



SISCI API

User Guide

Dolphin Interconnect Solutions AS

Olaf Helsets vei 6
P.O.Box 70, Bogerud
N-0621 Oslo, Norway

Phone: +47 23 16 70 00
Telefax: +47 23 16 71 80
e-mail: sisci-support@dolphinics.no

Date: 5/14/01
Version: 1.0
Part: DI950-10357

SISCI API User Guide

by Dolphin Interconnect Solutions, AS

Copyright © 2001 by Dolphin Interconnect Solutions, AS

The goal of this guide is to introduce the reader to the SISCI API, which allows the development of applications exploiting the powerful capabilities offered by the SCI interconnect technology.

Legal Notices and Information

This manual may be reproduced in whole or in part, without fee, subject to the following restrictions:

- The copyright notice above and this permission notice must be preserved complete on all complete or partial copies.
- Any translation or derived work must be approved by the author in writing before redistribution.
- If you distribute this work in part, instructions for obtaining the complete version of this manual must be included, and a means for obtaining a complete version provided.
- Small portions may be reproduced as illustrations for reviews or quotes in other works without this permission notice if proper citation is given. Exceptions to these rules may be granted for academic purposes: Write to the author and ask. These restrictions are here to protect us as authors, not to restrict you as learners and educators. Any source code (aside from the DocBook this document was written in) in this document is placed under the GNU General Public License, available via anonymous FTP from the GNU archive.

Revision History

Revision 1.0 14 May 2001

First version

Table of Contents

1. Introduction	9
1.1. The SISI API	9
1.1.1. Resources and resource dependencies	9
1.2. About this guide	9
1.2.1. Examples.....	10
1.2.2. Feedback.....	11
2. System aspects	13
2.1. Initialising the SCI environment	13
2.2. Virtual devices.....	13
2.3. Querying information.....	14
2.3.1. Determining the API version	14
2.3.2. Determining the node identifier of a certain adapter	15
2.4. Probing a remote node.....	15
3. Memory segments	17
3.1. Managing a local segment.....	17
3.1.1. Allocating memory space	17
3.1.2. Deallocating a memory segment	18
3.1.3. Making a segment available to the rest of the SCI world	19
3.1.4. Making a segment unavailable to the rest of the SCI world	20
3.1.5. A state machine for a local segment resource	21
3.2. Managing a remote segment.....	21
3.2.1. Connecting to a remote segment.....	22
3.2.2. Disconnecting from a remote segment	23
4. Shared memory	25
4.1. Mapping a local memory segment into addressable space.....	25
4.2. Mapping a remote memory segment into addressable space	26
4.3. Unmapping a mapped memory segment.....	28
4.4. Accessing data within mapped memory segments.....	29
4.5. Example.....	30
5. DMA	35
5.1. DMA queues	35
5.2. Creating a DMA queue	37
5.3. Removing a DMA queue.....	38
5.4. Querying the state of a DMA queue.....	39
5.5. Appending data transfers to a DMA queue	39
5.6. Processing a DMA queue.....	41

5.7. Waiting for completion.....	41
5.8. Aborting a DMA queue processing.....	43
5.9. Resetting a DMA queue	43
5.10. Example.....	44
6. Interrupts.....	51
6.1. Managing a local interrupt	51
6.1.1. Allocating an interrupt.....	51
6.1.2. Deallocating an interrupt	52
6.1.3. Waiting for an interrupt.....	52
6.2. Managing a remote interrupt	53
6.2.1. Connecting to an interrupt	53
6.2.2. Disconnecting from an interrupt.....	54
6.2.3. Triggering an interrupt.....	54
6.3. Example.....	55
7. Advanced Features.....	59
7.1. Caching and error checking.....	59
7.1.1. Sequences	59
7.1.2. Flushing buffers	60
7.1.3. Checking for data transfer errors	61
7.2. Events and callbacks	66
7.2.1. Events and local memory segments.....	66
7.2.2. Events and remote memory segments	70
7.2.3. DMA and callbacks	76
7.2.4. Interrupts and callbacks	76

List of Figures

3-1. State machine for a local segment21
5-1. State diagram for a DMA queue.....36

List of Examples

4-1. A sender program based on shared memory30
4-2. A receiver program based on shared memory32
5-1. A sender program based on DMA44
5-2. A receiver program based on DMA.....47
6-1. A sender program based on interrupts.....55
6-2. A receiver program based on interrupts.....56

Chapter 1. Introduction

The Scalable Coherent Interface (SCI) is a high-performance interconnect technology that allows to build scalable multiprocessor systems in a variety of different topologies. The powerful functionality offered by the SCI hardware is made available to applications through some software layers which by design don't add significant overhead, in particular to data transfers operations. For an application developer the SISCO API represents the interface to SCI hardware and lower-level software services.

1.1. The SISCO API

The SISCO API was one of the main outcomes of the EU-funded Esprit Project 23174 "Standard Software Infrastructures for SCI-based Parallel Systems", whose purpose was to encourage the development of software support for parallel processing on clusters of PCs or workstations connected with a fast interconnect like SCI.

The SISCO API supports data transfers based either on distributed shared memory or on Direct Memory Access (DMA). It also allows to trigger remote interrupts and to catch and manage events generated by the underlying SCI system (such as a cable being unplugged).

1.1.1. Resources and resource dependencies

The SISCO API makes extensive use of the "resource" concept: a virtual device is a resource, a memory segment is a resource, a DMA queue is a resource, and so on. The list of available resources will become clear going through this guide.

A resource is usually associated with a number of properties, which are collected in a descriptor. The contents of a descriptor, i.e. the resource properties, are not directly visible to a user, who needs to use appropriate API functions to manage them. In other words, a descriptor is opaque to a user. To refer to a descriptor a handle is provided, which is what is passed to the API functions.

Names of descriptors and handles are chosen after the resource name. For a local segment, for instance, the descriptor is called `sci_local_segment` and the handle is called `sci_local_segment_t`.

Resources may depend on other resources. For example the function that creates a local segment needs in input a reference to an open virtual device, meaning that a local segment depends on a virtual device.

The existence of dependencies implies that a resource should not be freed if there is another one relying on it; doing so would generate an error. Using the example above, a virtual device cannot be closed until all the local segments associated to it are released.

1.2. About this guide

This document will guide you through the fundamental features of the SISI API. At the end you'll be able to manage memory segments, to transfer data from one node to another in several ways, and to interrupt remote processes.

We will start with addressing some generic aspects like initializing the SISI API library and querying information about the SCI system. This is presented in Chapter 2.

Then in Chapter 3 you'll learn basic memory management: how to allocate memory segments, how to make them available to other nodes, and how to connect to a remote memory segment.

When you have completed the basic memory management section you'll be ready to perform data transfers between several nodes. Data transfers are described in Chapter 4 and Chapter 5.

An SCI system may contain several nodes sharing a global memory structure. In order for the nodes to operate together interrupts may be used. Interrupts are a fast way of notifying another node that something has occurred and you'll learn how to use them in Chapter 6.

Finally Chapter 7 deals with things like managing events and checking for data transfer errors.

It is recommended to keep a copy of the SISI API Functional Specification at hand. In particular the specification will be useful as a reference for function prototypes and for the list of possible errors generated by a function call.

This guide, the SISI API Functional Specification, other documentation and software distributions are available at the Dolphin ICS WWW site <http://www.dolphinics.no/>.

1.2.1. Examples

The textual explanation is enriched by C code excerpts to show how things are used in practice; from time to time whole programs, implementing a send-receive example, are included in order to summarise the concepts explained so far (such programs are supposed to correctly compile and run). The choice of a send-receive pattern for the examples is motivated by its simplicity, of course the SCI hardware and software allow to do much more than that.

Of course the SCI hardware and software allow to do much more than simply

A program making use of the SISI API library must include the header file `sisci_api.h`. For simplicity this is done only in the full programs but not in code excerpts. Please refer to the release notes that come with the software distribution to find out where this header file is located and for additional information on how to compile and link SCI applications.

Even if they are not part of the API, many examples that you find both in this guide and in the software distribution make use of the following constants:

```
#define NO_FLAGS 0
#define NO_CALLBACK 0
#define NO_ARG 0
```

In this guide we also assume that the following identifiers are defined. Their values are not important but must be legal:

- `ADAPTER_NO` is the identifier of the SCI adapter card. Although in general a machine can have several adapters, each with a host-wide unique identifier, in this guide we assume that there is only one card per machine and that its identifier is the same on all the machines.
- `SENDER_NODE_ID` and `RECEIVER_NODE_ID` are the SCI node identifiers of the two nodes involved in the example code. Each node in an SCI system is assigned a unique identifier.
- `SENDER_MEM_ID` and `RECEIVER_MEM_ID` are the identifiers of the SCI memory segments created on the two nodes of the example code. The sizes (in bytes) of the two segments are `SENDER_MEM_SIZE` and `RECEIVER_MEM_SIZE` respectively.

1.2.2. Feedback

For this guide we have tried to choose a license that allows you to actively contribute to its improvement. You are encouraged to send us comments, modifications and additions to `<sis-ci-support@dolphinics.no>`. We will do our best to include them in the next version of the document.

Chapter 2. System aspects

In this chapter we address some generic aspects of the SISCO API which are related to the initialization of the SCI program environment and to the retrieval of information about the underlying SCI system. In particular you'll learn:

- how to initialise the SISCO API library ;
- how to manage the so-called virtual devices;
- how to get useful information about the underlying SCI system;
- how to check if a remote node is on-line.

2.1. Initialising the SCI environment

Before calling any other function in the SISCO API one should initialise the library calling `SCIInitialize()`:

```
sci_error_t error;
SCIInitialize(NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* successful initialization */
} else {
    /* manage error */
}
/* go on with the program */
```

If it fails there is something fundamentally wrong with the SCI installation, typically the SISCO library and driver versions are not consistent, in which case the error is `SCI_ERR_INCONSISTENT_VERSIONS`.

Before exiting, but after any SISCO API call, the program should call `SCITerminate()` to properly release all the SCI resources:

```
/* no SISCO API calls here */
SCIInitialize(...);
/* SCI program goes here */
SCITerminate();
/* no SISCO API calls here */
```

Both `SCIInitialize()` and `SCITerminate()` should be called only once in your program, independently of how many SCI resources you use.

2.2. Virtual devices

Before accessing any of the functionalities made available by the SISI API and introduced in the following chapters, a virtual device has to be created. A virtual device can be seen as a communication channel with the underlying SCI driver.

The same virtual device can be used to interact with different remote nodes.

A virtual device is created with `SCIOpen()` and removed with `SCIClose()`:

```
sci_desc_t v_dev;
sci_error_t error;

SCIInitialize(...);
SCIOpen(&v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) {
    /* manage error */
}
/* use v_dev */
SCIClose(v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) {
    /* manage error */
}
SCITerminate();
```

A typical error in case of failure of `SCIOpen()` is `SCI_ERR_NOT_INITIALIZED`, which means that you forgot to call `SCIInitialize()` before `SCIOpen()`.

2.3. Querying information

The SISI API provides a way to retrieve some information about the underlying SCI system, in particular about the vendor identifier, the version of the API implemented and some adapter characteristics. Other information, both general and vendor-dependent, can also be provided. Refer to the SISI API specification for further query options.

The function used to query the above mentioned information is `SCIQuery()`. This function basically accepts as input a query command and gives as output the required information. Additional input information, like subcommands, can be passed in the same data structure used for the output.

The following two sections show how to get the API version and the node identifier associated with a certain adapter. The procedure to access other information is similar, please refer to the SISI API specification for the details.

