Application Report

使用 Δ-Σ 数据转换器检查通信中数据完整性的方法

Bob Benjamin *Precision ADC*

摘要

有许多方法可以验证器件之间的数据传输，一种方法是使用简单的加法来计算数据的校验和，另一种方法是对数据进行多项式除法，称为循环冗余校验 (CRC)。其他器件可能会传输数据两次，其中第二次传输的是数据的按位取反版本。此外，还可以使用汉明码进行纠错。有时，组合使用上述方法会更实用。但是，并非所有器件都具有特定的内置硬件方法来检查数据完整性。作为替代方案，嵌入式固件可以使用写入后通过读取进行验证或多次读取数据等方法。如何选择器件和数据完整性检查方法，取决于所需的精度要求和待传输数据包的大小。

内容

插图清单

表格清单

商标

1 引言

数字传输几乎涉及生活的方方面面，而且，传输的长度、距离和速度会各不相同。数字流在卫星电视和智能手机数据中很常见。在许多情况下，传输中发生错误仅仅会带来一些微小的影响，例如图像信号瞬间失真或音频出现咔嗒声或砰砰声。不过，在关键系统中，错误会导致严重的问题，甚至会危及生命。例如，自动驾驶汽车和飞机中的飞行控制系统就属于关键系统，而将数据保存到计算机上的存储介质，需要高度的数据完整性但又不会对生命造成威胁。

数字数据流受干扰的原因可能有多种。理想情况下，保持数据完整性的较好方法是首先防止产生干扰。但即使在系统设计中采用最佳做法，仍有可能发生不可预见的事件，从而改变数字数据的传输。

在设计系统时，首先要分析干扰所带来的影响。如果关键系统（如飞行控制系统）内存在干扰，则需要了解干扰发生的时间并在发生干扰时发出警报或防止数据受损。为了查明是否存在数字传输错误或故障，可使用一种数据分析方法来确定是否发生数据损坏。为了更好地了解故障情况，需要了解数据中存在的错误类型。

使用通信数据包中的简单通过或不通过（有时也被称为 ACK 或 NACK）示例，有助于理解数据验证的必要性。许多人都熟悉 I^2C 通信协议，因此本文将以该协议作为示例。I^2C 使用 8 位数据传输，随后的第 9 位表示接收器件的反馈数据传输是有效 (ACK) 还是无效 (NACK)。如果接收器件在第 9 个时钟 (SCL) 位使数据线 (SDA) 保持在低电平，则发送器件得知通信信息已接收并被确认 (ACK)。不过，这种类型的系统不会明确表示数据是否真正有效，而是仅表示是否已按预期接收数据（ACK 或 NACK）。这种方法中没有任何机制来确定器件间传输的数据是否有效。

另一种常见的数字通信方法是使用通用异步接收器/发送器 (UART)。根据具体实现方式，数据传输使用奇偶校验法来检查数据的完整性。数据的每个字节都配备一个开始位和一个停止位，数据位介于开始位和停止位之间。奇偶校验法用于确定 ASCII 字符是否按预期传输。数据包含一个 7 位 ASCII 字符和一个奇偶校验位，共 8 位数据。在二进制中，对所传输字符 7 个位中每一个值为“1”的位进行计数。例如，ASCII 字符“A”的值为 41h 或 1000001b，其中包含两个“1”。术语偶校验和奇校验告知最终用户在包含奇偶校验位时奇偶校验位的值是与偶数个“1”值相关还是与奇数个“1”值相关。使用奇偶校验可在一定程度上确认数据是有效还是无效。不过，尽管这种简单的奇偶校验很有用，但通信中的多个位可能发生翻转，从而显示为有效数据，但实际上可能无效。通信随着数据长度和速度的增加而不断发展，随着传输的数据包变得更多、更大，数据完整性问题日益凸显，这就要求采用更全面的方法来验证数据传输。

随着使用调制解调器连接计算机、服务器和互联网来进行二进制数据传输这种方式日益普及，数据完整性验证技术也得到了改进。调制解调器速度提高了，人们就更需要一种更好的方法来检测数字传输中是否发生错误。数据完整性不仅仅是一个点对点问题，而且还是涉及有线和无线网络以及数据向多个端点传输的主要问题。

嵌入式系统会产生类似的数据完整性问题，其中有许多可选方法来确保数据完整性。一种简单的方法是发送数据，然后让接收方将数据返回给发送方进行比较，以此来验证数据。其他系统可能采用对数据取反以进行比较或使用数据校验和的方法。还有一种更复杂的方法，即使用基于多项式的循环冗余校验 (CRC)。在上述方法中，每一种都具有不同的集成难度和不同的数据完整性结果。

表 1-1. 数据完整性方法

方法	优势	劣势
对数据取反	可以轻松快速地进行计算	传输长度加倍； 无纠错功能
奇偶校验	可以轻松快速地进行计算； 检测短数据包中的单个位错误	许多组合都会导致相同的奇偶校验值，因此容易出错； 仅对小数据包有用； 无纠错功能
校验和	可以轻松快速地进行计算； 检测短数据包中的单个位错误和某些多位错误	许多组合都会导致相同的校验和值，因此容易出错； 仅对小数据包有用； 无纠错功能
CRC	会产生相同 CRC 值的位翻转组合更少，因此可以进行更全面的分析	需要冗长的计算时间或可以使用 LUT，但会增加存储器使用量； 无纠错功能
汉明码	可以增加对多位错误的检测； 可以实现单个位纠错	需要冗长的计算时间

并非所有模数转换器 (ADC) 都具有用于确定通信错误的内置硬件方法。对于这些类型的器件，应通过读取来验证写入器件的任何内容。对于转换数据，应多次读取结果并比较接收到的结果。ADC 器件数据表将指明是否有可用于数据完整性检查的硬件方法，以及器件是检查传入和传出数据的完整性还是仅检查传出数据的完整性。在为特定应用选择 ADC 时，应考虑系统在受 ADC 支持的数据完整性方法方面的关键性。

校验和以及 CRC 是 ADC 常用的硬件方法，用于将编码值附加到传输的数据消息的末尾。附加的信息用于确定传输中是否发生错误。对于使用校验和的器件（例如 ADS1259），转换数据后跟一个校验和字节。校验和是 24 位转换结果中的三个数据字节和常数 9Bh 之和。该常数对数据应用了一个偏移量，从而降低同一个校验和多次出现的可能性。在将这些字节相加时，计算中会忽略任何溢出。将计算出的校验和结果与传输的校验和进行比较，如果计算出的校验和等于校验和字节，则验证通过。

当采用 CRC 检测数据完整性时（以 ADS124S08 为例），接收方必须在完成传输后验证数据，计算所传输数据的消息部分并与传输的 CRC 值进行比较。如果计算出的 CRC 值与传输的 CRC 值不匹配，则说明传输出现错误。这样的比较就类似于使用校验和，但计算方式存在很大不同。与校验和相比，CRC 可实现更高精度的数据验证，但 CRC 使用多项式除法而不是校验和的简单加法，因此涉及更多的处理工作。

除 CRC 之外，ADS122U04 和 ADS122C04 系列提供的一些硬件方法也支持不太复杂的数据完整性验证方法。这些器件中采用的功能是通过将数据传输两次来实现的。第一次传输的是未取反的数据，第二次传输的是按位取反值的原始数据。如果通信成功，则在对两次传输的数据执行异或运算后，每个位都为“1”（没有“0”）。

本应用手册的一个重点内容是介绍如何使用校验和及 CRC，包括使用“C”固件的嵌入式处理器所涉及的计算和函数。也可以使用其他计算方法，但这需在处理器内使用内部硬件外设。不过，处理器的硬件 CRC 外设可能与 ADC 使用的多项式实现不完全相同，具体取决于所用多项式。

大多数信息和示例也可在修改后用于其他 ADC，例如 ADS124S08、ADS1261 和 ADS1262 系列器件。此外，本应用手册还讨论了使用汉明码进行数据完整性检测，而且 ADS131A04 系列器件支持这一功能。ADS131A04 器件经配置可用于关键应用，对该器件而言可在传入和传出通信中验证数据完整性。汉明码选项支持单个位检错和纠错。汉明字节内有一个简单的 2 位校验和，有助于判定一些多位错误。除汉明码/校验和之外，还可将 CRC 添加到传输中以发现大多数多位错误。

