Designing Delay Insensitive Error-Correcting Unordered Codes for Robust Asynchronous Global Communication

D.RAJASEKHAR¹, C.HEMA SUNDER RAO², MOHD.SHAHBAZ KHAN³

¹M.Tech, VLSI Dept, CMRIT, Hyderabad, AP-India, Email: rajasekharg893@gmail.com,
²Prof, ECE Dept, CMRIT, Hyderabad, AP-India, E-mail: c.hemasunderkondu446@gmail.com,
³Assoc Prof, ECE Dept, CMRIT, Hyderabad, AP-India, E-mail: shahbazkhan434@gmail.com

Abstract: This paper introduces a new family of error correction unordered (ECU) codes for global communication, called Zero-Sum. They combine the timing-robustness of delay insensitive (i.e., unordered) codes with the fault-tolerance of error-correcting codes (providing 1-bit error correction or 2-bit detection). Two key features of the codes are that they are systematic, allowing direct extraction of data, and weighted, where the check field is computed as the sum of data index weights. A wide variety of weight assignments is shown to be feasible. Two practical enhancements are also proposed. The Zero-Sum+ code extends error detection to 3-bit errors, or alternatively handles 2-bit detection and 1-bit correction. The Zero-Sum* code supports heuristic 2-bit correction, while still guaranteeing 2-bit detection, under different strategies of weight assignment. Detailed hardware implementations of the supporting components (encoder, completion detection, error corrector) are given, as well as an outline of the system micro architecture. In comparison to the best alternative systematic ECU code, the basic Zero-Sum code provided better or comparable coding efficiency, with a 5.74%–18.18% reduction in average number of wire transitions for most field sizes. Several Zero-Sum* codes were also evaluated for their 2-bit error correction coverage; initial results are promising, where the best strategy corrected 52.92%–71.16% of all 2-bit errors for most field sizes, with only a moderate decrease in coding efficiency and increase in wire transitions. Technology mapped pre-layout implementations of the supporting Zero-Sum code hardware were synthesized with the tool using an industrial standard cell library. Results indicate that they have moderate area and 500ns delay overheads. In comparison, supporting hardware for the best nonsystematic ECU codes has 3.82–10.44× greater area for larger field sizes.

Keywords: Asynchronous design, delay-insensitive, error-correcting codes, fault-tolerance, unordered.

I. INTRODUCTION

As Digital systems grow in complexity, the challenges of design reuse, scalability, power, and reliability continue to grow at a rapid pace [5]. These parameters are expected to become major bottlenecks in less than a decade. One Promising direction of research has been to explore the use of asynchronous global communication [6], [13], [14], [17], to provide flexibility in system integration as well as demand-driven operation where dynamic power is determined solely by the current traffic. Such systems can be entirely asynchronous, or use a hybrid combination of synchronous computation blocks interconnected by asynchronous channels, thus forming a globally-asynchronous locally-synchronous (GALS) system [19], [4]. Some recent applications of asynchronous global communication, using delay-insensitive (DI) codes [7], include high-speed commercial field programmable gate array’s (Achronix) [45] and Ethernet routing chips (Fulcrum Microsystems) [29], as well as experimental chips for massively parallel neural simulation [18], [20].

All proposed Zero-Sum codes are systematic [10], [17], where data appears in unaltered form in each codeword and can be directly extracted by the receiver without any decoding hardware. They are also weighted, where the check field is computed as the sum of data index weights. We target two primary costs: average number of wire transitions per transaction (i.e., a rough metric for dynamic power [12]) and coding efficiency (i.e., number of bits per wire). In many recent applications, for both on-
chip and off-chip global communication, these metrics are regarded as critical contributors to overall system cost, including for dynamic and static power, chip area, and routing complexity [23], [35], [42]. In contrast, the design of processing or router nodes, which are interconnected by the global communication network, is an orthogonal issue [28], where the local overheads of their channel interfaces (e.g., encoders/decoders, error-correction units) play a lesser though still important role in system overhead. Hence, while we also consider hardware overhead as a secondary cost, the primary focus is on switching activity and coding efficiency of the communication channels.

The key contributions of this paper are as follows.

1) Defining the complete family of Zero-Sum codes, by exploring the feasible space of index weight assignments and permutations.

2) A new code, called Zero-Sum+, which guarantees detection of up to 3-bit errors, or alternatively can simultaneously support 1-bit correction and 2-bit detection. These codes can also detect all odd numbers of errors.

3) A new class of codes, called Zero-Sum*, which extends Zero-Sum+ through a variety of weight assignment strategies, to heuristically provide a high coverage for 2-bit correction.

4) The detailed design and implementation of three key hardware support blocks (e.g., encoder, completion detector, and error correcting unit) for the Zero-Sum code.

5) An outline of the overall micro architecture and system level asynchronous communication protocol.

