SerialLite Protocol Overview

Introduction

SerialLite is a lightweight, point-to-point serial protocol suitable for both packet and streaming data applications. It has the advantages of low protocol overhead, low gate count, and minimal data transfer latency. SerialLite defines packet encapsulation at the link layer, and data encoding at the physical layer. Physical layer encoding is based on the familiar XAUI specification. It provides a protocol that integrates transparently with existing networks without software support.

This paper provides an overview of the protocol, describing its relationship to other serial standards and its basic functionality. For complete details of the SerialLite protocol, please refer to SerialLite Protocol Specification, available on www.SerialLite.org.

SerialLite Highlights

A high-level block diagram of SerialLite is shown in Figure 1. SerialLite has the following features:

- 622 Mbps to 3.125 Gbps per lane
- 8B/10B physical layer encoding
- Support for streaming or packet-based data
- Support for two user packet types: data packet and priority packet
- Nesting of time-critical priority packets within regular data packets
- Unrestricted data packet size
- Priority packet size up to 256 bytes
- Optional lane polarity reversal
- Optional lane order reversal
- Optional packet integrity protection using CRC-32 or CRC-16
- Error detection
- Optional packet retransmit on error
- Optional flow control
- Low protocol overhead and logic usage
- Low point-to-point transfer latency
- No inter-frame gaps required

Figure 1. High-Level Block Diagram
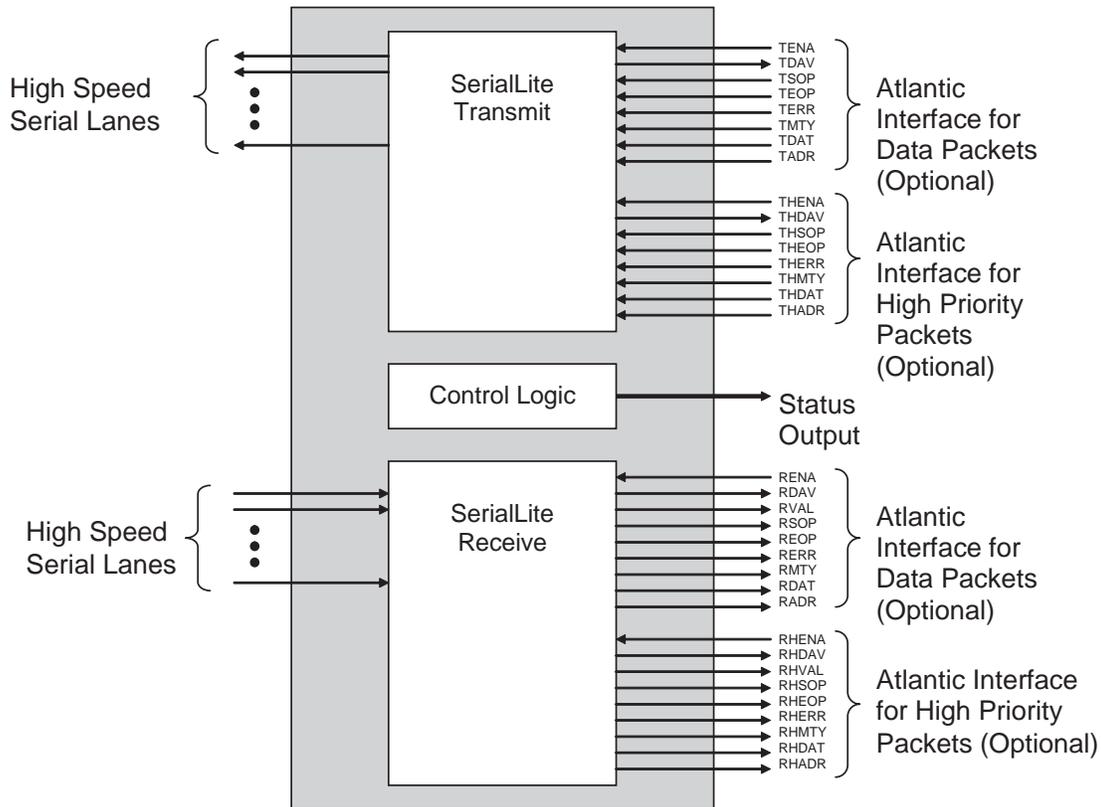
Atlantic Interface

The Atlantic interface is a synchronous, point-to-point protocol for transferring data. It has high throughput and provides flexible flow control. The *Atlantic Interface Functional Specification*, available at www.SerialLite.org, has complete details on the interface.

