```

Input

arg_pExeHandle: A handle to the engine running this thread.

arg_pEngineNumber: A unique engine number. The first engine created is engine number zero, the next is engine number one, and so on.
arg_ppContext: A pointer to the engine context initialized by the application-defined ix_cc_init() function passed into ix_cci_exe_run() when the engine is created.

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating the following error code:
• IX_CCI_ERR_ENG_THREAD_ID—error getting current thread ID or default engine no longer exists

18. ix_cci_exe_run()

Creates and runs an execution engine—that is, a thread of control. This function should be called from application code after ix_cci_init()has been called to initialize the core framework.

Note: This function calls the application-defined ix_exe_init()function associated with this engine and stores the output context internally so that it can be passed to the application-defined ix_exe_fini()function for this engine when ix_cci_exe_shutdown()is called.

C Syntax

```
ix_error ix_cci_exe_run(
                                const char* arg_FileName,
                            ix_exe_init_func arg_InitFunc,
                            ix_exe_fini_func arg_FiniFunc,
                                const char* arg_Name,
                            ix_exe_handle* arg_pHandle);
```

Input
arg_FileName: If the operating environment is process-based and is required to spawn a new process for each execution engine, this parameter specifies the filename where the execution engine's process entry point is defined. Otherwise, the parameter should be set to NULL.
arg_InitFunc: Specifies a application-defined initialization function that is called by ix_cci_exe_run(). This function is responsible for creating the execution-control tree and setting up the message and packet handlers. If the operating environment is process-based and is required to spawn a new process for each execution engine, this parameter should be set to NULL, and the initialization should be performed by the process' entry function. The function signature is specified as for, "ix_exe_init()." The memory for the context passed as an argument into ix_exe_init()must be dynamically allocated by the initialization function. The initialization function must return this context pointer using the same argument so that this pointer can be stored by the execution engine. This context pointer is passed to the execution engine's ix_exe_fini()function at shutdown. The ix_exe_fini() function uses this context to gain access to any core components and policies that the execution engine must delete.
arg_FiniFunc: Specifies a application-defined cleanup function that is called in ix_cci_exe_run() as a result of ix_cci_exe_shutdown() being called in a control thread. After the cleanup function has

returned the ix_cci_exe_run() function terminates. If the operating environment is process-based and is required to spawn a new process for each execution engine, this parameter should be set to NULL, and the engine's ix_exe_fini() pointer should be initialized by the process' entry function. The function signature is specified as for "ix_exe_fini()."
arg_Name: Name to be associated with the current execution engine.
arg_pHandle: A pointer to the new execution engine's handle.

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ERR_ENG_INIT_SEMA_PROP_READ—could not read registry properties for the global synchronization semaphore
• IX_CCI_ERR_ENG_PROP_CLOSE—there was an error closing the property for global semaphore
• IX_CCI_ERR_ENG_INIT_ENGINE_PROP_READ—could not read properties for shared-memory integers used by all execution engines
• IX_CCI_ERR_ENG_PROP_CLOSE—there was an error closing the property for shared-memory integers
• IX_CCI_ERR_ENG_THREAD_SPAWN—there was an error spawning the engine thread
• IX_CCI_ERR_OS_MEM_ALLOC—there was a memory allocation error
• IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC—there was a memory allocation error
• IX_CCI_ERR_ENG_THREAD_ID—could not read the current thread ID
• IX_CCI_ERR_SHARED_MEM_ALLOC—there was a shared-memory allocation error
• IX_CCI_ERR_ENG_INIT_SEMA_INIT—could not initialize this engine's semaphore
• IX_CCI_NO_SELECTED_CHILD—no event/message/packet handler was added—either directly or via one of its core components—to the engine
• IX_CCI_ASSERTION—invalid function parameter was passed in—applies to debug compilations only
• Errors returned from ix_cci_policy_create(), ix_cci_policy_add_branch(), and ix_cci_exe_add_policy() while trying to create the default policy tree
• Errors returned from the callback function whose prototype is ix_cc_init()

19. ix_exe_init()

The function prototype for the application-defined initialization function used when an execution engine is started.

C Syntax

```
ix_error (*ix_exe_init)(
                                    ix_exe_handle ExeHandle,
                                    void** ppContext);
```

Input

ExeHandle: A handle assigned to this execution engine. If this handle is saved in the execution engine's context, then when the engine is running, information about the engine can be retrieved using the function ix_cci_exe_get_info().
ppContext: The memory for the context pointed to by this argument must be dynamically allocated by the initialization function. The initialization function must return this context pointer using ppContext so that this pointer can be stored by the execution engine.

Output/Returns
Return Value:
Returns IX_SUCCESS if successful or an ix_error token.

20. ix_exe_fini()

The function prototype for the application-defined finalization function used when an execution engine is shut down.

C Syntax

```
ix_error (*ix_exe_fini)(
                ix_exe_handle ExeHandle,
                void* pContext );
```

Input
ExeHandle: Handle assigned to this execution engine. If this handle is saved in the execution engine's context, then when the engine is running, information about the engine can be retrieved using the function ix_cci_exe_get_info().
pContext: A pointer to the context created by and passed back from the matching ix_exe_init()function. This context provides access to any core components and policies that the execution engine must delete.

Output/Returns
Return Value Returns IX_SUCCESS if successful or an ix_error token.

21. ix_cci_exe_set_default()

Sets an engine as the default engine whose information is returned by ix_cci_exe_get_info() when that function is called from a non-engine thread. If this function is never called, the default engine is the first engine created after ix_cci_init().
C Syntax

