Low-level SCI software functional specification

Esprit Project 23174 - Software Infrastructure for SCI (SISCI)
Deliverable D.1.1.1

Version 2.1.1 - 15 March 1999

F. Giacomini[1,2], T. Amundsen[3], A. Bogaerts[1], R. Hauser[1], B. D. Johnsen[3], H. Kohmann[3], R. Nordstrøm[3], P. Werner[1]

[1]CERN, 1211 Geneva 23, Switzerland,

[2]Supported by the EU Contract Esprit 23174 (SISCI),

[3]Dolphin Interconnect Solutions AS, Oslo, Norway

Preface

Clusters of commodity processors interconnected by a fast network are an attractive option to build economically large multiprocessor systems. The Scalable Coherent Interface ("SCI", IEEE std. 1596-1992 [1]) provides a distributed shared memory architecture.

ESPRIT Project 23174 (Software Infrastructure for SCI, "SISCI") has set itself as a goal to define a common Application Programming Interface ("API") to serve as a basis for porting major applications to heterogeneous multi vendor SCI platforms. However, from the requirements analysis [2] it has appeared clear that a unique API for all applications is not realistic.

Recently other activities addressing the definition of an API for SCI or, more generally, for shared memory clusters have emerged. IEEE Working Group P1596.9 has proposed an API for the SCI Physical Layer, suitable for low level applications [3]. The Virtual Interface Architecture is an initiative supported by many major manufacturers (amongst them Intel, Compaq and Microsoft) to expedite the development of applications running in a System Area Network based on a high-performance network technology.

The functional specification of the API presented in this document is defined in ANSI C.

Contents

Preface . . . iii

Chapter 1
Introduction . . . 1
1.1 Basic Concepts . . . 2
1.2 Cluster Architecture . . . 3

Chapter 2
API Categories . . . 5
2.1 Data types . . . 6
2.2 General functions . . . 10
2.3 Shared Memory . . . 11
2.4 Direct Memory Access . . . 13
2.5 Block Operations . . . 15
2.6 Interrupts . . . 16
2.7 Privileged Operations . . . 17

Chapter 3
API Specification . . . 19
sci_address_t . . . 20
sci_cb_block_transfer_t . . . 21
sci_cb_dma_t . . . 22
sci_cb_interrupt_t . . . 23
sci_cb_local_segment_t . . . 24
sci_cb_remote_segment_t . . . 25
sci_dma_queue_state_t . . . 26
sci_segment_cb_reason_t . . . 27
sci_sequence_status_t . . . 28
SCIAbortBlockTransfer . . . 29
SCIAbortDMAQueue . . . 30
SCICheckSequence . . . 31
SCIClose . . . 32
SCIConnectInterrupt . . . 33
SCIConnectSCISpace . . . 34
SCIConnectSegment . . . 35
SCICreateDMAQueue . . . 37
SCICreateInterrupt . . . 38

SCICreateMapSequence 39
SCICreateSegment 40
SCIDisconnectInterrupt 42
SCIDisconnectSegment. 43
SCIDMAQueueState 44
SCIEnqueueDMATransfer 45
SCIFlushReadBuffers 47
SCIGetCSRRegister 48
SCIGetRemoteSegmentSize 49
SCIMapLocalSegment 50
SCIMapRemoteSegment 52
SCIMemCopy 54
SCIOpen 55
SCIPostDMAQueue 56
SCIPrepareSegment 57
SCIProbeNode 58
SCIQuery. 59
SCIRegisterSegmentMemory 61
SCIRemoveDMAQueue 62
SCIRemoveInterrupt 63
SCIRemoveSegment. 64
SCIRemoveSequence 65
SCIResetDMAQueue 66
SCISetCSRRegister 67
SCISetSegmentAvailable 68
SCISetSegmentUnavailable 69
SCIStartSequence 70
SCIStoreBarrier 71
SCITransferBlock 72
SCITransferBlockAsync 73
SCITriggerInterrupt 75
SCIUnmapSegment 76
SCIWaitForBlockTransfer 77
SCIWaitForDMAQueue 78
SCIWaitForInterrupt. 79
SCIWaitForLocalSegmentEvent 80
SCIWaitForRemoteSegmentEvent. 81

Bibliography. 83

Chapter 1

Introduction

This introductory chapter defines some terms that are used throughout the document and briefly describes the cluster architecture that represents the main objective of this functional specification.

1.1 Basic Concepts 2
1.2 Cluster Architecture 3

1.1 Basic Concepts

Processor
: a collection of one or more CPUs sharing memory using a local bus.

Host
: a processor with one or more SCI adapters.

SCI adapter
: an SCI interface. Each host can have one or more SCI adapters. A host-wide unique **number** is assigned to each adapter.

Local segment
: a memory segment located on the same processor where the application runs and accessed using the host memory interface. A unique host-wide **segment identifier** is assigned to each local segment.

Remote segment
: a memory segment accessed via an SCI link.

SCI node
: an **SCI node** is associated with each SCI adapter and has a unique network-wide **node identifier**[*].

Connected segment
: a remote segment for which the SCI base address and the size are known.

Mapped segment
: a local or remote (connected) segment mapped in the addressable space of a program.

[*] A node identifier usually means a logical node identifier, that is visible to the API user. It could not correspond to the value that actually appears as the source or the target in an SCI packet.

1.2 Cluster Architecture

The basic elements of a cluster are hosts interconnected to form SCI rings. Larger systems are obtained by interconnecting rings using switches. A host may be a single processor or an SMP containing several CPUs. Adapters connect a host to a ring. It is possible for a host to be connected to several rings. This allows for the construction of complex topologies (e.g. a two dimensional mesh). It may also be used to add redundancy and/or improve the bandwidth (e.g. two parallel counter rotating rings). Usually such architectures are obtained by using several adapters on one host or several link controllers on one adapter. These possibilities are illustrated in Figure 1.1.

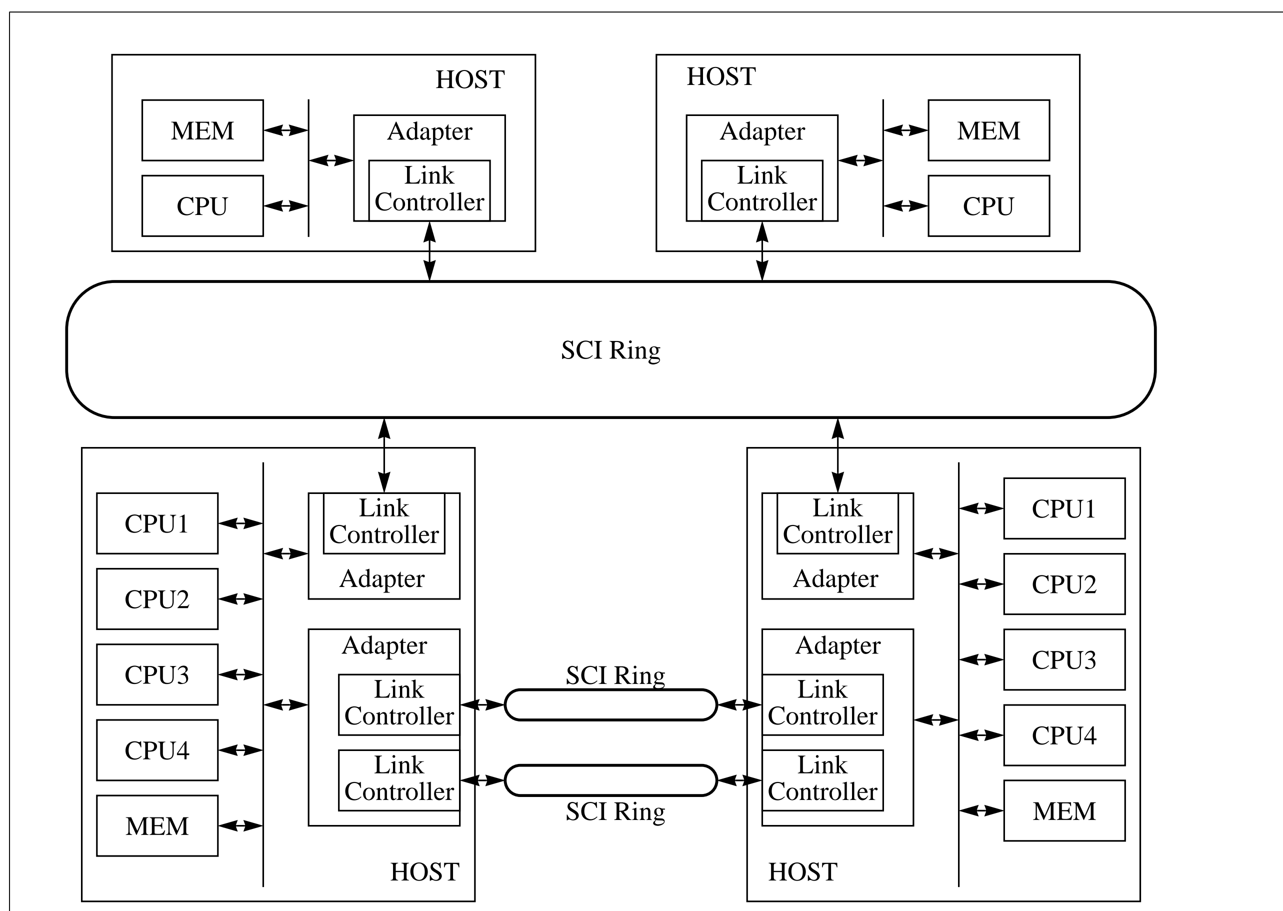

Figure 1.1 Example of a cluster of four hosts.

Adapters often contain subunits that implement specific SCI functions such as transparent remote memory access, DMA, packet mode, message mailboxes or interrupts. Most subunits have CSR registers that may be accessed locally via the host adapter interface or remotely over the SCI network. Unfortunately the IEEE standard 1212-1994 CSR architecture [5] is not always strictly followed.

Chapter 2

API Categories

This Application Programming Interface covers different aspects of the SCI technology and how it can be accessed by a user. The API items can then be grouped in different categories. The specification of functions and data types, presented in Chapter 3, is then preceded by a short introduction of these categories. For an easier consultation of the document, for each category a list of concerned API items is provided.

2.1 Data types 6
2.2 General functions 10
2.3 Shared Memory 11
2.4 Direct Memory Access 13
2.5 Block Operations 15
2.6 Interrupts. 16
2.7 Privileged Operations 17

2.1 Data types

2.1.1 Data formats

Parameters and return values of the API functions, other than the data types introduced in the following sections, are expressed in the machine native data types. The only assumption is that `int`s are at least 32 bits and `short`s are at least 16 bits. If, in the future, a specific size or endianness is needed, the Shared-Data Formats [6] shall be used.

2.1.2 Descriptors

Working with remote shared memories, DMA transfers and remote interrupts, the major communication features that this API offers, requires the use of logical entities like devices, memory segments, DMA queues. Each of these entities is characterize by a set of properties that should be manage as a unique object in order to avoid inconsistencies. To hide the details of the internal representation and management of such properties to an API user, a number of descriptors have been defined and made opaque: their contents can be referenced with a handle and can be modified only through the functions provided by the API.

