Use HOTLink™ For 9- And 10-Bit Data

Introduction

Long-distance data-communication that once evolved from slow-serial to fast-parallel, is now changing back to high-performance serial data links. As system speeds increase, the inherent skew between several parallel lines (and their clock), and crosstalk between adjacent lines on parallel data cables (bundled or ribbon-cable style) become increasingly problematic. HOTLink makes it easy and economical to move this same data over a high-speed serial connection using fiber-optic or copper cables.

For data that is represented by standard 8-bit bytes (characters), HOTLink offers a built-in 8B/10B encoder and PLL-based clock and data recovery circuits. Data that is represented as 9- or 10-bit characters may also be transmitted with HOTLink, but the data must be handled differently. This application note explores the issues involved in moving such non-standard data over a serial connection using the built-in features of HOTLink and simple PLD support chips. An example system is developed and, for purposes of illustration, an almost Protocol-Free data communication channel is described. A board was built and tested to validate the concepts documented in this application note. Schematics and VHDL code are included in this document.

Serial Communications Links

The goal of any data link is to reliably move data from one location to another. When using a parallel bus, it is necessary to insure that the transmitted data, on all the parallel signal lines, arrives at the destination with sufficient amplitude to be detected by each receiver, and with sufficient timing margin to be captured correctly by a common clock or strobe. Skew and loss effects generally limit parallel busses to distances of a few meters to tens of meters.

Serial interfaces are often used to extend the high performance of a parallel interface well beyond the distances possible in a parallel link. By only sending a single signal (though sending each bit much faster) both the signal-to-signal and signal-to-clock skew limits are removed. While this allows communications over much greater distances, serial interfaces still have their own limitations.

8-Bit Data Path Applications

HOTLink (CY7B923/CY7B933) is designed primarily for moving streams of 8-bit parallel bytes from one location to another. This is normally accomplished by encoding these 8-bit data characters into 10-bit transmission characters and sending these 10-bit characters serially across a link. At the receiving end of this link these serial bits are grouped together 10 bits at a time (framed), and decoded back into their equivalent 8-bit data characters. Making a serial link to support an 8-bit data path is simplified because the circuitry necessary to perform the encoding, serializing, deserializing, framing, and decoding are all self contained in the HOTLink transmitter (CY7B923) and receiver (CY7B933).

8B/10B Encoding

HOTLink was designed primarily for operation with a transmission code known as 8B/10B. This code maps all possible 8-bit data values (and a few non-data command codes) into 10-bit transmission characters that are optimized for serial transmission and reception. The specific mappings of the of 8-bit data characters to 10-bit transmission characters are listed in the CY7B923/933 datasheet.

The 8B/10B code is DC-balanced to limit baseline wander, has a high guaranteed transition density to control the spectral content of the source signal, and simplifies the clock and data recovery circuitry. Additional information on the 8B/10B code and its operation on transmission lines can be found in the “HOTLink Design Considerations” application note in the *HOTLink User’s Guide*.

The 10-bit character size of the 8B/10B code allows it to represent up to 1024 different transmission characters. Out of these possible characters, only a subset are valid. These valid characters consist of those transmission characters that map to the 256 valid data characters and twelve valid special codes. This relationship is shown in the Venn diagram in *Figure 1a*.

8B/10B Decoding

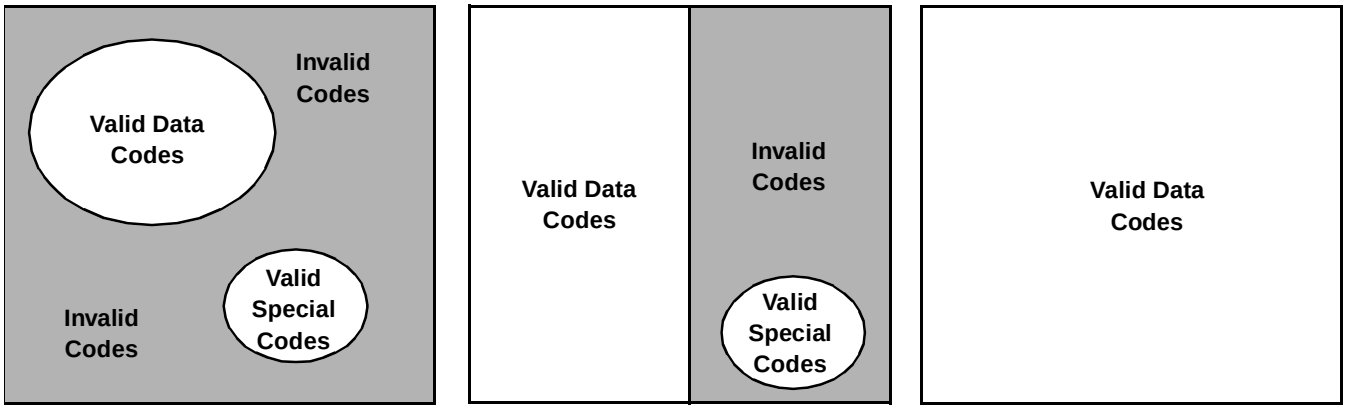
The HOTLink receiver at the end of the link uses a PLL (Phase Locked Loop) to synchronize its internal bit-clock to the serial data-stream. It uses the timing information contained in the received transitions to effectively “extract” a clock from the received data. While this allows individual bits to be detected, it does not identify character boundaries. This requires an additional mechanism, referred to as framing, to identify where in the serial data stream that bytes begin and end.

The framing operation is handled by HOTLink through use of an 8B/10B special-code character called a K28.5 or Sync character. This character contains a bit pattern that does not exist in (or across) any valid 8B/10B data character or characters. By sending this Sync code, and searching for it in the serial data stream, the receiver can determine the starting bit-position of the Sync character in the data stream. Once the beginning of one character is found, the start of all others is determined by counting off groups of ten bits.

Because of its numerous advantages for serial data transmission, the 8B/10B code has been selected for use in data transport for numerous standard communications interfaces, including Fibre Channel, ESCON (Enterprise System CONnection), SSA (Serial Storage Architecture), DVB-ASI (Digital Video Broadcast), and some implementations of FDDI (Fibre Distributed Data Interface) and ATM (Asynchronous Transfer Mode).

9-Bit Data Path Applications

HOTLink can also be used for 9-bit data path applications. These often exist in environments where specialized video or



a) 8B/10B Total Code Space b) 9B-Invert Total Code Space c) Scrambler Total Code Space

Figure 1. Data Space Relationships of 8B/10B, 9B-Invert, and Scrambled Codes

data-plus-parity information must be transmitted. Transmission of 9-bit data requires that the HOTLink transmitter and receiver operate in Bypass mode. In this mode the transmitter and receiver operate on a raw (unencoded) 10-bit interface. This means that the 10-bit data character present at the transmitter inputs at the rising edge of the system clock (CKW) is serialized and transmitted without modification.

In Bypass mode the HOTLink transmitter and receiver only operate with 10-bit characters. The simplest way to transmit 9-bit characters in bypass mode is to clock in a 9-bit parallel character on the transmit interface that contains an additional "fill" bit. While this fill-bit is in effect overhead information, it is necessary for a number of reasons.

Since the HOTLink parallel transmit interface (in Bypass mode) is ten bits wide, even if only nine bits are desired, all ten are captured on every CKW clock. The single bit of overhead in each character means that the transmitted bit-rate must be slightly over 11% faster than the equivalent bit-rate necessary for sending only nine bits of data.

9B-Invert Encoder

This overhead bit also improves the functionality of the interface. It allows creation of a simple encoding to guarantee at least one transition per transmitted character. This is accomplished by inverting one of the data bits in each character, as shown in *Figure 2*. If sufficient timing margin is not present between the transmit data-path pipeline stages for a separate inverter delay on one of the bits, this same function can be implemented using a 10-bit registered PLD (like a Cypress PLD20G10) as a pipeline register.

The Da/Db bits were selected in this example for the inverted-bit function. During transmission of normal data, these bits will always be complementary. This allows the standard 8B/10B framing character (K28.5) to be used for framing data on this interface.

The ten bit sequence for the K28.5 framing character exists in two forms: 001111010 or 110000101. These characters are automatically generated by the HOTLink transmitter when the $\overline{\text{ENA}}$ pin is pulled HIGH ($\overline{\text{ENN}}$ is held HIGH). The HOTLink receiver will correctly frame on either or both forms of this control character. In its correctly framed position in the data stream, the Da and Db bits for this character are always the same. This allows the K28.5 to be both transmitted and received as a valid special code that exists outside the normal 9-bit character data space. This relationship for the 9B-Invert encoder is shown in *Figure 1b*.

