Drive ESCON™ With HOTLink™

Introduction

The IBM® ESCON™ (Enterprise System CONnection) interface is presently experiencing rapid growth. Originally designed as a replacement for the older block-mux channel, it is also finding use as a high-performance system interface. This once IBM-proprietary interface is presently being processed to become an ANSI standard interface (known as SBCON) for computer to peripheral interconnect.

This application note contains an overview of ESCON operation and a design example of an ESCON physical interface, including a number of the low-level ESCON state machines (including the VHDL source code), implemented using HOTLink™ and a pASIC™ field programmable gate array.

Channels

The term *channel*, when referring to mainframes, carries a specific meaning. Rather than representing the connection between pieces of equipment, here it also represents a significant piece of equipment as well. The channel is, in effect, a sophisticated and intelligent DMA engine whose purpose is to move information between I/O devices and main storage. This channel function removes the burden of handling I/O activities from the main CPU.

Block-Multiplexer Channel

The original block-multiplexer channel dates back to the System 360/370 family of IBM mainframe CPUs. It uses a pair of parallel-bus copper cables (referred to as Bus and Tag cables) to move data between the host CPU and the I/O and storage peripherals as shown in *Figure 1*. These bus and tag cables were daisy-chained from the host channel adapter through multiple storage and I/O directors.

While quite powerful in its day, the block-mux channel shows both its heritage and its age. The bus and tag cables are quite bulky (around 1.5″ in diameter), heavy, *and* costly. The maximum length of the link between the host CPU channel adapter and the cable terminator is 400 feet, and operates at a maximum transfer rate of 4.5 MBytes/second. While originally designed to simultaneously support a larger number of peripherals, its is now possible to saturate the full I/O bandwidth capability of a block-mux channel with a single disk drive.

ESCON Channel

The ESCON channel was introduced in 1990 along with the ESA390 series of mainframe computers. It uses high-speed serial, point-to-point fiber-optic links to replace the daisy-chained parallel-bus copper cables of a block-mux channel. By maintaining the same host CPU software structures used with the block-mux channel, it was possible to dramatically change the architecture (and performance) of the I/O subsystem without effecting the major I/O routines present in the host CPU and channel microcode.

This new interconnect media was also merged with a dynamic switched connection scheme to improve both availability and access to the I/O peripherals. The use of switches (known as *directors*) allows many more paths to each peripheral, with multiple paths being active through each director at the same time. This new interconnect structure is shown in *Figure 2*. This switched I/O structure is now finding popular use in many other data communications in-

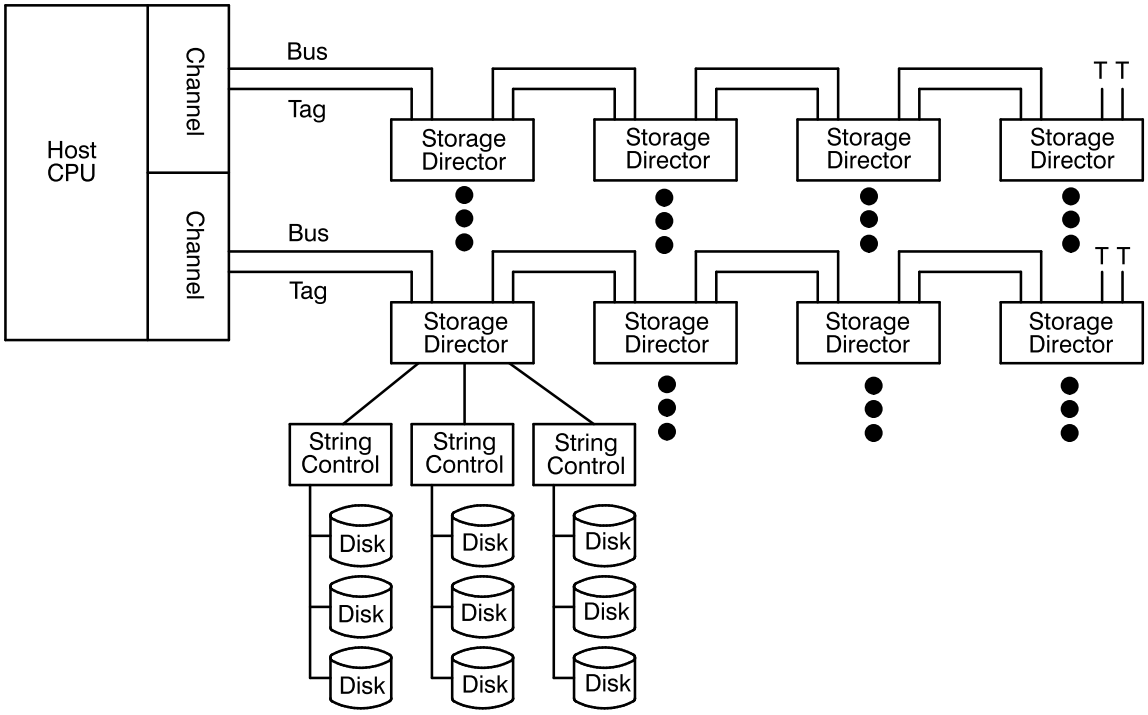

Figure 1. Block-Multiplexer Channel Subsystem

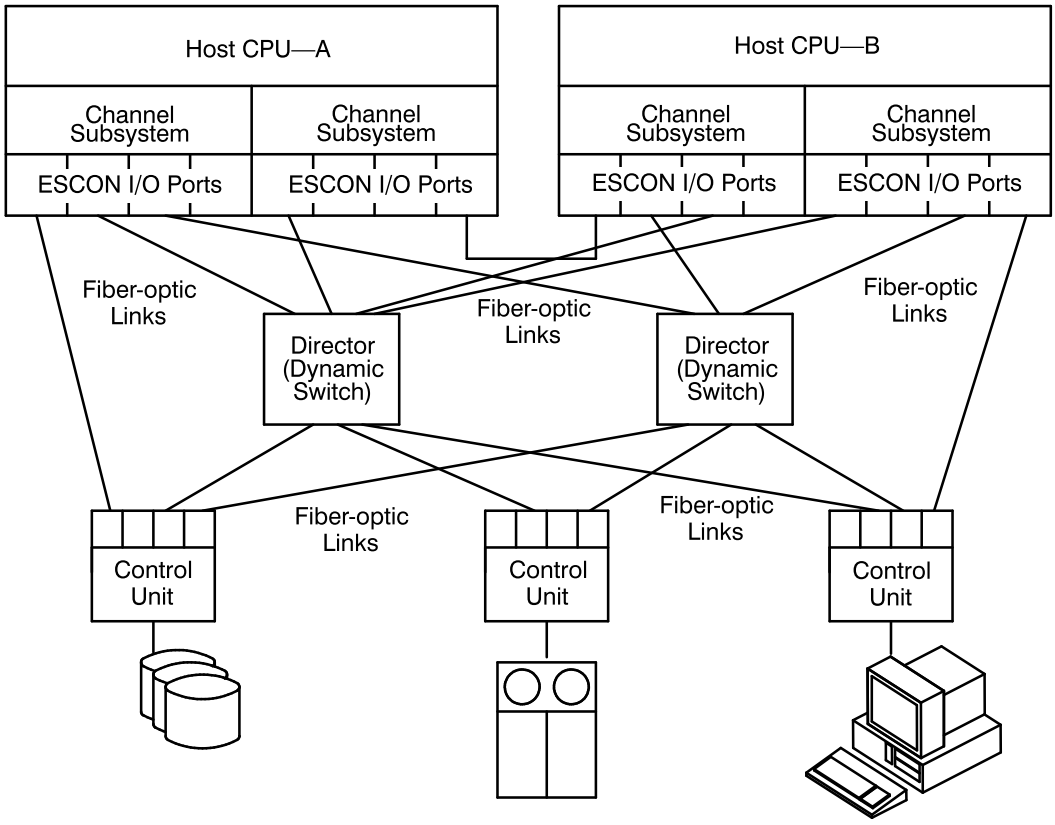

Figure 2. ESCON Channel Subsystem

terfaces like switched Ethernet, ATM, and Fibre Channel.

The ESCON interface provides numerous improvements over the older block-mux channel. A few of these are

- Improved transfer rate to 20 MBytes/second
- Longer distances—up to 3 km for each link and up to three links (two switches) between Channel and Control Unit
- Immunity from EMI/EMC concerns
- Improved access, redundancy, and availability through use of dynamic switches

ESCON Physical

The physical-level interconnections of ESCON are all made with 1300-nm LED-based optical links. These links operate through either 62.5 μm or 50 μm core multi-mode optical fibers at a fixed bit rate of 200 Mbits/second. This bit rate represents the encoded bit rate for the data being sent.

The data sent across ESCON links is encoded using the 8B/10B code built into HOTLink. (See the CY7B923/933 datasheet for a detailed description of the 8B/10B code.) This code converts normal 8-bit bytes into 10-bit transmission characters. While this encoding does have a 25% overhead, the benefits of using it far outweigh the data-rate penalty.

Part of the reason for the two extra bits in each character is to guarantee a minimum transition density for the receive PLL. Since no clock is present in the serial data, the HOTLink receiver PLL is used to extract a bit-rate clock from the data steam

Another benefit from this code is its DC-balance characteristic. This means that, over time, the net difference of all 1-bits versus 0-bits sent is at or near zero. This DC-balance characteristic allows the optical receiver circuits to be much simpler and lower in cost by reducing the complexity of the AGC (automatic gain control) in the receiver preamplifier.

With a transmission character being ten bits in length, there are actually 1024 possible transmission characters. Of these possible codes, only a fraction of them meet all the run length and DC-balance coding rules. The remainder are illegal codes and are detected as errors at the receive end of the link. Most of the valid codes are used to represent the 256 possible data bytes, with a few remaining legal transmission characters used for synchronization and in-band signaling.

The term *in-band* means that all delimiters, protocol, clocking, etc., are handled through the same serial interface as the data; i.e., there are no other control lines or interfaces used for this information. The 8B/10B code provides twelve transmission characters for these in-band functions. Of these twelve characters (referred to as special characters), only six are defined for use by ESCON.

Synchronization

With any serial interface some form of synchronization is necessary at the receiver-end of a link. The function of synchronization is to line up the receiver bit and byte clocks with the serial data stream.

Bit Synchronization

Bit synchronization is performed (for the most part) automatically by the receiver PLL. As transitions are detected, the phase detector in the receiver uses the position of the transition (relative to its internal bit-clock) to adjust the phase and frequency of the local bit-clock. This local bit-clock is optimally adjusted to allow the serial data stream to be sampled at the center of each bit. However, bit synchronization alone is not sufficient to recover and decode the transmitted information. This requires knowledge of which bit in the serial stream is the start of a character.

Framing

Proper detection of character boundaries is referred to as framing. Unlike bit synchronization, which occurs primarily in the analog domain, framing is a full-digital operation.

Framing is performed by examining the serial bit stream for a specific pattern (called a comma). This

test occurs on every bit-clock until an exact match is found. At this point the receiver byte-clock is reset to line up with the character boundary. Following this, all characters output from the receiver should remain properly synchronized, until some external event causes a significant disruption in the data stream.

The comma in the 8B/10B code is the seven bit pattern 0011111 (or its alternate 1100000). This bit pattern is part of the K28.5 special character. It cannot appear in any other location in any 8B/10B encoded character, and cannot be generated across the boundaries of any pair of characters.

