Application Report

Communication Methods for Data Integrity Using Delta-Sigma Data Converters

TEXAS INSTRUMENTS

Bob Benjamin *Precision ADC*

ABSTRACT

There are many approaches to verifying the transmission of data from one device to another. One method uses simple addition to create a checksum of the data. A different method uses polynomial division of the data called cyclic redundancy check (CRC). Other devices may transmit data twice where the second transmission is the bitwise-inverted version of the data. There is even the use of Hamming code for error correction. Sometimes a combination of methods is useful. However, not all devices have a specific built-in hardware approach to data integrity. As alternatives, embedded firmware can use methods such as a write then verify with a read, or read data multiple times. Selecting a device and data integrity method depends on the desired degree of certainty and the size of the data packet to be transmitted.

Table of Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

In just about every aspect of life there is a digital transmission. The transmission can vary in length, distance and speed. Digital streams are common in satellite TV and smartphone data. In many cases an error in transmission is nothing more than a nuisance such as a momentary distortion in a picture signal or a click or pop in the audio. However, in critical systems an error is problematic and even life threatening. Some examples of critical systems are autonomous vehicles and aircraft flight control systems. A non-life threatening example requiring a high degree of data integrity occurs when saving data to storage media on a computer.

There are many possible causes for a disruption of a digital stream of data. Ideally the best method for maintaining data integrity is to prevent any occurrence of a disruption in the first place. But even when using best practice in system design there is still the possibility for an unforeseen occurrence that can alter the transmission of digital data.

When designing a system, first analyze the affect of a disruption. If the disruption is within a critical system, such as used in flight control, then it is important to know when the event takes place and have a mechanism to alert or handle the data corruption when it happens. To ascertain whether there is a digital transmission error, or fault, a method of data analysis is used to determine if data corruption occurred. To better understand a fault condition it is important to know what type of error can exist in the data.

Using the example of a simple go or no go, and sometimes called ACK or NACK, within a communication packet helps in understanding the necessity of data validation. As many are familiar with I^{2}C communication, this protocol will be used as an example. I^{2}C uses 8-bit data transfers followed by a 9th bit indicating if the data transfer is valid (ACK) or invalid (NACK) by the receiving device. If the receiving device holds the data line (SDA) low at the 9th clock (SCL), the transmitting device understands that the communication is acknowledged (ACK) and the communication was received. However, with this type of system there is no clear indication if the data are truly valid only that data is received (ACK) or not received (NACK) as expected. There is no mechanism in this approach to determine if the data are valid from the transfer from device to device.

Another common method of digital communication is by universal asynchronous receiver transmitter (UART). Depending on the implementation, the data transfer uses a method called parity to check the integrity of the data. Each byte of data is framed with a START bit and STOP bit(s). The data bits are contained between the START and STOP frame. Parity is used as a method to determine whether an ASCII character was transmitted as expected. The data contains a 7-bit ASCII character along with a PARITY bit for a total of 8 bits of data. In binary terms, a "1" is counted for each of the 7 bits of the character transmitted. For example, the ASCII character "A" has the value 41h or 1000001b containing two "1"s. The terminology of EVEN and ODD parity communicates to the end user whether the value of the parity bit relates to an even number of "1" values or an odd number of "1" values when including the parity bit. Using parity allows for some acknowledgment that data are valid or invalid. But as this simple parity is useful it is possible for multiple bit flips within the communication to appear as valid data that may actually be invalid. As communication evolves with increasing data length and speed, data integrity is a greater issue as more and larger packets of data are transmitted. This requires more comprehensive methods for validating data transfers.

When binary data transmission became commonplace through the use of modems to connect computers, servers and the internet, data integrity verification techniques improved. Just as modem speeds increased, so did the need for better methods of error detection within the digital transmission. Not only is data integrity a point to point issue, but is also a major concern with both wired and wireless networks and the streaming of data to multiple end points.

Similar issues with data integrity can be found within an embedded system where there are many possible options to ensure data integrity. One simple option is to send data, then verify the data by having the receiver return the data back to the sender for comparison. Other systems may employ methods that invert the data for comparison or use a checksum of the data. A more complex method uses a polynomial based cyclic redundancy check (CRC). Each of these methods have differing degrees of difficulty for integration and differing results with respect to determining data integrity.

Table 1-1. Data Integrity Methods

Method	Advantages	Disadvantages
Inverted Data	Easy and quick computation	Doubles the length of transmission; No error correction capability

Table 1-1. Data Integrity Methods (continued)

Method	Advantages	Disadvantages
Parity	Easy and quick computation; Detects single-bit error in short data packets	Prone to error due to many combinations resulting in the same parity value; Useful for small data packets only; No error correction capability
Checksum	Easy and quick computation; Detects single-bit error and some multi-bit error in short data packets	Prone to error due to many combinations resulting in the same checksum value; Useful for small data packets only; No error correction capability
CRC	Fewer combinations of bit flips for the same CRC value resulting in a more comprehensive analysis	Lengthy computation times required or a LUT can be used but increases memory usage; No error correction capability
Hamming	Increased detection for multi-bit error; single-bit error correction	Lengthy computation times required

Not all analog-to-digital converters (ADCs) have a built-in hardware method for determining communication error. For these types of devices, anything written to the device should be verified with a read. For the conversion data, the result should be read multiple times comparing the results received. The ADC device datasheet will indicate if a hardware method is available for data integrity checks and whether or not the device is both ingoing and outgoing data integrity or just outgoing. When choosing an ADC for a particular application consider the criticality of the system with regards to the method of data integrity supported by the ADC.

Checksum and CRC are common hardware methods used by ADCs to append a coded value to the end of a data message transfer. The appended information is used to determine if an error occurred in transmission. For devices using checksum, such as the ADS1259, the conversion data are followed by a checksum byte. The checksum is the addition of the three data bytes, from the 24-bit conversion result, and a constant of 9Bh. The constant applies an offset to the data reducing the probability of the same checksum occurring multiple times. When adding the bytes together, any overflow is ignored in the computation. The result of the computed checksum is compared to the transmitted checksum. If the comparison of the computed checksum is equal to the checksum byte then the validation passes.

When the data integrity implementation is CRC, such as with the ADS124S08, the receiver must verify the data following the completed transmission. The message portion of the transmitted data is computed and compared to the transmitted CRC value. If the computed CRC value does not match the transmitted CRC value, an error in transmission has occurred. In this way the comparison is similar to using checksum, but the computation is much different. The CRC has a higher degree of data validation compared to the checksum, but requires more processing effort since the CRC uses polynomial division instead of simple addition of the checksum.

Some hardware methods that are offered by the ADS122U04 and ADS122C04 families in addition to CRC also support less complicated approaches to verifying data integrity. The feature employed in these devices is by transmitting the data twice. The first transmission is the non-inverted data, and the second transmission is a bitwise inversion of the original data. Using an exclusive 'OR '(XOR) of the two transmitted data should result in all "1"s and no "0"s if the communication was successful.

Regarding this application note, one focus covers the use of checksum and CRC. Included within the discussion are the computations and functions for use within an embedded processor using "C" firmware. It is possible to use other methods of computation. This would include the use of internal hardware peripherals within the processor. However a hardware CRC peripheral of a processor, depending on polynomials used, may not have the same exact polynomial implementation used by the ADC.

Most of the information and examples can also be adapted for use by other ADCs such as the ADS124S08, ADS1261 and ADS1262 family of devices. Additionally this application note includes a discussion on the use of Hamming code as a data integrity feature supported by the ADS131A04 family of devices. The ADS131A04 is a device that can be configured for use in critical applications. For the ADS131A04 the data integrity can be validated on both incoming and outgoing communication. The Hamming code option allows for single-bit error detection and correction. Within the Hamming byte is a simple 2-bit checksum to help in determining some multi-bit error. CRC can also be added to the transmission in addition to the Hamming/checksum to uncover most multi-bit error.

