SCI Physical Layer API


This paper describes the architecture of the SCI physical layer API standard. This software standard defines all functionality of a DMA capable shared memory interface. Although it is designed for SCI, its use is not limited to SCI. This work is the result of the IEEE P1596.9 working group. The latest P1596.9 draft is in balloting now.

SCI, API, Shared Memory, Address Translation

I. INTRODUCTION

This standard defines a standard C-language API for the IEEE Std. 1596 for Scalable Coherent Interface (SCI PHY-API). This API accommodates a broad range of SCI interface implementations by allowing software to transparently implement functionality that is not provided by the particular hardware. This standard supports packet mode and, without unnecessary overhead, transparent mode hardware. Automatic address mapping from the local address to an SCI address is supported.

There are many possible SCI implementations, with different trade-offs between direct hardware implementation and software emulation of features. A standard SCI PHY-API will allow the writing of SCI application software without knowing the specifics of the given physical SCI interface. For example, upgrading from packet mode SCI interface hardware to transparent mode hardware should be a trivial software change if the SCI driver API is well defined. A standard physical-level SCI API essentially adds a software layer where needed to give an SCI hardware interface a standard appearance to user-level software. This makes it very easy to interchange any SCI hardware in a given system and will improve software portability between systems.

The scope of this standard is to define a C-based SCI hardware abstraction software layer. It provides functionality to execute any defined SCI transaction and to access any known resource within an SCI network by providing the necessary setup and control procedures.

Those procedures allow direct access to the CSR space of local SCI interface chips. All remote nodes can be accessed across the SCI network using the IEEE 1212 CSR registers [2].

The SCI physical layer API does not impose any restrictions on the possible SCI topology. It merely provides an SCI access method that is SCI interface independent. It does not define an SCI global address resolution protocol (a method for finding, for example, a given shared memory resource within the system based on its unique shared memory ID without any knowledge of the system’s architecture). However, it defines the necessary building blocks allowing the implementation of such a protocol. For example, an optional set of procedures defines a standard API of reliable message queues.

The SCI Physical layer API is designed to serve as the foundation for higher-level software layers such as MPI, PVM, VIA, I2O, thus facilitating their portability as well.

II. SUPPORTED TRANSACTION TYPES

SCI PHY-API defines three classes of transactions as outlined in table 1. All of them include a callback procedure available for attention or exception handling. Depending on the transaction the attention handler can be used for asynchronous completion and error notification. Block transactions allow large blocks of data to be moved by one call. They are always posted. Upon completion of the block transaction the callback procedure is executed with the appropriate completion status descriptor. With chain-mode DMA transactions it is possible that, depending on the chain descriptors, the callback procedure is called multiple times during the DMA execution, thus allowing asynchronous notification to the calling software layer about the completion of a certain part of the block move.

Figure 1 shows the program flow chart of posted block transactions. Very little time is spent setting up the block transaction, which then executes in parallel with the continued program execution. The asynchronous execution of the callback procedure, which may be called once or several times during the execution of the block transaction,
is used for synchronization with the parent thread. The exact synchronization mechanism executed within the callback procedure is operating-system-dependent and outside the scope of this standard.

Figure 1: Flow chart of block or posted transactions

The second class of transactions, listed in Figure 2:, are selected byte transactions. They are typically used to read small amounts of data (1...16 bytes). Block transactions are different primarily because selected byte transactions are executed synchronously. The only exceptions are selected byte write transactions, which may be posted. Figure 2 shows a sketch of a selected byte read transaction. Since the transaction returns data it cannot indicate an error condition. Therefore an SCI attention handler is executed in case of read errors. Note: It is possible to specify a transaction-specific attention handler and transaction attribute field.

Figure 2: Flow chart of selected byte and transparent read transactions

Transparent transactions differ from selected byte transactions only by their implementation if transparent-transaction-capable SCI hardware is available. In this case SCI transactions can be directly executed by reading/writing from/to the appropriate window addresses. Transparent transactions are designed so that they can be mapped directly to assignment statements as illustrated in Table 2, thus avoiding any driver overhead. Due to the nature of the assignment statement no further options or parameters can be passed. In order to allow various transaction sizes a set of transparent transactions is defined to accommodate all common word sizes. The transaction attributes are the default window attributes.

