SISCI Users Guide

Remote memory access made safe and easy

Dolphin Interconnect Solutions
05.02.2015
Version 4.01

SISCI API User Guide

By Dolphin Interconnect Solutions.

The goal of this guide is to introduce the reader to the SISCI API, which allows the development of applications exploiting the powerful capabilities of remote memory access interconnects.

Contents

Introduction

The powerful functionality offered by memory mapped interconnect hardware is well suited to implement low latency high-performance distributed solutions. Writing the required device drivers and control software to initialize and manage memory mapped interconnects from scratch is a significant job and requires significant system and interconnect hardware experience and knowledge.

The goal of the SISCI API is create a portable software stack that which by design do not add significant overhead to data transfers operations and significantly simplifies the use of advanced remote memory access networks.

For an application developer, the SISCI API represents a rich interface to memory mapped hardware and lower-level software services.

The SISCI API

The SISCI 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 “memory mapped” interconnect, initially for the SCI (Scalable Coherent Interface).

The SISCI API supports data transfers based on either distributed remote memory access or on Direct Memory Access (DMA). It also allows to trigger remote interrupts and to catch and handle events generated by the underlying interconnect system (such as a cable being unplugged). The SISCI API has proven portable and very valuable for a number of various types of memory mapped interconnect solutions beyond SCI.

The SISCI API is currently available for the following interconnects:

- SCI (initially with a SBus and PCI interface),
- DX (Based on ASI / StarGen PCI Express Gen1)
- IX (Based on IDT PCI Express Gen2 chips)
- PX (Based on PLX PCI Express Gen3 chips)

System security

The regular SISCI API functionality and implementation are designed to create an easy to use and safe environment. The functionality is implemented in close interaction with standard IOMMU functionality and lower level drivers to prevent malfunctioning software (e.g. software bugs) from accessing remote memory outside of exported SISCI segments. This is true also after hot-plug events and system reboots even if the customer application software is not designed to fully support such events.

Interoperability

The SISCI API is designed to be operating system independent, allowing users to write portable applications that communicates across systems without adding overhead or performance penalties.

The SISCI API supports connecting both little and big endian systems – the interpretation of data in bi-endian systems is up to the application programmer.

Resources and resource dependencies

The SISCI 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. A descriptor handle is provided to the user, which 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 a reference to an open virtual device, meaning that a local segment depends on a virtual device.
The 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.

About this guide

This document will guide you through the fundamental features of the SISCI API. At the end, you will 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 SISCI API library and querying information about the interconnect fabric. This is presented in the "System aspects" chapter starting on page 9.

You will learn basic memory management in the "Memory segments" chapter starting on page 12. This includes 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 will be ready to perform data transfers between several nodes.

Data transfer methods are described in the "Accessing memory" chapter starting on page 17 and the "DMA" chapter starting on page 23.

A memory mapped interconnect system may contain several nodes sharing a global memory structure. Interrupts may be used for synchronization. Interrupts are a fast way of notifying another node that something has occurred and you will learn how to use them in the "Interrupts" chapter starting on page 32.

Finally, the "Advanced Features" chapter starting on page 37, deals with things like managing events and checking for data transfer errors.

It is recommended to keep the SISCI API Functional Specification at hand when reading this guide. 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 SISCI API Functional Specification, other documentation and software distributions are available from http://www.dolphinics.com/.

Examples in this guide

The textual explanations are 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 summarize 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 interconnect hardware and software allow to do much more than that.

A program making use of the SISCI API library must include the header files `sisci_api.h` and `sisci_error.h`. For simplicity, this is done only in the full programs but not in code excerpts. Please refer to the documentation 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 SISCI applications.

Applications that want make use of "callbacks" (more on this later in this guide), must be compiled using the D_REENTRANT compiler flag.

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

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

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 interconnect adapter card. This guide assumes there is only one card present on each system and that the identifier is the same on all systems. A system can have several adapters, each identified with a host-wide unique adapter number.
- `SENDER_NODE_ID` and `RECEIVER_NODE_ID` are the SISCI node identifiers of the two nodes involved in the example code. Each node connected to the network is assigned a unique node identifier.
- `SENDER_SEG_ID` and `RECEIVER_SEG_ID` are the identifiers (segment IDs) of the SISCI memory segments created on the two nodes of the example code. The sizes (in bytes) of the two segments are `SENDER_SEG_SIZE` and `RECEIVER_SEG_SIZE` respectively.

Examples in the software distribution

The SISCI software is normally distributed with a number of SISCI tests, benchmarks and example programs. SISCI programmers are encouraged to study these examples and play with the benchmark tools to learn more about the SISIC API and the power of remote memory accesses.

System aspects

This chapter address some generic aspects of the SISCI API related to the initialization of the SISCI program environment and to the retrieval of information about the underlying interconnect system.

In particular, you will learn:

- How to initialize the SISCI API library.
- How to manage the so-called virtual devices.
- How to get useful information about the underlying network system.
- How to check if a remote node is on-line.

Initializing the SISCI environment

Before calling any other function in the SISCI API one should initialize 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 `SCIInitialize()` fails there is something fundamentally wrong with the local software or hardware installation. One typical problem is that the SISCI API library and driver versions are not consistent, in which case the error is `SCI_ERR_INCONSISTENT_VERSIONS`.