Table 1-2. Example List of Devices and Data Integrity Methods

Device	Interface	Data Direction	Data Integrity Method(s) Used	Polynomial	Data Integrity Computation
ADS122C04	I^{2}C	MSB first	Inverted Data; CRC-16-CCITT	$x^{16} + x^{12} + x^5 + 1$; initial value FFFFh	Register reads, the 24-bit conversion result and the data counter byte when enabled
ADS122U04	UART	LSB first	Inverted Data; CRC-16-CCITT	$x^{16} + x^{12} + x^5 + 1$; initial value FFFFh	Register reads, the 24-bit conversion result and the data counter byte when enabled
ADS124S08	SPI	MSB first	CRC-8-ATM (HEC)	$x^8 + x^2 + x + 1$; initial value 00h	24-bit conversion result and the STATUS byte when enabled
ADS131A04	SPI	MSB first	CRC-16-CCITT and/or Hamming	$x^{16} + x^{12} + x^5 + 1$; initial value FFFFh	Ingoing and outgoing transmissions; CRC and Hamming can both be enabled at the same time
ADS1259	SPI	MSB first	Checksum	N/A	24-bit conversion data plus offset of 9Bh
ADS1261	SPI	MSB first	CRC-8-ATM (HEC)	$x^8 + x^2 + x + 1$; initial value FFh	Incoming commands, 24-bit conversion result and the STATUS byte when enabled
ADS1262	SPI	MSB first	Checksum or CRC-8-ATM (HEC)	$x^8 + x^2 + x + 1$; initial value 00h	Checksum for 32-bit conversion data plus offset of 9Bh or when using CRC 32-bit conversion data only

2 Simple Checksum

The checksum can be appended to conversion results on devices such as the ADS1259 and ADS1262. Checksum allows for error detection of single-bit error and some combinations of multi-bit error. The checksum byte is computed by adding the value for each of the conversion data bytes together along with the addition of a constant. For the mentioned devices the constant is 9Bh. As the checksum is a byte in length, any carry from the addition is ignored. In the case of the ADS1259, which is a 24-bit device, the MSB, Mid-Byte and LSB are added together with the constant. The result of the checksum is appended to and transmitted with the conversion data.

The checksum is easy to compute with minimal processing effort. However, if there are multiple bit errors present it is possible for the errors to go undetected since multiple combinations of different data added together will result in the same checksum value. For example, the addition of conversion data 12h, 34h, 56h and constant 9Bh results in a checksum of 37h. But the data could become corrupt in transmission. If instead the data are received as 12h, 35h, 55h and constant 9Bh, the addition of these bytes result in the same checksum of 37h.

Checksum can be useful when checking for 1 to 2 bit error in small data packets. With small data packets there is some potential for missing the error in the data, but as the data packet size increases so does the potential for missing the multiple bit errors. Using checksum alone is not the best method available for determining data integrity.

2.1 Checksum Code Example

The following code example can be used for devices such as the ADS1259 and ADS1262 that have the checksum data integrity feature. The checksum can be computed by sending a pointer to the data along with the length of the data packet to be considered. The data bytes are summed together along with the constant to create the checksum byte. The computed checksum byte is then compared with the checksum transmitted with the data. The byte values should match otherwise an error occurred.

```
/**
 * Computes a checksum for an array of data bytes.
 *
 * \details Computes a series of bytes pointed to
 * by an array of data for a specified length. A constant 0x9B is also added.
 *
 * \param    uint8_t *data, pointer to data array.
 * \param    uint32_t length, of data array.
 *
 * \returns  uint8_t result.
 */
uint8_t calcChecksum(uint8_t* data, uint32_t length)
{
        uint32_t result = 0;
        uint32_t i;
        for(i = 0; i < length; i++)
            result += data[i];  // Add conversion data bytes together
        result += 0x9B;         // Add constant
        return (uint8_t) result;
}
```

3 CRC

CRC is a polynomial based system and can vary in the number of bits used and the effectiveness of detecting an error for a given polynomial. Checksum differs from CRC in that checksum uses addition, whereas CRC uses division. The desired polynomial is divided into the data leaving a remainder to be transmitted following the data. The effectiveness of using any particular polynomial is indicated by the Hamming Distance (HD) which for the discussion of CRC is the boundary where an undetected bit error may occur. HD signifies the minimum number of bit errors that are undetectable. The number of bits when all errors are detected is one less than the HD. The HD can change relative to the length of the data and generally will decrease as the length of the number of data bits increases.

As a part of the CRC discussion, Hamming Weight (HW) refers to the number of undetectable bit errors for a polynomial for a given number of bit errors. The ideal HW is zero and when the HW is non-zero for a given number of bits then not all combinations of bit errors will be detected should an error occur. The HD relates to the first HW greater than 0. For example, if a polynomial has a HD of 4 , all combinations of 1, 2, and 3-bit error, also known as bit reversals or flips, are detected. If the HW is 11 for 4-bit errors with data lengths of 48 bits or more, then there are 11 combinations of undetectable 4-bit errors for the data packet.

Depending on the polynomial and the total number of bits in the data transfer, the HD might increase for up to a particular number of bits of data. Using the previous example, the HD may be increased from 4 to 5 for a data length less than 48 bits. Much data has been collected regarding the HD of various polynomials by independent research. The point of this discussion is to limit some confusion as to the effectiveness of various polynomials and show that undetected error may still occur when using CRC. Different polynomials are used by various ADCs and in most cases are fixed to a single polynomial. Thus, there may be little to no choice on which polynomial to use. The polynomials used for the ADC may not be the best polynomial to use in a given application, but is chosen due to familiarity or commonality of a particular polynomial. Another reason may be for the availability of hardware implementations on various microcontrollers. As many microcontrollers use 8-bit (byte) communication it becomes useful to use CRC polynomials that are oriented to 8-bits or multiples of 8-bits. This discussion will not focus on which polynomial to use, but rather focus on the actual polynomials used for devices such as the ADS124S08 family of devices (8-bit CRC) as well as the ADS122x04 family of devices (16-bit CRC).

As both the encoding and decoding of the CRC is similar the bit patterns of the remainder can be easily compared. If the bits match, then it is likely the transmission was valid for a given HD. However, if the bits do not match then an error occurred. From a software implementation, the polynomial is divided into the data word resulting in a remainder. From the ADC hardware perspective, these devices utilize some form of logic in relation to the output shift register to compute the CRC value regardless of the data length being transmitted. It is important that the data be analyzed in the same bit order as the initial CRC computation. So careful attention to Endianess (see Table 3-1) and byte order is needed.

3.1 CRC Generic Computations

CRC can be used on transmission of long data lengths into the hundreds, thousands or even longer number of bits. The generic CRC code being discussed for embedded applications is based on a short data transmission length of 32-bits or less in most cases. The data are byte oriented with transmission in byte multiples for a maximum of 4 bytes in the case of a 32-bit transfer. If the maximum data transfer length is a multiple of 8-bits, the CRC computations can be accomplished using pointers to a memory array.

As the CRC is a division of the polynomial into the data, the initial part of the discussion describes the process of determining the remainder using long division. The data value transmitted is divided by the polynomial in use. When the data is the dividend the initial value used is zero in this case. This would be the same as 0h exclusive-OR (XOR) with the data and the result being equal to the data. The remainder from the completed long division where the divisor is the polynomial and the dividend is the data becomes the computed CRC value. So the data (dividend) is divided by the polynomial (divisor) and the computed result is the quotient. The remainder from the computation is the CRC value.

When the initial value is zero and the data has leading zeros, the CRC computation would ignore the leading zeros in the computation. To make the computation of CRC more robust to include any leading zeros, an initial value other than zero is used. The most common non-zero initial value is all 1's for the size of the CRC being used. The non-zero initial value for 8-bit CRC is FFh, and FFFFh for 16-bit CRC. The initial value for the computation is specified in the ADC device datasheet. The initial value and the data are then XOR'ed together to create the dividend.

The implemented polynomial for ADCs is often a standard CRC polynomial, such as CRC-8-ATM or CRC-16-CCITT. These polynomials have been shown to be effective for data transmission integrity checks. Often one of these standard CRC algorithms are implemented in embedded microcontrollers for faster computation. The ADS122C04 uses the 16-bit polynomial CRC-16-CCITT which has been commonly used in communication data transfers. The polynomial is $x^{16} + x^{12} + x^5 + 1$ and is equivalent to 10001000000100001b. The generated CRC result is the remainder of the long division of the data divided by the polynomial. This discussion will not go into detail on the process of modulo 2 math other than to say a subtraction of two values becomes an XOR operation.

The initial 16-bit starting value for the ADS122C04 is given in the datasheet as FFFFh. The example in Figure 3-1 uses a data value that is 24-bits in length similar to a conversion result. The example uses a value of 4E6878h for the data.