While the detection of individual bits is controlled automatically by the PLL, the detection of framing for ESCON must be under the control of a separate state machine. This machine determines under what conditions the receiver is allowed to perform its framing function.

ESCON Synchronization

An ESCON interface is normally considered to be in one of two states regarding synchronization; either Synchronization_Acquired or Loss_Of_Synchronization (LOS). The transitions between these two primary states actually involve a number of substates that track error conditions and special characters on the interface. This state machine is shown in *Figure 2*.

In addition to its five states (four Sync Acquired and one Loss Of Sync), it operates with a 4-bit counter to track both valid characters and K28.5 characters. Since in any specific state of the machine only one thing is being counted (valid characters or K28.5 characters), only a single counter is needed.

Loss Of Synchronization

The ESCON interface automatically enters the LOS state following power-on. In this state (if a valid signal is present) the serial data receiver is enabled not only to received data, but also to frame on any received K28.5 character (RF=1).

While the receiver will frame on the first K28.5 received, this is not sufficient to leave the LOS state.

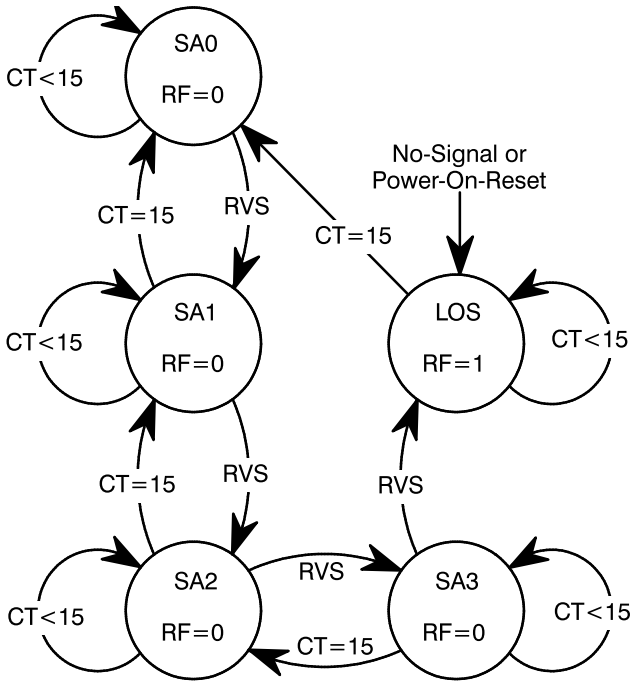

Figure 3. Synchronization State Machine

This requires reception of a minimum of fifteen K28.5 characters with no intervening code violations between any of the received characters. These K28.5 characters may be directly adjacent or more likely will have other characters interspersed. Once this string of K28.5 characters has been received, the receiver enters the Synchronization_Acquired state.

Synchronization Acquired

Exit from the LOS state also removes the reframe signal from the receiver (RF=0). In this condition the receiver will ignore (for framing purposes) all K28.5 characters embedded in the data stream. These characters are still properly received and decoded for use as part of the link protocol.

In the Sync Acquired state the state machine now tracks any code violations (RVS). If a code violation occurs the state machine changes from the basic Sync Acquired state (SA0) to SA1. In this state the machine has now detected a single error. It then enables the separate 4-bit counter to check for consecutive valid characters. If the following fifteen characters are received without error, the machine reverts back to the basic Sync_Acquired state.

If, however, additional character errors are detected, the state machine will advance through the SA1, SA2, and SA3 states—one change for each character received in error. At each of these states the machine will again check for valid characters and will revert to the previous state if fifteen are received without any errors. This would allow an interface receiving exactly one error every sixteen characters to remain in the SA0 and SA1 states, while a similar interface receiving one error every fifteen characters would quickly move to the LOS state and remain there.

Link-Level Operations

The actual functionality of an ESCON link is defined in terms of various ordered sets of special characters and data bytes. These ordered sets are used to define frame boundaries, control dynamic connections, and control synchronization between the transmitter and receiver circuits. All valid ESCON ordered sets are listed in *Table 1*.

Table 1. ESCON Ordered Sets

Ordered Set	Characters
Idle function	K28.5
Connect-start-of-frame delimiter	K28.1 K28.7
Passive-start-of-frame delimiter	K28.5 K28.7
Abort delimiter	K28.6 K28.4 K28.4
Disconnect-end-of-frame delimiter	K28.6 K28.1 K28.1
Passive-end-of-frame delimiter	K28.6, K28.2 K28.5
Not-operational	K28.5 D0.2
Unconditional-disconnect sequence	K28.5 D15.2
Unconditional-disconnect-response sequence	K28.5 D16.2
Off-line Sequence	K28.5 D24.2

Idle Function

The K28.5 character in ESCON is used for multiple purposes. It is

- the first character of many ordered sets
- used to provide byte framing of the serial data stream
- used as a fill or *Idle* character between frames and sequences

Because the K28.5 character is contained in many of the other ordered sets, a single K28.5 cannot be conferred to be an Idle function until the following character is detected. If the following character is also an K28.5, then the previous K28.5 is part of an Idle Function. If the following character is anything else, then the K28.5 character is part of a delimiter or sequence (or an error).

Delimiters

Delimiters are used to mark the start and end of frames. Frames are the real workhorse of the interface because they carry data. All frames have a start-of-frame delimiter (SOF) and an end-of-frame delimiter (EOF). (An Abort delimiter is considered to be a type of EOF.) These delimiters are only sent once per frame. Each frame must be separated by a minimum of four Idle characters.

Sequences

Sequences are used to indicate specific equipment conditions or states that cannot be identified through the use of frames. Unlike a delimiter, the ordered set defined for a specific sequence is sent repeatedly until the machine state changes or a specific response is received. At the receiver, a sequence is only detected as being valid if the defined ordered set is received a specific minimum number of times in succession.

Frames

Frames are used to carry information between the channel, switches, and the peripherals. Two generic types of frames exist; Link-Control and Device Level.

All frames follow the same three-field format:

- a 7-byte fixed-length link header
- a variable-length information field (may have a length of zero for some Link-Control frames)

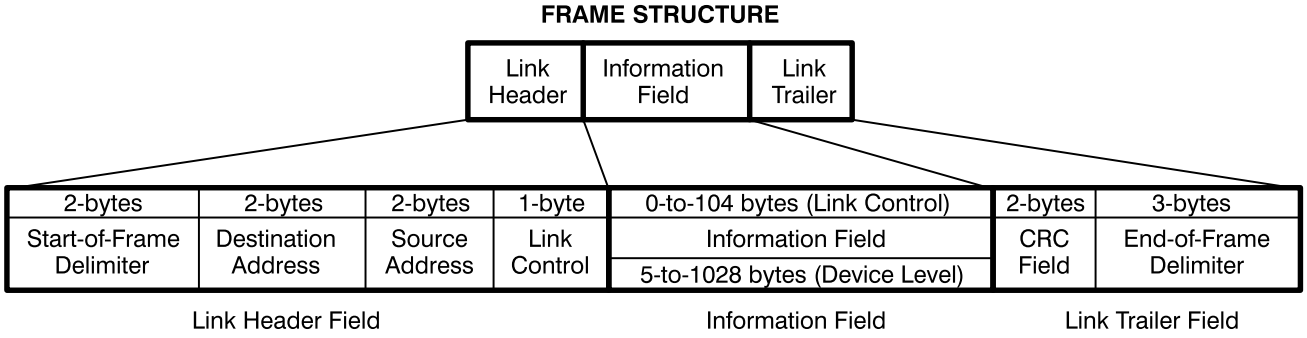

Figure 4. ESCON Frame Format

- and a 5-byte fixed-length link-trailer field

The structure of an ESCON frame is shown in *Figure 4*. The low-order bit of the Link Control field in the Link Header identifies the type of frame. When set to a one, the frame is a Link Control frame. When set to a zero, the frame is Device Level frame.

Link-Control frames are use to manage, configure, and maintain the link itself, and range in length from 12 to 116 bytes. Device Level frames carry data between the channel and the peripheral and range in size from 17 to 1040 bytes.

Frame Validation

Before a frame can be processed, it must be validated as a properly received frame. This involves making sure that there are no special characters or idles in the middle of the frame, no decoding errors are detected in the serial stream, and that the CRC Field (Cyclic Redundancy Check) shows no errors.

Cyclic Redundancy Check Field

The CRC field contains a 16-bit redundancy check code, used to insure that the received frame contents are the same as those sent. This field is generated at the transmitting end of a link and sent as the first two bytes of the Link Trailer field. It is calculated on all bytes between the start-of-frame delimiter and the Link Trailer field.

At the receiving end of the link the CRC is again generated using the received data stream. Now the CRC is generated on all bytes between the start-of-frame delimiter and the end-of-frame delimiter.

The CRC code used with ESCON is that defined by the ITU V.41 standard (previously known as CCITT). The polynomial for this CRC is listed in *Equation 1*.

$$x^{16} + x^{12} + x^5 + 1 \qquad \text{Eq. 1}$$

Normally with a code of this type the CRC remainder register is preset to an all 1s condition prior to the first bit of information being clocked through the polynomial. This is done to ensure that the polynomial will change state no matter what the data stream contains. At the end of the generation, the two bytes comprising the CRC remainder are sent as part of the data stream. At the receiving end the same process occurs, but the two CRC bytes are also clocked into the CRC register. If no errors exist in the serial stream then the contents of the CRC check register should be zero.

To increase the level of protection, the CRC is handled slightly differently in an ESCON interface. Here the CRC remainder generated at the transmitter is inverted prior to sending it across the link. When it is received (correctly) the CRC check register is no longer cleared, but must be set to exactly 1D0F (hexadecimal). Any other value indicates a transmission or reception error.

ESCON Design Example

The following design was originally done to replace an existing ESCON protocol component that was no longer available. All VHDL source code listed here has been both simulated and tested in a functioning ESCON system.

This design example covers

- an ESCON-compatible optical media interface
- ESCON-certified HOTLink serializer/deserializer components
- a pASIC383 protocol chip containing
  - — transmit and receive CRC circuits
  - — parity check and generate circuits
  - — synchronization state machine
  - — command code translation capability
  - — input/output pipeline registers
  - — miscellaneous flip-flops, muxes, and gates

The design is partitioned into transmit and receive data paths, and is implemented in four active devices:

- a pASIC383 containing both transmit and receive protocol functions
- a CY7B923 HOTLink transmitter for serialization and 8B/10B encode
- a CY7B933 HOTLink receiver for deserialization and 10B/8B decode
- a Siemens V23806−A1−M16 ESCON fiber-optic transceiver

The structure of how these components connect and major data paths are shown in *Figure 5*, with a complete schematic shown in *Figure 6*.

Fiber-optic Transceiver

The fiber-optic transceiver is an optoelectric device that both converts electrical signals to light (transmitter) and light into electrical signals (receiver).