2.3.1. Determining the API version

The `SCIQuery()` call is in this case quite simple. The query command is `SCI_Q_API` and the output goes in an `sci_query_string` structure. This structure contains a pointer to an already allocated character array and an integer representing the size of the array.

```
const unsigned int QUERY_STRING_LENGTH = 64;
char api_version[QUERY_STRING_LENGTH];
sci_query_string_t query;
sci_error_t error;

query.str = api_version;
query.length = QUERY_STRING_LENGTH;
SCIQuery(SCI_Q_API, &query, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* api_version contains the wanted value */
} else {
    /* manage error */
}
```

2.3.2. Determining the node identifier of a certain adapter

In this case additional input data to `SCIQuery()` is necessary to get the right information for a certain adapter: a subcommand specifying the kind of wanted information and the adapter identifier. The additional information is passed through the `sci_query_adapter` structure which also contains an integer field for the output.

```
unsigned int local_node_id;
sci_query_adapter_t query;
sci_error_t error;

query.subcommand = SCI_Q_ADAPTER_NODE_ID;
query.localAdapterNo = ADAPTER_NO;
query.data = &local_node_id;
SCIQuery(SCI_Q_ADAPTER, &query, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* local_node_id contains the wanted value */
} else {
    /* manage error */
}
```

2.4. Probing a remote node

SCIProbeNode() can be used to check if communication is possible to a certain remote node through a certain local SCI adapter:

```
sci_desc_t v_dev;
sci_error_t error;
int reachable;

SCIInitialize(...);
SCIOpen(&v_dev, ...);
reachable = SCIProbeNode(v_dev, ADAPTER_NO, SENDER_NODE_ID,
                        NO_FLAGS, &error);

if (reachable == 1) {
    /* node is reachable */
} else { /* error != SCI_ERR_OK */
    /* manage the error */
}
```

If the function fails, typical errors are:

- SCI_ERR_NO_LINK_ACCESS: there are problems with the local adapter;
- SCI_ERR_NO_REMOTE_LINK_ACCESS: there are problems with a remote switch port;
- SCI_ERR_NODE_NOT_RESPONDING: there are problems with the remote adapter;
- SCI_ERR_NO_SUCH_NODEID: no node on the SCI network accessible through the specified local adapter has such identifier.

Chapter 3. Memory segments

The possibility to have access to memory physically resident on another machine is the fundamental characteristic and the strength of the SCI technology. If the remote piece of memory is also mapped in the addressable space of a local process, then appearing as if it were local, a data transfer is as simple as a `memcpy()`. In such a case it is the CPU which actively reads from or writes to remote memory, what is known as Programmed I/O (PIO). Alternately the local process can choose the Direct Memory Access (DMA) approach, whereby the CPU simply gives instructions to the SCI interface about the transfer (for example source address, destination address and size) and then it is free to do other things.

In both cases what is needed is a way to manage local memory segments on one side and a way to attach to remote memory segments on the other side.

At the end of this chapter you'll have learned:

- how to allocate a memory segment on the local node;
- how to make a local segment available to other nodes;
- how to connect to a memory segment available on a remote node.

3.1. Managing a local segment

3.1.1. Allocating memory space

Before a memory segment could be used it must of course exist. The allocation of a segment on the local host is done with the function `SCICreateSegment()`. The main reason to have a special function for the allocation instead of a normal `malloc()`-like call is that the driver must be aware of the created segment and associated parameters. Moreover most operating systems require that memory used in the way SCI uses it have specific characteristics, such as being non-swappable and/or being physically contiguous.

A typical usage of `SCICreateSegment()` is as follows, assuming we are on the sender node:

```
sci_desc_t v_dev;
sci_local_segment_t local_segment;
sci_error_t error;

SCIInitialize(...);
SCIOpen(&v_dev,...);
SCICreateSegment(v_dev, /* virtual device */
                &local_segment, /* handle to the allocated segment */
                RECEIVER_MEM_ID, /* segment identifier */
```

```

        RECEIVER_MEM_SIZE, /* size */
        NO_CALLBACK, /* ignore this for the moment */
        NO_ARG, /* callback arg, ignore */
        NO_FLAGS,
        &error);
if (error == SCI_ERR_OK) {
    /* a segment is available for use */
} else {
    /* manage error */
}

```

Let's look at the different parameters:

`v_dev` is the virtual device, as it comes from the `SCIOpen()` call shown previously.

`local_segment` is a handle to the memory segment to be allocated. It is initialised by the call, if successful.

The segment identifier, which in this case is the constant `RECEIVER_MEM_ID`, is an integer that uniquely identifies the segment to the driver. A remote node that wants to use this segment needs to know such an identifier. The uniqueness of the identifier is checked by the driver, so applications should choose appropriate values. In some special cases, in which the segment is meant for private use, i.e. it will not be made available to remote nodes, such identifier is useless so it need not be specified, but in general this is a required parameter.

`RECEIVER_MEM_SIZE` is the size of the segment to be allocated.

The specification of a callback allows to trigger the execution of a certain function when something happens concerning this segment. This option is covered later in Section 7.2.

As usual, `error` contains an error code which, if representing failure, gives a hint about the cause. Typical errors for such function are the non-uniqueness of the segment identifier and the unavailability of free space.

3.1.2. Deallocating a memory segment

Once a memory segment is not used any more, for example before quitting the application, it should be destroyed, in order to release unneeded resources.

The way to do it is via the function `SCIRemoveSegment()`:

```

sci_error_t error;
sci_local_segment_t segment;

SCICreateSegment(..., &segment, ...); /* initialisation */
/* use the segment */
SCIRemoveSegment(segment, NO_FLAGS, error);
if (error == SCI_ERR_OK) {

```

```

/* the resource is freed */
} else {
    /* manage error */
}

```

Normally `SCIRemoveSegment()` succeeds. If it doesn't and the application is at the end, it is probably fine anyway, since the driver should release the left-over resources. A typical cause of failure for this call is the dependency of other resources on this segment.

`SCIRemoveSegment()` succeeds even if there are remote processes still connected to the local segment. In such a case the segment is kept available only for the connected processes until they disconnect.

3.1.3. Making a segment available to the rest of the SCI world

Once a segment has been allocated it needs to be made visible to the other nodes connected to the SCI network to be really useful. This export operation is actually performed by calling two different functions in sequence.

The first one is `SCIPrepareSegment()`. Logically its goal is to map the segment into the 64-bit SCI address space, shared by all the nodes in the SCI network. In practice what it does is to make sure that the concerned segment could be correctly accessed by the specified SCI adapter. This includes guaranteeing the requirements possibly set by the operating system, like non-swappability and physical contiguity.

The second step is performed by `SCISetSegmentAvailable()`, which makes the segment visible to the other nodes via a certain adapter.

```

sci_error_t prepare_error;
sci_error_t avail_error;
sci_local_segment_t segment;

SCICreateSegment(..., &segment, ...); /* initialisation */
SCIPrepareSegment(segment,
                  ADAPTER_NO,
                  NO_FLAGS,
                  &prepare_error);
if (prepare_error == SCI_ERR_OK)
    SCISetSegmentAvailable(segment,
                           ADAPTER_NO,
                           NO_FLAGS,
                           &avail_error);
if (avail_error == SCI_ERR_OK) {
    /* segment is now available to the remote nodes */
} else {
    /* manage availability error */
}

```

```

    }
} else {
    /* manage preparation error */
}

```

As shown in the above piece of code, the preparation and the availability of a memory segment are per adapter. Then typical errors for the two above functions are mainly related to problems with the specified adapter (for example it does not exist or the one specified in `SCIPrepareSegment()` doesn't match with the one specified in `SCISetSegmentAvailable()`).

3.1.4. Making a segment unavailable to the rest of the SCI world

`SCISetSegmentUnavailable()` changes the visibility of an exported segment, in such a way that new remote connections are not possible through the specified adapter.

```

sci_local_segment segment;
unsigned int local_adapter = 0;
sci_error_t error;

SCICreateSegment(..., &segment, ...);
SCIPrepareSegment(segment, ...);
SCISetSegmentAvailable(segment, ...);
/* use the segment */
SCISetSegmentUnavailable(segment, ADAPTER_NO, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the segment is not available for new remote connections */
} else {
    /* manage error */
}

```

Calling `SCISetSegmentUnavailable()` doesn't affect existing remote connections, which are not even aware of the change. `SCISetSegmentAvailable()` could then be used, for example, to make a segment available only to a certain number of remote nodes. It would work like this:

```

SCISetSegmentAvailable(segment, ...);
/* wait for n remote nodes to connect */
SCISetSegmentUnavailable(segment, ...);

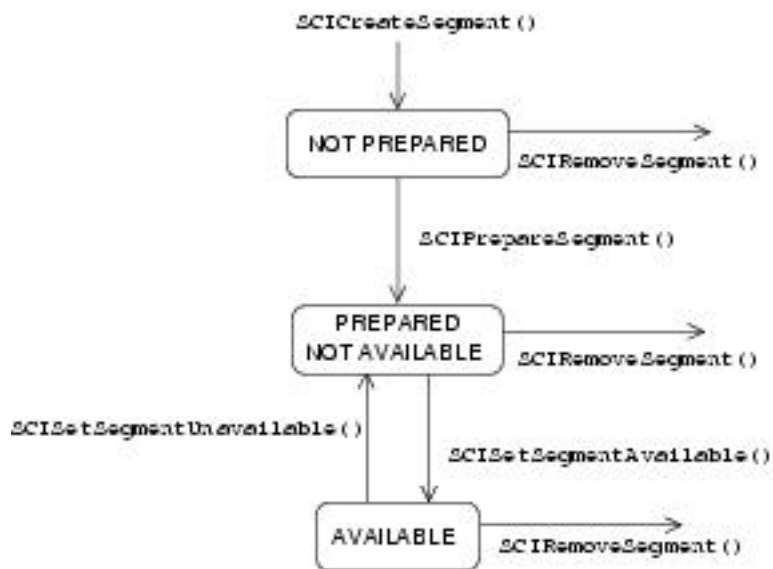
```

We will see later how it is possible to know, both synchronously and asynchronously, when a remote node has connected.

3.1.5. A state machine for a local segment resource

After having described the functions used to manage a local segment, it is possible to draw the state machine shown in Figure 3-1:

Figure 3-1. State machine for a local segment



A few comments are needed to fully understand the picture:

- NOT PREPARED means that the segment has been allocated successfully but that is not yet prepared to be used by an SCI adapter.
- SCIRemoveSegment() is a legal operation from any state. This does not mean of course that it will succeed (e.g. it won't if there is a dependency on it).
- It is not possible to "un-prepare" a segment.

3.2. Managing a remote segment

At this point the situation is as follows: a process on the receiver node, which has node id RECEIVER_NODE_ID, has

allocated and made available a piece of memory, with segment id equal to `RECEIVER_MEM_ID`.

Now we move onto the sender node, which has node id `SENDER_NODE_ID`. Here we would like to have a process which uses the memory segment on the exporting node.

Before looking at how to do this, it is worth clarifying some nomenclature, in order to better understand the next sections:

- from the receiver process' perspective what it has allocated is a local segment, which is represented by a local segment resource;
- from the sender process' perspective the memory segment allocated on the receiver node is a remote segment which is locally represented by a remote segment resource.

So a local segment and a local segment resource live on the same node, whereas a remote segment and a remote segment resource live on different ones.

3.2.1. Connecting to a remote segment

The first thing the remote application has to do in order to use a remote segment is to "connect" to it. Logically speaking the connection consists basically in finding the address and the size of the remote segment within the SCI address space.