As the final result will be a remainder of 16-bits, the initial value is appended with zeroes so that the total length will consist of the length of the data plus the 16-bit remainder.

1. Initial value = 0xFFFF (as per ADS122C04 datasheet)
2. Right pad the initial value with 24 "0"s so that the total length of the initial value is the length of the conversion data plus the length of the CRC (40 bits total)
3. XOR initial value left aligned with converted value: FFFF000000h XOR 4E68780000h = 0xB197780000 (See Figure 3-1)
4. Divide the result of step 2 by the polynomial: B197780000h ÷ 11021h and take the remainder B72Ch as the computed CRC value

The process of dividing the divisor into the dividend results in a remainder of B72Ch as shown in Figure 3-2 which would be the CRC value transmitted by the ADS122C04 following the transmission of the conversion data.

Initial Value = FFFFh	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Data = 4E6878h	0	1	0	0	1	1	1	0	0	1	1	0	1	0	0	0	0	1	1	1	1	0	0	0																
Dividend = B197780000h	1	0	1	1	0	0	0	1	1	0	0	1	0	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3-1. Example of Computing the Starting Value for CRC Computation by XOR of the Initial Value and Data

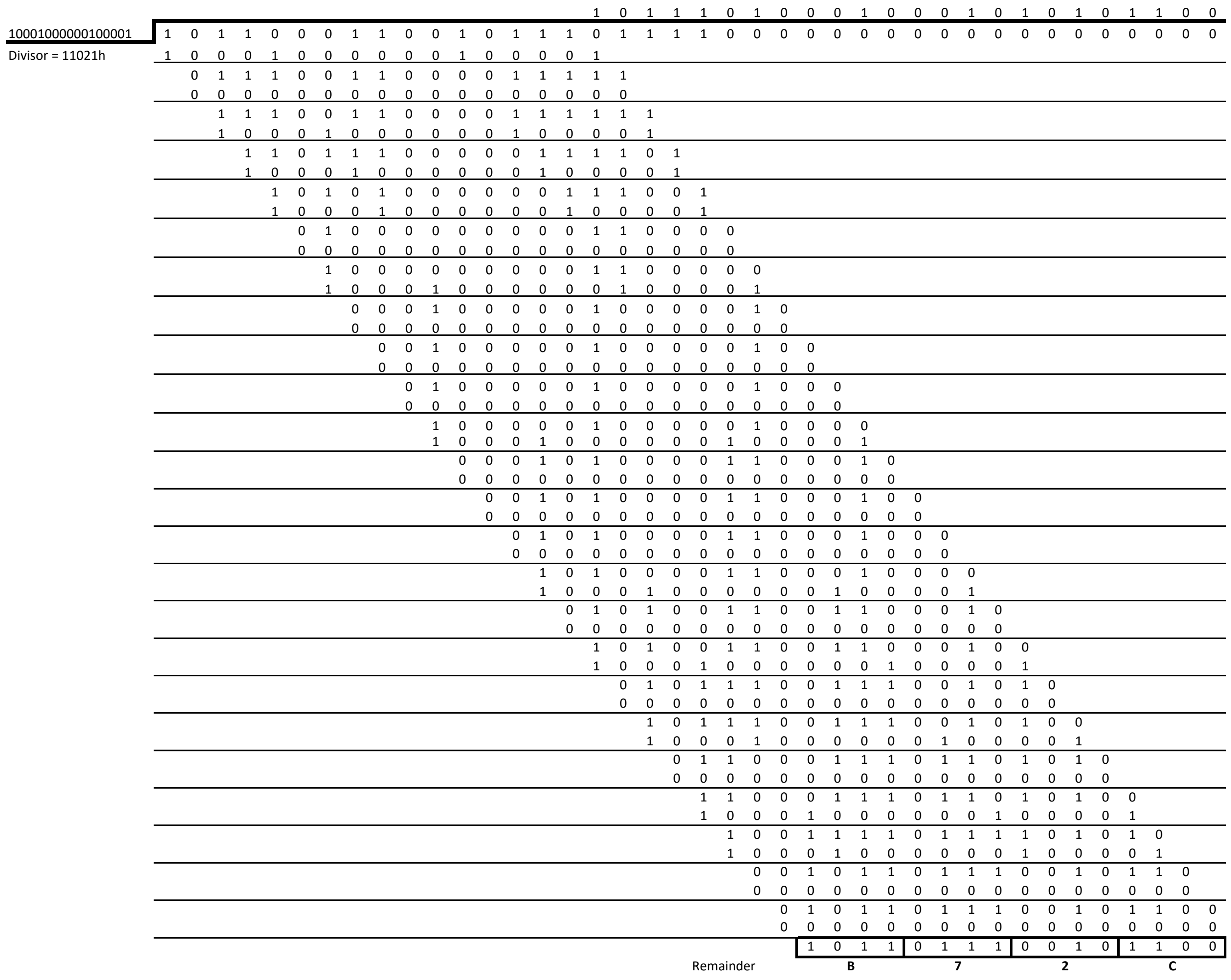


Figure 3-2. Binary Long Division of Two Polynomials of Binary Coefficients

For ADC devices transmitting data serially, the hardware implementation is often through a shift register with the CRC computed as the data are shifted out. Depending on the type of interface, the data may need to be reversed or reflected for the CRC computation. Data reflection is simply the swapping of the order of the bits transmitted. Why the order is important is discussed in more detail when comparing the CRC computations of the ADS122C04 (I^2C) and the ADS122U04 (UART) devices in Section 3.1.3. The difference in data transmission is the UART is LSB first and I^2C is MSB first. What becomes complicated is the reading of the UART receive buffer when transferred to memory. When reading the receive buffer the order of bits are reversed, or reflected, from the order transmitted for each byte. When computing the CRC the proper order of the bits as transmitted must be used.

The microcontroller may also use a hardware implementation for computing CRC. Besides CRC hardware implementations, the embedded processor code can accomplish the same task as long division using a similar XOR process and rotating through each bit and byte returning the CRC as the remainder. The XOR process takes considerable time within the function as a bitwise process. Another common method is to use a lookup table for processing the values. Both the XOR bitwise and lookup table methods are discussed.

3.1.1 Using XOR Bitwise Computation

Software bitwise functions ignore the most high order bit in the polynomial value. The bit can be ignored as the highest order bit is shifted out before the remainder is XORed. For demonstration, the polynomial for CRC-16-CCITT is $x^{16} + x^{12} + x^5 + 1$ (11021h) and is represented by 10001000000100001b. However, the highest order bit is dropped and the value used for computation in the bitwise operation becomes 1021h.

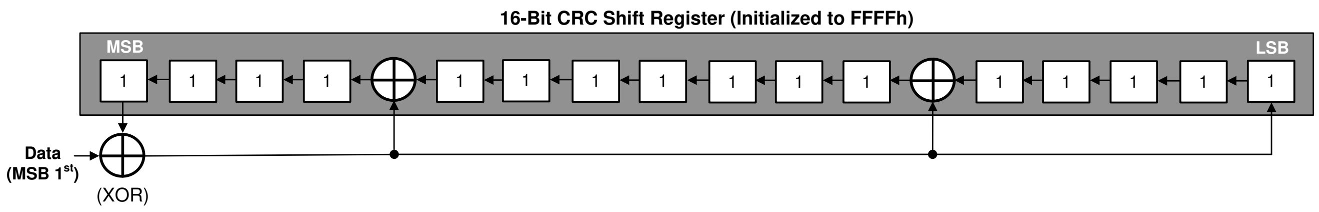


Figure 3-3. CRC-16-CCITT Hardware Implementation

For the 8-bit CRC-8-ATM (HEC) the polynomial is $X^8 + X^2 + X + 1$ (107h) and is represented by 100000111b. Again, the highest order bit is ignored and the value to use in the bitwise operation is 7h.

