1 SerDes 接收端数据链路层设计

1.1 8B/10B 解码器设计

根据 JESD204B 协议规定，SerDes 接口为保证信内数据的直流平衡，并且为了便于时钟恢复，采用 8B/10B 码作为数据链路层的编码方式。JESD204B 中采用的 8B/10B 编解码部分主要参考 IEEE802.3 以太网协议中关于 8B/10B 的编解码部分。但是由于应用环境的不同，JESD204B 协议的 8B/10B 编解码方式同以太网相比略有不同。

1.1.1 协议分析

以太网协议提供了完整的编码表，也可以看作为解码表，这几张表格的正确性毋庸置疑，是最值得参考的资料。表中将总共 268 种的编码情况全部列出，可以作为校验编码正确性的基准值。

在数字编码中有一个重要的概念，叫做极性[1]。极性是指一组编码中 0 的个数和 1 的个数之差，在一定长度的编码情况下，可以理解为 0 或 1 的个数。比如说 3B/4B 编码中，在 4B 编码结果中，若 1 的个数多余 0 的个数则称为正极性，反之称为负极性，二者相同则称为平衡。事实上，在具体的数据字编码中，不会出现编码结果 0 的个数多余 1 的个数三个或以上. 若编码中 1 的个数减去 0 的个数记，无论 6B 或者 4B 的编码结果只会为 -2,0,2，相对应与负极性、平衡和正极性。

以太网协议建议，将一个码组的极性分为三部分，第一是上一码组计算后得出的极性，第二是编码后 6B 部分的极性，第三是编码后 4B 部分的极性。而最后得到的 4B 部分的极性将作为这一组码的极性用于接下来码组的计算。极性运算的基本结构：$last\_code\_group\_RD \rightarrow 6B\_sub\_block \rightarrow 6B\_RD \rightarrow 4B\_sub\_block \rightarrow 4B\_RD(new\_last\_code\_group\_RD)$。

每个 sub-block 的判断可用以下伪代码表示（6B 和 4B 略有不同）：

if 000111 or 0011 or $1s > 0s$
    $6B\_RD = +; RD\_4B = +;$
else if 111000 or 1100 or $1s < 0s$
    $6B\_RD = -; RD\_4B = -;$
else
    $6B\_RD = last\_code\_group\_RD;$
    $4B\_RD = 6B\_RD;$
endif

最后是以太网协议中关于极性错误的处理，在协议附录中给出了一些接收当中的极性错误。可以发现，极性错误是不能精确定位的，它的检测主要是通过接收机本地的极性和所接收到的极性不符所产生的错误。但由于一系列的中性码并不会改变极性，前一码接收产生的错误可能因为一系列的中性码而直到几个码字后才能检测到。

在 JESD204B 协议中关于 8B/10B 编解码的规定，阅读接口协议的数据链路层内容，可以发现，在编解码器之前还有一级控制，主要是用来针对数据成帧结构中的 lane、帧、多帧的校准、同步和错误控制，而控制的依据就是解码器中获得的控制字。

在 JESD204B 协议中只用到了 8B/10B 所用控制字的 5 个，这将简化控制字的解码复杂度，控制字分别如下：

K.29.0 即 D，表示多帧的开始。

K.28.3 即 A，表示 Lane 校准，一般在多帧最后出现。

K.28.4 即 Q，表示 Link 设置数据的开始，在他之后跟一系列设置数据，配置 Link，他也是 ILAS[2]的组成部分。

K.28.7 即 K，表示 Group 同步，可以说是链接开头最重要的部分，用来保持同步，是 CGS[3]的重要控制字。

K.28.7 即 F，表示帧校准，一般表示一帧结束。

[1] RD，即 Running Disparity
[2] Initial Lane Alignment Sequence
[3] Code Group Synchronization

JESD204B 协议还规定了三种重要的解码错误，这些在解码器级别的错误是属于不太严重的错误，有可能经常发生，在错误并不严重的情况下并不需要进行重同步，但是需要上报给错误处理部分，供应用层决定如何处理。

错误包括以下三种：

Not-in-table Error 这种错误意味着接收到的码字在任何极性情况下都不存在于码表中，就是一些非法的码字。对于这些码字，协议规定接收端需要重复之前收到的最新的没有错误的帧。

Running Disparity Error 这种错误就是上文提到的极性错误，协议规定解码器要根据收到的数据和极性直接解码。由于在检测到极性错误时，可能产生错误的不是这个码字，这样的处理方式也比较合理。

Unexpected Control Character 这种错误就是指未出现在指定位置的控制字，这一错误的具体处理需要由 lane/帧监测部分来决定，属于接下来层级的处理，解码器并无法判断出这一错误。

1.1.2 解码思路

具体逻辑框图如图 1所示。

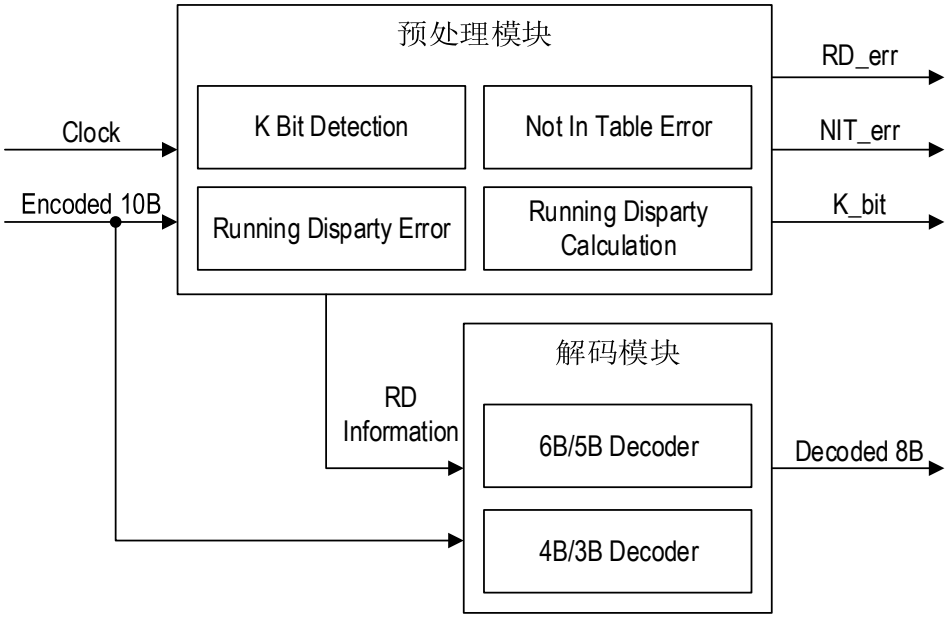

图 1: *8B/10B* 解码器系统框图

通过对现有论文编解码方式的理解分析，可以发现编码的方式比较多样，但解码仍主要停留在通过逻辑的方式。在 [6]、[5] 和 [3] 中采用的是纯逻辑解码方式；在 [4] 和 [2] 这两篇文章中采用的是多路选择器的方式。

在现有文章的解码中，极性仅作为差错检测，这是一种对信号资源的浪费，如果通过极性来进行解码，可以更好的利用现有的码表。对于已知码组的极性信息，可以得出下一组码可能的编码情况，这时就可以通过取反操作来压缩解码表的大小，更快的处理解码操作。例如，已知前一组码字的极性信息为 $RD+$，则可以推测出接下来的 $6B$ 数据的极性信息只有可能为 $RD-$ 或者均衡两种可能。因此只需要处理一种极性的码字就可以完成对整个编码的解码，复杂度变为原来的一半。

对于数据字而言，在获知当前 RD 状态的前提下，解码就分为两种情况。一种是相反的极性，还有一种是均衡的极性。对于相反的极性而言，不需要对另一种极性解码，整张解码用表就可以缩小一半，一方面节约了芯片面积额，一方面提高了解码效率。对于均衡极性而言，由 [3] 中编码原理分析可知，对于均衡码的解码其实非常简便，只需要输出其低 5 或 3 位。因此，只需要设计一个均衡码判断电路，就可以快速选择是否通过解码逻辑极性解码。

对于控制字而言，由 [2] 中提到的控制信息检测可以发现，通过 RD 和固定位置的比特就可以区分该码字是否为控制字，并且确定是哪一类控制字 ($K.28.x$ 还是 $K.23.7$、$K.27.7$、$K.29.7$、$K.30.7$)。在分析编码可知，控制字的 $3B$

或 5B 部分的编码规则同数据字是相同的。那么就可以“借用”数据字的解码部分来对控制字部分解码，准确输出控制码字。其中包括了均衡和非均衡的情况，处理逻辑同数据字，唯一不同的就是控制字状态标记是否拉高。

最后对于解码模块的考虑，一般情况下都是采用的是 verilog 语言的 case 逻辑，但是对于高效的电路来说要尽量避免对语言原生性能的依赖。通过逻辑化简的方法对更小的码表进行化简，这样的得到表达式速度更快，并且面积较小。