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

```
/* no SISCI API calls here */
SCIInitialize(...);

/* SISCI program functionality goes here */

SCITerminate();
/* no SISCI API calls here */
```

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

Virtual device handles

Most SISCI API functionality is managed through virtual device handles. A virtual device handle can be seen as a communication channel with the underlying SISCI driver.

NB: Each virtual device can only hold one of each resource. E.g., you would need two virtual devices to manage two segments, but only one virtual device to manage one interrupt and one segment.

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()`.

Querying information

The SISCI API provides a way to retrieve some information about the underlying interconnect 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 SISCI API specification for further query options.

The function used to query the above mentioned information is `SCIQuery()`. This function has a query command as input and returns the requested information. Additional input information, like subcommands, is 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 SISCI API specification for the details.

Determining the SISCI API version

The `SCIQuery()` call is in this case quite simple. The first parameter specifies the query type, which in this case is `SCI_Q_API. Results are stored in the` sci_query_string structure, referenced in the second paramter. 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 requested value */
} else {
       /* manage error */
}
```

Determining the node identifier of a certain adapter

`SCIQuery()` requires additional parameters to get the proper information for a certain adapter: a subcommand specifying the query type 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 requested value */
} else {
        /* manage error */
}
```

Probing a remote node

The SISCI API function `SCIProbeNode()` can be used to check if communication is possible to a certain remote node through a certain local adapter:

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

SCIInitialize(...);
SCIOpen(&v_dev, ...);

reachable = SCIProbeNode(v_dev, ADAPTER_NO, REMOTE_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 or cable.

`SCI_ERR_NODE_NOT_RESPONDING`
There are problems with the remote adapter.

`SCI_ERR_NO_SUCH_NODEID`
There is no reachable node with the specified identifier on the network accessible through the specified adapter.

Memory segments

The possibility to safely access memory physically resident on another machine is the fundamental characteristic and the strength of the SISCI technology. If the remote memory is mapped in the addressable space of a local process, thus appearing as if it were local, a data transfer is as simple as a normal memcpy(). Memcopy() is typically implemented as a sequence of CPU instructions that will use CPU load or store instructions to send or fetch data from remote memory. This transfer method is known as Programmed I/O (PIO). Alternately, the SISCI Application programmer can use the Direct Memory Access (DMA) approach, whereby the CPU simply gives instructions to the network interface DMA controller about the transfer (for example source address, destination address and size), which makes it free to do other things in parallel with the data transfer.
Both cases require a way to manage local memory segments on one side and a way to attach to remote memory segments on the other side.

In this chapter, you will learn:

- 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.

Managing local segments

Allocating memory space

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 SISCI uses it has specific characteristics, such as being non-swappable and/or being physically contiguous. Different hardware implementations may have different requirements, using SCICreateSegment() ensures the implementation details can be hidden from the user and ensures portability.
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_SEG_ID,    /* segment identifier */
       RECEIVER_SEG_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 us 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 will be initialized if the call is successful. The segment identifier, which in this case is the constant `RECEIVER_SEG_ID`, is an integer that uniquely identifies the segment to the driver. A remote node that wants to use this segment needs to know the value of this identifier. The driver checks the uniqueness of the identifier, so applications should choose appropriate values. `RECEIVER_SEG_SIZE` is the size of the segment that will be allocated.

The specification of a callback allows triggering the execution of a certain function when something happens concerning this segment. This option is covered later in the "Events and callbacks" chapter, starting on page 41.
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.

Deallocating a memory segment

When a memory segment is no longer needed, 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, ...); /* initialization */
/* use the segment */

SCIRemoveSegment(segment, NO_FLAGS, error);

if (error == SCI_ERR_OK) {
       /* the resource is freed */
} else {
       /* manage error */
}
```

`SCIRemoveSegment() normally` succeeds, if not the SISCI driver will release left-over resources when the application terminates. 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.

Making a segment available

Once a segment has been allocated, it needs to be made visible to local processes or the other nodes connected to the network. This export operation is 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 network address space, shared by all the nodes in the network. What it does in practice is to make sure that the segment can be correctly accessed by the specified network 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 the specified adapter.

```
sci_error_t prepare_error;
sci_error_t avail_error;
sci_local_segment_t segment;