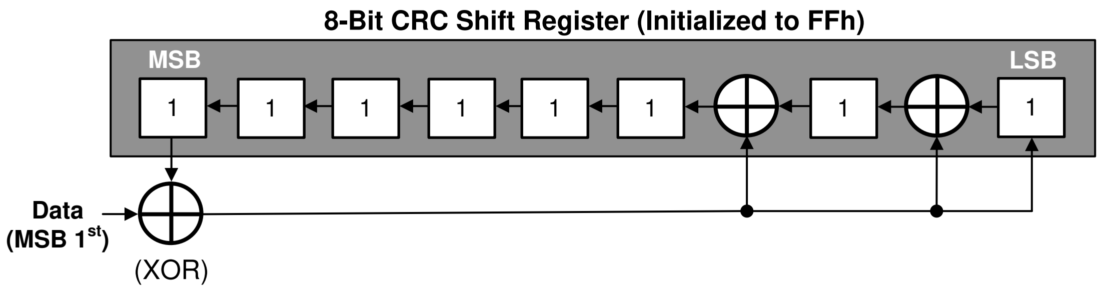


Figure 3-4. CRC-8-ATM (HEC) Hardware Implementation

The following code function is shown for CRC-16-CCITT, but can be easily adapted to other polynomials and lengths. The function replicates the hardware implementation and returns a value based on the size of the CRC length and is signified by the type definition for *crc_t*. For the code example the returned value is an unsigned 16-bit value. Parameters passed are the pointer to the *data* packet to be computed and the *length* of the data packet in number of bytes. The function processes each byte of data by an XOR of each bit with the remainder. The initial value of the *REMAINDER_INIT* is the starting seed value and will differ depending on what ADC is being used.

```
typedef uint16_t crc_t;          // for 8 bits, use uint8_t
#define POLYNOMIAL 0x1021       // CRC16-CCITT, but for 8 bits use CRC-8-ATM(HEC),
                                 // and the polynomial value 0x07
#define REMAINDER_INIT 0xFFFF    // 0xFFFF for 16-bit CRC on ADS1xC04, ADS1x2U04
                                 // 0xFF for 8 bit CRC on ADS1260, ADS1261, ADS1235
                                 // 0x00 for 8 bit CRC on ADS124S0x, ADS114S0x, ADS1262, ADS1263
#define WIDTH  (8 * sizeof(crc_t))
/**
 * Computes CRC for an array of data bytes.
 *
 * \details CRC computation of a series of bytes using XOR with desired polynomial.
 *
 * \param    uint8_t  *data, pointer to data array.
 * \param    uint32_t length, of data array.
 *
 * \returns  crc_t remainder.
 */
crc_t  crcBitwise(uint8_t *data, uint32_t length)
{
    crc_t remainder = REMAINDER_INIT;
    uint32_t byte, bit;
                                                      // For each byte in data packet, perform long
                                                      //    division of polynomials
    for(byte = 0; byte < length; byte++)
    {
        remainder ^= (data[byte] << (WIDTH - 8));     // Get next byte into remainder
        for(bit = 8; bit > 0; bit--)                  // For each bit in the remainder
        {
            if(remainder & (1 << (WIDTH - 1)))
                remainder = (remainder << 1) ^ POLYNOMIAL; // If the top bit is set, left shift
                                                            //   the remainder and XOR it with the
            else                                            //   divisor and store the result in
                                                            //   the remainder
                remainder = (remainder << 1);               // If the top bit is clear, left shift
                                                            //    the remainder
        }
    }
    return remainder;
}
```

The code example is useful for determining the CRC remainder for the data. The computed remainder is required for devices that have both ingoing and outgoing CRC data integrity checks. However, if the data is only being validated for transmission from the ADC, such as conversion data, the CRC value transmitted can be added to the data array. If the CRC remainder is included in the computation and if the CRC matches both the computed and transmitted values, the remainder will be zero. In other words there is an important property of CRC computation where shifting in a matching byte forces the value in the CRC shift register to zero. A non-zero result would indicate an error in transmission. Using this method may simplify the validation by eliminating the direct comparison of the CRC from the transmission and the computed CRC.

3.1.2 Using Lookup Tables

The lookup table requires a table of data for all of the possible combinations of remainders for any incoming byte of data. The table is stored in RAM or FLASH memory for quick access. The computation for the table entries use a method similar to the bitwise XOR when computing each table value. The number of table entries is 256 representing all combination of bits within a byte. Using a lookup table has the advantage of being on the order of four times faster than the bitwise operation and becomes comparable to the time it takes for a checksum operation at the expense of storing 512 bytes for 16-bit entries (256 bytes for 8-bit entries).

3.1.2.1 Table Initialization

The memory size of the table array will depend on the size of the CRC value to be returned. For efficiency, the table boundary should be set to byte alignment. The table consists of 256 16-bit entries for 16-bit CRC or 256 8-bit entries for 8-bit CRC .

```
typedef crc_t uint16_t;          // for 8 bits, use uint8_t
#define POLYNOMIAL 0x1021        // CRC16-CCITT, but for 8 bits use CRC-8-ATM(HEC),
                                 //     and the polynomial value 0x07
#define REMAINDER_INIT 0xFFFF    // CRC16-CCITT, but for 8 bits use CRC-8-ATM(HEC),
                                 // 0xFF for 8 bit CRC on ADS1260, ADS1261, ADS1235
                                 // 0x00 for 8 bit CRC on ADS124S0x, ADS114S0x, ADS1262, ADS1263
#define WIDTH (8 * sizeof(crc_t))
crc_t crcTable[256];
/**
 * Initialization for CRC lookup table to be stored in memory.
 *
 * \details CRC computation for each possible combination contained in a single byte
 * which is then stored to a crcTable array where each array element pertains to a
 * specific byte value.
 *
 *
 * \returns  void.
 */
void  initCRCtable(void)
{
    crc_t remainder;
    uint32_t byte, bit;
                                                    // For each byte in data packet, perform long
                                                    //    division of polynomials
    for(byte = 0; byte < 256; byte++)
    {
        remainder = (byte << (WIDTH - 8));          // Get next byte into remainder
        for(bit = 8; bit > 0; bit--)                // For each bit in the remainder
        {
            if(remainder & (1 << (WIDTH - 1)))
                remainder = (remainder << 1) ^ POLYNOMIAL; // If the top bit is set, left shift
                                                          //   the remainder and XOR it with the
            else                                          //   divisor and store the result in
                                                          //   the remainder
                remainder = (remainder << 1);             // If the top bit is clear, left shift
                                                          //   the remainder
        }
        crcTable[byte] = remainder;
    }
}
```

3.1.2.2 CRC Computation

Instead of using multiple XOR operations, the byte value is retrieved from the CRC table previously initialized and stored in memory.

```
/**
 * CRC computation based on a lookup table previously stored in memory.
 *
 * \details CRC value is computed by an XOR of table data stored from
 * the bitwise XOR of the polynomial.
 *
 * \param    uint8_t  *data, pointer to data array.
 * \param    uint32_t length, of data array.
 *
 * \returns  crc_t remainder, of CRC computation.
 */
crc_t tableCRCcalc(uint8_t *data, uint32_t length)
{
    crc_t remainder = REMAINDER_INIT;
    uint32_t byte, u32i;

    for(u32i = 0; u32i < length; u32i++)                   // For each byte in the data packet
    {
        byte = data[u32i] ^ (remainder >> (WIDTH - 8));
        remainder = crcTable[byte] ^ (remainder << 8);     // Perform table lookup
                                                           //    for the byte remainder
    }
    return remainder;
}
```

3.1.3 CRC Computation Differences Between the ADS122U04 and ADS122C04

As was mentioned in the overview, the CRC value is computed relative to the bit order of the data transmission as it is shifted out of the device. The data transmission from the "U" devices are much different as compared to the "C" devices. The "U" devices use UART transmission with data transmitted LSB first whereas "C" devices use I^2C transmission with data transmitted MSB first. The MSB first data are read into the microcontroller peripheral and reassembled in the same manner as the data are transmitted (when using Big Endian format) as shown in Figure 3-5.

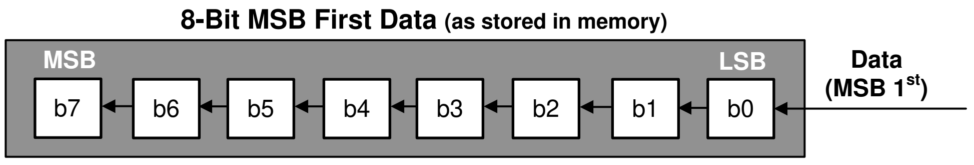


Figure 3-5. Contents of Memory for Shifting Data In MSB 1st

The CRC computation for the "U" devices follow the same process as shown in Figure 3-3. However, the difference for the "U" device is the shift register starts with the LSB first instead of the MSB. The outcome directly impacts the CRC computation and the byte order in which the data are stored into memory.