1.1.3 第一级，预处理级

预处理级系统框图如图 2所示。

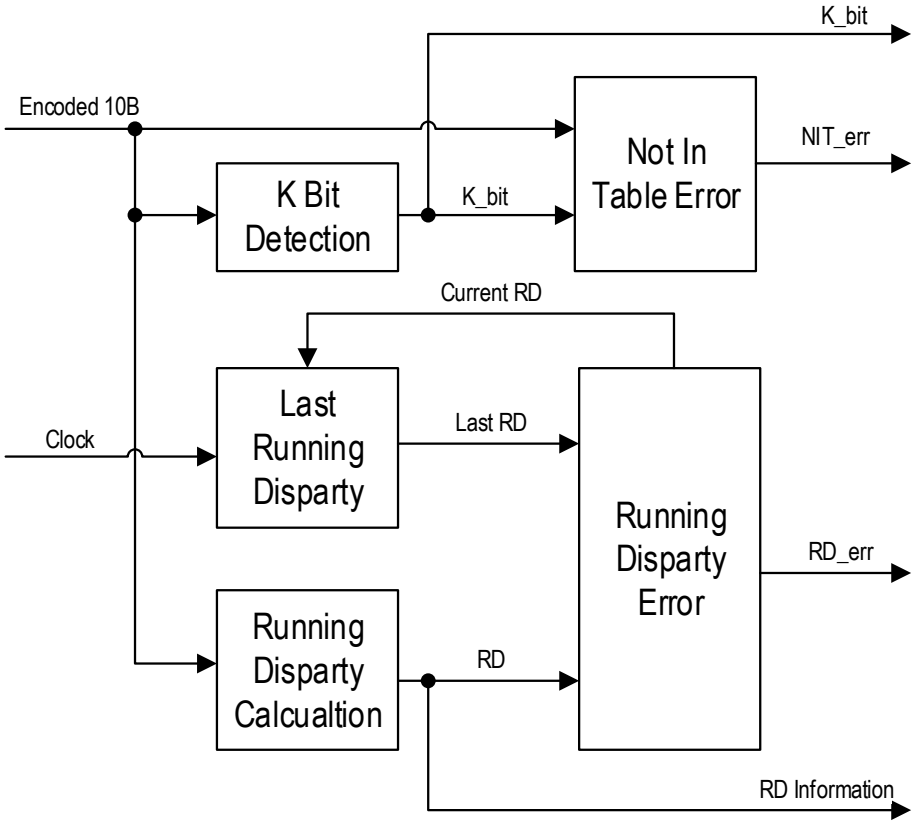

图 2: *8B/10B* 解码器预处理级框图

K Bit 检测 根据输入完整的 10B 数据判断是否为控制信号。有以下伪代码：

if $iedc == 1111|0000$
$K.28.x = 1; K.x.7 = 0;$
else if $jhgfie == 010111|101000$
$K.28.x = 0; K.x.7 = 1;$
else
$K.28.x = 0; K.x.7 = 0;$
endif

本设计中的 8B/10B 编码，只采用了 6B 部分为 K.28 的控制字。由完整检测逻辑可以得出，只需要检测 6B 部分的 cdei 位是否为 0000 或者 1111，就能判断出结果。所以判断控制字本设计可以采用最简单的逻辑表达式：

$$K\_bit = (c\&d\&e\&i) \quad | \quad (c|d|e|i)$$

6B 和 4B 平衡检测 就是对给出的并行数据，分别输出 6B 和 4B 是否平衡，平衡即指 0 和 1 的数量是否相同。有以下伪代码[4]：

if $RL(6B) == 3$
    $Balance\_6B = 1;$
else
    $Balance\_6B = 0;$
endif

if $RL(4B) == 2$
    $Balance\_4B = 1;$
else
    $Balance\_4B = 0;$
endif

RD 信息检测 RD 信息检测主要指分别计算 6B 和 4B 部分的极性，每一部分又分别有两个输出，$RD-$ 和 $RD+$。因为每一组码字的极性一共存在三种可能，即正、负和平衡，不属于负和正的码字，既为平衡，平衡的判断也可由上文中的伪代码确定。这些重要的信息主要用于极性错误检测和解码。通过对合法码字的真值表进行化简，可以得到快速 RD 极性检测模块，准确输出正确码字 RD 的负和正信号。

最终模块输出 4 个信号，即 RD_6B_pos、RD_6B_neg、RD_4B_pos、RD_4B_neg。

NIT 错误检测 Not-In-Table Error 的检测也分为 4B 和 6B 两部分。

6B 部分的错误码字一共为 14 种，如表 1，参考 [3] 一文的方法可以分为两类，既考虑 abcd 全为 0 或全为 1 的情况和 abcd 有且仅有 1 个 1 或仅有 1 个 0 的情况。前者直接可以判断该码字为错误码字，后者再观察 ef 是否全为 0 或全为 1，也可判断码字是否错误。

4B 部分的错误码字相对较为复杂，考虑到 K.28 中只有 5 个控制字是合法的，所以要对余下的码字报错。4B 部分报错情况如表 2所示。

表 1: *6B* 码字错误情况

abcdei	abcdei
000000	111111
000001	111110
000010	111101
000100	111011
001000	110111
010000	101111
100000	011111

表 2: *4B* 码字错误情况

abcdei	hgfj
xxxxxx	0000
	1111
001111	0101
	1001
	0110
110000	1010
	1001
	0110

极性错误检测部分负责检测关于极性的错误，主要就是指不能出现连续相同的极性变化。例如之前一个码字的 RD 为 RD-，则接下来收到的 6B 编码的 RD 必须为均衡或者是 RD+，这样就保证了信道上码字的均衡。极性检测实际上考虑的是三个部分的信息，可以细化为五个信号，包括上一个码字的极性，当前码字 6B 部分的 RD+ 或 RD-，当前码字 4B 部分的 RD+ 或 RD-。极性错误的监测并不能精确定位，因为均衡的码字是无法判断出是否有错的，只有当数据扩散到非均衡位置时才能判断出错误。由于正确的检测需要保存上一码字的极性信息，所以极性错误检测还负责解码器极性的刷新。检测可以通过极性信息计算模块提供的信号进行判断，并将得到新的极性存入寄存器，如表 3所示。

[4]其中 RL 表示游程长度计算，即 1 的个数。下同。

表 3: 极性错误检测及新极性生成表

last	6B+	6B-	4B+	4B-	err	new
0	0	0	0	0	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	0	1	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	1
1	0	0	0	0	0	1
1	0	0	0	1	0	0
1	0	0	1	0	1	1
1	0	1	0	0	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	1
1	1	0	0	0	1	1
1	1	0	0	1	1	0
1	1	0	1	0	1	1

根据此表化简逻辑表达式即可得到准确的极性错误和本码字的极性情况。也可以采用有限状态机的方法进行判断，通过几个固定的状态转换进行极性的判断和存储了 [1]。极性状态转移图如图 3所示。

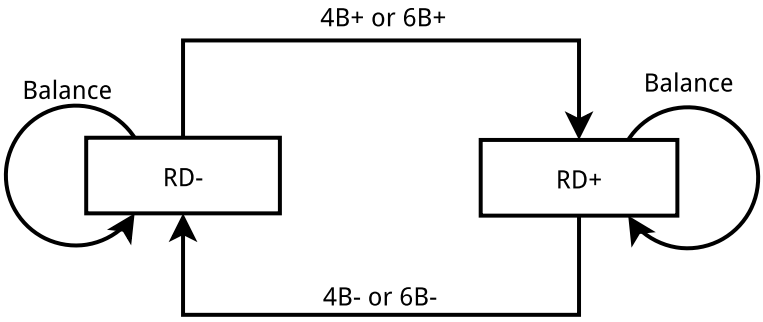

图 3: *8B/10B Running Disparity* 状态转移图

1.1.4 第二级，解码级

解码级系统框图如图 4所示。

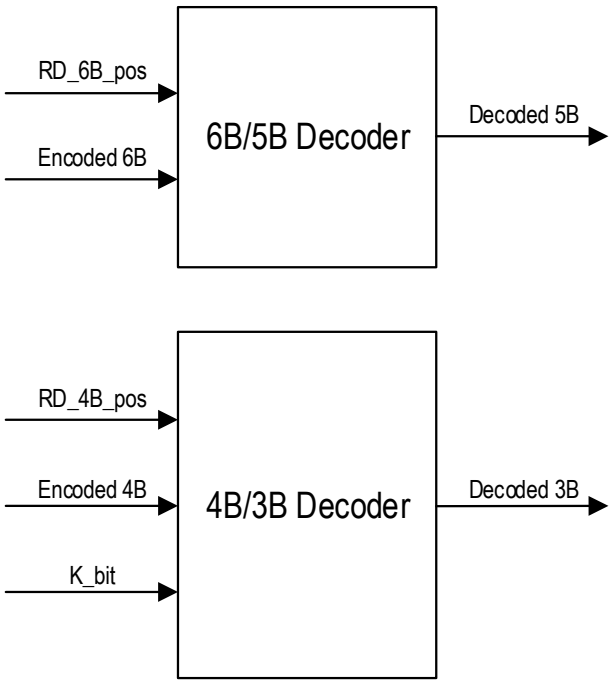

图 4: *8B/10B* 解码器解码级框图

解码输入的第一步就是根据极性情况，对输入码字进行反转。根据 8B/10B 原理，对于相同的原始码字编码，在不同极性情况下的变化就是互为反码。所以在知道具体码字的极性信息时对其进行反转就可以准确的得到同极性下的编码，使之后的解码器只需要处理一种极性。具体的反转判断伪代码如下：

if $RD == RD+$
　　$6B = 6B$;
else
　　$6B =!6B$;
endif

$RD\_6 = RD|Balance\_6B$
if $RD\_6 == RD+$
　　$4B = 4B$;
else
　　$4B =!4B$;
endif

$6B/5B$ 和 $4B/3B$ 解码部分，主要根据解码表对输入码字解码。解码表中存的查找项均为极性为正时的解码，根据输入输出解码结果，5B/6B 解码部分解码表如表 4所示，3B/4B 解码部分解码表如表 5所示。