表 1-2. 器件和数据完整性方法示例列表

器件	接口	数据方向	使用的验证数据完整性的方法	多项式	数据完整性计算
ADS122C04	I^2C	MSB 优先	对数据取反；CRC-16-CCITT	$x^{16} + x^{12} + x^5 + 1$；初始值 FFFFh	寄存器读取、24 位转换结果和数据计数器字节（启用时）
ADS122U04	UART	LSB 优先	对数据取反；CRC-16-CCITT	$x^{16} + x^{12} + x^5 + 1$；初始值 FFFFh	寄存器读取、24 位转换结果和数据计数器字节（启用时）
ADS124S08	SPI	MSB 优先	CRC-8-ATM (HEC)	$x^8 + x^2 + x + 1$；初始值 00h	24 位转换结果和状态字节（启用时）
ADS131A04	SPI	MSB 优先	CRC-16-CCITT 和/或汉明码	$x^{16} + x^{12} + x^5 + 1$；初始值 FFFFh	传入和传出传输；CRC 和汉明码可同时启用
ADS1259	SPI	MSB 优先	校验和	不适用	24 位转换数据加上 9Bh 的偏移量
ADS1261	SPI	MSB 优先	CRC-8-ATM (HEC)	$x^8 + x^2 + x + 1$；初始值 FFh	传入命令；24 位转换结果和状态字节（启用时）
ADS1262	SPI	MSB 优先	校验和或 CRC-8-ATM (HEC)	$x^8 + x^2 + x + 1$；初始值 00h	32 位转换数据加上 9Bh 偏移量的校验和或仅 32 位转换数据（使用 CRC 时）

2 简单校验和

在 ADS1259 和 ADS1262 等器件上可将校验和附加至转换结果。校验和支持对单个位错误和多位错误的某些组合进行检错。计算校验和字节的方法是将每个转换数据字节的值相加，并加上一个常数。对于上述器件，该常数为 9Bh。校验和的长度是一个字节，因此运用加法而产生的任何进位都将被忽略。对于 24 位器件 ADS1259，将 MSB、中间字节和 LSB 与该常数相加。校验和结果被附加至到转换数据并与其一起进行传输。

校验和易于计算，所需的处理工作量很少。不过，如果存在多位错误，则可能无法检测到错误，因为将不同数据的多种组合相加会产生相同的校验和值。例如，将转换数据 12h、34h、56h 和常数 9Bh 相加会产生校验和 37h。但数据在传输过程中可能会损坏。如果接收到的数据是 12h、35h、55h 和常数 9Bh，则将这些字节相加会产生相同的校验和 37h。

校验和在检查小数据包中的 1 至 2 位错误时会很有用。对于小数据包，可能会错过数据中的错误，但随着数据包大小的增加，错过多位错误的可能性也会增加。单独使用校验和并不是确定数据完整性的最佳方法。

2.1 校验和代码示例

以下代码示例可用于具有校验和数据完整性功能的器件，例如 ADS1259 和 ADS1262。可以通过发送指向数据的指针以及要考虑的数据包的长度来计算校验和。数据字节与常量一起相加以创建校验和字节。然后将计算出的校验和字节与随数据传输的校验和进行比较。字节值应匹配，否则会出错。

```
/**
 * 计算一系列数据字节的校验和。
 *
 * \详细信息：计算指定长度的
 * 数据数组指向的一系列字节。还加上常数 0x9B。
 *
 * \参数 uint8_t *data 是指向数据数组的指针。
 * \参数 uint32_t length 是数据数组的长度。
 *
 * \返回 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];  // 将转换数据字节相加
        result += 0x9B;         // 加上常数
        return (uint8_t) result;
}
```

3 CRC

CRC 是一个基于多项式的系统，对于给定的多项式，该系统使用的位数和检测错误的有效性会有所不同。校验和不同于 CRC 的地方在于校验和使用加法，而 CRC 使用除法。用数据除以所需多项式，然后将所得的余数放在数据之后进行传输。汉明距离 (HD) 指出了使用任何特定多项式的有效性，对于 CRC 讨论，汉明距离指的是可能发生未检测到的位错误的边界。HD 表示无法检测到的位错误的最小数量。检测到所有错误时的位数比 HD 少 1。HD 会相对于数据的长度而变化，通常随数据位长度的增加而减少。

作为 CRC 讨论的一部分，汉明权重 (HW) 是指对于给定数量的位错误，某个多项式无法检测到的位错误的数量。理想情况下的 HW 为零，当 HW 对于给定的位数不为零时，如果发生错误，则并非所有的位错误组合都会被检测到。HD 与第一个大于 0 的 HW 相关。例如，如果某个多项式的 HD 为 4，则会检测到 1、2 和 3 位错误（也被称为位反转或翻转）的所有组合。如果对于数据长度为 48 位或更多的 4 位错误，HW 为 11，则对数据包而言有 11 种无法检测到的 4 位错误组合。

根据多项式和数据传输中的总位数，对于高达特定的数据位数，HD 可能会增加。在上述示例中，对于小于 48 位的数据长度，HD 可能会从 4 增加到 5。通过独立研究，可将许多关于各种多项式 HD 的数据收集在一起。此处讨论的重点是消除对于各种多项式有效性的一些困惑，并表明在使用 CRC 时仍可能出现未检测到的错误。各种 ADC 使用不同的多项式，并且在大多数情况下固定为单个多项式。因此，对于使用哪个多项式几乎没有选择。用于 ADC 的多项式可能不是在给定应用中使用的最佳多项式，而是根据对特定多项式的熟悉度或其通用性而选择的。另一个原因可能是各种微控制器上硬件实现的可用性。许多微控制器使用 8 位（字节）通信模式，因此使用面向 8 位或 8 位倍数的 CRC 多项式变得很有用。此处讨论的重点并不在于使用哪个多项式，而是用于器件的实际多项式，例如 ADS124S08 系列器件（8 位 CRC）以及 ADS122x04 系列器件（16 位 CRC）。

由于 CRC 的编码和解码相似，因此可以轻松比较余数的位模式。如果位匹配，则传输很可能对给定的 HD 有效。不过，如果位不匹配，则说明发生了错误。从软件实现的角度而言，用数据字除以多项式，得到一个余数。从 ADC 硬件的角度而言，这些器件利用与输出移位寄存器相关的某种形式的逻辑来计算 CRC 值，无论传输的数据长度如何。应该以与初始 CRC 计算相同的位顺序分析数据，这一点很重要。因此需要十分注意字节序（请参阅表 3-1）和字节顺序。

3.1 CRC 通用计算

CRC 可用于传输长度达几百位、几千位甚至更长的数据。在大多数情况下，针对嵌入式应用讨论的通用 CRC 代码基于 32 位或更短的短数据传输长度。数据是面向字节的，以字节的倍数进行传输，最长为 4 个字节（对于 32 位传输）。如果最大数据传输长度是 8 位的倍数，则可使用指向存储器阵列的指针来完成 CRC 计算。

CRC 是通过用数据除以多项式来实现的，因此讨论的初始部分将说明使用长除法确定余数的过程。传输的数据值被所使用的多项式除。当数据是被除数时，在这种情况下使用的初始值为零。这与将 0h 和数据执行异或运算等效，结果与数据相等。完成的长除法（其中除数为多项式，被除数为数据）的余数成为计算出的 CRC 值。因此用数据（被除数）除以多项式（除数），计算出的结果为商。计算得到的余数是 CRC 值。

当初始值为零且数据有前导零时，CRC 计算将忽略计算中的前导零。为了使 CRC 计算更加稳妥以包括任何前导零，使用了零以外的初始值。最常见的非零初始值是针对所使用的 CRC 大小的全 1。8 位 CRC 的非零初始值为 FFh，16 位 CRC 的非零初始值为 FFFFh。ADC 器件数据表中指定了计算的初始值。然后将初始值和数据一起进行异或运算以得到被除数。