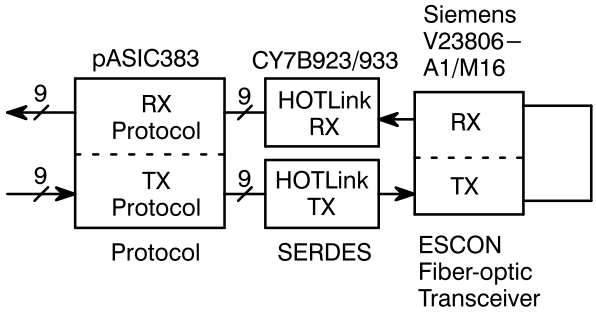

Figure 5. Design Example Structure

To operate with the ESCON interface the transceiver must meet a number of specific characteristics:

- operate at 200 Mbaud
- operate at 1300 nm wavelength
- use 62.5-μm or 50-μm core optical fiber
- meet the 0.7″ ferrule spacing and other dimensions of an ESCON optical connector

In addition to these criteria, compliant transceivers must meet numerous power level, receive sensitivity, and electrical interface criteria to properly operate in an ESCON environment. Manufacturers of ESCON compatible fiber-optic transceivers include Siemens, AMP, IBM, and others.

SERDES

The next section in an ESCON link is the serializer/deserializer block, also known as the SERDES. This section converts parallel bytes of information into an 8B/10B encoded serial data stream for transmission, and also converts a received 8B/10B encoded serial data stream back into parallel data bytes.

The Cypress CY7B923/933 HOTLink components are designed to perform this SERDES function. These components are specifically optimized to support the ESCON interface, as well as Fibre Channel, ATM (Asynchronous Transfer Mode), and proprietary communications links.

These HOTLink parts are especially well suited to the ESCON market because of their built-in 8B/10B encoders and decoders. This encode/decode function is required for ESCON operation. By building the encode/decode into the SERDES block, the complexity of this part of the interface design is removed from the design process. Its presence in the SERDES block also means that hardware resources are not required elsewhere to implement the encode/decode function.

The 8B/10B code used in the HOTLink components is licensed by Cypress Semiconductor from IBM. Any user of these parts is fully licensed to use the 8B/10B encoders and decoders contained in them at no cost and no royalties. For those applications that already have 8B/10B encoder/decoder circuits pres-

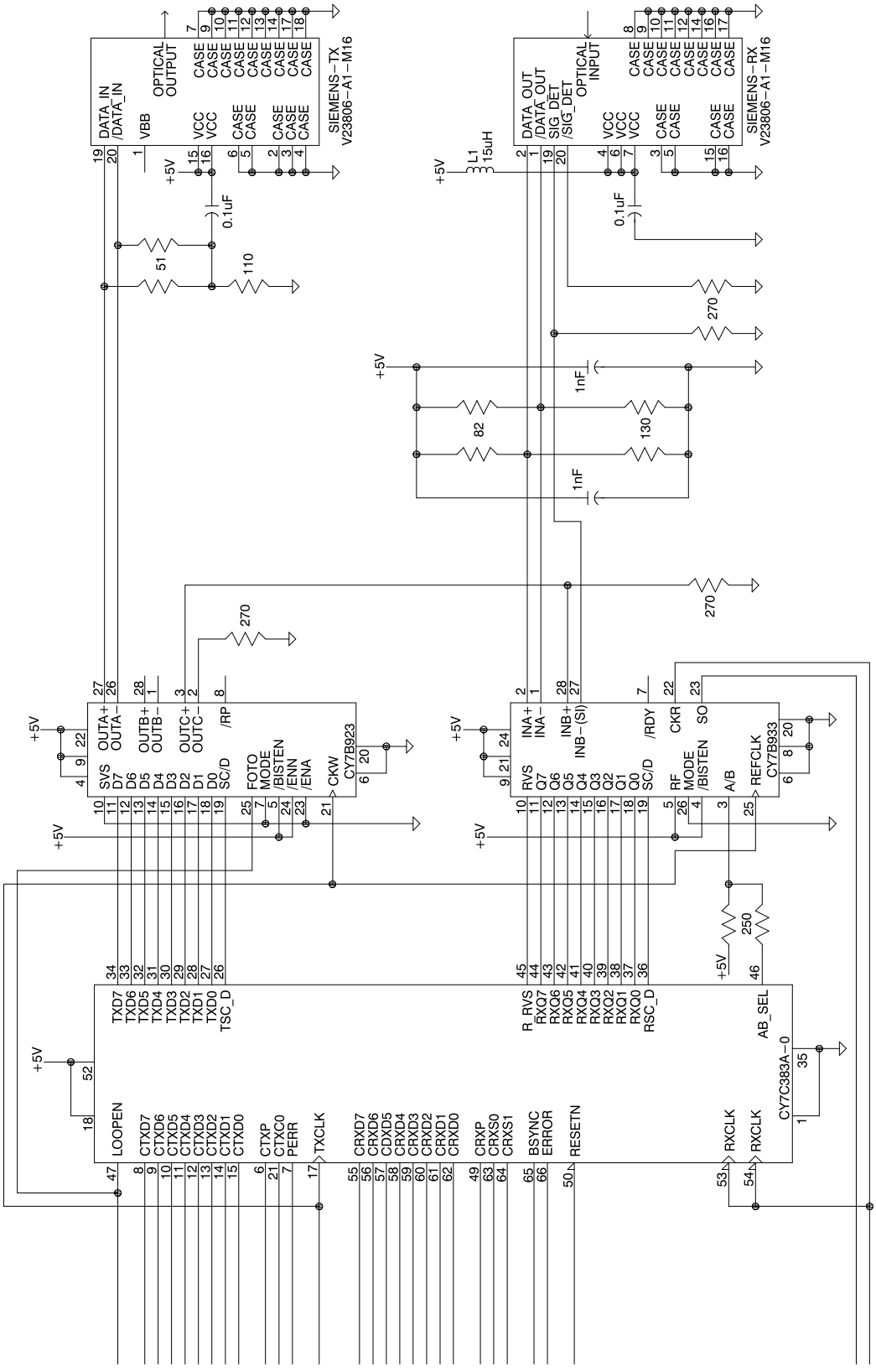

Figure 6. ESCON Physical I/O Interface Schematic

ent in their system, the encoder/decoder present in HOTLink can be bypassed through use of the MODE pin on each part.

An in-depth explanation of the operation and usage of the HOTLink components may be found in the CY7B923/933 datasheet and the *HOTLink User's Guide*.

Serial I/O Electrical Interface

The interface between the fiber-optic transceiver and the HOTLink SERDES operates at 200 Mbits/second. This interface is implemented with ECL (Emitter-Coupled-Logic) signaling to provide a low-noise, high-speed connection. Unlike standard ECL, which is normally operated below ground, both the fiber-optic transceiver and the HOTLink SERDES components are operated above ground. This allows the ECL portion of the design to use the same +5V supply as the surrounding logic. When ECL is operated from a positive supply it is referred to as Positive-ECL or PECL.

The source for the serial data stream is the CY7B923 HOTLink transmitter shown in *Figure 6.* A simplified schematic showing just the interconnect for the serial transmit path is shown in *Figure 7.*

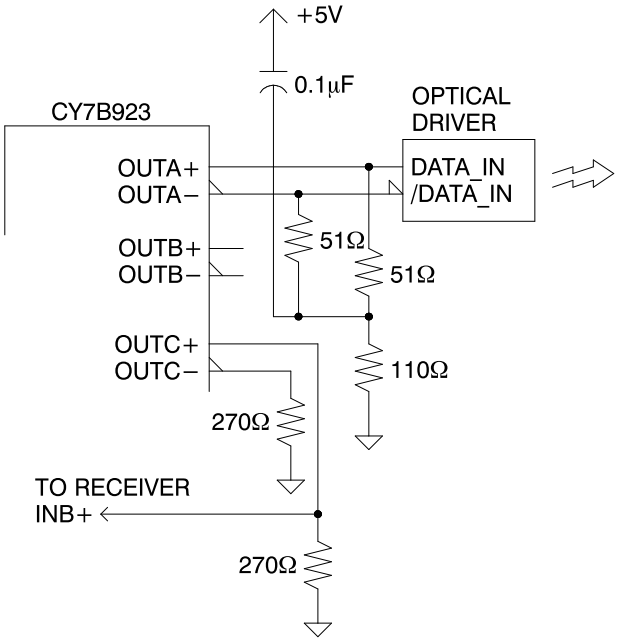

Figure 7. HOTLink Transmitter-to-Optical Serial Interface

The serial data is connected to the fiber-optic transmitter using a differential connection from the OUTA± differential output of the HOTLink transmitter. Because these are ECL/PECL signals, they require a pull-down bias to allow the outputs to switch.

With a transmission rate of 200 Mbits/second, the interconnect used for these signals should (in most cases) be constructed as a controlled-impedance transmission line. The bias network used on the OUTA± signals is referred to as a Y-bias network. It is designed to provide an equivalent transmission line termination impedance of 50Ω while providing a bias level of $V_{CC}-2V$.

The received serial data stream is output from the fiber-optic receiver as a differential signal, as shown in *Figure 6*, and is sent to the CY7B933 HOTLink receiver INA± inputs. A simplified schematic showing just the interconnect of the serial receive path is shown in *Figure 8*. Because this is also a PECL signal, it should be treated in a manner similar to the transmit serial path. This means controlled impedance transmission lines and a proper bias/termination network.

While the receive-path bias/termination network may be implemented using the same Y-bias network used with the transmit serial path, a Thévenin network is shown here. These two bias networks, when used with differential signals, are effectively interchangeable. For single-ended signals requiring the

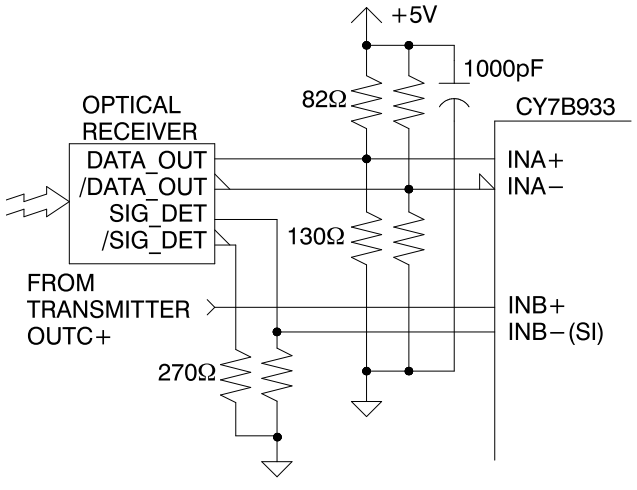

Figure 8. Optical-to-HOTLink Receiver Serial Interface

same electrical characteristics, the Thévenin network must be used. For additional information on terminating and biasing PECL signals, please see the application note "HOTLink Design Considerations" in the *HOTlink User's Guide*.

Serial I/O Support Interface

In addition to the transmit and receive serial data streams, two other PECL signals are normally present in an ESCON interface: signal-detect and local-loopback. The signal-detect function is performed by the fiber-optic receiver. It outputs a PECL logic signal to inform the upstream hardware if a valid signal is present or not. This signal is monitored to determine the synchronization state of the interface.

Because this is a PECL-level signal, it is necessary to convert it to a TTL-level signal for use by upstream logic. While there are components available that explicitly perform this level translation, they are not necessary for this application. Instead it is possible to use one of the design features of the HOTLink receiver INB± inputs to perform this signal-level conversion.

The INB± input can be configured as either a differential PECL receiver (like INA±), or as a single-ended serial PECL receiver and a PECL-to-TTL converter. To use INB± as a differential receiver it is necessary to pull the SO (Status Out) pin to V_{CC}. This disables the PECL-to-TTL converter and maintains both inputs as a differential pair.

To use INB± as two separate inputs requires that the SO pin be loaded as a normal TTL-level output. When configured this way the INB− pin is the input for the PECL-to-TTL converter, with SO being the TTL output. This is the configuration used in *Figures 6* and *8*.