需要特别注意的是 3B/4B 解码部分，不同于 5B/6B 解码，3B/4B 解码还需要考虑控制字的信息。这是因为对于特定的控制字 K.28 来说，为了避免整个 10B 编码偶然的出现连续 5 个 0 或 1，解码器就要根据情况来选择不同的 3B 编码。在解码过程中要着重考虑到这一点，为此增加了 K 位作为解码输入条件之一，K 位即表示控制位，由预处理级的控制字检测电路产生。通过化简增加信息后的真值表，得到表达式用以解码。

最后需要考虑的是均衡码字的情况，这种情况可以由之前的极性信息直接推断出来，这时候解码器需要做的就是输出相应解码部分的低位。

不同于 [2] 文中所描述的 CASE 方法，也不同于 [3] 中的纯逻辑方法。本设计引入了各个模块的 $RD+$ 信息，在读入数据之前先根据 RD_pos 信息对码字进行反转。这样，在解码时只要考虑 RD 为 $RD-$ 和平衡的情况。对于 6B

解码表由原来的 48 种情况减少为 34 种（如表 4所示）；对于 4B 解码表，由于其解码需要考虑 K Bit 的情况，由原来的 28 种情况减少为 16 种（如表 5所示）。并且由于快速的极性信息计算，使得解码模块能够更快的进行解码，缩短了解码所需时间。最后通过逻辑化简的方法对更小的码表进行化简，这样的得到表达式速度更快，并且面积较小。

表 4: *6B* 解码表

abcdei	EDCBA	abcdei	EDCBA
000011	11100	011001	11001
000101	01111	011010	11010
000110	00000	011100	11100
000111	00111	100001	11110
001001	10000	100010	11101
001010	11111	100011	00011
001011	01011	100100	11011
001100	11000	100101	00101
001101	01101	100110	00110
001110	01110	101000	10111
010001	00001	101001	01001
010010	00010	101010	01010
010011	10011	101100	01100
010100	00100	110001	10001
010101	10101	110010	10010
010110	10110	110100	10100
011000	01000	111000	00111

表 5: *4B* 解码表

K	hgfj	HGF
0	0001	111
0	0010	000
0	0011	011
0	0100	100
0	0101	101
0	0110	110
0	1000	111
0	1001	001
0	1010	010
0	1100	011
1	0001	111
1	0010	000
1	0011	011
1	0100	100
1	1010	101
1	1100	011

1.2 解扰器设计

JESD204 的发送端和接收端设备都需要支持加扰传输的数据流。并且每一对 lane 都要包含加扰器和解扰器。加扰器和解扰器的位置位于数据链路层和传输层之间，也就是位于 8B/10B 解码器之后进行解扰，加扰后进行编码。所以加扰操作主要是针对数据进行的，不涉及链路上的控制字符。对一个 link 启动加解扰意味着在 link 两端启动所有 lane 的加扰器和解扰器。但采用混合模式，只对一个 link 中的固定几条 lane 启动加解扰器是不允许的。

加扰的主要目的是避免频谱出现过大的峰值，这就意味着过多的帧到帧之间重复数据传输。在一个敏感的系统中，频谱的峰值会引起电磁不兼容或者互相干扰的问题。过多相同帧也会引起基于编码的直流偏置。另一个加扰的好处就是使传输信道的频谱和数据区分开，这样接口上可能的频率选择效应就不会影响到数据的独立性。

1.2.1 协议分析

每一种扰码都有相对应的符合实际传输需求的扰码多项式，作为加扰和解扰的依据。JESD204B 协议规定的扰码多项式如式 1所示。

$$1 + x^{14} + x^{15} \tag{1}$$

可见这个加扰多项式的周期长达 32767 位，足以符合敏感电磁系统频谱的要求。并且他允许解扰器在接收到 2 个 octet 数据后达成自同步。

加扰器和解扰器的根据它们的串行实现，一帧一阵的处理发送和接受的数据。一帧中最左边的位首先被转换，具体的转换顺序如图 5所示。

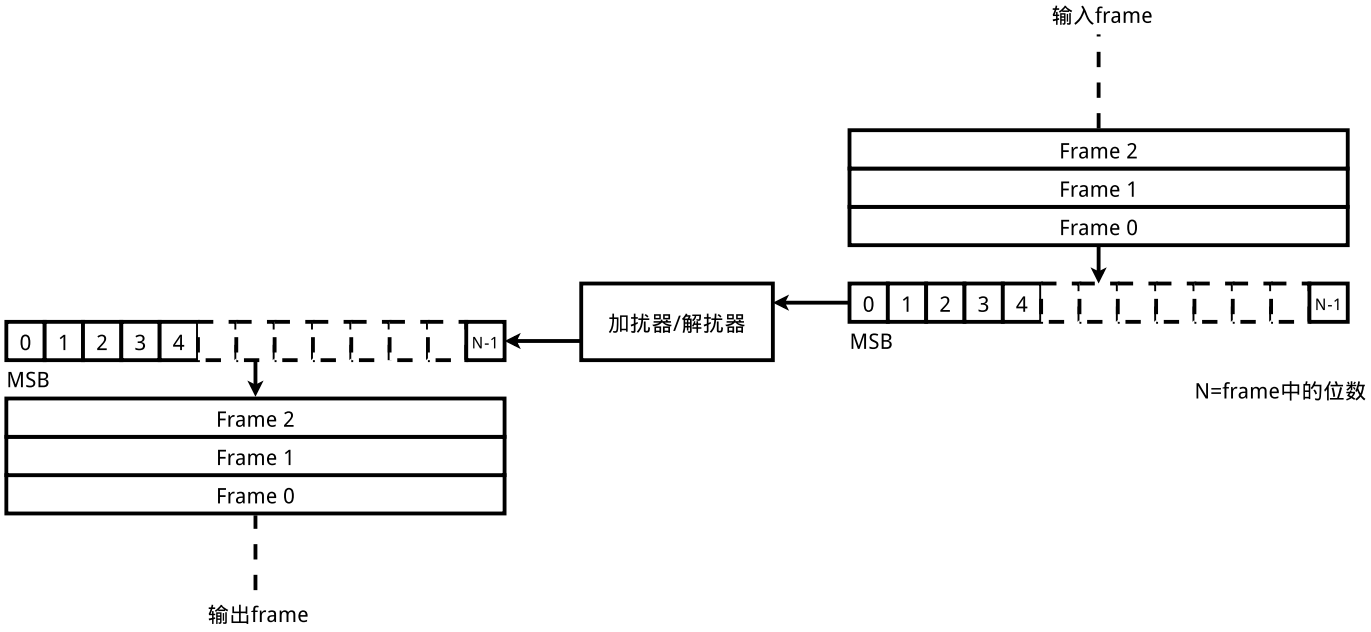

图 5: 扰码帧传输顺序

加扰器和解扰器要能够实现自同步，就是在没有额外初始信号的情况下，最终能够正确输出加扰编码。针对 JESD204B 的协议规定的扰码多项式，可以得到解扰器串行实现的流程框图，如图所示。

加扰器串行更新函数如式 2所示。

$$S_{15} = D_{15} + S_1 + S_0 \tag{2}$$

加扰器并行更新函数如式 3所示。

$$\begin{cases} S_{31} = D_{31} + S_{17} + S_{16} \\ S_{30} = D_{30} + S_{16} + S_{15} \\ \cdots \\ S_{23} = D_{23} + S_9 + S_8 \\ S_{22} = D_{22} + S_8 + S_7 \\ \cdots \\ S_{17} = D_{17} + S_3 + S_2 \\ S_{16} = D_{16} + S_2 + S_1 \end{cases} \tag{3}$$

解扰器串行更新函数如式 4所示。

$$D_{15} = S_{15} + S_1 + S_0 \tag{4}$$

解扰器并行更新函数如式 5所示。

$$\begin{cases} D_{31} = S_{31} + S_{17} + S_{16} \\ D_{30} = S_{30} + S_{16} + S_{15} \\ \cdots \\ D_{23} = S_{23} + S_9 + S_8 \\ D_{22} = S_{22} + S_8 + S_7 \\ \cdots \\ D_{17} = S_{17} + S_3 + S_2 \\ D_{16} = S_{16} + S_2 + S_1 \end{cases} \tag{5}$$