The Atlantic Interface as configured for use with SerialLite can have two interfaces: one for regular data and one for priority data. They are both optional, although a minimum of one interface must be provided. The interfaces at the near and far ends of the transmission link must be configured the same way.

Applications

Typical applications for SerialLite include:

- Packet or streaming data applications
- Chip-to-chip connectivity
- Board-to-board connectivity
- Shelf-to-shelf connectivity
- Simple backplane communication

Comparison to Other Serial Protocols

As a high-speed serial communication protocol, it is useful to compare SerialLite with other serial protocols, the highest-profile of which are Serial RapidIO and PCI Express. These protocols and others like them address applications that are different from those targeted by SerialLite. They are intended to be heavyweight protocols for supporting large-scale networks or meshes of inter-operating systems, where the choice of systems that may inter-operate may be made by an end user. These requirements result in the following characteristics:

- Features are defined at a high level, and require either a very rich set of features, or an elaborate scheme for managing options to allow seamless inter-operation. Software components are typically a part of the solution.
- Because of the existence of switches in the network, and the fact that data is intended to cross those switches to its intended destination, the protocols must include addressing information in the headers, leading to relatively complex framing and encapsulation.

Getting data from point A to point B

Unlike heavyweight protocols, the SerialLite protocol is intended to solve the simple problem of getting data from one place to another. The system designer is responsible for ensuring inter-operability of optional features. No addressing of the data is required since the protocol does not route data. This means both that the protocol can be streamlined significantly, and that many features can be made optional. The existence of IP cores implementing the SerialLite protocol relieves the designer from the significant details involved in designing a serial scheme by hand.

These characteristics make the SerialLite protocol particularly well suited to implementation in FPGAs, where the designer has complete control over options and can compile out any unneeded logic at design time. However, the relatively small amount of circuitry required to implement even a full version of the protocol suggests that there may be applications where off-the-shelf devices or ASICs can also provide SerialLite I/Os.

Table 1 below summarizes various contemporary serial protocols.

Table 1. Comparison of Serial Protocols

Feature	SerialLite	Serial RapidIO	PCI Express	10G Ethernet (XAUI)	InfiniBand	CSIX over PICMG	Notes
Layers above data link layer required		X	X	X	X	X	
Switching support required		X	X	X	X	X	Indicates required address fields
Arbitrary number of lanes	X						More than just a few discrete levels
Arbitrary frequency	X						More than just a few discrete levels
Lane polarity reversal	O				X		
Lane order reversal	O		X		X		
Packet data	O	X	X	X	X	X	Data that comes in discrete chunks
Unlimited packet size	X						
Streaming data	O						Data in an unending stream
Synchronous operation	O						Tx and Rx on same crystal
Asynchronous operation	O			X	X		Tx and Rx on different crystals
Priority data	O	X			X	X	
No data error detection	O						
CRC-16	O	X	X		X		
CRC-32	O			X	X		
Channel multiplexing	O	X	X	X	X	X	See note 1 below
Flow control	O	X	X	X	X	X	Pause or XON/XOFF
Retry on error	O	X	X		X		

X = required behavior
O = optional behavior

Notes to Table 1:
 (1) Because destination addressing can be thought of as a more elaborate form of channel multiplexing, all protocols supporting switching addresses can theoretically support channel multiplexing if the feature is so utilized.

Functional Description

SerialLite defines packet encapsulation at the link layer, and data encoding at the physical layer. Table 2 below details SerialLite function relative to each layer.

