

# Self-Synchronizing Sequential Coding with Low Redundancy

By PETER G. NEUMANN

(Manuscript received August 31, 1970)

*In this paper, we present sequential codes which have interesting properties in three respects. First, these codes may be used to achieve low redundancy (e.g., bandwidth compression through coding) by employing a multiplicity of variable-length codes to encode transitions between successive source symbols. Second, the coding complexity is surprisingly low. Third, many of these codes have exceedingly good intrinsic recoverability properties following errors. These codes compare favorably with a difference code environment in which the differences between successive source symbols are encoded. The scope of the sequential codes presented here includes, but is much wider than, difference code schemes. Where comparable, the sequential codes have slightly greater complexity and may have lower redundancy. They normally have vastly superior error recovery. These codes are applicable in situations such as video transmission in which the message source is highly correlated and where errors can be tolerated for a short period of time.*

## I. INTRODUCTION

In several previous papers, the author has pursued two apparently separate paths of development. The first path involves classes of slightly suboptimal variable-length prefix codes<sup>1,2</sup> whose self-synchronizing abilities are vastly superior to the optimal (Huffman) codes which minimize redundancy. The second path involves self-synchronizing sequential codes using information-lossless sequential machines as encoders and decoders.<sup>3,4</sup> In this paper, these two paths of development are joined. The result produces highly efficient sequential codes (with low redundancy) which have good self-synchronizing abilities and surprisingly low decoding complexity. These codes are applicable in situations in which the message source is highly correlated.

### 1.1 *Difference Codes*

Given an environment in which successive source signals are likely to be similar (or at least strongly correlated), considerable compression can be obtained by encoding the level difference between successive quantized signal levels (temporally or spatially). Such is the situation in a video picture environment, for example, with respect to adjacent points horizontally or vertically, or even to the same point in successive frames. By encoding differences, however, any error in quantizing, encoding, transmitting, decoding or reconstructing the image tends to persist. That is, once an error has occurred in a signal level, subsequent levels will continue to be in error by the same offset, unless terminated by boundary effects or by compensating errors. In an encoding of level differences between successive points in a line, for example, errors tend to propagate until the end of the line; in an encoding of level differences between the same point in successive frames, on the other hand, errors may continue forever. In order to prevent such error effects from propagating indefinitely, it may be necessary to terminate the propagation forcibly, for example by transmitting periodically the set of signal levels for the entire frame ("replenishment") rather than their frame-to-frame differences. Thus the use of difference coding for compression may be compromised by the need to resynchronize. This is true in general of frame-to-frame difference codes. An example of the use of such codes in a differential pulse-code modulation environment is given in Ref. 5.

## II. SELF-SYNCHRONIZATION

The main purpose of this paper is to present codes which have compression capabilities at least as good as difference codes, along with roughly comparable decoding complexity, as well as having rather remarkable intrinsic self-synchronization properties. (These codes are in fact much more general than difference codes in terms of compression capabilities.) These codes recover quickly from the effects of errors in that arbitrary errors give incorrect results for a period of time, after which the entire system resumes correct operation without any explicit effort. (Note that self-synchronization is a property of the coding scheme, and should not be confused with video picture frame synchronization.) It is important to note that in such a scheme the errors are not corrected (in the sense of error-correcting codes); instead errors are *tolerated*, with the expectation that their effect will cease quickly. Video coding is an example where such an

approach is reasonable since loss of information for a short period of time can often be tolerated.

### 2.1 *Self-Synchronization in Variable-Length Codes*

The use of variable-length codes for reducing redundancy is well understood. For example, D. A. Huffman<sup>6</sup> shows how to obtain a code which minimizes the transmitted information for a given independent distribution of source symbols. However, there has been relatively little quantitative concern for the effects of errors on these codes. In earlier papers<sup>1,2</sup> the author has shown that a slight sacrifice in efficiency (i.e., a slight increase in transmitted information) can be rewarded with tremendous gains in self-synchronizing capability. Some of this work is required here and is reviewed briefly, although from a different viewpoint.

A *code* is a collection of sequences of digits (code digits), each sequence being called a *code word*. *Code text* is obtained by concatenating code words. An *encoding* is a mapping of source symbols  $S(i)$  onto code words  $W(i)$ . A code is a prefix code if and only if no code word occurs as the beginning (prefix) of any other code word. Thus in prefix code text, a code word can be decoded as soon as it is received, even though there are no explicit interword markers. A code is *exhaustive* if and only if every sequence of code digits is the prefix of some code text (i.e., of some sequence of code words). (A uniquely decodable code must be a prefix code if it is exhaustive.<sup>7</sup>) A sequence of code digits is a *synchronizing sequence* for a given code if the occurrence of the end of that sequence in (correct) code text must correspond to the end of a code word (although not necessarily to a particular code word), irrespective of what preceded that sequence. M. P. Schützenberger<sup>8</sup> and E. N. Gilbert and E. F. Moore<sup>7</sup> have shown that most exhaustive prefix codes tend to resynchronize themselves following loss of synchronization (e.g., after arbitrary errors, or at start-up). If an exhaustive code has at least one synchronizing sequence, then the code tends to resynchronize itself following errors with a finite average delay (assuming a suitable randomness). All codes considered here are exhaustive unless explicitly stated otherwise. Note that resynchronization is an intrinsic property of the code, and no externally implied synchronization is required. Synchronization following ambiguity occurs as a result of any synchronizing sequence occurring naturally in code text.

As an example, consider the code of Fig. 1, consisting of the five code words 00, 01, 10, 110, 111. The tree of Fig. 1 may be interpreted

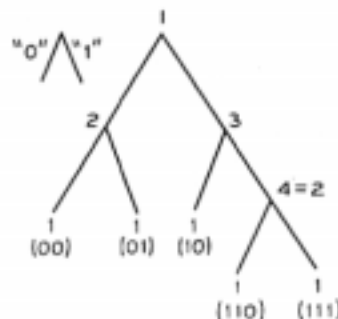


Fig. 1—A simple prefix code.

as the state diagram for a sequential machine<sup>9</sup> which detects the end of a code word whenever the initial state 1 recurs. In this figure (and throughout this paper) a "0" code digit corresponds to left-downward motion, a "1" to right-downward motion. These digits are the inputs to the sequential machine. In Fig. 1, state 4 is equivalent to state 2 (in the usual sequential machine sense<sup>9</sup>), since the respective next states are equivalent, for each input digit. Thus Fig. 1 represents a 3-state machine whose recurrence of state 1 indicates the end of a code word in text.

The synchronizing diagram<sup>3,10</sup> for the code of Fig. 1 is shown in Fig. 2. It is obtained from Fig. 1 by examining the set of next states resulting from each input digit, beginning with the set of all states, i.e., total ambiguity. For example, a "0" digit can lead only to state 1 or 2, and a "1" can lead only to state 1, 2 (formerly 4), or 3. Given the set of states 1, 2, a "1" can lead only to state 1 or 3. In this way it is seen that the sequence 0110 always culminates in the occurrence of state 1, irrespective of the actual state at the beginning. (This is indicated in Fig. 2 by the dark path.) Thus 0110 is a synchronizing sequence for the code. (Note that its occurrence in code text following ambiguity does not imply the conclusion of a particular code word; either 10 or 110 could be involved.) There is an infinite set of synchronizing sequences described by Fig. 2, including as other examples 0111110 and 10110.

If a sequential machine (or a code) has at least one synchronizing sequence, then it tends to resynchronize itself with probability one,<sup>11</sup> assuming all input sequences are possible. In order to obtain a measure of how well text for a given code is self-synchronizing following arbitrary errors, the code digits "0" and "1" are assumed to occur independently and equiprobably. This is in fact a meaningful and useful assumption under various real-life circumstances, even when it is only

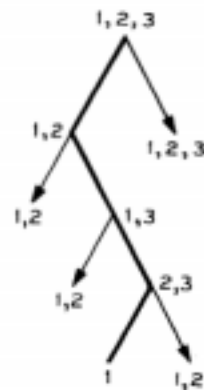


Fig. 2—Synchronizing diagram for the code of Fig. 1, showing unresolved ambiguity.

approximately valid. (This assumption holds exactly whenever each code word occurs independently with its *characteristic probability*  $2^{-d}$ , where  $d$  is the length of the code word in digits.) Assuming this randomness of 0 and 1 in code text, the synchronizing diagram of Fig. 2 is redrawn in Fig. 3 to show that on the average  $I = 16$  digits of code text are required for code text to resynchronize itself following arbitrary errors. (This computation may be done in several ways<sup>1</sup> which are not relevant to the present discussion. Note that the randomness assumption implies that at each node in the synchronizing diagram the residual lag is one more than the average between the residual lags of the two nodes below.) In general, the *synchronization lag* (or, simply, the lag)  $I$  of a (prefix) code is defined as the average number of code digits until synchronization can be guaranteed to the end of some (not necessarily known) code word following total ambiguity, assuming