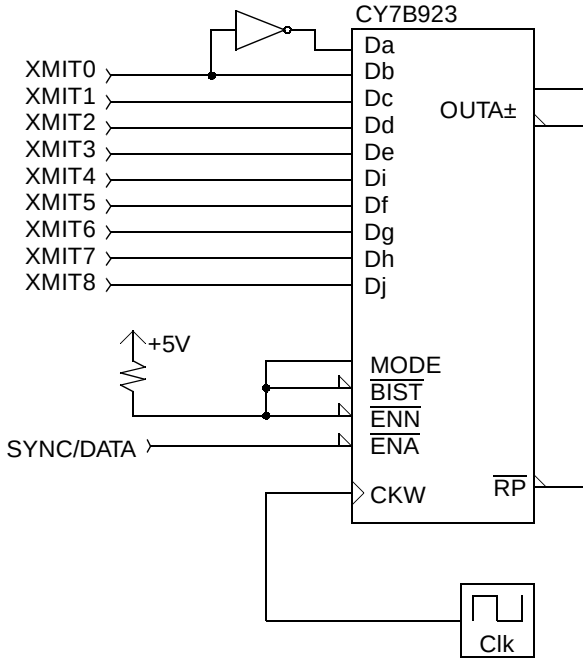


Figure 2. 9B-Invert Transmitter

9B-Invert Decoder

The 9-bit receiver uses the same clock extraction and framing circuitry as that used for an 8-bit receiver. The receiver still uses the received edges to adjust and position its internal bit-clock to the data stream, and framing is still performed using the K28.5 special character. However, since this interface is decoding 9-bit characters instead of 8-bit, there are also significant differences.

Now the receiver is configured to operate in Bypass mode. This allows the received raw 10-bit transmission character to appear at the receiver outputs. By using the same bit assignments as those used on the transmit end, a simple decoder can be built that detects valid, invalid, and framing (K28.5)

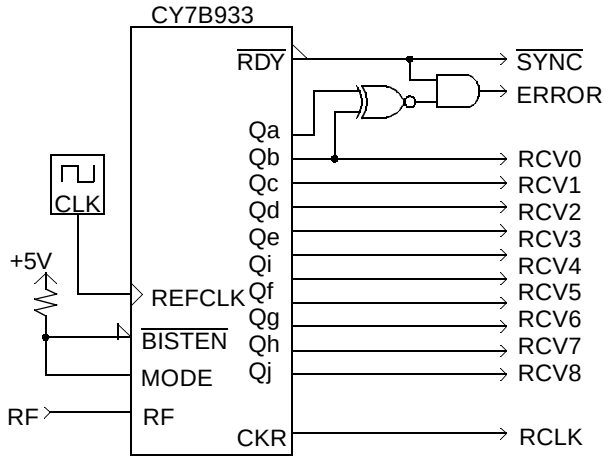


Figure 3. 9B-Invert Receiver

characters. This 9B-Invert receiver/decoder circuit is shown in *Figure 3.*

When a K28.5 Sync character is detected, the $\overline{\text{SYNC}}$ signal will be active LOW during the RCLK period that the Sync character is on the output data bus. *Figure 1* shows that almost half of the available 10-bit characters are invalid; i.e., bits Qa and Qb are not different. These invalid characters are detected using an XNOR gate to monitor the Qa and Qb signal lines. This error signal is gated with $\overline{\text{SYNC}}$ to prevent an error indication when either of the K28.5 patterns are detected.

This receiver uses the internal K28.5 detector to control framing of the serial data stream. For this to work, it is necessary to have these Sync characters present from time-to-time in the serial data stream. Because the Sync code exists outside the normal data space, it is possible to use this character as part of a higher-level protocol. This also allows the Sync code to be used as a fill character or boundary marker for packetized data.

The presence of the ERROR signal allows a small supervisor state machine to be built to controls the HOTLink receiver reframe (RF) input. This state machine checks the received data for erroneous data codes and, if too many are detected in a specific period of time, enables RF to reframe the data stream. This type of supervisor machine is documented for use on Fibre Channel and ESCON interfaces. A sample design for a state machine of this type can be found in the Cypress application note titled “Drive ESCON With HOTLink.”

9B-Invert Interface Limitations

The 9B-Invert interface does have its limitations. In the form documented here, only one non-data code (Sync) is available, compared to the twelve available command codes in the 8B/10B encoded interface.

Framing is also a bit more difficult. Unlike the 8B/10B code where the K28.5 cannot exist across character boundaries, it can in the 9B-invert coded interface. These are referred to as aliased Sync codes. An example of this is shown in *Figure 4.* Here the three 9B-Invert coded characters for x’09F’, x’160’, and x’0EC’ are shown to contain two aliased Sync codes. If framing were enabled at all times (even using the HOTLink multi-byte framer), patterns of this type would cause the receiver to frame to an incorrect character boundary. Following

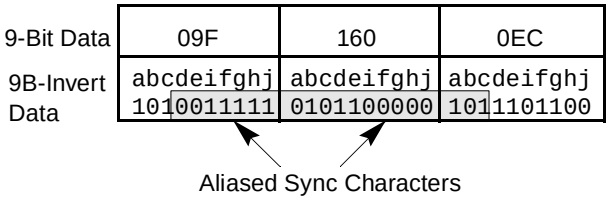


Figure 4. 9B-Invert Aliased Sync Characters

such a misframe, the receiver will decode a corrupted data stream until proper framing can be achieved.

The 9B-Invert data stream can also have a significant impact on propagation across the link. Worst case data characters of x’000’ and x’1FF’ contain only a single transition per 10-bit field. This is half that available for the corresponding worst case 8B/10B patterns.

Even more difficult to deal with is the lack of DC-balance in the 9B-Invert data stream. Repeating character streams of x’1FF’ (0111111111 when encoded) contain a 90% imbalance. This imbalance will force a considerable amount of baseline wander and significantly limit operating link lengths and data rates.

10-Bit Data Path Applications

In addition to 8- and 9-bit applications, HOTLink can also be used to transport 10-bit data. Like 9-bit applications, this requires that the HOTLink transmitter and receiver operate in Bypass mode.

This at first appears to be quite simple. Now the data can be directly serialized and transmitted without any overhead information. This means that the transmitted bit-rate is the same as the source data rate. But this brings with it additional problems. The receiver must now contend with serial data that may have few (if any) transitions to allow it to properly track the data. Transmitting as few as twenty bytes of zeros or ones requires the receiver to maintain exact bit timing, without any timing reference information, for 200 bit-intervals. While HOTLink has been verified to operate correctly under even this extreme interval, a truly random serial stream could easily go much longer than this without transitions.

Scrambling

To attempt to circumvent some of these problems, scrambler codes are often used to attempt to force greater numbers of transitions into the data stream. These codes operate by selectively inverting bits in the serial data stream, as directed by a randomizer or scrambler polynomial. This allows transitions to be created without adding any redundant information; i.e., extra bits or special signalling.

Scrambling is also used by many industry standard serial interfaces such as SMPTE (Society of Motion Picture and Television Engineers), ATM, and SONET (Synchronous Optical NETwork). Each of these interfaces uses a specific randomizer polynomial with characteristics selected for the type of data to be transmitted.

Descrambling

Descrambling a serial data stream performs the opposite function of scrambling. This is done by combining the received data stream with a complementary polynomial that removes the randomization effects added to the data at the

transmitter. The result is a serial data stream equivalent to that transmitted.

A problem still remains at the receive end of the link. The end goal is a recovered set of parallel words, not just a retimed serial data stream. This means that a framing operation is again required to locate the start and end bit-positions of each data character.

Figure 1c shows that all possible bit combinations of a serialized scrambled interface are allocated for valid data characters. Unlike an 8B/10B encoded interface, which can transmit and decode framing specific characters that exist outside the set of valid data characters, scrambled interfaces normally use combinations of characters from the normal set of valid data characters to also perform framing. Another method used is to restrict the range of valid characters to a subset of the full character set. While this does prohibit the transmission of random data, it does allow the interface to create special command and framing characters.

Scrambled Interface Limitations

Scrambled interfaces also cause degraded signalling on the media. Scrambled codes, sending random data, have an unbounded run-length. Specific data codes, when mixed with the scrambler polynomial, can cause either continuous strings of zeros or continuous strings of ones to be transmitted. This is effectively a DC signal, generating the maximum possible baseline wander and DDJ in a link. While the continuous-ones problem can be overcome by adding an NRZI (non-return-to-zero, invert-on-ones) encoder after the scrambler, this still does not solve the problem of continuous zeros.