Table 2. Physical Layer and Link Layer Functions

Physical Layer	
Transmit	**Receive**
Parallel to Serial Conversion 8B/10B Encoding IDLE Character Conversion Insert Clock Compensation Characters Link initialization	Serial to Parallel Conversion 8B/10B Decoding Lane Alignment Character Alignment using Comma control symbol Check for running disparity error and invalid character error Clock Tolerance Compensation Link initialization
Link Layer	
Transmit	**Receive**
Packet Encapsulation Packet Nesting IDLE Character Generation Flow Control (Optional) CRC Generation (Optional) Lane Striping for multi-lane link	Remove Packet Encapsulation Separate Nested Packets IDLE Character Deletion Verification (Optional) Generate Flow Control Commands Error handling Faulty packet retransmit request commands (Optional)

Link Configuration

A serial link can be composed of one or more lanes. Each lane is a full-duplex connection between a pair of transceivers. The transceivers on one side of the link are jointly referred to as a port. This is illustrated in Figure 2.
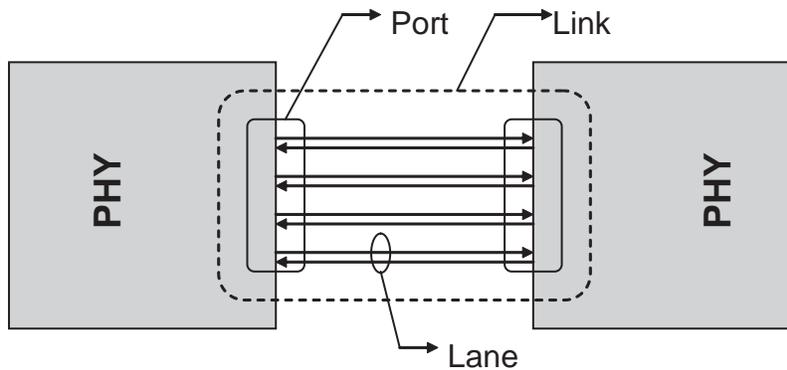
Figure 2. Link Configuration

Figure 3 illustrates the high-level data path for both the transmit and the receive direction of a single lane. Everything within the figure is specified in the protocol, and any compliant IP core that implements the protocol in a device with serial transceivers ensures that all of the details are managed without intervention by the designer.

Figure 3. High-Level SerialLite Data Path
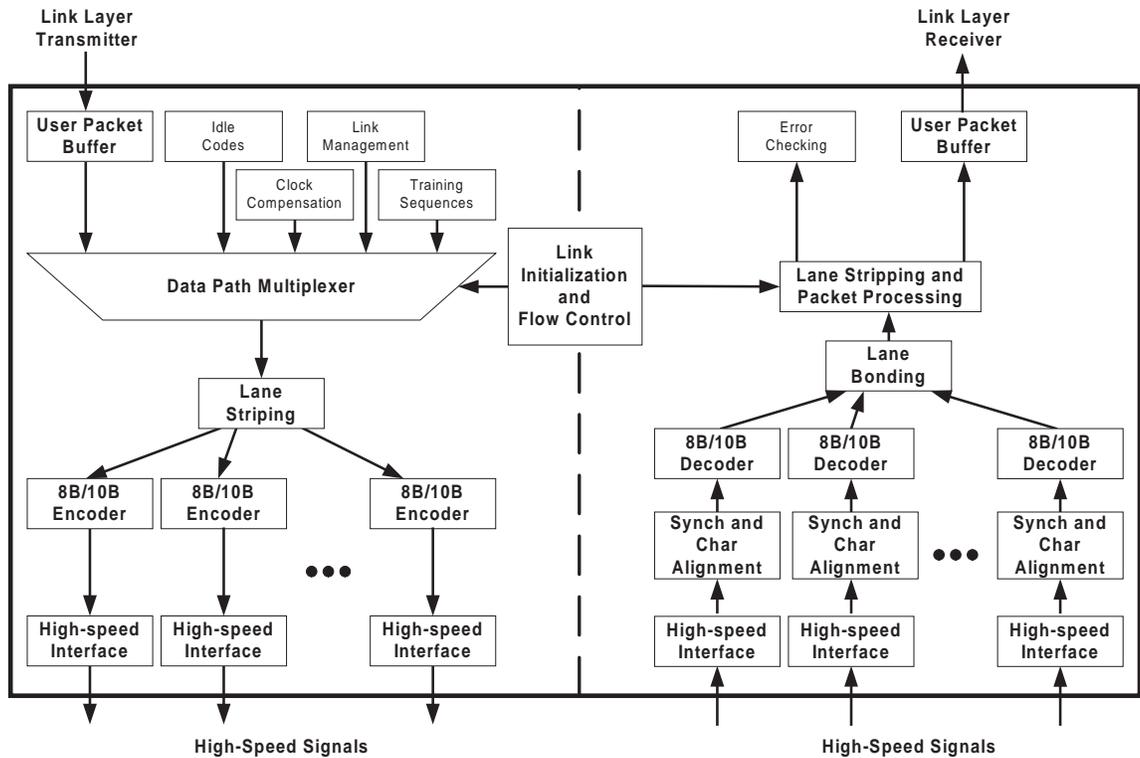
Data Encoding