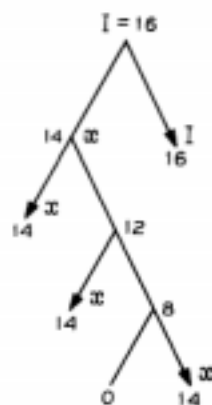


Fig. 3—Copy of Fig. 2, showing lag at each node.

randomness of "0" and "1" as above. Thus the lag is the average length of the synchronizing sequences. (It is also convenient to speak of the *actual lag*, the same average but based on the probabilities of the actual source distribution rather than on random text. If the randomness assumption is not roughly applicable in a given case, then it is necessary to investigate the actual lag. If the likelihood of the various synchronizing sequences actually occurring in text is smaller than under the randomness assumption [e.g., because of constraints on the source symbols], the actual lag is larger than  $I$ . However, in such cases it is often possible to change the encoding slightly to assure that the actual lag is less than  $I$ , without adversely affecting compression. It may also be possible to use a different code with the same set of code-word lengths whose lag is less. For example, the code 00, 01, 11, 100, 101 has  $I = 5$ , a marked improvement over the code of Fig. 1, with  $I = 16$ ; for purposes of compression, these two codes are equivalent. In general, the randomness assumption and the resulting lag are quite useful.)

[Synchronizing sequences also exist for uniquely decodable non-prefix codes. For example, consider the code 00, 01, 11, 001, 011.<sup>3</sup> The sequence 10 in code text guarantees that a code-word end occurred between the "1" and the "0"; the lag is 4. Since such codes may require decoding delays (in some cases infinite) beyond the end of their code words, they are of little practical significance. (This paragraph and others delimited by square brackets may be omitted on casual reading.)]

Codes vary widely in their ability to resynchronize code text. For each number  $n$  of code words, there is a code with  $I = 2$  (the "best"), and a related code with  $I = 2^{(n-1)} - 2$  (the "bad" code). These codes are shown in Fig. 4 for each  $n \leq 9$ , along with a few other examples. (Note that the "best" code and the "bad" code for each  $n$  have the same set of code-word lengths, and are thus equivalent with respect to compression considerations.) The only finite exhaustive codes with  $I = \infty$  known to Schützenberger<sup>8</sup> (and to the author) are the block codes (e.g., the fixed-length codes (a), (b) in Fig. 4) and three classes of non-block codes: the codes with greatest common divisor of their code-word lengths greater than one (e.g., code (c) in Fig. 4), the uniformly composed codes  $C'$  obtained by concatenating  $f$  times the code words of some code in all combinations (e.g., code (d) in Fig. 4, composed from 0, 10, 11 with  $f = 2$ ), and Schützenberger's "anagrammatic" codes<sup>8</sup> which when scanned backwards are also prefix codes (e.g., code (e) in Fig. 4). Note that block codes have the properties of all of these three classes. (By sacrificing exhaustivity [and optimality], i.e., by














$n$	BEST $I$	BAD $I$	WORST $I$
3			
4			
5			
6			
7	2 ETC.	62 ETC.	
8	2	126	
9	2	254	

Fig. 4—Prefix codes with external lags for  $n \leq 9$ .

eliminating at least one code word from a code, the lag is never increased; in some cases  $I$  is reduced substantially. [A slight extension of the definition of the lag is necessary for non-exhaustive prefix codes to handle sequences which cannot arise.] Thus there are synchronization advantages of nonexhaustive codes.)

The only finite exhaustive codes known to the author which have  $2^{(n-1)} - 2 < I < \infty$  for any  $n$  are the two codes (f) and (g) in Fig. 4. Excluding these two codes and the "bad" codes, all remaining finite lag codes seem to have  $I < 2^{(n-1)} - 2$ , for all  $n$ ; the worst of the re-







of a code word. A *definite* code is one in which a code-word end occurs in code text if and only if one of a finite set of finite sequences occurs; in the example, the sequence "10" forms the set. The other codes are not definite, but nevertheless have small lags. The number  $N(d)$  of code words of each length  $d$  is given for each code. The function  $N(d)$  is usually called the *structure function* of the code. The average code-word length (assuming randomness of "0" and "1" in code text) is  $L$ , the sum of  $d N(d) 2^{-d}$  over all  $d$ . (If each code word occurs independently with its characteristic probability, then  $L$  is equal to the entropy  $H$  of the source distribution.) Note that  $I \geq L$  for all prefix codes, with equality if and only if the code is definite. That is, on the average, one code word is sufficient to resynchronize a definite code, longer being required for a non-definite code.

It appears that asymptotically almost all [completely specified, strongly connected, deterministic] sequential machines have synchronizing sequences; for those that do, the resulting systematic prefix codes are self-synchronizing (i.e.,  $I < \infty$ ). The spectrum of values of  $I$ , however, is wide. As is the case with exhaustive finite prefix codes, the infinite codes with  $I = \infty$  may be of several classes. There exist codes with a non-unit greatest common divisor of their code-word lengths (e.g., Fig. 6a), with uniform composition (e.g., Fig. 6b), and with the anagrammatic property (e.g., Fig. 6c). Unlike the case for finite exhaustive codes, infinite exhaustive codes can easily be constructed which belong to none of these three classes (e.g., Fig. 6d). (Uniformly composed codes are studied in Ref. 13.)

[The worst value of  $I$  for a code derived from a synchronizable  $r$ -state machine appears to be about  $I \leq (r - \frac{1}{2})(2^r - 2) + 1$ ; that is, just about a factor of  $r$  worse than the "bad" code of Fig. 4 with  $r$  states,  $r = n - 1$ . A. E. Laemmel and B. Rudner<sup>14</sup> exhibit for each  $r$  a machine whose shortest synchronizing sequence is  $(r - 1)^2$ . For all  $r$ -state machines with synchronizing sequences, the best bound known to the author for the longest synchronizing sequence is that given by M. A. Fischler and M. Tannenbaum<sup>15</sup>:  $(r - 1)^2$ , exact for  $r \leq 4$ ;  $(r^3 - r)/6$  for small  $r \geq 5$ ;  $11r^3/48$  for large  $r$ . On the other hand, there are many machines with extremely short synchronizing sequences and small values of  $I$ .]

### III. SYNCHRONIZATION IN SEQUENTIAL PREFIX CODES

For purposes of this paper, sequential coding implies the use of a sequential machine for the encoder, and a corresponding sequential

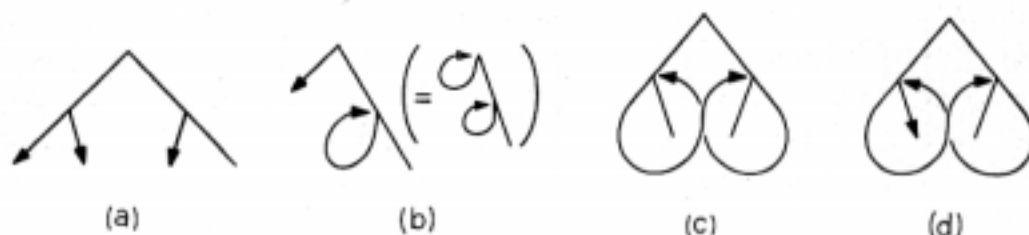


Fig. 6—Examples of systematic codes with infinite lag ( $I = \infty$ ). (a) Greatest common divisor 2;  $N(d) = 0\ 1\ 0\ 3\ 0\ 9\ 0\ 27\ 0\ 81\ \dots$ . (b) Uniform composition;  $f = 2$ ,  $N(d) = 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ \dots$ . (c) Anagrammatic;  $N(d) = 0\ 2\ 2\ 2\ 2\ 2\ 2\ \dots$ . (d) None of the three classes;  $N(d) = 0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ \dots$ .

machine (the inverse or "quasi-inverse") for the decoder.<sup>18</sup> In an earlier paper,<sup>3</sup> the author explored the use of particular input sequences to the encoder (synchronizing input sequences) each of which synchronizes the encoder to a particular known state. Also involved are particular output sequences from the encoder (synchronizing output sequences) whose presence in code text (in the absence of errors) guarantees the occurrence of a particular state of the encoder at a particular point in the code text. Synchronizing output sequences correspond to the synchronizing sequences for prefix codes.