```
ix_error ix_cci_exe_set_default(ix_exe_handle arg_Handle);
```

Input
arg_Handle: A handle to the execution engine to set as the default.

Output/Returns

Return Value Returns IX_SUCCESS if successful or an ix_error token encapsulating the following error code:
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only

22. ix_cci_exe_shutdown()

Sets a flag in the execution engine, telling it to shut down. Each execution engine contains a semaphore that the engine unlocks when it has finished processing and has executed the execution engine application-defined function specified by the ix_cc_fini()function prototype. This shutdown function blocks until the execution engine indicates shutdown is complete. All applications should call this function to shut down the application.

C Syntax

ix_error ix_cci_exe_shutdown(ix_exe_handle arg_Handle);

Input
arg_Handle: Handle to execution engine originally initialized by ix_cci_exe_run().

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only
• Fatal errors returned by event, message, or packet handlers
• Errors returned by callback functions with an ix_cc_fini()function prototype

23. ix_cci_init()

Initializes the core component infrastructure. This function should be called from application code before calling any other functions in the core-component infrastructure.

Note: The function ix_rm_init()must be called before calling this function.

C Syntax

ix_error ix_cci_init(void);

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ERR_INIT_ERR_LOG—unable to initialize error-message log
• IX_CCI_ERR_SHARED_MEM_ALLOC—unable to allocate shared memory needed by the Core Component Infrastructure
• IX_CCI_ERR_GLOB_SEMA_INIT—unable to initialize the global Core Component Infrastructure semaphore

• IX_CCI_ERR_PROPERTY_CREATE—error creating Resource Manager properties
• IX_CCI_ERR_PROPERTY_SET—error setting a property

24. ix_cci_fini()

Terminates the core component infrastructure. This function should be called in application code after all execution engines have been shut down and destroyed.

Note: The function ix_cci_fini() should be called before calling ix_rm_term().

C Syntax

```
ix_error ix_cci_fini( void );
```

Output
Return Value Returns IX_SUCCESS if successful or an ix_error token returned from the Resource Manager or Operating System Services Layer when trying to free a resource.

25. ix_cci_policy_add_branch()

Adds a branch node—a scheduling policy—to another scheduling policy. Up to 32 nodes, that is, branches and leaves, can be added to an individual policy. This function should be called from the application-defined ix_exe_init()function of an execution engine on an existing policy created using ix_cci_policy_create().

C Syntax

```
ix_error ix_cci_policy_add_branch(
                                  ix_policy_handle arg_hPolicy,
                                   ix_policy_handle arg_hChild,
                                ix_uint32 arg_WeightOrPriority);
```

Input
arg_hPolicy: A handle to the scheduling policy to which the branch is added.
arg_hChild: A handle to the branch—or policy—to add.
arg_WeightOrPriority: Weight if the parent is a weighted round robin policy or priority if the parent is a strict priority policy. This parameter is ignored if the parent is a round robin policy.

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL—the parent policy is full and was initialized to be fixed in size
• IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC—there was a memory allocation error

• IX_CCI_ASSERTION—an invalid function parameter was passed in—applies to debug compilations only

26. ix_cci_policy_add_leaf()

Adds a leaf node—packet or message input node—to a scheduling policy. Up to 32 nodes—branches and leaves—can be added to an individual policy. This function should be called from the application-defined ix_exe_init()function of an execution engine on an existing policy, created using ix_cci_policy_create().

C Syntax

```
ix_error ix_cci_policy_add_leaf(
                                    ix_policy_handle arg_hPolicy,
                                    ix_uint32 arg_InputId,
                                    ix_uint32 arg_IsPacketId,
                                    ix_uint32 arg_WeightOrPriority);
```

Input

arg_hPolicy: A handle to the scheduling policy to which to add a leaf.
arg_InputId: The packet or message input ID.
arg_IsPacketId: Indicates if the leaf is a packet input node. Used as a boolean:
• 1—for packet-input ID
• 0—for not a packet-input ID—that is, the node is a message-input ID
arg_WeightOrPriority: A weight if parent is a weighted round robin policy or a priority if parent is a strict priority policy. This parameter is ignored if the parent is a round robin policy.

Output

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_TOKEN_PROCESSOR_CONTAINER_FULL—the parent policy is full and was initialized to be fixed in size
• IX_CCI_ERR_TKP_CONTAINER_MEM_ALLOC—there was a memory allocation error
• IX_CCI_ASSERTION—an invalid function parameter was passed in—applies to debug compilations only

27. ix_cci_policy_create()

Creates a scheduling policy. This function should be called in the application-defined ix_exe_init()function of an execution engine.

C Syntax

```
ix_error ix_cci_policy_create(
                                    ix_policy_type argPolicyType,
                                    ix_policy_handle* arg_pHandle );
```

Input
argPolicyType: The policy type—valid values defined by the infrastructure include:
• IX_POLICY_TYPE_RR
• IX_POLICY_TYPE_WRR
• IX_POLICY_TYPE_SP
arg_pHandle: A pointer to a handle to the new policy.

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ERR_POLICY_CREATE—there was a memory allocation error
• IX_CCI_ASSERTION—an invalid function parameter was passed in—applies to debug compilations only

28. ix_cci_policy_destroy()

Destroys a scheduling policy. This function should be called from the application-defined ix_exe_fini()function of an execution engine.

Note: This function does not destroy any subtree policies. The application code is responsible for destroying all policies that have been created. The core component infrastructure ensures that notifications are dropped gracefully from a tree that is in the process of being dismantled.

C Syntax