When computing the CRC by a code function it is helpful to understand the method by which data are stored. As an example, consider a memory made up of byte (8-bit) addressable locations. A 32-bit integer is made up of four adjacent bytes. Consider the integer as an array of four bytes. If the value stored in the first element of the array is the most significant byte (MSB) of the integer, the value is stored as Big Endian. If the least significant byte (LSB) is stored as the first element of the array, then the integer is stored as Little Endian. Table 3-1 shows a comparison of how the 32-bit value is stored in memory. When data are transmitted MSB first, the order of transmission follows the Big Endian format. However when data are transmitted as LSB first, the order of transmission follows the Little Endian format. For the purposes of this discussion the Big Endian format will be used by the microcontroller. However, in both methods the most significant bit of the byte array element is bit 7 and the least significant bit is bit 0.

Table 3-1. Memory Addressing for 32-bit Data

32-bit Data	Memory Address	Big Endian	Little Endian
0A0B0C0Dh	0h	0Ah (MSB)	0Dh (LSB)
	1h	0Bh	0Ch
	2h	0Ch	0Bh
	3h	0Dh (LSB)	0Ah (MSB)

Computation of the CRC is based simply on the data for the bit and byte order it was transmitted. However, the LSB first data is transferred to the microcontroller UART peripheral where the interface will shift the data LSB first, but the contents in memory will be in the reverse order that it was transmitted. As the transmitted data are in reverse order of the data in memory, the computed value of the CRC by the microcontroller will be incorrect in both bit and byte order. For LSB first transmitted data, both the byte order and the bit order within each byte must be reversed, or reflected, for the computation (see Figure 3-6).

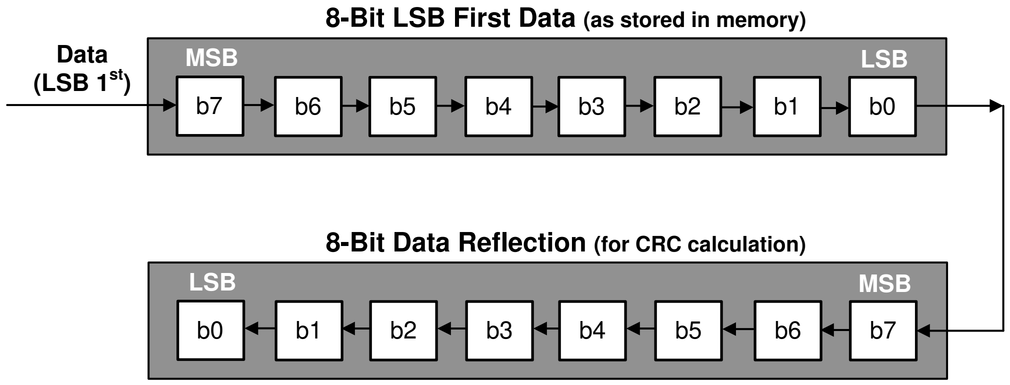
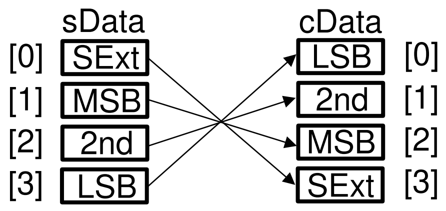

8-Bit LSB First Data (as stored in memory)
Data (LSB 1st)
MSB
LSB
b7
b6
b5
b4
b3
b2
b1
b0
8-Bit Data Reflection (for CRC calculation)
LSB
MSB
b0
b1
b2
b3
b4
b5
b6
b7

Figure 3-6. Contents of Memory for Shifting Data In LSB 1st Requiring Reflection for CRC Computation

If the microcontroller memory is Big Endian and the UART conversion data received is stored as a signed 32-bit integer, it is not enough to simply reverse the order of the bytes representing the integer value. Part of the reason is the original data are 24-bit (16-bit for 16-bit devices) and the integer value is sign-extended. In addition, most microcontroller UART peripherals will clock the data into an 8-bit shift register LSB first, but store the byte in the opposite orientation in memory. The peripheral will realign the bit order in memory that is the reverse, or the reflection, of the order of bits transmitted. However, when computing the CRC for the data packet the order and number of bits must match the same order and number of bits transmitted. For example, if a 24-bit conversion result is stored to a signed 32-bit value, then the CRC computation must take place using the original binary two's complement 3 bytes of data with the proper reflection as opposed to a 4 byte signed number that has been sign-extended.

As an example of the steps required, an array (cData) representing 24-bit conversion data is created from an array (sData) for a signed 32-bit integer. The new array (cData) is used to compute the CRC for comparison to the CRC transmitted from the ADS122U04.

1. Copy the sData array to the cData array where the cData array represents the Little Endian format as illustrated in Figure 3-7.
2. As the conversion data is 24-bit, do not use the sign-extended byte in the cData array.
3. The cData array now contains the correct byte order, but not in the correct bit order.
4. For each byte element in the cData array, reflect the order of bits so that the bit order is the same as originally transmitted as shown in Table 3-2.
5. Compute the CRC for the first three elements in the cData array.



Note: Sign-extended byte (SExt) is not used for calculating CRC for a 24-bit conversion result

Figure 3-7. Changing Byte Order to Reflect LSB 1st

Table 3-2. Reflection of Signed 32-bit Integer to 24-bit Using LSB 1st for CRC Computation

Array Element	sData (32-bit)	cData (24-bit)	cData Reflected
0	00h	78h (0111 1000b)	1Eh (0001 1110b)
1	4Eh	68h (0110 1000b)	16h (0001 0110b)
2	68h	4Eh (0100 1110b)	72h (0111 0010b)
3	78h	N/A	N/A

3.1.3.1 Byte Reflection Example

The following code example is a method to reverse, or reflect, the order of bits within a byte. The order of b^7, b^6, b^5, b^4, b^3, b^2, b^1, b^0 is reflected to the returned bit order of b^0, b^1, b^2, b^3, b^4, b^5, b^6, b^7.

```
/**
 * Reverses the bit order of a byte.
 *
 * \details Reorders the bits in reverse order of byte submitted
 * and returns byte as unsigned byte. This is required as the data is always
 * transmitted out the UART lsb first when comparing TX and RX data.
 *
 * \param    uint8_t u8Byte data byte to be reversed.
 *
 * \returns  uint8_t u8RevByte.
 */
uint8_t revByte(uint8_t u8Byte)
{
    uint8_t u8i, u8RevByte = 0;        // For each bit in the byte, starting with bit 0
    for (u8i = 0; u8i < 8; u8i++)
    {
        u8RevByte |= (u8Byte & 0x01);  // Take one bit of the input byte and store it
                                       //    in the new byte
        u8Byte >>= 1;                  // Get the next bit
        if (u8i < 7) u8RevByte <<= 1;  // Do not shift the last bit of the new byte
    }
    return u8RevByte;
}
```

3.1.3.2 Reassembling Data Using Byte Reflection for CRC Computation

Reassembling the data can be done in various ways while accomplishing the same result. One method is to use a pointer to an array of data and then store the contents to another array in reflected order both by bit and byte order.

```
/**
 * Reconfigures the data for CRC computation when comparing to LSB first transmissions.
 *
 * \details Reorders the bits in reverse order of byte submitted
 * and changes the order of bytes computed for CRC. This is required as the data is always
 * transmitted out the UART lsb first.
 *
 * \param    uint8_t  *data pointer to data array.
 * \param    uint32_t length of data array.
 *
 * \returns crc_t CRCvalue from computation.
 */
crc_t formatCRCdata(uint8_t *data, uint32_t length)
{
    uint32_t u32i, u32j;
    uint8_t checkCRC[length];
    crc_t dataCRC;
    // Start copying with LSB as first element in the CRC array
    u32j = length - 1;
    for(u32i = 0; u32i < length; u32i++)          // For each byte in the data packet
    {
        checkCRC[u32i] = revByte((data[u32j]));  // Reverse the byte
        u32j--;
    }
    dataCRC = crcBitwise(checkCRC, length);      // or TableCRCcalc(checkCRC, length)
    return dataCRC;
}
```

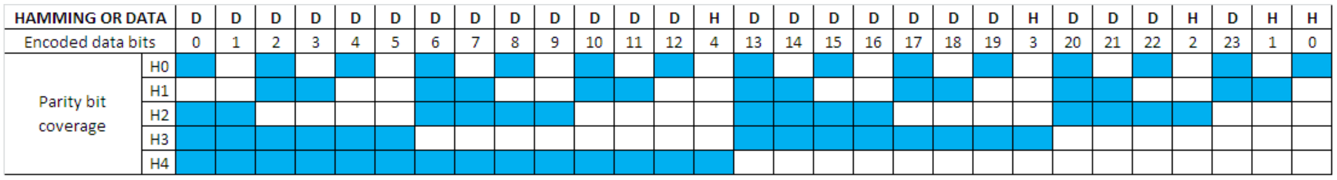
4 Hamming Code

The discussion of Hamming Code (HC) is limited to how the code is used for data integrity checks for ADCs. HC is used to correct one bit of error for a transmitted data block. Hamming codes work well for verifying small data packets. HC cannot detect all multiple bit errors, and multiple bit errors cannot be corrected. The implementation of the Hamming byte used for the ADS131A0x devices includes the HC as well as checksum bits for determining multiple bit error.