Given an encoder with both synchronizing input sequences and synchronizing output sequences, the entire system is self-synchronizing on the average. That is, following some arbitrary errors, two things happen. First the encoder resynchronizes itself and begins to encode correct text again. Then the decoder tends to resynchronize itself with the encoder (the synchronizing output sequences for the encoder act as synchronizing input sequences for the decoder), and correct decoding resumes. This occurs spontaneously as an intrinsic property of the coding system, with no externally imposed resynchronization required.

A (first-order) *sequential encoding* is a mapping of symbols  $S(i)$  onto code words  $w(i|j)$  where the code word selected depends on the previously encoded symbol  $S(j)$  as well as on  $S(i)$ . If the set of code words  $\{w(i|j)\}$  for each  $j$  is a prefix code, then the set of code words  $\{w(i|j)\}$  for all  $i, j$  is a *sequential prefix code*. For such codes a *synchronizing sequence* is a sequence of code digits the end of which must correspond to the end of a code word (possibly unknown) resulting from a known symbol  $S(i)$ , irrespective of what preceded that sequence. Thereafter subsequent decoding is correct, irrespective of the initial ambiguity. The remainder of this paper is concerned with sequential prefix codes, and investigates their compression, decoding and self-synchronizing properties. (The development is also applicable

to higher-order sequential encodings, with the code word depending on the present message  $S(i)$  and some finite number of previous messages.)

As an example, consider the sequential encoding given by Table I. A, B, C and D represent four source symbols  $S(i)$ ,  $i = 0, 1, 2, 3$ , where  $i$  is the *level* of the symbol. This is an example of an encoding in which the code word  $w(i|j)$  to be transmitted is a function of the cyclic difference between the level of the symbol  $S(i)$  to be encoded (column headings) and the level of the symbol  $S(j)$  just previously encoded (row headings):  $w(i|j) = W(k)$ , where  $k = i - j \pmod{4}$ . This encoding is thus a difference encoding. Note that, irrespective of the choice of the code  $\{W(k)\}$ , there is always ambiguity in decoding as soon as an error is made. If for example  $S(2)$  is decoded instead of  $S(1)$  as a result of a transmission error, subsequent decoding will consistently produce  $S(i+1)$  instead of  $S(i)$  where  $i+1$  is modulo 4, as long as further errors do not compensate for the original errors. (Throughout the paper, all additive operations involving  $i$  and  $j$  are modulo  $n$ .)

As a second example, consider the sequential prefix code of Table II. In this example four different prefix encodings  $w(i|j)$  are used for  $j = 0, 1, 2, 3$ , depending upon the symbol  $S(j)$  previously encoded. For example, if "A" was just encoded, then A, B, C, D are encoded as 0, 11, 100, 101, respectively. Thus any symbol  $S(i)$  as input to the encoder acts as a synchronizing input sequence. (The encoder is a 1-definite machine,<sup>1</sup> with its output being a function of the present input symbol and the previous input symbol.)

A state diagram for the decoder is given in Fig. 7. The states A, B, C, D represent the successful decoding of these four symbols, while the states a, b, c, d, a', b', c', d' are intermediate states. The state diagram is shown in four pieces, which fit together as the upper-case letters indicate. The synchronization diagram for this state diagram is shown in Fig. 8. Its construction follows the usual technique<sup>3,10</sup> and is similar to the synchronization diagram for prefix codes (cf. Fig. 3). The top

TABLE I—FOUR-LEVEL DIFFERENCE CODE

$S(j)$	$i = 0$ $S(i) = A$	1 B	2 C	3 D
$j = 0$ A	W(0)	W(1)	W(2)	W(3)
$j = 1$ B	W(3)	W(0)	W(1)	W(2)
$j = 2$ C	W(2)	W(3)	W(0)	W(1)
$j = 3$ D	W(1)	W(2)	W(3)	W(0)

TABLE II—A GOOD SEQUENTIAL PREFIX CODE

$S(j)$	$i = 0$ $S(i) = A$	1 B	2 C	3 D
$j = 0$ A	0	11	100	101
$j = 1$ B	010	1	00	011
$j = 2$ C	100	11	0	101
$j = 3$ D	010	011	00	1

node corresponds to the set of all states. Each terminal node corresponds to the occurrence of the end of a code word resulting from a particular symbol  $S(i)$ . Given a "0" input, the next state must be one of A, C, b, d, a', c', for example, beginning with the set of all states. The synchronizing diagram of Fig. 8 results after noting several equivalences (e.g., Abda'c' with ACbda'c').

From the synchronizing diagram of Fig. 8 it is seen that the sequence 0011 can arise in code text only if the corresponding source sequence ends in a "B". Thus 0011 synchronizes the decoder to the end of a code word corresponding to the symbol "B" irrespective of what preceded it; similarly 00101 synchronizes to "D", 1100 to "C", and 11010 to "A". Assuming 0 and 1 are random in the above sense, it is easily shown that synchronization results from total ambiguity after an average of  $J = 7.67$  digits. The *sequential synchronization lag*  $J$  is the average number of code digits until the end of a code word is achieved corresponding to a known symbol; that is,  $J$  is the average length of the synchronizing sequences. In this example, the occurrences in code text of the encoded versions of CB, CD, BC and BA imply synchronizing sequences for the decoder. (For example, note that CB is encoded as 0011 following a "B" or "D", as 10011 following an "A",

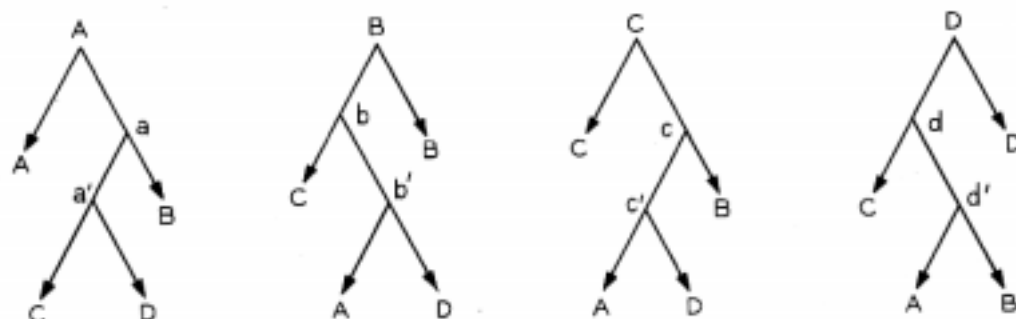


Fig. 7—State diagram for the decoder for the code of Table III.

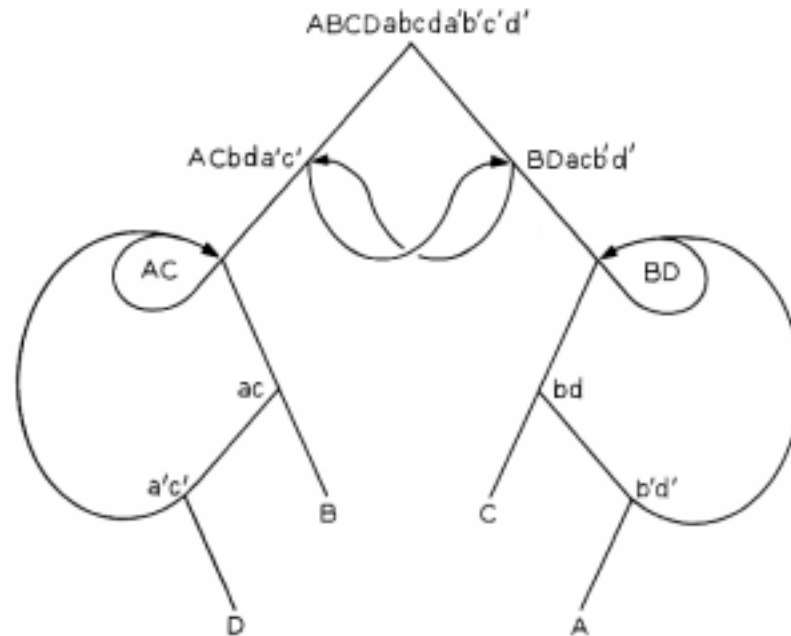


Fig. 8—Synchronizing diagram for the decoder of Fig. 7,  $J = 7.67$ ,  $J' = 3$ ,  $J'' = 4.67$ .

and as 011 following a "C"; in the last case, 011 must have been preceded by a "0".)

In the example, any sequence ending in 00 or 11 (cf. Fig. 8) guarantees the end of a code word, although not the code word for a known symbol. For purposes of this paper, synchronization to the end of some (unspecified) code word is called *first-stage* synchronization. Synchronization to the end of a code word corresponding to a particular symbol is called *second-stage* synchronization. The former is of concern in prefix codes, and both are of concern in sequential codes. In rare cases (particularly asymmetric ones), the second stage is achieved simultaneously with the first stage. In many useful cases, however, they may be treated independently. (In all but one example given in this paper, second-stage synchronization implies a particular known code word as well as a particular known symbol.)