```
ix_error ix_cci_policy_destroy( ix_policy_handle arg_Handle );
```

Input
arg_Handle A handle to the policy to destroy—the policy is invalid after this function returns.

Output
Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating the following error code:
• IX_CCI_ASSERTION—an invalid function parameter was passed in—applies to debug compilations only

29. ix_cci_register_fatal_error_handler()

Allows a control application to register a fatal-error handler. The Core Component Infrastructure calls this handler if an event, message, or packet handler returns a fatal error. This allows the control application to disable any microcode associated with this engine and shut the engine or whole Core

Component Infrastructure system down. This function is optional. If used, it must be invoked after ix_cci_init().

C Syntax

```
ix_error ix_cci_register_fatal_error_handler(
                                    ix_ferror_func arg_Handler,
                                    void* arg_pContext);
```

Input

arg_Handler: A fatal error handler that is invoked whenever a fatal error occurs in the Core Component Infrastructure.
The function signature is: void (* ix_ferror_func)(ix_error arg_FatalException, ix_exe_handle arg_hEngine, void* arg_pContext);

The arg_FatalException argument reports the fatal error detected by the Core Component Infrastructure.The arg_hEngine argument specifies the execution engine where the error occurred. The arg_pContext argument is the context registered by ix_cci_register_fatal_error_handler.
arg_pContext: The context to return when the handler is invoked.

Output/Returns

Return Value:
Returns IX_SUCCESS if successful or an ix_error token encapsulating one of the following error codes:
• IX_CCI_ERR_ENG_INIT_ENGINE_PROP_READ—error reading shared memory property
• IX_CCI_ASSERTION—invalid function parameter—debug compilations only

30. ix_ferror_func()

This is the function prototype for the fatal-error handler callback provided by the calling application.

C Syntax

```
void (* ix_ferror_func)(
                                    ix_error arg_FatalException,
                                    ix_exe_handle arg_hEngine,
                                    void* arg_pContext);
```

Input

arg_FatalException: Reports the fatal error detected by the Core Component Infrastructure.
arg_hEngine: Specifies the execution engine where the error occurred.
arg_pContext: The context registered by ix_cci_register_fatal_error_handler().

Output/Returns

Return Value:
Returns IX_SUCCESS if successful or an ix_error token.

31. ix_cci_send_message()

Sends a message from a core component's packet or message handler to another core component. The application may call this function at any time after the core framework has been initialized, but it is typically called from within a core component's packet or message handler to send a message to another core component or to itself. The infrastructure also calls this function, or an internal equivalent, when a microblock sends a message to a core component. If this function is called to send a message to an ID that does not have a handler attached, then there is a call to a default message handler which simply drops the message and frees the message's shared-memory resources.

Note: This function encapsulates ix_rm_message_send() with an exception code of zero. Messages can be sent to core components and microblocks using either API.

C Syntax

```
ix_error ix_cci_send_message(
                                    ix_uint32 arg_OutputId,
                                ix_buffer_handle arg_Handle,
                                    ix_uint32 arg_UserData);
```

Input
arg_OutputId: The output ID identifying the message destination.
arg_Handle: A handle to the message to be sent.
arg_UserData: Application-defined content that can be used by the receiving message handler for any purpose—for example, identifying the message type.

Output
Return Value Returns IX_SUCCESS if successful or a valid ix_error token retrieved from ix_rm_message_send().

32. ix_cci_send_packet()

Sends a packet from a core component's packet/message handler to another core component or microblock. The calling application may invoke this function at any time after the core framework has been initialized. Typically this operation is called from within a core component's packet handler to forward a packet to another core component or even to itself. The infrastructure also calls this function—or an internal equivalent—when a microblock sends a packet to a core component. If this function is called to send a packet to an ID that does not have a handler attached, then there is a call to a default packet handler which simply drops the packet and frees the packet's shared-memory resources.

Note: This function encapsulates ix_rm_packet_send() with an exception code of zero. Packets can be sent to core components and microblocks using either API.

C Syntax

```
ix_error ix_cci_send_packet(
                                    ix_uint32 arg_OutputId,
                                    ix_buffer_handle arg_Handle );
```

Input
arg_OutputId: The output ID—the packet destination.
arg_Handle: A handle to the packet to be sent.

Output
Return Value:
Returns IX_SUCCESS if successful or a valid ix_error type retrieved from ix_rm_packet_send().

2. Symbolic Constants—Tuning Behavior and Memory

Footprint

The symbolic constants defined in this section allow the system designer to tune the behavior and memory footprint of the Core Component Infrastructure.

IX_CCI_EVENT_CHECK_INTERVAL
The default frequency at which execution engines check for events and shutdown flag. The check interval is in units of milliseconds. If an event is set that is of shorter duration than this interval, the engine semaphore's timeout uses the event timeout instead. Therefore, events can be handled even if this value is set to wait forever. However, if the value is set to wait forever, it might not be possible to shut down the execution engines.
Note: Do *not* set this value to zero.
C Syntax and Default