ADC 实现的多项式通常是标准 CRC 多项式，例如 CRC-8-ATM 或 CRC-16-CCITT。这些多项式已被证明对数据传输完整性检查是有效的。通常，这些标准 CRC 算法之一是在嵌入式微控制器中实现的，可加快计算速度。ADS122C04 使用 16 位多项式 CRC-16-CCITT，它已广泛用于通信数据传输。该多项式是 $x^{16} + x^{12} + x^5 + 1$，等效于 10001000000100001b。生成的 CRC 结果是用数据除以多项式的长除法得到的余数。除了提及两个值的减法变成异或运算之外，此处的讨论不详细介绍模 2 数学运算的过程。

ADS122C04 的初始 16 位起始值在数据表中作为 FFFFh 给出。图 3-1 中的示例使用长度为 24 位的数据值，这类似于转换结果。该示例使用值 4E6878h 作为数据。

由于最终结果将是 16 位的余数，初始值附加零，这样总长度将由数据长度加上 16 位余数构成。

1. 初始值 = 0xFFFF（根据 ADS122C04 数据表）
2. 在初始值右侧填充 24 个“0”，从而使初始值的总长度为转换数据的长度加上 CRC 的长度（一共 40 位）

3. 将左对齐的初始值与转换后的值进行异或运算：FFFF000000h XOR 4E68780000h = 0xB197780000（请参阅图 3-1）
4. 将步骤 2 的结果除以多项式：B197780000h ÷ 11021h，然后取余数 B72Ch 作为计算出的 CRC 值

除数除被除数的过程会得到余数 B72Ch（如图 3-2 所示），这将是 ADS122C04 在发送转换数据后发送的 CRC 值。

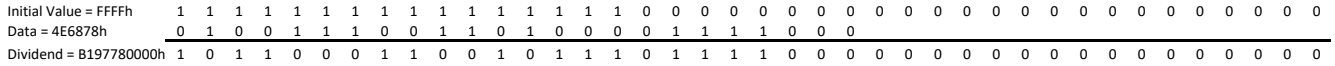

图 3-1. 通过将初始值和数据执行异或运算来计算 CRC 计算的起始值示例

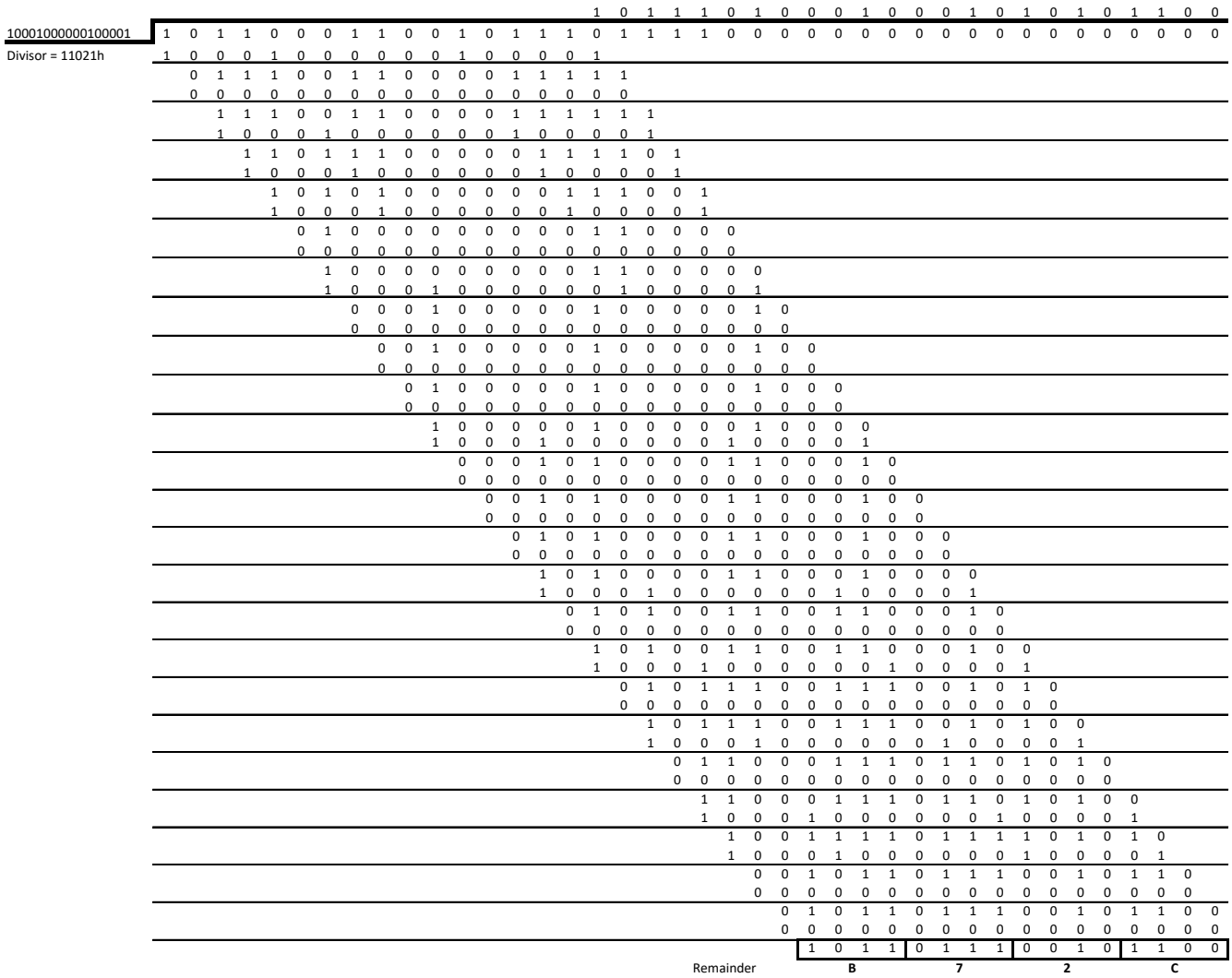

图 3-2. 两个具有二进制系数的多项式的二进制长除法

对于串行传输数据的 ADC 器件，硬件实现通常通过移位寄存器来执行，此时随着数据的移出计算出 CRC。可能需要对数据进行反转或反射以进行 CRC 计算，具体取决于接口类型。数据反射只是传输位顺序的交换。在节 3.1.3 中比较 ADS122C04 (I^2C) 和 ADS122U04 (UART) 器件的 CRC 计算时，更详细地讨论了该顺序为何如此重要。数据传输的区别在于 UART 是 LSB 优先，而 I^2C 是 MSB 优先。在数据传输到存储器时对 UART 接收缓冲区的读取变得复杂起来。在读取接收缓冲区时，位的顺序相对于每个字节的传输顺序进行了反转或反射。在计算 CRC 时，必须使用传输时位的正确顺序。

微控制器还可以使用硬件实现来计算 CRC。除了 CRC 硬件实现之外，嵌入式处理器代码也可以完成与长除法所执行的任务相同的任务，方法是使用类似的异或过程并循环遍历每个位和字节，然后将 CRC 作为余数返回。异或

过程在函数内作为按位运算过程需要耗费大量的时间。另一种常用的方法是使用查找表来处理这些值。以下内容分别对按位异或和查找表这两种方法进行了讨论。

3.1.1 使用按位异或计算

软件按位运算函数忽略多项式值中的最高位。在对余数进行异或运算之前会将最高位移出，因此可以忽略该位。例如，CRC-16-CCITT 的多项式为 $x^{16} + x^{12} + x^5 + 1$ (11021h)，由 10001000000100001b 表示。不过，最高位被丢弃，按位运算中用于计算的值变为 1021h。

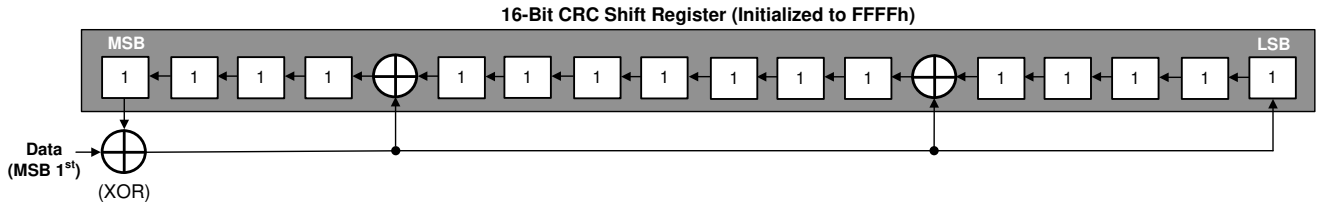

图 3-3. CRC-16-CCITT 硬件实现

对于 8 位 CRC-8-ATM (HEC)，多项式是 $X^8 + X^2 + X + 1$ (107h)，由 100000111b 表示。同样地，最高位被忽略，用于按位运算的值为 7h。