[A word of caution is again needed regarding the randomness assumption. Again, the actual lag may be defined in terms of the actual probabilities. If synchronizing sequences occur naturally in code text, then the lag  $J$  is a fair estimate of the actual lag. If the sequences occur only as a result of unlikely sequences of symbols, then the lag  $J$  is smaller than the actual lag. However, in such cases the encoding can often be altered so that  $J$  is realistic. In general, it is desirable to have

codes with widely distributed likely synchronizing sequences, as in Fig. 8, rather than being totally dependent on a few obscure sequences.]

#### IV. CODING COMPLEXITY

It is clear that the choice of the code of Table II (with  $J = 7.67$ ) is (infinitely) superior to the code of Table I (with  $J = \infty$ ) in terms of resynchronizability. If  $W(k) = 0, 11, 100, 101$  in Table I for  $k = 0, 1, 2, 3$ , then the codes of Tables I and II are identical with respect to compression capabilities. The significant concern remaining is the computational complexity of the encoder and the decoder for the code of Table II. Intuitively, one might expect that an  $n$ -state sequential prefix code would be almost  $n$  times as complex as a difference code in terms of its encoding and decoding circuitry. Somewhat surprisingly, codes are developed below for which the complexity is essentially the same as the difference codes, while attaining good synchronizability and compression.

Examination of the code  $w(i|j)$  of Table II shows that the prefix code  $w(i|1)$  for state  $B$  ( $j = 1$ ) is the binary complement of the code  $w(i|0)$  for state  $A$  ( $j = 0$ ), cyclically shifted by one word:  $w(i|1) = w'(i - 1|0)$ . Further, the code  $w(i|2)$  for state  $C$  ( $j = 2$ ) is related to  $w(i|0)$ , having the same code words but with a different mapping. In particular,  $w(i|2) = w(2 - i|0)$ . Finally, the code for state  $D$  ( $j = 3$ ) is the shift of the complement of  $w(i|2)$ , or the complement of the shift, and thus  $w(i|3) = w'(3 - i|0)$ . Thus all four of the prefix codes are closely related to any one of them.

The code  $w(i|j)$  as a function of  $w(i|0)$  is summarized in Table III for each  $j$ . The structure of the code is somewhat more transparent when related to the difference code of Table I, with  $W(k) = 0, 11, 100, 101$  for  $k = 0, 1, 2, 3$  and  $k \equiv i - j \pmod{4}$ . In this case,  $w(i|j) = W(k), W'(k), W(-k), W'(-k)$  for  $j = 0, 1, 2, 3$ , respectively. The encoder and decoder for Table II are thus easily specified in terms of the difference code. If  $(p, q)$  is the binary representation of  $j$  as shown in Table III, then  $W(k)$  is replaced by  $W(-k)$  if  $p = 1$ , and the result is complemented if  $q = 1$ . The encoder for the difference code is shown in Fig. 9, while the encoder for the sequential prefix code of Table II is given in Fig. 10. (" $\Delta$ " represents a one-digit delay.) It is seen that an AND gate ( $\cdot$ ) and two EXCLUSIVE OR gates ( $\oplus$ ) represent the marginal cost of encoding the latter code, compared to the difference code. The same is true of the decoder, which employs precisely the same set of gates.

TABLE III—SUMMARY OF ENCODER AND DECODER FOR THE CODE OF TABLE II

$j$	$p$	$q$	Code $w(i j)$
0	0	0	$w(i 0) = w(i 0) = W(k)$
1	0	1	$w(i 1) = w'(i-1 0) = W'(k)$
2	1	0	$w(i 2) = w(2-i 0) = W(-k)$
3	1	1	$w(i 3) = w'(3-i 0) = W'(-k)$

## V. CONSTRUCTION OF GOOD SEQUENTIAL PREFIX CODES (WITH SMALL LAGS)

The synchronizing properties of sequential prefix codes fall into two classes, those which depend on the individual choice of the encoding  $w(i|j)$  for each  $i, j$ , and those (in varying degrees) which are independent of the actual choice of  $w(i|j)$ . [The encoding of Table I, for example, has  $J = \infty$  irrespective of the  $w(i|j)$ .] Such properties are said to be *choice-dependent* and *choice-independent* for these two classes. An example of how choice-independent properties may be treated separately is given by the following theorem.

5.1 Very Bad Codes ( $J = \infty$ )

*Theorem 1:* A sequential prefix code has  $J = \infty$  in each of the following cases: (a) if there are precisely  $n$  distinct code words among the  $n \times n$   $\{w(i|j)\}$ , each of which occurs exactly once for each  $i$  (a "Latin square" code); (b) if for some value of  $s$  ( $0 < s < n$ ) the relation  $w(i+s|j+s) = w(i|j)$  holds for all  $i$  and  $j$ .

*Proof:* Case (a). Consider an ambiguity between any two symbols which could have led to a given code word. Then any subsequent code word could have resulted from either of two distinct symbols. Thus ambiguity is never reduced, and  $J = \infty$  irrespective of the choice of the  $n$  independent  $w(i|j)$ . [Note that if the prefix code for any  $j$  (the same code words, but a different encoding for each  $j$ ) is self-synchronizing ( $I < \infty$ ), then it is possible to reduce ambiguity to the end of some (unknown) code word, but no further. Recall that in the definitions, a distinction is made between the code (the set of code words) and

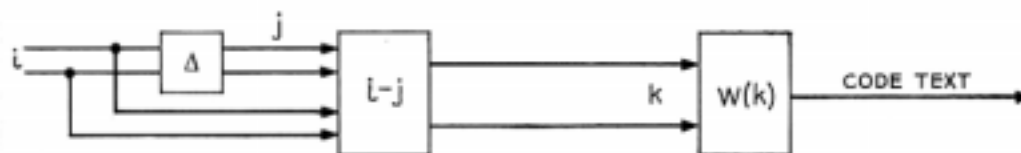


Fig. 9—Encoder for the difference code of Table I.



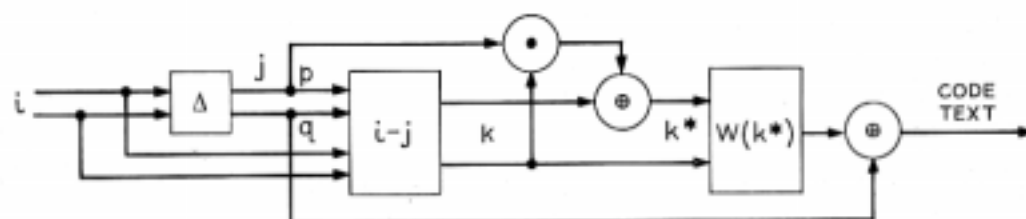


Fig. 10—Encoder for the sequential prefix code of Table II.

the encoding (the mapping between code words and source messages).]

Case (b). Without loss of generality, suppose that  $s$  is the smallest value of  $s$  for which  $w(i + s | j + s) = w(i | j)$  for all  $i$  and  $j$ . (If  $s$  is 1, the code is a difference code.) Then  $w(i + gs | j + gs) = w(i | j)$  for all  $i, j$ , for every integer  $g$ . Let  $u$  be the smallest value of  $g > 0$  for which  $gs \equiv 0 \pmod{n}$ . Then since  $s$  is as small as possible, it follows that  $su = n$ , i.e.,  $s$  divides  $n$ . Thus there are  $s$  prefix codes  $w(i | j)$ , e.g., for  $j = 0, 1, \dots, s - 1$ , which completely determine the prefix codes for  $j = s, s + 1, \dots, n - 1$ . That is, the same prefix code (shifted) occurs for each  $j = gs + h \pmod{n}$  for  $g = 0, 1, \dots, (n - s)/s$ , for any fixed value of  $h, 0 \leq h \leq s - 1$ . Thus there must always remain an ambiguity among all the symbols with  $i = gs + h \pmod{n}$  for  $g = 0, 1, \dots, (n - s)/s$ , for any particular value of  $h, 0 \leq h \leq s - 1$ . Therefore  $J$  is infinite again, irrespective of the choice of the  $sn$  independent  $w(i | j)$ . QED.

[Case (b) of Theorem 1 can easily be extended to include cases for which  $w(i + s | j + t) = w(i | j)$  for all  $i, j$ : if  $s$  and  $t$  are each relatively prime to  $n$ , or more generally if  $s$  and  $t$  have the same order in the field of integers {i.e., if  $u = v$ , where  $u$  and  $v$  are the smallest non-zero values for which  $us \equiv 0 \pmod{n}$  and  $vt \equiv 0 \pmod{n}$ }. As these cases are less natural to the present environment, they are mentioned parenthetically.]