```
#define IX_CCI_EVENT_CHECK_INTERVAL 1000
```

IX_CCI_MEMORY_CHANNEL
The channel number for Core Component Infrastructure memory.
C Syntax and Default

```
#define IX_CCI_MEMORY_CHANNEL IX_DRAM_CHANNELS_NUMBER - 1
```

IX_CCI_SHARED_MEMORY_TYPE
The type of memory used for Core Component Infrastructure shared memory.
C Syntax and Default

```
#define IX_CCI_SHARED_MEMORY_TYPE IX_MEMORY_TYPE_DRAM
```

IX_CCI_MAX_INPUT_IDS
Defines the maximum number of input IDs.
C Syntax and Default
#define IX_CCI_MAX_INPUT_IDS IX_COMM_LOCAL_ID_NUMBER - \
IX_COMM_LAST_UBLOCK_ID - 1

IX_CCI_MAX_ENGINES
The maximum number of engines on an ingress or egress board. This value does not need to be a power of two. It should not be greater than 64 in the current implementation.
C Syntax and Default
#define IX_CCI_MAX_ENGINES 64

IX_CCI_MAX_POLICIES
The maximum number of policies on an ingress or egress board. This value does not need to be a power of two. It should not be greater than 512 in the current implementation.
C Syntax and Default
#define IX_CCI_MAX_POLICIES 256

IX_CCI_MAX_CCS
The maximum number of core components in an engine. This value does not need to be a power of two. It should not be greater than 256 in the current implementation.
C Syntax and Default
#define IX_CCI_MAX_CCS 64

IX_CCI_MAX_EVENTS
The maximum number of events in an engine. This value does not need to be a power of two. It should not be greater than 2,000,000 in the current implementation.
C Syntax and Default
#define IX_CCI_MAX_EVENTS 256

IX_CCI_NUM_DEFAULT_ENGINES
The number of default engines associated with non-engine threads. The code presently only allows for one default engine.
C Syntax and Default
#define IX_CCI_NUM_DEFAULT_ENGINES 1

IX_CCI_EXE_RUN_GRANULARITY
The number of handlers the executable engine runs before passing control to the next execution engine.

C Syntax and Default

```
#if !defined(IX_CCI_EXE_RUN_GRANULARITY)
#define IX_CCI_EXE_RUN_GRANULARITY 4
#endif
```

Policy Container Capacity Constants

The following constants determines the initial capacity of the policy container. Policy containers can be grown in size, so if most execution engines have few handlers associated with them, this number can be set to a low value to save memory usage.

C Syntax and Default

```
#define IX_CCI_INIT_WRR_POLICY_SIZE 1
#define IX_CCI_INIT_RR_POLICY_SIZE 1
#define IX_CCI_INIT_SP_POLICY_SIZE 2
#define IX_CCI_WRR_POLICY_GROW_INCR 1
#define IX_CCI_RR_POLICY_GROW_INCR 1
#define IX_CCI_SP_POLICY_GROW_INCR 1
```

3. Memory Management

Apart from programming the core components, the other vital part of programming the processor is using memory. The following table summarizes the memory management APIs provided by the portability framework's resource manager. A detailed description of the APIs follows the table.

Resource Manager Memory Management API

Name	Description
ix_memory_type	An enumerated type listing the types of memory supported by the Resource Manager.
ix_memory_info	Memory information data structure.
ix_memory_alignment_type	Alignment types for the aligned memory allocation and reservation calls.
ix_rm_mem_alloc()	Allocates memory—SRAM DRAM and scratch.
ix_rm_mem_alloc_aligned()	Allocates memory with alignment—SRAM DRAM and scratch.
ix_rm_mem_reserve()	Reserves memory.
ix_rm_mem_reserve_aligned()	Reserves memory with alignment.
ix_rm_mem_free()	Frees memory.
ix_rm_mem_info()	Retrieves memory information for the specified memory type and specified channel.
ix_rm_mem_local_alloc()	Allocates local memory.
ix_rm_mem_local_reserve()	Reserves local memory
ix_rm_mem_local_free()	Frees local memory.
ix_rm_mem_local_info()	Retrieves local memory information.
ix_rm_get_phys_offset()	Returns the physical offset of a memory block.
ix_rm_get_virtual_address()	Returns the virtual address of a memory block.
Read/Write Macros	Macros to read and write memory locations.

The Resource Manager manages SRAM, DRAM, scratch and local memory for the IXP2400 and Intel® IXP2800 Network Processors. It exports an interface to allocate, access and free memory chunks. The Resource Manager owns SRAM and scratch memory completely and DRAM partially. The DRAM is partly owned by the operating system. The memory managed by the Resource Manager is used to support system-wide data structures such as the buffer free pools, control blocks for building blocks, route table, trie table, and so on. The difference between this memory and memory allocated from the operating system is that this memory has no MMU protection and is always addressed at the same memory location by all processes. This has meaning only in Linux*. In VxWorks*, the memory model is a flat memory space shared by every task. Since this memory is shared with the microengines it is typically uncached. All applications including the building block infrastructure must use this API to allocate memory in order to work in conjunction with the microengines. The operating system memory is not accessible from microcode.

Defined Types, Enumerations, and Data Structures

This section describes data structures used in Memory Management API.

1. ix_memory_type

This data type represents available memory types in the system under the control, in whole or in part, of the Resource Manager.

C Syntax