The descriptors and their meaning are the following:

`sci_desc`
: It represents an SCI virtual device, that is a communication channel with the driver. Many virtual devices can be opened by the same application. It is initialized by calling the function `SCIOpen`.

`sci_local_segment`
: It represents a local memory segment and it is initialized when the segment is allocated by calling the function `SCICreateSegment`.

`sci_remote_segment`
: It represents a segment residing on a remote node. It is initialized by calling either the function `SCIConnectSegment` or the function `SCIConnectSCISpace`.

`sci_map`
: It represents a memory segment mapped in the process address space. It is initialized by calling either the function `SCIMapRemoteSegment` or the function `SCIMapLocalSegment`.

`sci_sequence`
: It represents a sequence of operations involving communication with remote nodes. It is used to check if errors have occurred during a data transfer. The descriptor is initialized when the sequence is created by calling the function `SCICreateMapSequence`.

`sci_dma_queue`
: It represents a chain of specifications of data transfers to be performed using the DMA engine available on the SCI adapter. The descriptor is initialized when the chain is created by calling the function `SCICreateDMAQueue`.

`sci_local_interrupt`
: It represents an instance of an interrupt that an application has made available to remote nodes. It is initialized when the interrupt is created by calling the function `SCICreateInterrupt`.

`sci_remote_interrupt`
: It represents an interrupt that can be triggered on a remote node. It is initialized by calling the function `SCIConnectInterrupt`.

`sci_block_transfer`
: It represents an asynchronous transfer of a block of data. It is initialized when the function `SCITransferBlockAsync` is invoked.

Each of the above descriptors is an opaque data type and can be referenced only via a handle. The name of the handle type is given by the name of the descriptor type with a trailing `_t`.

No automatic cleanup of the resources represented by the above descriptors is performed, rather it should be provide by the API client*. Resources cannot be released (and the corresponding descriptors deallocated) until all the dependent resources have been previously released. The dependencies between resource classes can be derived by the function specifications and are illustrated in Figure 2.1.

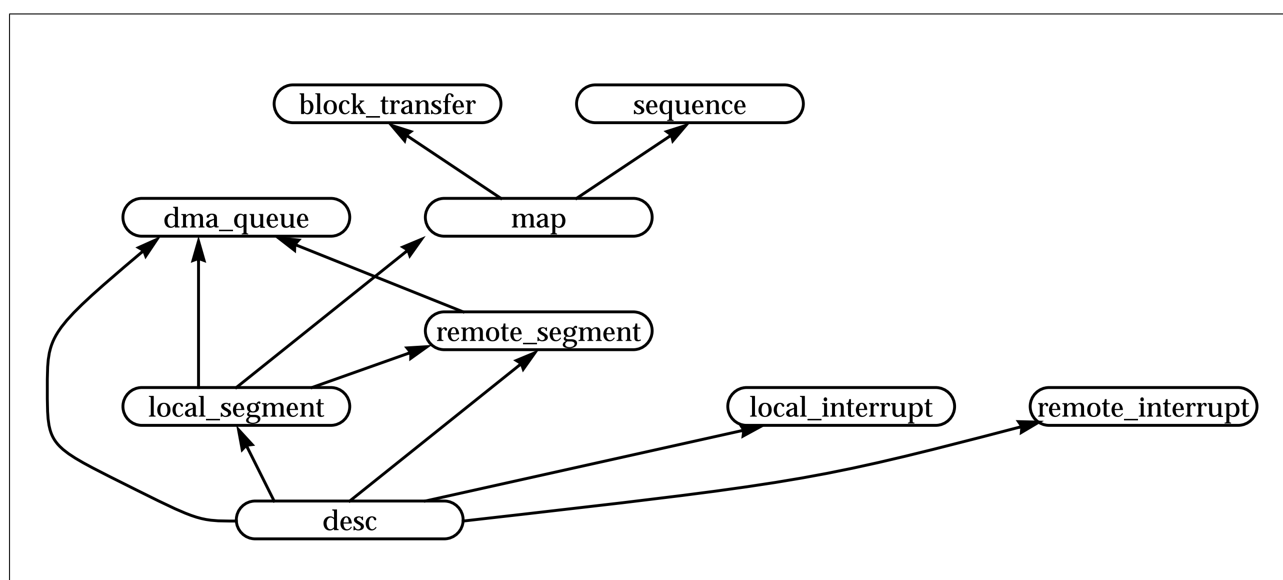

Figure 2.1 Resource class dependencies. Each resource class is represented by the corresponding descriptor (without the `sci_` prefix).

* There is only one exception to this statement: the descriptor representing a block transfer sci_block_transfer is temporary and is valid until the end of the execution of the corresponding callback.

2.1.3 Flags

Nearly all the functions included in this API accept a `flags` parameter in input. It is used to obtain from a function invocation an effect that slightly differs from its default semantics (e.g. choosing between a blocking and a non-blocking version of an operation).

In Chapter 3 each function specification is followed by a list of accepted flags. Only the flags that change the default behaviour are defined. Several flags can be `OR`ed together to specify a combined effect. The `flags` parameter, represented with an `unsigned int`, has then to be considered a bitmask.

Most of the functions do not accept any flag. The parameter is nonetheless left in the specification, because it could become useful in view of future extensions, and the implementation shall check it to be 0.

A flag value starts with the prefix `SCI_FLAG_`.

2.1.4 Errors

Most of the API functions return an error code as an output parameter to indicate if the execution succeeded or failed. The error codes are collected in an enumeration type called `sci_error_t`. Each value starts with the prefix `SCI_ERR_`. The code denoting success is `SCI_ERR_OK` and an application should check that each function call returns this value.

In Chapter 3 each function specification is followed by a list of possible errors that are typical for that function. There are however common or very generic errors that are not repeated every time, unless they do not have a particular meaning for that function:

`SCI_ERR_NOT_IMPLEMENTED` the function is not implemented

`SCI_ERR_ILLEGAL_FLAG` the `flags` value passed to the function contains an illegal component. The check is done even if the function does not accept any flag (i.e. it accepts only the default value 0)

`SCI_ERR_FLAG_NOT_IMPLEMENTED` the `flags` value passed to the function is legal but the operation corresponding to one of its components is not implemented

`SCI_ERR_ILLEGAL_PARAMETER` one of the parameters passed to the function is illegal

`SCI_ERR_NOSPC` the function is unable to allocate some needed operating system resources

`SCI_ERR_API_NOSPC` -the function is unable to allocate some needed API resources

`SCI_ERR_HW_NOSPC` the function is unable to allocate some hardware resources

`SCI_ERR_SYSTEM` the function has encountered a system error; `errno` should be checked

Each function requiring a local adapter number can generate the following errors:

`SCI_ERR_ILLEGAL_ADAPTERNO` the adapter number is out of the legal range

`SCI_ERR_NO_SUCH_ADAPTERNO` the adapter number is legal but it does not exist

Each function requiring a node identifier can generate the following errors:

`SCI_ERR_NO_SUCH_NODEID` a node with the specified identifier does not exist

`SCI_ERR_ILLEGAL_NODEID` the node identifier is illegal

2.1.5 Other data types

Besides the data types specified in the previous sections others are used:

sci_address_t 20
sci_segment_cb_reason_t 27
sci_dma_queue_state_t 26
sci_sequence_status_t 28

2.1.6 Callback functions

Some API functions allow the specification of a callback function among their input parameters to be invoked when an asynchronous event occurs. There are different types of callback functions, depending on their use:

sci_cb_block_transfer_t 21
sci_cb_dma_t 22
sci_cb_interrupt_t 23
sci_cb_local_segment_t 24
sci_cb_remote_segment_t 25

2.2 General functions

In order to use correctly the SCI network, an application is required to execute some operations like opening or closing a communication channel with the SCI driver. For using effectively the SCI network an application may also need some information about the local or a remote node.

SCIOpen. 55
SCIClose. 32
SCIQuery 59
SCIProbeNode 58

2.3 Shared Memory

The Scalable Coherent Interface implements a remote shared memory approach in the data transfers between processors: an application can map into its own address space a memory segment actually residing on another node; then read and write operations from or to this memory segment are automatically and transparently converted by the hardware in remote operations. This API provides full support for creating and exporting local memory segments, for connecting to and mapping remote memory segments, for checking whether errors have occurred during a data transfer.

The functions included in this category actually concern three different aspects:

- memory management

SCICreateSegment 40
SCIRegisterSegmentMemory 61
SCIRemoveSegment 64
SCIPrepareSegment. 57

- connection management

SCISetSegmentAvailable 68
SCISetSegmentUnavailable 69
SCIConnectSegment 35
SCIConnectSCISpace 34
SCIDisconnectSegment. 43
SCIWaitForLocalSegmentEvent 80
SCIWaitForRemoteSegmentEvent. 81

- proper shared memory

SCIMapLocalSegment 50
SCIMapRemoteSegment 52
SCIUnmapSegment. 76
SCIMemCopy 54
SCICreateMapSequence 39
SCIRemoveSequence 65
SCIStartSequence 70
SCICheckSequence 31
SCIFlushReadBuffers 47
SCIStoreBarrier 71

The functions concerning memory management and connection management are in common with Direct Memory Access, presented in Section 2.4. To simplify the document, however, they are included in the Shared Memory category.

Memory and connection management functions affect the state of a local segment, whose state diagram is shown in Figure 2.2.

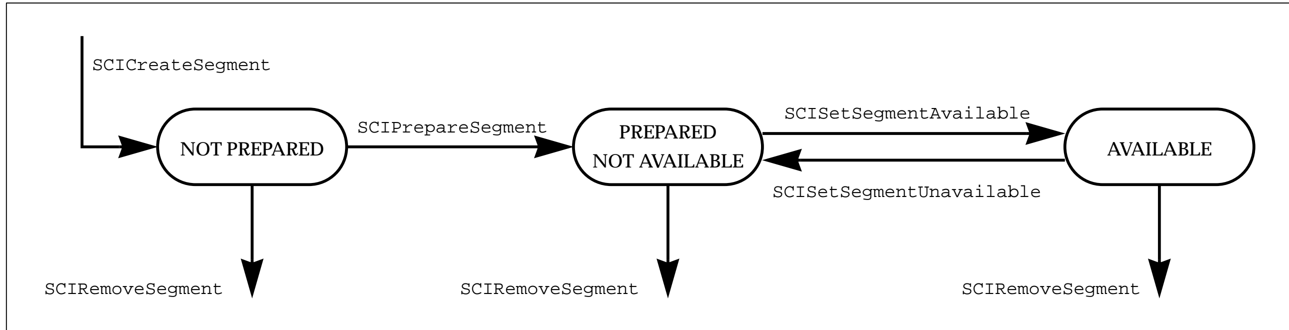

Figure 2.2 State diagram for a local segment.

The state of a remote segment, shown in Figure 2.3, instead depends on what happens on the network or on the node where the segment physically resides.