This is achieved by calling the function `SCIConnectSegment()`:

```
sci_desc_t v_dev;
sci_remote_segment_t remote_segment;
sci_error_t error;

SCIInitialize(...);
SCIOpen(&v_dev, ...); /* initialise the virtual device */
SCIConnectSegment(v_dev, /* virtual device */
                 &remote_segment, /* handle to the remote segment resource*/
                 RECEIVER_NODE_ID, /* remote node id */
                 RECEIVER_MEM_ID, /* remote segment id */
                 ADAPTER_NO, /* local adapter number */
                 NO_CALLBACK, /* ignore this for the moment */
                 NO_ARG, /* callback arg, ignore */
                 SCI_INFINITE_TIMEOUT, /* timeout */
                 NO_FLAGS,
                 &error);
if (error == SCI_ERR_OK) {
    /* the remote segment is connected */
} else {
    /* manage error */
}
```

```
}

```

Let's look at the list of parameters of the above call:

`v_dev` represents as usual an initialised virtual device, needed to communicate with the driver.

`remote_segment` is a handle to an `sci_remote_segment` descriptor, which contains information about the connection. The descriptor is allocated and initialised by the call, if successful.

The remote node and segment identifiers (`RECEIVER_NODE_ID` and `RECEIVER_MEM_ID` respectively) uniquely locate a memory segment on the SCI network.

The adapter number `ADAPTER_NO` refers to the local adapter we want to use to access the remote segment.

The `SCIConnectSegment()` call is by default synchronous, it waits until either the connection request is resolved or the specified timeout, expressed in milliseconds, expires. If the timeout is infinite, as shown above, only the first option can cause the function to return.

Alternatively the call can be made asynchronous, but this behaviour is described in Section Section 7.2 .

Once the remote segment is connected, its size (expressed in bytes) can be determined calling the function `SCIGetRemoteSegmentSize()`:

```
sci_remote_segment_t remote_segment;
unsigned int remote_segment_size;
SCIConnectSegment(..., &remote_segment, ...);
remote_segment_size = SCIGetRemoteSegmentSize(remote_segment);

```

3.2.2. Disconnecting from a remote segment

Once an application has finished with a remote segment it should disconnect from it, releasing the remote segment resource.

```
sci_remote_segment_t segment;
sci_error_t error;

SCIConnectSegment(..., &segment, ...);
/* use the segment */
SCIDisconnectSegment(segment, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the remote segment resource is released */
} else {
    /* manage error */
}

```

A typical error for `SCIDisconnectSegment()` is `SCI_ERR_BUSY`, meaning that there are other resources depending on the remote segment resource.

Chapter 4. Shared memory

Once a memory segment is available, i.e. you have a valid handle to either a local or remote segment resource, you can access it in two ways: either you map it into the address space of your process and then you access it as you normally access memory, e.g. via pointer operations, or you can choose the so-called Direct Memory Access (DMA) approach, which consists in delegating the actual data transfer to the SCI adapter.

In this chapter you'll learn how to use memory-mapped segments, in particular:

- how to map a local memory segment into the addressable space of your program;
- how to map a remote memory segment into the addressable space of your program;
- how to access data within a mapped segment.

The use of DMA is instead address in Chapter 5.

4.1. Mapping a local memory segment into addressable space

Let's start with a local segment:

```
sci_desc_t v_dev;
sci_local_segment_t local_segment;
sci_map_t local_map;
sci_error_t error;
void* map_address;

SCIInitialize(...);
SCIOpen(&v_dev,...);
SCICreateSegment(v_dev,
                 &local_segment,
                 RECEIVER_MEM_ID,
                 RECEIVER_MEM_SIZE,
                 NO_CALLBACK,
                 0,
                 NO_FLAGS,
                 &error);
if (error == SCI_ERR_OK) {
    /* the segment has been successfully created */
    unsigned int offset = 0;
    unsigned int size = RECEIVER_MEM_SIZE;
```

```

void* suggested_address = 0;
map_address = SCIMapLocalSegment(local_segment,
                                &local_map,
                                offset,
                                size,
                                suggested_address,
                                NO_FLAGS,
                                &error);

if (error == SCI_ERR_OK) {
    /* the segment has been successfully mapped at virtual
    * address map_address */
} else {
    /* manage mapping error */
}
} else {
    /* manage allocation error */
}

```

In the above sample code we have mapped the just created segment from its beginning (`offset = 0`) and for its entire size (`size = RECEIVER_MEM_SIZE`). It is possible to map only part of the segment varying these two parameters, with the constraint that the sum of `size` and `offset` does not go beyond the end of the segment. If it happens the function returns the error `SCI_ERR_OUT_OF_RANGE`. `size` and `offset` must also satisfy some implementation-dependent alignment constraints, otherwise `error` is set to either `SCI_ERR_SIZE_ALIGNMENT` or `SCI_ERR_OFFSET_ALIGNMENT`. If you need these implementation-dependent values you can use `SCIQuery()` with command `SCI_Q_ADAPTER` and sub-commands `SCI_Q_ADAPTER_DMA_SIZE_ALIGNMENT` and `SCI_Q_ADAPTER_DMA_OFFSET_ALIGNMENT`.

`suggested_address` is a suggested virtual address where the segment should be mapped. Its value is taken into account only if the flag `SCI_FLAG_FIXED_MAP_ADDR` is set.

`SCIMapLocalSegment()` also accepts the flag `SCI_FLAG_READONLY_MAP`, which causes the segment to be mapped in read-only mode.

If the function call succeeds the returned value `map_address` is a pointer to the beginning of the mapped segment and `map` is a handle to a correctly initialised mapped segment descriptor.

4.2. Mapping a remote memory segment into addressable space

Mapping a remote memory segment into the addressable space of a program is very similar to the procedure followed for a local segment. The following code shows the usage of `SCIMapRemoteSegment()`:

```

sci_desc_t v_dev;
sci_remote_segment_t remote_segment;
sci_map_t remote_map;
sci_error_t error;
volatile void* map_address;

SCIInitialize(...);
SCIOpen(&v_dev, ...);
SCIConnectSegment(v_dev,
                  &remote_segment,
                  RECEIVER_NODE_ID,
                  RECEIVER_MEM_ID,
                  ADAPTER_NO,
                  NO_CALLBACK,
                  NO_ARG,
                  SCI_INFINITE_TIMEOUT,
                  NO_FLAGS,
                  &error);
if (error == SCI_ERR_OK) {
    /* the remote segment has been successfully connected */
    unsigned int offset = 0;
    unsigned int size = SCIGetRemoteSegmentSize(remote_segment);
    void* suggested_address = 0;
    map_address = SCIMapRemoteSegment(remote_segment,
                                      &remote_map,
                                      offset,
                                      size,
                                      suggested_address,
                                      NO_FLAGS,
                                      &error);

    if (error == SCI_ERR_OK) {
        /* the segment has been successfully mapped at virtual
         * address map_address */
    } else {
        /* manage mapping error */
    }
} else {
    /* manage connection error */
}

```

In the above sample code we have mapped the just connected segment from its beginning (`offset = 0`) and for its entire size (which has been determined using the function `SCIGetRemoteSegmentSize()` and should be equal to `RECEIVER_MEM_SIZE`). It is possible to map only part of the segment varying these two parameters, with the

constraint that the sum of `size` and `offset` does not go beyond the end of the segment. If it happens the function returns the error `SCI_ERR_OUT_OF_RANGE`. `size` and `offset` must also satisfy some implementation-dependent alignment constraints, otherwise `error` is set to either `SCI_ERR_SIZE_ALIGNMENT` or `SCI_ERR_OFFSET_ALIGNMENT`.

`suggested_address` is a suggested virtual address where the segment should be mapped. Its value is taken into account only if the flag `SCI_FLAG_FIXED_MAP_ADDR` is set.

`SCIMapRemoteSegment()` also accepts the flag `SCI_FLAG_READONLY_MAP`, which causes the segment to be mapped in read-only mode.

`remote_map` is a handle to the mapped segment descriptor and is initialised by the call, if successful. Note that the type of `remote_map` is `sci_map_t`, that is the same type of the handle to the local mapped segment descriptor.

If the function call succeeds the returned value `map_address` is a pointer to the beginning of the mapped segment. Note that the pointer is declared as `volatile` to prevent the compiler from doing wrong optimisations of the code.

4.3. Unmapping a mapped memory segment

When the mapped memory segment is not needed any more it has to be unmapped. Since the type of a local and a remote mapped segment is the same, there is a unique function, called `SCIUnmapSegment()`, to perform this operation.

For a local segment the sequence of operations from creation to removal, for what concerns local access, is then the following:

```
sci_local_segment_t local_segment;
sci_map_t local_map;
sci_error_t error;

SCICreateSegment(..., &local_segment, ...);
SCIMapLocalSegment(local_segment, &local_map, ...);
/* use the mapped segment */
SCIUnmapSegment(local_map, NO_FLAGS, &error)
if (error == SCI_ERR_OK) {
    /* the segment is not mapped any more */
} else {
    /* manage error */
}
SCIRemoveSegment(local_segment, ...);
```

For a remote segment:

```

sci_remote_segment_t remote_segment;
sci_map_t remote_map;
sci_error_t error;
void* addr;

SCIConnectSegment(..., &remote_segment, ...);
SCIMapRemoteSegment(remote_segment, &remote_map, ...);
/* use the mapped segment */
SCIUnmapSegment(remote_map, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the remote segment is not mapped any more */
} else {
    /* manage error */
}
SCIDisconnectSegment(remote_segment, ...);

```

Of course, for a remote segment to be connectable, it must have been properly exported on the other node.

The syntax of `SCIUnmapSegment()` is very simple: `local_map (remote_map)` is a handle to a mapped local (remote) segment descriptor; there are no special flags and, if the function fails, `error` typically has the value `SCI_ERR_BUSY`, meaning that you haven't correctly considered all the dependencies on the mapped segment.

4.4. Accessing data within mapped memory segments

Once a local or remote memory segment is mapped into the addressable space, a program sees that memory as any other piece of memory it has allocated using `malloc()` or similar functions and it can access it via the normal use of pointers: it can read from it, write to it, do a `memcpy()` and so on.

For a local segment:

```

sci_local_segment_t local_segment;
sci_map_t local_map;
int* l_addr; /* address to local segment */

SCICreateSegment(..., &local_segment, ...);
l_addr = (int*)SCIMapLocalSegment(local_segment, &local_map, ...);
*l_addr = 1; /* local write operation */
*l_addr; /* local read operation */

```

For a remote segment:

```

sci_remote_segment_t remote_segment;
sci_map_t remote_map;
volatile int* r_addr; /* address to remote segment */

SCIDeconnectSegment(..., &remote_segment, ...);
r_addr = (volatile int*)SCIMapRemoteSegment(remote_segment, &remote_map, ...);
*r_addr = 1; /* remote write operation */
*r_addr; /* remote read operation */

```

In particular remote operations allow to access transparently memory which is physically resident on another SCI node. Having transparent access to remote memory allows data transfers based on simple read or write operations. This is known as Programmed I/O, whereby the movement of data is actively done by the CPU.

4.5. Example

It's time to summarise all the things learned so far in a sort of “big picture”, in order to have a proper understanding of the available features that the SISI API offers for what concerns the use of shared memory. There are indeed other important aspects, for example how to check for data transfer errors, but we are already able to implement a simple “Hello, World!” application. This is actually composed of two programs: a sender sends a command to a receiver which then prints the string “Hello, World!”.

Example 4-1 shows the sender: it opens a virtual device, connects to a known remote segment, maps it into its own address space, writes the print command at the beginning of that piece of memory, which is automatically converted by the hardware into a remote write operation, and finally cleans everything up and exits.