Table 1: An overview of the major transaction types supported by SCI-PHY API

Table 2: A transparent transaction call translated into an assignment statement.

III. SHARED MEMORY TRANSACTION

Although not part of the SCI specification, the SCI physical layer API specifies a set of procedures that allow creation, mapping, and removal of global shared memory objects. Those objects differ with respect to their attributes from generic shared memory segments. For example in order to guarantee access to that local memory resource to any given SCI host at any point in time, the shared memory has to be either locked in physical memory or virtual DMA needs to be implemented.

Once a global shared memory segment has been created properly within the memory of a given node it can be mapped into the address space of any process running on that node. The shared memory is identified by virtue of its unique memory ID. The SCI physical layer API allows definition of system specific local access control attributes for that global shared memory based on another local shared memory, which is used as reference. Access control for SCI requestors is implemented by allowing to restrict access to known, trusted SCI hosts.

In order to make the shared memory visible to other SCI nodes it needs to be exported into the SCI global address...
space. There are no restrictions as to where a given local shared memory may be mapped in the SCI transport space.

The SCI Physical layer API supports remote access to local I/O resources such as CSR registers and the like. This is implemented by first defining the given I/O window as shared memory region with an appropriate unique memory ID at a specified I/O address. Now the defined local I/O address region can be exported into the SCI transport address space based on its memory ID like a shared memory.

IV. ADDRESS TRANSLATION

The local address space of any given host processor, including its virtual memory management, does not necessarily allow direct one-to-one mapping of the 64-bit SCI address space to the local address space. Especially when host systems support only 32-bit addressing, some address translation is needed. There are many different address translation models already defined. In order to be flexible enough to accommodate any of them SCI PHY-API makes very few assumptions with respect to address translation. The only assumption made is that the address translation functionality allows mapping of a given contiguous memory segment to another memory address. The size of that segment is not constrained. All local addresses (APIadr) are within the default (virtual) address space. Any necessary local virtual/physical address translation has to be handled within the driver.

Remote nodes can gain access to the shared memory by mapping the appropriate SCI window into their local (virtual) address space as indicated in Figure 3.

Remote access to local I/O regions is implemented similar to the mapping of a shared memory. For a remote node there is no difference whether it accesses a shared memory or an I/O segment.

In order to be able to map a remote shared memory a given node needs to know the SCI address of that object, which is identified by its unique shared memory ID. There are two principal different ways to implement this.

1. The shared memory is exported to a predefined address within the SCI address space, and the given static SCI address mapping is made available to all nodes in the system requiring access.

2. The SCI address of any known shared memory is dynamically resolved through an address resolution protocol based on the shared memory ID. The SCI Physical Layer API supports an address resolution message passing protocol by defining the appropriate request and response calls. However it does not define the detailed protocol.

V. TRANSPARENT TRANSACTIONS

Transparent transactions are defined as read/write transactions in the local physical or virtual address space that are transparently translated into the appropriate SCI request packets.

Transparent transactions may result in SCI activity with zero software overhead, depending on the given address mapping, if the underlying SCI interface hardware supports transparent transactions. If it does not support that functionality the SCI PHY-API compliant driver will add appropriate software wherever necessary. Therefore transparent transactions are defined like functions or procedures. If transparent transaction capable hardware is available, however, these functions and procedures can be redefined as assignment statements in the include file SCIPhyAPI.h as shown in table 2. Due to this implementation duality concept it is not possible to define further arguments like the object size. Therefore there is a

Figure 3: A sketch of the address mapping scheme

Figure 3 sketches the address model. Four processes on two separate CPUs connect to the same global shared memory residing on processor 2. Note the shared memory may be fragmented in physical address space as indicated in Figure 3. All processes map the complete shared memory into their address space. The address mapping scenario is setup by first creating the global shared memory. Once it exists it can be mapped by all local processes by virtue of its unique shared memory ID. The shared memory is made accessible to other nodes by making it visible in the SCI address space. This is accomplished by exporting it into the SCI address space. There is no restriction on where the shared memory may reside. P1596.9 recommends to export a given shared memory into one contiguous window in the SCI transport space, thus hiding any potential local fragmentation.

In order to be able to map a remote shared memory a given node needs to know the SCI address of that object, which is identified by its unique shared memory ID. There are two principal different ways to implement this.