Error detection is also quite limited in a scrambled interface. Since all 10-bit fields are valid characters, a single-bit (or multi-bit) corruption in the serial data stream translates into another valid character. This requires all error detection (when needed) to occur at a higher level in the interface or protocol. For many interfaces, such as those used for real-time video, error detection may not be an issue at all. The corruption of a small number of pixels that are overwritten every few milliseconds is easily tolerated.

The last limitation is that a scrambled interface uses the entire code space for data characters, as shown in *Figure 1c*. This means that no codes, other than valid data characters, are (normally) available for framing or signalling.

Scrambler Codes

Scrambler codes are often used when the primary transmission goal is to limit the maximum bit-rate on the physical media, or to transmit fixed format data over hardware having specific transmission characteristics.

PRBS Generators

Some types of scrambling can be done using forms of look-up tables. In most cases a Pseudo Random Binary Sequence (PRBS) is combined with the source data. A PRBS is most easily created from a tapped shift register with feedback, like that shown in *Figure 5*. This type of shift register is also known as a Linear Feedback Shift Register (LFSR). When started from an all 1s condition, this PRBS generator will produce the sequence shown in *Table 1.*

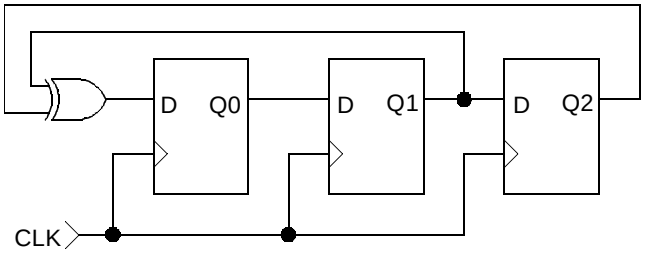


Figure 5. RBS Generator

Table 1. PRBS Sequence

State	Q0	Q1	Q2
0	1	1	1
1	0	1	1
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	1
6	1	1	0
7	1	1	1

Notice in *Table 1* that states 0 and 7 contain the exact same values. This means that the PRBS repeats at this point, and has a length of seven. The one state missing from the table is 000. For a LFSR to operate, the all-zeros condition must never be entered. In this zero-state, no 1-bits exist to create other 1-bits. To prevent this condition, LFSR pattern generators usually contain extra logic to check for the all-zero state, and force one or more bits active if it is detected.

The number of active states for a LFSR is both significant and predictable. For this 3-bit register the LFSR has seven active states. A four bit register would have fifteen active states. Since all states are valid except one (all zeros), an N-bit LFSR would have $2^N - 1$ states.

Scrambler/Descrambler

To actually scramble some source data, a PRBS is combined with a source data stream as shown in *Figure 6*. The data here is added to a PRBS generator polynomial using modulo-2 arithmetic. This addition is performed without carry-out or carry-in, and reduces in hardware to combinations of XOR gates.

The usage of modulo-2 arithmetic allows the source data to be recovered (descrambled) by reversing the scrambler function. This works because any number XORed with itself is zero. This effectively subtracts out the PRBS modifications and leaves the source serial data as a remainder.

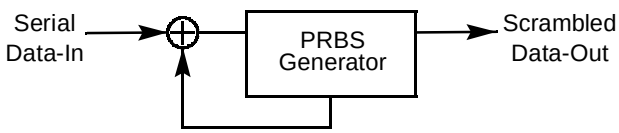


Figure 6. Scrambler Block Diagram

Scrambler Polynomials

Many different scrambler polynomials are in common use. Each polynomial is selected to operate with specific source data types, and to generate specific scrambled data characteristics. The PRBS code listed in *Table 1* shows a total of four 1s and three 0s for any of the bits in the shift register. This 4:3 ratio contains a significant DC content. By lengthening the register, the degree of randomness is increased while the DC-content is decreased.

Three common scrambler polynomials are listed in Equations 1, 2, and 3. The polynomial in Equation 1 is commonly used for scrambling digital video information, and has been standardized for this function by ANSI (American National Standards Institute) and SMPTE (Society of Motion Picture and Television Engineers). The polynomial in Equation 2 is used for ATM distribution of data and audio information, while that in Equation 3 is used with SONET for similar data/audio transmission

$$f(x) = x^9 + x^4 + 1 \quad \text{Eq. 1}$$

$$f(x) = x^{43} + 1 \quad \text{Eq. 2}$$

$$f(x) = x^7 + x^6 + 1 \quad \text{Eq. 3}$$

Figure 7 shows Equation 1 translated to a serial hardware form. Now the additions are shown as XOR gates. Remember that with binary arithmetic, a single bit shift is equivalent to a multiply. The input to the function is the source serial data stream. The input to the first bit of the shifter consists of three terms added together; the input data, the data after five shifts, and the data after nine shifts.

To handle a “live” video data stream this scrambler would have to be clocked at 270 MHz, and be fed a serial data stream at that same rate. Few logic families are available that allow operation at this high clock-rate. However, the scrambler function in *Figure 7* can be remapped from a serial to a parallel form. This reduces the clock rate by a factor of ten, and allows the circuit to operate with the native 10-bit parallel format of the source data.

Parallel Scramblers

Mapping a serial function to a parallel equivalent requires adding gates to perform a look-ahead calculation. This logic is used to determine the contents of the scrambler registers after each clock cycle. By adding logic that looks ten clocks ahead, it is possible to clock the shift register once with a 10-bit parallel data word as input, and perform a scrambling function equivalent to ten clocks with serial data input.

The 9-bit size of the original scrambler makes this a bit more difficult to visualize. To map this to handle 10-bit parallel data first requires adding a stage to the register to allow it to match the input data width. In the serial form, this additional bit could be added to either end of the shift register without effecting the scrambling results. For a parallel implementation this bit needs to be added to the output end of the shift register to avoid bypassing the scrambler function. This would add an R0 flip-flop to the output end of the structure shown in *Figure 7*.

In its default state the register bits R1 through R0 of the scrambler will contain data of varying value, based on the previous data that was clocked into the register. To track this present-state data in the following example, current register content names of X1 through X0 are assigned to these registers respectively.

Prior to clocking a new data bit into the scrambler register, the contents of the register are

```
R1 = X1
R2 = X2
R3 = X3
R4 = X4
R5 = X5
R6 = X6
R7 = X7
R8 = X8
R9 = X9
R0 = X0
```

Data is input to the scrambler LSB first. If the input data is numbered off as D0, D1, D2, etc., the register contents after the first bit is clocked in would be (⊕ is equivalent to XOR)

```
R1 = D0 ⊕ X5 ⊕ X9
R2 = X1
R3 = X2
R4 = X3
R5 = X4
R6 = X5
R7 = X6
R8 = X7
R9 = X8
R0 = X9
```

This shows the data being shifted down one register location, with the first flip-flop containing the input data (D0) XORed with two feedback terms. Following a second clock the registers would contain

```
R1 = D1 ⊕ X4 ⊕ X8
R2 = D0 ⊕ X5 ⊕ X9
R3 = X1
R4 = X2
R5 = X3
R6 = X4
R7 = X5
R8 = X6
R9 = X7
R0 = X8
```

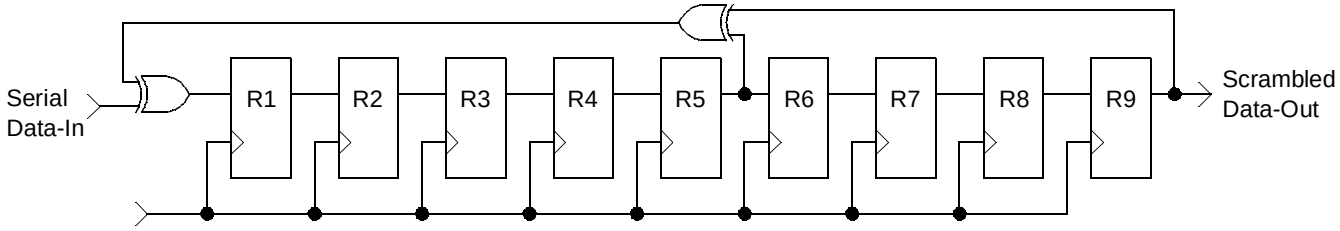


Figure 7. Serial Scrambler For $X^9 + X^4 + 1$

Now the R1 flip-flop contains the second input data bit (D1) XORed with the two appropriate feedback terms. If the ten register bits were actually connected as these equations list, and presented with two data bits (D0 and D1), both bits would be encoded in a single clock rather than requiring separate clock cycles for each bit.