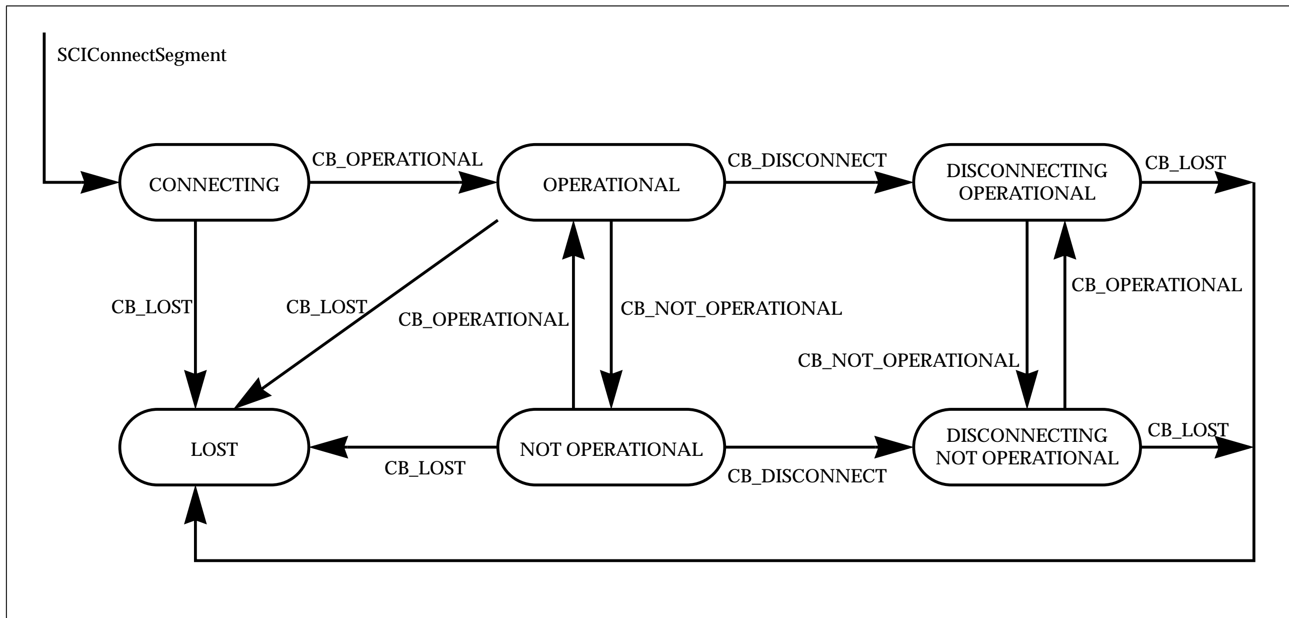

Figure 2.3 State diagram for a remote segment. The transitions are marked with callback reasons. `SCIDisconnectSegment` can be called from each state to exit the state diagram.

2.4 Direct Memory Access

The drawback of the shared memory approach to data transfers is that the CPU is busy reading or writing data from or to remote memory (programmed I/O). An alternative is to use the DMA engine available on an SCI adapter. The application (i.e. the CPU) specifies a queue of data transfers and passes it to the DMA engine. Then the CPU is free either to wait for the completion of the transfer or to do something else. In the latter case it is possible to specify a callback function that is invoked when the transfer has finished.

SCICreateDMAQueue 37
SCIRemoveDMAQueue 62
SCIEnqueueDMATransfer 45
SCIPostDMAQueue. 56
SCIWaitForDMAQueue 78
SCIAbortDMAQueue. 30
SCIResetDMAQueue 66
SCIDMAQueueState 44

Other than the typical DMA functions listed above there are others that concern the memory and the connection management, that have been classified under Shared Memory, in Section 2.3:

SCICreateSegment 40
SCIRegisterSegmentMemory 61
SCIRemoveSegment 64
SCIPrepareSegment. 57
SCISetSegmentAvailable 68
SCISetSegmentUnavailable 69
SCIConnectSegment 35
SCIConnectSCISpace 34
SCIDisconnectSegment. 43
SCIWaitForLocalSegmentEvent 80
SCIWaitForRemoteSegmentEvent. 81

The state diagram for a DMA queue is shown in Figure 2.4.

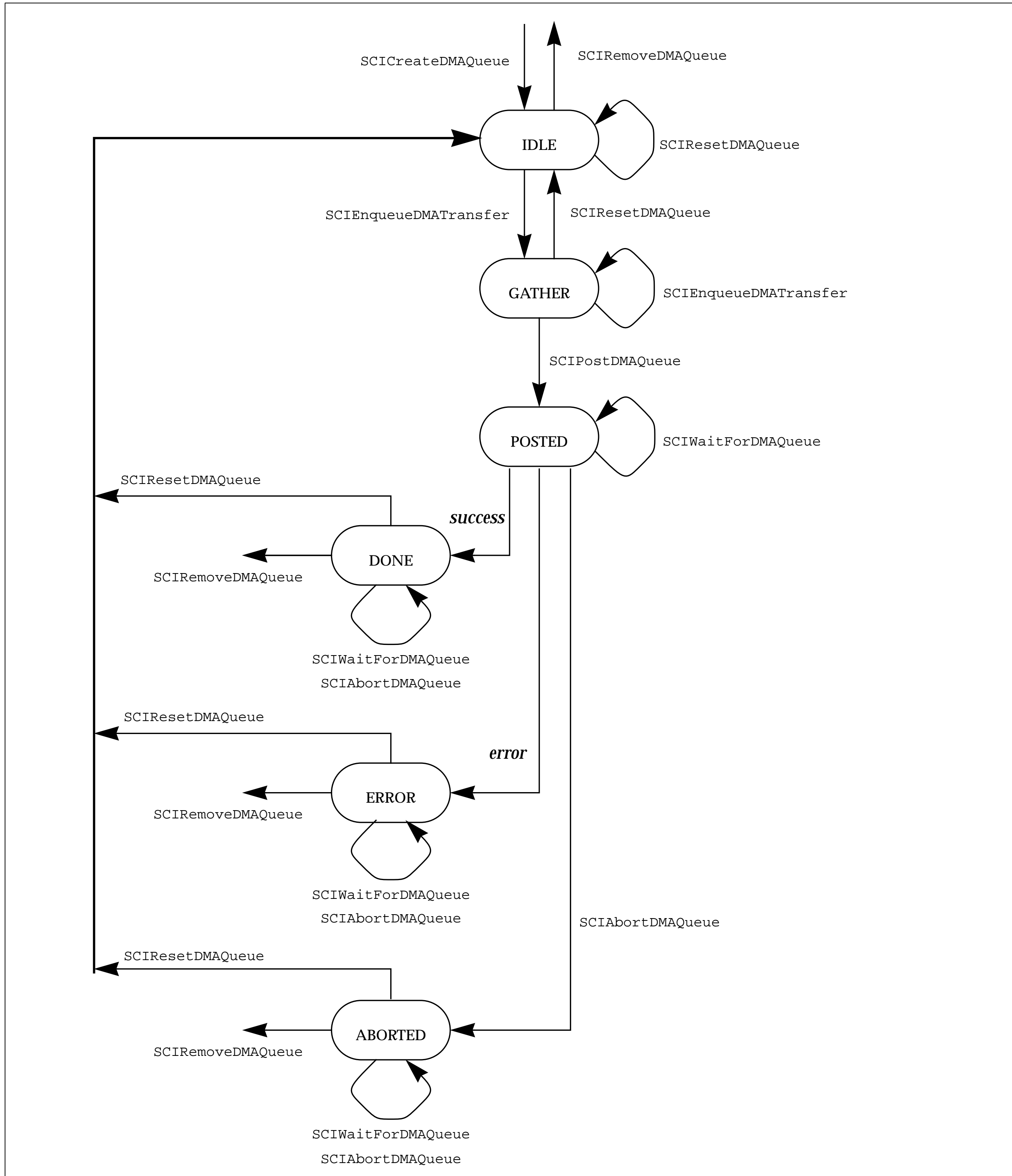

Figure 2.4 State diagram for DMA operations; the transitions not specified in the diagram are illegal.

2.5 Block Operations

Block operations allow to move large blocks of data between hosts in a single operation, using the underlying most efficient approach. The operation can be synchronous or asynchronous; in the latter case a callback can be specified to be invoked after the transfer has completed or when an error occurs.

SCITransferBlock 72
SCITransferBlockAsync. 73
SCIWaitForBlockTransfer. 77
SCIAbortBlockTransfer 29

2.6 Interrupts

Triggering an interrupt on a remote node should be considered a fast way to notify an application running remotely that something has happened. An interrupt is identified by a unique number and this is practically the only information an application gets when it is interrupted, either synchronously or asynchronously.

SCICreateInterrupt 38
SCIRemoveInterrupt 63
SCIConnectInterrupt 33
SCIDisconnectInterrupt 42
SCITriggerInterrupt 75
SCIWaitForInterrupt 79

2.7 Privileged Operations

One of the guidelines of the specification of this API has been to encapsulate most of the details of the underlying hardware and of the low-level software. If this approach helps in avoiding inconsistent use of the available resources, it could reduce the flexibility in using the technology and this might be unacceptable for certain applications.

It could then be useful to have some functions to access low-level features of the technology. The compromise is to enable this low-level functions only to some “expert” users, while not to provide them to a general user. For the moment the API provides the possibilities to connect to an SCI space window using the direct SCI address and to access local and remote CSR spaces.

SCIConnectSCISpace 34

SCIGetCSRRegister 48

SCISetCSRRegister 67

Chapter 3
API Specification

In this chapter the API items are specified in alphabetical order. For each item (data type or function), the specification consists of:

- the category to which the item belongs, as from Chapter 2;
- the item name;
- a description of the item's functionality;
- the item syntax and its description.

A parameter in a function specification is always preceded by a qualifier indicating if it is an input (IN), an output (OUT) or an input/output (INOUT) parameter.

The default behaviour of a function is not thread-safe. The thread-safety characteristic is associated with a virtual device descriptor and it is explicitly mentioned when the virtual device is opened. The behaviour of a function is then thread-safe if it uses a thread-safe virtual device.

Data Types

sci_address_t

Synopsis A variable of type `sci_address_t` represent a 64-bit SCI address. It is passed to the `SCIConnectSCISpace` function as the base address of a window in the SCI address space.

```
typedef struct {
    unsigned int   hi;
    unsigned short lo;
    unsigned short nodeId;
} sci_address_t;
```

Members

`hi`	high bits of the address offset
`lo`	low bits of the address offset
`nodeId`	node identifier

Data Types

sci_cb_block_transfer_t

Synopsis A function of type `sci_cb_block_transfer_t` can be specified when a remote memory block is transferred asynchronously with `SCITransferBlockAsync` and will be invoked when the operation completes or when an error occurs.

```
int
(*sci_cb_block_transfer_t)( IN void *arg,
                            IN sci_block_transfer_t block,
                            IN sci_error_t status );
```

Parameters

`arg`	user-defined argument passed to the callback function
`block`	handle to the block transfer descriptor
`status`	

Return value The return value (currently not specified) is intercepted by the driver.