For the ADS131A0x devices the HC is made up of five bits (H0-H4) and the bits are interleaved with the data as shown in Figure 4-1. Parity is used to determine the value of each bit in the HC relative to specific bits mapped to the data. Conceptually, the HC bits are interleaved within the data bits. The result is multiple Hamming bits are covering each data bit. This is an improvement upon early data communication when only a single parity was used for data validation. The HC now uses multiple bits to find and correct a single-bit error. Simple parity computation involves counting the number of 1's in a data packet where the parity bit will be either one or zero depending on whether the parity is considered to be odd or even. Odd parity correlates to an odd count, and even parity to an even count. The Hamming bit is set to make the count even parity for the ADS131A0x devices.

The data word for the ADS131A0x family of devices can be either 16-bits or 24-bits. The Hamming byte consists of five Hamming bits with H4 as the most significant bit. Following the Hamming bits are two checksum bits. The least significant bit of the Hamming byte is a constant equal to zero. The total transfer of the data word and the Hamming byte can be either 24-bits or 32-bits. For this discussion we will consider the 32-bit word length consisting of the 24-bits of data along with the Hamming byte.

The five Hamming bits become a part of the data word and are compared to the parity representation of each bit so that the combination of the Hamming parity covers the data bits at least twice so that if a bit is incorrect it can be reconstructed from the remaining Hamming parity bits. For checking if multiple bits have error, then the 2-bit checksum data is used to verify if the result shows a single bit error, or multiple bit error.

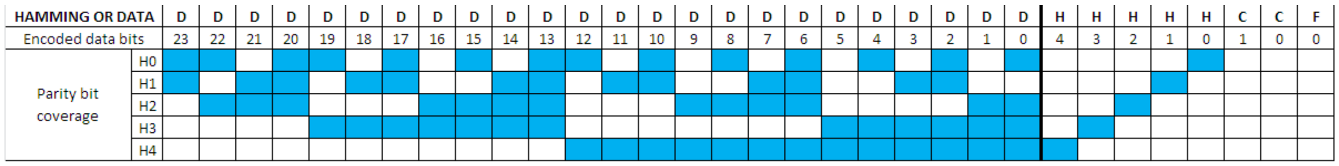


HAMMING OR DATA		D	D	D	D	D	D	D	D	D	D	D	D	D	H	D	D	D	D	D	D	D	H	D	D	D	H	D	H	H
Encoded data bits		0	1	2	3	4	5	6	7	8	9	10	11	12	4	13	14	15	16	17	18	19	3	20	21	22	2	23	1	0
Parity bit coverage	H0	■		■		■		■		■		■		■		■		■		■		■		■		■		■		■
	H1			■	■			■	■			■	■			■	■			■	■			■	■			■	■	
	H2	■	■					■	■	■	■					■	■	■	■					■	■	■	■			
	H3	■	■	■	■	■	■									■	■	■	■	■	■	■	■							
	H4	■	■	■	■	■	■	■	■	■	■	■	■	■	■															

Figure 4-1. ADS131A0x Hamming Codes as Computed

The table data in Figure 4-1 shows the parity bit coverage for each Hamming bit with the Hamming bits interleaved within the data. Each row of the table indicates which data bits are included in each Hamming bit. Note that the table is showing the bits with the least significant bit at the left-hand side of the table and the most significant at the right-hand side. The bits that are checked are indicated in the columns, and the rows are representing the Hamming parity bit. Information from the table can be used to determine a Hamming mask for each Hamming bit. The Hamming mask is used to validate the Hamming bit associated for the mask. The mask only includes the data bits used for a particular Hamming bit. *When converting the table to the Hamming mask used in computation, the 32-bit values are read from left to right and the data would exclude the Hamming bits in the mask.*

Transmission of the data to and from the ADS131A0x devices is most significant bit (msb) first through the least significant bit (lsb) of the data. The Hamming bits will follow in the byte after the data are transmitted. The Hamming is not interleaved within the output as shown in the Figure 4-1. Instead the Hamming is appended to the transmission of the device data as a separate byte. The Hamming bit values will still follow the same pattern and computation but the placing of the Hamming byte following the data is by design. Placing the Hamming byte as the least significant byte (LSB) transmitted allows the user to easily ignore the byte if desired or not transmitted at all if disabled. This allows the device data to appear the same independent of having the Hamming enabled. The Hamming byte also contains the checksum of the data as a 2-bit value (C1 and C0) representing the number of bits set to "1". The least-significant bit (F0) is a fixed constant with a value of "0". The appearance of the transmitted data and Hamming byte will follow the format shown in Figure 4-2.



HAMMING OR DATA		D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	H	H	H	H	H	C	C	F
Encoded data bits		23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	4	3	2	1	0	1	0	0
Parity bit coverage	H0	🟦	🟦		🟦	🟦		🟦		🟦		🟦	🟦		🟦		🟦		🟦		🟦		🟦		🟦					🟦			
	H1	🟦		🟦	🟦		🟦	🟦			🟦	🟦		🟦	🟦			🟦	🟦			🟦	🟦						🟦				
	H2		🟦	🟦	🟦				🟦	🟦	🟦	🟦				🟦	🟦	🟦	🟦					🟦	🟦			🟦					
	H3					🟦	🟦	🟦	🟦	🟦	🟦	🟦								🟦	🟦	🟦	🟦	🟦	🟦		🟦						
	H4												🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦							

Figure 4-2. ADS131A0x Data Hamming Code as Transmitted

When computing the Hamming bit parity, each Hamming bit will follow the pattern shown in Table 4-1 where each bit can be determined by masking out the bits not computed. Using this method greatly simplifies the computation for verifying the data transmission.

Table 4-1. Hamming Bit Mask

HAMMING BIT	MASK VALUE (32-bit)
H0	00DAB555h
H1	00B66CCCh
H2	0071E3C3h
H3	000FE03Fh
H4	00001FFFh

The mask is applied to the data multiple times while counting the number of "1" bits and comparing to the appropriate Hamming parity bit. The masking of the bits is allowed as no Hamming bit includes any other Hamming bit in the parity computation. Also, every data bit is checked by more than one Hamming bit. A Hamming bit error could occur, but will be indicated if the data bits are verified within the other Hamming bit checks. If a data bit is in error, there will be more than one Hamming bit indicating the error in the parity check. When the parity computation is compared to the Hamming bits, the difference will point directly to the data bit in error.

To demonstrate the computation, a 24-bit data example is shown using the value of AC9538h. The results are shown in Table 4-2 for each of the Hamming bits. The process of computation is as follows:

1. H0 computation: Apply mask of H0 to the data (AC9538h & 00DAB55h = 889510h)
2. Find the parity for the result in step 1 (889510h has 7 ones, so parity is odd and H0 becomes 1 to make the total parity even)
3. Repeat steps 1 and 2 for H1 through H4 Hamming bits

Table 4-2. Hamming Bit Computation Example

HAMMING BIT	DATA (HEX)	MASK (HEX)	RESULT (HEX)	RESULT (BINARY)	BIT COUNT	HAMMING VALUE
H0	AC 95 38	DA B5 55	88 95 10	1000 1000 1001 0101 0001 0000	7 (odd)	1
H1	AC 95 38	B6 6C CC	A4 04 08	1010 0100 0000 0100 0000 1000	5 (odd)	1
H2	AC 95 38	71 E3 C3	20 81 00	0010 0000 1000 0001 0000 0000	3 (odd)	1
H3	AC 95 38	0F E0 3F	0C80 38	0000 1100 1000 0000 0011 1000	6 (even)	0
H4	AC 95 38	00 1F FF	00 15 38	0000 0000 0001 0101 0011 1000	6 (even)	0

The computation of the checksum bits is based on the number of "1" values in the binary representation of the data. Continuing to use the hex data of AC 95 38h, the hex value is converted to binary and the number of "1"s counted that represent the value. The binary conversion results in "1010 1100 1001 0101 0011 1000b" and the number of "1"s represented is decimal 11. The value of 11 in binary is "1011b." The checksum is computed by adding all 24 data bits with the modulo of 4. To simplify the computation and avoiding the modulo 4 math only the two least significant bits are used as the checksum bits. For this example the checksum is simply "11b".