Most ESCON interfaces are also equipped with numerous self-diagnostic capabilities. At the physical interface the most common is a selectable loopback of the serial data stream. This allows all components (with the exception of the fiber-optic transceiver) of the interface to be tested by transmitting data and verifying that it can be properly received. This loopback function is normally implemented using the OUTC+ output of the HOTLink transmitter and the INB+ input on the HOTLink receiver in a single-ended PECL connection, as shown in *Figures 6, 7,* and *8*.

While the best PECL connection is always a differential connection (like that used on INA±), the usage of INB+ in a single-ended mode is fine under these conditions. Because the HOTLink transmitter and receiver are close together in the system and operate from a common power supply, the normal noise-margin concerns of single-ended connections do not apply.

This local loopback functionality is selected through the LOOPBACK signal on the pASIC FPGA. When active (HIGH), this signal drives the HOTLink receiver A/$\overline{B}$ select input LOW to selected the INB+ input for the deserializer, and drives the FOTO input to the HOTLink transmitter HIGH. This FOTO pin is used to disable the OUTA± and OUTB± outputs of the transmitter. This is normally done during loopback diagnostics to prevent the diagnostic data from being interpreted at the other end of the fiber-optic link.

ESCON Protocol Controller

The control of the serial data stream is performed using a pASIC383 FPGA. This part has been programmed to manage both the transmit and receive serial data streams. The programming and verification were done using VHDL (VHSIC Hardware Description Language) using Cypress's *Warp3™* logic synthesis and simulation tools. Complete source code of the design VHDL modules is listed in Appendixes A through H of this application note, and is available for download from the Cypress Bulletin Board system.

The design shown in this application note is effectively a logic replacement for a Triquint GA9104 ESCON protocol chip. Due to the flexibility of the pASIC family of parts, it is possible to add, replace, or remove logic that is not optimal for a specific application. In this design, the 8B/10B encoders present in the normal GA9104 were not implemented in the pASIC383 because they are already present in the HOTLink CY7B923/933. This allowed the entire functionality to be duplicated in a 2K-equivalent gate FPGA. The functions present in this design are

- Transmit Path
  - — input and output pipeline registers
  - — parity checker and status bit
  - — CRC generator and control state machine
  - — Command/data mux
  - — Command translator
- Receive Path
  - — input and output pipeline registers
  - — CRC checker, control state machine, and status bit
  - — parity generator
  - — Command/data mux
  - — Command translator
- Byte-Sync State Machine

Transmit Path

A block diagram of the transmit path is shown in *Figure 9*. Data is captured into a 10-bit register on each rising edge of the transmit clock (CKW). The data consists of an 8-bit data byte, a single control line (CTXC0), and a parity bit. The CTXC0 line is used to identify whether the data on the inputs is a command code (HIGH) or a data byte (LOW). If the latched character is a data byte, the data is simultaneously presented to the CRC register, the parity checker, and the output multiplexers. At the next rising edge of the transmit clock, this data byte is clocked into the CRC register, checked for proper parity, and loaded into the output register along with TSC_D set LOW.

The detection of a parity error is only a reported event, and occurs one cycle after the data (or command) is latched into the input register. Recovery from detected parity errors would normally require abnormal termination of the current frame using the Abort delimiter.

The CRC/MUX Control block is the heart of the transmit path logic. It monitors the CTXC0 line to determine when to

- preset the CRC register
- accumulate a CRC
- output the CRC bytes
- translate/send command codes

This block is implemented as a simple shift register that tracks the current and previous three states of CTXC0. These sixteen possible combinations (with don't care states removed) and their resulting outputs are listed in *Table 2*. The VHDL source code for this block is listed in Appendix C.

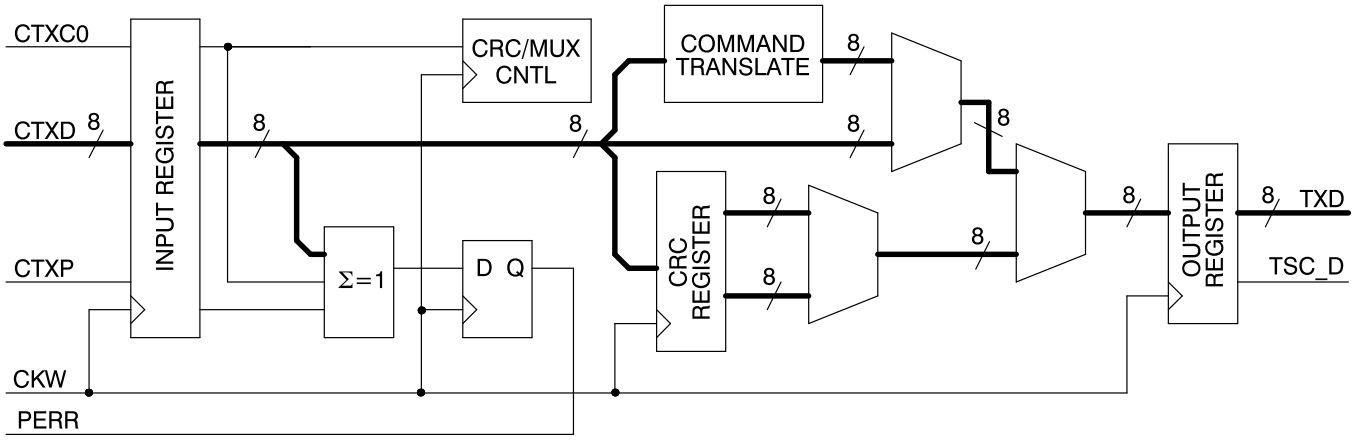

Figure 9. pASIC Transmit Path Block Diagram

Table 2. Transmit Path Control

CTXC0				Mux Select/ CRC Control
t+3	t+2	t+1	t+0	
X	X	X	0	Data
X	0	0	1	CRC High Byte
0	0	1	1	CRC Low Byte
X	X	1	1	Preset CRC
X	1	0	1	Command
1	0	1	1	Command

The CRC block implements the CRC-16 function in a byte-parallel fashion. This allows a full byte to be accumulated in a single clock cycle. While this does require a much larger number of XOR gates to implement than a serial CRC function, it allows the design to be constructed from much slower logic. Here the main CRC register is clocked at 20 MHz, rather than having to operate at a 200-MHz bit-clock rate. The VHDL source code for this function is listed in Appendix B.

The command-translate block is not normally needed for new designs. For this specific design it was necessary to translate an existing set of command codes to the native HOTLink command set. This translation is quite simple with the logic reduction performed manually for the transmit path. Here an 8-bit input command is decoded into a 4-bit command field (with the upper four bits of the byte set to zero).

The translation block actually implements circuitry to translate all twelve command codes in the 8B/10B character set. For ESCON implementations this logic could be simplified because only half of these (six) are actually allowed for use in ESCON ordered sets. The VHDL source code for this function is listed in Appendix D.

The last section in the transmit path is the output pipeline register. This block receives the multiplexed output of either the input pipeline register, the high-CRC byte, the low-CRC byte, or the translated command. It serves to keep the data presented to the HOTLink transmitter synchronous with the transmit clock.

Receive Path

A block diagram of the receive path is shown in *Figure 10*. Data is captured from the HOTLink receiver into the input register on each falling edge of the HOTLink recovered receive clock (CKR). Note that this could also be implemented using a rising edge clock, but that a falling edge clock was used for compatibility with the implementation being replaced.

All received data characters are clocked into the CRC register. Like the transmit path, this function is implemented in a byte-parallel form. The CRC register is synchronously preset if any command code is present in the input register. For all data codes it accumulates the CRC remainder.

The CRC register is constantly compared for the x'1D0F' pattern. The output of this compare is clocked into the output register. It is forced to a LOW for all clocks except the first command character received following a data character. This CRC status remains valid for only one clock cycle. The

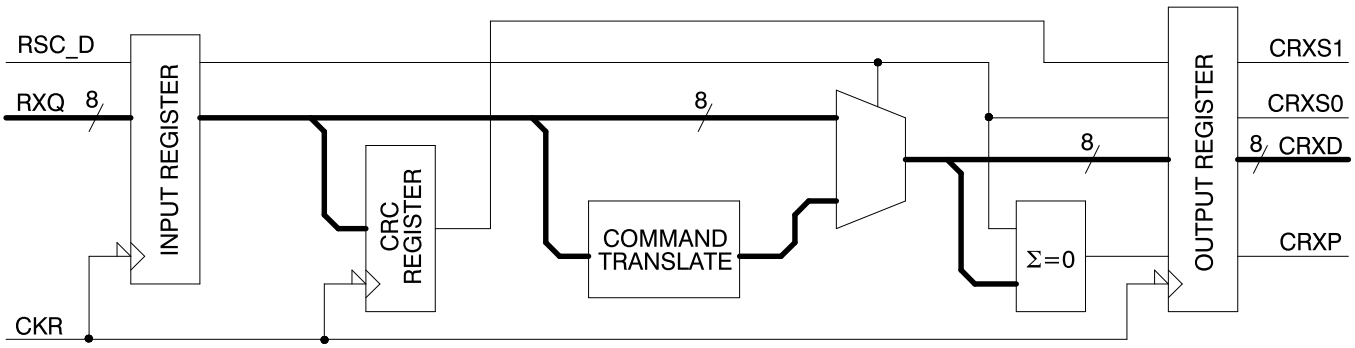

Figure 10. pASIC Receive Path Block Diagram

VHDL source code for this function is listed in Appendix E.

Just as in the transmit path, a command translation block is present in the design. This command translate block is not normally needed for new designs. For this specific design it was necessary to translate an existing set of command codes from the native HOTLink command set to a different set of command codes embedded in upstream logic. This block allows the HOTLink command codes to be translated to any host command set.

The translation block actually implements circuitry to translate all twelve command codes in the 8B/10B character set. For ESCON implementations this logic could be simplified because only half of these (six) are actually allowed for use in ESCON ordered sets. The VHDL source code for this function is listed in Appendix D.

Odd parity is generated on the output data byte and the CRXS0 status bit. This allows upstream logic to validate that the byte received is the same as that generated by the pASIC FPGA.

The last block in the receive section is the output pipeline register. This block receives the multiplexed output of either the input pipeline register or the translated command. It serves to keep the data presented to the upstream logic synchronous with the receive clock.

Byte-Sync State Machine

A block diagram of the byte-sync state machine is shown in *Figure 11*. The two primary structures in the machine are a 4-bit counter and a controlling state machine. The controlling state machine is programmed to follow the state diagram shown in *Figure 2*. It tracks the state of the RVS signal from the receiver and a decode from the input register of all C5.0 command codes (Idle characters). The four-bit counter is used to alternately count either valid characters (the absence of RVS) or valid Idle characters, based on the state of the machine.

The present form of this state machine was designed to duplicate the functionality of a previous implementation. Because of this it does not take into account the the additional condition of Signal Detected that is generated by the fiber-optic receiver. Sufficient I/O and logic resources are still available in the FPGA to add this into the state machine equations.

Design Summary

The small size of the FPGA design is made possible by the enhanced functionality present in the HOTLink transmitter and receiver. This removes the need to design and implement the 8B/10B encoders and decoders, and provides full received character validation. The embedded PECL-to-TTL converter also allows a small footprint by removing the need for an external conversion circuit.