After clocking the original LFSR six times (with the appropriate data), the registers would contain

```
R1 = D5 ⊕ (D0 ⊕ X5 ⊕ X9) ⊕ X4
R2 = D4 ⊕ X1 ⊕ X5
R3 = D3 ⊕ X2 ⊕ X6
R4 = D2 ⊕ X3 ⊕ X7
R5 = D1 ⊕ X4 ⊕ X8
R6 = D0 ⊕ X5 ⊕ X9
R7 = X1
R8 = X2
R9 = X3
R0 = X4
```

Note here that the current equation for flip-flop R1 now contains terms that have already passed through the lower bits of the shifter once. When implemented with these equations the LFSR would scramble six bits on every clock cycle to the t register.

Extending this to a full 10-bit clocking would yield equations of

```
R1 = D9 ⊕ (D4 ⊕ X1) ⊕ (D0 ⊕ X9)
R2 = D8 ⊕ (D3 ⊕ X2 ⊕ X6) ⊕ X1
R3 = D7 ⊕ (D2 ⊕ X3 ⊕ X7) ⊕ X2
R4 = D6 ⊕ (D1 ⊕ X4 ⊕ X8) ⊕ X3
R5 = D5 ⊕ (D0 ⊕ X5 ⊕ X9) ⊕ X4
R6 = D4 ⊕ X1 ⊕ X5
R7 = D3 ⊕ X2 ⊕ X6
R8 = D2 ⊕ X3 ⊕ X7
R9 = D1 ⊕ X4 ⊕ X8
R0 = D0 ⊕ X5 ⊕ X9
```

The actual equation for R1 contains two instances of X5 XORed together. Because this always reduces to a zero, these terms were removed from the equation.

When implemented in this form, a full ten input bits are scrambled on each clock cycle of the scrambler register. While this allows the scrambler to be clocked at a much slower rate, it also requires many more XOR gates to implement. Unlike the serial LFSR that only required two XOR gates to implement the scrambler polynomial, it now takes thirty to implement the same polynomial in parallel. As a general rule of thumb for these types of serial to parallel conversions, each time the number of bits encoded in a given clock cycle is doubled, it doubles the number of XOR gates needed in the design.

Parallel Descrambler

The schematic for the serial descrambler is shown in *Figure 8*. Unlike the scrambler, which was based entirely on feedback construction, the descrambler is instead a feed-forward design. Converting this to a parallel implementation requires similar look-ahead logic as that used in the scrambler.

Because the data size of this descrambler will also be 10 bits wide, an additional R0 register is added to the serial data output end of the descrambler. Prior to clocking a new data bit into the descrambler register, the contents of the register are

```
R1 = X1
R2 = X2
R3 = X3
R4 = X4
R5 = X5
R6 = X6
R7 = X7
R8 = X8
R9 = X9
R0 = X0
```

Data is input to the descrambler LSB first. Just as in the ca example, this input data is numbered off as D0, D1, D2, etc. Following the clocking in of the first serial data bit the register contents would be

```
R1 = D0
R2 = X1
R3 = X2
R4 = X3
R5 = X4
R6 = X5
R7 = X6
R8 = X7
R9 = X8
R0 = X9 ⊕ D0 ⊕ X5
```

If a second bit were clocked into the descrambler register at this time, the first descrambled bit (located at this time in R0) would get shifted out and lost. To keep the descrambled bits until they are actually output in parallel form requires extending the descrambler register. Since the end system will operate with 10-bit quantities of data, this actually requires a second 10-bit register. Clocking a second bit into the now 20-bit long descrambler register would yield register contents of

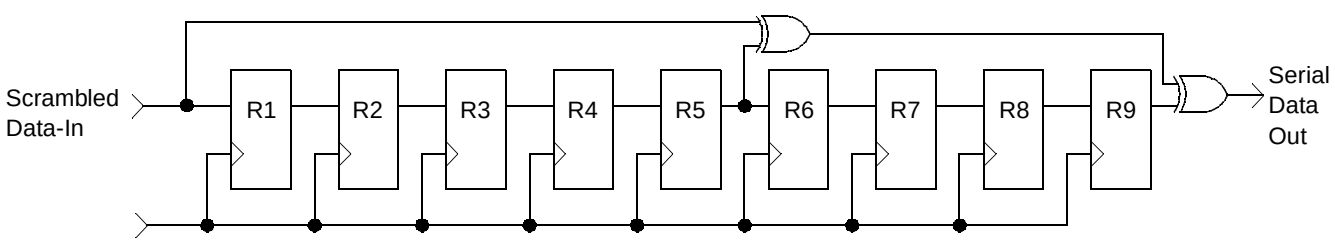


Figure 8. Serial Descrambler For $X^9 + X^4 + 1$

```
R1 = D1
R2 = D0
R3 = X1
R4 = X2
R5 = X3
R6 = X4
R7 = X5
R8 = X6
R9 = X7
R0 = X8 ⊕ D1 ⊕ X4
r1 = X9 ⊕ D0 ⊕ X5
r2 = X0
r3 = x1
r4 = x2
r5 = x3
r6 = x4
r7 = x5
r8 = x6
r9 = x7
r0 = x8
```

Following similar shifting, the descrambler register would contain the following values after clocking in a full ten bits.

```
R1 = D9
R2 = D8
R3 = D7
R4 = D6
R5 = D5
R6 = D4
R7 = D3
R8 = D2
R9 = D1
R0 = D0 ⊕ D9 ⊕ D4
r1 = X1 ⊕ D8 ⊕ D3
r2 = X2 ⊕ D7 ⊕ D2
r3 = X3 ⊕ D6 ⊕ D1
r4 = X4 ⊕ D5 ⊕ D0
r5 = X5 ⊕ D4 ⊕ X1
r6 = X6 ⊕ D3 ⊕ X2
r7 = X7 ⊕ D2 ⊕ X3
r8 = X8 ⊕ D1 ⊕ X4
r9 = X9 ⊕ D0 ⊕ X5
r0 = X0
```

When implemented in this form a full ten bits are descrambled in a single clock cycle. To output a full 10-bit character requires assigning the proper bits of the 20-bit descrambler register as outputs. For this specific descrambler, this requires assigning register bits r9 through r1 to data output bits 0 through 8, and descrambler bit R0 to output bit 9.

10-Bit Scrambler Design Example

To validate these concepts, a parallel scrambler and descrambler (such as that described previously), was implemented in a pair of small PLDs. When used along with HOTLink, this design can move 10-bit characters across a serial link. A block diagram of this design is shown in *Figure 9*. It consists of four major logic blocks: the scrambler PLD, the CY7B923 HOTLink transmitter, the CY7B933 HOTLink receiver, and the descrambler PLD. A complete schematic of the interconnect of all the scrambler/descrambler components is shown in Appendix A.

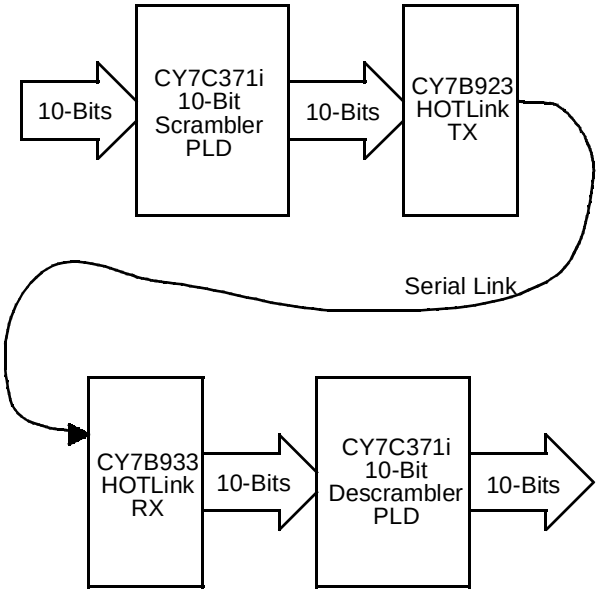


Figure 9. Scrambled Interface Block Diagram

Scrambler PLD

The scrambler function was implemented in a Cypress CY7C371 Flash programmable PLD. A block diagram of the internal functions of this PLD is shown in *Figure 10*. Raw 10-bit data is first captured into an input holding register on each rising clock edge. The clock used here is also connected to the HOTLink transmitter CKW clock. This data then passes through an XOR feedback array where the parallel scrambler logic equations are implemented. The data is finally latched into an output pipeline register that contains the scrambled version of the input data.