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

Making a segment unavailable

`SCISetSegmentUnavailable()` changes the visibility of an exported segment, preventing new remote connections 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 in not available for new remote connections */
} else {
        /* manage error */
}
```

Calling `SCISetSegmentUnavailable()` doesn't affect existing remote connections, as they are not 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 that there are ways to determine when a remote node has connected.

A state machine for a local segment resource

Figure 1 - State machine for a local segment" below shows the life of a local segment.

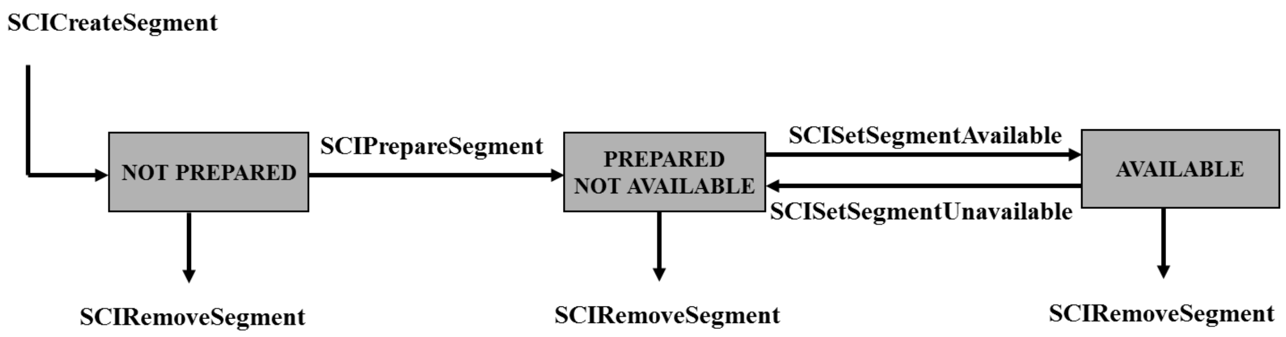

Figure 1 - State machine for a local segment

Some additional comments:

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

Managing remote segments

Imagine the following scenario: A process on the receiver node, whit node id `RECEIVER_NODE_ID`, has allocated and made a piece of memory available, with segment id equal to `RECEIVER_SEG_ID`.
The sender node has node id `SENDER_NODE_ID`. We would like to have a process that uses the memory segment on the receiver node.

It is worth clarifying some nomenclature to better understand the next sections:

- A local segment is allocated on the receiver node.
- From the sender node 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.

Connecting to a remote segment

The remote application has to "connect" to a remote segment. Logically speaking the connection process consists in finding the address and the size of the remote segment within the network 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, ...); /* initialize the virtual device */

SCIConnectSegment(v_dev,   /* virtual device */
       &remote_segment,    /* handle to the remote segment resource*/
       RECEIVER_NODE_ID,   /* remote node id */
       RECEIVER_SEG_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 us look at the list of parameters of the above call:
`v_dev` represents an initialized virtual device, needed to communicate with the driver.

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

The remote node and segment identifiers (`RECEIVER_NODE_ID` and `RECEIVER_SEG_ID` respectively) uniquely locate a memory segment on the 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. Only the first option can cause the function to return if the timeout is infinite, as shown above. The call can be made asynchronous; this behavior is described in the "Events and callbacks" section starting on page 41.

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);
```

Disconnecting from a remote segment

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

```
sci_remote_segment_t segment;
sci_error_t errror;

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.

Accessing memory

You can start using a memory segment once it is made available, which means 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 network adapter.

In this chapter, you will 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 addressed in the "DMA" chapter on page 23.

Mapping a local memory segment into addressable space

Let us 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_SEG_ID,
       RECEIVER_SEG_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_SEG_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 */
}
```

The code sample above maps the allocated segment from its beginning (`offset = 0`) and for its entire size (`size = RECEIVER_SEG_SIZE`) to a local pointer. It is possible to map only parts 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 that 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 initialized mapped segment descriptor.

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_SEG_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 */
}
```

The code sample above maps the 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_SEG_SIZE`). It is possible to only map parts 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 that 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 initialized by the call, if successful. Note that the type of remote_map is sci_map_t, which 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 optimizations of the code.

Unmapping a mapped memory segment

A mapped memory segment should be unmapped once it is no longer neede. Since the type of a local and a remote mapped segment is the same, there is a unique function, called SCIUnmapSegment(), that performs 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 errror;
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, ...);
```

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 have not correctly considered all the dependencies to the mapped segment.

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 */

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

Remote operations enables access to memory that is physically resident on another node. Having mapped access to remote memory allows very low latency, very low overhead data transfers based on simple CPU load or store operations.

Remote memory access example

It's time to summarize all the topics so far in a sort of "big picture", in order to have a proper understanding of the available features offered by the SISCI API for the use of remote 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, and 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 remote memory access

```
/* sender program */
#include "sisci_api.h"
#define RECEIVER_NODE_ID 4
#define RECEIVER_SEG_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_SEG_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, initializes the first word of that memory to something different than the print command, exports the segment into the network address 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 initialization of the first word of the memory segment is done before the segment itself is made available to other nodes to avoid a race condition – to ensure the value updated from remote is not overwritten by a late initialization.

Example 4-2. A receiver program based on remote memory access

```
/* receiver program */
#include "sisci_error.h"
#include "sisci_api.h"
#include <stdio.h>
#define RECEIVER_SEG_ID 4
#define RECEIVER_SEG_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_SEG_ID, RECEIVER_SEG_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_SEG_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-accessed 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;
}
```

DMA

DMA data transfers are typically implemented by a separate hardware transfer function on the adapter card or within the system. This allows the CPU to do something else during the transfer, though latencies and overhead usually increase because of the time required to setup and control the DMA engine. However, several data transfers can be joined and sent together to amortize the overhead. Higher layer software may want to implements it data transfer by sending small amounts of data using PIO and larger amounts of data using DMA operations.

In this chapter, you will learn:

- What a DMA queue is.
- How to manage DMA queues.
- How to transfer data with DMA.

Creating a DMA queue

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

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

SCIInitialize(...);
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 communicating with the driver.

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

`ADAPTER_NO` states on which network 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.

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 */
}
```

The DMA queue must not be used after it has been removed.