The VHDL design both auto-routes and auto-places into a pASIC383 FPGA. Because of the high-speed operation of the pASIC cells and interconnect, this design meets or exceeds all design performance parameters, over worst case temperature and voltage, using the slow −0 speed bin of the pASIC383.

The 100% routability of the pASIC family allows the circuit board signal routing to be improved by selecting pins that best match the system interconnect. The pinouts listed in the top-level VHDL file were selected to allow straight-through routing (no cross-overs) of the signals between the FPGA and the HOTLink transmitter and receiver. In addition, the

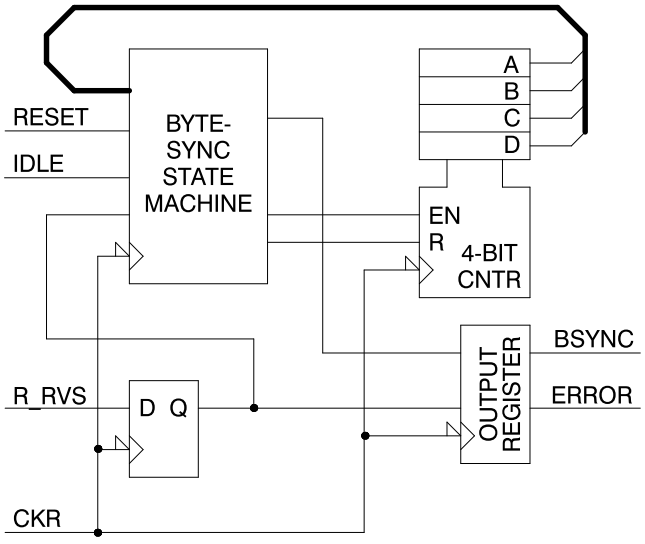

Figure 11. Byte-Sync State Machine Block Diagram

placement of the HOTLink transmitter and receiver were selected to line up with the transmit and receive halves of the fiber-optic transceiver. This pinout selection and interconnect are shown in *Figure 12*.

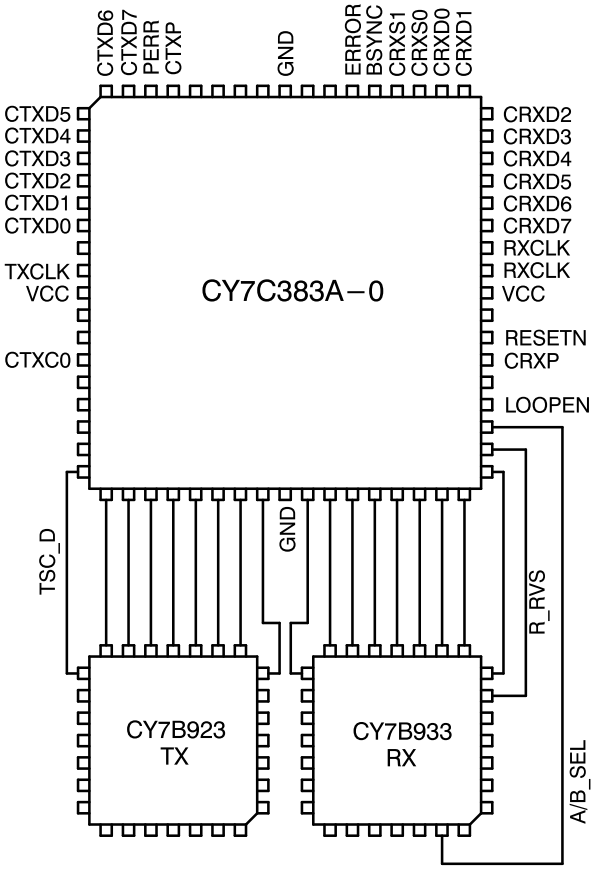

Figure 12. HOTLink/pASIC Pinout and Interconnect

Conclusions

The ESCON interface is both an elegant and powerful replacement for the older block-mux channels. The use of the HOTLink serializer/deserializer components to implement an ESCON interface guarantees both compliance with the 8B/10B coding rules and all jitter and timing specifications of the ESCON interface.

Due to the high-speed operation of the ESCON interface, the byte-level control is best implemented in hardware. The flexibility of the VHDL language and the unlimited routing of the Cypress pASIC family of FPGAs make them a perfect choice for building the control state machines. While only the lower level of the ESCON protocol is controlled in the design documented here, much of the higher level control may also be implemented through the use of either larger or additional FPGA components.

References

1. *ESCON I/O Interface*, IBM, 1990, 1991
2. *HOTLink User's Guide,* Cypress Semiconductor, 1995
3. *GA9104 Datasheet,* Triquint Semiconductor, Inc, 1992

Appendix A. Top-Level pASIC Code

```
-- ESC_TOP.VHD
--
-- ESCON Interface Control PLD
-- Equivalent to the Triquint GA9104 but designed for operation
-- with the Cypress Semiconductor HOTLink chipset
ENTITY esc_top IS PORT (
  txclk: IN BIT;                   -- transmit path byte clock
  rxclkA: IN BIT;                  -- receiver path byte clock
  rxclkB: IN BIT;                  -- receiver path byte clock
  resetn: IN BIT;                  -- active low reset
  rxq: INOUT X01Z_VECTOR(0 TO 7);  -- HOTLink receiver data in
  rsc_d: INOUT X01Z;               -- HOTLink receiver SC/D
  r_rvs: INOUT X01Z;               -- HOTLink receiver RVS
  txd: INOUT X01Z_VECTOR(0 TO 7);  -- HOTLink transmitter data out
  tsc_d: INOUT X01Z;               -- HOTLink transmitter SC/D
  crxd: INOUT X01Z_VECTOR(0 TO 7); -- receive path data output
  ctxd: INOUT X01Z_VECTOR(0 TO 7); -- transmit path data input
  crxs0: INOUT X01Z;               -- receive status 0 (command/data)
  crxs1: INOUT X01Z;               -- receive status 1 (CRC)
  ctxc0: INOUT X01Z;               -- transmit control 0 (command/data)
  bsync: INOUT X01Z;               -- byte sync acquired
  error: INOUT X01Z;               -- receive bad character error
  perr: INOUT X01Z;                -- transmit-in parity error
  crxp: INOUT X01Z;                -- odd parity output
  ctxp: INOUT X01Z;                -- odd parity input
  loopen: INOUT X01Z;              -- local loopback enable
  ab_sel: INOUT X01Z);             -- receiver A/B select

ATTRIBUTE part_name OF esc_top:ENTITY IS "C383A";
ATTRIBUTE pin_numbers OF esc_top:ENTITY IS
    "txclk:17 rxclkA:53 rxclkB:54 resetn:50 rxq(7):44 rxq(6):43 "
  & "rxq(5):42 rxq(4):41 rxq(3):40 rxq(2):39 rxq(1):38 rxq(0):37 "
  & "rsc_d:36 r_rvs:45 txd(7):34 txd(6):33 txd(5):32 txd(4):31 "
  & "txd(3):30 txd(2):29 txd(1):28 txd(0):27 tsc_d:26 crxd(0):62 "
  & "crxd(1):61 crxd(2):60 crxd(3):59 crxd(4):58 crxd(5):57 "
  & "crxd(6):56 crxd(7):55 ctxd(0):15 ctxd(1):14 ctxd(2):13 "
  & "ctxd(3):12 ctxd(4):11 ctxd(5):10 ctxd(6):9 ctxd(7):8 "
  & "crxs0:63 crxs1:64 ctxc0:21 bsync:65 error:66 perr:7 "
  & "crxp:49 ctxp:6 loopen:47 ab_sel:46";

END esc_top;

USE work.cypress.all;
USE work.rtlpkg.all;
USE work.memorypkg.all;
USE work.ttlpkg.all;
USE work.registerpkg.all;
```

Appendix A. Top-Level pASIC Code (continued)

```
USE work.iopkg.all;
USE work.mcpartspkg.all;
USE work.gatespkg.all;
USE work.resolutionpkg.all;     -- used to double-buffer
USE work.bv_math.all;           -- allow use of INV function
USE work.crc_t.all;             -- add in CRC transmit function
USE work.crc_r.all;             -- add in CRC receive function
USE work.crc_ctl.all;           -- add in transmit CRC control machine
USE work.sync_det.all;          -- add in SYNC detect state machine
USE work.triq_code.all;         -- add in command decoder section
USE work.iopluspkg.all;         -- add in enhanced I/O buffers

ARCHITECTURE escon_top OF esc_top IS
-- add internal signal equivalents of signals after I/O pads
SIGNAL tclk : BIT;                 -- transmit clock
SIGNAL rclk : BIT;                 -- negative edge receiver clock
SIGNAL reset : BIT;                -- reset controller
SIGNAL HL_rx : BIT_VECTOR(0 to 7); -- HOTLink receiver data bus
SIGNAL HL_rsc_d : BIT;             -- HOTLink receiver SC/D
SIGNAL HL_r_rvs : BIT;             -- HOTlink receiver RVS
SIGNAL HL_tx : BIT_VECTOR(0 to 7); -- HOTLink transmitter data bus
SIGNAL HL_tsc_d : BIT;             -- HOTLink transmitter SC/D
SIGNAL HL_tsc_q : BIT;             -- clocked HOTLink transmitter SC/D
SIGNAL sync_r : BIT;               -- receiver byte sync
SIGNAL c_rxd : BIT_VECTOR(0 to 7); -- controller receive path data out
SIGNAL c_txd : BIT_VECTOR(0 to 7); -- controller transmit path dataout
SIGNAL c_rxs0 : BIT;               -- receive status 0 (command/data)
SIGNAL c_rxs1 : BIT;               -- receive status 1 (CRC)
SIGNAL c_txc0 : BIT;               -- transmit control 0 (command/data)
SIGNAL b_sync : BIT;               -- byte sync acquired
SIGNAL r_error : BIT;              -- receive bad character error
SIGNAL p_err : BIT;                -- parity error
SIGNAL c_rxp : BIT;                -- odd parity output
SIGNAL c_txp : BIT;                -- odd parity input
SIGNAL b_loopen : BIT;             -- buffered loop enable

-- transmit internal signals
SIGNAL t_data : BIT_VECTOR(0 TO 7);  -- transmit data bus
SIGNAL t_mux : BIT_VECTOR(0 TO 7);   -- muxed transmit data path
SIGNAL t_comm : BIT_VECTOR(0 TO 7);  -- re-encoded transmit commands
SIGNAL tp_odd : BIT;                 -- transmit data parity input
SIGNAL t_parity : BIT;               -- transmit parity checker output
```