Example 4-1. A sender program based on shared memory

```

/* sender program */

#include "sisci_api.h"

#define RECEIVER_NODE_ID 4
#define RECEIVER_MEM_ID 4
#define ADAPTER_NO 0
#define NO_CALLBACK 0
#define NO_ARG 0
#define NO_FLAGS 0
#define PRINT_COMMAND 1

int
main(int argc, char* argv[])

```

```

{
    sci_desc_t v_dev;
    sci_error_t error;
    sci_remote_segment_t remote_segment;
    unsigned int remote_segment_size;
    sci_map_t remote_map;
    volatile int* remote_address;

    /* initialize the SCI environment */
    SCIInitialize(NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* create a virtual device */
    SCIOpen(&v_dev, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* connect to the remote segment */
    SCIConnectSegment(v_dev, &remote_segment,
                     RECEIVER_NODE_ID, RECEIVER_MEM_ID,
                     ADAPTER_NO, NO_CALLBACK, NO_ARG,
                     SCI_INFINITE_TIMEOUT, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    remote_segment_size = SCIGetRemoteSegmentSize(remote_segment);

    /* map the remote segment */
    remote_address =
        (volatile int*)SCIMapRemoteSegment(remote_segment, &remote_map,
                                           0 /* offset */,
                                           remote_segment_size,
                                           0 /* address hint */,
                                           NO_FLAGS, &error);

    if (error != SCI_ERR_OK) return 1;

    /* send the print command */
    *remote_address = PRINT_COMMAND;

    /* cleanup */
    SCIUnmapSegment(remote_map, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    SCIDisconnectSegment(remote_segment, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    SCIClose(v_dev, NO_FLAGS, &error);
}

```

```

    if (error != SCI_ERR_OK) return 1;

    SCITerminate();

    return 0;
}

```

Example 4-2 shows the receiver: it opens a virtual device, allocates a local memory segment, maps it into its address space in such a way that it can access it, initialises the first word of that memory to something different than the print command, exports the segment into the SCI space, waits for the print command, which simply means checking the first word of the memory segment in a polling loop, prints a message and exits, after having cleaned up.

Note that the initialisation of the first word of the memory segment is done before the segment itself is made available to other nodes. This is done to avoid a race condition on who touches first that memory location.

Example 4-2. A receiver program based on shared memory

```

/* receiver program */

#include "sisci_api.h"
#include <stdio.h>

#define RECEIVER_MEM_ID 4
#define RECEIVER_MEM_SIZE 4096
#define ADAPTER_NO 0
#define NO_CALLBACK 0
#define NO_ARG 0
#define NO_FLAGS 0
#define PRINT_COMMAND 1

int
main(int argc, char* argv[])
{
    sci_desc_t v_dev;
    sci_error_t error;
    sci_local_segment_t local_segment;
    sci_map_t local_map;
    int* local_address;

    /* initialize the SCI environment */
    SCIInitialize(NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

```

```

/* create a virtual device */
SCIOpen(&v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* allocate a local segment */
SCICreateSegment(v_dev, &local_segment,
                 RECEIVER_MEM_ID, RECEIVER_MEM_SIZE,
                 NO_CALLBACK, NO_ARG, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* map the local segment */
local_address =
    (int*)SCIMapLocalSegment(local_segment, &local_map,
                             0 /* offset */, RECEIVER_MEM_SIZE,
                             0 /* address hint */, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* initialise the contents of the first word of
the to-be-shared segment */
*local_address = ~PRINT_COMMAND;

/* export local segment */
SCIPrepareSegment(local_segment, ADAPTER_NO, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCISetSegmentAvailable(local_segment, ADAPTER_NO, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* wait for the sender process to send the print command */
while (*l_addr != PRINT_COMMAND) ;

printf("Hello, World!");

/* cleanup */
SCISetSegmentUnavailable(segment, ADAPTER_NO, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIRemoveSegment(local_segment, NO_FLAGS, error);
if (error != SCI_ERR_OK) return 1;

SCIClose(v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCITerminate();

```

```
    return 0;  
}
```

Chapter 5. DMA

With DMA it's not the CPU which actively copies data from one node to another, instead the data transfer is mandated to the DMA engine available on the SCI adapter. This approach allows the CPU to do something more useful during the transfer, though latencies usually slightly increase because of the time required to setup the DMA engine. However, more data transfers can be joined and sent together to the SCI adapter in order to amortize the overhead.

In this chapter you'll learn:

- what a DMA queue is;
- how to manage DMA queues;
- how to transfer data with DMA.

5.1. DMA queues

A DMA queue is one of the resources specified in the SISI API and is the fundamental mechanism to access the DMA functionality provided by the API. Its type is `sci_dma_queue` and the handle to it is of type `sci_dma_queue_t`. The queue is the vehicle used to pass one or more specifications of data transfers to the DMA engine available on the SCI adapter.

The life of a DMA queue can be summarized with a state diagram, shown in Figure 5-1. A transition from one state to another is caused either by an API call or by an asynchronous event. Only the transitions specified in the state diagram are legal; if an API function is illegally called the `SCI_ERR_ILLEGAL_OPERATION` error is returned.

The meaning of the states is as follows:

IDLE

the queue has been successfully created and it is empty;

GATHER

the queue is being filled with data transfer requests;

POSTED

the queue has been passed to the DMA engine on the SCI adapter and it is being processed;

DONE

all the data transfers specified in the queue have been successfully completed;

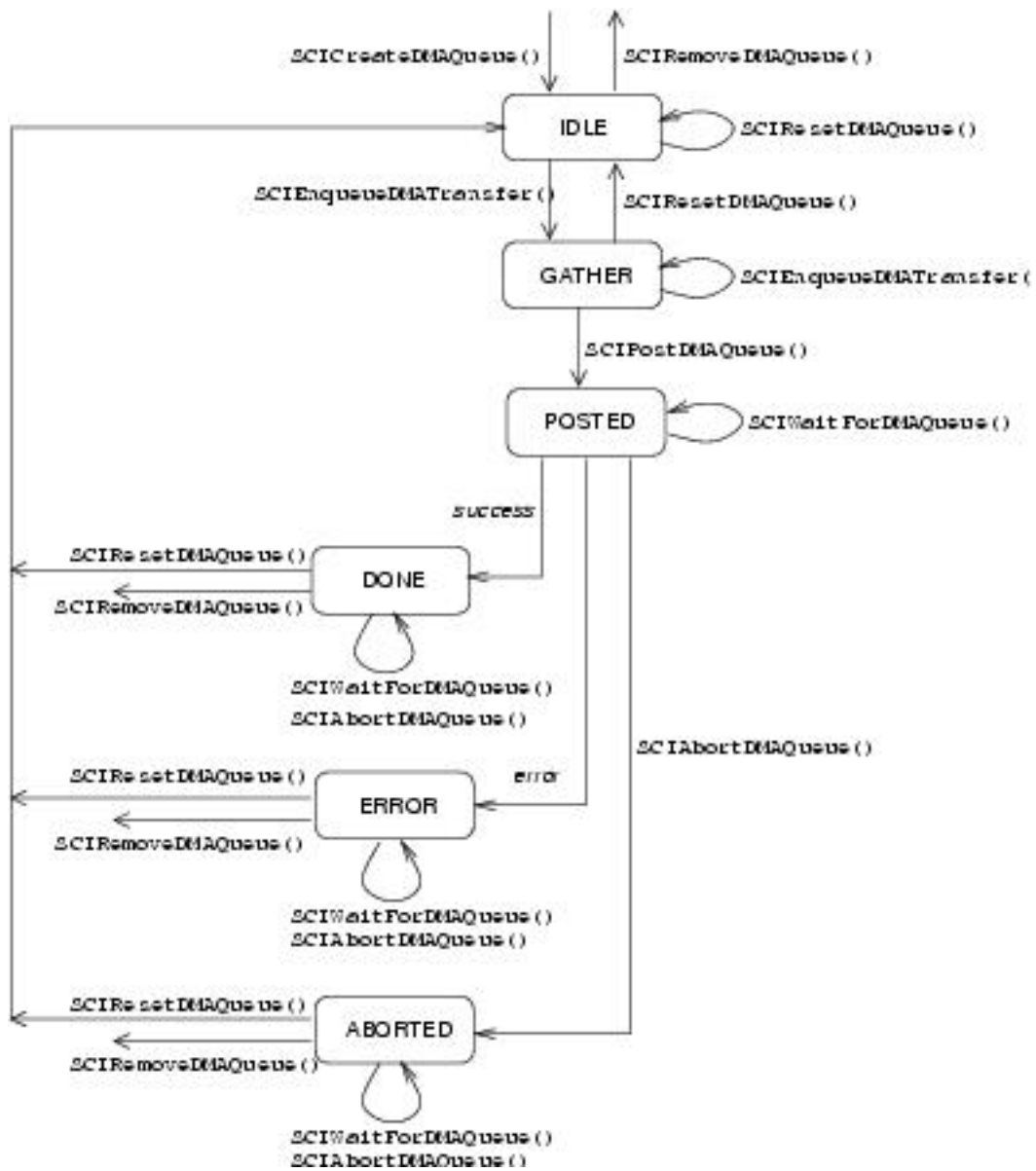
ERROR

at least one of the data transfers specified in the queue has failed;

ABORTED

the program has interrupted the DMA engine while it was processing the queue.

Figure 5-1. State diagram for a DMA queue



5.2. Creating a DMA queue

`SCICreateDMAQueue()` is used to allocate and initialise a DMA queue resource:

```
sci_desc_t v_dev;
sci_dma_queue_t dma_queue;
sci_error_t error;
unsigned int max_entries = 4;

SCIIInitialize(...);
SCIOpen(&v_dev, ...);
SCICreateDMAQueue(v_dev, &dma_queue, ADAPTER_NO,
                  max_entries, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the queue is available */
} else {
    /* manage error */
}
```

As usual `v_dev` is the virtual device that allows to communicate with the driver.

`dma_queue` is the handle to the DMA queue descriptor just allocated and initialised.

`ADAPTER_NO` states on which SCI adapter is the DMA engine we want to use.

`max_entries` is the maximum length of the queue, so in this case we can enqueue up to 4 data transfer specifications.

If the call to `SCICreateDMAQueue()` is successful the DMA queue moves to the `IDLE` state.

5.3. Removing a DMA queue

Once a DMA queue is not needed any more it can be released:

```
sci_dma_queue_t dma_queue;
sci_error_t error;

SCICreateDMAQueue(..., &dma_queue, ...);
/* use the queue */
SCIRemoveDMAQueue(dma_queue, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* queue successfully released */
} else {
    /* manage error */
}
```

```
}

```

Once the queue has been released the handle is not valid any more and should not be used.

According to Figure 5-1, `SCIRemoveDMAQueue()` can be called only if the queue is either in its initial state (IDLE) or in a final one (DONE, ERROR or ABORTED). If the call is successful the queue exits from the state diagram.

5.4. Querying the state of a DMA queue

Before executing an operation on a DMA queue it can be worthwhile to query its state to be sure that the operation would be legal. This can be done calling `SCIDMAQueueState()`:

```
sci_dma_queue_t dma_queue;
sci_error_t error;
sci_dma_queue_state_t dma_q_state;

SCICreatedDMAQueue(..., &dma_queue, ...);
dma_q_state = SCIDMAQueueState(dma_queue);
/* do something with the queue */

```

The type `sci_dma_queue_state_t` enumerates all the possible states a DMA queue can be in:

```
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;

```

Querying the state of a queue does not affect its current state.

5.5. Appending data transfers to a DMA queue

Once a DMA queue is available you can start appending data transfer specifications to it.