According to "Figure 2 - DMA queue states and transitions" on page 24, `SCIRemoveDMAQueue()` can only be called if the queue is either in its initial state (IDLE) or in a final one (DONE, ERROR or ABORTED). If the call succeeds, the queue exits from the state diagram.

DMA queues

A DMA queue is one of the resources specified in the SISCI 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 network adapter.

The state diagram for a DMA queue is shown in "Figure 2 - DMA queue states and transitions". A transition from one state to another is either triggered 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.

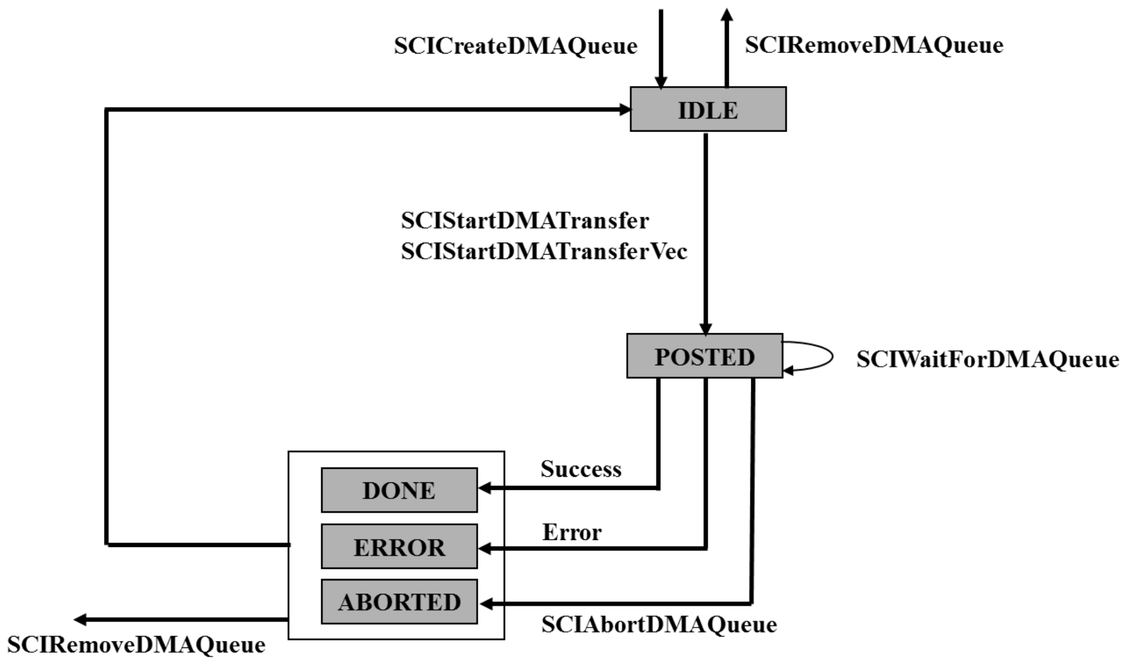

Figure 2 - DMA queue states and transitions

The meaning of the states is as follows:

IDLE
    The queue has been successfully created and it is empty.
POSTED
    The queue has been passed down to the DMA engine on the network 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.

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;

SCICreateDMAQueue(..., &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_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.

Starting a single DMA transfer

Once a DMA queue is available, you can use it to transfer data.

A data transfer specification / DMA descriptor consists of:

- A local memory segment
- A remote memory segment
- An offset within the local segment
- An offset within the remote segment
- The number of bytes to transfer

The function used for sending a single block of data is `SCIStartDMATransfer()`.

```
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, ...);

SCIStartDmaTransfer(dma_queue,
                        local_segment,
                        remote_segment,
                        local_offset,
                        size,
                        remote_offset,
                        NO_CALLBACK,
                        NULL,
                        NO_FLAGS,
                        &error);

if (error == SCI_ERR_OK) {
       /* the transfer has been correctly started */
} else {
       /* manage error */
}
```

Data is by default transferred from the local segment (starting at offset `local_offset` and for `size` bytes) to the remote segment (starting at offset `remote_offset`). To transfer data from a remote segment to a local segment, you have to specify the `SCI_FLAG_DMA_READ` flag.

The local segment must be "prepared" in order for the network 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 network adapter used.
An unprepared local segment, a not connected remote segment, invalid specification of offsets and/or size are all cause of errors (check the SISCI API specification for the details). The function call will also fail if the DMA queue already is in use (by a previous, but not completed DMA transfer).

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

Starting multiple DMA transfers

If the DMA queue consists of several elements, the function `SCIStartDMATransferVec()` must be used.