```
typedef enum ix_e_memory_type {
IX_MEMORY_TYPE_FIRST = 0,
IX_MEMORY_TYPE_DRAM = IX_MEMORY_TYPE_FIRST,
IX_MEMORY_TYPE_SRAM,
IX_MEMORY_TYPE_SCRATCH,
IX_MEMORY_TYPE_LOCAL,
IX_MEMORY_TYPE_LAST
}ix_memory_type;
```

2. ix_memory_info

This structure is used to retrieve memory information from the Resource Manager including the type of memory and the particular channel of memory that is managed by the Resource Manager.

C Syntax

```
typedef struct ix_s_memory_info {
ix_uint32* m_pStartAddress;
ix_uint32 m_TotalSize;
```

ix_uint32 m_FreeSize;
ix_uint32 m_DefaultAlignmentShift;
ix_uint32 m_ChannelPhysicalOffset;
void* m_pChannelPhysicalAddress;
} ix_memory_info;

Data Members
m_pStartAddress: A pointer to the start of the memory block.
m_TotalSize: The total memory size.
m_FreeSize: The size of the free memory area.
m_DefaultAlignmentShift: The default alignment expressed as a power of two.
m_ChannelPhysicalOffset: The physical offset of the start of the memory block inside the memory channel.
m_pChannelPhysicalAddress: A pointer to the physical address of the start of the current memory channel.

3. ix_memory_alignment_type

This enumerated type describes the possible requests for allocating or reserving memory with alignment constraints. IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL allocates memory providing alignment for the virtual memory. The physical offset and/or physical address might not be aligned with the same alignment requirements. IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL allocates memory providing alignment for the physical memory. The physical offset and/or virtual address might not be aligned with the same alignment requirements.

IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL_OFFSET allocates memory providing alignment for the physical memory offset for the channel. The virtual address and/or physical address might not be aligned with the same alignment requirements. For large alignment the physical address, physical offset and virtual address might not satisfy the same alignment at the same time.

C Syntax
typedef enum ix_e_memory_alignment_type {
IX_MEMORY_ALIGNMENT_TYPE_FIRST = 0,
IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL = IX_MEMORY_ALIGNMENT_TYPE_FIRST,
IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL,
IX_MEMORY_ALIGNMENT_TYPE_PHYSICAL_OFFSET,
IX_MEMORY_ALIGNMENT_TYPE_LAST
} ix_memory_alignment_type;

4. API Functions

1. ix_rm_mem_alloc()

This function allocates common microengine and Intel XScale® core memory on behalf of the calling application. On return, if the allocation fails the pointer arg_pMemoryAddr is set to NULL and if the allocation succeeds arg_pMemoryAddr returns a virtual memory pointer to the allocated memory block address. The returned address is aligned according to the default alignment for the type of memory. If the size does not comply with memory alignment then the actual allocated size is rounded up to insure that alignment is satisfied for further allocations.

The Resource Manager allocates memory so that:
• DRAM alignment is always at an 8-byte boundary and any request is rounded off to an 8-byte boundary
• SRAM and scratch memory is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary

C Syntax

```
ix_error ix_rm_mem_alloc(
                ix_memory_type arg_MemType,
                ix_uint32 arg_MemChannel,
                ix_uint32 arg_Size,
                void** arg_pMemoryAddr);
```

Input
arg_MemType: The type of memory requested.
arg_MemChannel: The channel from which memory is requested.
arg_Size: The size in bytes of the requested memory block.
arg_pMemoryAddr: The pointer to the address of the newly allocated memory. The value of this pointer is NULL if the allocation fails and the address of the allocated memory block if successful.

Output/Returns
Return Value:
Returns IX_SUCCESS if successful and a valid ix_error value otherwise.

2. ix_rm_mem_alloc_aligned()

The calling application uses this function to allocate memory common to microengines and the Intel XScale® core. On return, if the allocation fails the pointer arg_pMemoryAddr is set to NULL and if the allocation succeeds the pointer is set to the allocated memory block address. The Resource Manager provides default alignment for different types of memory, but some applications might require a higher alignment than the default one. The alignment can requested relative to the virtual address, the physical address or the physical offset. For example hardware rings require certain physical offset alignment for the controlled memory. The returned address is aligned according to the arg_AlignmentSfift parameter and arg_AlignmentType alignment type. If arg_AlignmentSfift parameter is less than default alignment then the return memory is aligned according to the default

alignment. If the alignment type is not IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL then the resulting virtual address might not be aligned according to the arg_AlignmentShift—instead, the physical offset or physical address is aligned properly. When changes in alignment are required, the actual allocated size is rounded up to accommodate the required alignment.

C Syntax