The entire Hamming/checksum byte would appear in the format shown in Figure 4-2, which is H4, H3, H2, H1, H0, C1, C0, 0. For the example using 0xAC9538, the bit binary representation of the Hamming/checksum byte becomes "0011 1110b".

4.1 Hamming Code Computation

The Hamming bits are considered interleaved with the data, but the actual transmission is not interleaved. However the data are consistently transmitted the same way. As was previously discussed, simple bit masks can be used to evaluate and compute the Hamming bits instead of using a method of shifting and XORing the data. There are two possible methods for counting the bits. One method is to loop through the masked data and count each time a "1" occurs. This method is rather time consuming similar to the CRC bitwise operation. The second method uses a table to evaluate the data directly which is a much faster method of determining the number of "1"s in the masked data.

After computing the number of "1"s for each of the Hamming bits, the computed value is XORed with the corresponding Hamming bits from the transmitted data. The correct response is zero. If the value is nonzero, then the result can be evaluated to determine which bit is in error. What is special about this process is a single bit error can be corrected. The bit in error could either be a Hamming parity bit error, or an error with a bit in the data. Single bit error correction is possible as each data bit is covered by more than one Hamming parity bit. However, there are some possible combinations that can result in multiple bit error, or combinations of the Hamming bits relative to the data where the data may appear to be correct but is not. The use of the checksum bits along with the Hamming bits represent a more comprehensive solution than analysis of the Hamming bits alone.

To determine if there is a transmission error, the incoming data is computed to determine the expected Hamming bits. The computed Hamming bits and checksum are compared using an XOR to the Hamming byte received. If an error is found with the computation returning a value greater than zero, a correction table is used to make the bit correction once an error is found. The correction table shown in Section 4.1.2.3 would be used with the ADS131A0x devices. The table applies to the complete SPI word transmitted for both the data and Hamming bits. The least significant 3 bits, which are the checksum bits and the constant zero, are excluded. Following the bit-correction, a checksum of the data is then computed by counting the number of "1" bits contained in the restored data. From the computed checksum, the least significant two bits are compared to the checksum bits transmitted in the Hamming byte by XOR. If the checksum fails, then there are multiple bit errors in the data.

4.1.1 Hamming Code Computation Example

The Hamming code computation can be used for both the computation of data to be transmitted as well as the verification of data received. The Hamming bit mask is applied to the data for each of the five Hamming bits. The checksum is also computed and appended to the result. The value returned is the 24-bit data and Hamming byte as an unsigned 32-bit integer.

```
#define HAMMING_BIT0_MASK 0x00DAB555
#define HAMMING_BIT1_MASK 0x00B66CCC
#define HAMMING_BIT2_MASK 0x0071E3C3
#define HAMMING_BIT3_MASK 0x000FE03F
#define HAMMING_BIT4_MASK 0x00001FFF
#define CHECKSUM_BIT_MASK 0xFFFFFF00
#define HC_FIX_FAIL 0xFFFFFFFF
/**
 * Computation of the Hamming bits for the 24-bit data that is input.
 *
 * \details Computation of Hamming bits using a bit mask for each Hamming bit then
 * counting of the number of bits set in the masked value.  This is completed 5 times
 * once for each Hamming bit. The checksum is also computed returning the complete byte
 * that is appended to the 24-bit data. The 'in' value used for the computation assumes
 * the data transferred is not including a hamming byte. When the computation is made the
 * Hamming byte is added to the input value which is accomplished by shifting the 24-bit
 * value left by 8 bits.
 *
 * \param    uint32_t in of the value to be computed.
 *
 * \returns  uint32_t out containing the 24-bit data appended with the Hamming byte value.
 */
uint32_t calcHamming(uint32_t in)
{
    // Take the integer value passed and convert to a format that will include the hamming
    //    byte when passed back
    uint32_t out = in << 8;
    // The 5 hamming bits are computed by taking the input value, and ANDing with the
    //   mask value followed by counting the number of bits with the result ANDed with
    //   0x01. If the count is odd, then the hamming bit is set to make the count even.
    //   If the count is even, then hamming bit is not set.
    uint32_t hamming =
        ((countBits(HAMMING_BIT0_MASK & in) & 0x01)      ) |
        ((countBits(HAMMING_BIT1_MASK & in) & 0x01) << 1) |
        ((countBits(HAMMING_BIT2_MASK & in) & 0x01) << 2) |
        ((countBits(HAMMING_BIT3_MASK & in) & 0x01) << 3) |
        ((countBits(HAMMING_BIT4_MASK & in) & 0x01) << 4) ;
    // The returned result is the data shifted left by 8, ORed with the hamming bits
    //   shifted left by 3, ORed with the checksum of the data ANDed with 2-bits (0x03)
    //   which is shifted left by 1 with the LSB as 0 for a total of 5 hamming,
    //   2 checksum and a '0' for a total of 8 bits.
    return    (out | (hamming << 3) | ((countBits(in) & 0x03) << 1));
}
```

4.1.1.1 Counting Bits for Parity and Checksum Computations

There are a couple of possible methods for determining the number of "1" bits within a value being considered. One method is to simply check the data bit by bit. A faster method is using a lookup table.

4.1.1.1.1 Example of Counting Set Bits in the Data

One method of computing the number of bits set in the data is to evaluate each bit one at a time. The following method requires 32 passes through the data to evaluate each of the 32 bits.

```
/**
 * Computation of the number of set bits in a 32-bit value.
 *
 * \details Counting of the number of bits set in a value using a loop and
 * evaluating each least significant bit and then right shifting the remaining value by 1.
 *
 * \param    uint32_t in of the value to be computed.
 *
 * \returns  uint32_t numBits of the computation.
 */
uint32_t countBits (uint32_t in)
{
    uint32_t numBits = 0;
    while (in != 0)
    {
        if (in & 0x01) numBits++;
        in >>= 1;
    }
    return numBits;
}
```

4.1.1.1.2 Example of Counting Set Bits Using a Lookup Table

A faster method of computing the number of set bits in the data is by using a lookup table. The following example uses data in a 4-bit lookup table. This requires eight passes to evaluate 32 bits. A larger lookup table reduces the number of passes, but increases the size of the table in memory.

```
uint32_t bitsArray[] = {
    0, // 0
    1, // 1
    1, // 2
    2, // 3
    1, // 4
    2, // 5
    2, // 6
    3, // 7
    1, // 8
    2, // 9
    2, // 10
    3, // 11
    2, // 12
    3, // 13
    3, // 14
    4, // 15
};
/**
```

```
 * Computation of the number of set bits in a 32-bit value by lookup table.
 *
 * \details Counting of the number of bits set in a value using a loop and
 * evaluating each nibble in the lookup table and then right shifting the remaining value by 4.
 *
 * \param    uint32_t in of the value to be computed.
 *
 * \returns  uint32_t numBits of the computation.
 */
uint32_t countBits (uint32_t in)
{
    uint32_t numBits = 0;
    while (in != 0)
    {
        numBits += bitsArray[in & 0x0F];
        in >>= 4;
    }
    return numBits;
}
```

4.1.2 Validation of Transmitted Data

As data are received, the Hamming bits can be computed from the received data. For verification, the Hamming bits received are compared to the computed Hamming bits. For a complete comparison, the checksum bits are also computed and compared to the received checksum.