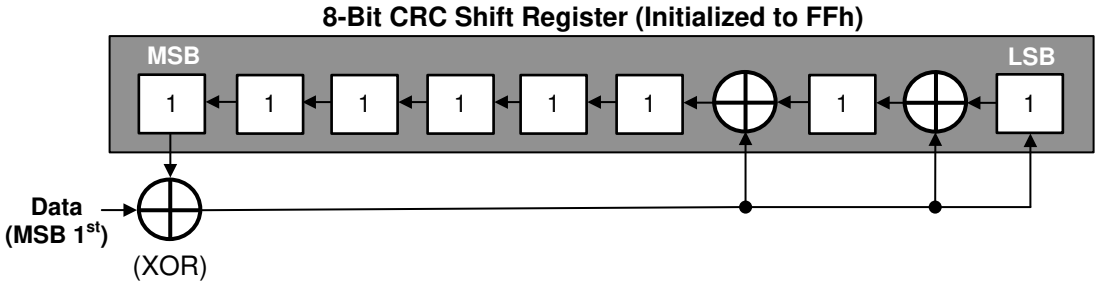

图 3-4. CRC-8-ATM (HEC) 硬件实现

以下代码函数适用于 CRC-16-CCITT，但可以轻松适应其他多项式以及长度。该函数复制硬件实现并根据 CRC 长度的大小返回一个值，该值由 *crc_t* 的类型定义表示。在该代码示例中，返回值是一个无符号的 16 位值。传递的参数是指向待计算*数据*包的指针和数据包的*长度*（字节数）。该函数通过与余数执行按位异或运算来处理每个数据字节。*REMAINDER_INIT* 的初始值是起始种子值，会因使用的 ADC 而异。

```
typedef uint16_t crc_t;           // 对于 8 位 CRC，使用 uint8_t
#define POLYNOMIAL 0x1021        // CRC16-CCITT，但对于 8 位 CRC，使用 CRC-8-ATM (HEC)
                                  // 和多项式值 0x07
#define REMAINDER_INIT 0xFFFF     // 对于 ADS1xC04 和 ADS1x2U04 上的 16 位 CRC，使用 0xFFFF
                                  // 对于 ADS1260、ADS1261 和 ADS1235 上的 8 位 CRC，使用 0xFF
                                  // 对于 ADS124S0x、ADS114S0x、ADS1262 和 ADS1263 上的 8 位 CRC，使用
0x00
#define WIDTH (8 * sizeof(crc_t))
/**
 * 计算一系列数据字节的 CRC。
 *
 * \详细信息：通过与所需的多项式进行异或运算来计算一系列字节的 CRC。
 *
 * \参数 uint8_t *data 是指向数据数组的指针。
 * \参数 uint32_t length 是数据数组的长度。
 *
 * \返回 crc_t remainder。
 */
crc_t  crcBitwise(uint8_t *data, uint32_t length)
{
    crc_t remainder = REMAINDER_INIT;
    uint32_t byte, bit;
                                                  // 对数据包中的每个字节
                                                  //     执行多项式长除法
    for(byte = 0; byte < length; byte++)
    {
        remainder ^= (data[byte] << (WIDTH - 8));     // 将下一个字节取到余数中
```

```
        for(bit = 8; bit > 0; bit--)                    // 对于余数中的每个字节
        {
            if(remainder & (1 << (WIDTH - 1)))
                remainder = (remainder << 1) ^ POLYNOMIAL; // 如果最高位置 1，则将余数
                                                        //    左移并将其与除数异或，
            else                                        //    然后将结果存储
                                                        //    在余数中
                remainder = (remainder << 1);           // 如果最高位清零，则左移
                                                        //    余数
        }
    }
    return remainder;
}
```

该代码示例对于确定数据的 CRC 余数很有用。对于具有传入和传出 CRC 数据完整性检查的器件，需要计算出余数。不过，如果仅针对 ADC 的传输验证数据（例如转换数据），则可将传输的 CRC 值添加到数据数组中。如果计算中包含 CRC 余数，并且 CRC 与计算值以及传输值都匹配，则余数将为零。换句话说，CRC 计算有一个重要的特性，即在匹配字节中移位会强制将 CRC 移位寄存器中的值设置为零。非零结果将指示传输错误。使用该方法则无需对传输的 CRC 和计算的 CRC 进行直接比较，从而简化验证。

3.1.2 使用查找表

查找表法需要一个数据表，用于存储所有传入数据字节的所有可能的余数组合。该表存储在 RAM 或闪存中，以便快速访问。在计算每个表值时，对于表条目的计算使用类似于按位异或的方法。表条目的数量是 256，表示一个字节内的所有位组合。使用查找表法的优点是其速度约为按位运算法的四倍，并且与校验和运算所需的时间相当，代价是要为 16 位条目存储 512 个字节（为 8 位条目存储 256 个字节）。

3.1.2.1 表初始化

用于存储表数组的存储器大小将取决于待返回的 CRC 值的大小。为了提高效率，表边界应设置为字节对齐。该表包含 256 个 16 位条目（对于 16 位 CRC）或 256 个 8 位条目（对于 8 位 CRC）。

```
typedef crc_t uint16_t;          // 对于 8 位 CRC，使用 uint8_t
#define POLYNOMIAL 0x1021        // CRC16-CCITT，但对于 8 位 CRC，使用 CRC-8-ATM (HEC)
                                 //     和多项式值 0x07
#define REMAINDER_INIT 0xFFFF    // CRC16-CCITT，但对于 8 位 CRC，使用 CRC-8-ATM(HEC)，
                                 // 对于 ADS1260、ADS1261 和 ADS1235 上的 8 位 CRC，使用 0xFF
                                 // 对于 ADS124S0x、ADS114S0x、ADS1262 和 ADS1263 上的 8 位 CRC，使用
0x00
#define WIDTH (8 * sizeof(crc_t))
crc_t crcTable[256];
/**
 * 初始化要存储在存储器中的 CRC 查找表。
 *
 * \详细信息：对单个字节中包含的每个可能组合进行 CRC 计算，
 * 然后将其存储到 crcTable 数组中，其中每个数组元素与一个
 * 特定字节值相关。
 *
 *
 * \返回 void。
 */
void  initCRCtable(void)
{
    crc_t remainder;
    uint32_t byte, bit;
                                                    // 对数据包中的每个字节
                                                    //    执行多项式长除法
    for(byte = 0; byte < 256; byte++)
    {
        remainder = (byte << (WIDTH - 8));          // 将下一个字节取到余数中
        for(bit = 8; bit > 0; bit--)                // 对于余数中的每个字节
        {
            if(remainder & (1 << (WIDTH - 1)))
                remainder = (remainder << 1) ^ POLYNOMIAL; // 如果最高位置 1，则将余数
                                                    //    左移并将其与除数异或，
            else                                    //    然后将结果存储
                                                    //    在余数中
                remainder = (remainder << 1);       // 如果最高位清零，则左移
                                                    //    余数
        }
        crcTable[byte] = remainder;
```

```
    }
}
```

3.1.2.2 CRC 计算

从先前经过初始化并存储在存储器中的 CRC 表中检索字节值，而不是使用多个异或运算。

```
/**
 * 基于先前存储在存储器中的查找表进行 CRC 计算。
 *
 * \详细信息：通过对表数据（这些数据是通过对多项式进行按位异或运算而得到并存储在存储器中的）
 * 进行异或运算来计算出 CRC 值。
 *
 * \参数 uint8_t *data 是指向数据数组的指针。
 * \参数 uint32_t length 是数据数组的长度。
 *
 * \返回 CRC 计算得到的 crc_t remainder。
 */
crc_t tableCRCcalc(uint8_t *data, uint32_t length)
{
    crc_t remainder = REMAINDER_INIT;
    uint32_t byte, u32i;

    for(u32i = 0; u32i < length; u32i++)                    // 对于数据包中的每个字节
    {
        byte = data[u32i] ^ (remainder >> (WIDTH - 8));
        remainder = crcTable[byte] ^ (remainder << 8);      // 针对字节余数
                                                            //     执行表查找
    }
    return remainder;
}
```