The difference codes satisfy both cases (a) and (b). Another example of case (b) is given in Table IV. If  $e, f, g, h$  are replaced by  $b, c, d, a$ , respectively; this example also satisfies case (a), and is a "sum" code rather than a difference code.

## 5.2 Properties of Good Codes

It is highly desirable that sequential codes  $\{w(i | j)\}$  have considerable structure, in order to simplify encoding and decoding, to simplify the analysis of synchronizing properties, and to facilitate the construction of good large codes. Several intuitively evolved properties have

been examined which are found to contribute considerably to this desired structure. *Firstly*, to simplify encoding and decoding, the number  $m$  of distinct prefix codes (not counting complements) used to form a sequential code should be small (one or at most two). If a prefix code and its complement are both used, the sequential code is *complemented*. In the complemented code of Tables II and III,  $m$  is one since the prefix code for each  $j$  is  $W(k)$  or its complement. Since the circuitry required is up to  $m$  times as complex as when  $m = 1$ , large values of  $m$  are to be diligently avoided. The  $m$  distinct prefix codes (ignoring complements) are called *kernels*. The property of keeping  $m$  small is called the *symmetrization* property; it is choice-independent. *Secondly*, if all code words  $w(i | j)$  for a given  $i$  end in the same digit, irrespective of  $j$ , then first-stage synchronization is greatly enhanced. This property of making code-word endings uniform by column (as in Table II) is called the *columnization* property. (It is more or less choice-independent.) A third property involves the number of different symbols to which a given code word can correspond. (In the example of Table II, half of the code words occur only for one value of  $i$  each.) Second-stage synchronization is greatly aided by having almost all code words occur only for a relatively small number of different symbols, avoiding having each occurrence correspond to a different symbol  $S(i)$ . This is called the *association* property, and is also choice-independent.

It should be noted that none of the above properties is necessary for obtaining finite lag codes; however, these properties are found to be helpful in achieving low lags. Although the example of Table V is a (complemented) one-kernel code, it violates both the columnization property and the association property. (Note that each code word occurs only twice, but always for different symbols.) Its lag  $J$ , though finite, is quite horrendous (around 200), with the shortest synchronizing sequences being of length ten (e.g., 0101000100). It thus compares badly with the code of Table II with respect to its lag. (It even compares badly with the worst columnized one-kernel finite-lag code using the given prefix code, for which  $J = 23.1$ .)

TABLE IV—EXAMPLE OF A CODE WITH  $J = \infty$  BY THEOREM 1(b)  
 $n = 4, s = 2$

a	b	c	d
e	f	g	h
c	d	a	b
g	h	e	f

TABLE V—A BAD FINITE LAG CODE

0	11	100	101
1	00	011	010
101	100	11	0
010	011	00	1

An example of a two-kernel code which satisfies the columnization property and the association property is shown in Table VI. The code words which occur for only one value of  $i$  are indicated by asterisks. This code has  $J = 8.9$ , with synchronizing sequences including 0011 (for B) and 11100 (for C).

In a columnized code, the set of symbols  $S(i)$  for which all code words end in "0" ("1") is called the 0-set (1-set). In the example of Table VI, the sequence 00 (among others) guarantees the end of a code word corresponding to the 0-set A, C or E ( $i$  even), while a 111 guarantees the end of a code word corresponding to the 1-set B or D ( $i$  odd). These are two of the first-stage synchronizing sequences. Having reduced the ambiguity to a 0-set symbol or to a 1-set symbol, the association property within these sets is of great aid to second-stage synchronization. For example, the code words which optimally satisfy the association property (e.g., those with asterisks in Table VI) themselves act as second-stage synchronizing sequences in this example. Association is especially helpful to second-stage synchronization if the prefix code is the same for each symbol  $S(j)$  in the 0-set, and similarly for the 1-set. This is the *bifurcation* property of columnized codes, that the 0-set and the 1-set use one prefix code each (although the two codes may be identical). Note that this property is found in the code of Table VI, in which two distinct prefix codes are used. If present, the bifurcation property implies that the symmetrization property is met with  $m = 1$  or 2. (By definition, bifurcated codes must be columnized.)

An example of a one-kernel code satisfying all four of the above properties is given in Table VII. Apart from 00 and 01, no code word

TABLE VI—A TWO-KERNEL EXAMPLE,  $J = 8.9$ ,  $I' = 4$ ,  $J'' = 5.1$ 

$S(j)$	$S(i) = A$	B	C	D	E
A	0	11*	100	1011*	1010
B	010*	1	00*	0111	0110*
C	1010	11*	0	1011*	100
D	010*	0111	00*	1	0110*
E	1010	11*	100	1011*	0

TABLE VII—A ONE-KERNEL EXAMPLE,  $J = 20.2$ ,  $I = 8$ ,  $J'' = 14.7$ 

$S(j)$	$S(i)=A$	B	C	D	E	F	G	H
A, B	00	01	100	101	1100	1101	1110	1111
C, D	1100	1101	00	01	100	101	1110	1111
E, F	1100	1101	100	101	00	01	1110	1111
G, H	1100	1101	1110	1111	100	101	00	01

occurs for more than two symbols  $S(i)$  (i.e., in more than two columns). The lag is seen to be  $J = 20.2$ .

### 5.3 Balanced Codes

If the columnization property is to be achieved in a complemented sequential code, then any kernel prefix code and its complement must each have the same number of code words ending in zero (or one). Consequently, the prefix code (and its complement) must have half of its code words ending in each digit. Such a prefix code is called a *balanced* code, and of course must have an even number of code words. The codes of Tables II and VII are examples of balanced prefix codes used in complemented and uncomplemented one-kernel codes, respectively.

*Theorem 2: For every set of code-word lengths with  $n$  even for which there exists an exhaustive prefix code, there exists at least one balanced prefix code.*

*Proof:* A simple proof involves a construction procedure during which the number of code words ending in "0" differs by at most one from the number of code words ending in "1". Thus since  $n$  is even, the resulting code is balanced. Such a procedure is easy to construct, but is omitted here since it does not contribute to a basic understanding of the paper. [It can in fact be shown that the ratio of balanced exhaustive codes to all exhaustive codes is asymptotic for large  $n$  to  $1/(n\pi)^{1/2}$ .]

A few balanced exhaustive codes are shown in Fig. 11, including one for each structure function  $N(d)$  with  $n = 4$  and 6. In each of these cases, the code shown has the smallest possible lag  $I$  among all balanced codes with the given  $N(d)$ .

If the sequential code is not complemented, the kernels need not be balanced. However, the use of balanced prefix codes as kernels is highly beneficial. It greatly enhances flexibility in the assignment of the  $w(i|j)$  according to the needs of synchronization (e.g., via columnization and association), coding complexity and compression.

CODE	$n = 4$		$n = 6$				
$N(d)$	0 4	1 1 2	1 1 1 1 2	1 0 3 2	1 1 0 4	0 2 4	0 3 1 2
$I$	$\infty$	6	5, 2	5, 3	7	10	12
$I'$	$\infty$	3	3	6, 1	5, 5	10	11
$I''$	$\infty$	5	3 1	6, 3	5, 9	10	13
$I^*$	$\infty$	2	2	5, 1	4, 5	9	10

Fig. 11—Some balanced codes for  $n \leq 6$ .

#### 5.4 Analysis of First-Stage Synchronization

Another property emerges from considering the relation between the lag  $J$  of a bifurcated sequential code and the lags of the kernel codes. For an uncomplemented one-kernel code, first-stage synchronization is guaranteed by the synchronizing sequences of the kernel, i.e., with the lag  $I$  of the kernel. Thus this lag should clearly be small. On the other hand, for a bifurcated complemented one-kernel code, the lag  $I$  is not relevant to first-stage synchronization. Instead the mutual lag  $I'$  of a prefix code and its complement is needed. The *mutual lag* of a 0-set code and a 1-set code is obtained by considering the state diagrams of both codes. Imposing the restriction that a code word ending in "0" ("1") is followed by a 0-set (1-set) code word, these two state diagrams become one just as the four diagrams in Fig. 7 become one. The mutual lag  $I'$  is then the lag of this combined state diagram, obtained from the *mutual synchronizing diagram*, i.e., the synchronizing diagram for the combined state diagram. Terminal nodes consist of first-stage synchronization, i.e., solely of 0-set or 1-set code-word ends. (Note that the mutual lag is partially choice-independent, depending on the choice of the 0-set and 1-set prefix codes, but not on the actual  $w(i|j)$ .) Mutual lags are relevant primarily for balanced prefix codes and their complements, as used in (one-kernel) bifurcated complemented codes; this case is assumed unless otherwise specified. They are also meaningful for two-kernel bifurcated codes, for which the mutual synchronization diagram also guarantees first-state synchronization. (For example, the two prefix codes used in Table VI have  $I' = 4$ .) Note that the mutual lag of a code with itself is  $I' = I$  (since the 0-set code and the 1-set code are identical).