The PLD also contains a separate supervisor state machine to control the generation of framing characters for use by the receiving end of the link. A scrambler bypass function is also provided that allows raw (non-scrambled) source data to be moved between the transmitter and the receiver PLDs. The

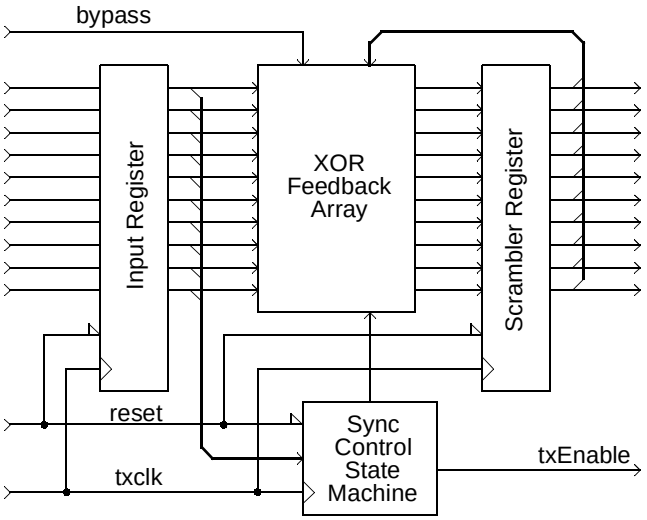


Figure 10. Scrambler PLD Internal Structure

complete VHDL source-code listing of this scrambler PLD can be found in Appendix B.

Descrambler PLD

The descrambler function was also implemented in a CY7C371i Flash programmable PLD. A block diagram of the internal functions of this PLD is shown in *Figure 11*. It performs the reverse function of the scrambler PLD. It accepts a 10-bit scrambled data character into an input holding register on every rising edge of the HOTLink receiver CKR recovered-data clock. The scrambled data in the holding register is unscrambled using the XOR feed-forward array between the holding register and the two descrambler registers. A descrambler bypass function is provided to allow raw data from a bypassed transmitter to pass through the descrambler PLD.

The descrambler PLD also contains a framing-control state machine. This state machine operates in conjunction with the sync-control state machine in the scrambler PLD to allow the receiver to properly frame to the data stream.

One additional function (not shown in *Figure 11*) is also contained in the descrambler PLD. This function allows testing of the serial link itself (without any scrambler/descrambler functions). It uses the Built-In Self-Test (BIST) capability of the HOTLink transmitter and receiver to generate and validate a pseudo-random data stream. This capability (when enabled) is monitored in the descrambler PLD where any detected errors are latched and displayed. The complete VHDL source-code listing for the descrambler PLD can be found in Appendix C.

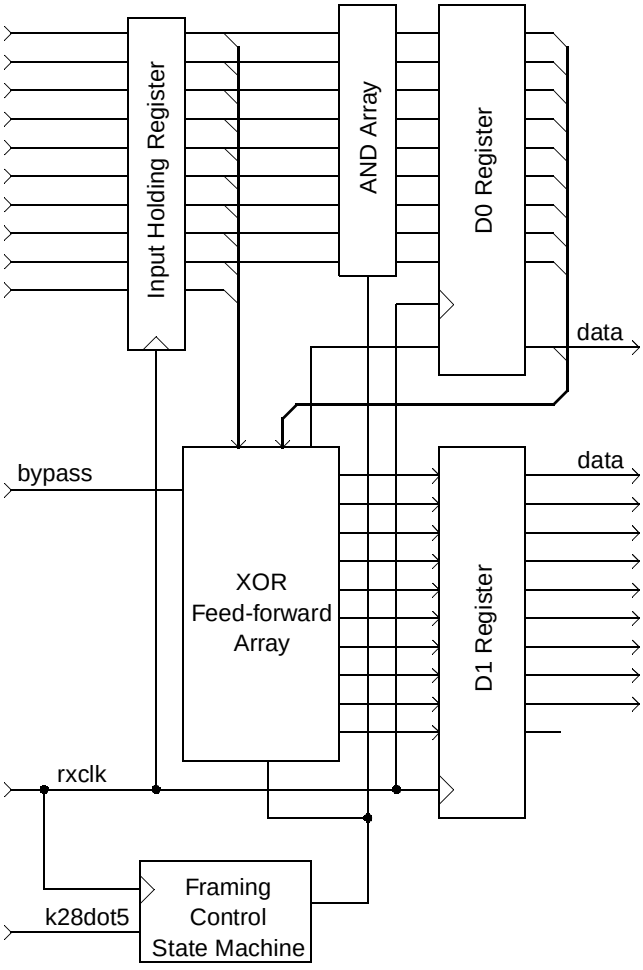


Figure 11. Descrambler PLD Internal Structure

Framing

In a normal scrambled data interface, the framing operation uses special sequences of normal data characters to determine where character boundaries fall. This is necessary because non-data characters are no longer available in a scrambled data interface; i.e., all possible bit combinations decode into valid data characters. These framing sequences are often composed of characters that are not used as part of the normal data transfer, such as multi-character sequences of all zeros or all ones.

In a standard scrambled interface design, the actual detection of where framing occurs is performed after the data has been descrambled. For this design example the framing operation is performed directly on the scrambled data stream by making use of a feature built into the HOTLink receiver.

Multi-Byte Framer

The HOTLink receiver is designed to automatically (when framing is enabled) frame to the first K28.5 code received in the serial data stream. If the RF input to the HOTLink receiver is kept active (HIGH) for more than approximately 2048 REFCLK clocks, the receiver enters into a special framing mode called Multi-Byte framing. In this mode the receiver will no longer frame to first K28.5 code received. Instead, it waits for a *pair* of K28.5 characters, separated by *exactly* 0, 1, 2, or three 10-bit fields.

Sync-Control State Machine

The Sync-Control state machine in the scrambler PLD monitors the source data-stream for sequences of x'000' characters. When a sequence (two or more) of these characters is detected, the scrambling function is disabled. The HOTLink transmitter is then allowed to automatically replace these groups of x'000' characters in the data stream with K28.5 sync codes. When the data input changes back to non-zero characters, the scrambler is re-enabled and scrambled characters are again transmitted.

To time the substitutions correctly, the state machine needs a one-state look-ahead in the data stream for x'000' characters. This look-ahead is achieved by the presence of the input holding register in the scrambler PLD. For applications that do not require this type of framing action, the input holding register and Sync-Control state machine may be removed.

Framing Control State Machine

The HOTLink receiver is configured with RF hardwired HIGH. This keeps the receiver in a constant multi-byte framer mode. In this mode, when the strings of K28.5 characters (substituted for strings of non-scrambled x'000's) are received, the HOTLink receiver will frame to the correct character boundaries in the data stream.

To make sure that the data output of the descrambler is maintained correctly during these intervals, the framing control state machine detects strings of K28.5 characters (using the $\overline{\text{RDY}}$ signal from the HOTLink receiver), and substitutes x'000' characters at the descrambler output port.

The substitution of multiple Sync codes with x'000' characters also requires a look-ahead operation. This look-ahead is achieved by the presence of the input holding register in the descrambler PLD. For applications that perform framing at a different level, it is expected that this input register will not normally be necessary. However, an external framer (operating on byte or character parallel data) will generally consume significantly more logic than a single pipeline register

Scrambled Interface Concerns

This scrambler interface, as implemented using HOTLink and this pair of PLDs, does implement and operate with the $x^9 + x^4 + 1$ scrambler code. However, use of this design in its present form for actual data transmission may not be desirable.

NRZI Encoding

The scrambler algorithm used here is normally modified by the addition of NRZI (Non-Return to Zero, Invert on ones) encoding to both increase the transition density and limit the DC-content of the signal. This added layer of encoding has a secondary benefit of creating a polarity-free link. This means that the data (when used with a similar NRZI decoder at the receiver end) will decode correctly if the differential connections to the receiver are made correctly or transposed.

An example of a serial NRZI encoder is shown in *Figure 12*. With an NRZ serial data stream as a source input, one of two output data streams are generated based on the starting state of the encoder flip-flop.

This NRZI data stream is decoded to NRZ by routing the serial data (at the receiver end of the serial data link) through the same logic as used in the NRZI encoder, but configured for feed-forward instead of feed-back. By adding a second XOR to the data, the changes made to the data stream by the NRZI encoder are canceled out.