A data transfer specification consists of:

- a local memory segment
- a remote memory segment
- an offset within the local segment
- an offset within the remote segment
- a size

The function used for the append operation is `SCIEnqueueDMATransfer()`:

```
sci_error_t error;
sci_local_segment_t local_segment;
sci_remote_segment_t remote_segment;
sci_dma_queue_t dma_queue;
unsigned int local_offset = 0;
unsigned int remote_offset = 0;
unsigned int size = 4096;

SCICreateSegment(..., &local_segment, ...);
SCIPrepareSegment(local_segment, ADAPTER_NO, ...);
SCIConnectSegment(..., &remote_segment, ..., ADAPTER_NO, ...);
SCICreateDMAQueue(..., &dma_queue, ADAPTER_NO, ...);
SCIEnqueueDMATransfer(dma_queue, local_segment, remote_segment,
                      local_offset, remote_offset, size,
                      NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the transfer has been correctly enqueued */
} else {
    /* manage error */
}
```

By default data is written from the local segment (starting at offset `local_offset` and for `size` bytes) to the remote segment (starting at offset `remote_offset`). For the data to be read from the remote segment and copied locally, you have to specify the `SCI_FLAG_DMA_READ` flag.

The local segment must be “prepared” in order for the SCI adapter to be able to access it. Moreover note that the local segment preparation, the remote segment connection and the creation of the DMA queue are all consistent from the point of view of the SCI adapter used.

A non prepared local segment, a non connected remote segment, invalid specification of offsets and/or size are all cause of errors (check the SISI API specification for the details). The function call also fails if the DMA queue is already full.

According to the state diagram in Figure 5-1, the enqueue operation is allowed only if the queue is either in the IDLE or GATHER states. If the call is successful the queue enters the GATHER state.

5.6. Processing a DMA queue

Once the DMA queue contains one or more data transfer specifications it can be passed to the DMA engine on the SCI adapter for execution calling the function `SCIPostDMAQueue()`:

```
sci_error_t error;
sci_dma_queue_t dma_queue;

SCICreateDMAQueue(..., &dma_queue, ...);
SCIEnqueueDMATransfer(dma_queue, ...);
/* other SCIEnqueueDMATransfer() can go here */
SCIPostDMAQueue(dma_queue,
                NO_CALLBACK,
                NO_ARG,
                NO_FLAGS,
                &error);
if (error == SCI_ERR_OK) {
    /* queue execution has started */
} else {
    /* manage error */
}
```

The function implementation does not wait for the transfers to complete but returns simply when the queue has been passed to the DMA engine, so that the program can continue doing something else which doesn't depend on the data transfers.

Looking at the state diagram the posting operation can be performed only from the GATHER state and, if successful, moves the queue to the POSTED state.

5.7. Waiting for completion

How do we know when the processing of a DMA queue has terminated? There are several approaches to the problem and you can choose the solution more suitable for your application.

If you want a synchronous behaviour and you don't care about waiting for the transfer to complete you can just sit and wait:

```

sci_error_t error;
sci_dma_queue_t dma_queue;
sci_dma_queue_state_t dma_q_state;

SCICreatedDMAQueue(..., &dma_queue, ...);
SCIEnqueueDMATransfer(dma_queue, ...);
SCIPostDMAQueue(dma_queue, ...);
dma_q_state = SCIWaitForDMAQueue(dma_queue, SCI_INFINITE_TIMEOUT, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the value of dma_q_state tells if the data transfers
    * have been successful or failed */
} else {
    /* manage error */
}

```

If you don't want to wait forever you can just specify another value for the timeout, expressed in milliseconds, other than `SCI_INFINITE_TIMEOUT`. In such a case, if the timeout expires, error is set to `SCI_ERR_TIMEOUT`.

A call to `SCIWaitForDMAQueue()` is meaningful only from the `POSTED` state but it is allowed also from a final state (`ERROR`, `DONE`, `ABORTED`), where it is considered as a no-op. The function returns the state of the queue, which should be either `DONE` or `ERROR`, depending on the success of the data transfers.

Using `SCIWaitForDMAQueue()` is possible only if no callbacks have been specified in `SCIPostDMAQueue()`.

If you don't want to sit and wait for the completion of the DMA queue processing you can poll from time to time the queue state until it is in a final state, in particular if it is in the `DONE` or `ERROR` states. The code would look something like the following:

```

sci_error_t error;
sci_dma_queue_t dma_queue;
sci_dma_queue_state_t dma_q_state;

SCICreatedDMAQueue(..., &dma_queue, ...);
SCIEnqueueDMATransfer(dma_queue, ...);
SCIPostDMAQueue(dma_queue, ...);
while (...) {
    /* do something */
    sci_dma_queue_state_t dma_q_state;
    dma_q_state = SCIDMAQueueState(dma_queue);
    switch (dma_q_state) {
    case SCI_DMAQUEUE_DONE:
        /* good! all the transfers have completed successfully */
        break;
    case SCI_DMAQUEUE_ERROR:
        /* less good; manage the failed transfers, e.g. repost the queue */

```

```

        break;
default:
    /* other cases */
    break;
}
/* do something else */
}

```

Finally, if you want neither to wait explicitly for the completion of the queue processing nor to poll the state of the queue, the solution for you is to use a callback mechanism (see Section 7.2).

5.8. Aborting a DMA queue processing

If you want to stop the processing of a DMA queue you can do so by calling the function `SCIAbortDMAQueue()`. The call is meaningful only from the `POSTED` state but it is allowed also from a final state (`ERROR`, `DONE`, `ABORTED`) where it is considered as a no-op. In principle the state of the queue after a successful abort operation is `ABORTED`, but there is a potential race condition if the call happens at about the same time the queue processing is terminating and the state changing from `POSTED` to `DONE` (or to `ERROR`). To know what has really happened check the state of the queue with `SCIDMAQueueState()`.

```

sci_error_t error;
sci_dma_queue_t dma_queue;

SCICreateDMAQueue(..., &dma_queue, ...);
SCIEnqueueDMATransfer(dma_queue, ...);
SCIPostDMAQueue(dma_queue, ...);
/* do something */
SCIAbortDMAQueue(dma_queue, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    sci_dmaqueue_state_t dma_q_state;
    dma_q_state = SCIDMAQueueState(dma_queue);
    /* check the value of dma_q_state */
} else {
    /* manage error */
}

```

5.9. Resetting a DMA queue

Once the contents of a queue are not useful any more and you want to post some other transfers, you don't need to create a brand new queue, you can just reuse the old one, provided that you reset it:

```
sci_error_t error;
sci_dma_queue_t dma_queue;

SCICreatedDMAQueue(..., &dma_queue, ...);
SCIEnqueueDMATransfer(dma_queue, ...);
SCIPostDMAQueue(dma_queue, ...);
/* wait for the queue processing to complete */
SCIResetDMAQueue(dma_queue, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the queue is empty; you can fill it again */
    SCIEnqueueDMATransfer(dma_queue, ...);
    SCIPostDMAQueue(dma_queue, ...);
} else {
    /* manage error */
}
```

In this way you save the time for a queue allocation and deallocation.

The operation is legal from any state but POSTED, i.e. when the queue is being processed.

5.10. Example

Here we present the same example shown at the end of the previous chapter, a simple send-receive application: a sender program sends a command, this time using DMA, to a receiver program which then prints the “Hello, World!” string.

Example 5-1 shows the operations performed by the sender. Notice that it has to:

- allocate itself a local memory segment which will be the source of the DMA data transfer;
- map that memory segment into its own address space in order to be able to initialize it.

Example 5-1. A sender program based on DMA

```
/* DMA based sender program */
```

```

#include "sisci_api.h"

#define RECEIVER_NODE_ID 4
#define RECEIVER_MEM_ID 4
#define SENDER_MEM_ID 4
#define SENDER_MEM_SIZE 4
#define ADAPTER_NO 0
#define NO_CALLBACK 0
#define NO_ARG 0
#define NO_FLAGS 0
#define PRINT_COMMAND 1

int
main(int argc, char* argv[])
{
    sci_desc_t v_dev;
    sci_error_t error;
    sci_remote_segment_t remote_segment;
    unsigned int remote_segment_size;
    sci_local_segment_t local_segment;
    sci_map_t local_map;
    int* local_address;
    sci_dma_queue_t dma_queue;
    sci_dma_queue_state_t dma_queue_state;

    /* initialize the SCI environment */
    SCIInitialize(NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* create a virtual device */
    SCIOpen(&v_dev, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* allocate a local segment */ /* could use private segments */
    SCICreateSegment(v_dev, &local_segment, SENDER_MEM_ID, SENDER_MEM_SIZE,
                    NO_CALLBACK, NO_ARG, NO_FLAGS, &error);

    /* map the local segment */
    local_address =
        (int*)SCIMapLocalSegment(local_segment, &local_map,
                                0 /* offset */, SENDER_MEM_SIZE,
                                0 /* address hint */, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* create the DMA queue */

```

```

SCICreatedDMAQueue(v_dev, &dma_queue, ADAPTER_NO,
                   1 /* max entries */, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* connect to the remote segment */
SCIConnectSegment(v_dev, &remote_segment,
                  RECEIVER_NODE_ID, RECEIVER_MEM_ID,
                  ADAPTER_NO, NO_CALLBACK, 0,
                  SCI_INFINITE_TIMEOUT, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* initialize the local segment to contain the PRINT command */
*local_address = PRINT_COMMAND;

/* append a data transfer to the DMA queue */
SCIEnqueueDMATransfer(dma_queue, local_segment, remote_segment,
                      0 /* local offset */, 0 /* remote offset */,
                      sizeof(PRINT_COMMAND) /* transfer size */,
                      NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* pass the DMA queue to the SCI adapter for processing */
/* this sends the print command */
SCIPostDMAQueue(dma_queue, NO_CALLBACK, NO_ARG, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* wait for the DMA queue processing to complete */
SCIWaitForDMAQueue(dma_queue, INFINITE_TIMEOUT, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* be sure the transfer has completed successfully */
dma_queue_state = SCIDMAQueueState(dma_queue);
if (dma_queue_state != SCI_DMAQUEUE_DONE) return 1;

/* cleanup */
SCIRemovedDMAQueue(dma_queue, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIUnmapSegment(local_map, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIRemoveSegment(local_segment, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIDisconnectSegment(remote_segment, NO_FLAGS, &error);

```

```

    if (error != SCI_ERR_OK) return 1;

    SCIClose(v_dev, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    SCITerminate();

    return 0;
}

```

Example 5-2 shows the operations performed by the receiver. Notice that the program is *identical* to the one based on shared memory (see Example 4-2).

Example 5-2. A receiver program based on DMA

```

/* DMA based receiver program */

#include "sisci_api.h"
#include <stdio.h>

#define RECEIVER_MEM_ID 4
#define RECEIVER_MEM_SIZE 4096
#define ADAPTER_NO 0
#define NO_CALLBACK 0
#define NO_ARG 0
#define NO_FLAGS 0
#define PRINT_COMMAND 1

int
main(int argc, char* argv[])
{
    sci_desc_t v_dev;
    sci_error_t error;
    sci_local_segment_t local_segment;
    sci_map_t local_map;
    int* local_address;

    /* initialize the SCI environment */
    SCIInitialize(NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* create a virtual device */

```

```

SCIOpen(&v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* allocate a local segment */
SCICreateSegment(v_dev, &local_segment,
                 RECEIVER_MEM_ID, RECEIVER_MEM_SIZE,
                 NO_CALLBACK, NO_ARG, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* map the local segment */
local_address =
    (int*)SCIMapLocalSegment(local_segment, &local_map,
                              0 /* offset */,
                              RECEIVER_MEM_SIZE,
                              0 /* address hint */,
                              NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* initialise the contents of the first word of
the to-be-shared segment */
*local_address = ~PRINT_COMMAND;

/* export local segment */
SCIPrepareSegment(local_segment, ADAPTER_NO, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCISetSegmentAvailable(local_segment, ADAPTER_NO, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* wait for the sender process to send the print command */
while (*l_addr != PRINT_COMMAND) ;

printf("Hello, World!");

/* cleanup */
SCISetSegmentUnavailable(segment, ADAPTER_NO, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIRemoveSegment(local_segment, NO_FLAGS, error);
if (error != SCI_ERR_OK) return 1;

SCIClose(v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCITerminate();

```

```
    return 0;  
}
```


Chapter 6. Interrupts

Interrupts provide a way to notify a remote application that a certain predefined condition, identified by a positive integer number, has occurred.