SerialLite encodes physical lanes using the industry-standard 8B/10B encoding scheme as specified in Clause 36 of IEEE 802.3-2002 Specification. This ensures that a clock signal can be successfully transmitted with the data, and that the clock can be recovered reliably at the receiver.

Synchronous and Asynchronous Operation

If the transceivers at both ends of a lane are clocked by signals derived from the same crystals, then data is moved from transmitter to receiver with no loss of synchronization. This is referred to as a "synchronous" or "single-crystal" configuration, and is illustrated in Figure 4.

Figure 4. Synchronous (Single-Crystal) Configuration
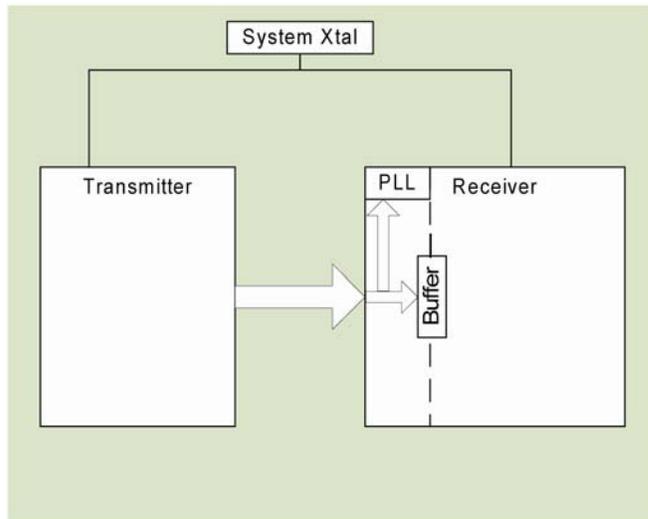
In other cases, the transmitter is driven by a reference frequency with a given tolerance; the main logic of the receiver is driven by a different crystal of the same frequency and tolerance. This is referred to as an "asynchronous" or "multi-crystal" configuration. While the nominal frequencies of the transmitter and receiver are the same, the actual frequencies will differ by some amount because of the allowed tolerances. The worst case frequency difference occurs when the transmitter is at the fast end of its range and the receiver is at the slow end.

The transmitter embeds its clock signal into the transmitted data stream; the receiver, using a phase-lock-loop (PLL), recovers the transmit clock from the incoming data stream. There are two frequencies present in the receiver: the transmit clock frequency, as recovered from the data stream, and the receiver reference clock, as driven by the receiver's crystal shown in Figure 5. This slight mismatch in frequencies within the receiver means that either the data arrives slightly faster than the receiver can process it (which can cause data to be lost if not handled properly), or slightly slower (which may not be a problem).