Consider as an example the prefix code  $\{0, 11, 100, 101\}$  used in

Table II. Suppose that it is used for the 0-set symbols A and C (Fig. 12a), while its complement is used for the 1-set symbols B and D (Fig. 12b). The state 0 (1) corresponds to a code word ending in "0" ("1"), and implies that the next code word is taken from the 0-set (1-set) code. (Note that irrespective of the actual  $w(i|j)$ , the occurrence of a 00 or a 11 suffices to guarantee the end of a code word, as in Fig. 8.) The mutual synchronization diagram is given in Fig. 12c, and has  $I' = 3$ .

Values of the mutual lag  $I'$  are given in Fig. 11 for each code shown there and its complement. In each case, the code shown has the smallest value of  $I'$  for any balanced code with the given set of code-word lengths  $N(d)$ . A *numerological curiosity* is provided by Fig. 11: for each  $N(d)$ , the indicated best value of  $I'$  is precisely one greater than the best value  $I^*$  of  $I$  attainable by any code (unbalanced, in fact) with the given  $N(d)$ . Since this curiosity is true of all  $N(d)$  for exhaustive codes with  $n \leq 6$ , including those for odd  $n$ , it seems highly likely for all  $n$ , for all  $N(d)$  for which exhaustive codes exist.

[It should be noted that the value of  $I'$  for a given prefix code and its complement is obtained with the given code as the 0-set code. If it is used instead as the 1-set code, the resulting value of  $I'$  is the mutual lag of the complemented prefix code, and is designated by  $I$ . Values of  $I$  are also shown in Fig. 11. It is seen that  $I'$  and  $I$  are not usually the same.]

The *kernel lag*  $J'$  of a bifurcated code is then defined as the synchronization lag of the first stage. For a one-kernel uncomplemented code,  $J' = I$ ; for a two-kernel (uncomplemented) code, or a one-kernel complemented code,  $J' = I'$ . The *kernel lag property* then states that the kernel lag  $J'$  of a bifurcated code should be small, in order to help minimize the lag  $J$ .

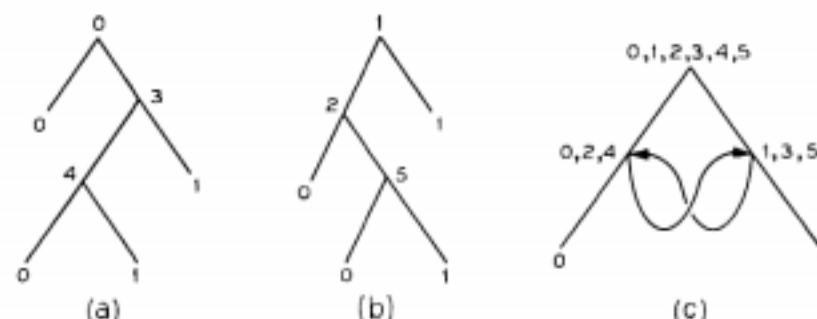


Fig. 12—Example of first-stage synchronization in a complemented code with  $I' = 3$ .

### 5.5 Further Properties of Bifurcated Codes

The second-stage lag may be approximated by the *structural lag*  $J''$  which assumes first-stage synchronization, and results in a known symbol (at the end of a code word). It is obtained by averaging the second-stage lags,  $K$  beginning with the 0-set and  $K'$  beginning with the 1-set;  $K$  and  $K'$  are weighted according to the probabilities of reaching the 0-set and 1-set, respectively, in the mutual synchronization diagram. (In the example of Table VI, for example, it is seen that these weights are  $\frac{2}{3}$  and  $\frac{1}{3}$ .) The *structural lag property* states that this lag should be as small as possible. A rough measure of  $J''$  may be made independent of the prefix code by assuming all code words of equal length, and the 0-set and 1-set equiprobable.

*Theorem 3: For a bifurcated code,*

$$J \leq J' + J''.$$

*Proof:* The theorem follows from the definitions. Inequality occurs when, in the sequential synchronization diagram, first-stage synchronization occurs in at least one case as the result of an advantageous subset of either the 0-set or the 1-set. By "advantageous" is meant a subset which accelerates second-stage synchronization. The value of  $J'$  gives precisely the first-stage synchronization lag in any event. When there are no such subsets, equality holds, as in Fig. 8. (For example, the code of Table VI has  $J' = 4$ ,  $J'' = 5.1$  and  $J = 8.9$ ; the sequence 1100 guarantees not just the 0-set, but specifically a "C" or an "E" at its end. The code of Table VII also has such subsets. These subsets are obtainable from the first-stage synchronization diagram, and may be used to calculate the exact second-stage lag  $J^-$  based on the subsets with their appropriate weights, rather than just on the 0-set and the 1-set. Then  $J = J' + J^-$ . In all cases  $J'' \geq J^-$ . Also, if  $J''$  is infinite, then so is  $J^-$ .)

*Theorem 4: A bifurcated code has  $J = \infty$  if and only if either  $J'$  or  $J''$  is infinite.*

*Proof:* The "only if" follows as a corollary of Theorem 3. The "if" requires two cases. If  $J'' = \infty$ , then  $J = \infty$  since the 0-set and 1-set are subsets of the set of all states, on which  $J$  is based. If  $J' = \infty$ , first-stage synchronization is never reached. Since a bifurcated code must go through this stage in order to reach second-stage synchronization, and since  $J'$  is the true value of first-stage lag even when there is inequality in Theorem 3, it follows that  $J = \infty$ . QED. (Note that in Theorem 4  $J''$  may be replaced by  $J^-$ .)



In many symmetric cases, the 0-set and the 1-set are equivalent structurally, and  $K = K' = J''$ . This occurs in the example of Table II, since the code obtained under the transformation  $A \leftrightarrow D, B \leftrightarrow C$  is the complement of the Table II code (a complementary reflective symmetry). It also occurs in Table VII, since the 0-set and 1-set symbols are paired with identical encodings. Such symmetric cases are advantageous for encoding/decoding and analytic simplicity.

In certain cases it is desirable to use complemented bifurcated codes. If the above numerological curiosity is true in general, there is essentially no sacrifice in the best  $I'$  of a balanced code compared to the best  $I = I^*$  for the given  $N(d)$ . That is, first-stage synchronization is just as rapid for these complemented codes, while avoiding the difficulties arising from the use of an unbalanced code. Further, considering only balanced codes for a given  $N(d)$ , the best value of  $I'$  is often better than the best value of  $I$  (cf. Fig. 11). Thus the overall lag is frequently better. Furthermore, the flexibility of compression available with the complemented codes is often greater. For example, consider again the code of Table II, this time only in terms of its code-word lengths. Using an exhaustive prefix code, it is impossible to have length one on the  $i = j$  diagonal with an uncomplemented one-kernel code unless the columnization property is violated; this in turn may greatly increase the lag of the code.

[Surprisingly, prefix codes with  $I = \infty$  are not altogether useless. If a one-kernel code uses a block code or a code the greatest common divisor (gcd) of whose lengths is greater than one, then it follows that  $J' = J = \infty$ . It is interesting to note, however, that many uniformly composed codes and anagrammatic codes (with  $I = \infty$ ) have  $I' < \infty$ , and thus may give rise to one-kernel codes with  $J < \infty$ , assuming  $J' < \infty$ . The smallest possible finite exhaustive anagrammatic code with  $I' = I = \infty$  has  $n = 18$ , and is its own complement. An infinite example with the same properties is provided by the self-complementing anagrammatic code of Fig. 6c. Thus the use of such prefix codes in a one-kernel sequential code must result in  $J = \infty$ , by Theorem 4.]

#### VI. GOOD STRUCTURE FOR LARGE SEQUENTIAL CODES

The codes of Tables II and VII are rather small examples, albeit good ones, of one-kernel codes. Since great compressions are found primarily in large codes, the next question is whether low lags can be achieved for large codes without sacrificing coding simplicity and compression.