Data Types

sci_cb_dma_t

Synopsis A function of type `sci_cb_dma_t` can be specified when a DMA queue is posted using `SCIPostDMAQueue` and will be invoked when the transfer has completed, either successfully or with an error.

```
int
(*sci_cb_dma_t)( IN void* arg,
                 IN sci_dma_queue_t queue );
```

Parameters

`arg`	user-defined argument passed to the callback function
`queue`	handle to the DMA queue descriptor

Return value The return value (currently not specified) is intercepted by the driver.

Data Types

sci_cb_interrupt_t

Synopsis A function of type `sci_cb_interrupt_t` can be specified when an interrupt is created with `SCICreateInterrupt` and it will be invoked asynchronously when the interrupt will be triggered from a remote node.

```
int
(*sci_cb_interrupt_t)( IN void *arg,
                       IN sci_local_interrupt_t interrupt,
                       IN sci_error_t status );
```

Parameters

`arg`	user-defined argument passed to the callback function
`interrupt`	handle to the triggered interrupt descriptor
`status`	status information

Return value The return value (currently not specified) is intercepted by the driver.

Data Types

sci_cb_local_segment_t

Synopsis A function of type `sci_cb_local_segment_t` can be specified when a segment is created with `SCICreateSegment` and will be invoked asynchronously when a remote node connects to or disconnects from the segment using respectively `SCIConnectSegment` and `SCIDisconnectSegment`. The same callback function is also invoked whenever a problem affects the connection.

```
int
(*sci_cb_local_segment_t)( IN void* arg,
                           IN sci_local_segment_t segment,
                           IN sci_segment_cb_reason_t reason,
                           IN unsigned int nodeId,
                           IN unsigned int localAdapterNo );
```

Parameters

`arg`	user-defined argument passed to the callback function
`segment`	handle to the local segment descriptor affected by the asynchronous event
`reason`	reason why the callback function has been invoked
`nodeId`	identifier of the remote node that has provoked, directly or indirectly, the asynchronous event
`localAdapterNo`	number of the local adapter that received the asynchronous event

Return value The return value (currently not specified) is intercepted by the driver.

Data Types

sci_cb_remote_segment_t

Synopsis A function of type `sci_cb_remote_segment_t` can be specified when the connection to a memory segment is requested calling `SCIConnectSegment` and will be invoked asynchronously when the connection completes (if `SCIConnectSegment` is asynchronous), when the local node asks for disconnecting (calling `SCISetSegmentUnavailable` with the `SCI_FLAG_NOTIFY flag`) or when a problem affects the connection.

```
int
(*sci_cb_remote_segment_t)( IN void* arg,
                            IN sci_remote_segment_t segment,
                            IN sci_segment_cb_reason_t reason,
                            IN sci_error_t status );
```

Parameters

`arg`	user-defined argument passed to the callback function
`segment`	handle to the remote segment descriptor
`reason`	reason why the callback function has been invoked
`status`	status of the remote segment

Return value The return value (currently not specified) is intercepted by the driver.

Data Types

sci_dma_queue_state_t

Synopsis The `sci_dma_queue_state_t` type enumerates the values that denote the states that a DMA queue can occupy. Figure 2.4 shows the state diagram for a DMA queue.

```
typedef enum {
    SCI_DMAQUEUE_IDLE,
    SCI_DMAQUEUE_GATHER,
    SCI_DMAQUEUE_POSTED,
    SCI_DMAQUEUE_DONE,
    SCI_DMAQUEUE_ABORTED,
    SCI_DMAQUEUE_ERROR,
} sci_dma_queue_state_t;
```

Values

`SCI_DMAQUEUE_IDLE` the queue has been created but no data transfers have been yet enqueued

`SCI_DMAQUEUE_GATHER` data transfers are being enqueued

`SCI_DMAQUEUE_POSTED` the queue has been posted to the DMA engine

`SCI_DMAQUEUE_DONE` all the transfers included in the queue have been completed successfully

`SCI_DMAQUEUE_ABORTED` the queue has been aborted

`SCI_DMAQUEUE_ERROR` at least one transfer included in the queue has failed

Data Types

sci_segment_cb_reason_t

Synopsis The type `sci_segment_cb_reason_t` enumerates the values that are used to indicate to a callback function of type `sci_cb_remote_segment_t` or `sci_cb_local_segment_t` why it has been invoked, that could be specified respectively when a segment is connected (`SCIConnectSegment`) or when a segment is created (`SCICreateSegment`).

```
typedef enum {
    SCI_CB_CONNECT,
    SCI_CB_DISCONNECT,
    SCI_CB_NOT_OPERATIONAL,
    SCI_CB_OPERATIONAL,
    SCI_CB_LOST
} sci_segment_cb_reason_t;
```

Values The meaning of the values is different on a local and on a remote node.

For a local segment:

`SCI_CB_CONNECT` a new connection has been established from a remote node

`SCI_CB_DISCONNECT` an existing connection from a remote node has been released

`SCI_CB_NOT_OPERATIONAL` the route to a connected remote node is currently not operational

`SCI_CB_OPERATIONAL` the route to a connected remote node is operational again

`SCI_CB_LOST` an unrecoverable event has occurred on a connected remote node (e.g. it can have crashed)

For a remote segment:

`SCI_CB_CONNECT` an asynchronous connection has completed successfully

`SCI_CB_DISCONNECT` the local node has invoked `SCISetSegmentUnavailable` with the `SCI_FLAG_NOTIFY` flag on

`SCI_CB_NOT_OPERATIONAL` the route to the local node is currently not operational

`SCI_CB_OPERATIONAL` the route to the local node is operational again

`SCI_CB_LOST` an unrecoverable event has occurred on the local node (e.g. the local node has invoked `SCISetSegmentUnavailable` with the `SCI_FLAG_FORCE_DISCONNECT` flag)

In Figure 2.3 each of the values denotes a transition in the state diagram of a remote segment.

Data Types

sci_sequence_status_t

Synopsis The type `sci_sequence_status_t` enumerates the values returned by `SCIStartSequence` and `SCICheckSequence`.

```
typedef enum {
    SCI_SEQ_OK,
    SCI_SEQ_RETRIABLE,
    SCI_SEQ_NOT_RETRIABLE,
    SCI_SEQ_PENDING
} sci_sequence_status_t;
```

Values

`SCI_SEQ_OK`	no errors: the sequence of reads and writes can continue
`SCI_SEQ_RETRIABLE`	non-fatal error: the failed operation can be immediately retried
`SCI_SEQ_NOT_RETRIABLE`	fatal error (probably notified also via a callback to the segment): need to wait until the situation is normal again
`SCI_SEQ_PENDING`	an error is pending, `SCIStartSequence` should be called until it returns `SCI_SEQ_OK`

`SCIStartSequence` can only return `SCI_SEQ_OK` or `SCI_SEQ_PENDING`.

Block Operations

SCIAbortBlockTransfer

Synopsis `SCIAbortBlockTransfer` aborts an ongoing asynchronous block transfer started with `SCITransferBlockAsync`. There is a potential race condition if the call happens when the transfer is completing. If the transfer has already completed a call to `SCIAbortBlockTransfer` fails because the block transfer descriptor has been already automatically destroyed and its handle has become invalid.

```
void
SCIAbortBlockTransfer( IN sci_block_transfer_t block,
                       IN unsigned int flags,
                       OUT sci_error_t* error );
```

Parameters

`block`	handle to the block descriptor
`flags`	not used
`error`	error information

Errors On successful completion, error points to the SCI_ERR_OK value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_OPERATION`	the call to the function is invoked when the transfer has already completed

DMA

SCIAbortDMAQueue

Synopsis `SCIAbortDMAQueue` aborts a DMA transfer initiated with `SCIPostDMAQueue`.

Calling this function is really meaningful only if the queue is in the POSTED state. If the function is successful the final state is ABORTED (see `sci_dma_queue_state_t`). If the state is already ABORTED or if it is DONE or ERROR, the call is equivalent to a no-op. In all the other cases the call is illegal and the error is detected. There is a potential race condition if the call happens when the state is already changing from POSTED to either DONE or ERROR because the transfer has completed or an error has occurred. To check what happened the program should call `SCIDMAQueueState`.

```
void
SCIAbortDMAQueue( IN sci_dma_queue_t dq,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

`dq`	handle to the DMA queue descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_OPERATION`	The state transition implied by this operation is not allowed in the current state of the queue

Shared Memory

SCICheckSequence

Synopsis `SCICheckSequence` checks if any error has occurred in a data transfer controlled by a sequence since the last check. The previous check can have been done by calling either `SCIStartSequence`, that also initiates the sequence, or `SCICheckSequence` itself. `SCICheckSequence` can be invoked several times in a row without calling `SCIStartSequence`, as far as it does not fail, returning SCI_SEQ_OK (i.e. there were no transmission errors in the sequence). If the return value is SCI_SEQ_RETRIABLE the operation can be immediately retried. A return value SCI_SEQ_NOT_RETRIABLE means that there have been a fatal error, probably also notified via callbacks to the corresponding mapped segment; it is not legal to execute other read or write operations on the segment until a call to `SCIStartSequence` does not fail. As well, if the return value is SCI_SEQ_PENDING it is not legal to perform read or write operations on the segment until a call to `SCIStartSequence` does not fail.

The default behaviour of `SCICheckSequence` is to flush the write buffers of the SCI adapter and to wait for all the outstanding write requests to be completed. To prevent this actions the caller has to use specific flags.

```
sci_sequence_status_t
SCICheckSequence( IN sci_sequence_t sequence,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

`sequence`	handle to a sequence descriptor
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_NO_FLUSH`	do not flush the write buffers
`SCI_FLAG_NO_STORE_BARRIER`	do not wait for outstanding write requests

Return value The function returns the status of the sequence.

Errors If the return value is SCI_SEQ_OK, `error` points to the `SCI_ERR_OK` value. There are no specific error values for this function.

General

SCIClose

Synopsis `SCIClose` closes an open SCI virtual device, destroying its descriptor. After this call the handle to the descriptor becomes invalid and should not be used. `SCIClose` does not deallocate possible resources that are still in use, rather it fails if some of them exist, according to the dependencies shown if Figure 2.1.

```
void
SCIClose( IN sci_desc_t sd,
          IN unsigned int flags,
          OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_BUSY`	some resources depending on this virtual device are still in use

Interrupts

SCIConnectInterrupt

Synopsis `SCIConnectInterrupt` connects the caller to an interrupt resource available on a remote node (see `SCICreateInterrupt`). The function creates and initializes a descriptor for the connected interrupt.