```
ix_error ix_rm_mem_alloc_aligned(
                                    ix_memory_type arg_MemType,
                                    ix_uint32 arg_MemChannel,
                                    ix_uint32 arg_Size,
                                    ix_memory_alignment_type arg_AlignmentType,
                                    ix_uint32 arg_AlignmentShift,
                                    void** arg_pMemoryAddr);
```

Input

arg_MemType: Represents the type of memory requested.
arg_MemChannel: Represents the channel from which memory is requested.
arg_Size: Represents the size in bytes of the block requested.
arg_AlignmentType: Specifies the type of alignment required as explained for, "ix_memory_alignment_type."
arg_AlignmentShift: This value specifies the required memory block alignment as a power of two of the alignment. If the specified alignment is less than the default alignment for the corresponding memory then the default alignment is used.
arg_pMemoryAddr: A pointer to the allocated memory address. The pointer arg_pMemoryAddr is set to NULL if the allocation fails, or to the allocated memory block address if the allocation succeeds.

Output/Returns

Return Value:
Returns IX_SUCCESS if successful or a valid ix_error for failure.

3. ix_rm_mem_reserve()

This function reserves a region of memory of the specified type at the specified offset. If memory of that type and at that offset has already been allocated an error is returned. This call is used to reserve SRAM, DRAM, or scratch memory for Microengine C. To reduce the chances of failure, call this function before any memory allocations are made.
The Resource Manager reserves memory such that:
• DRAM alignment is always at an 8-byte boundary and any request is rounded off to a 8-byte boundary
• SRAM and scratch memory is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary

C Syntax

```
ix_error ix_rm_mem_reserve(
                                    ix_memory_type arg_MemType,
```

```
                                        ix_uint32 arg_MemChannel,
                                            ix_uint32 arg_Size,
                                    ix_uint32* arg_pByteOffset);
```

Input
arg_MemType: The type of memory to reserve.
arg_MemChannel: The memory channel.
arg_Size: The size in bytes of the memory block to reserve.

Input/Output
arg_pByteOffset: The byte offset of the memory block requested for reservation. On return this value is adjusted to accommodate the memory type default alignment.

Output/Returns
Return Value:
Returns IX_SUCCESS if successful and a valid ix_error value otherwise. If any portion of the memory requested for reservation has been already allocated this call fails.

4. ix_rm_mem_reserve_aligned()

This function reserves an area of memory at the specified memory channel byte offset and of specified size. The function is provided as support for the Microengine C compiler and linker. This call should be made before any memory allocation has been made in order to reduce the chance of failure. This function reserves memory with specified alignment for SRAM, DRAM and scratch memory. If the byte offset is not be aligned according to alignment type and arg_AlignmentShift. If arg_AlignmentShift is less than the default alignment then the default alignment is used and more data than was requested might be reserved. If the alignment type is not IX_MEMORY_ALIGNMENT_TYPE_VIRTUAL then the resulting virtual address might not be aligned according to the arg_AlignmentShift—instead, the physical offset or physical address is properly aligned. In general the byte offset and size should comply with the memory alignment, otherwise more memory is reserved to insure the right alignment. On return *arg_pByteOffset might be modified to reflect alignment requirements.

C Syntax

```
                                ix_error ix_rm_mem_reserve_aligned(
                                                ix_memory_type arg_MemType,
                                                    ix_uint32 arg_MemChannel,
                                                        ix_uint32 arg_Size,
                                            ix_memory_alignment_type arg_AlignmentType,
                                                    ix_uint32 arg_AlignmentShift,
                                                ix_uint32* arg_pByteOffset);
```

Input
arg_MemType: The type of memory to be reserved.
arg_MemChannel: The channel from which memory is to be reserved.

arg_Size: The size in bytes of the memory block to be reserved.
arg_AlignmentType: The type of alignment required as explained at ix_memory_alignment_type type description
arg_AlignmentShift: The required memory block alignment as the power of two of the alignment. If the specified alignment is less than the default alignment for the corresponding memory then the default alignment is used.

Input/Output
arg_pByteOffset: Represents the address of byte offset of the memory block requested for reservation. On return this value is adjusted to accommodate the memory type default alignment.

Output/Returns
Return Value Returns IX_SUCCESS if successful or a valid ix_error for failure. If any portion of the memory requested for reservation has been already allocated then the call fails.

5. ix_rm_mem_free()

This function frees memory of a particular type previously allocated or reserved through a call to ix_rm_mem_alloc(). If the memory was not previously allocated by a call to ix_rm_mem_alloc() the results of this call are unpredictable. Passing an address that has not been allocated or that has already been freed results in an error. Passing NULL results in a no op. Attempting to accessing a memory location within the block after that memory block has been freed results in unpredictable behavior.

C Syntax

```
ix_error ix_rm_mem_free(
                                                    void* arg_pMemory);
```

Input
arg_pMemory: The address of the memory block to free.

Output/Returns
Return Value:
Returns IX_SUCCESS if successful and a valid ix_error value otherwise. Passing an address that has not been allocated or that has already been freed results in an error. Passing NULL results in a no op.

6. ix_rm_mem_info()

This function retrieves memory information for the specified memory type and specified channel. The structure pointed to by arg_pMemoryInfo is returned with the requested memory information.

C Syntax

```
ix_error ix_rm_mem_info(
```

```
                                        ix_memory_type arg_MemType,
                                            ix_uint32 arg_MemChannel,
                                    ix_memory_info* arg_pMemoryInfo);
```

Input
arg_MemType: The type of memory for which we want to get the info.
arg_MemChannel: The channel for which to get information.
arg_pMemoryInfo: On return points to the structure containing memory information

Output/Returns
Return Value Returns IX_SUCCESS if successful or a valid ix_error for failure.

7. ix_rm_mem_local_alloc()

This function is used to allocate local memory for a specific microengine. The memory allocated cannot be used on the Intel XScale® core. This call returns an offset into the local memory of the microengine. This offset is patched into the microcode by the Intel XScale® core. The Resource Manager allocates local memory such that the offset is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary. The pointer, arg_pByteOffset, is set to (ix_uint32) (-1) if the allocation fails or to the allocated memory block offset if the allocation succeeds. The local memory is not accessible to Intel XScale® core applications. The applications retrieve the offset and patch it into the microcode for the specific microengine. The returned offset is 4-byte aligned. If the size does not comply with memory alignment then the actual allocated size is rounded up to a 4-byte boundary.

C Syntax