Assume for now that it is possible to approximate the second-order probabilities associated with the code words  $w(i | j)$  as a function of  $k = i - j \pmod{n}$ , and hence that from a compression point of view a difference code is meaningful. (This assumption will be generalized below.) Now consider the general  $n$ -level one-kernel encoding framework shown in Fig. 13. The difference  $k$  is modified as a function of  $j$ , and the result  $k^*$  is encoded by an ordinary prefix encoding  $W(k^*)$ . If the code is a complemented code, the result is complemented (e.g., if  $j$  is odd). (Otherwise the complementing circuitry is not needed, as in Table VII.) The corresponding decoder is shown in Fig. 14. (Again the complementing circuitry may not be required.) In many cases the demodification circuitry of Fig. 14 is identical to the modification circuitry of Fig. 13, with the input and output interchanged. Note that the codes of Tables II (see Figs. 9 and 10) and VII are examples of codes amendable to this framework. The incremental cost of encoding and decoding compared to difference codes is embodied in the modification logic and the trivial complementing logic; as long as the modification logic can be kept simple, the incremental cost is low. The compression can in general be made at least as good as the difference code.

If a balanced code is used, the framework defined by Fig. 13 makes it easy to satisfy the columnization property, the symmetrization property (with  $m = 1$ ), and the bifurcation property. The choice of a prefix code is dictated by the kernel lag property, and by the conditional probability distribution of the symbols  $S(i)$ , given  $S(j)$ . The suitable structure is dictated by those probabilities, mitigated by the structural lag property, and by the complexity of the modification logic desired. Considerable experimentation has shown that the properties discussed here are central to the construction of good codes. Columnization and the choice of prefix codes with small  $J'$  greatly facilitate first-stage synchronization. Association greatly influences second-stage synchronization. Bifurcation greatly simplifies coding complexity and improves second stage synchronization. The use of balanced codes is

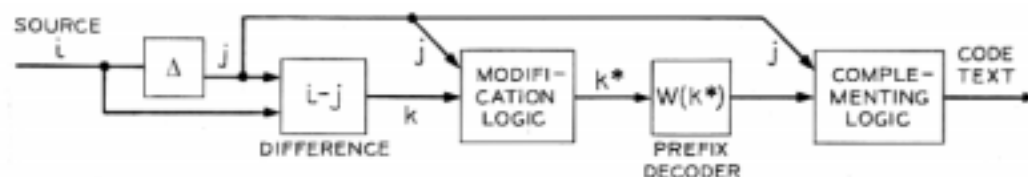


Fig. 13—Generalized encoder for one-kernel codes.

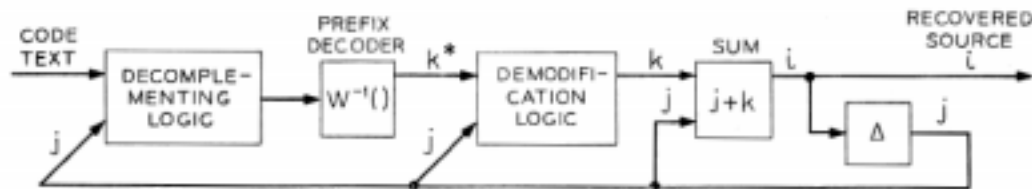


Fig. 14—Generalized decoder for one-kernel codes.

very helpful. If the number  $n$  of messages is odd, however, it may be desirable to use a two-kernel code as in Table VI to achieve columnization.

If the conditional probabilities are strongly asymmetric and/or not approximately representable by difference probabilities, it may be advantageous from a compression standpoint to use a multi-kernel code. Alternatively, or additionally, it may be desirable to eliminate the difference/sum circuitry in Figs. 13 and 14, and to deal directly with the  $i$  and  $j$ . The properties described in this paper are equally relevant in such cases. No restrictive assumptions have been made regarding the first-order probabilities. It is also unnecessary for 0-set and 1-set code words to match even and odd  $i$ , respectively; this is merely a descriptive convenience.

### 6.1 Large Balanced Codes

In order to enhance the kernel lag property for codes with large  $n$ , it may be desirable to use truncated systematic prefix codes rather than optimal prefix codes as kernels. In this way it is possible for  $I$  to remain small as  $n$  increases. Several examples of such codes which may easily be truncated to give balanced codes are summarized in Fig. 15, along with their structure functions  $N(d)$  and their lags  $I$ ,  $I'$  and  $I''$ . The best lag  $I^*$  for any code with the specified  $N(d)$  is also given, along with the random average code-word length  $L$  for  $N(d)$ . The selection of efficient codes for compression purposes is considered in Ref. 1.

There exist many classes of codes for which  $J$  remains finite (and in fact quite small) as  $n$  increases without limit. A simple (rather extreme) example is indicated in Table VIII for  $n = 8$ . The code is columnized, complemented, bifurcated, and maximally associated without being trivial. (It assumes very high probability of  $i = j$  for compression purposes.) To avoid confusion, the symbols are given as  $A$  to  $H$  for  $i$  and  $j$  from 0 to 7; the integer  $k$  is given to indicate the occurrence of the code word  $W(k)$ . If  $j$  is odd, the complementary code word







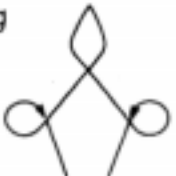
CODE	I	I'	I'	I*	L	N(d) FOR d =										
						1	2	3	4	5	6	7	8	9	...	
a 	4	3	∞	2	2	1	1	1	1	1	1	1	1	1	...	
b 	5	5	5	4	3	0	2	2	2	2	2	2	2	2	...	
c 	6	5	8	4	4	0	1	2	3	4	5	6	7	8	...	
d 	8	-	-	8	8	0	0	2	2	4	4	8	12	24	...	
e 	9	9	9	8	7	0	0	2	2	4	6	10	16	26	...	
f 	8	9	10.5	≤8	6	0	0	1	3	6	10	15	21	28	...	
g 	8.5	8.5	8.5	≤7.5	4	0	0	4	4	4	4	4	4	4	...	

Fig. 15—Some systematic prefix codes easily truncated to balanced codes.

$W'(k)$  is used, and the table is read from the bottom up. The code thus has complementary reflective symmetry. If  $W(k) = 0, 11, 100, 1011, 10100, 101011, 1010100, 1010101$  for  $k = 0, 1, \dots, 7$ , then  $J = 8.62$ . This code is readily extended to arbitrary even  $n$  with a similar pattern of  $k$ 's: for each even  $j$  other than 0,  $k = 0$  for  $i = j$ ,  $k = j$  for  $i = 0$ ; otherwise  $k = i$ ; code words for odd  $j$  are then specified by the complementary reflective symmetry. The modification

logic of Figs. 13 and 14 is thus exceedingly simple, and the complementing logic is again a single EXCLUSIVE OR gate as in Fig. 10. The limiting prefix code is code (a) of Fig. 15. Analysis shows that a sequential code with  $J' = I' = 3$ ,  $J'' = 4[1 + \frac{1}{3} + \frac{1}{15} + \frac{1}{63} + \frac{1}{255} + \dots] < 5.7$ ,  $J < 8.7$  exists for every even  $n$ . (For suitably skewed distributions, the compression factor arising from this code approaches the base two logarithm of  $n$ .) This is an example of a class of codes for which the differential circuitry is not relevant; it is easier to encode directly as  $k = i$ , with  $(n - 1)/2$  pairs of exceptions. Variants for which the 1-set is treated in this fashion while the 0-set is treated differentially as before are also easily implemented. These variants provide a very simple structure which results in low  $J$  and simple circuitry.

[A word of caution is in order when considering sequences of exhaustive prefix codes approaching systematic codes in the limit: the limit of a sequence of values of  $I$  (or  $I'$ ) thus obtained is normally somewhat larger than the value of  $I$  (or  $I'$ ) for the limiting systematic code. (The previous example is an exception with respect to  $I'$ .) Code (b) of Fig. 15 is an example in point, with  $I = I' = 5$ . The limit of the value of  $I$  (or  $I'$ ) for the codes 01, 10, 001, 110, 000, 111 ( $I = I' = 10$ ), 01, 10, 001, 110, 0001, 1110, 0000, 1111, ( $I = I' = 8$ ), etc., is seven, although this sequence approaches the systematic code (with  $I = I' = 5$ ) in the limit. In general, truncated systematic codes have better synchronization properties than their finite exhaustive approximations, since the values of  $I$  and  $I'$  are essentially unaffected by truncation. For large codes, there is little if any noticeable degradation in compression caused by using truncated infinite codes.]