```
void
SCIConnectInterrupt( IN sci_desc_t sd,
                     OUT sci_remote_interrupt_t* interrupt,
                     IN unsigned int nodeId,
                     IN unsigned int localAdapterNo,
                     IN unsigned int interruptNo,
                     IN unsigned int timeout,
                     IN unsigned int flags,
                     OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`interrupt`	handle to a new remote interrupt descriptor
`nodeId`	identifier of the remote node where the interrupt has been created
`localAdapterNo`	number of the local adapter used for the connection
`interruptNo`	interrupt number
`timeout`	time in milliseconds to wait before giving up
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_NO_SUCH_INTNO`	an interrupt with this number does not exist
`SCI_ERR_CONNECTION_REFUSED`	the connection attempt has been refused by the remote node
`SCI_ERR_TIMEOUT`	timeout has expired

Privileged

SCIConnectSCISpace

Synopsis `SCIConnectSCISpace` connects an application directly to a window in the SCI address space, defined by its base address and its size, without any restriction. The function creates and initializes a descriptor for the connected segment. The whole responsibility of handling the segment is left to the programmer: no state diagram is defined, no callbacks can be specified, no callbacks are invoked on the node where the memory actually resides. The function has only the synchronous version; if the connection fails the returned handle is not valid and should not be used; in this case the related descriptor need not be destroyed with `SCIDisconnectSegment`.

Once the SCI window has been connected, it can either be mapped in the address space of the program (see `SCIMapRemoteSegment`) or be used directly for DMA transfers (see `SCIEnqueueDMATransfer`).

```
void
SCIConnectSCISpace( IN sci_desc_t sd,
                    OUT sci_remote_segment_t* segment,
                    IN sci_address_t address,
                    IN unsigned int size,
                    IN unsigned int localAdapterNo,
                    IN unsigned int flags,
                    OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`segment`	handle to the new connected segment descriptor
`address`	base address of the SCI window
`size`	size of the SCI window
`localAdapterNo`	number of the local adapter used for the connection
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_SIZE_ALIGNMENT`	the window size is not correctly aligned as required by the implementation

Shared Memory

SCIConnectSegment

Synopsis `SCIConnectSegment` connects an application to a memory segment made available on a remote node (see `SCISetSegmentAvailable`) and creates and initializes a descriptor for the connected segment. A call to this function enters the state diagram for a remote segment shown in Figure 2.3. If a timeout different from `SCI_INFINITE_TIMEOUT` is passes to the function, the attempt to connect gives up after the specified number of milliseconds.

The connection operation is by default synchronous: the function returns only when the operation has completed; a failure exits the state diagram and gives back a handle that is not valid and that should not be used.

If the flag `SCI_FLAG_ASYNCHRONOUS_CONNECT` is specified the connection is instead asynchronous: the function returns immediately with a valid handle. In case of failure, the descriptor has to be explicitly destroyed calling `SCIDisconnectSegment`.

A callback function can be specified to be invoked when an event concerning the segment happens; the intention to use the callback has to be explicitly declared with the flag `SCI_FLAG_USE_CALLBACK`. Alternatively, interesting events can be caught using the function `SCIWaitForRemoteSegmentEvent`.

Once a memory segment has been connected, it can either be mapped in the address space of the program (see `SCIMapRemoteSegment`) or be used directly for DMA transfers (see `SCIEnqueueDMATransfer`).

A successful connection also generates an `SCI_CB_CONNECT` event directed to the application that created the segment (see `SCICreateSegment and sci_cb_local_segment_t`).

```
void
SCIConnectSegment( IN sci_desc_t sd,
                   OUT sci_remote_segment_t* segment,
                   IN unsigned int nodeId,
                   IN unsigned int segmentId,
                   IN unsigned int localAdapterNo,
                   IN sci_cb_remote_segment_t callback,
                   IN void* callbackArg,
                   IN unsigned int timeout,
                   IN unsigned int flags,
                   OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`segment`	handle to the new connected segment descriptor
`nodeId`	identifier of the node where the segment is allocated
`segmentId`	identifier of the segment to connect
`localAdapterNo`	number of the local adapter used for the connection
`callback`	function called when an asynchronous event affecting the segment occurs
`callbackArg`	user-defined parameter passed to the callback function
`timeout`	time in milliseconds to wait for the connection to complete
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_USE_CALLBACK` the specified callback is active

`SCI_FLAG_ASYNCHRONOUS_CONNECT` the connection is asynchronous

Errors

On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_NO_SUCH_SEGMENT` the segment to connect could not be found

`SCI_ERR_CONNECTION_REFUSED` the connection attempt has been refused by the remote node

`SCI_ERR_TIMEOUT` the function timed out

`SCI_ERR_NO_LINK_ACCESS` it was not possible to communicate via the local adapter

`SCI_ERR_NO_REMOTE_LINK_ACCESS` it was not possible to communicate via a remote switch port

DMA

SCICreateDMAQueue

Synopsis `SCICreateDMAQueue` allocates resources for a queue of DMA transfers and creates and initializes a descriptor for the new queue. After the creation the state of the queue is IDLE (see `sci_dma_queue_state_t`).

All the segments involved in the transfers included in the same DMA queue must use the same adapter, which is specified as a parameter in this function.

If a handle to an existing queue is passed to this function it is overwritten with the handle to a new queue. The old queue is not affected but it may not be accessible any more.

```
void
SCICreateDMAQueue( IN sci_desc_t sd,
                   OUT sci_dma_queue_t* dq,
                   IN unsigned int localAdapterNo,
                   IN unsigned int maxEntries,
                   IN unsigned int flags,
                   OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`dq`	handle to the new DMA queue descriptor
`localAdapterNo`	number of the adapter whose DMA engine will be used for the transfers
`maxEntries`	maximum number of entries allowed in the DMA queue
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

Interrupts

SCICreateInterrupt

Synopsis `SCICreateInterrupt` creates an interrupt resource and make it available to remote nodes and initializes a descriptor for the interrupt. An interrupt is associated by the driver with a unique number. If the flag `SCI_FLAG_FIXED_INTNO` is specified, the function tries to use the number passed by the caller.

```
void
SCICreateInterrupt( IN sci_desc_t sd,
                    OUT sci_local_interrupt_t* interrupt,
                    IN unsigned int localAdapterNo,
                    INOUT unsigned int* interruptNo,
                    IN sci_cb_interrupt_t callback,
                    IN void* callbackArg,
                    IN unsigned int flags,
                    OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`interrupt`	handle to the new interrupt descriptor
`localAdapterNo`	number of the local adapter used to make the interrupt available to remote nodes
`interruptNo`	number assigned to the interrupt
`callback`	function called when the interrupt is triggered
`callbackArg`	user-defined parameter passed to the callback function
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_USE_CALLBACK`	the specified callback is active
`SCI_FLAG_FIXED_INTNO`	the interrupt number is specified by the caller

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_INTNO_USED`	the interrupt number is already used

Shared Memory

SCICreateMapSequence

Synopsis `SCICreateMapSequence` creates and initializes a new sequence descriptor that can be used to check for errors occurring in a transfer of data from or to a mapped segment.

If the flag `SCI_FLAG_FAST_BARRIER` is specified, when a store barrier operation is applied to the sequence, it is executed in the fastest possible way allowed by the SCI adapter. There could be a limited number of fast store barrier resources available, therefore `SCICreateMapSequence` fails if none are left.

```
void
SCICreateMapSequence( IN sci_map_t map,
                      OUT sci_sequence_t* sequence,
                      IN unsigned int flags,
                      OUT sci_error_t* error );
```

Parameters

`map`	handle to a valid mapped segment descriptor
`sequence`	handle to the new sequence descriptor
`flags`	see below
`error`	error information

Flags `SCI_FLAG_FAST_BARRIER` use the fast store barrier

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

Shared Memory

SCICreateSegment

Synopsis `SCICreateSegment` allocates a memory segment and creates and initializes a descriptor for a local segment. A host-wide unique identifier is associated to the new segment. This function causes a local segment to enter its state diagram, shown in Figure 2.2.

A callback function can be specified to be invoked when an event concerning the segment happens (see `sci_segment_cb_reason_t`); the intention to use the callback has to be explicitly declared with the flag `SCI_FLAG_USE_CALLBACK`. Alternatively, interesting events can be caught using the function `SCIWaitForLocalSegmentEvent`.

If the flag `SCI_FLAG_EMPTY` is specified, no memory is allocated for the segment and only the descriptor is initialized. Using the flag `SCI_FLAG_PRIVATE` declares that the segment will never be made available for external connections (see `SCISetSegmentAvailable`); in this case the specified segment identifier is meaningless, avoiding the internal check for its uniqueness. These two flags are useful to transform a user-allocated piece of memory (e.g. via `malloc`) into a mapped segment to be used in a block transfer (see `SCITransferBlock` and `SCITransferBlockAsync`): an empty and private segment is first created and then associated to the user-allocated memory (see `SCIRegisterSegmentMemory`); the segment can then be transformed in a mapped segment (see `SCIMapLocalSegment`) and possibly prepared for a DMA transfer (see `SCIPrepareSegment`).

```
void
SCICreateSegment( IN sci_desc_t sd,
                  OUT sci_local_segment_t* segment,
                  IN unsigned int segmentId,
                  IN unsigned int size,
                  IN sci_cb_local_segment_t callback,
                  IN void* callbackArg,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`segment`	handle to the new local segment descriptor
`segmentId`	segment identifier
`size`	segment size; if `SCI_FLAG_EMPTY` is specified, `size` means the maximum size of the memory area that can be associated with this local segment
`callback`	callback function called when an asynchronous event

	affecting the local segment occurs
callbackArg	user-defined argument passed to the callback function
flags	see below
error	error information

Flags

SCI_FLAG_USE_CALLBACK	the specified callback is active
SCI_FLAG_EMPTY	no memory is allocated
SCI_FLAG_PRIVATE	the segment will not be made available to external nodes

Errors On successful completion, error points to the SCI_ERR_OK value; otherwise to one of the following:

SCI_ERR_SEGMENTID_USED	the segment identifier is already used
SCI_ERR_SIZE_ALIGNMENT	the segment size is not correctly aligned as required by the implementation

Interrupts

SCIDisconnectInterrupt

Synopsis `SCIDisconnectInterrupt` disconnects an application from a remote interrupt resource and deallocates the corresponding descriptor. After this call the handle to the descriptor becomes invalid and should not be used.

```
void
SCIDisconnectInterrupt( IN sci_remote_interrupt_t interrupt,
                        IN unsigned int flags,
                        OUT sci_error_t* error );
```

Parameters

`interrupt`	handle to the remote interrupt descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

Shared Memory

SCIDisconnectSegment

Synopsis `SCIDisconnectSegment` disconnects from a remote segment connected by calling `SCIConnectSegment` or `SCIConnectSCISpace` and deallocates the corresponding descriptor. After this call the handle to the descriptor becomes invalid and should not be used.

If the segment was connected using `SCIConnectSegment` the execution of `SCIDisconnectSegment` also generates an `SCI_CB_DISCONNECT` event directed to the application that created the segment (see `SCICreateSegment and sci_cb_local_segment_t`).

```
void
SCIDisconnectSegment( IN sci_remote_segment_t segment,
                      IN unsigned int flags,
                      OUT sci_error_t* error );
```

Parameters

`segment`	handle to the connected segment descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_BUSY`	the segment is currently mapped or in use

DMA

SCIDMAQueueState

Synopsis SCIDMAQueueState returns the state of a DMA queue (see sci_dma_queue_state_t). The call does not affect the state of the queue (see Figure 2.4).