3.1.3 ADS122U04 和 ADS122C04 之间的 CRC 计算差异

正如概述中提到的，CRC 值是相对于数据传输的位顺序计算出的，因为它被移出器件。与“C”器件相比，“U”器件的数据传输有很大不同。“U”器件使用 UART 传输，数据以 LSB 优先的方式进行传输，而“C”器件使用 I^2C 传输，数据以 MSB 优先的方式进行传输。如图 3-5 所示，在采用 MSB 优先的传输方式时，数据被读入微控制器外设并以与数据传输相同的方式重新组合（在使用大端字节序格式时）。

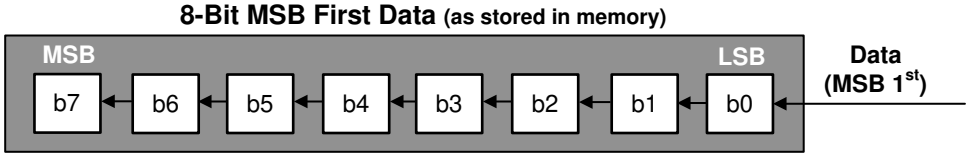

图 3-5. MSB 优先方式中用于移动数据的存储器的内容

“U”器件的 CRC 计算遵循与图 3-3 中所示过程相同的过程。不过，“U”器件的不同之处在于移位寄存器首先从 LSB（而不是 MSB）开始。结果直接影响 CRC 计算和数据存储到存储器中的字节顺序。

当通过代码函数计算 CRC 时，了解数据存储的方法用处极大。例如，考虑一个由字节（8 位）可寻址位置构成的存储器。一个 32 位整数由四个相邻的字节组成。将该整数视为一个包含四个字节的数组。如果存储在数组第一个元素中的值是该整数的最高有效字节 (MSB)，则该值以大端字节序的方式进行存储。如果最低有效字节 (LSB) 存储为数组的第一个元素，则该整数以小端字节序的方式进行存储。表 3-1 所示为有关 32 位值如何存储在存储器中的对比情况。当数据以 MSB 优先的方式进行传输时，传输顺序遵循大端字节序格式。相反，当数据以 LSB 优先的方式进行传输时，传输顺序遵循小端字节序格式。对于此处讨论的内容，微控制器将使用大端字节序格式。不过，在这两种方法中，字节数组元素的最高有效位都是第 7 位，最低有效位都是第 0 位。

表 3-1. 32 位数据的存储器寻址

32 位数据	存储器地址	大端字节序	小端字节序
0A0B0C0Dh	0h	0Ah (MSB)	0Dh (LSB)
	1h	0Bh	0Ch
	2h	0Ch	0Bh
	3h	0Dh (LSB)	0Ah (MSB)

CRC 的计算仅基于数据传输位和字节的顺序。不过，LSB 优先的数据被传输到微控制器 UART 外设，在此处，接口将以 LSB 优先的方式移动数据，但存储器中的内容顺序将与其传输的顺序相反。传输的数据与存储器中的数据顺序相反，因此微控制器计算出的 CRC 值在位和字节顺序上都不正确。对于以 LSB 优先的方式传输的数据，必须对字节顺序和每个字节内的位顺序进行反转或反射才能进行计算（请参阅图 3-6）。

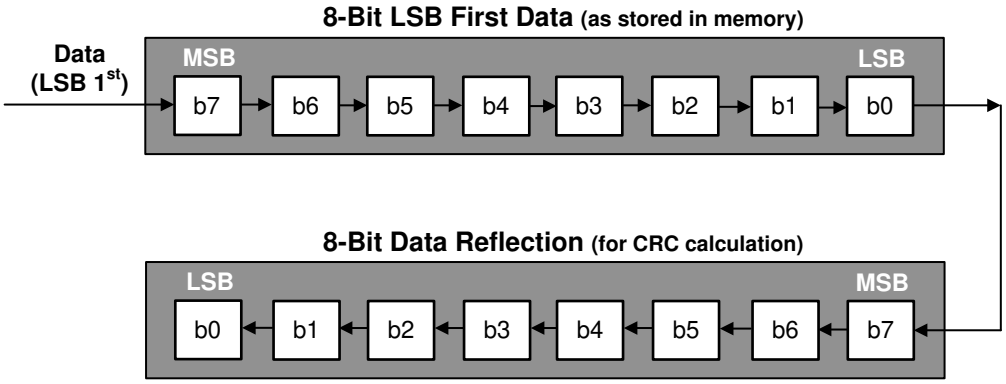

图 3-6. LSB 优先方式中用于移动数据的存储器的内容，需要反射才能进行 CRC 计算

如果微控制器存储器采用大端字节序方式并且接收到的 UART 转换数据存储为有符号的 32 位整数，那么仅仅反转表示整数值的字节的顺序是不够的。部分原因是原始数据是 24 位（16 位器件为 16 位），而整数值进行了符号扩展。此外，大多数微控制器 UART 外设会在时钟脉冲下以 LSB 优先的方式将数据输入到 8 位移位寄存器中，但会以相反的方向将字节存储在存储器中。外设将在存储器中重新对齐位顺序，这是位传输顺序的相反顺序或是其的反射。不过，在计算数据包的 CRC 时，位的顺序和数量必须与传输的位的顺序和数量相匹配。例如，如果将 24 位转换结果存储为有符号的 32 位值，则在执行 CRC 计算时，必须使用已进行正确反射的原始二进制补码 3 字节数据，而不是已进行符号扩展的 4 字节有符号数。

作为所需步骤的示例，下面介绍了如何通过一个有符号 32 位整数数组 (sData) 创建一个表示 24 位转换数据的数组 (cData)。新数组 (cData) 用于计算 CRC，以便与从 ADS122U04 传输的 CRC 进行比较。

1. 将 sData 数组复制到 cData 数组中，其中 cData 数组表示小端字节序格式，如图 3-7 所示。
2. 转换数据为 24 位，因此请勿在 cData 数组中使用进行了符号扩展的字节。
3. cData 数组现在包含正确的字节顺序，但其位顺序不正确。
4. 对于 cData 数组中的每个字节元素，反射位顺序，使位顺序与最初的传输顺序相同，如表 3-2 所示。
5. 计算 cData 数组中前三个元素的 CRC。

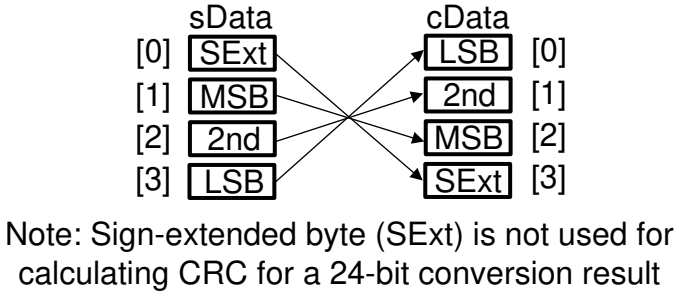

图 3-7. 更改字节顺序以反映 LSB 优先

表 3-2. 使用 LSB 优先方式将有符号 32 位整数转换为 24 位整数后进行反射，以用于 CRC 计算

数组元素	sData (32 位)	cData (24 位)	反射后的 cData
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 字节反射示例

以下代码示例是一种反转或反射一个字节内的位顺序的方法。顺序 b^7、b^6、b^5、b^4、b^3、b^2、b^1、b^0 被反射为返回位顺序 b^0、b^1、b^2、b^3、b^4、b^5、b^6、b^7。

```
/**
 * 反转字节的位顺序。
 *
 * \详细信息：以提交的字节的相反顺序重新排列位，
 * 并将字节作为无符号字节返回。必须执行该操作，因为在比较 TX 和 RX 数据时，
 * 总是按照 lsb 优先的顺序从 UART 向外传输数据。
 *
 * \参数 uint8_t u8Byte 是要进行反转的数据字节。
 *
 * \返回 uint8_t u8RevByte。
 */
uint8_t revByte(uint8_t u8Byte)
{
    uint8_t u8i, u8RevByte = 0;         // 从位 0 开始对于字节中的每个位
    for (u8i = 0; u8i < 8; u8i++)
    {
        u8RevByte |= (u8Byte & 0x01);   // 取输入字节的一位并将其存储
                                        //     在新字节中
        u8Byte >>= 1;                   // 获取下一位
        if (u8i < 7) u8RevByte <<= 1;   // 不要移动新字节的最后一位
    }
    return u8RevByte;
}
```