其中，D 表示为未加扰的原始数据位，S 表示加扰状态位即加扰数据输出。

扰码的使能也是协议中重点规定的部分。加扰只是针对用户数据，并不包括控制字信息，例如码群同步控制字符串和初始化帧对齐序列。事实上，在传输中，前两个 octet 并不会全部进行加扰，因为发送端的加扰器也需要这两个 octet 作为初始化信息，才能正确进行之后的扰码输出。但是两个 octet 中的最后一位需要根据第一位、第二位和最后一位自己，加扰得到一个新的加扰位进行传输，接下来的数据将需要进行正规的加扰。在解扰端的状态寄存器跟踪上扰码同步前需要先接收两个 octet 的前 15 位作为初始化信息，之后才能根据第 16 位和之前接收到的第一第二位正确的输出解扰后数据。为了避免在最初的两个 octet 处理时同正常的加扰区分开来，需要加解扰器有使能开关来切换是否进行解扰或者加扰。在接收端，解扰器的输入端连接在 8B/10B 解码器的输出端，有一个使能开关决定数据是进行解扰还是直接通过。

所以，总体的加扰流控制图如图 6所示。

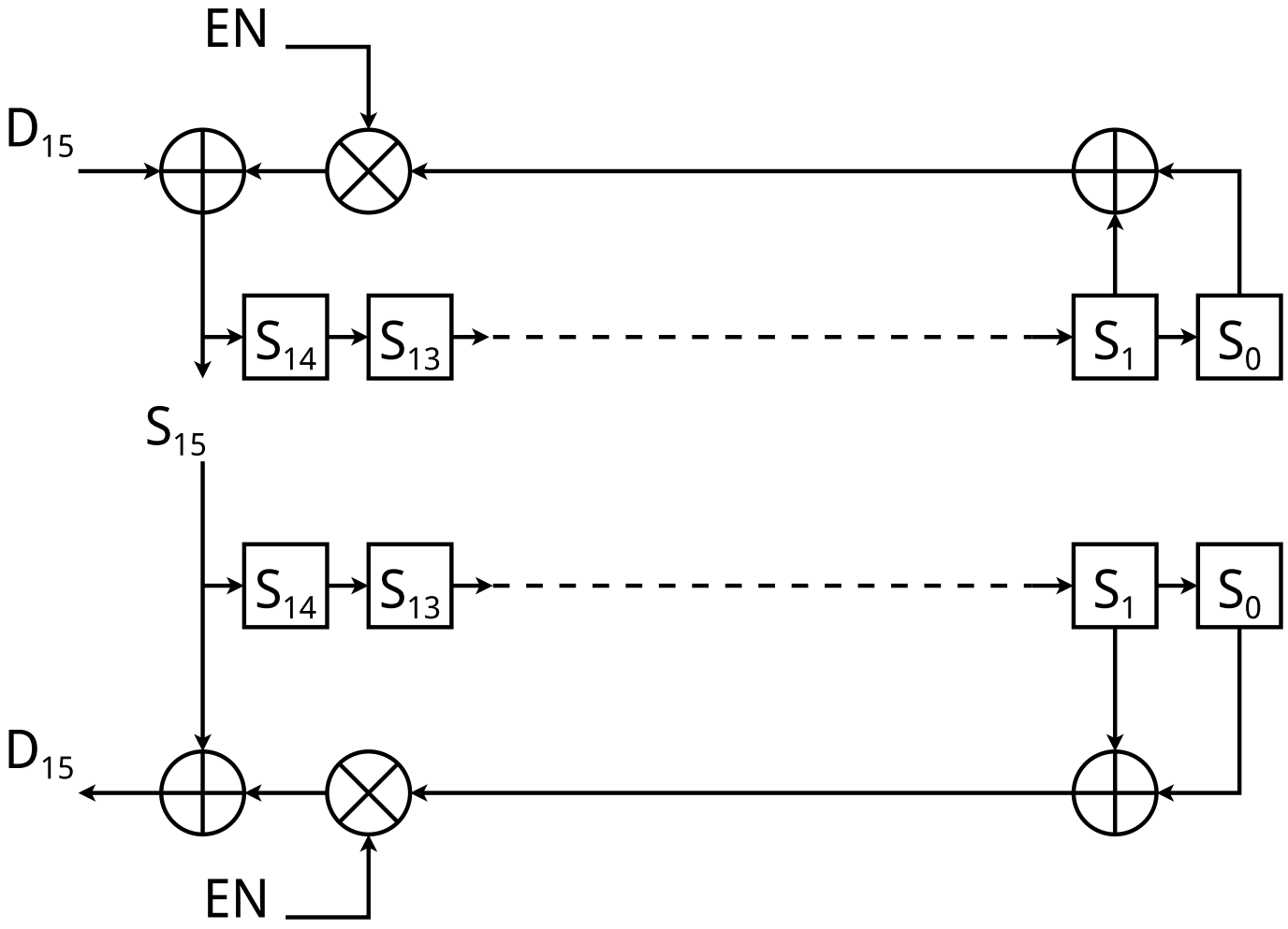

图 6: 解扰器和加扰器实现流程框图

最后需要考虑的是加解扰的初始状态问题。同很多使用自同步的加扰器一样，当输入数据是重复的初始状态值

时，将会产生重复的加扰输出。这就失去了加扰的意义，使信号的波形产生较大的峰值，会导致借口的电磁串扰。为了尽量避免重复输出，加扰器的初始状态必须设置为一个在传输层不太可能产生的 octet。推荐的初始值是高位 8 个 1，余下各位为 0，即 111111110000000。

1.2.2 解扰思路

具体逻辑框图如图 7所示。

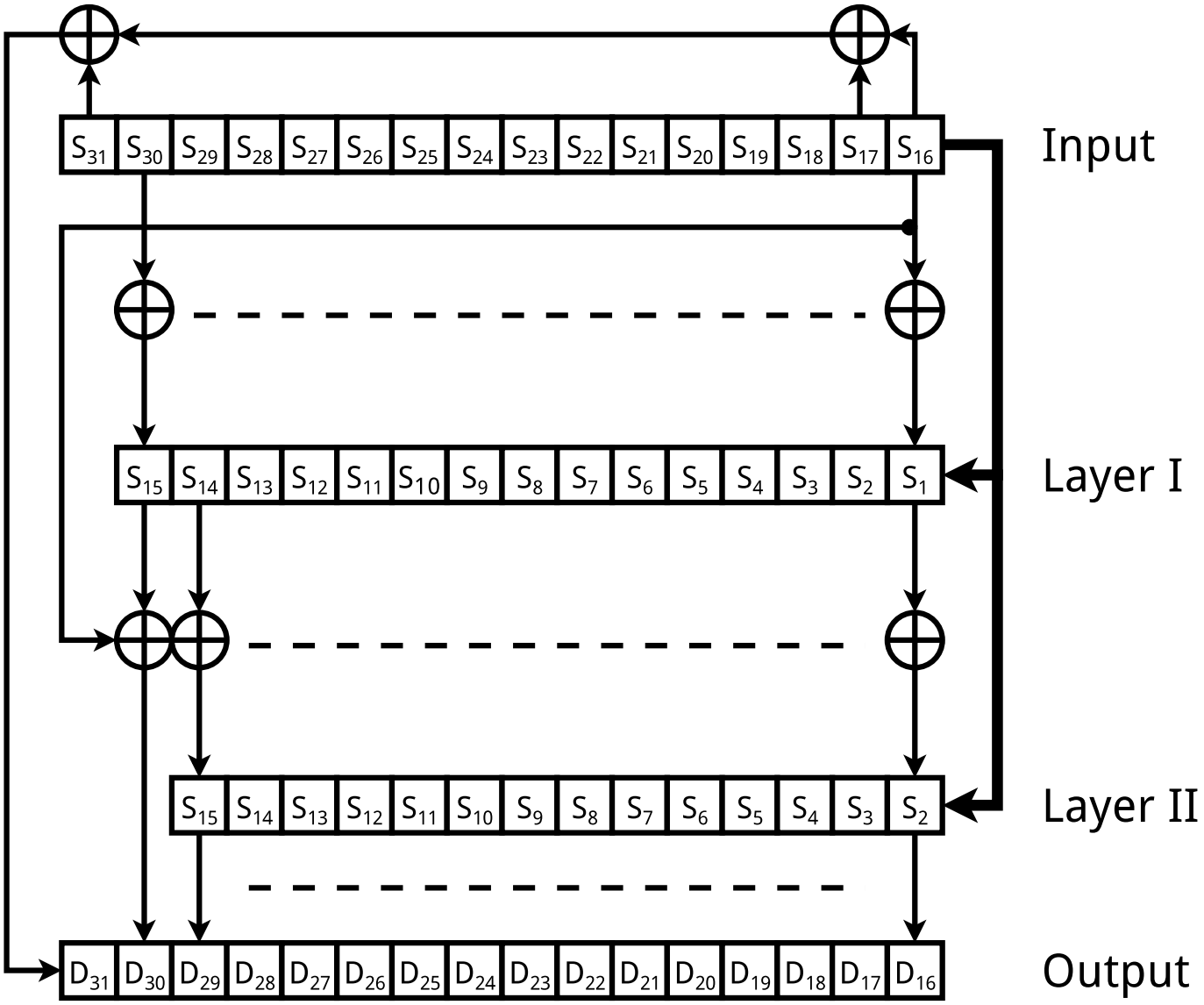

图 7: 解扰器并行实现

为了适应高速的处理环境，解扰器的尽量采用并行设计的思路。原先解扰器采用的是串行设计，一个时钟周期只能处理一位。在高速系统中，位于数据链路层的解扰器不属于关键部分，不应采用过高的时钟频率。图 7中将解扰器分为四层进行处理，一次可以 16 位加扰数据。模块以两个 octet 的周期作为时钟，通过并行处理加快处理速度，牺牲了一定的芯片面积。