The multiple transfer elements needs to be organized into an array of descriptors where each descriptor specifies each transfer similar to the single transfer argument to `SCIStartDMATransfer()`.

```
dis_dma_vec_t dis_dma_vec[num_elements];

dis_dma_vec[i].size          = 'number of bytes in this transfer element';
dis_dma_vec[i].local_offset  = 'offset in local segment';
dis_dma_vec[i].remote_offset = 'offset in remote segment';


SCIStartDmaTransferVec(dma_queue,
                       local_segment,
                       remote_segment,
                       number_of_transfers,
                       dis_dma_vec,
                       NO_CALLBACK,
                       NULL,
                       flags,
                       &error);
```

The `SCIStartDMATransferVec()` does not wait for the transfers to complete but returns simply when the queue has been passed to the DMA engine

Waiting for DMA 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 behavior, you can just sit and wait:

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

SCICreateDMAQueue(..., &dma_queue, ...);
SCIStartDMATransfer(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 successfull or failed */
} else {
       /* manage error */
}
```

If you do not 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 also allowed 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 `SCIStartDMAQueue()`.
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;

SCICreateDMAQueue(..., &dma_queue, ...);
SCIStartDMATransfer(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 transfers have completed successfully */
              break;
       case SCI_DMAQUEUE_ERROR:
              /* less good; manage the failed transfers, e.g. restart 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 the "Events and callbacks" section starting on page 41).

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 only meaningful from the POSTED state, but it is also allowed 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, ...);
SCIStartDMATransfer(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 */
}
```

Reusing a DMA queue

A DMA queue can be reused once the transfer has completed and the state is `SCI_DMAQUEUE_IDLE, SCI_DMAQUEUE_DONE, SCI_DMAQUEUE_ABORTED` or `SCI_DMAQUEUE_ERROR`.

DMA 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 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_error.h"
#include "sisci_api.h"
#define RECEIVER_NODE_ID 4
#define RECEIVER_SEG_ID 4
#define SENDER_SEG_ID 4
#define SENDER_SEG_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 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_SEG_ID, SENDER_SEG_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 */, SENDER_SEG_SIZE,
                     0 /* address hint */, NO_FLAGS, &error);
        if (error != SCI_ERR_OK) return 1;

        /* create the DMA queue */
        SCICreateDMAQueue(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_SEG_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;

        /* Start the DMA transfer */
        SCIStartDMATransfer(dma_queue, local_segment, remote_segment,
                     0 /* local offset */, sizeof(PRINT_COMMAND) /* transfer size */,
                     0 /* remote offset */, NO_CALLBACK, NULL, 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 */
        SCIRemoveDMAQueue(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 below shows the operations performed by the receiver. Notice that the program is *identical* to Example 4-2.

Example 5-2. A receiver program based on DMA

```
/* DMA based receiver program */
#include "sisci_error.h"
#include "sisci_api.h"
#include <stdio.h>
#define RECEIVER_SEG_ID 4
#define RECEIVER_SEG_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 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_SEG_ID, RECEIVER_SEG_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_SEG_SIZE,
	             0 /* address hint */,
	             NO_FLAGS, &error);
	if (error != SCI_ERR_OK) return 1;

	/* initialize the contents of the first word of the to-be-accessed 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;
}
```

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, a SISCI interrupt is allocated on a node and it is connected to and used from another one.

Managing SISCI 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 will 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.

Managing local interrupts

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, the driver assigns an identifier to the allocated interrupt. A remote application can trigger the interrupt by using the same identifier. (You have to make the identifier known to the remote node it in some way – e.g. by transferring it through shared memory). 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`.

Example 6-2 demonstrates the use of `SCI_FLAG_FIXED_INTNO` to avoid implementing an additional mechanism to communicate the identifier to the application running on the other node.

Note that the `SCICreateInterrupt()` allocates and initializes the interrupt and makes it available to other nodes.

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 */
}
```

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. It is alternatively possible to use a callback mechanism, this is addressed in the "Interrupts and callbacks" section starting on page 48.

Managing remote interrupts

Connecting to an interrupt

Like memory segments, the application has to "connect" to a remote interrupt to be able to trigger an interrupt on a remote node. 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 driver software, in which case another mechanism must be foreseen to make this number available to other nodes.

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 */
}
```

Triggering an interrupt

A 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 */
}
```

If the application that exported the interrupt is waiting, it should then wake up and proceed its execution. If the application does not wait for interrupt, the interrupt will be lost.

Interrupt example

As in the previous examples, 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 below 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_error.h"
#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 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_error.h"
#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 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; assume 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, SCI_FLAG_FIXED_INTNO, &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;
}
```

Advanced Features

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

In this chapter, you will 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 adapter card and interconnect.
- How to use the available callback mechanism in a number of situations.

Error checking

Remote data access is more error-prone than local access. Although the network protocol is robust, an error can always happen, due for example to a cable being unplugged or a switch being power cycled. Error situations detected by the network 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 or switch over to an alternative network. The SISCI API provides the means to cope with both buffering and with data transfer errors. In both cases, the solution is based on the concept of a "sequence".