Figure 5. Asynchronous (Multi-Crystal) Configuration
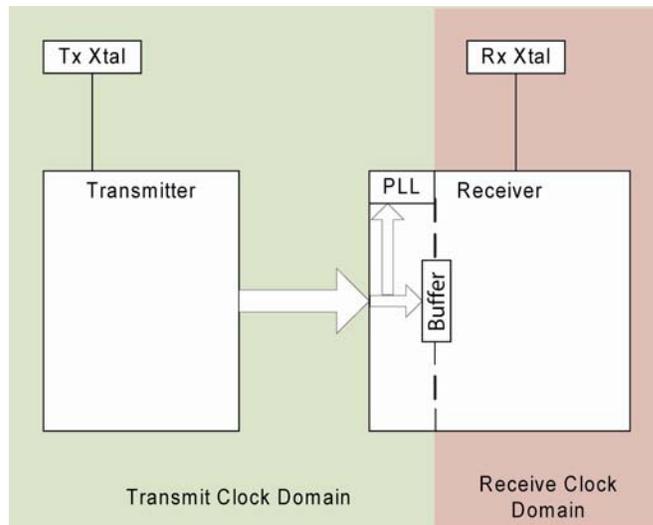
To compensate for this, an "elastic buffer" is placed in the receive path. Data is written into the elastic buffer using recovered transmit clock; data is read out of the elastic buffer using the receive reference clock. Clock compensation sequences are inserted into the data stream at regular intervals, before the buffer can overflow; these sequences are removed from the stream at the elastic buffer, and give the buffer a chance to catch up.

The management of clock compensation sequences is handled by the SerialLite protocol transparently to any logic outside the SerialLite logic. It does mean that there is some slight bandwidth lost to the clock compensation sequence.

Uniquely among serial protocols, SerialLite allows operation in asynchronous mode, with automatic compensation, or in synchronous mode, where there is no compensation and the overhead and logic required for compensation can be eliminated. This is a choice that is made at design time by the system designer.

Link Initialization & Training

Link initialization and training is the process of synchronizing data within a lane and lanes within a link. This process occurs in three stages, as illustrated in Figure 6. Initialization will be automatically handled by a SerialLite-compliant core. For maximal logic savings, the user can elect not to reverse lane polarity and/or lane order.

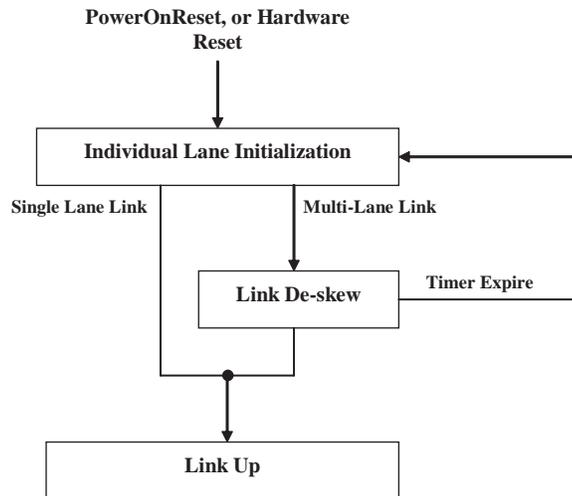> **Lane Initialization:** Each individual lane goes through this procedure to establish code group synchronization and determine correct lane polarity.

> **Link De-skew:** This process applies to multi-lane links only. It performs lane-to-lane de-skew within a link and determines correct lane order.

> **Link Up**: This stage waits for the remote port to complete its initialization and then declare link initialization complete.

Figure 6. SerialLite Link Initialization

PowerOnReset, or Hardware Reset

Individual Lane Initialization

Single Lane Link   **Multi-Lane Link**

Link De-skew   **Timer Expire**

Link Up

User Packet Types

SerialLite protocol supports two different user packet types: data packets and priority packets. These two types of packets are transferred through the two Atlantic interfaces shown in Figure 7.

- Data Packets: A "cut-through" data flow is used for data packets. This means that the transmitter starts transmitting the packet as soon as enough data has been received to place across all lanes. For large packets, it means that the receiver will have passed early portions of the packet on to the higher-level logic even before the transmitter has received the end of the packet.
- Priority Packets: A "store-and-forward" data flow is used for priority packets. This means that the transmitter will not begin transmitting a packet until it has received the entire packet. This is required to support packet retransmission option.

In addition to data packets, the regular data Atlantic interface can accommodate streaming data instead of packet data. Streaming data has no beginning and no end, and there is no encapsulation. This is the configuration requiring the fewest elements to implement.