3.1.3.2 使用字节反射重新组合数据以进行 CRC 计算

可以通过多种方式重新组合数据，同时实现相同的结果。一种方法是使用指向数据数组的指针，然后按照经过反射的位和字节顺序将内容存储到另一个数组中。

```
/**
 * 重新配置数据以进行 CRC 计算（与 LSB 优先传输方式进行对比）。
 *
 * \详细信息：以提交的字节的相反顺序重新排列位，
 * 并为 CRC 更改计算的字节顺序。必须执行该操作，因为总是按照
 * LSB 优先的顺序从 UART 向外传输数据。
 *
 * \参数 uint8_t *data 是指向数据数组的指针。
 * \参数 uint32_t length 是数据数组的长度。
 *
 * \返回计算得到的 crc_t CRCvalue。
 */
crc_t formatCRCdata(uint8_t *data, uint32_t length)
{
    uint32_t u32i, u32j;
    uint8_t checkCRC[length];
    crc_t dataCRC;
    // 将 LSB 作为 CRC 数组中的第一个元素开始复制
    u32j = length - 1;
    for(u32i = 0; u32i < length; u32i++)          // 对于数据包中的每个字节
    {
        checkCRC[u32i] = revByte((data[u32j]));   // 反转字节
        u32j--;
    }
    dataCRC = crcBitwise(checkCRC, length);       // 或 TableCRCcalc(checkCRC, length)
    return dataCRC;
}
```

4 汉明码

对汉明码 (HC) 的讨论仅限于如何使用 HC 对 ADC 进行数据完整性检查。HC 用于纠正传输的数据块中的某个位的错误，适用于验证小数据包，但无法检测所有多位错误，并且无法纠正多位错误。ADS131A0x 器件适用的汉明字节方案包括 HC 以及校验和位，可用于确定多位错误。

对于 ADS131A0x 器件，HC 由五位 (H0-H4) 组成，这些位与数据交错，如图 4-1 所示。奇偶校验用于确定 HC 中每个位相对于映射到数据的特定位的值。从概念上讲，HC 位在数据位内交错。结果是多个汉明位覆盖了每个数据位。这是对早期数据通信方式的改进，早期的数据通信仅使用单个奇偶校验进行数据验证。现在，HC 使用多个位来查找和纠正单个位错误。简单的奇偶校验计算涉及对数据包中 1 的数量进行计数，其中奇偶校验位将为 1 或 0，具体取决于是要进行奇校验还是偶校验。奇校验与奇数个 1 相关，偶校验与偶数个 1 相关。在 ADS131A0x 器件中，汉明位的设置实现了偶校验。

ADS131A0x 系列器件的数据字可以是 16 位或 24 位。汉明字节由五个汉明位组成，其中 H4 为最高有效位。在汉明位之后是两个校验和位。汉明字节的最低有效位是一个等于零的常数。数据字和汉明字节的总传输可以是 24 位或 32 位。在本文中，我们设定 32 位字长包括 24 位数据和汉明字节。

五个汉明位成为了数据字的一部分，并与每个位的奇偶校验表示进行比较，使汉明奇偶校验的组合至少覆盖数据位两次，这样一来，如果某个位不正确，则可以通过其余的汉明奇偶校验位进行重建。为了检查多个位是否有错误，使用 2 位校验和数据来验证结果是否显示单个位或多位错误。

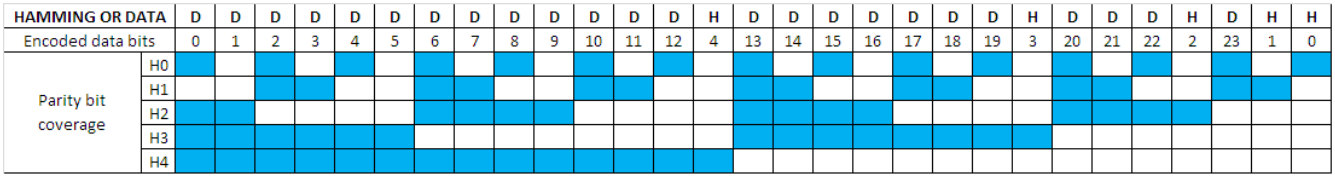

图 4-1. 计算出的 ADS131A0x 汉明码

图 4-1 中的表数据显示了每个汉明位的奇偶校验位覆盖情况，其中汉明位在数据中交错。表中的每一行指示每个汉明位中包含哪些数据位。请注意，在该表显示的位中，表左侧是最低有效位，右侧是最高有效位。检查的位在列中指示，行表示汉明奇偶校验位。该表中的信息可用于确定每个汉明位的汉明掩码。汉明掩码用于验证与掩码关联的汉明位。该掩码仅包含用于特定汉明位的数据位。*在将表转换为计算中使用的汉明掩码时，从左到右读取 32 位值，数据将排除掩码中的汉明位。*

与 ADS131A0x 器件之间的数据传输顺序是从数据的最高有效位 (msb) 到最低有效位 (lsb)。传输数据后，汉明位将跟随在字节中。如图 4-1 所示，汉明位未在输出中交错。相反，汉明位作为单独的字节附加到器件数据的传输中。汉明位值仍将遵循相同的模式和计算方式，但按照设计将汉明字节放置在数据之后。通过将汉明字节设置为传输的最低有效字节 (LSB)，用户可以在需要时轻松忽略该字节，或者在禁用时完全不传输该字节。这样一来，无论是否启用汉明码，器件数据都能够以相同的方式显示。汉明字节还包含数据的校验和，该校验和是一个 2 位值（C1 和 C0），表示设置为“1”的位的数量。最低有效位 (F0) 是一个值为“0”的固定常数。传输的数据和汉明字节具有如图 4-2 所示的格式。

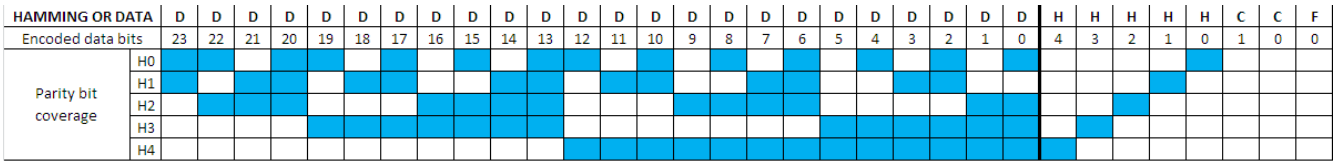

图 4-2. 传输的 ADS131A0x 数据汉明码

在计算汉明位奇偶校验时，每个汉明位将遵循表 4-1 中所示的模式，其中每个位都可以通过屏蔽掉未计算的位的方式来确定。使用这种方法极大地简化了数据传输验证的计算。

表 4-1. 汉明位掩码

汉明位	掩码值（32 位）
H0	00DAB555h

表 4-1. 汉明位掩码 (continued)

汉明位	掩码值（32 位）
H1	00B66CCCh
H2	0071E3C3h
H3	000FE03Fh
H4	00001FFFh

在计算“1”位的数量并与适当的汉明奇偶校验位进行比较时，多次将掩码应用于数据。在奇偶校验计算中没有汉明位包含任何其他汉明位，因此允许对这些位进行掩码。此外，每个数据位都由多个汉明位进行检查。可能会发生某个汉明位错误，但如果在其他汉明位检查中验证了数据位，则会指示该错误。如果某个数据位出错，则在奇偶校验中会有多个汉明位指示该错误。在将奇偶计算与汉明位进行比较时，差异将直接指向出错的数据位。