第一层输入层，作为最先输入的需要解扰的数据，在解扰使能刚开启时只需要对最后一位进行解扰。在完成第一个解扰周期的工作后，会将输入的低 15 位复制到第二层，低 14 位复制到第三层，作为之后解扰的依据。第二、第三层在一个解扰周期中主要是作为暂存器，将之前的输入存储下来，为之后的解扰做准备。通过三层数据对应位置的亦或操作，最后输出正确的解扰结果送到第四层输出层。

1.3 码群同步设计

码群同步即是指 Code Group Synchronization，它的主要作用就是对控制字/K28.5/进行检测，以判断链路是否开始工作。当接收端收到连续四个/K28.5/控制字后，既可以判断码群同步成功。在这一时刻，接收端就会取消 SYNC 请求，通知发送端开始发送 ILA 序列或者具体数据。SYNC 请求是反映在接收端的 SYNC 接口上的电平信号，根据不同的 Subclass，有不同的置位方式。由于 SYNC 接口还承担着报错、请求同步等各种功能，所以码群同步阶段的 SYNC 请求指示 SYNC 接口需要处理的一种。针对来自码群同步阶段的取消 SYNC 请求，在 Subclass 0 类设备是在每一个本地的 frame 时钟沿进行触发，而对于 Subclass 2 类和 Subclass 1 类设备，都是在本地 multiframe 时钟沿进行触发。

1.3.1 协议分析

码群同步主要需要检测链路中比特是否同步的问题，发送端在这一阶段会不断的根据自己的时钟发送/K28.5/字符，接收端需要根据自身的接收电路和解码器恢复出/K28.5/字符。对于这一阶段的处理，主要是完成/K28.5/字符的检测，也需要对接收到的字符进行判断，是否出现错误。这里的错误主要指的是极性的错误，即当前接收端的极性和接收到字符的极性不相符的错误。当在累计出现三次极性错误，并且期间没有发现连续的四个正确字符的情况下，需要对 lane 重新进行码群同步。主要指的是发起 SYNC 请求，这时的接收端就会重新进入码群同步的初始化状态，等待接收正确的/K28.5/字符。

对于 Subclass 0 类设备的具体的握手流程，如图 8所示。

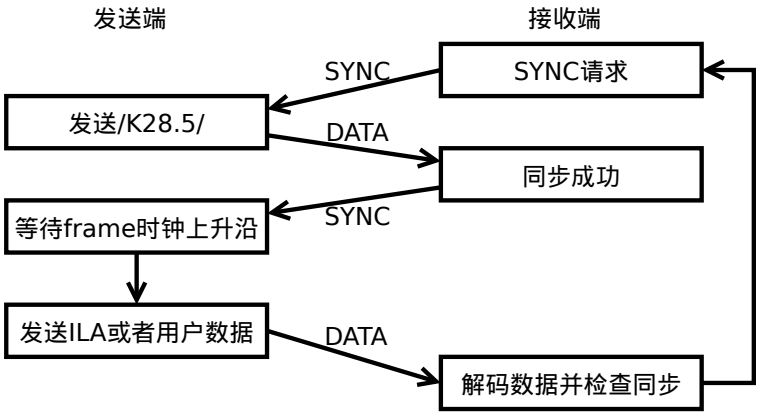

图 8: *Subclass 0* 类设备码群同步过程

对于 Subclass 1 类或 Subclass 2 类设备的具体的握手流程，如图 9所示。

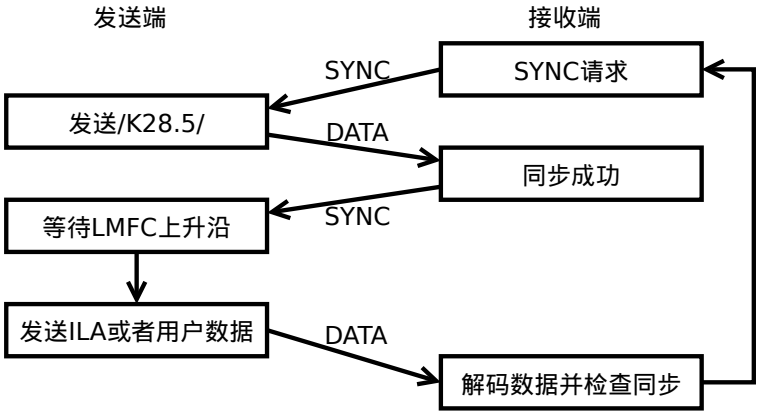

图 9: *Subclass 1* 类和 *2* 类设备码群同步过程

1.3.2 设计思路

实际上，码群同步的过程是一直伴随着整个链路的启动过程的，它需要时刻监控链路上的字符信息。当发送端发送的数据出现连续的错误，或者链路状况变得非常恶劣时，码群同步需要能够检测到这些情况并及时的跳转状态进入到需要重新初始化的状态。这一过程可以通过一个状态机来描述，如图 10所示。

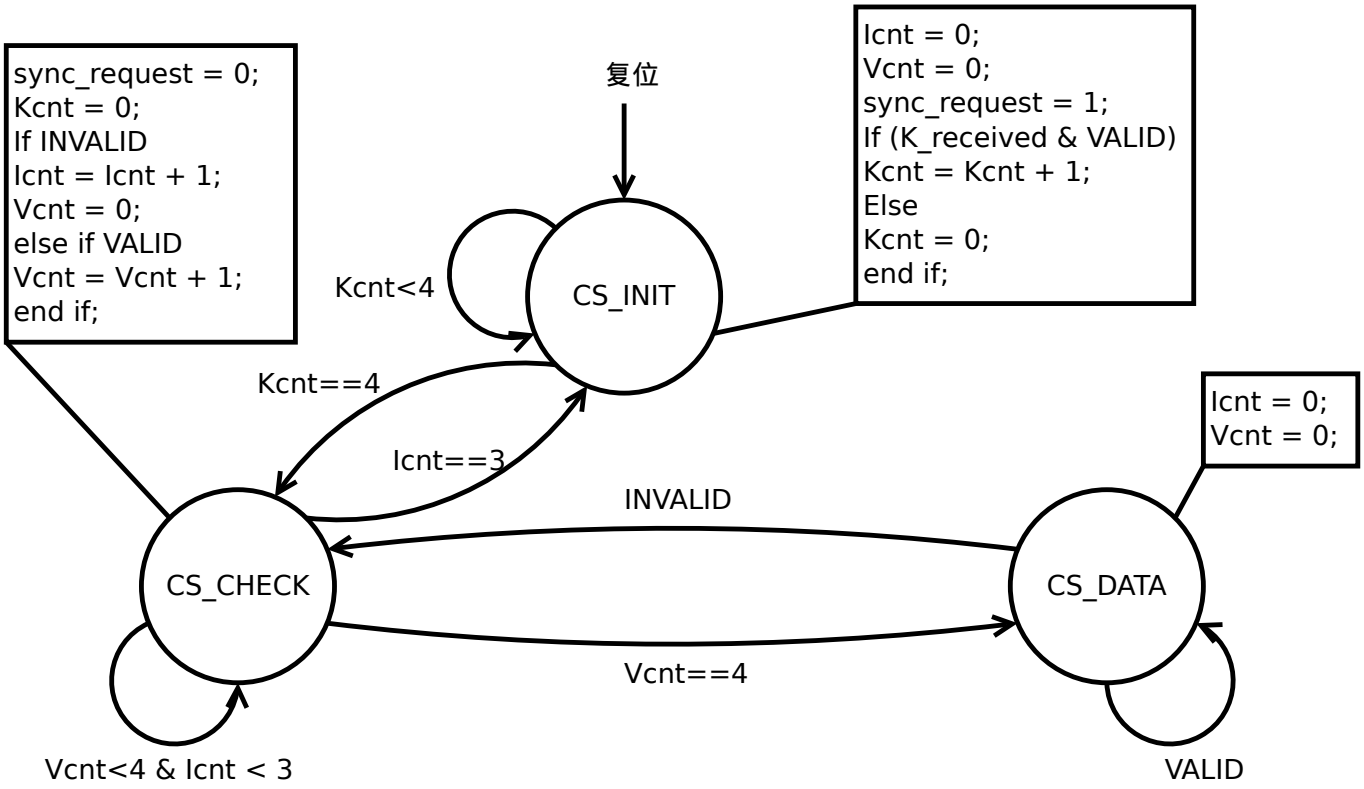

图 10: 码群同步有限状态机

其中 Icnt 指的是错误数据的计数，Vcnt 指的是正确数据的计数，Kcnt 指的是/K28.5/数量的计数，VALID 指的是接收到正确的数据，INVALID 指的是接收到错误的数据，sync_request 指的是 SYNC 请求信号。