```
sci_dma_queue_state_t
SCIDMAQueueState( IN sci_dma_queue_t dq );
```

Parameters dq handle to the DMA queue descriptor

Return value The function returns the state of the DMA queue.

Errors No error information is provided by this function.

DMA

SCIEnqueueDMATransfer

Synopsis `SCIEnqueueDMATransfer` adds the specification of a new transfer to a DMA queue. Either the source or the destination of the transfer must be a local segment. By default the transfer is from the local segment to the remote one; if the transfer is in the opposite direction, the flag `SCI_FLAG_DMA_READ` has to be specified.

As shown in Figure 2.4, this function can be called only if the queue is either in the IDLE or in the GATHER states, otherwise the operation is illegal and the error is detected. If the function is successful the final state is GATHER (see `sci_dma_queue_state_t`).

The local adapter used by the local and the remote segments must be the same than the one specified when the queue was created (see `SCICreateDMAQueue`).

```
void
SCIEnqueueDMATransfer( IN sci_dma_queue_t dq,
                       IN sci_local_segment_t localSegment,
                       IN sci_remote_segment_t remoteSegment,
                       IN unsigned int localOffset,
                       IN unsigned int remoteOffset,
                       IN unsigned int size,
                       IN unsigned int flags,
                       OUT sci_error_t* error );
```

Parameters

`dq`	handle to the DMA queue descriptor
`localSegment`	handle to the local segment descriptor
`remoteSegment`	handle to the remote segment descriptor
`localOffset`	base address inside the local segment where data reside (or where data are transferred to, if the transfer is from the remote segment to the local one)
`remoteOffset`	base address inside the remote segment where data are transferred to (or where data reside, if the transfer is from the remote segment to the local one)
`size`	size of the data to be transferred
`flags`	see below
`error`	error information

Flags `SCI_FLAG_DMA_READ` the data transfer is from the remote segment to the local one

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_OUT_OF_RANGE` for one of the two segments, the sum of `offset` and `size` is larger than the segment size or larger than the

maximum DMA size

`SCI_ERR_MAX_ENTRIES` the DMA queue is full

`SCI_ERR_ILLEGAL_OPERATION` the operation is illegal in the current state of the queue

`SCI_ERR_SIZE_ALIGNMENT` for a segment the size is not correctly aligned as required by the implementation

`SCI_ERR_OFFSET_ALIGNMENT` for a segment the offset is not correctly aligned as required by the implementation

`SCI_ERR_SEGMENT_NOT_PREPARED` the local segment has not been prepared for access from the adapter associated with the queue

`SCI_ERR_SEGMENT_NOT_CONNECTED` he remote segment is not connected through the adapter associated with the queue

Shared Memory

SCIFlushReadBuffers

Synopsis `SCIFlushReadBuffers` flushes the prefetch buffers associated with a sequence.

```
void
SCIFlushReadBuffers( IN sci_sequence_t sequence );
```

Parameters `sequence` handle to the sequence descriptor.

Errors No information error is provided by the function.

Privileged

SCIGetCSRRegister

Synopsis `SCIGetCSRRegister` reads the value contained in a location of the CSR space of an SCI node. A node is specified using its physical node identifier. The location is determined by an offset from the base address of the CSR space.

```
unsigned int
SCIGetCSRRegister( IN sci_desc_t sd,
                   IN unsigned int localAdapterNo,
                   IN unsigned int SCINodeId,
                   IN unsigned int CSROffset,
                   IN unsigned int flags,
                   OUT sci_error_t *error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`localAdapterNo`	number of the local adapter used for the communication
`SCINodeId`	physical identifier of the remote node where the CSR space resides
`CSROffset`	offset in the CSR space
`flags`	not used
`error`	error information

Return value If successful, the function returns the value read from the specified CSR location.

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_NO_LINK_ACCESS`	it was not possible to communicate via the local adapter
`SCI_ERR_NO_REMOTE_LINK_ACCESS`	it was not possible to communicate via a remote switch port

Shared Memory

SCIGetRemoteSegmentSize

Synopsis `SCIGetRemoteSegmentSize` returns the size in bytes of a remote segment after it has been connected with `SCIConnectSegment` or `SCIConnectSCISpace`.

```
unsigned int
SCIGetRemoteSegmentSize( IN sci_remote_segment_t segment );
```

Parameters

`segment`	handle to the remote segment descriptor

Return value The function returns the size in bytes of the remote segment.

Errors No information error is provided by the function.

Shared Memory

SCIMapLocalSegment

Synopsis `SCIMapLocalSegment` maps an area of a memory segment created with `SCICreateSegment` into the addressable space of the program and returns a pointer to the beginning of the mapped area. The function also creates and initializes a descriptor for the mapped segment.

If a virtual address is suggested, together with the flag `SCI_FLAG_FIXED_MAP_ADDR`, the function tries first to map the segment at that address.

If the flag `SCI_FLAG_READONLY_MAP` is specified, the local segment is mapped in read-only more.

```
void*
SCIMapLocalSegment( IN sci_local_segment_t segment,
                    OUT sci_map_t* map,
                    IN unsigned int offset,
                    IN unsigned int size,
                    IN void* addr,
                    IN unsigned int flags,
                    OUT sci_error_t* error );
```

Parameters

`segment`	handle to the descriptor of the local segment to be mapped
`map`	handle to the new mapped segment descriptor
`offset`	offset inside the local segment where the mapping should start
`size`	size of the area of the local segment to be mapped, starting from `offset`
`addr`	suggested virtual address where the segment should be mapped
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_FIXED_MAP_ADDR` the function should try first to map at the suggested virtual address

`SCI_FLAG_READONLY_MAP` the segment is mapped in read-only mode

Return value If successful, the function returns a pointer to the beginning of the mapped area. In case of error it returns 0.

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_OUT_OF_RANGE` the sum of the offset and the size is larger than the

segment size

`SCI_ERR_SIZE_ALIGNMENT` the size is not correctly aligned as required by the implementation

`SCI_ERR_OFFSET_ALIGNMENT` the offset is not correctly aligned as required by the implementation

Shared Memory

SCIMapRemoteSegment

Synopsis `SCIMapRemoteSegment` maps an area of a remote segment connected with either `SCIConnectSegment` or `SCIConnectSCISpace` into the addressable space of the program and returns a pointer to the beginning of the mapped area. The function also creates and initializes a descriptor for the mapped segment.

If a virtual address is suggested, together with the flag `SCI_FLAG_FIXED_MAP_ADDR`, the function tries first to map the segment at that address.

If the flag `SCI_FLAG_READONLY_MAP` is specified, the remote segment is mapped in read-only more.

```
volatile void*
SCIMapRemoteSegment( IN sci_remote_segment_t segment,
                     OUT sci_map_t* map,
                     IN unsigned int offset,
                     IN unsigned int size,
                     IN void* addr,
                     IN unsigned int flags,
                     OUT sci_error_t* error );
```

Parameters

`segment`	handle to the descriptor of the remote segment to be mapped
`map`	handle to the new mapped segment descriptor
`offset`	offset inside the remote segment where the mapping should start
`size`	size of the area of the remote segment to be mapped, starting from `offset`
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_FIXED_MAP_ADDR`	the function should try first to map at the suggested virtual address
`SCI_FLAG_READONLY_MAP`	the segment is mapped in read-only mode

Return value If successful, the function returns a pointer to the beginning of the mapped area. In case of error it returns 0.

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_OUT_OF_RANGE` the sum of the offset and the size is larger than the

segment size

`SCI_ERR_SIZE_ALIGNMENT` the size is not correctly aligned as required by the implementation

`SCI_ERR_OFFSET_ALIGNMENT` the offset is not correctly aligned as required by the implementation

Shared Memory

SCIMemCopy

Synopsis `SCIMemCopy` transfers efficiently a block of data from local memory to a mapped segment using the shared memory mode.

If the flag `SCI_FLAG_ERROR_CHECK` is specified the function also checks if errors have occurred during the data transfer.

If the flag `SCI_FLAG_BLOCK_READ` is specified, the transfer is from the mapped segment to the local memory.

```
void
SCIMemCopy( IN void* memAddr,
            IN sci_map_t remoteMap,
            IN unsigned int remoteOffset,
            IN unsigned int size,
            IN unsigned int flags,
            OUT sci_error_t* error );
```

Parameters

`memAddr`	base address in virtual memory of the source
`remoteMap`	handle to the descriptor of the mapped segment that is the destination of the transfer
`remoteOffset`	offset inside the mapped segment where the transfer should start
`size`	size of the transfer
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_ERROR_CHECK`	perform error checking
`SCI_FLAG_BLOCK_READ`	the data transfer is from the remote segment to the local memory

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_OUT_OF_RANGE`	the sum of the offset and the size is larger than the mapped segment size
`SCI_ERR_SIZE_ALIGNMENT`	the size is not correctly aligned as required by the implementation
`SCI_ERR_OFFSET_ALIGNMENT`	the offset is not correctly aligned as required by the implementation
`SCI_ERR_TRANSFER_FAILED`	the error checking, if performed, has detected an error in the data transfer

General

SCIOpen

Synopsis `SCIOpen` opens an SCI virtual device, that is a channel to the driver. It creates and initializes a new descriptor for an SCI virtual device, to be used in subsequent calls to API functions. The same virtual device can be used to talk to different remote nodes.

If the flag `SCI_FLAG_THREAD_SAFE` is specified, operations on resources depending on the virtual device are executed in a thread-safe manner.

```
void
SCIOpen( OUT sci_desc_t* sd,
         IN unsigned int flags,
         OUT sci_error_t* error );
```

Parameters

`sd`	handle to the new SCI virtual device descriptor
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_THREAD_SAFE`	operations on resources associated with this descriptor will be performed in a thread-safe manner

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

DMA

SCIPostDMAQueue

Synopsis `SCIPostDMAQueue` starts the execution of a DMA queue. The function returns as soon as the transfer specifications contained in the queue are passed to the DMA engine of the SCI adapter.

If a callback function is specified and explicitly activated using the flag `SCI_FLAG_USE_CALLBACK`, it is asynchronously invoked when all the transfers have completed or if an error occurs during a transfer. Alternatively, an application can block waiting for the queue completion calling `SCIWaitForDMAQueue`.

As shown in Figure 2.4, `SCIPostDMAQueue` can be called only if the queue is in the GATHER state, otherwise the operation is illegal and the error is detected. If the function is successful the final state is POSTED (see `sci_dma_queue_state_t`).

```
void
SCIPostDMAQueue(  IN sci_dma_queue_t dq,
                  IN sci_cb_dma_t callback,
                  IN void* callbackArg,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

`dq`	handle to the DMA queue descriptor
`callback`	callback function to be invoked when all the DMA transfers have completed or in case an error occurs during a transfer
`callbackArg`	user-defined parameter passed to the callback function
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_USE_CALLBACK` the end of the transfer or an error will cause the callback function to be invoked

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_OPERATION` the operation is illegal in the current state of the queue

Shared Memory

SCIPrepareSegment

Synopsis `SCIPrepareSegment` guarantees that a local segment is accessible by an SCI adapter. According to the state diagram shown in Figure 2.2 a local segment can be made available (see `SCISetSegmentAvailable`) only after it has been prepared.