Appendix A. Top-Level pASIC Code (continued)

```
SIGNAL t_CRC : BIT_VECTOR(0 TO 7);   -- transmit CRC vector
SIGNAL c_txc_0 : BIT;                -- transmit command/data
SIGNAL mux_hi : BIT;                 -- enable HI/LOW transmit CRC byte
SIGNAL mux_low : BIT;                -- enable LOW transmit CRC byte
sIGNAL ctxc3 : BIT;                  -- 3x registered c_txc_0
SIGNAL t_CRC_reset : BIT;            -- preset transmit CRC register
-- receive internal signals
SIGNAL r_data : BIT_VECTOR(0 TO 7);  -- registered receiver data bus
SIGNAL r_mux : BIT_VECTOR(0 TO 7);   -- muxed data and translated commands
SIGNAL rp_odd : BIT;                 -- receive data parity output
SIGNAL rcom_data : BIT;              -- registered SC/D pin
SIGNAL r_com_data: multi_buffer BIT; -- double buffered registerd SC/D pin
SIGNAL r_crc_err : BIT;              -- un-registered CRC status
SIGNAL r_CRC_d : BIT;                -- CRC check D-input
SIGNAL rvs : BIT;                    -- registered RVS signal
SIGNAL sync : BIT;                   -- decoded K28.5 signal
SIGNAL t_code : BIT_VECTOR(0 to 7);  -- Triquint pattern for K-codes

--------------------------------------------------------------------------
--------------------------------------------------------------------------
BEGIN
-- instantiate pASIC buffers/drivers on I/O signals
-- clocks
p1: CKPAD PORT MAP (txclk, tclk);       -- transmit path clock
p2: HDI2PAD PORT MAP (rxclkA, rxclkB, rclk); -- receive path clock on
                                             -- on negative edge
-- high drive pads
p3: HDIPAD PORT MAP (resetn ,reset);    -- active HIGH system reset
-- data buses
-- HOTLink receiver data bus (input)
p4:  INPAD PORT MAP (rxq(0), HL_rx(0));
p5:  INPAD PORT MAP (rxq(1), HL_rx(1));
p6:  INPAD PORT MAP (rxq(2), HL_rx(2));
p7:  INPAD PORT MAP (rxq(3), HL_rx(3));
p8:  INPAD PORT MAP (rxq(4), HL_rx(4));
p9:  INPAD PORT MAP (rxq(5), HL_rx(5));
p10: INPAD PORT MAP (rxq(6), HL_rx(6));
p11: INPAD PORT MAP (rxq(7), HL_rx(7));
p12: INPAD PORT MAP (rsc_d,  HL_rsc_d);        -- receive SC/D
p13: INPAD PORT MAP (r_rvs,  HL_r_rvs);        -- RVS
-- HOTLink transmitter data bus (output)
p14: OUTPAD PORT MAP (HL_tx(0), txd(0));
p15: OUTPAD PORT MAP (HL_tx(1), txd(1));
p16: OUTPAD PORT MAP (HL_tx(2), txd(2));
p17: OUTPAD PORT MAP (HL_tx(3), txd(3));
p18: OUTPAD PORT MAP (HL_tx(4), txd(4));
p19: OUTPAD PORT MAP (HL_tx(5), txd(5));
```

Appendix A. Top-Level pASIC Code (continued)

```
p20: OUTPAD PORT MAP (HL_tx(6), txd(6));
p21: OUTPAD PORT MAP (HL_tx(7), txd(7));
p22: OUTPAD PORT MAP (HL_tsc_q, tsc_d);
-- controller transmit data bus (input)
p24: INPAD PORT MAP (ctxd(0), c_txd(0));
p25: INPAD PORT MAP (ctxd(1), c_txd(1));
p26: INPAD PORT MAP (ctxd(2), c_txd(2));
p27: INPAD PORT MAP (ctxd(3), c_txd(3));
p28: INPAD PORT MAP (ctxd(4), c_txd(4));
p29: INPAD PORT MAP (ctxd(5), c_txd(5));
p30: INPAD PORT MAP (ctxd(6), c_txd(6));
p31: INPAD PORT MAP (ctxd(7), c_txd(7));
-- controller receiver data bus (output)
p34: OUTPAD PORT MAP (c_rxd(0), crxd(0));
p35: OUTPAD PORT MAP (c_rxd(1), crxd(1));
p36: OUTPAD PORT MAP (c_rxd(2), crxd(2));
p37: OUTPAD PORT MAP (c_rxd(3), crxd(3));
p38: OUTPAD PORT MAP (c_rxd(4), crxd(4));
p39: OUTPAD PORT MAP (c_rxd(5), crxd(5));
p40: OUTPAD PORT MAP (c_rxd(6), crxd(6));
p41: OUTPAD PORT MAP (c_rxd(7), crxd(7));
-- misc input pads
p44: INPAD PORT MAP (loopen, b_loopen); -- loopback enable
p45: INPAD PORT MAP (ctxc0, c_txc0);    -- transmit control 0
p49: INPAD PORT MAP (ctxp, c_txp);      -- odd parity input
-- misc output pads
p50: OUTPAD PORT MAP (c_rxs0, crxs0);   -- receiver status 0 output
p51: OUTPAD PORT MAP (c_rxs1, crxs1);   -- receiver status 1 output
p53: OUTPAD PORT MAP (b_sync, bsync);   -- byte sync acquired
p54: OUTPAD PORT MAP (r_error, error);  -- received bad character
p55: OUTPAD PORT MAP (p_err, perr);     -- parity error
p56: OUTPAD PORT MAP (c_rxp, crxp);     -- odd parity output
p57: OUTPAD PORT MAP (INV(b_loopen),ab_sel); -- HOTLink receiver A/B select
-------------------------------------------------------------------------
-------------- TRANSMIT PATH --------------------------------------------
-------------------------------------------------------------------------
-- add in transmit path input data pipeline register
t1a: DFF PORT MAP (c_txd(0), tclk, t_data(0));
t1b: DFF PORT MAP (c_txd(1), tclk, t_data(1));
t1c: DFF PORT MAP (c_txd(2), tclk, t_data(2));
t1d: DFF PORT MAP (c_txd(3), tclk, t_data(3));
t1e: DFF PORT MAP (c_txd(4), tclk, t_data(4));
t1f: DFF PORT MAP (c_txd(5), tclk, t_data(5));
t1g: DFF PORT MAP (c_txd(6), tclk, t_data(6));
t1h: DFF PORT MAP (c_txd(7), tclk, t_data(7));
```

Appendix A. Top-Level pASIC Code (continued)

```
-- add parity and control bits
t1j: DFF PORT MAP (c_txp, tclk, tp_odd);
t1k: DFF PORT MAP (c_txc0, tclk, c_txc_0);
-------------------------------------------------------------------------
-- add transmit data parity checker (10 bit parity tree)
t_parity <= NOT(t_data(0) XOR t_data(1) XOR t_data(2) XOR t_data(3)
  XOR t_data(4) XOR t_data(5) XOR t_data(6) XOR t_data(7)
  XOR tp_odd XOR c_txc_0);
-------------------------------------------------------------------------
-- add parity check F-F
t2: DFF PORT MAP (
  t_parity,                          -- parity of inputs
  tclk,                              -- transmit clock
  p_err);                            -- output parity status
-------------------------------------------------------------------------
-- add transmitter CRC generator
t3: crc_tx PORT MAP (
  tclk,                           -- transmit clock
  t_CRC_reset,                    -- from tx CRC control state machine
  c_txc_0,                        -- from tx input register
  mux_hi,                         -- enable low byte onto bus
  t_data,                         -- transmit data bus
  t_CRC);                         -- 8-bit transmit CRC output vector
t_CRC_reset <= '0' WHEN (c_txc_0 = '0' OR mux_hi = '0') ELSE '1';
-------------------------------------------------------------------------
-- add transmit output register
t5a: DFF PORT MAP (t_mux(0), tclk, HL_tx(0));
t5b: DFF PORT MAP (t_mux(1), tclk, HL_tx(1));
t5c: DFF PORT MAP (t_mux(2), tclk, HL_tx(2));
t5d: DFF PORT MAP (t_mux(3), tclk, HL_tx(3));
t5e: DFF PORT MAP (t_mux(4), tclk, HL_tx(4));
t5f: DFF PORT MAP (t_mux(5), tclk, HL_tx(5));
t5g: DFF PORT MAP (t_mux(6), tclk, HL_tx(6));
t5h: DFF PORT MAP (t_mux(7), tclk, HL_tx(7));
HL_tsc_d <= (mux_low AND c_txc_0) OR
            (c_txc_0 AND mux_hi AND ctxc3);
-- add in SC/D output bit
t5j: DFF PORT MAP (HL_tsc_d, tclk, HL_tsc_q);

-------------------------------------------------------------------------
-- add in transmit CRC supervisor machine
-- contains the double pipelined C/D bit
t6: tx_ctl_crc PORT MAP (
  tclk,                    -- transmit clock
  c_txc_0,                 -- registerd command/data control bit
  mux_hi,                  -- registered c_txc_0
  mux_low);                -- 2x registered c_txc_0
```

Appendix A. Top-Level pASIC Code (continued)

```
-------------------------------------------------------------------------
-- transmit path data/command/CRC mux
t8: PROCESS (c_txc_0, mux_low, mux_hi)
BEGIN
  IF (c_txc_0 = '0') THEN
    t_mux <= t_data;
  ELSIF (c_txc_0 = '1' AND ((mux_low = '0' AND mux_hi='0') OR
        (ctxc3 = '0' AND mux_low = '0' AND mux_hi = '1'))) THEN
    -- output CRC bytes
    t_mux <= t_CRC;
  ELSE
    -- output re-encoded command codes
    t_mux <= t_comm;
  END IF;
END PROCESS t8;

-- Add in transmit command decoder
t9: t_decode PORT MAP (t_data, t_comm); -- translate to HOTLink commands

-------------------------------------------------------------------------
-------------- RECEIVE PATH -------------------------------------------
-------------------------------------------------------------------------
-- add in receive path input data pipeline register
r1a: DFF PORT MAP (HL_rx(0), rclk, r_data(0));
r1b: DFF PORT MAP (HL_rx(1), rclk, r_data(1));
r1c: DFF PORT MAP (HL_rx(2), rclk, r_data(2));
r1d: DFF PORT MAP (HL_rx(3), rclk, r_data(3));
r1e: DFF PORT MAP (HL_rx(4), rclk, r_data(4));
r1f: DFF PORT MAP (HL_rx(5), rclk, r_data(5));
r1g: DFF PORT MAP (HL_rx(6), rclk, r_data(6));
r1h: DFF PORT MAP (HL_rx(7), rclk, r_data(7)); -- add SC/D bit and RVS
r1j: DFF PORT MAP (HL_rsc_d, rclk, rcom_data); -- registerd SC/D
r1k: DFF PORT MAP (HL_r_rvs, rclk, rvs);        -- registered RVS signal
-- create double buffered signals
db1: BUF PORT MAP (rcom_data, r_com_data);
db2: BUF PORT MAP (rcom_data, r_com_data);
-------------------------------------------------------------------------
-- receive path output register
r2a: DFF PORT MAP (r_mux(0), rclk, c_rxd(0));
r2b: DFF PORT MAP (r_mux(1), rclk, c_rxd(1));
r2c: DFF PORT MAP (r_mux(2), rclk, c_rxd(2));
r2d: DFF PORT MAP (r_mux(3), rclk, c_rxd(3));
r2e: DFF PORT MAP (r_mux(4), rclk, c_rxd(4));
r2f: DFF PORT MAP (r_mux(5), rclk, c_rxd(5));
r2g: DFF PORT MAP (r_mux(6), rclk, c_rxd(6));
r2h: DFF PORT MAP (r_mux(7), rclk, c_rxd(7));-- command/data bit and rvs
r2j: DFF PORT MAP (r_com_data, rclk, c_rxs0);
r2k: DFF PORT MAP (rvs, rclk, r_error);
```