```
                    ix_error ix_rm_mem_local_alloc(
                                            ix_uint32 arg_MeNumber,
                                                ix_uint32 arg_Size,
                                        ix_uint32* arg_pByteOffset);
```

Input
arg_MeNumber: The microengine number for which local memory should be allocated. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
arg_Size: The size in bytes of the requested memory block.
arg_pByteOffset: The pointer used to return the allocated memory offset in bytes. arg_pByteOffset is set to (ix_uint32) (-1) if the allocation fails

Output/Returns
Return Value:
Returns IX_SUCCESS if successful and a valid ix_error value otherwise.

8. ix_rm_mem_local_reserve()

This function reserves a region of local memory at the specified offset. If that memory has already been allocated then an error is returned. This call is used to reserve local memory for Microengine C. To reduce the chances of failure, call this function before any memory allocations are made. The Resource Manager reserves memory such that local memory is always returned aligned at a 4-byte boundary and any request is rounded off to a 4-byte boundary.

C Syntax

```
ix_error ix_rm_mem_local_reserve(
                                    ix_uint32 arg_MeNumber,
                                    ix_uint32 arg_Size,
                                    ix_uint32* arg_pByteOffset);
```

Input

arg_MeNumber: The microengine number for which local memory is reserved. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
arg_Size: The size in bytes of the requested memory block to be reserved.

Input/Output

arg_pByteOffset: On input, a pointer to the byte offset of the memory block to be reserved. On return this value is adjusted to reflect the default alignment for the memory block.

Output/Returns

Return Value:
Returns IX_SUCCESS if successful or a valid ix_error for failure. If any portion of the memory requested for reservation has been already allocated then the call fails.

9. ix_rm_mem_local_free()

This function frees memory previously allocated or reserved. Passing an address that has not been allocated or that has already been freed results in an error. Passing NULL results in a no op. Accessing the memory location after the block has been freed results in unpredictable behavior.

C Syntax

```
ix_error ix_rm_mem_local_free(
                                    ix_uint32 arg_MeNumber,
                                    ix_uint32 arg_MemoryOffset);
```

Input

arg_vMeNumber: The microengine number for which local memory should be freed. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
arg_MemoryOffset: The offset of the memory block to be freed.

Output/Returns
Return Value:
Returns IX_SUCCESS if successful or a valid ix_error for failure. Passing an address that has not been allocated or that has already been freed results in an error. Passing NULL results in a no op.

10. ix_rm_mem_local_info()

This function retrieves memory information for the specified memory type and specified channel. On return the structure pointed to by arg_pMemoryInfo contains the requested memory information. This function retrieves local memory information for each microengine.

C Syntax

```
ix_error ix_rm_mem_local_info(
                                    ix_memory_type arg_MeNumber,
                                ix_memory_info* arg_pMemoryInfo);
```

Input
arg_MeNumber: Represents the microengine number for which to get memory information. Allowed values for the microengine number are 0x00 through 0x03 and 0x10 through 0x13 for the Intel® IXP2400 Network Processor and 0x00 through 0x07 and 0x10 through 0x17 for the Intel® IXP2800 Network Processor. The validity of the microengine number is checked only in debug mode.
arg_pMemoryInfo: On return points to the memory information data structure.

Output/Returns
Return Value Returns IX_SUCCESS if successful or a valid ix_error for failure.

11. ix_rm_get_phys_offset()

Returns the physical offset in bytes for memory allocated by the Resource Manager. This routine takes in as input a virtual memory pointer used to access the memory on the Intel XScale® core. It returns the physical offset into the channel in bytes, the physical address, the memory type, and the channel for this memory area. If any of the *Out* parameters are NULL then the corresponding piece of data is not returned.

C Syntax

```
ix_error ix_rm_get_phys_offset(
                                    const void* arg_pMemory,
```

```
                                        ix_memory_type* arg_pMemType,
                                        ix_uint32* arg_pMemChannel,
                                        ix_uint32* arg_pOffset,
                                        void** arg_pPhysicalAddress);
```

Input
arg_pMemory: The pointer to the memory block to query.
arg_pMemType: A pointer to the type of memory at the location in question.
arg_pMemChannel: A pointer to the memory channel for the location in question.
arg_pOffset: A pointer to the offset for the memory location in question.
arg_pPhysicalAddress: The pointer to the physical address.

Output/Returns
Return Value Returns IX_SUCCESS if successful and a valid ix_error value otherwise.

12. ix_rm_get_virtual_address()

Returns the virtual pointer into memory given a physical offset into the memory buffer, specified in bytes. The virtual pointer returned can be used to access this memory on the Intel XScale® core.

C Syntax