User Packet Encapsulation

SerialLite encapsulates data and priority packets by wrapping start and end of packet symbols around them. The Pad symbol is conditionally inserted to maintain a word boundary. Optionally a CRC can also be generated and appended to the end of the user packet to protect against errors. Figure 7 below illustrates how user packets are encapsulated.
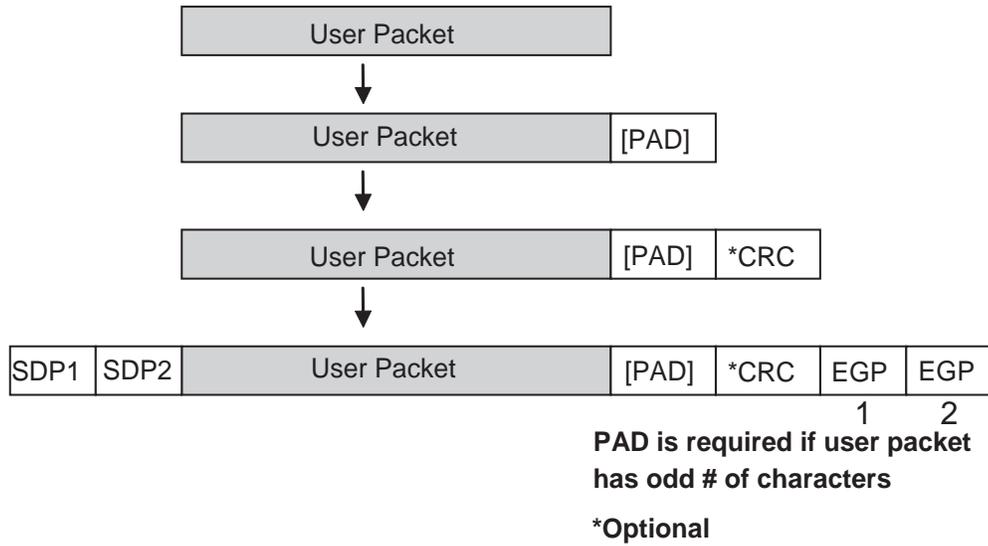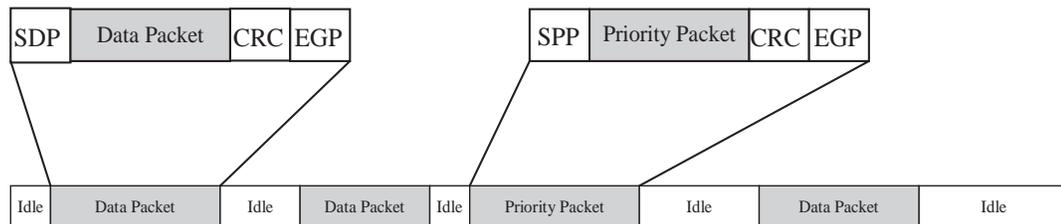
Figure 7. User Packet Encapsulation

User Packet

User Packet	[PAD]

User Packet	[PAD]	*CRC

SDP1	SDP2	User Packet	[PAD]	*CRC	EGP 1	EGP 2

PAD is required if user packet has odd # of characters

***Optional**

Figure 8 shows an example of packet activity during transmission. "Idle" refers to transmitted Idle symbols when data is unavailable. Idle symbol insertion is automatically handled by SerialLite logic, whether in the middle of a packet (due to flow control, for example) or between packets.

Figure 8. Sample Packet Activity

SDP	Data Packet	CRC	EGP

SPP	Priority Packet	CRC	EGP

Idle	Data Packet	Idle	Data Packet	Idle	Priority Packet	Idle	Data Packet	Idle

Error Protection

SerialLite optionally provides CRC error-checking coverage to data and/or priority packets to protect against errors. Two types of CRC polynomials are supported: CRC-16 and CRC-32. The CRC covers all bytes from the second symbol of the Start of Packet sequence to the optionally inserted Pad symbol, as illustrated in Figure 9. The second Start-of-Packet symbol is protected because it contains the packet number, which is used if the Retry-on-Error feature is implemented.