码群同步在接收端主要有以下几步：

1. 当连接启动，接受端发送一个同步请求，发送端传回/K28.5/。
2. 接收端终止同步条件根据 Subclass 决定
   - Subclass 0: 在收到 4 个连续的/K28.5/后的帧边界终止同步请求。
   - Subclass 1/2: 在收到 4 个连续的/k28.5/后的本地多帧时钟边界终止同步请求。
3. 在正确接收另外 4 个 8B/10B 字符后，接收端认为码群同步完成。
4. 当继续收到错误码字后，接收端进入“check”状态。
5. 如果又有三个错误码字在“check”状态被接收，就表示失去同步。
6. 如果在“check”状态连续收到 4 个正常码字，接收端退出“check”状态进入正常运行。

同步请求是通过 SYNC 接口传送到发送端的。同步请求被编码成 SYNC 信号，保持低电平，并且只可能在接收端 frame 时钟的上升沿发生变化。SYNC 信号上最短的同步请求时长是 5 个 frame 加 9 个 octets。

接收端只有在激活了同步请求的阶段才允许将/K28.5/作为同步码组。如果在数据传输阶段，码群码群之间检测到了/K28.5/则视作位错误。

1.4 初始化帧同步

当一个 link 启动时，帧同步的由以下几点实现：

- 在码群同步过程中，发送端会一直发送/K28.5/的逗点符号。
- 在码群同步结束后，收端会假设收到第一个非/K28.5/符号作为帧的开始。如果发送端开始发送 ILAS，那么第一个非/K28.5/字符就是/K28.0/字符。
- 接收端假设在每 F 个 octet 后就是新的一帧的开始。

帧的初始化就是依赖于非/K28.5/字符的开始，并且以配置参数 F 作为新一帧的界定。这种以 F 个 octet 周期作为一帧的时钟可以作为帧时钟信号，用于界定新的帧，并且能够同步到第一个非/K28.5/符号，以保证帧时钟对齐。帧时钟是一个很重要的时钟信号，并且在确定性时延中有重要的作用。

1.4.1 帧对齐和帧纠错

帧对齐是通过监视帧对齐字符来保证的，这些对齐字符是由发射端在确定情况下加在每一个帧的结尾处。当接受端收到的错误并不是一个在 lane 中的位错误时，需要用对齐字符来进行重同步。通常情况下，只有在相同的位置连续接收到不正确的字符时，才需要进行重同步。如果是 lane 对齐导致的帧对齐字符的不正确接收，则不需要在同一位置等待对齐字符。

这里的对齐字符即/K28.7/。如果两边的 lane 都支持 lane 同步，那么 lane 的对齐字符/K28.3/应该在一个多帧的最后一个帧。/F/字符即/K28.7/，被编码成 0xFC；/A/字符即/K28.3/，被编码成 0x7C。符号替换是否启用，由是否启用扰码和 lane 同步是否支持替换来决定。下面就分扰码是否开启来讨论字符的替换问题。

不开启扰码 如果两边均支持 lane 同步，那么发送端和接受端中的符号替换在传输层中遵循以下规则：

- 当最后的 octet 在当前帧中并且没有和多帧的结尾发生冲突并且等于前一帧的最后一个 octet，那么这个 octet 将由发送端替换为控制字符/K28.7/。如果前一帧中已经传送了对齐字符，那就发送原数据。
- 当现有多帧的最后一个帧的最后一个 octet 等于前一帧的最后一个 octet，发送端需要将这个 octet 替换成/K28.3/，即使一个控制字符已经在之前发送过了。
- 当收到/K28.7/或者/K28.3/时，接收端需要用解码后的数据代替她，或者用上一帧同样位置的数据代替。

如果至少有一边不支持 lane 同步，那么发送端和接受端中的符号替换在传输层中遵循以下规则：

- 当现有帧的最后一个 octet 等于前一帧的最后一个 octet，发送端需要用/K28.7/替换这个 octet。但是如果在之前的帧已经传输过该控制字，则按原数据发送。
- 当收到/K28.7/时，接收端需要用上一帧同样位置的数据代替。

开启扰码 如果两边均支持 lane 同步，那么发送端和接受端中的符号替换在传输层中遵循以下规则：

- 当一个加扰过后的 octet 位于一个帧的最后位置等于 0xFC，并且不在多帧的最后，发送端需要将其编码为/K28.7/。
- 当一个加扰过后的 octet 位于一个多帧的最后位置等于 0x7C，发送端需要将其编码为/K28.3/。
- 当收到/K28.7/或者/K28.3/时，接收端需要对其解扰。

如果至少有一边不支持 lane 同步，那么发送端和接受端中的符号替换在传输层中遵循以下规则：

- 当一个加扰过后的 octet 位于一个帧的最后位置等于 0xFC，发送端需要用/K28.7/替换这个 octet。
- 当收到/K28.7/时，接收端需要将 0xFC 放入解扰器。

帧同步在接收端的纠错 当纠错启动时，需要执行以下步骤：

- 如果连续两次有效的对齐字符在同一位置被检测到，并且不在接收端认为的正确帧结束位置。在两个对齐字符之间也没接收到一个在期望位置的对齐字符，接收端将会重新对齐帧边界到那个收到的对齐字符位置。
- 在重新对 lane 对齐过后，可能会导致帧对齐错误，接收端将会重新对齐帧到第一次接收到对齐字符的位置。
- 接收端需要有一个配置去取消重同步，因为如果没有进行加扰，确定类型的周期信号可能无法产生足够的对齐字符可以用来侦测对齐的失败与否。

1.4.2 设计思路

根据协议建议，帧同步的过程可以通过有限状态机实现，可以分为两种情况。一是需要支持数据接口重同步请求的设备中，另一种是不需要支持重同步请求的设备。第一种设备的有限状态机设计如图 11所示，第二种设备的有限状态机设计如图 12所示。