Similarly to what happens for a memory segment, an SCI interrupt is allocated on a node and it is connected to and used from another one.

Managing SCI interrupts involves two types of resources:

- a local interrupt is the resource available on the allocating node; it is represented by a descriptor of type `sci_local_interrupt`, accessible via a handle of type `sci_local_interrupt_t`;
- a remote interrupt is the resource available on the importing node and corresponds to a local interrupt allocated on another node. A remote interrupt is represented by a descriptor of type `sci_remote_interrupt`, accessible via a handle of type `sci_remote_interrupt_t`.

In this chapter you'll learn:

- how to allocate an interrupt on the local node and make it available to other nodes;
- how to wait synchronously for an interrupt;
- how to connect to an interrupt available on a remote node;
- how to trigger a remote application using an interrupt.

6.1. Managing a local interrupt

6.1.1. Allocating an interrupt

An interrupt resource is allocated, initialized and made available to remote nodes calling the function `SCICreateInterrupt()`:

```
sci_desc_t v_dev;
sci_local_interrupt_t local_interrupt;
unsigned int interrupt_no;
sci_error_t error;

SCIInitialize(...)
SCIOpen(&v_dev, ...);
/* possibly set interrupt_no */
```

```

SCICreateInterrupt(v_dev, &local_interrupt, ADAPTER_NO,
                  &interrupt_no, NO_CALLBACK, NO_ARG,
                  NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the interrupt is available to remote applications */
} else {
    /* manage error */
}

```

By default it is the driver that assigns an identifier to the just created interrupt. For a remote application to use the interrupt, you have to make the identifier known to it in some way. Alternatively you can propose an identifier to the driver, setting in advance the parameter `interrupt_no` and using the flag `SCI_FLAG_FIXED_INTNO` when calling `SCICreateInterrupt()`. If the number is already in use the error returned is `SCI_ERR_INTNO_USED`. This is actually what we'll do in Example 6-2 to avoid to implement an additional mechanism to communicate the identifier to the application running on the other node.

Note that the same function not only allocates and initialises the interrupt but also makes it available to other nodes.

6.1.2. Deallocating an interrupt

Once an interrupt is no longer needed and all the remote nodes have disconnected from it, the local interrupt resource can be freed:

```

sci_error_t error;
sci_local_interrupt_t local_interrupt;

SCICreateInterrupt(..., &local_interrupt, ...);
/* use the interrupt */
SCIRemoveInterrupt(local_interrupt, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the interrupt resource has been correctly freed */
} else {
    /* manage error */
}

```

6.1.3. Waiting for an interrupt

The simplest way to wait for an interrupt to be triggered is to explicitly wait for that event.

```

sci_error_t error;
sci_local_interrupt_t local_interrupt;

SCICreateInterrupt(..., &local_interrupt, ...);
SCIWaitForInterrupt(local_interrupt, SCI_INFINITE_TIMEOUT, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the interrupt has been triggered */
} else {
    /* manage error */
}

```

The timeout is expressed in milliseconds.

If an error occurs it can be due to two reasons: the timeout has expired or the interrupt has been removed in the meantime, presumably by another thread.

Alternatively it's possible to use a callback mechanism, but this is addressed only in Section 7.2.4.

6.2. Managing a remote interrupt

6.2.1. Connecting to an interrupt

Like for memory segments, the first thing an application has to do in order to be able to trigger an interrupt on a another node is to “connect” to that interrupt. The operation, achieved by calling `SCIConnectInterrupt()` allocates and initializes a remote interrupt resource, providing a handle to it.

```

sci_desc_t v_dev;
unsigned int remote_interrupt_no;
sci_remote_interrupt_t remote_interrupt;
sci_error_t error;

SCIInitialize(...);
SCIOpen(&v_dev, ...);
/* set remote_interrupt_no */
SCIConnectInterrupt(v_dev, &remote_interrupt, RECEIVER_NODE_ID,
                   ADAPTER_NO, remote_interrupt_no,
                   SCI_INFINITE_TIMEOUT, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the interrupt is available for triggering */
} else {

```

```

    /* manage error */
}

```

A remote interrupt is identified by the node identifier of the exporting node and an integer number, the same one used to create it with `SCICreateInterrupt()`. This number may be assigned directly by the underlying SCI software, in which case another mechanism must be foreseen to make this number available to other nodes.

6.2.2. Disconnecting from an interrupt

Once a remote interrupt resource isn't needed any more it has to be released with `SCIDisconnectInterrupt()`.

```

sci_remote_interrupt_t remote_interrupt;
sci_error_t error;

SCIConnectInterrupt(..., &remote_interrupt, ...)
/* use the remote interrupt */
SCIDisconnectInterrupt(remote_interrupt, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the interrupt is not available any more */
} else {
    /* manage error */
}

```

6.2.3. Triggering an interrupt

An remote interrupt is triggered using `SCITriggerInterrupt()`.

```

sci_remote_interrupt_t remote_interrupt;
sci_error_t error;

SCIConnectInterrupt(..., &remote_interrupt, ...);
SCITriggerInterrupt(remote_interrupt, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the remote interrupt has been successfully triggered */
} else {
    /* manage error */
}

```

The application that exported the interrupt, if waiting, should then wake up and proceed its execution. If instead it's not waiting the interrupt is lost.

6.3. Example

As in the previous examples for shared memory and DMA, also in this case the example application is actually composed of two programs: a sender which connects to and triggers an interrupt exported by a receiver, which after receiving the interrupt prints "Hello, World!".

Example 6-1 and Example 6-2 show the operations performed by the sender and by the receiver respectively.

Example 6-1. A sender program based on interrupts

```
/* interrupt based sender program */

#include "sisci_api.h"

#define RECEIVER_NODE_ID 4
#define RECEIVER_INTERRUPT_NO 12345
#define ADAPTER_NO 0
#define NO_CALLBACK 0
#define NO_ARG 0
#define NO_FLAGS 0

int
main(int argc, char* argv[])
{
    sci_desc_t v_dev;
    sci_error_t error;
    sci_remote_interrupt_t remote_interrupt;

    /* initialize the SCI environment */
    SCIInitialize(NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* create a virtual device */
    SCIOpen(&v_dev, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) return 1;

    /* connect to the remote interrupt */
    SCIConnectInterrupt(v_dev, &remote_interrupt, RECEIVER_NODE_ID,
```

```

        ADAPTER_NO, RECEIVER_INTERRUPT_NO,
        SCI_INFINITE_TIMEOUT, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* send the print command, i.e. trigger the interrupt */
SCITriggerInterrupt(remote_interrupt, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* cleanup */
SCIDisconnectInterrupt(remote_interrupt, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIClose(v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCITerminate();

return 0;
}

```

Example 6-2. A receiver program based on interrupts

```

/* interrupt based receiver program */

#include "sisci_api.h"
#include <stdio.h>

#define RECEIVER_INTERRUPT_NO 12345
#define ADAPTER_NO 0
#define NO_CALLBACK 0
#define NO_ARG 0
#define NO_FLAGS 0

int
main(int argc, char* argv[])
{
    sci_desc_t v_dev;
    sci_error_t error;
    unsigned int interrupt_no;
    sci_local_interrupt_t local_interrupt;

```

```
/* initialize the SCI environment */
SCIInitialize(NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* create a virtual device */
SCIOpen(&v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

/* use a predefined interrupt number; hope it's not in use... */
interrupt_no = RECEIVER_INTERRUPT_NO;

/* allocate an interrupt and make it available */
SCICreateInterrupt(v_dev, &local_interrupt, ADAPTER_NO, &interrupt_no,
                  NO_CALLBACK, NO_ARG, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIWaitForInterrupt(local_interrupt, SCI_INFINITE_TIMEOUT,
                  NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

printf("Hello, World!");

/* cleanup */
SCIRemoveInterrupt(local_interrupt, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCIClose(v_dev, NO_FLAGS, &error);
if (error != SCI_ERR_OK) return 1;

SCITerminate();

return 0;
}
```


Chapter 7. Advanced Features

This chapter addresses some issues which are slightly more advanced than the basic features presented up to now. Nonetheless they are fundamental for writing a complete SCI application.

In this chapter you'll learn:

- how to exploit caching techniques to improve performance;
- how to check for data transfer errors and cope with them;
- how to exploit events generated by the underlying SCI system;
- how to use the available callback mechanism in a number of situations.

7.1. Caching and error checking

Reading and writing data residing on a remote node takes more time than a corresponding operation performed on local memory. To improve the efficiency of remote access the SCI adapter employs some caching techniques, trying to read or write full SCI packets.

This means for example that even if a read instruction is for a single word, the actual read gets more than that, exploiting the knowledge that applications typically read data sequentially. The next word, when requested, would then be returned directly by the adapter without issuing a new remote operation. This design can lead to serious problems with dynamically changing data; that's why a timeout is applied to such buffers; when the timeout expires the data is invalidated. Alternatively the application can explicitly ask for the contents of these buffers to be discarded.

For the write case the situation is similar. The adapter keeps data to be written to a remote node in a buffer until a full packet can be filled, which would then lead to a single write operation. Also in this case the rationale is the locality principle, which states that applications tend to write data sequentially. As happens for reads this is not always true, so when a timeout expires the available data is forced out. Alternatively the application can explicitly flush the buffer.

Remote data access is also more error-prone than local access. Although the SCI protocol is robust an error can always happen, due for example to congestion in a switch or a cable being unplugged. Error situations detected by the SCI hardware are made available to the upper software layers, which can then react appropriately; for example an application may decide to ignore the error while another may retry the affected data transfer.

The SISCO API provides the means to cope both with buffering and with data transfer errors based on shared memory. In both cases the solution is based on the concept of sequence.

7.1.1. Sequences

A sequence is a resource associated to a remote mapped segment, which allows

- to control the buffering of data for that segment;
- to check for errors in data transfers from and to that segment.

The name “sequence” comes from the fact that you are supposed to use it when you execute a sequence of read or write operations on that segment.

Like all the resources a sequence has its own descriptor type (`sci_sequence`) and its own handle type (`sci_sequence_t`).

A sequence resource is allocated using the API function `SCICreateMapSequence()`:

```
sci_map_t remote_map;
sci_sequence_t sequence;
sci_error_t error;

SCIMapRemoteSegment(..., &remote_map, ...);
SCICreateMapSequence(remote_map, &sequence, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the sequence is available */
} else {
    /* manage error */
}
```

`remote_map` represents a remote segment which has been memory-mapped locally. `sequence` is the handle to the sequence descriptor which is allocated and initialised by the function call, if successful.

Once a sequence is not used any more it is destroyed invoking `SCIRemoveSequence()`:

```
sci_sequence_t sequence;
sci_error_t error;

SCICreateMapSequence(..., &sequence, ...);
/* use the sequence */
SCIRemoveSequence(sequence, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* the sequence resource is released */
} else {
    /* manage error */
}
```

7.1.2. Flushing buffers

As mentioned above one may want to flush read and write buffers at convenience.

`SCIFlushReadBuffers()` is provided to clear the read buffers:

```
sci_sequence_t sequence;
sci_map_t remote_map;
volatile void* map_address;
map_address = SCIMapRemoteSegment(..., &remote_map, ...);
SCICreateMapSequence(remote_map, &sequence, ...);
*map_address; /* remote read operations */
SCIFlushReadBuffers(sequence);
*map_address; /* read again */
```

If the value of the remote memory location pointed to by `map_address` changes between the first and the second read operation you are sure that the second time you would get the new value.