为了演示该计算，下面提供了一个使用值 AC9538h 的 24 位数据示例。每个汉明位的结果如表表 4-2 所示。计算过程如下：

1. H0 计算：对数据应用 H0 的掩码 (AC9538h & 00DAB55h = 889510h)
2. 求第 1 步中结果的奇偶校验位（889510h 有 7 个 1，所以奇偶性是奇，H0 变为 1 使总奇偶性为偶）
3. 对 H1 至 H4 汉明位重复执行步骤 1 和 2

表 4-2. 汉明位计算示例

汉明位	数据（十六进制）	掩码（十六进制）	结果（十六进制）	测试（二进制）	位数	汉明值
H0	AC 95 38	DA B5 55	88 95 10	1000 1000 1001 0101 0001 0000	7（奇数）	1
H1	AC 95 38	B6 6C CC	A4 04 08	1010 0100 0000 0100 0000 1000	5（奇数）	1
H2	AC 95 38	71 E3 C3	20 81 00	0010 0000 1000 0001 0000 0000	3（奇数）	1
H3	AC 95 38	0F E0 3F	0C80 38	0000 1100 1000 0000 0011 1000	6（偶数）	0
H4	AC 95 38	00 1F FF	00 15 38	0000 0000 0001 0101 0011 1000	6（偶数）	0

校验和位的计算基于数据二进制表示中“1”值的数量。继续使用十六进制数据 AC 95 38h，将该十六进制值转换为二进制，并计算代表该值的“1”的数量。二进制转换结果为“1010 1100 1001 0101 0011 1000b”，出现的“1”的数量为十进制 11。值 11 用二进制表示为“1011b”。通过将全部 24 个数据位相加并按 4 取模来计算校验和。为了简化计算并避免按 4 取模数学运算，仅使用两个最低有效位作为校验和位。在本例中，校验和仅为“11b”。

整个汉明/校验和字节将采用图 4-2 所示的格式，即 H4、H3、H2、H1、H0、C1、C0、0。对于使用 0xAC9538 的示例，汉明/校验和字节的位二进制表示变为“0011 1110b”。

4.1 汉明码计算

汉明位被视为与数据交错，但实际传输并未交错。不过，数据始终以相同的方式传输。如前所述，可使用简单的位掩码来评估和计算汉明位，而不是使用对数据进行移位和异或运算的方法。可以通过两种方法来对位进行计数。一种方法是遍历掩码数据并在每次出现“1”时计数一次。这种方法与 CRC 按位运算法类似，相当耗时。第二种方法使用表来直接评估数据，该方法能够以更快的速度确定掩码数据中“1”的数量。

在针对每个汉明位计算“1”的数量后，将所得值与传输数据的相应汉明位进行异或运算。正确的结果为零。如果结果不为零，则通过评估来确定哪个位出错。该过程的特别之处在于能够纠正单个位的错误。出错的位可能是汉明奇偶校验位，也可能是数据中的某个位。每个数据位都由多个汉明奇偶校验位覆盖，因此可以 针对单个位进行纠错。不过，存在一些可能导致多位错误的组合或者与数据相关的汉明位组合，其中数据可能看起来正确，但实际上并不正确。与单独分析汉明位相比，与汉明位一起使用校验和位是一种更全面的解决方案。

为了确定是否存在传输错误，会计算输入数据以确定预期的汉明位。然后使用异或运算将计算出的汉明位以及校验和与接收到的汉明字节进行比较。如果发现错误（计算返回的值大于零），则在发现错误后使用纠错表对位进

行纠错。将对 ADS131A0x 器件使用节 4.1.2.3 中显示的纠错表。该表适用于为数据位和汉明位传输的完整 SPI 字。不包括 3 个最低有效位（即校验和位和常量零）。在进行位纠错之后，通过计算恢复数据中包含的“1”位的数量来计算数据的校验和。根据计算出的校验和，通过异或运算将两个最低有效位与汉明字节中传输的校验和位进行比较。如果校验和失败，则说明数据中存在多位错误。

4.1.1 汉明码计算示例

汉明码计算法既可用于计算待传输的数据，也可用于验证接收的数据。汉明位掩码应用于五个汉明位中的每一个位的数据。此外，还会计算校验和并将其附加到结果中。返回的值是以无符号 32 位整数表示的 24 位数据和汉明字节。

```
#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
/**
 * 计算输入的 24 位数据的汉明位。
 *
 * \详细信息：通过为每个汉明位使用位掩码，然后计算掩码值中
 * 置 1 的位的数量，来计算汉明位。   该操作执行 5 次，
 * 针对每个汉明位执行一次。此外，会计算校验和，返回附加到 24 位
 * 数据的完整字节。用于计算的“in”值假定
 * 传输的数据不包含汉明字节。计算完成后，
 * 将汉明字节添加到输入值中，这是通过将 24 位值
 * 左移 8 位来完成的。
 *
 * \参数 uint32_t in 是要计算的值。
 *
 * \返回 uint32_t out，其中包含附加了汉明字节值的 24 位数据。
 */
uint32_t calcHamming(uint32_t in)
{
    // 获取传递的整数值并将其转换为在传回时将包含
    //    汉明字节的格式
    uint32_t out = in << 8;
    // 通过获取输入值，与掩码值进行与运算，将结果
    //   与 0x01 进行与运算，然后计算位的数量，来计算
    //   5 个汉明位。如果数量为奇数，则将汉明位置 1，使数量为偶数。
    // 如果数量为偶数，则不将汉明位置 1。
    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) ;
    // 将数据左移 8 位，与左移 3 位后的汉明位进行或运算，其结果与数据的校验和
    //   与 2 个位 (0x03) 进行与运算后左移 1 位的结果进行或运算，LSB 为 0，
    //   这样就得到返回值，其中共有 5 个汉明位、2 个校验和位和一个“0”位，
    //   总共 8 位。
    return    (out | (hamming << 3) | ((countBits(in) & 0x03) << 1));
}
```

4.1.1.1 计算位数以进行奇偶校验和校验和计算

可通过两种方法来确定所考虑的值中“1”位的数量。一种方法是简单地逐位检查数据，另一种更快速的方法是使用查找表。

4.1.1.1.1 计算数据中置 1 的位的数量示例

一种计算数据中置 1 的位的数量的方法是以一次评估一个位的方式来评估每个位。以下方法需要对数据进行 32 遍评估，以评估 32 个位中的每一个位。

```
/**
 * 计算 32 位值中置 1 的位的数量。
 *
 * \详细信息：通过使用循环，评估每个最低有效位，
 * 然后将剩余值右移 1 位，来计算置 1 的位的数量。
 *
 * \参数 uint32_t in 是要计算的值。
 *
```

```
 * \返回计算得到的 uint32_t numBits。
 */
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 使用查找表计算置 1 的位的数量示例

计算数据中置 1 的位的数量的更快方法是使用查找表。以下示例使用 4 位查找表中的数据。这需要进行八遍评估来评估 32 个位。更大的查找表可以减少评估遍数，但会增加存储器中表的大小。

```
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
};
/**
 * 使用查找表计算 32 位值中置 1 的位的数量。
 *
 * \详细信息：通过使用循环，评估查找表中的每个半字节，
 * 然后将剩余值右移 4 位，来计算置 1 的位的数量。
 *
 * \参数 uint32_t in 是要计算的值。
 *
 * \返回计算得到的 uint32_t numBits。
 */
uint32_t countBits (uint32_t in)
{
    uint32_t numBits = 0;
    while (in != 0)
    {
        numBits += bitsArray[in & 0x0F];
        in >>= 4;
    }
    return numBits;
}
```

4.1.2 验证传输的数据

接收到数据后，可以根据接收到的数据计算汉明位。为了进行验证，将接收到的汉明位与计算出的汉明位进行比较。为了进行完整的比较，还会计算校验和位并将其与接收到的校验和进行比较。

4.1.2.1 汉明验证

汉明位是通过以下方式计算得出的：获取接收到的数据并计算汉明位，然后通过按位异或的方式与接收到的汉明位进行比较。

```
/**
 * 通过重新计算并与传输的数据执行异或运算以进行比较来验证汉明/数据位。
 *
 * \详细信息：通过为每个汉明位使用位掩码，然后计算掩码值中
 * 置 1 的位的数量，来计算汉明位。   该操作执行 5 次，
 * 针对每个汉明位执行一次。此外，会计算校验和，返回附加到 24 位
 * 数据的完整字节。用于计算的"in"值假定
```