11

Figure 9. CRC Coverage

FCS Coverage

| SDP1 | SDP2 | User Data Packet | *PAD | CRC | EGP1 | EGP2 |

*Optional
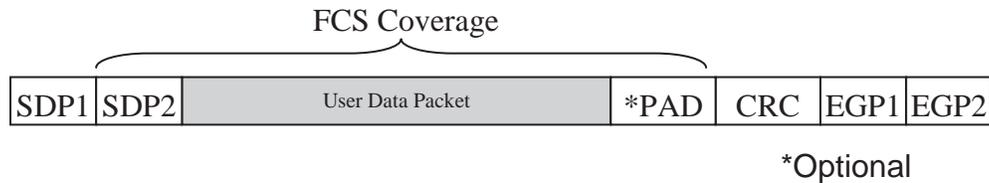
Lane Striping

Lane striping is the process of distributing packet byte streams to multiple lanes within a link. SerialLite distributes a word (two bytes) of data at a time to each lane. Therefore all transmitted data is word-aligned. An example of striping of a nine-byte packet across a four-lane link is shown in Figure 10.

Figure 10. Lane Striping Example
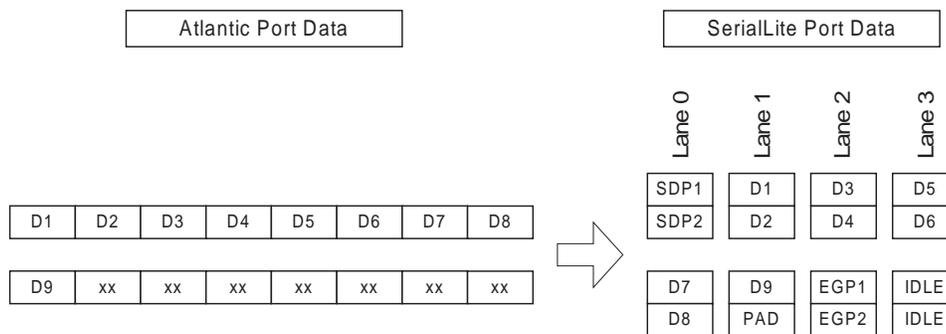
Link Management Packets

SerialLite defines special internal packets to facilitate the Flow Control and Retry-on-Error features. These packets are generated and processed entirely within the SerialLite data link layer; no evidence of link management packets is presented to the Atlantic interface. To ensure transmission reliability, link management packets have a BIP-8 byte at the end for detecting transmission errors.

Flow Control (Optional)

Flow control is an optional feature that can be implemented to ensure that data isn't transmitted faster than the receiving system can consume it. A typical setup consists of a near transmitter sending data to a far receiver. Flow control allows the far receiver to exert "back pressure" on the data it is receiving. This gives the receiver a chance to catch up before the incoming data overflows the receiver buffer. Note that flow control isn't required to compensate for transmit and receive clock frequency tolerance discrepancy; that is handled automatically by the Clock Compensation feature. Flow control would only be required for systems in which the designer expected that the logic outside of SerialLite might not consume the data quickly enough. If the outside logic isn't accepting the data fast enough, the internal receive storage will

fill up; flow control, if implemented, automatically causes data transmission to pause when the storage becomes too full.

SerialLite utilizes a 'pause' flow control scheme. When the far receiver is not able to receive more data, the far transmitter sends a flow control packet (pause packet) to the near receiver to instruct the near transmitter to stop transmitting data for a user-defined period of time. Note that the designer decides how far ahead of "full" to pause transmission. That consideration must take into account the amount of time it takes the Pause command to reach the receiver, and how much data is already "in flight" by the time the Pause command is enacted.

Retry-on-Error (Optional)

Retry-on-Error is an optional feature that provides a mechanism for ensuring the reliability of priority data transmission. A typical set up consists of a near transmitter sending data to a far receiver. Retry-on-Error allows the far end to request retransmission of priority packets that were received with errors. This requires the far transmitter to acknowledge every priority packet received. Retransmission is only possible for priority packets, not for regular data packets. There are two types of acknowledgement:

1. ACK: packet received is good and error free
2. NACK: packet received with an error.

SerialLite keeps track of packet numbers so that out-of-order packets can be trapped, and retransmission restarted to recapture any lost data, and to discard any packets sent more than once. There is also a timeout mechanism so that any lost acknowledgment packets can be accommodated.