This NRZI encoding function is not implemented in the PLDs documented here, but can be done using slightly more complex programmable parts. Because the interface to the HOTLink transmitter and receiver are parallel in nature, parallel forms of the NRZI encoder and decoder would have to be implemented. An example of a parallel NRZI encoder is shown in the application note titled "Replace Your Am7968 TAXI transmitter With a CY7B923 HOTLink".

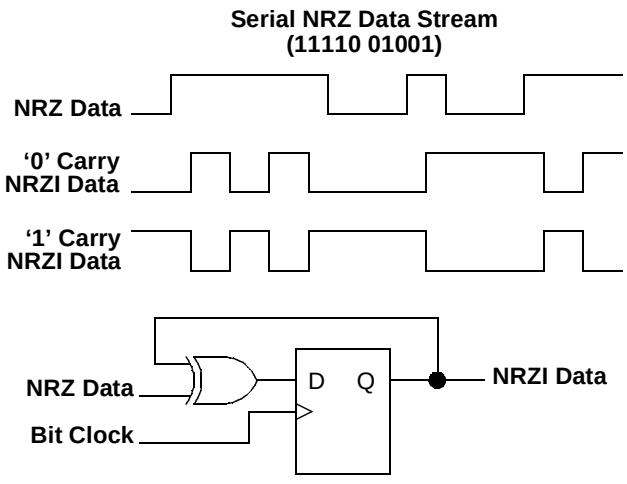


Figure 12. NRZI Encoder

Long Bit Strings

The usage of all possible bit-combinations also means that it is still possible to generate long strings of zeros and ones in the transmitted data stream. No matter what character is currently being transmitted, it is always possible to find a specific data character that will generate either an all zeros or all ones character.

Mis-Framed Data

This same code-space limitation in the data stream also effects the framing operation used in this example. While the HOTLink receiver will correctly frame to the strings of K28.5 characters generated by the HOTLink transmitter, it will also frame to other pairs of K28.5 characters generated by normal scrambling of the source data stream. These other K28.5 codes may at times exist across pairs of characters, similar to the aliased Sync characters shown in *Figure 4*.

When sets of such mis-aligned K28.5 codes are present (meeting the framing requirements of the multi-byte framer), the HOTLink receiver will reframe to a new character boundary in the data stream. Following such a mis-frame, all subsequent characters will be decoded incorrectly, until a new reframe operation occurs to reset the proper character boundaries. While the usage of the multi-byte framer greatly lessens the probability of such a mis-frame, it does not eliminate the possibility. This K28.5-based framer could be improved through use of an external state machine that only permits framing operations during those portions of the data stream where K28.5 or synchronization codes are explicitly expected.

For systems whose serial protocol does not permit the use of the K28.5 character for synchronization, it is possible to develop a framer that operates on unframed 10-bit parallel characters. This framing logic would normally be implemented following the descrambling of the data stream.

This logic is significantly more complex than using the integrated framer present in the HOTLink receiver. It would require logic capable of comparing for a specific framing character (or sequence of characters) simultaneously across all possible bit positions of a received data stream. Following detection of the framing sequence, additional logic would be necessary (most likely in the form of a barrel shifter) to shift the non-aligned 10-bit characters into their proper bit positions. This parallel-framer function is not implemented in the PLD documented here, but can be done in a more complex programmable part.

Conclusions

HOTLink is optimized for serial transport of 8-bit data. Through the use of external hardware, it is also possible to transport both 9- and 10-bit data.

The interfaces described in this application note operate at a very low level. As documented here they do not contain any higher level protocol information, other than that minimal portion necessary to control character framing of the data stream. For those applications where standardized protocols or packetization is necessary, these functions can be included in the data stream (through use of additional hardware or software) in addition to the present framing hardware.

The 10-bit scrambled interface design example listed here has been verified to transport 10-bit data streams. While based on the SMPTE scrambler polynomial, the design example is not compatible with the SMPTE video serial data stream in its present form. It may be made SMPTE compatible through the addition of NRZI encoding and external SMPTE framer logic, however, such design efforts are beyond the scope of this application note.

The full VHDL source code for the PLDs in this application note is listed in Appendices B and C. This source code is also available in electronic form from the Cypress BBS.

References

1. *ESCON I/O Interface*, IBM Corporation, 1990, 1991
2. *HOTLink User's Guide,* Cypress Semiconductor, 1999
3. *ANSI/SMPTE 259M,* Serial Digital Interface for 10-bit 4:2:2 Component and 4 fsc NTSC Composite Digital Signals, 1997
4. *Replace Your Am7968 TAXI Transmitter With a CY7B923 HOTLink*, Cypress Semiconductor, 1995