```
 * 传输的数据不包含汉明字节。计算完成后，
 * 将汉明字节添加到输入值中，这是通过将 24 位值
 * 左移 8 位来完成的。
 *
 * \参数 uint32_t in 是要计算和比较的值。
 *
 * \返回 uint32_t ham，如果与汉明位/校验和的计算结果相匹配，则返回 0。
 */
uint32_t checkHamming(uint32_t in)
{
    // 验证汉明位时，取输入数据并计算汉明值，然后减去
    //    汉明字节
    // 所以取"in"并右移 8 位，然后计算汉明值
    uint32_t ham = calcHamming((in & 0xFFFFFF00) >> 8);
    // 计算出的汉明字节与从数据返回的汉明字节进行异或运算，
    //    在理想情况下结果对于汉明码加上校验和为 0，因为汉明值
    //    是完整的 32 位字，汉明字节是从数据中计算出来的，然后
    //    与实际传输的汉明字节进行比较
    uint32_t res = ham ^ in;
    if(res == 0) return res;
    else return (ham & 0x000000FF);
}
```

4.1.2.2 校验和验证

根据接收到的数据计算出校验和，然后使用按位异或运算将其与传输的校验和值进行比较。如果通过异或计算出非零值，则表示发生了校验和错误。用于 ADS131A0x 的校验和与节 2.1 中讨论的用于 ADS1259 的计算不同。

```
/**
 * 计算以及验证汉明字节中的校验和。
 *
 * \详细信息：通过使用校验和位掩码并计算掩码值中
 * 置 1 的位的数量来计算校验和位。
 *
 * \参数 uint32_t in 是要计算和比较的值。
 *
 * \返回值 uint32_t 针对匹配返回 0，针对失败返回非零值。
 */
uint32_t checkSum(uint32_t in)
{
    // 在去除汉明/校验和字节后计算数据的校验和。
    // 通过异或的方式将计算出的校验和与汉明/校验和字节进行比较，
    // 如果校验和正确，则结果应为"0"。   必须在将校验和向左移动 1 位
    // 以将校验和放置在正确的位置之后再进行比较。   此外，
    // 通过与 0x06（将 0x03 左移 1 位得到的值）进行与运算来删除比较中不属于
    // 校验和的所有其他位
    return (in ^ ((countBits(in & CHECKSUM_BIT_MASK) << 1))) & 0x06;
}
```

4.1.2.3 误差校正

使用汉明和校验和验证后，可以纠正单个位错误。首先进行数据验证和纠错，如果可能，则纠正位。如果恢复了数据，则重新评估校验和。如果出现多位错误或校验和错误，则传输无效。

```
uint32_t fixResults[] = {
    0,         // 数据块正常
    1 << 3,    // 数据中没有错误 (H0)
    1 << 4,    // 数据中没有错误 (H1)
    1u << 31, // 数据[23] 错误
    1 << 5,    // 数据中没有错误 (H2)
    1 << 30,   // 数据[22] 错误
    1 << 29,   // 数据[21] 错误
    1 << 28,   // 数据[20] 错误
    1 << 6,    // 数据中没有错误 (H3)
    1 << 27,   // 数据[19] 错误
    1 << 26,   // 数据[18] 错误
    1 << 25,   // 数据[17] 错误
    1 << 24,   // 数据[16] 错误
    1 << 23,   // 数据[15] 错误
    1 << 22,   // 数据[14] 错误
    1 << 21,   // 数据[13] 错误
    1 << 7,    // 数据中没有错误 (H4)
    1 << 20,   // 数据[12] 错误
    1 << 19,   // 数据[11] 错误
```

```
        1 << 18,  // 数据[10] 错误
        1 << 17,  // 数据[9] 错误
        1 << 16,  // 数据[8] 错误
        1 << 15,  // 数据[7] 错误
        1 << 14,  // 数据[6] 错误
        1 << 13,  // 数据[5] 错误
        1 << 12,  // 数据[4] 错误
        1 << 11,  // 数据[3] 错误
        1 << 10,  // 数据[2] 错误
        1 << 9,   // 数据[1] 错误
        1 << 8,   // 数据[0] 错误
                  // 不可纠正的 30
                  // 不可纠正的 31
                  // H0 覆盖 14 个位，H1-H4 覆盖 13 个位，
                  //   因此如果汉明码为 30 或 31，
                  //   则会出现多位错误且无法纠正
};
/**
 * 数据验证和单个位错误修复。
 *
 * \详细信息：使用汉明位信息验证和
 * 恢复数据以纠正单个位错误。   首先检查“in”值，
 * 如果异或运算返回 0，则相对于汉明位验证数据内容，
 * 然后使用校验和检查数据内容是否存在多位错误。
 * 如果汉明位与数据不匹配，则在可能的情况下
 * 使用汉明位来纠正数据，方法是使用计算出的
 * 汉明位与传输的汉明位进行异或。   使用基于校正系数和异或值的查找表
 * 来恢复数据，然后计算校验和以确保没有
 * 多位错误。
 *
 * \参数 uint32_t in 是要计算和比较的值。
 *
 * \返回 uint32_t res，如果数据可纠正或有效，则返回 0。
 *                          非零值是无效数据。
 */
uint32_t fixHamming(uint32_t in)
{
    uint32_t res;
    uint32_t fix;
    // checkHamming 将从数据“in”返回 5 个汉明位
    // 在条件语句中使用赋值时，条件会对赋值进行
    //    评估。
    // 如果 checkHamming 返回“0”以外的值，则尝试修复结果
    if(res = checkHamming(in))
    {
        // 根据非“0”的汉明码结果，通过调整来自包含
        // 2^5（或 32）种可能组合的表中“in”的 32 位值，
        // 使用查找表来纠正结果。总之，32 个值中有 24 个值用于 1 位纠错，
        // 5 个值表示数据无错误（汉明码中的 1 位错误），1 个值表示
        // 数据和汉明码均无错误，2 个值表示数据、汉明码
        // 或两者中存在多位错误。对于多位错误的情况只分配了
        // 两个值，因此算法很可能错误地
        // 将多位错误视为 1 位错误。为了降低这种
        // 可能性，除了汉明检测之外，还添加了 2 位校验和
        // 以进行检错。

        // 快速检查汉明值中是否存在多位错误
        fix = res >> 3;
        // 然后将计算出的汉明位与“in”值汉明位进行异或运算
        fix = fix ^ ((in & 0xFF) >> 3);
        // 如果存在已知的多位错误，则指示修复失败
        if (fix > 29) return HC_FIX_FAIL;
        else
        {
            // 发生了位错误，尝试进行修复
            fix = (fixResults[fix]);
            // 识别的错误可能是汉明错误或数据错误，在此处将“in”值
            //   与 fix 值进行异或
            res = in ^ fix;
        }
    }
    else
    {
        res = in;
    }
    // res 现在包含数据的原始值或修复值，即经过
    // 汉明校验和数据纠正后的数据 + 汉明值，
    // 使用校验和位检查多位错误，checkSum 将
    // 在计数中排除汉明位。如果 checkSum 返回 0 以外的值，则
```

```
    // checkSum 失败，否则返回 0
    if(checkSum(res))
    {
        return HC_FIX_FAIL;
    }
    return res;
}
```

5 总结

随着任务关键型系统日益普及，安全问题越来越受到关注，因此对数据传输进行验证成为当务之急。分析数据完整性的方式方法可谓是多种多样，各有千秋，所以，为应用选择一种合适的方法并非易事。不过，了解本文讨论的方法对于决定所需的数据完整性级别和选择合适的 ADC 大有裨益。

6 参考文献

- 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*
- 德州仪器 (TI)，*ADS122C04 数据表*
- 德州仪器 (TI)，*ADS122U04 数据表*
- 德州仪器 (TI)，*ADS131A04 数据表*
- 德州仪器 (TI)，*ADS124S08 数据表*
- 德州仪器 (TI)，*ADS1235 数据表*
- 德州仪器 (TI)，*ADS1259 数据表*
- 德州仪器 (TI)，*ADS1260 数据表*
- 德州仪器 (TI)，*ADS1262 数据表*

7 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from Revision * (June 2020) to Revision A (August 2021)	Page
• 更新了整个文档中的表格、图和交叉参考的编号格式。	2

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265