If the Retry-on-Error option is not implemented, then all packets are transmitted without any acknowledgements from the receiver.

Nesting Packets

In order to reduce delays in transmitting time critical control packets, SerialLite inserts high priority packets within any data packets that are already in transmission. Link management packets have the highest packet priority, as shown in the example below. Figure 11 below shows double-nesting of a link management packet within a priority user packet, which in turn is nested within a user data packet.

Figure 11. Packet Nesting Example
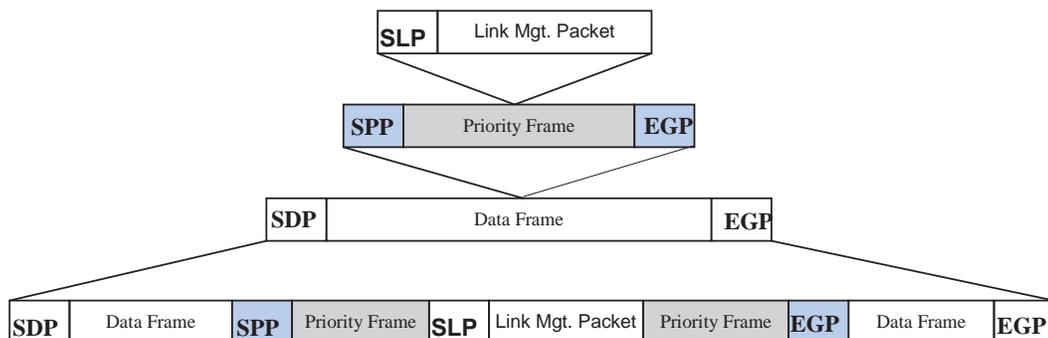
Error Events & Handling

SerialLite classifies all errors as catastrophic, link, data, or oversized.

- A catastrophic error is an unrecoverable error cause by the initialization state machines.
- A link error results when the link is not able to transmit or receive data and it triggers the initialization process.
- A data error results in the error bit being activated on the Atlantic receive port, or, if implemented, a request for retransmission from the receiving port.
- The transmitter marks a priority packet as oversized if it detects that the packet has exceeded the maximum allowed size.

Table 3 below summarizes all errors and how they are handled.

Table 3. Error Types and Actions

Error Type	Cause	Action
Catastrophic	1. Reverse signal polarity detected, and the design does not have polarity reversal enabled. 2. Reverse lane order detected, and the design does not have lane order reversal enabled. 3. Lanes in non-sequential order.	SerialLite terminates its operation. This should only happen during system debug, never during end user operation.
Link	1. Eight consecutive ||TS1|| received in all lanes simultaneously. 2. Character loss due to overflow/underflow of the clock tolerance compensation elastic buffer. 3. Loss of character alignment 4. Loss of lane alignment 5. Minor error threshold exceeded 6. Retry-on-Error Timer Expired three times	Cause link status to indicate link_down and trigger link initialization state machine.
Data	1. Invalid 8B/10B codes groups 2. Running disparity errors 3. Unsupported valid code groups 4. CRC errored packets with /EGP/ ordered_set	Two possibilities: 1. Data Packet: marked as bad and forwarded to user link layer. 2. Priority Packet: Request for retransmission.
Oversized	Priority Packets Only: maximum packet length violation	Packet is discarded without being transmitted.

Summary

The SerialLite protocol provides a robust mechanism for transporting data from one point to another. Because it focuses only on the physical and data link layers, it is a simple, lightweight protocol. It is the only serial protocol to provide both an extremely small minimal configuration with the option of adding numerous capabilities only as required by the application.

SerialLite is targeted for designers that don't need higher OSI Reference Model layers, blind inter-operability, or data routing supported by heavier-weight protocols. It also encompasses all of the key requirements for a wide variety of data transfer applications so that designers making the transition from older schemes to a serial scheme do not need to be burdened with the complex details involved in serial communication. By focusing on the higher-level requirements of the application and allowing a SerialLite-compliant core to implement the details, the designer will save many weeks of work, and will be able to focus on high-level aspects of the application.