Appendix A. Scrambler/Descrambler Interconnect Schematic

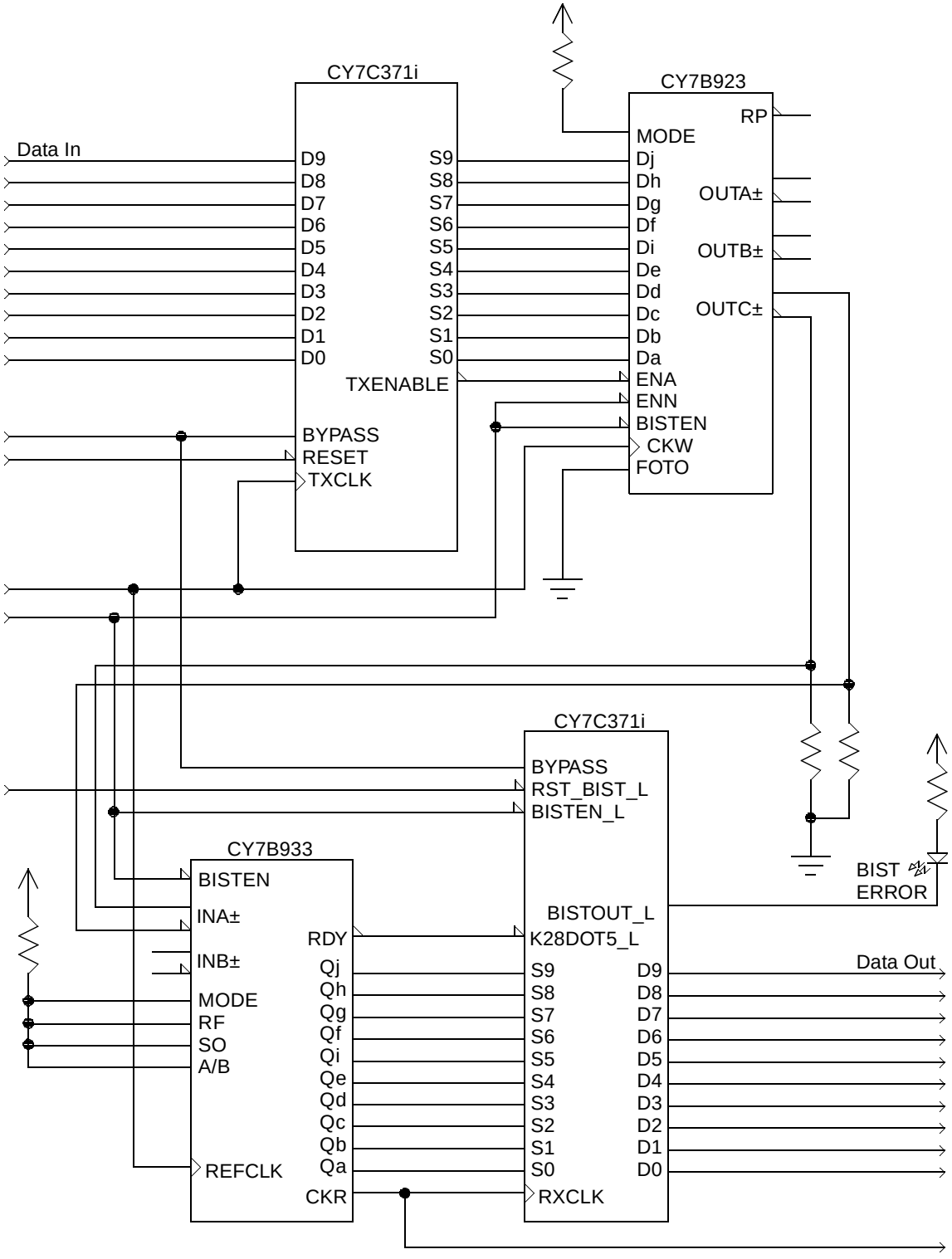


Figure 13. Schematic For HOTLink Based 10-Bit Scrambler/Descrambler Link

Appendix B. Scrambler VHDL Source Code

```
-- SCRAMTX.VHD
-- x9 + x4 + 1 scrambler
PACKAGE scramtx IS
    COMPONENT scram
    PORT (
        reset,              -- active low reset for state machines
        txclk,              -- HOTLink TX CKW clock
        bypass: IN BIT;     -- raw data mode - bypass scrambler
                            -- 10-bit raw data interface
        data: IN BIT_VECTOR(0 TO 9);
        txEnable: OUT BIT;  -- HOTLink TX /ENA
        scrData: OUT BIT_VECTOR(0 TO 9));
    END COMPONENT;
END scramtx;

ENTITY scram IS
    PORT (
        reset,              -- active low reset for state machines
        txclk,              -- HOTLink TX CKW clock
        bypass: IN BIT;     -- raw data mode - bypass scrambler
                            -- 10-bit raw data interface
        data: IN BIT_VECTOR(0 to 9);
        txEnable: OUT BIT;  -- HOTLink TX /ENA
                            -- 10-bit scrambled data for HOTLink TX
        scrData: OUT BIT_VECTOR(0 to 9));
END scram;

ARCHITECTURE archscram of scram IS

SIGNAL  ina,inb,inc: BIT;       -- intermediate XOR terms
SIGNAL  ind,ine,inf: BIT;       -- intermediate XOR terms
SIGNAL  inp        : BIT_VECTOR(0 to 9);    -- internal XOR terms
SIGNAL  dout       : BIT_VECTOR(0 to 9);    -- scrambler/output register
SIGNAL  holdRegA   : BIT_VECTOR(0 to 9);    -- input data register

-- declare synchronization controller states
TYPE    txFSMtype IS (
            s0,         -- wait state
            s1,         -- first x'00' detected
            s2,         -- second x'00' detected, send k28.5s
            s3          -- normal data again
            );
SIGNAL  txFSM: txFSMtype;
```

Appendix B. Scrambler VHDL Source Code (continued)

```
-- do not expand these XOR signals
ATTRIBUTE synthesis_off of ina: SIGNAL is true;
ATTRIBUTE synthesis_off of inb: SIGNAL is true;
ATTRIBUTE synthesis_off of inc: SIGNAL is true;
ATTRIBUTE synthesis_off of ind: SIGNAL is true;
ATTRIBUTE synthesis_off of ine: SIGNAL is true;

BEGIN
-------------------------------------------------------------------
-------------------------------------------------------------------
-- Declare the input holding register. This register accepts raw
-- 10-bit data on each rising edge of the CKW clock for scrambling.
-- IF RESET is active, the input register is forced to all zeros.
RegA: PROCESS (txclk, reset)
BEGIN
    IF (reset = ’0’) THEN
    -- if RESET is active, force the holding register to all zeros
        holdRegA <= ”0000000000”;
    ELSIF (txclk’event AND txclk = ’1’) THEN
    -- else capture input data word at rising edge of CKW
        holdRegA <= data;
    END IF;
END PROCESS RegA;

-------------------------------------------------------------------
-------------------------------------------------------------------
txFSMseq: PROCESS (txclk, reset)
-- reset/synchronization control state machine
-- This machine tracks the detection of an all zeros condition on
-- the input data bus. If the condition persists, then the x’000’
-- fields are changed to K28.5 SYNC codes by controlling the TX enable.
BEGIN
    IF (reset = ’0’) THEN
    -- If RESET is active, then force the state machine to the
    -- s0 or wait state
        txFSM <= s0;
    ELSIF (txclk’event and txclk = ’1’) THEN
        CASE txFSM IS
            WHEN s0 =>
            -- when in WAIT, check for any received x’00’ characters
                IF (data = ”0000000000”) THEN
                -- if x’00’ then go to s1
                    txFSM <= s1;
```

Appendix B. Scrambler VHDL Source Code (continued)

```
                ELSE
                -- if not x'00' then stay in the wait state
                    txFSM <= s0;
                END IF;
            WHEN s1 =>
            -- one x'00' detected, check for more
                IF (data = "0000000000") THEN
                -- if a second x'00' detected then send K28.5 sync codes
                    txFSM <= s2;
                ELSE
                -- if not return to wait state
                    txFSM <= s0;
                END IF;
            WHEN s2 =>
            -- remove TX enable and check for another K28.5 sync code
                IF (data = "0000000000") THEN
                -- if x'00' remains, keep sending sync codes
                    txFSM <= s2;
                ELSE
                -- if non-zero data then encode the data
                    txFSM <= s3;
                END IF;
            WHEN s3 =>
            -- send data and check for more x'00' characters
                IF (data = "0000000000") THEN
                    txFSM <= s1;
                ELSE
                    txFSM <= s0;
                END IF;
        END CASE;
    END IF;
END PROCESS txFSMseq;
-- enable the HOTLink TX if not encoding a pair of x'00' fields
txEnable <= '0' WHEN (txFSM = s0 OR txFSM = s1) ELSE '1';

-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- The following equations map the functionality of the scrambler.
-- The inp() assignments are the data inputs to the output/scrambler
-- register. This allows a full 10-bits of input data to be
-- scrambled in a single clock cycle.
```

Appendix B. Scrambler VHDL Source Code (continued)

```
-- These are intermediate XOR terms that are used in
-- multiple locations. They are broken out separately to
-- simplify the following equations
    ina    <= holdRegA(4)   XOR dout(1)     XOR dout(5);
    inb    <= holdRegA(0)   XOR dout(9)     XOR dout(5);
    inc    <= dout(1)       XOR dout(2)     XOR dout(6);
    ind    <= dout(2)       XOR dout(3)     XOR dout(7);
    ine    <= dout(3)       XOR dout(4)     XOR dout(8);
    inf    <= dout(4)       XOR dout(5)     XOR dout(9);

-- these assignments define the scrambler as implemented for
-- 10-bit parallel operation
    inp(1) <= holdRegA(9)   XOR ina         XOR inb;
    inp(2) <= holdRegA(8)   XOR holdRegA(3) XOR inc;
    inp(3) <= holdRegA(7)   XOR holdRegA(2) XOR ind;
    inp(4) <= holdRegA(6)   XOR holdRegA(1) XOR ine;
    inp(5) <= holdRegA(5)   XOR holdRegA(0) XOR inf;
    inp(6) <= holdRegA(4)   XOR dout(1)     XOR dout(5);
    inp(7) <= holdRegA(3)   XOR dout(2)     XOR dout(6);
    inp(8) <= holdRegA(2)   XOR dout(3)     XOR dout(7);
    inp(9) <= holdRegA(1)   XOR dout(4)     XOR dout(8);
    inp(0) <= holdRegA(0)   XOR dout(5)     XOR dout(9);

-------------------------------------------------------------------
-- This process defines the operation of the output register.
-- If RESET is active, then the output/scrambler register is forced
-- to a known state (all zeros). Otherwise, the output register
-- is either loaded with the new scrambled data, or with the data
-- in the input holding register, based on the state of the BYPASS
-- signal.
reg: PROCESS (txclk, reset)
BEGIN
    IF (reset = '0') THEN       -- if RESET is active then
        dout <= "0000000000";   -- force the output register to x'000'
    ELSIF (txclk'event AND txclk = '1') THEN
    -- otherwise check the state of the BYPASS signal
        IF (bypass = '1') THEN  -- if BYPASS is active
            dout(0) <= data(0); -- then route the raw source data
            dout(9) <= data(1); -- to the output register
            dout(8) <= data(2); -- the bit order set here must match
            dout(7) <= data(3); -- output bit order
            dout(6) <= data(4);
            dout(5) <= data(5);
            dout(4) <= data(6);
            dout(3) <= data(7);
            dout(2) <= data(8);
            dout(1) <= data(9);
```

Appendix B. Scrambler VHDL Source Code (continued)

```
        ELSE                    -- if BYPASS is not active
            IF (txFSM = s0 OR txFSM = s1 OR txFSM = s3) THEN
                                -- and the sync control machine
                                -- is in normal data states, then
                dout <= inp;    -- load scrambled data into the
                                -- output register
            ELSE
                -- however, in the sync state (S2), force the output
                dout <= ”0000000000”;   -- register to all zeros
            END IF;
        END IF;
    END IF;
END PROCESS reg;-- assign the output register to the output pins of the PLD
-- the mapping is out of order because the inp and dout signals
-- were numbered based on the scrambler shifter bit numbers
-- instead of the LSB/MSB data position
scrData(0) <= dout(0);
scrData(1) <= dout(9);
scrData(2) <= dout(8);
scrData(3) <= dout(7);
scrData(4) <= dout(6);
scrData(5) <= dout(5);
scrData(6) <= dout(4);
scrData(7) <= dout(3);
scrData(8) <= dout(2);
scrData(9) <= dout(1);END archscram;
```