Sequences

A sequence is a resource associated to a remote mapped segment, which allows error checking in data transfers 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 to a remote 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 that is allocated and initialized by the function call. 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 */
}
```

Flushing buffers

As mentioned above one may want to flush write buffers to make sure the data transfer has started to the remote memory. `SCIFlush()` is provided to clear any write buffers, the data may still be in transit to the remote memory when the function returns:

```
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; /* writes the value 1 to remote memory */
SCIFlush(sequence);
```

`SCIStoreBarrier()` is used to empty the write buffers and force any transfer packets to be transmitted and wait for its completion:

```
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 when 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 unless there has been a transmission error. You should use SCICheckSequence() if you want to include a check for transmission errors.

Note that `SCIFlush()` is a local-only operation, affecting only write buffers on the local adapter card, while `SCIStoreBarrier()` usually causes some data transmission and therefore some interaction with remote nodes. SCIStoreBarrier() and SCICheckSequence() are relatively costly operations and you may want to design your application to not use these extensively.

Checking for data transfer errors

As sequence is also used to check for errors during data transfers. A sequence needs to be cleared before starting to move data. This means 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()`, if successful, whether 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 code below demonstrates the use of `SCIStartSequence()` before the data transfer:

```
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 by either `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 intrconnect 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 perform the flush but not wait for the completion of all the outstanding network 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.

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

1. Start the sequence.
2. Transfer the data.
3. Check the sequence.

a) It is ok continue.

b) If it is not ok and the error is not fatal, retry the transfer and check the sequence.
c) If it is 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 your application does not 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 occurence 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 reliable communication, the only difference being that you now do not 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 the error flags would not be in a clean state at the beginning of the next data transfer otherwise.

Events and callbacks

On certain conditions a component of the network, 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 etc.

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

Local segment events

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 may have rebooted).

All these events are passed to the application, which may wish to intercept them. All the resources depending on a local segment should be freed before the segment itself could be removed. Some of these 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.

Local segment synchronous event handling

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, ...);
/* The segment is now 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 */
}
```

The code sample above makes a local segment available to remote nodes and waits for an event concerning the segment. The call to `SCIWaitForLocalSegmentEvent()` provides 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 when the segment has been removed (`error` is set to `SCI_ERR_CANCELLED`).

Local segment asynchronous event handling

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 the "Managing local segment" section on page 12:

```
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_SEG_ID, /* segment identifier */
             RECEIVER_SEG_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 as 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 function gets the same information as is available after a `SCIWaitForLocalSegmentEvent() call`.

Note how the value of the `flags` parameter 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;
}
```

Remote segment events

The same five events related to local memory segments also applies to remote segments, though their meaning is different. Let us 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 is:

`SCI_CB_CONNECT`
: An asynchronous connection (see the "Managing remote segment" section on page 15) has completed successfully.

`SCI_CB_DISCONNECT`
: An asynchronous connection (see the "Managing remote segment" section on page 15) 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 (again).

`SCI_CB_LOST`
: An unrecoverable situation has occurred on the exporting node.

Asynchronous connection

In the "Connecting to a remote segment" section on page 15 we have seen how to connect to a remote segment in a synchronous way, that is, we wait until `SCIConnectSegment()` returns, either because the connection has completed or because a timeout has expired.

Alternatively, it is 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 does not 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 the "Disconnecting from a remote segment" section on page 16).

Remote segment synchronous event handling

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

Let us 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 succeded;
                     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);
                     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()`.

Remote segment asynchronous event handling

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 the "Connecting to a remote segment" section on page 15:

```
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_SEG_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 function gets the same information as is available after `SCIWaitForRemoteSegmentEvent()`.

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_sement_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;
}
```

A state machine for a remote segment

The remote segment callback state machine can be found in the Figure 3 – State machine for remote segment callbacks below.