Appendix A. Top-Level pASIC Code (continued)

```
-------------------------------------------------------------------------
-- add receive parity generate
r3: TTL180 PORT MAP (
  r_mux(0), r_mux(1), r_mux(2), r_mux(3), r_mux(4), r_mux(5),
  r_mux(6), r_mux(7), INV(r_com_data), r_com_data, rp_odd, open);

r3a: DFF PORT MAP (rp_odd, rclk, c_rxp);
-------------------------------------------------------------------------
-- add in receive CRC block
r4: crc_rx PORT MAP (
  rclk,                   -- receive path clock
  r_com_data,             -- enable only for data bytes
  r_data,                 -- receiver data bus
  r_crc_err);             -- receive path crc status
-------------------------------------------------------------------------
-- add CRC check register
r5: DFF PORT MAP (r_CRC_d, rclk, c_rxs1);
r_CRC_d <= r_crc_err AND r_com_data AND (NOT(c_rxs0));
-------------------------------------------------------------------------
-- add in byte-sync state machine
r6: byte_syn PORT MAP (
  rclk,                   -- receiver clock
  reset,                  -- system reset
  rvs,                    -- receiver RVS signal
  sync,                   -- decoded k28.5
  b_sync);                -- byte sync acquired

sync <= '1' WHEN (r_com_data='1' AND r_data(0 TO 3)="1010") ELSE '0';
-------------------------------------------------------------------------
-- add command transposition logic and mux
r7: PROCESS (r_com_data, r_data(0), r_data(1), r_data(2), r_data(3))
BEGIN
  IF (r_com_data='0') THEN
    r_mux <= r_data;
  ELSE
    r_mux <= t_code;        -- add in command decoder
  END IF;
END PROCESS r7;
-------------------------------------------------------------------------
-- add receiver path command encoder
-- t_code is output vector
r8: t_encode PORT MAP (
  r_data,                 -- HOTLink data bus
  t_code);                -- decoded Triquint commands

END escon_top;
```

Appendix B. Transmit Path CRC Generator

```
-- CRC_T.VHD
--
-- transmit 16-bit CCITT CRC for use in data mover
--
-- When sequencing bytes out, the qt(15)-qt(8) byte must be sent out first.
-- Per the ESCON spec, the CRC is the 1's compliment (inversion) of the
-- qt[15:0] bus.
--
PACKAGE crc_T IS
  COMPONENT crc_tx PORT (
  clk,                    -- system clock
  preset: IN      BIT;    -- synchronous preset, set to all 1s
  enable: IN      BIT;    -- enable when not a command byte
  mux_hi: IN      BIT;    -- enable high-byte onto bus
  dt:     IN      BIT_VECTOR (0 TO 7); -- Input data byte
  q_out:  OUT     BIT_VECTOR (0 TO 7)  -- CRC register
  );
  END COMPONENT;
END crc_T;

use work.rtlpkg.all;
use work.cypress.all;

ENTITY crc_tx IS PORT (
  clk,                    -- system clock
  preset: IN      BIT;    -- synchronous reset, set to all 1s
  enable: IN      BIT;    -- enable when not a command byte
  mux_hi: IN      BIT;    -- enable high CRC byte out
  dt:     IN      BIT_VECTOR (0 TO 7); -- Input data byte
  q_out:  OUT     BIT_VECTOR (0 TO 7)  -- CRC register
  );
END crc_tx;

ARCHITECTURE ccitt_tx OF crc_tx IS
SIGNAL qt: BIT_VECTOR (0 TO 15); -- CRC register
BEGIN
proc1: PROCESS BEGIN
  WAIT UNTIL (clk='1');
    IF (preset='1') THEN
      qt <= x"FFFF";     -- Preset to 1's for reset
    ELSIF (enable='1') THEN
      qt <= qt;          -- keep same value
    ELSE
      qt(0) <= qt(8) XOR qt(12) XOR dt(3) XOR dt(7);
      qt(1) <= qt(9) XOR qt(13) XOR dt(2) XOR dt(6);
      qt(2) <= qt(10) XOR qt(14) XOR dt(1) XOR dt(5);
      qt(3) <= qt(11) XOR qt(15) XOR dt(0) XOR dt(4);
```

Appendix B. Transmit Path CRC Generator (continued)

```
      qt(4) <= qt(12) XOR dt(3);
      qt(5) <= qt(13) XOR qt(12) XOR qt(8) XOR dt(7) XOR dt(3) XOR dt(2);
      qt(6) <= qt(14) XOR qt(13) XOR qt(9) XOR dt(1)XOR dt(2) XOR dt(6);
      qt(7) <= qt(15) XOR qt(14) XOR qt(10) XOR dt(0) XOR dt(1) XOR dt(5);
      qt(8) <= qt(15) XOR qt(11) XOR qt(0) XOR dt(0) XOR dt(4);
      qt(9) <= qt(12) XOR qt(1) XOR dt(3);
      qt(10) <= qt(13) XOR qt(2) XOR dt(2);
      qt(11) <= qt(14) XOR qt(3) XOR dt(1);
      qt(12) <= qt(15) XOR qt(12) XOR qt(8) XOR qt(4)
                XOR dt(0) XOR dt(3) XOR dt(7);
      qt(13) <= qt(13) XOR qt(9) XOR qt(5) XOR dt(2) XOR dt(6);
      qt(14) <= qt(14) XOR qt(10) XOR qt(6) XOR dt(1) XOR dt(5);
      qt(15) <= qt(15) XOR qt(11) XOR qt(7) XOR dt(0) XOR dt(4);
    END IF;
END PROCESS;

-- mux and Invert CRC and swap bits
m1: PROCESS (mux_hi)
BEGIN
-- Mux out high and low bytes and transpose bit order
IF mux_hi = '0' THEN
  q_out(7) <= not qt(8);
  q_out(6) <= not qt(9);
  q_out(5) <= not qt(10);
  q_out(4) <= not qt(11);
  q_out(3) <= not qt(12);
  q_out(2) <= not qt(13);
  q_out(1) <= not qt(14);
  q_out(0) <= not qt(15);
ELSE
  q_out(7) <= not qt(0);
  q_out(6) <= not qt(1);
  q_out(5) <= not qt(2);
  q_out(4) <= not qt(3);
  q_out(3) <= not qt(4);
  q_out(2) <= not qt(5);
  q_out(1) <= not qt(6);
  q_out(0) <= not qt(7);
END IF;
END PROCESS m1;

END ccitt_tx;
```

Appendix C. Transmit Path CRC Controler

```
-- CTL_CRCT.VHD
--
-- Control transmit CRC function
--
--
-- All actions are based on the CTXC0 input.  This input is active
-- at the end of every data sequence and is a 1 (HIGH) for all
-- non-data bytes.
--

PACKAGE crc_ctl IS
  COMPONENT tx_ctl_crc PORT (
  clk,                    -- transmit clock
  ctxc0: IN BIT;          -- command/data control bit
  ctxc1,                  -- registered ctxc0
  ctxc2,                  -- 2x registered ctxc0
  ctxc3: OUT BIT);        -- 3x registered ctxc0
  END COMPONENT;
END crc_ctl;

ENTITY tx_ctl_crc IS PORT (
  clk,                    -- transmit clock
  ctxc0: IN BIT;          -- command/data control bit
  ctxc1,                  -- registered ctxc0
  ctxc2,                  -- 2x registered ctxc0
  ctxc3: OUT BIT);        -- 3x registered ctxc0
END tx_ctl_crc;

USE work.cypress.all;
USE work.rtlpkg.all;

ARCHITECTURE ctl_1 OF tx_ctl_crc IS

SIGNAL cq1: BIT;               -- single registered c/d
SIGNAL cq2: BIT;               -- double registered c/d

BEGIN

-- Instantiate DFF to track status of ctxc0 bit
d1: DFF PORT MAP (ctxc0, clk, cq1);
d2: DFF PORT MAP (cq1, clk, cq2);
d3: DFF PORT MAP (cq2, clk, ctxc3);

-- assign outputs
ctxc1 <= cq1;
ctxc2 <= cq2;

END ctl_1;
```

Appendix D. Command Mapper

```
-- TRI_CODE.VHD
--
-- Command decode/translate between the Triquint GA9104 and HOTLink
-- K-code command sets

-------------------------------------------------------------------------
-- Triquint/Cypress Command mapping
--         GA9104          HOTLink HEX
--         HEX BIN         TX RX        BIN
-- k28.0*  1C 00011100     00           00000000
-- k28.1   3C 00111100     01           00000001
-- k28.2   5C 01011100     02           00000010
-- k28.3*  7C 01111100     03           00000011
-- k28.4   9C 10011100     04           00000100
-- k28.5   BC 10111100     05,E1,E2     00000101
-- k28.6   DC 11011100     06           00000110
-- k28.7   FC 11111100     07,27,47     00000111
-- k23.7*  F7 11110111     08           00001000
-- k27.7*  FB 11111011     09           00001001
-- k29.7*  FD 11111101     0A           00001010
-- k30.7*  FE 11111110     0B           00001011
-- * - Illegal for use in ESCON operations

PACKAGE triq_code IS
  COMPONENT t_encode PORT (
  c_code : IN BIT_VECTOR(0 TO 7); -- Cypress HOTLink C-codes
  t_code : OUT BIT_VECTOR(0 TO 7) -- Triquint K-codes
  );
  END COMPONENT;

  COMPONENT t_decode PORT (
  t_data : IN BIT_VECTOR(0 TO 7); -- Triquint K-codes
  t_comm : OUT BIT_VECTOR(0 TO 7) -- Cypress HOTLink C-codes
  );
  END COMPONENT;
END triq_code;

USE work.cypress.all;
USE work.table_bv.all;          -- use for command encoder

ENTITY t_encode IS PORT (
  c_code : IN BIT_VECTOR(0 TO 7); -- Cypress HOTLink C-codes
  t_code : OUT BIT_VECTOR(0 TO 7) -- Triquint K-codes
  );
END t_encode;
```

Appendix D. Command Mapper (continued)

```
ARCHITECTURE t_encoder OF t_encode IS
-- use TTF function to translate from one command set to the other
-- Command constants
-- T-codes (output vectors)
CONSTANT K28_0: x01_VECTOR(0 TO 7) := "00111000";
CONSTANT K28_1: x01_VECTOR(0 TO 7) := "00111100";
CONSTANT K28_2: x01_VECTOR(0 TO 7) := "00111010";
CONSTANT K28_3: x01_VECTOR(0 TO 7) := "00111110";
CONSTANT K28_4: x01_VECTOR(0 TO 7) := "00111001";
CONSTANT K28_5: x01_VECTOR(0 TO 7) := "00111101";
CONSTANT K28_6: x01_VECTOR(0 TO 7) := "00111011";
CONSTANT K28_7: x01_VECTOR(0 TO 7) := "00111111";
CONSTANT K23_7: x01_VECTOR(0 TO 7) := "11101111";
CONSTANT K27_7: x01_VECTOR(0 TO 7) := "11011111";
CONSTANT K29_7: x01_VECTOR(0 TO 7) := "10111111";
CONSTANT K30_7: x01_VECTOR(0 TO 7) := "01111111";
-- C-codes (input vectors)
CONSTANT C00_0: x01_VECTOR(0 TO 7) := "0000xxxx";
CONSTANT C01_0: x01_VECTOR(0 TO 7) := "1000xxx0";
CONSTANT C02_0: x01_VECTOR(0 TO 7) := "0100xxx0";
CONSTANT C03_0: x01_VECTOR(0 TO 7) := "1100xxxx";
CONSTANT C04_0: x01_VECTOR(0 TO 7) := "0010xxxx";
CONSTANT C05_0: x01_VECTOR(0 TO 7) := "1010xxxx";
CONSTANT C06_0: x01_VECTOR(0 TO 7) := "0110xxxx";
CONSTANT C07_0: x01_VECTOR(0 TO 7) := "1110xxxx";
CONSTANT C08_0: x01_VECTOR(0 TO 7) := "0001xxxx";
CONSTANT C09_0: x01_VECTOR(0 TO 7) := "1001xxxx";
CONSTANT C10_0: x01_VECTOR(0 TO 7) := "0101xxxx";
CONSTANT C11_0: x01_VECTOR(0 TO 7) := "1101xxxx";
CONSTANT C12_0: x01_VECTOR(0 TO 7) := "0011xxxx";
-- errors and special mappings
CONSTANT C01_7: x01_VECTOR(0 TO 7) := "1000xxx1";
CONSTANT C02_7: x01_VECTOR(0 TO 7) := "0100xxx1";

CONSTANT table: x01_TABLE(0 TO 13, 0 TO 15) := (   -- command mappings
--         Command
--      Input   Output
--      -----   ------
        C00_0 & K28_0,
        C01_0 & K28_1,
        C02_0 & K28_2,
        C03_0 & K28_3,
        C04_0 & K28_4,
        C05_0 & K28_5,
        C06_0 & K28_6,
        C07_0 & K28_7,
```