`SCIStoreBarrier()` is instead used to empty the write buffers and force SCI packets to be sent out:

```
sci_sequence_t sequence;
sci_map_t remote_map;
volatile void* map_address;

map_address = SCIMapRemoteSegment(..., &remote_map, ...);
SCICreateMapSequence(remote_map, &sequence, ...);
*map_address = 1; /* remote write operation */
SCIStoreBarrier(sequence, NO_FLAGS);
```

`SCIStoreBarrier()` returns only when all the SCI protocol transactions related to the associated segment have completed. In other words when `SCIStoreBarrier()` returns you are sure that all the possibly buffered data for that segment has arrived at the receiver's memory.

Note that `SCIFlushReadBuffers()` is a local-only operation, affecting only some read buffers on the local SCI interface, while `SCIStoreBarrier()` usually causes some data transmission and therefore some interaction with remote nodes.

7.1.3. Checking for data transfer errors

As said above a sequence is also used to check for errors during data transfers based on shared memory.

First of all a sequence needs to be cleared before starting to move data. What this means is that no pending errors should exist for the concerned segment.

```

sci_sequence_t sequence;
sci_error_t error;
sci_sequence_status_t status;
sci_map_t remote_map;

SCICreateMapSequence(remote_map, &sequence, ...);
status = SCIStartSequence(sequence, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* check the status */
} else {
    /* manage error */
}

```

The return value of `SCIStartSequence()`, provided it is successful, tells if the data transfer can start (status equal to `SCI_SEQ_OK`) or if there are some pending errors (status equal to `SCI_SEQ_PENDING`). In the latter case the function must be called again. The type `sci_sequence_status_t` contains four different values but the other two are meaningless for `SCIStartSequence()`.

The call to `SCIStartSequence()` before a data transfer then becomes:

```

sci_error_t error;
sci_sequence_t sequence;
sci_sequence_status_t status;

do {
    status = SCIStartSequence(sequence, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) {
        /* manage error */
    }
} while (status != SCI_SEQ_OK);
/* can transfer data */

```

`SCICheckSequence()` checks if a data transfer was affected by errors. What this function actually does is to check that no errors have occurred since the last successful check, which could have been performed either by `SCICheckSequence()` or `SCIStartSequence()`.

```

sci_sequence_t sequence;
sci_error_t error;
sci_sequence_status_t status;

do {
    status = SCIStartSequence(sequence, NO_FLAGS, &error);
    if (error != SCI_ERR_OK) {

```

```

        /* manage error */
    }
} while (status != SCI_SEQ_OK)
/* transfer data */
status = SCICheckSequence(sequence, NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    /* check the status */
} else {
    /* manage errors */
}

```

status can assume all the four possible values of the `sci_sequence_status_t` type:

SCI_SEQ_OK

the transfer was error free;

SCI_SEQ_RETRIABLE

the transfer failed due to a non-fatal error (e.g. system busy because of heavy traffic) but can be immediately retried;

SCI_SEQ_NOT_RETRIABLE

the transfer failed due to a fatal error (e.g. cable unplugged) and can be retried only after a successful call to `SCIStartSequence()`;

SCI_SEQ_PENDING

the transfer failed but the driver hasn't been able yet to determine the severity of the error (if fatal or non-fatal); `SCIStartSequence()` must be called until it succeeds.

By default `SCICheckSequence()` also flushes the write buffers and waits for all the SCI transactions to complete; in other words it internally performs an action similar to what `SCIStoreBarrier()` does. If you don't want the flush to be performed pass `SCI_FLAG_NO_FLUSH` as a flag. Alternatively you can allow the flush but not the waiting for the completion of all the outstanding SCI transactions; in this case pass `SCI_FLAG_NO_STORE_BARRIER` as a flag. They can be OR'ed together if you want to pass both the flags.

So, if you want an error-free data transfer you should:

1. start the sequence;
2. transfer the data;
3. check the sequence;

- a. if it's ok continue;
- b. if it's not ok and the error is not fatal, retry the transfer and check the sequence;
- c. if it's not ok and the error is fatal, restart the sequence, retry the transfer and check the sequence.

The corresponding code would be something like the following:

```

sci_error_t error;
sci_sequence_t sequence;
sci_sequence_status_t status;

do {
    status = SCIStartSequence(sequence, ..., &error);
    if (error != SCI_ERR_OK) {
        /* manage error */
    }
} while (status != SCI_SEQ_OK);
/* transfer data */
status = SCICheckSequence(sequence, ..., &error);
if (error != SCI_ERR_OK) {
    /* manage error */
}
while (status != SCI_SEQ_OK) {
    switch (status) {
        case SCI_SEQ_RETRIABLE:
            break;
        case SCI_SEQ_PENDING:
        case SCI_SEQ_NOT_RETRIABLE:
            /* need a successful SCIStartSequence() before retrying */
            do {
                status = SCIStartSequence(sequence, ..., &error);
                if (error != SCI_ERR_OK) {
                    /* manage error */
                }
            } while (status != SCI_SEQ_OK);
            break;
        default:
            /* shouldn't happen; manage error */
    }
}
/* transfer data */
status = SCICheckSequence(sequence, ..., &error);
if (error != SCI_ERR_OK) {
    /* manage error */
}

```

```

    }
}

```

If instead your application doesn't require reliable data transfers but you still want to log when an error occurs you could use something like the following:

```

sci_error_t error;
sci_sequence_t sequence;
sci_sequence_status_t status;

do {
    status = SCIStartSequence(sequence, ..., &error);
    if (error != SCI_ERR_OK) {
        /* manage error */
    }
} while (status != SCI_SEQ_OK);
/* transfer data */
status = SCICheckSequence(sequence, ..., &error);
if (error != SCI_ERR_OK) {
    /* manage error */
}
while (status != SCI_SEQ_OK) {
    switch (status) {
        case SCI_SEQ_RETRIABLE:
            break;
        case SCI_SEQ_PENDING:
        case SCI_SEQ_NOT_RETRIABLE:
            /* need a successful SCIStartSequence() before retrying */
            do {
                status = SCIStartSequence(sequence, ..., &error);
                if (error != SCI_ERR_OK) {
                    /* manage error */
                }
            } while (status != SCI_SEQ_OK);
            break;
        default:
            /* shouldn't happen; manage error */
    }
    /* log the occurrence of the error */
    status = SCICheckSequence(sequence, ..., &error);
    if (error != SCI_ERR_OK) {
        /* manage error */
    }
}
}

```

Notice how the code is almost identical to the previous example for a reliable communication, the only difference being that now you don't retry the data transfer after a failure; instead you simply log somewhere that an error occurred. You still need to restart the sequence in case of error because otherwise the error flags wouldn't be in a clean state at the beginning of the next data transfer.

7.2. Events and callbacks

On certain conditions a component of an SCI system, be it either a piece of hardware, a driver or an application, generates a so-called event. Examples of an event are a cable being plugged or unplugged, the disappearance of a remote segment because of a node failure, the completion of a DMA queue processing, a triggered interrupt.

Some events are managed directly by the SCI driver, whereas others can be forwarded to an application, which can then either ignore or catch them.

There are basically two ways you can catch an event: either you wait for it, blocking the process, or you can register a callback function, which will be called when the event occurs.

Each of the following sections concerns a different context for events: local memory segments, remote memory segments, DMA queues, interrupts.

7.2.1. Events and local memory segments

Five events are related to local memory segments. Their names are collected in an enumeration type called `sci_segment_cb_reason_t`. The name comes from the fact that a member of such enumeration is passed to a callback function as the reason for its invocation (see later in the section).

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

Their meaning is the following:

SCI_CB_CONNECT

a new connection has been established from a remote node;

SCI_CB_DISCONNECT

an existing connection has been released;

SCI_CB_NOT_OPERATIONAL

the route to a connected node is temporarily unavailable;

SCI_CB_OPERATIONAL

the route to a connected node is available (again);

SCI_CB_LOST

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

All these events are passed to the application, which may wish to intercept them.

As anticipated in Section 3.1, all the resources depending on a local segment should be freed before the segment itself could be removed. Some of this dependencies are due to connected nodes, so, for example, the exporting application can catch the above events, in particular the CONNECT, DISCONNECT and LOST events, in order to keep an up-to-date table of all the established connections, together with their state.

7.2.1.1. Synchronous event catching

The simplest way to catch events is just to wait for them:

```
sci_local_segment_t segment;
unsigned int source_node_id;
unsigned int local_adapter_no;
sci_error_t error;
sci_segment_cb_reason_t reason;

SCICreateSegment(..., &segment, ..., NO_FLAGS, ...);
SCIPrepareSegment(..., segment, ...);
SCISetSegmentAvailable(..., segment, ...);
/* now the segment is available for remote connections */
reason = SCIWaitForLocalSegmentEvent(segment,
                                     &source_node_id,
                                     &local_adapter_no,
                                     SCI_INFINITE_TIMEOUT,
                                     NO_FLAGS,
                                     &error);
```

```

if (error == SCI_ERR_OK) {
    switch (reason) {
        case SCI_CB_CONNECT:
            /* update the connection table for the segment */
            break;
        case SCI_CB_DISCONNECT:
            /* update the connection table for the segment */
            break;
        case SCI_CB_OPERATIONAL:
            /* the segment is usable */
            break;
        case SCI_CB_NOT_OPERATIONAL:
            /* the situation may recover */
            break;
        case SCI_CB_LOST:
            /* update the connection table for the segment */
            break;
        default:
            /* error */
            break;
    }
} else {
    /* manage error */
}

```

In the above piece of program, after having made a local segment available to remote nodes, we sit and wait for an event concerning the segment. As output the call to `SCIWaitForLocalSegmentEvent()` gives the event that caused it to return; moreover it sets the identifier of the node that generated the event and the local adapter that received that event.

`SCIWaitForLocalSegmentEvent()` can fail because the timeout has expired (with `error` set to `SCI_ERR_TIMEOUT`) or because the handle is invalid, for example because the segment has been removed (`error` is set to `SCI_ERR_CANCELLED`).

7.2.1.2. Asynchronous event catching

Alternatively an event can be caught asynchronously, through a callback mechanism. For this a callback function must be registered when calling `SCICreateSegment()`. Compare the following excerpt of C code with the one used in Section 3.1:

```

sci_desc_t v_dev;
sci_local_segment_t segment;
sci_error_t error;

```

```

void* arg = 0;

SCIInitialize(...);
SCIOpen(&v_dev,...);
/* possible setting of arg */
SCICreateSegment(v_dev,
                 &segment,
                 RECEIVER_MEM_ID, /* segment identifier */
                 RECEIVER_MEM_SIZE, /* size */
                 local_segment_cb, /* callback function */
                 arg, /* callback argument */
                 SCI_FLAG_USE_CALLBACK, /* enables callback */
                 &error);
if (error == SCI_ERR_OK) {
    /* a segment is available for use */
} else {
    /* manage error */
}

```

where `local_segment_cb` is a function with the following prototype:

```

sci_callback_action_t
local_segment_cb(void* arg,
                sci_local_segment_t segment,
                sci_segment_cb_reason_t reason,
                unsigned int node_id,
                unsigned int local_adapter_no,
                sci_error_t error);

```

which corresponds to the type `sci_cb_local_segment_t`.

In the callback prototype `arg` is the same parameter specified in `SCICreateSegment()`. It is defined as a `void*` so that anything can be passed to it: from a null pointer, to a primitive value, to a pointer to a larger data structure.

The other parameters guarantee that the same information that is available after `SCIWaitForLocalSegmentEvent()` returns is also available within the body of the callback function.