图 11所示是带重同步功能的初始化帧同步的有限状态机描述。

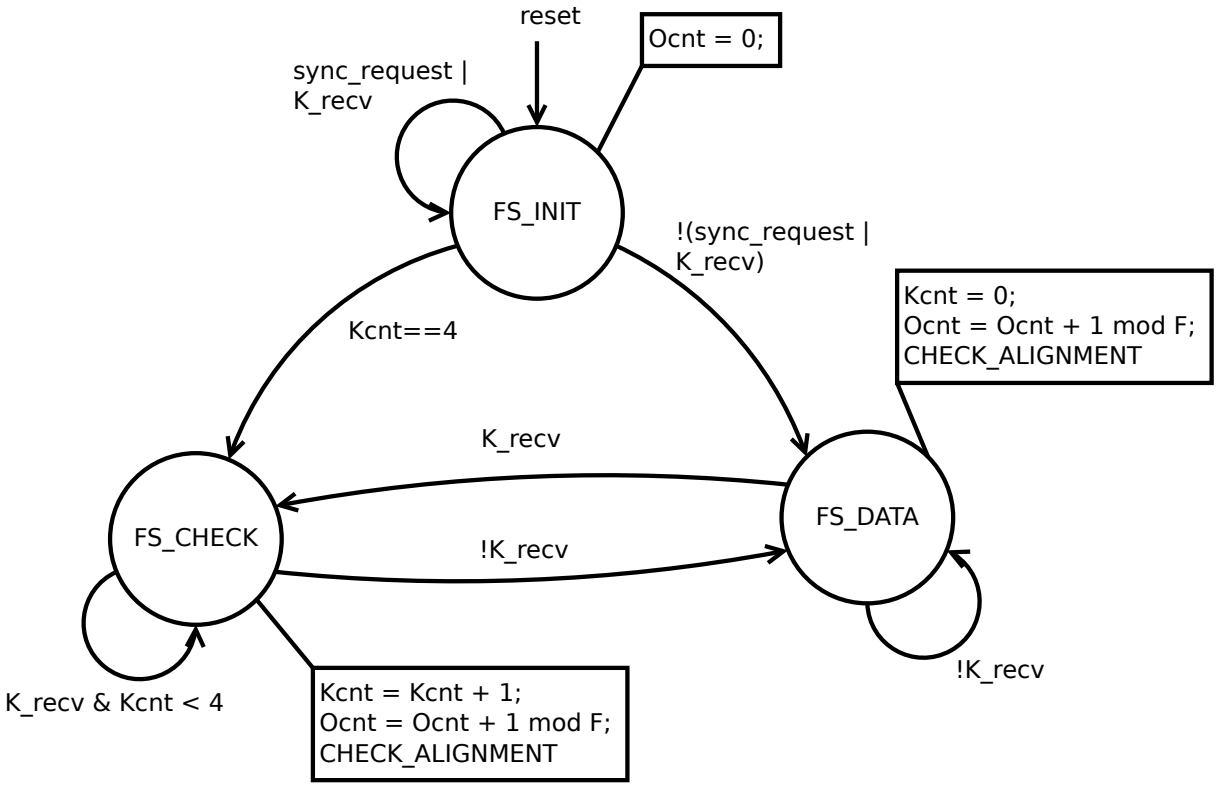

图 11: 有重同步的初始化帧同步有限状态机

图 12所示是不带重同步功能的初始化帧同步的有限状态机描述。

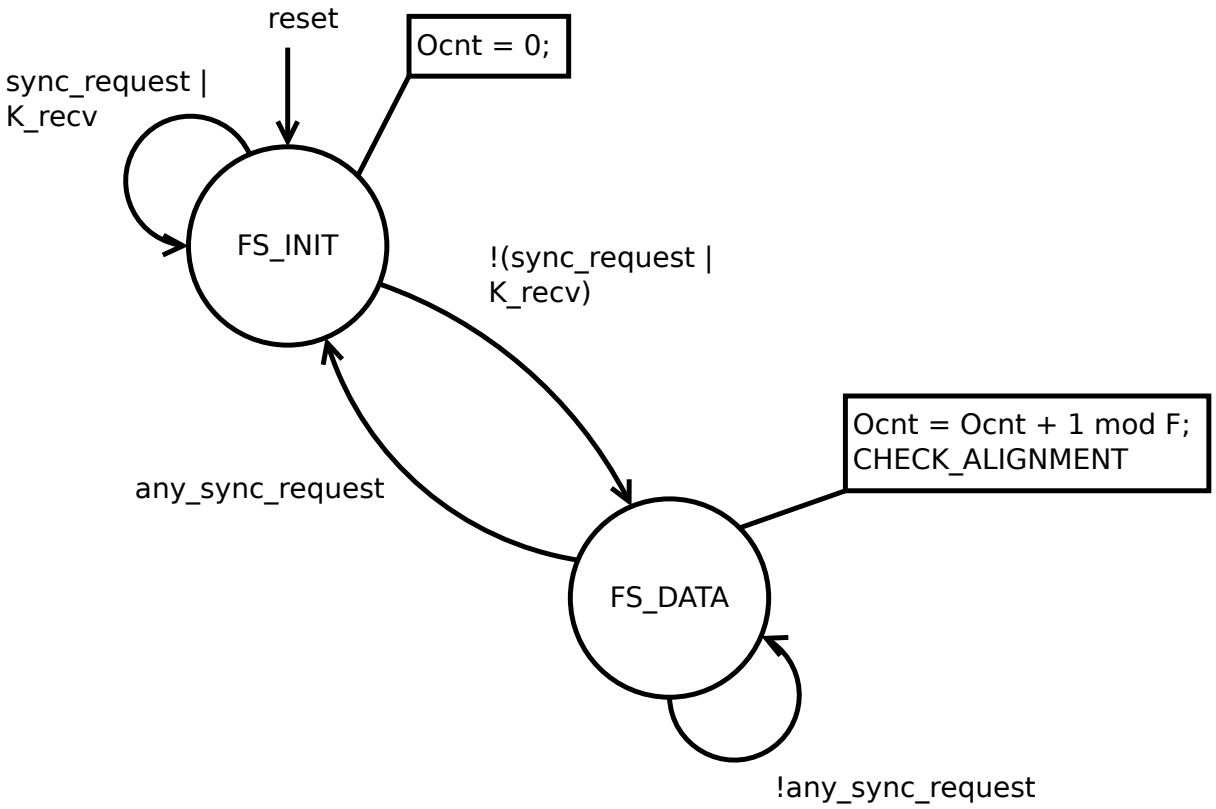

图 12: 没有重同步初始化帧同步有限状态机

其中，any_sync_request 指的是任意连接在 link 上的接收端的同步请求；CHECK_ALIGNMENT 指的是对帧对齐监控；K_recv 指的是标示现在收到的符号为/K28.5/；Kcnt 指的是在 FS_CHECK 状态对有效的/K28.5/计数；sync_request 指的是接收端检测到失去码群同步，发出同步请求，或者发生别的错误需要重初始化；F 指的是每个帧的 octet 数；Ocnt 指的是标记此帧中的当前 octet 位置。

在设备重置后，状态机进入初始化状态，octet 计数器被清零，码群同步激活，接收端启动同步请求。同步请求的激活状态会使帧同步保持在初始化状态。当同步请求被取消后，状态机转入 FS_DATA 状态，同时发射端停止发送/K28.5/符号。在 FS_DATA 状态，octet 计数器 Ocnt 统计该帧中的 octet 数目，在 0 和 F-1 之间。如果 link 中任何一个接收端启动了同步请求，就要从 FS_DATA 状态进入 FS_INIT 状态。这种情况只会发生在监视器检测接收到了/K28.5/，在多接收端的设备中。

如果/K28.5/被接收到，状态机进入 FS_CHECK 状态，在这个状态 octet 计数器 Ocnt 继续工作。如果连续 4 个/K28.5/被接收到，帧同步回到 FS_INIT 状态。对于只配置成单接收端的设备，可以监测到所有接收端各自的同步请求。

同样的，对于重对齐的检测和纠错，协议也给出了建议。监测和纠错的过程由以下伪代码所示。纠错功能应该可以通过寄存器设置来关闭或者开启（RESET_OCTET_COUNTER），尤其是当用户数据不能产生有效的对齐字符以供监测的时候。

Algorithm 1 帧对齐监测和纠错

```
1: if (A_recv | F_recv) then
2:     REPLACE_ALIGNMENT_CHARACTER;
3:     if (((Ocnt == previous_AF_position) | CROSS_COUPLING) & VALID) then
4:         RESET_OCTET_COUNTER;
5:     end if
6:     if VALID | (Ocnt == F-1) then
7:         previous_AF_position = Ocnt;
8:     end if
9: end if
```

其中，A_recv 表示接收到了/K28.3/，即/A/字符；CROSS_COUPLING 指的是由于 lane 和帧之间的对齐交叉配对导致帧失去同步；F 指的是每个帧的 octet 数；F_recv 表示接收到了/K28.7/；Ocnt 表示此帧中的当前 octet 位置；previous_AF_position 记录/K28.3/或/K28.7/在帧中的位置；RESET_OCTET_COUNTER 指在接收到下一个 octet 时，重置 octet 计数至 0；VALID 表示在给定极性情况下收到的正确码字。

REPLACE_ALIGNMENT_CHARACTER 用以下字符替换在解码输出处的对齐字符：

- 当扰码取消时，用上一帧同样位置的数据，或者实用解码后的数据；
- 当扰码启用时，数据字符保持不变。

记录下/K28.3/的位置，如果在 lane 同步或者 lane 对齐监测中需要。

1.5 初始化 lane 同步

1.5.1 协议分析

在具体数据发送之前需要进行初始化 lane 同步。lane 同步使用具体的控制字/K28.3/。对于发送端来说，会在一个固定的时间点，所有的 lane 同时发送 lane 对齐字符/K28.3/。对于不同的 lane 会产生不同的延时，不同 lane 的对齐字符可能在不同时刻被接收到。每个接收端在收到/K28.3/后，会对对收到的数据进行缓存，并且对别的接受端标志出“ready”状态，表示缓存里已经有了一个有效的对齐起点。当所有的接收端标示了对齐接收到的标志后，会在规定的同一个时刻传输接收到的数据，以保证数据的同步。

对于 JESD204B 设备，支持确定时延，所以具体的开始数据传输时刻会有具体的规定，将在确定性时延章节具体讨论。确定性时延能够设置一段时间的延时用来保证所有的 lane 收到对齐字符。

初始化 lane 同步在码群同步结束后立即开始执行，依靠的是发送 ILAS。不需要对 ILAS 进行加扰。作为模数转换器设备会固定的传输 4 个多帧的 ILAS，对于 Subclass 1 和 2 的数模转换设备会固定接收 4 个多帧的 ILAS。同时配置多个 Subclass 0 类的数模转换器设备可能需要更多多帧的 ILAS 来保证 lane 的对齐。而对于逻辑设备来说，ILAS 需要能够设置成 4 到 256 个多帧的长度。这里一个多帧就指的是 K 个连续的帧，这里的 K 值是介于 1 和 32 之间的数，即表示每个多帧的 octet 数是介于 17 和 1024 之间的值。

具体 K 的范围如下式所示：

$$ceil(17/F) \leq K \leq min(32, floor(1024/F))$$

对于 JESD204 发送设备来说，K 的值应该是可编程的。对于 JESD204 接收设备来说，K 的值也建议设置成可编程。与此同时，接收设备也要给出建议的 K 配置值，方便发送设备配置具体的数值。