1. The shared memory is exported to a predefined address within the SCI address space, and the given static SCI address mapping is made available to all nodes in the system requiring access.

2. The SCI address of any known shared memory is dynamically resolved through an address resolution protocol based on the shared memory ID. The SCI Physical Layer API supports an address resolution message passing protocol by defining the appropriate request and response calls. However it does not define the detailed protocol.

V. TRANSPARENT TRANSACTIONS

Transparent transactions are defined as read/write transactions in the local physical or virtual address space that are transparently translated into the appropriate SCI request packets.

Transparent transactions may result in SCI activity with zero software overhead, depending on the given address mapping, if the underlying SCI interface hardware supports transparent transactions. If it does not support that functionality the SCI PHY-API compliant driver will add appropriate software wherever necessary. Therefore transparent transactions are defined like functions or procedures. If transparent transaction capable hardware is available, however, these functions and procedures can be redefined as assignment statements in the include file SCIPhyAPI.h as shown in table 2. Due to this implementation duality concept it is not possible to define further arguments like the object size. Therefore there is a
specific function and procedure call defined for each data type. All transaction attributes are defined by the appropriate default window attributes. If the transaction causes an exception, a call back procedure is executed with the appropriate context structure.

VI. LOCK TRANSACTIONS

This standard supports subroutines that perform SCI atomic transactions. There is a set of commands available supporting the full SCI atomic transaction set including atomic-add, compare-and-swap and mask-and-swap.

VII. BLOCK TRANSACTIONS

In order to support complex data movements including but not limited to scatter-gather scenarios, the SCI PHY-API supports chain mode DMA functionality. Extensive studies on chain mode DMA functionality have been carried out by the P1285 [5] working group. The DMA architecture defined in this standard uses the P1285 concept, but removes specifics with respect to the mass storage frame work.

![DMA Root Pointer](HEAD)

![STATUS_LIST](HEAD)

![command](command)

![command](command)

![command](command)

![JUMP](JUMP)

![status](status)

![status](status)

![status](status)

Figure 4: A sketch of the chain mode DMA architecture

Figure 4 shows a sketch of the overall DMA chain list architecture. All chain descriptors are adjacent in memory, forming a DMA chained command list block. There is, however, a command allowing implementation of several disjoint chained command list blocks. The number of those blocks or the number of chain command descriptors is only limited by the amount of available memory. The end of a chained DMA list is defined by a specific command.

VIII. MESSAGE PASSING TRANSACTIONS

The SCI protocol does not implement message queues. This standard defines an optional message queuing and signaling procedure set. It allows any process in the network to interrupt any other process provided the appropriate setup procedures have been executed. If an asynchronous message is received for a listening process, the appropriate call-back procedure is executed with the received message as argument.

The discussed messaging protocol requires a message queue to be implemented in hardware or software. All required flow-control that prevents messages from being lost (FiFo overflow) has to be taken care of by the SCI interface and SCI PHY-API. If a given SCI interface does not support that feature it can be easily added later. There needs to be a FiFo somewhere in the local address space. If an SCI address window is defined accordingly all SCI write transactions will write to that FiFo. If the FiFo full status produces an appropriate error message to the SCI requester causing it to retry the message send, the handshake is complete.

SCI PHY-API defines a specific CSR address within the standard CSR address space [2]. Any packet writes to that address are stored in a message queue (FiFo). If the FiFo is full RESP_CONFLICT [3] is returned, requesting the message to be resent. A message is a fixed length data block (16, 64 or 256 bytes). Any other data size is an error.

There may be many different processes listening to their private message stream. The message port defined by the first four bytes of the message allow to route a given message within the target system. Each listening process defines a message port. Messages being received with that message port are routed to the defined process by calling its specified call-back procedure.
IX. Cache Transactions

An optional support level of SCI PHY-API is the SCI [3] cache support. There are three levels of cache support defined. Which implementation level is available is transparent to the higher level software. If cache support is implemented, all cache support procedures have to be implemented. However, if one of the cache support subroutines is not supported by the hardware, it needs to be emulated.

All cache related subroutines identify a cache line with respect to its mapped virtual or physical address APIadr. The SCI address window APIadr pointing to that window has to have the cacheable attribute set. The cache line size can be determined by calling a defined procedure.

X. Bibliography

1. ANSI X3.159-1989, Programming Language-C