4.1.2.1 Hamming Validation

Hamming bits are computed by taking the received data and computing the Hamming bits and then comparing to Hamming bits received as a bitwise XOR.

```
/**
 * Verification of the Hamming/Data by recomputing and comparing by XOR to the transmitted data.
 *
 * \details Computation of Hamming bits using a bit mask for each Hamming bit then
 * counting of the number of bits set in the masked value.  This is completed 5 times
 * once for each Hamming bit. The checksum is also computed returning the complete byte
 * that is appended to the 24-bit data. The 'in' value used for the computation assumes
 * the data transferred is not including a hamming byte. When the computation is made the
 * Hamming byte is added to the input value which is accomplished by shifting the 24-bit
 * value left by 8 bits.
 *
 * \param    uint32_t in of the value to be computed and compared.
 *
 * \returns  uint32_t ham returns 0 if a match or the computation from the Hamming bits/checksum.
 */
uint32_t checkHamming(uint32_t in)
{
    // When validating the hamming, take the input data and compute the hamming less the
    //    hamming byte
    // So take 'in' and right shift 8 and then compute the hamming
    uint32_t ham = calcHamming((in & 0xFFFFFF00) >> 8);
    // The computed hamming byte is XORed with the hamming byte returned from the data which
    //    ideally would be 0 for hamming code plus checksum
    //    as the ham value is the complete 32-bit word with hamming that was computed from
    //    the data and compared to what was actually transmitted
    uint32_t res = ham ^ in;
    if(res == 0) return res;
    else return (ham & 0x000000FF);
}
```

4.1.2.2 Checksum Validation

The checksum is computed from the received data and compared to the checksum value transmitted by using a bitwise XOR. A checksum error occurs if a nonzero value is computed from the XOR. The checksum used for the ADS131A0x differs from the computation for the ADS1259 discussed in Section 2.1.

```
/**
 * Computation and verification of the checksum from the Hamming byte.
 *
 * \details Computation of checksum bits using a checksum bit mask and counting
 * the number of bits set in the masked value.
 *
 * \param    uint32_t in of the value to be computed and compared.
 *
 * \returns  uint32_t Returns 0 if a match or non-zero for a failure.
 */
uint32_t checkSum(uint32_t in)
{
    // Compute the checksum for the data after stripping out the hamming/checksum byte
    // compare the the computed checksum with the hamming/checksum byte by XORing
    // the result should be '0' for a proper checksum.  Remember to compare after moving
    // the checksum left by 1 to place the checksum in the proper position.  Also remove
    // all other bits in the compare that are not checksum by ANDing with
    // 0x06 (0x03 shifted left by one)
    return (in ^ ((countBits(in & CHECKSUM_BIT_MASK) << 1))) & 0x06;
}
```

4.1.2.3 Error Correction

After using the Hamming and checksum validation, it is possible to correct single bit error. The data validation and correction takes place first and the bit is corrected if possible. If the data are restored, the checksum is reevaluated. If a multiple bit error or checksum error occurs, the transmission is invalid.

```
uint32_t fixResults[] = {
    0,         // datablock OK
    1 << 3,    // no error in data (H0)
    1 << 4,    // no error in data (H1)
    1u << 31,  // data[23] error
    1 << 5,    // no error in data (H2)
    1 << 30,   // data[22] error
    1 << 29,   // data[21] error
    1 << 28,   // data[20] error
    1 << 6,    // no error in data (H3)
    1 << 27,   // data[19] error
    1 << 26,   // data[18] error
    1 << 25,   // data[17] error
    1 << 24,   // data[16] error
    1 << 23,   // data[15] error
    1 << 22,   // data[14] error
    1 << 21,   // data[13] error
    1 << 7,    // no error in data (H4)
    1 << 20,   // data[12] error
    1 << 19,   // data[11] error
    1 << 18,   // data[10] error
    1 << 17,   // data[9] error
    1 << 16,   // data[8] error
    1 << 15,   // data[7] error
    1 << 14,   // data[6] error
    1 << 13,   // data[5] error
    1 << 12,   // data[4] error
    1 << 11,   // data[3] error
    1 << 10,   // data[2] error
    1 << 9,    // data[1] error
    1 << 8,    // data[0] error
               // not correctable 30
               // not correctable 31
               // As H0 covers 14 bits and H1-H4 cover 13 bits a
               //   multi-bit error will occur if hamming is 30
               //   or 31 and cannot be corrected
};
/**
 * Data validation and repair of single-bit error.
 *
 * \details Validation and restoration of data using the Hamming bit
 * information to correct for single-bit error.  The 'in' value is checked
 * first and if the XOR returns a 0, then the data contents are validated relative
 * to the Hamming bits and then are checked for multi-bit error using the checksum.
 * If the Hamming bits do not match the data, then the Hamming bits are used to
 * correct the data if possible by using an XOR of the computed Hamming to the transmitted
 * Hamming.  Using a lookup table based on the correction factors and the XOR value the
 * data is restored and then a checksum is computed to make sure there were no multi-bit
 * errors.
 *
 * \param    uint32_t in of the value to be computed and compared.
 *
 * \returns  uint32_t res returns 0 if the data was correctable or valid.
 *                        A non-zero is invalid data.
 */
uint32_t fixHamming(uint32_t in)
{
    uint32_t res;
    uint32_t fix;
    // checkHamming will return the 5 hamming bits from the data 'in'
    // When using an assignment in the conditional statement, the conditional evaluates the
    //    assignment.
    // If checkHamming returns something other than '0', the result is attempted to be fixed
    if(res = checkHamming(in))
    {
        // Based on the results of the hamming code not '0' the result is corrected from the
        // lookup by adjusting the 32 bit value from 'in' from the table of 2^5 (or 32
        // possible combinations). In summary, 24 values out of 32 are for 1-bit correction,
        // 5 values for no error in data (1bit error in Hamming Code), 1 value for no error
        // in both data and Hamming code and 2 values for more than 1 bit error in either
        // data or Hamming Code or both. Since only two values are assigned for more than
        // 1 bit error cases, there are high probabilities that more than 1 bit-error will
        // be mistakenly treated as 1-bit error by the algorithm. To reduce such
        // possibilities, 2 bit checksum are added for the error detection in addition to
        // the Hamming Detection.

        // Quick check for multi bit error in hamming value
        fix = res >> 3;
        // Then XOR computed hamming bits with 'in' value hamming bits
        fix = fix ^ ((in & 0xFF) >> 3);
        // If there is a known multi-bit error then fail the fix
        if (fix > 29) return HC_FIX_FAIL;
        else
        {
            // Bit error occurred and attempting to fix
            fix = (fixResults[fix]);
            // Identified error could be either hamming or data error where 'in' value is
            //   XORed with the fix value
            res = in ^ fix;
        }
    }
    else
    {
        res = in;
    }
    // res is now containing either the original or the fixed value of data which is
    // data + hamming following the hamming check and data correction, multi-bit
    // errors are checked using the checksum bits checkSum will exclude the hamming bits
    // in the count. If the checkSum returns something other than 0, the
    // checkSum fails, otherwise return 0
    if(checkSum(res))
    {
        return HC_FIX_FAIL;
    }
    return res;
}
```

5 Summary

As mission critical systems become more commonplace with concerns of safety, validation of the transfer of data becomes a priority. Data integrity can be analyzed in a variety of ways using many differing techniques. Choosing the correct method for the application may be difficult. However, learning the methods discussed can be useful in deciding the level of data integrity required and choosing the best ADC for the job.

6 References

- Philip Koopman, Carnegie Mellon University, *Cyclic Redundancy Checks (CRCs) and Checksums*
- Philip Koopman, Carnegie Mellon University, *CRC Summary Tables*
- Philip Koopman, Carnegie Mellon University, *Selection of Cyclic Redundancy Code and Checksum Algorithms to Ensure Critical Data Integrity*
- Barr Group, *CRC Implementation Code in C/C++*
- California State University, Sacramento, *Modulo 2 Arithmetic*
- Texas Instruments, *ADS122C04 Data Sheet*
- Texas Instruments, *ADS122U04 Data Sheet*
- Texas Instruments, *ADS131A04 Data Sheet*
- Texas Instruments, *ADS124S08 Data Sheet*
- Texas Instruments, *ADS1235 Data Sheet*
- Texas Instruments, *ADS1259 Data Sheet*
- Texas Instruments, *ADS1260 Data Sheet*
- Texas Instruments, *ADS1262 Data Sheet*

7 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (June 2020) to Revision A (August 2021) **Page**

- Updated the numbering format for tables, figures and cross-references throughout the document...................2