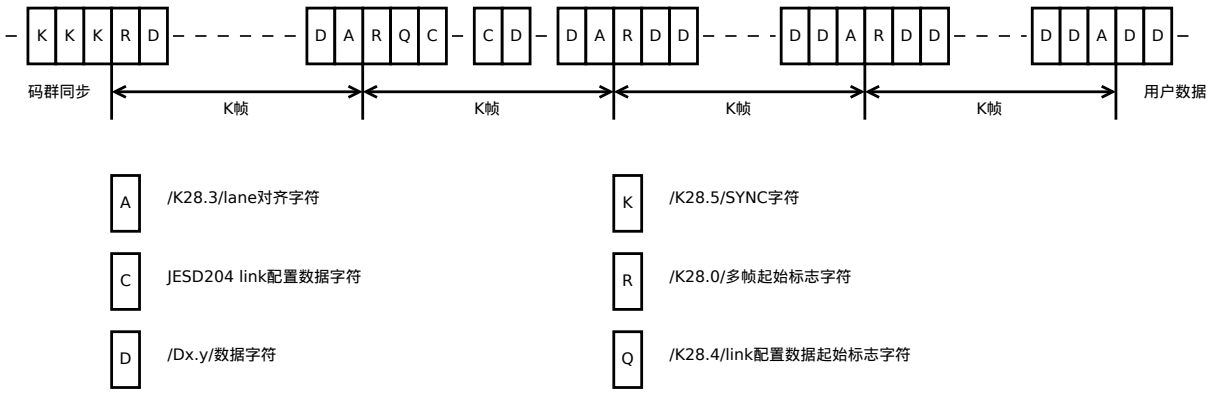

图 13: *ILAS*

ILAS 的结构如图 13所示。每一个多帧由/K28.0/作为起始，/K28.3/作为结束。/K28.0/告诉了接收端这个多帧是 ILAS 的一部分。/K28.3/作为多帧的结尾，也同时作用于 lane 和帧同步。第二个多帧需要包含设置信息，用来配置接收端和发送端的 link，从第三个字符开始。第二个多帧的第二个字符则为/K28.4/，作为再次确认信息，标示之后的数据作为设置参数。

在初始化帧和 lane 对齐后，信道转换到对齐监视模式。lane 对齐通过观察/K28.3/来实现。因为/K28.3/是由发射端在确定情况下插入在多帧结尾的控制字。通常情况下不是所有的 lane 会同时传送/K28.3/。每一个接收端可以分别检查所收到的/K28.3/，并同各自的本地时钟比较，以确定是否需要纠正对齐错误。

每个多 lane 环境下的接收端可以通过更高层的应用确认是否进行重对齐，当以下三种情况同时发生：

- 准确的检测到了失去对齐，并且
- 发现了一个可靠的新的对齐位置，并且
- 在现有缓存中的数据能够实现重对齐。

重对齐命令通过接收端的控制接口由更高的层级授权。lane 的重对齐类似于前面提到的帧重对齐。

- 如果连续两次有效的对齐字符在同一位置被检测到，并且不在正确的帧结束的位置，而且在两个对齐字符之间也没接收到一个对齐字符，在期望的位置，接收端将会重新对齐帧在那个收到对齐字符的位置。
- 在重新对 lane 对齐过后，可能会导致帧对齐错误，接收端将会重新对齐帧到第一次接收到对齐字符的位置。

这里要注意的是，只有当本地的时钟参考是正确时，动态的重对齐能够正确的恢复出 lane 的时延。如果 lane 失去对齐是因为本地时钟参考的相位发生了改变，那么动态的重对齐最终会将所有的 lane 对齐到相同的参考上，不过 link 的所有延时都会改变。为了避免可能的延时改变，Subclass 1 类和 2 类的接收端能够通过 SYSREF 或者 SYNC 信号将本地多帧时钟重新对齐，来实现动态的重对齐。所有 Subclass 的接收设备也可以通过发起一个重同步请求来实现动态的重对齐。

在确定错误产生的情况下，接收端能够请求重新初始化 link，通过重启一个同步请求。如果所有的接收端在同一个设备中，所有的同步请求能够被所有的接收端检测到。这种方式的设备内部同步请求能够对同一设备的所有接收端的帧和 lane 的同步重新初始化。

另一个方法是通过发送端重置接收端。因为所有接收端的同步请求被整合到一个 SYNC 信号中，这样是同时对所有发送端的同步请求，所有的发送端都会重新发送/K28.5/对任何接收端。发送端能够通过转换状态到 SYNC 模式请求重初始化，发送一系列的/K28.5/。

在单独一个设备中的接受端，应该能在 4 个 ILAS 多帧的时间内完成相互的 lane 对齐。Subclass 1 类或者 2 类设备应该能根据在 4 个多帧内，对齐新来的对齐字符到内部的本地多帧时钟边界。当接收端被分成多个 Subclass 0 类设备，需要设置指定数量的 ILAS 多帧来完成对齐，但不超过 256 个多帧。

在设置了多帧的情况下，当任何一个在同一 link 上的接收端发起了对齐请求，每个接收端都要准备一个新的初始化 lane 同步。在设置了单接收端设备时，需要监控所有接收端的设备的同步请求。

1.5.2 设计思路

实际上的初始化 lane 对齐就是对 ILAS 序列的检测、分析工作和 lane 的监测工作。在完成了初始化帧同步后紧接着的就是初始化 lane 对齐的工作。当初始化 lane 对齐完成后就需要对 lane 进行监控，判断是否需要重对齐。

ILAS 序列的检测 对于接收设备来说，ILAS 的长度是数量固定的 4 个多帧长度，其中的第二个多帧包含了关键的 link 配置信息。接收端首先要做的是将 4 个多帧长度的 ILAS 拆解，一是提取出对齐信息，一是提取出 link 配置信息。比较关键的是对 link 配置信息的分析，里面主要包含了发送端的一些重要参数。这些参数会以数据的形式存放，并且确定的位于 ILAS 多帧中的第二个多帧，紧跟在/K28.4/字符之后。配置数据的长度为固定的 14 个 octet 长度。

其具体组成的块状结构如图 14所示。

Configuration octet no.	Bits							
	MSB	6	5	4	3	2	1	LSB
0	DID<7:0>							
1	ADJCNT<3:0>				BID<7:0>			
2	X	ADJDIR	PHADJ	LID<4:0>				
3	SCR	X	X	L<4:0>				
4	F<7:0>							
5	X	X	X	K<4:0>				
6	M<7:0>							
7	CS<1:0>		X	N<4:0>				
8	SUBCLASSV<2:0>			N'<4:0>				
9	JESDV<2:0>			S<4:0>				
10	HD	X	X	CF<4:0>				
11	RES1<7:0>-Set to all X							
12	RES2<7:0>-Set to all X							
13	FCHK<7:0>							

图 14: *link* 配置信息映射图

几个主要的具体参数如下所示：

ADJCNT 调整幅度计数，主要是在 Subclass 2 类设备中用于标识本地多帧时钟的调整数量。

ADJDIR 调整方向，主要是在 Subclass 2 类设备中用于标识本地多帧时钟的调整方向。

PHADJ 调整使能，主要是在 Subclass 2 类设备中用于标识是否调整本地多帧时钟。

JESDV 标识 JESD 版本信息，000 为 JESD204A，001 为 JEAD204B。

SCR 加扰使能，标识之后的用户数据是否加扰。

CHKSUM 校验和，之前数据的和模 256。

其余如 CS、CF 等参数在本章节前面内容有具体介绍，故不再追赘述。

lane 对齐监测和纠错 监测和纠错的过程由以下伪代码所示。lane 对齐监控和纠错通帧对齐监测和纠错非常的相似。但是它包含了一个额外的步骤，当对齐字符没有到达正确的位置，他会建议在 LMFC 之间初始化一个同步检测。当发送端无法发送有效的对齐字符时，lane 对齐检查应该可以通过设置来取消。

Algorithm 2 lane 对齐监测和纠错

```
1: if A_recv then
2:     REPLACE_A;
3:     if (((Fcnt == previous_A_position) | CROSS_COUPLING) & VALID) then
4:         RESET_FRAME_COUNTER;
5:         if (Fcnt != K-1) then
6:             INITIATE_SYNC_CHECK;
7:         end if
8:     end if
9:     if VALID | (Fcnt == F-1) then
10:        previous_A_position = Fcnt;
11:    end if
12: end if
```

A_recv 表示接收到了/K28.3/；CROSS_COUPLING 指由于 lane 和帧之间的对齐交叉配对导致帧失去同步；Fcnt 表示此多帧中的当前帧位置；K 表示单个多帧中帧的数量；previous_A_position 记录/K28.3/在多帧中的帧位置；RESET_FRAME_COUNTER 指在接收到下一个帧时，重置帧计数至 0；INITIATE_SYNC_CHECK 指如果通过控制接口，授权被允许，就需要在本地多帧时钟之间初始化一个同步检查；VALID 表示在给定极性情况下收到的正确码字。

REPLACE_A 指用以下字符替换在解码输出处的/K28.3/：

- 当扰码取消时，用上一帧同样位置的数据，或者实用解码后的数据；
- 当扰码启用时，用/D28.3/。

如果在帧对齐监测中需要，不替换/K28.3/或者记录下它的位置。

参考文献

[1] Abdullah-Al-Kafi. Development of fsm based running disparity controlled 8b-10b encoder-decoder. *HCTL Open Int. J. of Technology Innovations and Research*, 2, 2013.

[2] Actel. Implementing an 8b/10b encoder/decoder for gigabit ethernet in the actel sx fpga family. Technical report, Actel Corporation, October 1998.

[3] A. X. Widmer. A dc-balanced, partitioned-block, 8b/10b transimission code. *IBM Journal of research and development*, 27‘(5):440–451, September 1983.

[4] 温龙. 8b/10b 解码器设计. 科学技术与工程, 18(7), 2007.

[5] 贺传峰. 一种新的 8b/10b 编解码硬件设计方法. 高技术通讯, 15(3), 2005.

[6] 赵王虎. 基于逻辑设计的光纤通信 8b/10b 编解码方法研究. 电路与系统学报, 8(2), 2003.