```
ix_error ix_rm_get_virtual_address(
                                        ix_memory_type arg_MemType,
                                        ix_uint32 arg_MemChannel,
                                        ix_uint32 arg_Offset,
                                        ix_uint8** arg_pMemoryAddr);
```

Input
arg_MemType: The type for the physical memory location.
arg_MemChannel: The channel for the physical memory location.
arg_Offset: The physical offset in bytes for the physical memory location.
arg_pMemoryAddr: A virtual pointer to the given physical address.

Output/Returns
Return Value Returns IX_SUCCESS if successful and a valid ix_error value otherwise.

5. Read/Write Macros

To provide portability between code running on hardware and code running under a foreign model, the following macros are provided. Use of these macros makes it easier to write code that runs both under the foreign model and real hardware. In general for access to memory that is used by both the Intel XScale® core and the microengines, these macros should be used instead of direct dereference of the addresses. These macros support code running on microengines in big-endian mode and on the Intel XScale® core in little-endian mode without any change to the code. Using these macros facilitates writing code that runs both under the foreign model and real hardware. The following table lists the macros provided.

Resource Manager Memory Management Macros

Macro Name	Description
IX_RM_MEM_UINT8_READ	Returns an ix_uint8 representing the value at the specified location.
IX_RM_MEM_UINT16_READ	Returns an ix_uint16 representing the value at the specified location.
IX_RM_MEM_UINT32_READ	Returns an ix_uint32 representing the value at the specified location.
IX_RM_MEM_UINT64_READ	Returns an ix_uint64 representing the value at the specified location.
IX_RM_MEM_UINT8_WRITE	Writes an ix_uint8 value to the specified location.
IX_RM_MEM_UINT16_WRITE	Writes an ix_uint16 value to the specified location.
IX_RM_MEM_UINT32_WRITE	Writes an ix_uint32 value to the specified location.
IX_RM_MEM_UINT64_WRITE	Writes an ix_uint64 value to the specified location.

1. IX_RM_MEM_UINT8_READ

A macro used to read an 8-bit value from memory.

C Syntax

#define IX_RM_MEM_UINT8_READ(arg_pMemoryAddr)

Input

arg_pMemoryAddr: The pointer to the address of the memory location to be read. The value at this location should be of the type ix_uint8*.

Output/Returns

Return Value An ix_uint8 representing the value at the specified location

2. IX_RM_MEM_UINT16_READ

A macro used to read a 16-bit value from memory.

C Syntax

#define IX_RM_MEM_UINT16_READ(arg_pMemoryAddr)

Input

arg_pMemoryAddr: A pointer to the address of the memory location to be read. The value at this location should be of the type ix_uint16* and aligned on a 16-bit boundary.

Output/Returns

Return Value An ix_uint16 representing the value at the specified location.

3. IX_RM_MEM_UINT32_READ

A macro used to read a 32-bit value from memory.

C Syntax

#define IX_RM_MEM_UINT32_READ(arg_pMemoryAddr)

Input

arg_pMemoryAddr: A pointer to the address of the memory location to be read. The value at this location should be of the type ix_uint32* and aligned on a 32-bit boundary.

Output/Returns

Return Value An ix_uint32 representing the value at the specified location.

4. IX_RM_MEM_UINT64_READ

A macro used to read a 64-bit value from memory.

C Syntax

#define IX_RM_MEM_UINT64_READ(arg_pMemoryAddr)

Input

arg_pMemoryAddr: The pointer to the address of the memory location to be read. The value at this location should be of the type ix_uint64* and aligned on a 64-bit boundary.

Output/Returns

Return Value An ix_uint64 representing the value at the specified location.

5. IX_RM_MEM_UINT8_WRITE

Writes an 8-bit value to memory.

C Syntax

#define IX_RM_MEM_UINT8_WRITE(arg_pMemoryAddr, arg_Value)

Input

arg_pMemoryAddr: The address of the memory location to be written. The value at this location should be of the type ix_uint8*. The address need not be aligned.
arg_vValue: The value to be written to the specified location. The value should be of the type ix_uint8.

Output/Returns
Return Value An ix_uint8 representing the value written at the specified location.

6. IX_RM_MEM_UINT16_WRITE

Writes a 16-bit value to memory.
C Syntax
#define IX_RM_MEM_UINT16_WRITE(arg_pMemoryAddr, arg_Value)

Input
arg_pMemoryAddr: The address of the memory location to be written. The value at this location should be of the type ix_uint16*. The address should be 16-bit aligned.
arg_vValue: The value to be written to the specified location. The value should be of the type ix_uint16.

Output/Returns
Return Value An ix_uint16 representing the value written at the specified locatio

7. IX_RM_MEM_UINT32_WRITE

Writes a 32-bit value to memory.
C Syntax
#define IX_RM_MEM_UINT32_WRITE(arg_pMemoryAddr, arg_Value)

Input
arg_pMemoryAddr: The address of the memory location to be written. The value at this location should be of the type ix_uint32*. The address should be 32-bit aligned.
arg_vValue: The value to be written to the specified location. The value should be of the type ix_uint32.

Output/Returns
Return Value An ix_uint32 representing the value written at the specified location.

8. IX_RM_MEM_UINT64_WRITE

Writes a 64-bit value to memory.
C Syntax
#define IX_RM_MEM_UINT64_WRITE(arg_pMemoryAddr, arg_Value)

Input

arg_pMemoryAddr: The address of the memory location to be written. The value at this location should be of the type ix_uint64*. The address should be 64-bit aligned.

arg_vValue: The value to be written to the specified location. The value should be of the type ix_uint64.

Output/Returns

Return Value An ix_uint64 representing the value written at the specified location.