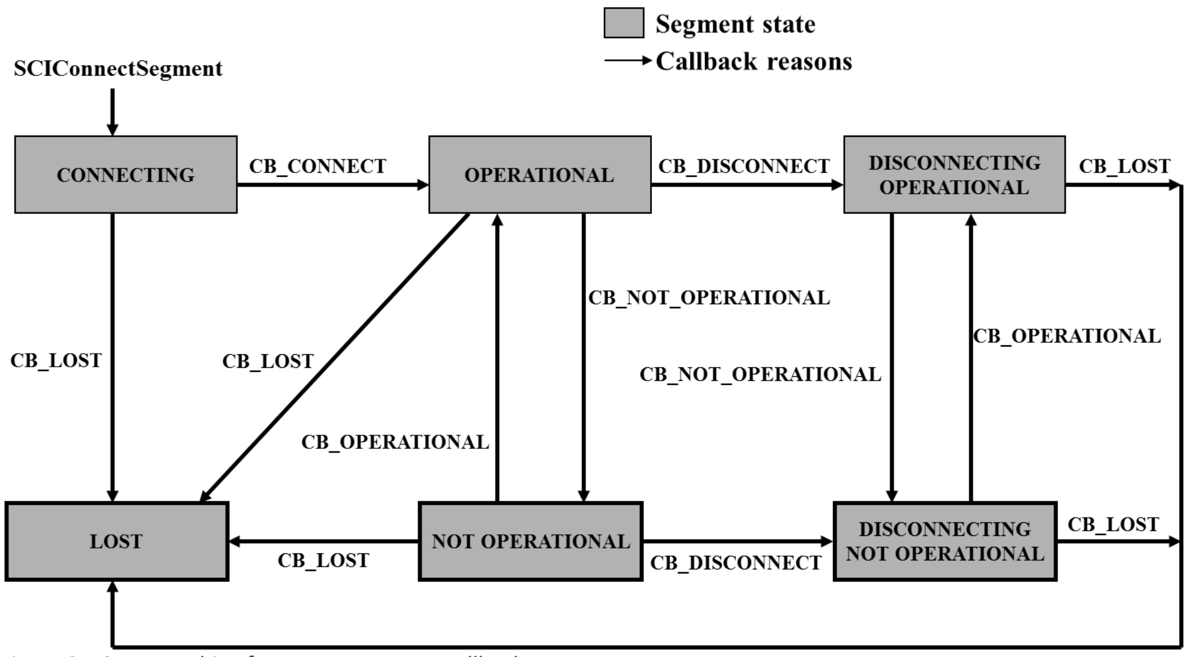

Figure 3 – State machine for remote segment callbacks

DMA and callbacks

In the "Waiting for DMA completion" section on page 26 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 will learn how to use a third way, which consists on a callback mechanism: you declare that a certain function is to be called when the DMA queue processing has completed. 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;

SCICreateDMAQueue(..., &dma_queue, ...);
SCIStartDMATransfer(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 set `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()`.

Interrupts and callbacks

In the "Waiting for an interrupt" section on page 33 we have seen that interrupts can be caught synchronously calling `SCIWaitForInterrupt()`. Alternatively, you can specify a function to be called asynchronously when an interrupt arrives, on creation time.

```
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`.

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()`.

Reflective memory – Multicast

Certain types of memory mapped interconnect solutions provides a hardware based multicast environment. Multicast functionality enables a single write transaction to reach multiple targets. A CPU store, DMA write or some other device that can do a posted write to the mapped multicast address can typically initiate the transaction.

Dolphin has extended the SISCI API to support multicast/reflective memory, currently available with the Dolphin Express DX and IX interconnect family. The multicast functionality is normally only available when your systems are connected using a network switch (The multicast is normally implemented in the switch).

The initialization of a multicast segment is very similar to initialization of regular remote segments;

The segment Id is used to specify the reflective memory Id and must be between 0 and max number of reflective memory segments supported by the hardware (max 6 for Dolphin Express DX, max 4 for Dolphin Express IX).

To allocate, prepare or map a segment, the `SCI_FLAG_BROADCAST` flag must be set. You should use `DIS_BROADCAST_NODEID_GROUP_ALL` as nodeId, for remote operations that needs the remote nodeId as parameter.

The important functionality and parameters are:

```
nodeId = DIS_BROADCAST_NODEID_GROUP_ALL;

SCICreateSegment( …, SCI_FLAG_BROADCAST, …);

SCIPrepareSegment(…, SCI_FLAG_BROADCAST, …);
```

```
        SCIConnectSegment(…, nodeId , SCI_FLAG_BROADCAST, …);
```

More information on multicast

The SISCI software distribution from Dolphin contains several source code examples demonstrating the real use of the SISCI reflective / Multicast functionality.

More details on reflective memory can be found in a separate white paper that can be downloaded from the from the whitepaper section on the Dolphin web site.

Peer to Peer transfers

The SISCI API supports registering of arbitrary physical addresses as SISIC segments and identifying physical addresses of remote mapped devices.

More information on peer to peer transfers

More details on Peer to peer communication can be found in a separate white paper that can be downloaded from the whitepaper section on the Dolphin web site.

Privileged functions

The SISCI API contains some functions that may cause severe system problems, corrupting data or causing crashes if used incorrectly.

Registering physical memory as a segment

The SISCI API also allows physical memory e.g. memory on a PCIe device (FPGA, GPU etc.) to be registered as a SISCI segment. The requirements are that the user knows the physical address and its size, and that the memory is always available (Main memory needs to be locked down).

If registering a PCI/PCIe device, the user also needs to ensure the local system supports direct PCI Express peer to peer transfers (The network card are able to directly access the physical memory device).

The concept is to allocate an "empty" segment – without any memory allocated. This is done by specifying the flag `SCI_FLAG_EMPTY` when calling `SCICreateSegment`() and later on use the `SCIAttachPhysicalMemory`() function to register the physical address.

The following code will demonstrate how to do this:

```
sci_error_t error;
sci_local_segment_t segment;
sci_ioaddr_t   ioa = 0x1234567800000000;  /* 64 bit physical address of memory */

SCICreateSegment(… , SCI_FLAG_EMPTY, …); /* initialization */
If (error == SCI_ERR_OK) {
       SCIAttachPhysicalMemory(ioa, NULL, 0, size, segment, 0, &error);
       if(error != SCI_ERR_OK) {
              /* Terminate or handle error */
       }
 }
/* from this point we can use it as a regular segment */
```

Once the physical segment is properly registered and set up, it is managed and used just as a regular SISCI segment. A full example of how to work with physical segments can be found in the SISCI source code example `rpcia.c`

Querying the remote address of a mapped SISCI segment

If you would like to set up a local FPGA, GPU etc to access a remote SISCI segment directly, you need to identify its local physical address. This address will typically reside within the network adapters address space and will only be valid as long as the remote segment is and remains mapped from this system. The following code can be used to identify this address:

```
query.subcommand = SCI_Q_REMOTE_SEGMENT_IOADDR;
query.segment = segment;

SCIQuery(SCI_Q_REMOTE_SEGMENT, &query, 0, &error);
if(error == SCI_ERR_OK) {
       printf("Remote segment ioaddr = 0x%0*llX\n", 16, query.data.ioaddr);
}
```

It is up to the programmer to find a good way to propagate this address down to the device that wants to use this address.

How to write optimized SISCI code

The SISCI API provides a rich powerful API for deploying applications over a remote memory access network. This section contains suggestions and ideas for application programmers to optimize performance. Which one to use depends on the nature of the application and system? Please consider the following:

1. Most of the SISCI API setup and management functions are either communicating with remote systems or doing local adapter CSR register accesses. These functions should normally not be used in the “application performance critical inner loop”. Allocating memory segments, connecting to remote segments, creating interrupts etc. should ideally be established when the application starts and removed before application exit.
2. Avoid interrupts when not needed. The SISCI API offers a set of functions to create and trigger remote interrupts. The SISCI driver will use remote memory access internally to trigger a remote interrupt, but the system interrupt latency and overhead will be added to the time budget.
   - 2.1. Use polling to avoid interrupts: Polling a local SISCI memory address is very low cost operation. The first reference will re-load the local CPU cache, all following poll operations will only reference the cache until the remote update arrives and the CPU cache is invalidated and fetched. It may be possible to do some polling between other application tasks?
   - 2.2. Combine polling and interrupts. Poll for a short while before going to sleep – waiting for a remote interrupt.
   - 2.3. Use remote memory access to inform the remote node that an interrupt is required to wake up the other process.
   - 2.4. `SCICreateInterrupt()` should be called once and not for every loop.
3. Optimized error checking. The `SCICheckSequence()` operation typically require 1-2us to complete. The Dolphin Express IX network is a reliable network and data will not be lost unless there is a real problem – broken hardware, cable unplugged, switch power down etc.
   - 3.1. Try to move the `SCICheckSequence()` out of the communication inner loop, maybe the application can implement a more relaxed error recover model ?
   - 3.2. Do not use `SCICheckSequence()` for small messages – implement a memory checksum mechanism where the sender calculates a checksum that can be verified by the receiver.
4. Use PIO as alternative to DMA. Transferring small amount of data using DMA has a lot of overhead compared to a PIO posted write. Small transfers will in most cases benefit from using PIO. DMA uses less CPU overhead for large transfers.
   - 4.1. Implement a mixed transport model using PIO for small transfers and DMA for larger transfers.
   - 4.2. If you are doing direct remote writes using pointer arithmetic, try to optimize the size written, send one 8 byte (dword) instead of many chars etc.
   - 4.3. The use of `SCIMemCpy()` will ensure optimal performance, especially if large amounts of data are copied to a remote system. If `SCIMemCpy()` is not desired, then flush the CPU buffers with `SCICheckSequence(NULL, SCI_FLAG_FLUSH_CPU_BUFFERS_ONLY)` or `SCIFlush()`.
     If balancing between PIO and DMA is implemented, do not assume that the balance will be kept across heterogeneous hardware. Thresholds do change with CPU and IO chipsets.
5. Implement a way to benchmark your transport and compare the results with the standard Dolphin SISCI benchmarks found in the software distribution (scibench2, scipp, intr_bench , dma_bench and reflective_bench). Review the code of the Dolphin benchmarks if you do not reach similar performance.
   - 5.1. Request support from Dolphin if you do not reach your expected results.

Using Dolphin Express IX hardware, data can be transmitted to remote memory in 0.74us. Message passing at the SISCI API can be implemented using the above listed techniques in 1us – 1.2 us latency for 1 byte.

Availability

Dolphin has been providing the SISCI API for all its Dolphin Express interconnect solutions. Support for SCI networks was introduced in 1998. Support for the Dolphin Express DX was introduced in 2006. Support for Dolphin Express IX was added in 2010.

Supported operating systems

SISCI support was introduced on Solaris 2.5.1, LynxOS 3.1, VxWorks 5.5, Windows NT 4.0 and Linux 2.0.

SISCI is currently available for Windows XP – Windows 8.1 32 and 64 bits, RTX20012 and RTX2013 - 32 and 64 bits, Linux 2.4, 2.6 and 3.x 32 and 64 bits. The software is currently being enabled on VxWorks 6.9 + 7.0.

Please consult the appropriate software release note for details on supported operating systems and versions.

Support for 3rd party hardware

The SISCI API is also available for Logan, Northstar and Phoenix PCI Express chipsets from IDT and NTB enabled PCI Express Gen2 and Gen3 chips from PLX. Please contact Dolphin if you are interested in running SISCI on your custom PCI Express design.

More information

Please visit http://dolphinics.com/products/embedded-sisci-developers-kit.html for an overview of available resources.

The SISCI functional specification can be found at http://ww.dolphinics.no/download/SISCI_DOC/index.html

Please contact Dolphin support on pci-support@dolphinics.com if you have any questions on using the SISCI API.