The use of definite codes is suggested since their lags are minimal ( $I = L$ ), while  $I > L$  for non-definite codes. However, non-definite codes can be quite good. Codes (a) and (c) in Fig. 15, for example,

TABLE VIII—EXAMPLE OF A GOOD CODE FROM AN INFINITE CLASS OF CODES FOR WHICH  $J < 8.7$  FOR ALL  $n$

$j$ even	$k$ for desired $W(k)$								$H=S(i)$
	A	B	C	D	E	F	G		
$j$ : A	0	1	2	3	4	5	6	7	H
C	2	1	0	3	4	5	6	7	F
E	4	1	2	3	0	5	6	7	D
G	6	1	2	3	4	5	0	7	B: $j$
$S(i) =$	H	G	F	E	D	C	B	A	$j$ odd
	$k$ for desired $W'(k)$								

have  $N(d)$  for which definite codes exist (whence  $I^* = L$ ). In these cases,  $I' = L + 1$ , recalling the above numerological curiosity. No counterexample to this curiosity is known among  $N(d)$  for which definite codes exist, or even among all systematic codes. In general, there are many non-definite cases for which  $I$  or  $I'$  is quite close to  $L + 1$ . Thus in these cases the non-definite systematic codes do almost as well as comparable definite codes in achieving first-stage synchronization. Subsequently the non-definite codes may be much better in reaching the second stage. Definite codes are necessarily completely unbalanced,<sup>1,2</sup> with all code words ending in the same digit; therefore a complemented columnized code is impossible. Although a non-complemented code is thus automatically columnized if it uses a definite prefix code as its kernel, second-stage synchronization often must begin with the set of symbols, not just half of them as in the case of a balanced code. However, some definite codes are balanced on two distinct terminal sequences, e.g., on the next to last digit of each code word or on some earlier digit position. In these cases, it is possible to columnize on the two distinct terminal sequences. As an example, consider code (d) of Fig. 15. This is a definite code defined by the set of sequences  $\{0100, 0110\}$ , with  $N(d) = 0 \ 0 \ 2 \ 2 \ 4 \ 4 \ 8 \ 12 \ 24 \ \dots$ . For each length there are exactly as many code words ending in 00 as in 10. If the sequential code is columnized according to the last two digits of each code word into a 00-set and a 10-set, first-stage synchronization results in one of these two sets, as in the balanced code situation. Since  $I = L = 8$ , this code has synchronization properties excelling any non-definite code with the given  $N(d)$ .

### 6.2 Guidelines for Choosing Good Encodings

Because of the perversity of the three-dimensional tradeoffs among compression, complexity and synchronizability, it is pointless to try to give a specific algorithm for choosing the best encoding for a given application. Optimality in any one dimension is of little concern, for slight sacrifices in any of these dimensions often result in great savings in the other two. Besides, no sensible cost metric is known. Nevertheless, the techniques of this paper provide a set of guidelines for the construction of good codes and good encodings.

The first step in selecting a sequential encoding is to establish for each  $j$  the code-word lengths which are optimal for the code word  $w(i | j)$  in the prefix code for the given  $j$ , based on the conditional probabilities of  $S(i)$ , given  $S(j)$ . This may be done simply using any variant of the Huffman algorithm<sup>6</sup> which derives the code-word lengths. In-

spection of the matrix of lengths thus obtained (i.e., of the set of  $n$  structure functions, one for each  $j$ ) indicates what kind of symmetries the code might reasonably have, e.g., whether a reflective symmetry is in order, and whether the code can be a one-kernel code. A complemented bifurcated code may be required for compression reasons, as in Table II. The next step is to choose the basis code(s), considering the set of lengths and the first-stage lag  $J'$ . For large codes, techniques of Ref. 1 may be required to aid code selection. The encodings should then be arranged to have the columnization and association properties to help minimize the first- and second-stage lags, respectively. The desired code-word lengths should be taken as suggestive rather than as mandatory; slight departures from these lengths are generally not harmful to compression (especially among large lengths), and may help greatly in decreasing  $J'$ . Care should be taken to avoid having short synchronizing sequences occur only as the result of unlikely sequences of source messages.

#### VII. CONCLUSIONS

The object of this paper is to present codes with low redundancy, reasonable complexity and intrinsic error tolerance, within a single class of codes designed for that purpose. The approach taken is by no means the only one, although the codes exhibited here seem quite powerful in view of their capabilities. In combination with some additional redundancy (e.g., as in Refs. 1, 4, 17 and 18) to be used for error-detection and/or correction and/or forced framing, the intrinsic properties described here may be used to great advantage.

The sequential prefix codes of this paper can be useful for a wide range of conditional probability distributions. Quantized video picture information provides an example of a class of distributions<sup>19-21</sup> to which these codes seem suited, with respect to compression and synchronizability, as well as to the meaningfulness of tolerating errors for a short period of time. Such distributions are strongly geometric,<sup>20</sup> for which the codes presented here are naturally applicable.

The techniques described here are also applicable to other compression situations. Examples include run-length coding (which has the same essential synchronization problem as differential coding) and predictive, averaging or smoothing schemes, or combinations thereof. In addition, the same techniques are applicable when code words  $w(i|j)$  need not be provided for many of the transitions, for example, when only relatively small differences are possible.



In conclusion, a wide variety of sequential prefix codes has been presented, with a considerable range of tradeoffs among coding complexity, compression and synchronizability. By not insisting on optimality in any of these, it is possible to obtain codes which are highly satisfactory in all of these respects at the same time.

#### VIII. ACKNOWLEDGMENT

The author is indebted to E. N. Gilbert and J. O. Limb for helpful comments on the manuscript.

#### REFERENCES

1. Neumann, P. G., "Efficient Error-Limiting Variable-Length Codes," IRE Trans. Inform. Theory, *IT-8*, (July 1962), pp. 292-304.
2. Neumann, P. G., "On a Class of Efficient Error-Limiting Variable-Length Codes," IRE Trans. Inform. Theory, *IT-8*, (September 1962), pp. S260-266.
3. Neumann, P. G., "Error-Limiting Coding Using Information-Lossless Sequential Machines," IEEE Trans. Inform. Theory, *IT-10*, (April 1964), pp. 108-115.
4. Neumann, P. G., "On Error-Limiting Variable-Length Codes," IEEE Trans. Inform. Theory, *IT-9*, (July 1963), p. 209.
5. Chow, M. C., "Shannon-Fano Encoding for Differentially Coded Video Telephone Signals," Mervin J. Kelly Communication Conference, University of Missouri, Rolla, October 5-7, 1970.
6. Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," PROC IRE, *40*, (September 1952), pp. 1098-1101.
7. Gilbert, E. N., and Moore, E. F., "Variable-Length Binary Encodings," B.S.T.J., *38*, No. 4 (July 1959), pp. 933-967.
8. Schützenberger, M. P., "On an Application of Semi-Group Methods to Some Problems in Coding," IRE Trans. Inform. Theory, *IT-2*, (September 1956), pp. 47-60.
9. Gill, A., *Introduction to the Theory of Finite-State Machines*, New York: McGraw-Hill Book Co., (1962).
10. Hennie, F., *Finite State Models for Sequential Machines*, John Wiley and Sons, (1968).
11. Winograd, S., "Input-Error-Limiting Automata," J. Assn. Computing Machinery, *XI*, (1964), pp. 338-351.
12. Mandelbrot, B., "On Recurrent Noise Limiting Coding," Proc. Symp. on Information Networks, Polytechnic Inst. of Brooklyn, (April 1954), pp. 205-221.
13. Schützenberger, M. P., "On Synchronizing Prefix Codes," Information and Control, *7*, (1964), pp. 23-26.
14. Laemmel, A. E. and Rudner, B., "Study of the Application of Coding Theory," Internal Report PIBEP-69-034 (AD 691 764), Polytechnic Inst. of Brooklyn (June 1969).
15. Fischler, M. A., and Tannenbaum, M., "Synchronizing and Representation Problems for Sequential Machines with Masked Outputs," Eleventh Annual Symposium on Switching and Automata Theory, Santa Monica, California, October 1970.
16. Huffman, D. A., "Canonical Forms for Information-Lossless Finite-State Logical Machines," IRE Trans., *CT-6* or *IT-5* (Supplement) (May 1959), pp. 41-59. Also appears in E. F. Moore, *Sequential Machine Papers*, New York: Addison-Wesley (1964).
17. Hatcher, T. R., "On a Family of Error-Correcting and Synchronizable

- Codes," IEEE Trans. Inform. Theory, *IT-15*, (September 1969), pp. 620-624.
18. Scholtz, R. A., and Welch, L. R., "Mechanization of Codes with Bounded Synchronization Delays," IEEE Trans. Inform. Theory, *IT-16*, (July 1970), pp. 438-446.
  19. Oliver, B. M., "Efficient Coding," B.S.T.J., *31*, No. 4 (July 1952), pp. 724-750.
  20. Schreiber, W. F., "The Measurement of Third Order Probability Distributions of Television Signals," IRE Trans. Inform. Theory, *IT-2*, (September 1956), pp. 94-105.
  21. Graham, R. E., "Predictive Quantizing of Television Signals," IRE WESCON, Part 4, (August 1958), pp. 147-157.