```
void
SCIPrepareSegment( IN sci_local_segment_t segment,
                   IN unsigned int localAdapterNo,
                   IN unsigned int flags,
                   OUT sci_error_t* error );
```

Parameters

`segment`	handle to the local segment descriptor
`localAdapterNo`	number of the adapter for which the segment is prepared
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

General

SCIProbeNode

Synopsis `SCIProbeNode` checks if a remote node is reachable.

```
int
SCIProbeNode( IN sci_desc_t sd,
              IN unsigned int localAdapterNo,
              IN unsigned int nodeId,
              IN unsigned int flags,
              OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`localAdapterNo`	number of the local adapter used for the check
`nodeId`	identifier of the remote node
`flags`	not used
`error`	error information

Return value The function returns 1 if the remote node can be reached, 0 otherwise.

Errors If the function returns 1, the error value points to SCI_ERR_OK; otherwise it points to one of the following:

`SCI_ERR_NO_LINK_ACCESS` It was not possible to communicate via the local adapter

`SCI_ERR_NO_REMOTE_LINK_ACCESS` It was not possible to communicate via a remote switch port

General

SCIQuery

Synopsis

SCIQuery provides some information about the underlying SCI system. The information can be vendor dependent, but some requests are specified in the API and shall be satisfied: the vendor identifier, the version of the API implemented, some adapter characteristics. Each request defines its own data structure to be used as input and output to SCIQuery. The memory management (allocation and deallocation) of the data structures has to be performed by the caller.

```
void
SCIQuery( IN unsigned int command,
          INOUT void* data,
          IN unsigned int flags,
          OUT sci_error_t* error );
```

Parameters

command	type of information required
data	generic data structure for possible sub-commands and output information (see below)
flags	not used
error	error information

Commands

The three major commands are:

SCI_Q_VENDORID	the vendor identifier is returned in a data structure of type sci_query_string
SCI_Q_API	the version of the API implemented is returned in a data structure of type sci_query_string
SCI_Q_ADAPTER	certain adapter information, depending on the sub-command (see below), is returned in a data structure of type sci_query_adapter

Data structures

There are two predefined data structures used to provide further sub-commands to the query function and to get back the requested information.

sci_query_string includes an allocated string of characters and its length. The string is filled by SCIQuery with the requested information.

```
struct sci_query_string {
    char* str;
    unsigned int length;
};
```

`sci_query_adapter` includes the number of the local adapter to interrogate, a sub-command specifying which characteristic is wanted and a pointer to an allocated data container.

```
struct sci_query_adapter {
    unsigned int localAdapterNo;
    unsigned int subcommand;
    void* data;
};
```

The sub-commands for adapter-specific information are:

`SCI_Q_ADAPTER_NODEID` return an `unsigned int` with the node identifier associated with the adapter

`SCI_Q_ADAPTER_DMA_SIZE_ALIGNMENT` return an `unsigned int` with the alignment requirement for a segment size

`SCI_Q_ADAPTER_DMA_OFFSET_ALIGNMENT` return an `unsigned int` with the alignment requirement for a segment offset

`SCI_Q_ADAPTER_DMA_MTU` return an `unsigned int` with the maximum unit size in bytes for a DMA transfer

`SCI_Q_ADAPTER_SUGGESTED_MIN_DMA_SIZE` return an `unsigned int` with the suggested minimum size for a DMA transfer

`SCI_Q_ADAPTER_SUGGESTED_MIN_BLOCK_SIZE` return an `unsigned int` with the suggested minimum size for a block transfer

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_QUERY` the command is not recognized

Shared Memory

SCIRegisterSegmentMemory

Synopsis `SCIRegisterSegmentMemory` associates an area memory allocated by the program (e.g. using `malloc`) with a local segment created passing the flag SCI_FLAG_EMPTY to `SCICreateSegment`. The memory area is identified by its base address in virtual address space and its size.

It is illegal to use the same local segment to register different memory areas.

The function can try to determine if the specified address is legal or not, but this highly depends on the underlying platform.

```
void
SCIRegisterSegmentMemory( IN void* address,
                          IN unsigned int size,
                          IN sci_local_segment_t segment,
                          IN unsigned int flags,
                          OUT sci_error_t* error );
```

Parameters

`address`	base address of the user-allocated memory in the program's virtual address space
`size`	size of the user-allocated memory to be associated with the local segment
`segment`	handle to the local segment descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_SIZE_ALIGNMENT`	the size is not correctly aligned as required by the implementation
`SCI_ERR_ILLEGAL_ADDRESS`	the specified virtual address is not legal
`SCI_ERR_OUT_OF_RANGE`	the size is larger than the maximum size of the local segment

DMA

SCIRemoveDMAQueue

Synopsis `SCIRemoveDMAQueue` frees the resources allocated for a DMA queue and destroys the corresponding descriptor. After this call the handle to the DMA queue descriptor becomes invalid and should not be used.

As shown in the state diagram in Figure 2.4, this function can be called only if the queue is either in the initial (IDLE) or in a final (DONE, ERROR or ABORTED) state, otherwise the operation is illegal and the error is detected (see `sci_dma_queue_state_t`).

```
void
SCIRemoveDMAQueue( IN sci_dma_queue_t dq,
                   IN unsigned int flags,
                   OUT sci_error_t* error );
```

Parameters

`dq`	handle to the DMA queue descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_OPERATION`	the operation is illegal in the current state of the queue

Interrupts

SCIRemoveInterrupt

Synopsis `SCIRemoveInterrupt` deallocates an interrupt resource and destroys the corresponding descriptor. After this call the handle to the descriptor becomes invalid and should not be used.

```
void
SCIRemoveInterrupt( IN sci_local_interrupt_t interrupt,
                    IN unsigned int flags,
                    OUT sci_error_t* error );
```

Parameters

`interrupt`	handle to the local interrupt descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

Shared Memory

SCIRemoveSegment

Synopsis `SCIRemoveSegment` frees the resources used by a local segment. The physical memory is deallocated only if it was allocated when the segment was created with `SCICreateSegment`. The function also destroys the descriptor associated with the local segment; after this call the handle to the descriptor becomes invalid and should not be used.

`SCIRemoveSegment` fails if other resources, either locally or remotely, depend on it (see Figure 2.1). Before calling this function, the program should consider the use of `SCISetSegmentUnavailable` with the flags NOTIFY or FORCE_DISCONNECT.

```
void
SCIRemoveSegment( IN sci_local_segment_t segment,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

`segment`	handle to the local segment descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_BUSY`	the segment is currently mapped or in use

Shared Memory

SCIRemoveSequence

Synopsis `SCIRemoveSequence` destroys a sequence descriptor. After this call the handle to the descriptor becomes invalid and should not be used.

```
void
SCIRemoveSequence( IN sci_sequence_t sequence,
                   IN unsigned int flags,
                   OUT sci_error_t* error );
```

Parameters

`sequence`	handle to the sequence descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

DMA

SCIResetDMAQueue

Synopsis `SCIResetDMAQueue` resets a DMA queue (without removing it), so it can be reused for another chain of transfers.

According to the state diagram in Figure 2.4, this function can be called when the queue is in any state other than the POSTED state. If the function is successful the final state is IDLE (see `sci_dma_queue_state_t`).

```
void
SCIResetDMAQueue( IN sci_dma_queue_t dq,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

`dq`	handle to the DMA queue descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_OPERATION`	The state transition implied by this operation is not allowed in the current state of the queue.

Privileged

SCISetCSRRegister

Synopsis `SCISetCSRRegister` writes a value to a location of the CSR space of an SCI node. A node is specified using its physical node identifier. The location is determined by an offset from the base address of the CSR space.

```
void
SCISetCSRRegister( IN sci_desc_t sd,
                   IN unsigned int localAdapterNo,
                   IN unsigned int SCINodeId,
                   IN unsigned int CSROffset,
                   IN unsigned int CSRValue,
                   IN unsigned int flags,
                   OUT sci_error_t* error );
```

Parameters

`sd`	handle to an open SCI virtual device descriptor
`localAdapterNo`	number of the local adapter used for the communication
`SCINodeId`	physical identifier of the remote node where the CSR space resides
`CSROffset`	location offset in the CSR space
`CSRValue`	value to write in the location
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_NO_LINK_ACCESS`	it was not possible to communicate via the local adapter
`SCI_ERR_NO_REMOTE_LINK_ACCESS`	it was not possible to communicate via a remote switch port

Shared Memory

SCISetSegmentAvailable

Synopsis SCISetSegmentAvailable makes a local segment visible to remote nodes, that can then connect to it. According to the state diagram shown in Figure 2.2 a local segment can be made available only after it has been prepared (see SCIPrepareSegment).

```
void
SCISetSegmentAvailable( IN sci_local_segment_t segment,
                        IN unsigned int localAdapterNo,
                        IN unsigned int flags,
                        OUT sci_error_t* error );
```

Parameters

segment	handle to the local segment descriptor
localAdapterNo	number of the local adapter where the local segment is made available for connections
flags	not used
error	error information

Errors On successful completion, error points to the SCI_ERR_OK value; otherwise to one of the following:

SCI_ERR_SEGMENT_NOT_PREPARED the segment has not yet been prepared to be accessed by this adapter

SCI_ERR_ILLEGAL_OPERATION the segment was created with the flag SCI_FLAG_PRIVATE and therefore has no segment identifier

Shared Memory

SCISetSegmentUnavailable

Synopsis `SCISetSegmentUnavailable` hides an available segment to remote nodes; no new connections will be accepted on that segment.

If the flag `SCI_FLAG_NOTIFY` is specified, the operation is notified to the remote nodes connected to the local segment. The notification should be interpreted as an invitation to disconnect. If the flag `SCI_FLAG_FORCE_DISCONNECT` is specified, the remote nodes are forced to disconnect. These two flags can be used to implement a smooth removal of a local segment (see `SCIRemoveSegment`).

```
void
SCISetSegmentUnavailable( IN sci_local_segment_t segment,
                          IN unsigned int localAdapterNo,
                          IN unsigned int flags,
                          OUT sci_error_t* error );
```

Parameters

`segment`	handle to the local segment descriptor
`localAdapterNo`	number of the local adapter where the local segment was made available
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_NOTIFY`	the connected nodes receive a notification of the operation
`SCI_FLAG_FORCE_DISCONNECT`	the connected nodes are forced to disconnect

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_OPERATION`	the operation is illegal in the current state of the segment

Shared Memory

SCIStartSequence

Synopsis SCIStartSequence performs the preliminary check of the error flags on the SCI adapter before starting a sequence of read and write operations on the concerned mapped segment. Subsequent checks are done calling SCICheckSequence, as far as no errors occur, in which case SCIStartSequence shall be called again until it returns SCI_SEQ_OK.