Appendix D. Command Mapper (continued)

```
        C08_0 & K23_7,
        C09_0 & K27_7,
        C10_0 & K29_7,
        C11_0 & K30_7,
        C01_7 & K28_5,
        C02_7 & K28_5);

BEGIN
p1: PROCESS (c_code)
  BEGIN
    t_code <= ttf(table,(c_code));
  END PROCESS p1;
END t_encoder;

USE work.cypress.all;

ENTITY t_decode IS PORT (
  t_data : IN BIT_VECTOR(0 TO 7); -- Triquint K-codes
  t_comm : OUT BIT_VECTOR(0 TO 7) -- Cypress HOTLink C-codes
  );
END t_decode;

ARCHITECTURE t_decoder OF t_decode IS

BEGIN

t_comm(7) <= '0';
t_comm(6) <= '0';
t_comm(5) <= '0';
t_comm(4) <= '0';
t_comm(3) <= '0' WHEN (t_data(0 TO 1) = "00") ELSE '1';
t_comm(2) <= '1' WHEN ((t_data(7) = '1')
             AND (t_data(0 TO 1) = "00")) ELSE '0';

t1: PROCESS (t_data(0), t_data(1), t_data(6),
           t_data(5), t_data(3), t_data(2))
  BEGIN
    IF (t_data(0 TO 1) = "00") THEN
      t_comm(1) <= t_data(6);
      t_comm(0) <= t_data(5);
    ELSE
      t_comm(1) <= t_data(3) AND t_data(2);
      t_comm(0) <= t_data(2) AND t_data(0);
    END IF;
  END PROCESS t1;

END t_decoder;
```

Appendix E. Receive Path CRC Checker

```
-- CRC_R.VHD
--
-- receiver 16-bit CCITT CRC for use in data mover
PACKAGE crc_r IS
  COMPONENT crc_rx PORT (
  clk,                    -- system clock
  preset: IN      BIT;    -- synchronous reset, set to all 1s
  dr:      IN      BIT_VECTOR (0 TO 7); -- Input data byte
  crc_err:  OUT   BIT     -- error detected
  );
  END COMPONENT;
END crc_r;

use work.rtlpkg.all;
use work.cypress.all;

ENTITY crc_rx IS PORT (
  clk,                    -- system clock
  preset:  IN BIT;        -- synchronous preset, set to all 1s
  dr:      IN      BIT_VECTOR (0 TO 7); -- Input data byte
  crc_err:  OUT   BIT     -- error detected
  );
END crc_rx;

ARCHITECTURE ccitt_rx OF crc_rx IS
-- declare CRC register
SIGNAL qr: BIT_VECTOR (0 TO 15);    -- CRC register
ATTRIBUTE POLARITY OF qr:SIGNAL IS PL_KEEP;  -- maintain polarity f
BEGIN
proc1: PROCESS BEGIN
  WAIT UNTIL (clk='1');
    IF (preset='1') THEN
      qr <= x"FFFF";     -- Preset to 1's for reset
    ELSE
      qr(0) <= qr(8) XOR qr(12) XOR dr(3) XOR dr(7);
      qr(1) <= qr(9) XOR qr(13) XOR dr(2) XOR dr(6);
      qr(2) <= qr(10) XOR qr(14) XOR dr(1) XOR dr(5);
      qr(3) <= qr(11) XOR qr(15) XOR dr(0) XOR dr(4);
      qr(4) <= qr(12) XOR dr(3);
      qr(5) <= qr(13) XOR qr(12) XOR qr(8) XOR dr(7) XOR dr(3) XOR dr(2);
      qr(6) <= qr(14) XOR qr(13) XOR qr(9) XOR dr(1) XOR dr(2) XOR dr(6);
      qr(7) <= qr(15) XOR qr(14) XOR qr(10) XOR dr(0) XOR dr(1) XOR dr(5);
      qr(8) <= qr(15) XOR qr(11) XOR qr(0) XOR dr(0) XOR dr(4);
      qr(9) <= qr(12) XOR qr(1) XOR dr(3);
      qr(10) <= qr(13) XOR qr(2) XOR dr(2);
      qr(11) <= qr(14) XOR qr(3) XOR dr(1);
```

Appendix E. Receive Path CRC Checker (continued)

```
      qr(12) <= qr(15) XOR qr(12) XOR qr(8) XOR qr(4)
                XOR dr(0) XOR dr(3) XOR dr(7);
      qr(13) <= qr(13) XOR qr(9) XOR qr(5) XOR dr(2) XOR dr(6);
      qr(14) <= qr(14) XOR qr(10) XOR qr(6) XOR dr(1) XOR dr(5);
      qr(15) <= qr(15) XOR qr(11) XOR qr(7) XOR dr(0) XOR dr(4);
    END IF;
END PROCESS;

-- Need to look for a 1D0F at the receiver
-- output is LOW when 1D0F present
crc_err <= NOT(qr(0)) OR NOT(qr(1)) OR NOT(qr(2)) OR NOT(qr(3))
      OR qr(4) OR qr(5) OR qr(6) OR qr(7)
      OR NOT(qr(8)) OR qr(9) OR NOT(qr(10)) OR NOT(qr(11))
      OR NOT(qr(12)) OR qr(13) OR qr(14) OR qr(15);

END ccitt_rx;
```

Appendix F. Byte Sync Controller

```
-- B_SYNC.VHD – byte synchronization state machine
--
-- This machine has a five state supervisor machine that tracks
-- the number of errors detected within a specific period of time.
-- It also tracks valid characters and SYNC codes.

PACKAGE sync_det IS
  COMPONENT byte_syn PORT (
  clk,                    -- Receiver clock
  reset,                  -- system reset
  error,                  -- bad character
  sync: IN BIT;           -- valid k28.5
  bsync: OUT BIT);        -- byte–sync acquired
  END COMPONENT;
END sync_det;

USE work.cypress.all;
USE work.rtlpkg.all;
USE work.counterpkg.all;

ENTITY byte_syn IS PORT (
  clk,                    -- Receiver clock
  reset,                  -- system reset
  error,                  -- bad character
  sync: IN BIT;           -- valid k28.5
  bsync: OUT BIT);        -- byte–sync acquired
END byte_syn;

ARCHITECTURE arch1 OF byte_syn IS
-- declare internal signals
SIGNAL ctr_en: BIT;             -- counter enable
SIGNAL ctr_reset: BIT;          -- counter reset
SIGNAL bbsync: BIT;             -- interface in sync
SIGNAL cnt: BIT_VECTOR(0 TO 3); -- 4–bit counter vector
-- declare state machine
TYPE sync_state IS (
        state0,                 -- reset or errors, waiting for SYNC codes
        state1,                 -- no errors, in sync
        state2,                 -- 1 error, in sync
        state3,                 -- 2 errors, in sync
        state4);                -- 3 errors, in sync

-- declare state machine encoding, state variable, and initial state
SIGNAL s_state : sync_state := state0;
```

Appendix F. Byte Sync Controller (continued)

```
BEGIN
proc1: PROCESS BEGIN
  WAIT UNTIL (clk='1');
  IF (reset='1') THEN
    s_state <= state0;      -- don't even look yet
  ELSE
    CASE s_state IS
      WHEN state0 =>
        IF ((cnt="1111") AND (error='0')) THEN
          s_state <= state1;
        ELSE
          s_state <= state0;
        END IF;
      WHEN state1 =>
        IF (error='1') THEN
          s_state <= state2;
        ELSE
          s_state <= state1;
        END IF;
      WHEN state2 =>
        IF (error='1') THEN
          s_state <= state3;
        ELSIF (cnt="1111") THEN
          s_state <= state1;
        ELSE
          s_state <= state2;
        END IF;
      WHEN state3 =>
        IF (error='1') THEN
          s_state <= state4;
        ELSIF (cnt="1111") THEN
          s_state <= state2;
        ELSE
          s_state <= state3;
        END IF;
      WHEN state4 =>
        IF (error='1') THEN
          s_state <= state0;
        ELSIF  (cnt="1111") THEN
          s_state <= state3;
        ELSE
          s_state <= state4;
        END IF;
      WHEN others =>
        s_state <= state0;
    END CASE;
  END IF;
END PROCESS proc1;
```

Appendix F. Byte Sync Controller (continued)

```
-- build 4-bit counter with enable and reset
ctr_en <=  '1' WHEN ((s_state=state0 AND reset='0' AND sync='1')
        OR (s_state=state2)
        OR (s_state=state3)
        OR (s_state=state4))
        ELSE '0';

ctr_reset <= '1' WHEN ((reset='1') OR (error='1')) ELSE '0';

-- add standard counter module
ctr1: cntr4 PORT MAP (                  -- contains the 4 bits of ctr1
  one,                              -- set carry in always active
  open,                             -- carry out unused
  ctr_en,                           -- counter enable
  zero,                             -- never load this counter
  zero, zero, zero, zero,           -- load inputs are not used
  clk,                              -- counter clock
  ctr_reset,                        -- will need to expand this signal
  cnt(3), cnt(2),                   --  counter holding register inputs
  cnt(1), cnt(0)
  );

-- assign output
bbsync <= '0' WHEN (s_state=state0) ELSE '1';
d1: DFF PORT MAP (bbsync, clk, bsync);

END arch1;
```

Appendix G. I/O Support

```
-- IOPLUS.VHD

-- Create enhanced I/O buffer that is not part of the io.vhd
-- package for the pASIC 380 family

PACKAGE iopluspkg IS
  COMPONENT HDI2PAD PORT (
    p0 : IN BIT;
    p1 : IN BIT;
    qn : OUT BIT);
  END COMPONENT;
END iopluspkg;

USE work.cypress.all;
USE work.rtlpkg.all;
USE work.iopkg.all;
USE work.resolutionpkg.all;

ENTITY HDI2PAD IS PORT (
  p0 : IN BIT;
  p1 : IN BIT;
  qn : OUT BIT);
END HDI2PAD;

ARCHITECTURE archHDI2PAD OF HDI2PAD IS

SIGNAL o : multi_buffer BIT;

BEGIN

u0: PAINCELL PORT MAP ( ip => p0, ini => o, iz => OPEN);
u1: PAINCELL PORT MAP ( ip => p1, ini => o, iz => OPEN);
qn <= o;

END archHDI2PAD;
```