Appendix C. Descrambler VHDL Source Code

```
-- DCSRAMRX.VHD
-- x9 + x4 + 1 descrambler

PACKAGE dscramPkg IS
    COMPONENT dscramrx
    PORT (
        rxclk,              -- HOTLink RX CKR recovered clock
        k28dot5_l,          -- HOTLink RX /RDY signal
        bisten_l,           -- RX and TX BIST is enabled
        rst_bist_l,         -- reset BIST-error flag
        bypass: IN BIT;     -- non-scrambled data mode
                            -- 10-bit scrambled data input from
                            --  the HOTLink receiver
        scrData: IN BIT_VECTOR (0 TO 9);
                            -- 10-bit unscrambled data output from
                            --  the descrambler PLD
        bistOut_l: OUT BIT; -- active low BIST error indicator
                            -- 10-bit scrambled data for HOTLink TX
        Data: OUT BIT_VECTOR (0 TO 9));
    END COMPONENT;
END dscramPkg;

ENTITY dscramrx IS
    PORT (
        rxclk,              -- CY7B933 CKR recovered clock
        k28dot5_l,          -- CY7B933 /RDY signal
        bisten_l,           -- RX and TX BIST is enabled
        rst_bist_l,         -- reset BIST-error flag
        bypass: IN BIT;     -- non-scrambled data mode
                            -- 10-bit scrambled data input from
                            --  the HOTLink receiver
        scrData: IN BIT_VECTOR (0 TO 9);
                            -- 10-bit unscrambled data output from
                            --  the descrambler PLD
        bistOut_l: OUT BIT; -- active low BIST error indicator
                            -- 10-bit scrambled data for HOTLink TX
        Data: OUT BIT_VECTOR (0 TO 9));
END dscramrx;

ARCHITECTURE structural OF dscram2_rx IS
SIGNAL holdRegB: BIT_VECTOR (0 TO 9);   -- input pipeline register
SIGNAL din: BIT_VECTOR (0 TO 9);        -- descrambler equations
SIGNAL d0: BIT_VECTOR (0 TO 9);         -- low order descrambler register
SIGNAL d1: BIT_VECTOR (0 TO 9);         -- high order descrambler register
SIGNAL bistOut_i : BOOLEAN;
SIGNAL resetData: BIT;                  -- clear descrambler registers
SIGNAL rvs: BIT;                        -- input bit used for BIST
```

Appendix C. Descrambler VHDL Source Code (continued)

```
-- declare state machine variable
TYPE rxFSMtype IS (
        s0,                 -- wait in this state
        s1,                 -- first k28.5 detected
        s2                  -- second k28.5 detected
        );
SIGNAL rxFSM: rxFSMtype;

BEGIN
-- Descrambler equations, uses same X^9 + x^4 + 1 polynomial as
-- the scrambler. These equations are assigned to the respective bits
-- of registers d0 and d1.
    din(0) <= holdRegB(0) XOR holdRegB(9) XOR holdRegB(4);
    din(1) <= d0(1) XOR holdRegB(8) XOR holdRegB(3);
    din(2) <= d0(2) XOR holdRegB(7) XOR holdRegB(2);
    din(3) <= d0(3) XOR holdRegB(6) XOR holdRegB(1);
    din(4) <= d0(4) XOR holdRegB(5) XOR holdRegB(0);
    din(5) <= d0(5) XOR holdRegB(4) XOR d0(1);
    din(6) <= d0(6) XOR holdRegB(3) XOR d0(2);
    din(7) <= d0(7) XOR holdRegB(2) XOR d0(3);
    din(8) <= d0(8) XOR holdRegB(1) XOR d0(4);
    din(9) <= d0(9) XOR holdRegB(0) XOR d0(5);

-- input pipeline register
hldreg: PROCESS
BEGIN
    WAIT UNTIL rxclk = '1';
        holdRegB <= scrData;
END PROCESS hldreg;

-- descrambler register
regIntFF: PROCESS
BEGIN
    -- on rising CKR clock capture data from HOTLink receiver into
    -- the input capture register of the descrambler
    WAIT UNTIL rxclk = '1';
        IF (bypass = '1') THEN
        -- if bypass is active then send data from the input direct
        -- to the output register
            d0(0) <= holdRegB(9);
            d1(1) <= holdRegB(8);
            d1(2) <= holdRegB(7);
            d1(3) <= holdRegB(6);
            d1(4) <= holdRegB(5);
            d1(5) <= holdRegB(4);
            d1(6) <= holdRegB(3);
```

Appendix C. Descrambler VHDL Source Code (continued)

```
            d1(7) <= holdRegB(2);
            d1(8) <= holdRegB(1);
            d1(9) <= holdRegB(0);
        ELSIF (resetData = '1') THEN
            d1 <= "0000000000"; -- clear descrambler registers
            d0 <= "0000000000";
        ELSE
        -- descramble data
            d0(1) <= holdRegB(9);
            d0(2) <= holdRegB(8);
            d0(3) <= holdRegB(7);
            d0(4) <= holdRegB(6);
            d0(5) <= holdRegB(5);
            d0(6) <= holdRegB(4);
            d0(7) <= holdRegB(3);
            d0(8) <= holdRegB(2);
            d0(9) <= holdRegB(1);
            d0(0) <= din(0);

            d1(9) <= din(1);
            d1(8) <= din(2);
            d1(7) <= din(3);
            d1(6) <= din(4);
            d1(5) <= din(5);
            d1(4) <= din(6);
            d1(3) <= din(7);
            d1(2) <= din(8);
            d1(1) <= din(9);
            d1(0) <= d0(0);
        END IF;
END PROCESS;

-- assign outputs from scrambler registers
        Data(9) <= d0(0);
        Data(8) <= d1(9);
        Data(7) <= d1(8);
        Data(6) <= d1(7);
        Data(5) <= d1(6);
        Data(4) <= d1(5);
        Data(3) <= d1(4);
        Data(2) <= d1(3);
        Data(1) <= d1(2);
        Data(0) <= d1(1);
```

Appendix C. Descrambler VHDL Source Code (continued)

```
-- descrambler supervisor state machine
rxFSMseq: PROCESS
BEGIN
    WAIT UNTIL rxclk = '1';
    CASE rxFSM IS
        WHEN s0 =>
        -- normal data, wait for first k28.5 to be received
            IF (k28dot5_l = '0') THEN
                rxFSM <= s1;
            ELSE
            -- if none detected, keep checking
                rxfsm <= s0;
            END IF;
        WHEN s1 =>
        -- first k28.5 detected, look for second
            IF (k28dot5_l = '0') THEN
            -- if second k28.5 located then stuff pipeline with x'000'
                rxFSM <= s2;
            ELSE
            -- if not, allow current character to be descrambled
                rxFSM <= s0;
            END IF;
        WHEN s2 =>
            IF (k28dot5_l = '0') THEN
                rxFSM <= s2;
            ELSE
                rxFSM <= s0;
            END IF;
        WHEN OTHERS =>
            rxFSM <= s0;
    END CASE;
END PROCESS rxFSMseq;

resetData <= '1' WHEN (rxFSM = s2 OR (rxFSM = s1 AND k28dot5_l = '0'))
                 ELSE '0';
```

Appendix C. Descrambler VHDL Source Code (continued)

```
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- BIST (Built-In Self-Test) controller process. This process
-- is used to control and monitor the status of the BIST function.
-- This state machine was placed in the transmitter scrambler PLD
-- (rather than the receiver descrambler PLD) because of available
-- PLD resources.
--
-- This state machine is initialized by the RESET signal.

rvs <= scrData(9);      -- map scrambled data bit to RVS signal

bist: PROCESS
BEGIN
    WAIT UNTIL rxclk = '1';
    IF (rst_bist_l = '0' OR bisten_l = '1') THEN
        bistout_i <= false;
    ELSE
        IF (rvs = '1') THEN
        -- if the HOTLink RX is in BIST mode and an RVS
        -- error is indicated, then a BIST miscompare has occurred
            bistOut_i <= true;  -- error detected
        ELSE
        -- if an error is not preset on this specific cycle,
        -- then maintain the current state of the BIST error flag
        -- This means that if an error is detected in any cycle
        -- the error indication can only be cleared by a RESET
            bistOut_i <= bistOut_i;
        END IF;
    END IF;
END PROCESS bist;

-- Map internal BIST error indicator to an active low output pin.

bistOut_l <= '0' WHEN bistOut_i ELSE '1';

END structural;
```