If the return value is SCI_SEQ_PENDING there is a pending error and the program is required to call SCIStartSequence until it succeeds, before doing other transfer operations on the segment.

```
sci_sequence_status_t
SCIStartSequence( IN sci_sequence_t sequence,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

sequence	handle to a sequence descriptor
flags	not used
error	error information

Error value The function returns the status of the sequence.

Errors If the return value is SCI_SEQ_OK, error points to the SCI_ERR_OK value. There are no specific error codes for this function.

Shared Memory

SCIStoreBarrier

Synopsis `SCIStoreBarrier` synchronizes all the accesses to a mapped segment. When the function returns the write buffers have been flushed and all outstanding SCI transactions related to the mapped segment have completed.

```
void
SCIStoreBarrier( IN sci_sequence_t sequence,
                 IN unsigned int   flags,
                 OUT sci_error_t* error );
```

Parameters

`map`	handle to the mapped segment descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error code for this function.

Block Operations

SCITransferBlock

Synopsis `SCITransferBlock` copies a block of data between two mapped segments. The function returns only when the transfer has finished.

```
void
SCITransferBlock( IN sci_map_t sourceMap,
                  IN unsigned int sourceOffset,
                  IN sci_map_t destinationMap,
                  IN unsigned int destinationOffset,
                  IN unsigned int size,
                  IN unsigned int flags,
                  OUT sci_error_t* error );
```

Parameters

`source`	handle to the mapped segment descriptor representing the transfer source
`sourceOffset`	offset inside the source segment, where the transfer starts
`destination`	handle to the mapped segment descriptor representing the transfer destination
`destinationOffset`	offset inside the destination segment, where the transfer starts
`size`	size of the data to be transferred
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_OUT_OF_RANGE`	the sum of the offset and the size is larger than one of the sizes of the mapped segments
`SCI_ERR_SIZE_ALIGNMENT`	the size is not correctly aligned as required by the implementation
`SCI_ERR_OFFSET_ALIGNMENT`	the offset is not correctly aligned as required by the implementation
`SCI_ERR_TRANSFER_FAILED`	the error checking, if performed, has detected an error in the data transfer

Block Operations

SCITransferBlockAsync

Synopsis `SCITransferBlockAsync` copies a block of data between two mapped segments. The function posts the transfer and returns immediately. A block transfer descriptor is created and initialized; its lifetime is the lifetime of the transfer and it is automatically destroyed when the transfer has completed.

A callback function can be specified to be invoked when the transfer completes or if an error occurs; the intention to use the callback has to be explicitly declared with the flag `SCI_FLAG_USE_CALLBACK`. Alternatively, the program can block and wait for the completion calling `SCIWaitForBlockTransfer`.

```
void
SCITransferBlockAsync( IN sci_map_t sourceMap,
                       IN unsigned int sourceOffset,
                       IN sci_map_t destinationMap,
                       IN unsigned int destinationOffset,
                       IN unsigned int size,
                       OUT sci_block_transfer_t* block,
                       IN sci_cb_block_transfer_t callback,
                       IN void* callbackArg,
                       IN unsigned int flags,
                       OUT sci_error_t* error );
```

Parameters

`source`	handle to the mapped segment descriptor representing the transfer source
`sourceOffset`	offset inside the source segment, where the transfer starts
`destination`	handle to the mapped segment descriptor representing the transfer destination
`destinationOffset`	offset inside the destination segment, where the transfer starts
`size`	size of the data to be transferred
`callback`	callback function invoked when the transfer has completed or if there is a failure during the transfer
`callbackArg`	user-defined argument passed to the callback function
`flags`	see below
`error`	error information

Flags

`SCI_FLAG_USE_CALLBACK`	the specified callback will be invoked when the transfer has completed

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_OUT_OF_RANGE` the sum of the offset and the size is larger than one of

the sizes of the mapped segments

`SCI_ERR_SIZE_ALIGNMENT` the size is not correctly aligned as required by the implementation

`SCI_ERR_OFFSET_ALIGNMENT` the offset is not correctly aligned as required by the implementation

`CI_ERR_TRANSFER_FAILED` the error checking, if performed, has detected an error in the data transfer

Interrupts

SCITriggerInterrupt

Synopsis `SCITriggerInterrupt` triggers an interrupt on a remote node, after having connected to it with `SCIConnectInterrupt`. What happens to the remote application that made the interrupt resource available depends on what it specified at the time it called `SCICreateInterrupt`.

```
void
SCITriggerInterrupt( IN sci_remote_interrupt_t interrupt,
                     IN unsigned int flags,
                     OUT sci_error_t* error );
```

Parameters

`interrupt`	handle to the remote interrupt descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value. There are no specific error codes for this function.

Shared Memory

SCIUnmapSegment

Synopsis `SCIUnmapSegment` unmaps from the program's address space a segment that was mapped either with `SCIMapLocalSegment` or with `SCIMapRemoteSegment`. It also destroys the corresponding descriptor, therefore after this call the handle to the descriptor becomes invalid and should not be used.

```
void
SCIUnmapSegment( IN sci_map_t map,
                 IN unsigned int flags,
                 OUT sci_error_t* error );
```

Parameters

`map`	handle to the mapped segment descriptor
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_BUSY`	the mapped segment is currently in use

Block Operations

SCIWaitForBlockTransfer

Synopsis `SCIWaitForBlockTransfer` suspends the program waiting for an asynchronous block transfer to complete. It is illegal to use this function if a callback is active on the same transfer.

```
void
SCIWaitForBlockTransfer( IN sci_block_transfer_t block,
                         IN unsigned int timeout,
                         IN unsigned int flags,
                         OUT sci_error_t* error );
```

Parameters

`block`	handle to the block transfer descriptor
`timeout`	time in millisecond to wait before giving up
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_ILLEGAL_OPERATION`	the handle is not valid any more, since the transfer has already completed
`SCI_ERR_TIMEOUT`	the timeout has expired

DMA

SCIWaitForDMAQueue

Synopsis SCIWaitForDMAQueue blocks a program until a DMA queue has finished (because of the completion of all the transfers or due to an error) or the timeout has expired. If timeout is SCI_INFINITE_TIMOUT the function blocks until a relevant event arrives. The function returns the current state of the queue.

According to the state diagram shown in Figure 2.4, calling this function is really meaningful only if the queue is in the POSTED state. If the state is in the ABORTED, DONE or ERROR states, the call is equivalent to a no-op. In all the other cases the call is illegal and the error is detected (see sci_dma_queue_state_t).

SCIWaitForDMAQueue cannot be used if a callback associated with the DMA queue is active.

```
sci_dma_queue_state_t
SCIWaitForDMAQueue( IN sci_dma_queue_t dq,
                    IN unsigned int timeout,
                    IN unsigned int flags,
                    OUT sci_error_t* error );
```

Parameters

dq	handle to a DMA queue descriptor
timeout	timeout in milliseconds to wait before giving up
flags	not used
error	error information

Return value On successful completion, the function returns the current state of the DMA queue. In case of error the returned value is undefined.

Errors On successful completion, error points to the SCI_ERR_OK value; otherwise to one of the following:

SCI_ERR_ILLEGAL_OPERATION	the operation is illegal in the current state of the DMA queue
SCI_ERR_TIMEOUT	the timeout has expired

Interrupts

SCIWaitForInterrupt

Synopsis `SCIWaitForInterrupt` blocks a program until an interrupt is received. If a timeout different from `SCI_INFINITE_TIMOUT` is specified the function gives up when the timeout expires.

`SCIWaitForInterrupt` cannot be used if a callback associated with the interrupt is active (see `SCICreateInterrupt`).

```
void
SCIWaitForInterrupt( IN sci_local_interrupt_t interrupt,
                     IN unsigned int timeout,
                     IN unsigned int flags,
                     OUT sci_error_t* error );
```

Parameters

`interrupt`	handle to the local interrupt descriptor
`timeout`	time in milliseconds to wait before giving up
`flags`	not used
`error`	error information

Errors On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_TIMEOUT`	the function timed out after the specified timeout value
`SCI_ERR_CANCELLED`	the interrupt has been removed

Shared Memory

SCIWaitForLocalSegmentEvent

Synopsis SCIWaitForLocalSegmentEvent blocks a program until an event concerning the local segment has arrived. If a timeout different from SCI_INFINITE_TIMOUT is specified the function gives up when the timeout expires.

SCIWaitForLocalSegmentEvent cannot be used if a callback associated with the local segment is active (see SCICreateSegment).

```
sci_segment_cb_reason_t
SCIWaitForLocalSegmentEvent( IN sci_local_segment_t segment,
                             OUT unsigned int* sourceNodeId,
                             OUT unsigned int* localAdapterNo,
                             IN unsigned int timeout,
                             IN unsigned int flags,
                             OUT sci_error_t* error );
```

Parameters

segment	handle to the local segment descriptor
sourceNodeId	identifier of the node that have generated the event
localAdapterNo	number of the local adapter that receive the event
timeout	time in milliseconds to wait before giving up
flags	not used
error	error information

Return value If successful, the function returns the reason corresponding to the received event.

Errors On successful completion, error points to the SCI_ERR_OK value; otherwise to one of the following:

SCI_ERR_TIMEOUT	the function timed out after the specified timeout value
SCI_ERR_CANCELLED	the segment has been removed. The handle is invalid when this error is returned

Shared Memory

SCIWaitForRemoteSegmentEvent

Synopsis

`SCIWaitForRemoteSegmentEvent` blocks a program until an event concerning the remote segment has arrived. If a timeout different from `SCI_INFINITE_TIMOUT` is specified the function gives up when the timeout expires.

`SCIWaitForRemoteSegmentEvent` cannot be used if a callback associated with the remote segment is active (see `SCIConnectSegment`).

```
sci_segment_cb_reason_t
SCIWaitForRemoteSegmentEvent( IN sci_remote_segment_t segment,
                              IN sci_error_t* status,
                              IN unsigned int timeout,
                              IN unsigned int flags,
                              OUT sci_error_t* error );
```

Parameters

`segment`	handle to the remote segment descriptor
`status`	status information
`timeout`	time in milliseconds to wait before giving up
`flags`	not used
`error`	error information

Return value

If successful, the function returns the reason that generated the received event.

Errors

On successful completion, `error` points to the `SCI_ERR_OK` value; otherwise to one of the following:

`SCI_ERR_TIMEOUT`	the function timed out after the specified timeout value
`SCI_ERR_CANCELLED`	the segment has been disconnected. The handle is invalid when this error is returned

Bibliography

1 **IEEE Standard for Scalable Coherent Interface (SCI)**
IEEE Computer Society; IEEE Std 1596-1992, August, 1993

2 **Low-level SCI software requirements, analysis and pre-design**
F. Giacomini et al.; Version 2.0; May 1998

3 **Physical layer Application Programming Interface for the Scalable Coherent Interface (SCI PHY-API)**
IEEE Computer Society; Proposed IEEE Standard 1596.9, Draft Version 0.51, July, 1997

4 **Virtual Interface Architecture Specification**
Compaq Computer Corp., Intel Corporation, Microsoft Corporation; Version 1.0; December 1997

5 **Control and Stats Registers (CSR) Architecture for microcomputer buses**
ANSI/IEEE Std 1212, 1994 Edition

6 **IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors**
IEEE Computer Society; IEEE Std 1596.5-1993, April, 1994