Note how the value of the parameter `flags` in `SCICreateSegment()` is now `SCI_FLAG_USE_CALLBACK`. According to the API specification the use of this option prevents from using `SCIWaitForLocalSegmentEvent()`.

The return value of a callback is of type `sci_callback_action_t` which is defined as follows:

```

typedef enum {
    SCI_CALLBACK_CANCEL = 1,
    SCI_CALLBACK_CONTINUE
}

```

```
} sci_callback_action_t;
```

The return value of the callback function tells the driver whether the callback is still active (SCI_CALLBACK_CONTINUE) or not (SCI_CALLBACK_CANCEL) after the function execution.

A possible implementation of the callback function could simply include the switch statement introduced above, where `arg` can be, for example, the connection table:

```
sci_callback_action_t
local_segment_cb(void* arg,
                 sci_local_segment_t segment,
                 sci_segment_cb_reason_t reason,
                 unsigned int node_id,
                 unsigned int local_adapter_no)
{
    /* convert arg to a pointer to a connection table */
    switch (reason) {
    case SCI_CB_CONNECT:
        /* update the connection table for the segment */
        break;
    case SCI_CB_DISCONNECT:
        /* update the connection table for the segment */
        return SCI_CALLBACK_CANCEL;
        break;
    case SCI_CB_OPERATIONAL:
        /* the segment is usable */
        break;
    case SCI_CB_NOT_OPERATIONAL:
        /* the situation may recover */
        break;
    case SCI_CB_LOST:
        /* update the connection table for the segment */
        return SCI_CALLBACK_CANCEL;
        break;
    default:
        /* error */
        break;
    }

    return SCI_CALLBACK_CONTINUE;
}
```

7.2.2. Events and remote memory segments

The same five events related to local memory segments apply also to remote segments, though their meaning is different. Let's recall their enumeration type `sci_segment_cb_reason_t`:

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

Their meaning for a remote segment, though it will become clear only after you have read the next sections, is as follows:

`SCI_CB_CONNECT`

an asynchronous connection (see Section 3.2) has completed successfully;

`SCI_CB_DISCONNECT`

an asynchronous connection (see Section 3.2) has failed;

`SCI_CB_NOT_OPERATIONAL`

the route to the exporting node is temporarily unavailable;

`SCI_CB_OPERATIONAL`

the route to the exporting node is available, or is available again;

`SCI_CB_LOST`

an unrecoverable situation has occurred on the exporting node.

7.2.2.1. Asynchronous connection

In Section 3.2.1 we have seen how to connect to a remote segment in a synchronous way, that is, we wait that `SCIConnectSegment()` returns either because the connection has completed or because a timeout has expired. Alternatively it's possible to only initiate the connection and wait for the completion while doing other things. This is possible calling `SCIConnectSegment()` with the flag `SCI_FLAG_ASYNCHRONOUS_CONNECT`:

```
sci_remote_segment_t remote_segment;
sci_error_t error;
```

```

SCIConnectSegment(..., &segment, ..., SCI_FLAG_ASYNCHRONOUS_CONNECT, &error);
if (error == SCI_ERR_OK) {
    /* the handle is valid, but the remote segment
    * is NOT necessarily connected */
} else {
    /* manage error */
}

```

In this case the `SCIConnectSegment()` returns immediately with a valid handle. Be careful that a valid handle doesn't mean a valid descriptor. If the connection request is satisfied the descriptor will be validated later, once the driver has filled all the fields with proper values. If the connection is refused the descriptor will never become valid. In both cases the driver generates an appropriate event (`SCI_CB_CONNECT` and `SCI_CB_DISCONNECT` respectively) to notify the result to the application, which has then to react accordingly. In particular, in case of failure it has to release the descriptor, even if invalid, with `SCIDisconnectSegment()` (see Section 3.2.2).

7.2.2.2. Synchronous event catching

As for a local segment resource, events can be caught either synchronously, with a wait function, or asynchronously, enabling the callback mechanism.

Let's start with the synchronous approach:

```

sci_remote_segment_t segment;
sci_error_t status;
sci_error_t error;

SCIConnectSegment(..., &segment, ...);
reason =
    SCIWaitForRemoteSegmentEvent(segment, &status,
                                SCI_INFINITE_TIMEOUT,
                                NO_FLAGS, &error);
if (error == SCI_ERR_OK) {
    switch (reason) {
    case SCI_CB_CONNECT:
        /* the previous call to SCIConnectSegment() has succeeded;
        the handle to the descriptor is now usable */
        /* this case makes sense only if SCIConnectSegment() was
        called with the flag SCI_FLAG_ASYNCHRONOUS_CONNECT */
        break;
    case SCI_CB_DISCONNECT:
        /* SCISetSegmentUnavailable() has been called
        on the exporting node with SCI_FLAG_NOTIFY;

```

```

        we should clean up and disconnect */
    /* clean up */
    SCIDisconnectSegment(segment, NO_FLAGS, &error);
    break;
case SCI_CB_OPERATIONAL:
    /* the connection is established (or re-established);
       the segment is usable */
    break;
case SCI_CB_NOT_OPERATIONAL:
    /* wait for the connection to recover */
    break;
case SCI_CB_LOST:
    /* the connection is lost,
       the segment is not usable any more */
    break;
default:
    /* error */
    break;
}
} else {
    /* manage error */
}

```

SCIWaitForRemoteSegmentEvent() fails if the timeout expires (with error set to SCI_ERR_TIMEOUT) or if the segment has already been disconnected, so that the handle is not valid anymore (error is set to SCI_ERR_CANCELLED).

In case the event is SCI_CB_LOST it is responsibility of the application to clean things up so that resources, for example descriptors, be appropriately released. So the segment, if mapped, has to be unmapped with SCIUnmapSegment() and then it has to be disconnected with SCIDisconnectSegment().

7.2.2.3. Asynchronous event catching

Also for remote segment events there is the alternative to catch them asynchronously specifying an appropriate callback function in the call to SCIConnectSegment(). Compare the following code with the one shown in Section 3.2.1:

```

sci_desc_t v_dev;
sci_remote_segment_t remote_segment;
sci_error_t error;
void* arg = 0;

SCIInitialize(...);

```

```

SCIOpen(&v_dev, ...);
/* possible setting of arg */
SCIConnectSegment(v_dev,
                  &remote_segment,
                  RECEIVER_NODE_ID,
                  RECEIVER_MEM_ID,
                  ADAPTER_NO,
                  remote_segment_cb, /* callback function */
                  arg, /* callback argument */
                  SCI_INFINITE_TIMEOUT,
                  SCI_FLAG_USE_CALLBACK, /* enables callback */
                  &error);
if (error == SCI_ERR_OK) {
    /* the remote segment is connected */
} else {
    /* manage error */
}

```

where `remote_segment_cb()` has the following prototype:

```

sci_callback_action_t
remote_segment_cb(void* arg,
                  sci_remote_segment_t remote_segment,
                  sci_segment_cb_reason_t reason,
                  sci_error_t status);

```

which corresponds to the type `sci_cb_remote_segment_t`.

In the callback prototype, `arg` is the same parameter specified in `SCIConnectSegment()`. It is defined as a `void*` so that it can be casted easily to any other type. `arg` can for example represent some parameters associated with segment, such as the remote node and segment identifier.

The other parameters guarantee that the same information that is available after `SCIWaitForRemoteSegmentEvent()` returns is also available within the body of the callback function.

Note how the value of the parameter flags in `SCIConnectSegment()` is now `SCI_FLAG_USE_CALLBACK`. According to the API specification the use of this option prevents from using `SCIWaitForRemoteSegmentEvent()`.

The return value of a callback is of type `sci_callback_action_t` which is defined as follows:

```

typedef enum {
    SCI_CALLBACK_CANCEL = 1,
    SCI_CALLBACK_CONTINUE
}

```

```
} sci_callback_action_t;
```

The return value of the callback function tells the driver whether the callback is still active (SCI_CALLBACK_CONTINUE) or not (SCI_CALLBACK_CANCEL) after the function execution.

A possible implementation of `remote_segment_cb()` could for example include the switch statement shown in the previous Section:

```
sci_callback_action_t
remote_segment_cb(void* arg,
                  sci_remote_ement_t segment,
                  sci_segment_cb_reason_t reason,
                  sci_error_t status)
{
    switch (reason) {
    case SCI_CB_CONNECT:
        /* an asynchronous connection, i.e. SCIConnectSegment()
         * called with the flag SCI_FLAG_ASYNCHRONOUS_CONNECT,
         * has succeeded */
        break;
    case SCI_CB_DISCONNECT:
        /* SCISetSegmentUnavailable() has been called
         * on the exporting node with SCI_FLAG_NOTIFY;
         * we should clean up and disconnect */
        /* clean up */
        SCIDisconnectSegment(segment, NO_FLAGS, &error);
        break;
    case SCI_CB_OPERATIONAL:
        /* the connection is established (or re-established);
         * the segment is usable */
        break;
    case SCI_CB_NOT_OPERATIONAL:
        /* wait for the connection to recover */
        break;
    case SCI_CB_LOST:
        /* the connection is lost,
         * the segment is not usable any more */
        break;
    default:
        /* error */
        break;
    }

    return SCI_CALLBACK_CONTINUE;
}
```

```
}

```

7.2.3. DMA and callbacks

In Section 5.7 we have explored two ways for checking if the processing of a DMA queue has completed. One is based on a blocking wait function, the other is based on polling the state of the queue until it is in a final state. In this section you'll learn how to use a third way, which consists on a callback mechanism: you declare that a certain function be called when the DMA queue processing has terminated.

You declare the callback function when you post the queue for processing:

```
sci_error_t error;
sci_dma_queue_t dma_queue;
void* arg = 0;

SCICreatedDMAQueue(..., &dma_queue, ...);
SCIEnqueueDMATransfer(dma_queue, ...);
/* other enqueue's */
/* possible setting of arg */
SCIPostDMAQueue(dma_queue,
                dma_queue_cb,
                arg,
                SCI_FLAG_USE_CALLBACK,
                &error);
if (error == SCI_ERR_OK) {
    /* queue posted successfully */
} else {
    /* manage error */
}

```

where `dma_queue_cb()` has the following prototype:

```
sci_callback_action_t
dma_queue_cb(void* arg, sci_dma_queue_t dma_queue, sci_error_t status);

```

which corresponds to the type `sci_cb_dma_t`.

Notice also that you must explicitly `SCI_FLAG_USE_CALLBACK` as a flag for the callback function to be considered.

Remember that if you use a callback you are not allowed to use `SCIWaitForDMAQueue()`.

7.2.4. Interrupts and callbacks

In Section 6.1.3 we have seen that interrupts can be caught synchronously calling `SCIWaitForInterrupt()`. Alternatively at creation time you can specify a function to be called asynchronously when an interrupt arrives.

```
sci_desc_t v_dev;
sci_local_interrupt_t local_interrupt;
unsigned int interrupt_no;
sci_error_t error;
void* arg = 0;

SCIInitialize(...);
SCIOpen(&v_dev, ...);
/* possible setting of arg */
SCICreateInterrupt(v_dev, &local_interrupt, ADAPTER_NO,
                  interrupt_no, interrupt_cb, arg,
                  SCI_FLAG_USE_CALLBACK, &error);
if (error == SCI_ERR_OK) {
    /* the interrupt is available to remote applications */
} else {
    /* manage error */
}
```

where `interrupt_cb()` has the following prototype:

```
sci_callback_action_t
interrupt_cb(void* arg, sci_local_interrupt_t interrupt, sci_error_t error);
```

which corresponds to the type `sci_cb_interrupt_t`.

Also in this case remember that:

- you must pass the `SCI_FLAG_USE_CALLBACK` for the callback to be considered;
- if you use a callback you are not allowed to use `SCIWaitForInterrupt()`.