6) Three types of code evaluation: a) a detailed code evaluation of the Zero-Sum code for the two key cost metrics of coding efficiency and wire transitions per transaction; b) an analytical evaluation of the Zero-Sum+

code, for the same metrics; c) an analytical evaluation of the Zero-Sum* code, using three alternative strategies for assigning the index weights.

7) A detailed comparative evaluation of the Zero-Sum hardware support.

II. Background and Related Work

Background and related work on asynchronous communication and error correction are now briefly reviewed. A brief review of the Zero-Sum code is also presented.

A. Point-to-Point Asynchronous Communication

The proposed methods assume point-to-point communication [6], [30], [34], [47] between a sender and a receiver.

1) Asynchronous Communication Channels: An asynchronous communication channel [47] is the means by which information is transmitted. Fig. 1(a) gives an example of point-to-point communication. Abstractly, the sender provides a request output signal (REQ) to the receiver; the receiver in turn provides an acknowledgment input signal (ACK) to the sender. If the sender passes actual data to the receiver (rather than providing simple control synchronization), the REQ is typically replaced by the encoded data, as shown in the figure. A codeword is placed by the sender on the data channel (where each wire corresponds to a bit of the codeword), and passed to the receiver. The ACK indicates that data has been received by the receiver and new data can eventually be sent [47].

2) Four-Phase Communication Protocol: Given an asynchronous communication channel, a protocol is needed to transfer information from sender to receiver. The most widely used protocol is four-phase or return-to-zero (RZ) [6], [20], [47]. As illustrated in Fig. 1(b), the protocol has two operations: a) evaluate, and b) reset. During the evaluate operation, the sender first indicates the start of an event by issuing a rising REQ+ to the receiver. Once this signal has been received, the receiver asserts an ACK+. At this point, the reset operation begins. The sender de-asserts the REQ− and in turn, the receiver de-asserts its ACK− which is the final event of the reset stage and the four-phase transaction. If data is used, the data channel replaces the REQ wire. The channel is initially all-0 (spacer state); data is transmitted by asserting high the individual 1 bit of the codeword (equivalent to REQ+); data is reset after receiving ACK− by de-asserting low the individual 1 bit (equivalent to
Designing Delay Insensitive Error-Correcting Unordered Codes for Robust Asynchronous Global Communication

REQ–). An alternative to the RZ protocol is a two-phase or non-return-to-zero (NRZ) protocol [22], [30], which avoids the reset phase; however, codes designed for an NRZ protocol (e.g., level-encoded dual-rail [22]) typically have large overheads and require complex circuit structures. Therefore, a four-phase asynchronous communication protocol is assumed in this paper.

B. DI Codes

When asynchronous communication is used, data must be suitably encoded so that the receiver can identify when a packet has been received. DI codes [4], [39], [47] (i.e., unordered codes [9], [11], [15]) are insensitive to propagation delays on individual bits in a codeword. The key property is that no valid codeword is covered by another (i.e., no valid codeword is ever a proper subset of another valid code word).

Definition 1 (Unordered Code [14]): A codeword \( X = x_1 x_2 \ldots x_n \) covers another codeword \( Y = y_1 y_2 \ldots y_n \) if and only if, for each bit position \( i \), if \( y_i = 1 \) then \( x_i = 1 \). In these cases, \( Y \) is covered by \( X \), or \( Y \leq X \). Code words \( X \) and \( Y \) are unordered if \( X \) and \( Y \). A code, \( C \), is unordered if each pair of code words in \( C \) is unordered.

Example 1: Given code words, \( X = 001 \), \( Y = 100 \), \( Z = 011 \), the DI pairs are \( \{X, Y\} \) and \( \{Y, Z\} \). \( X \) is not unordered when compared to \( Z \).

There is a direct relationship between the covering property of unordered codes and delay-insensitivity in asynchronous communication. Assuming a four-phase protocol, where the channel is reset to all-0 (i.e., spacer) between successive transmissions, the receiver can unambiguously identify the arrival of a valid codeword. In particular, as the individual 1 bit of a codeword arrive (i.e., rising wire transitions), no other codeword will be seen transiently during the transmission, since the 1-bit pattern of each valid codeword is never covered (i.e., not a subset) by any other.

As the bits arrive, several codeword’s may be candidates for the transmitted one, but a unique valid codeword will only be seen when precisely all bits have arrived. As a result, in an asynchronous system, these codes have an inherent timing-robustness, where individual bits of a codeword can arrive in any order and at any time during transmission, and the final valid codeword can be uniquely identified. There are two classes of DI codes: systematic and non-systematic codes. A systematic code [27], [47] contains two types of fields: 1) a data (or information) field which contains the original data bits, and 2) a check field. For asynchronous communication, the check field provides extra bits to guarantee that the entire code is DI. Common types of systematic codes are Berger and Knuth codes. A key benefit of systematic codes is ease of data extraction, where no hardware decoders are necessary since the original data appears directly in the codeword.

In contrast, in nonsystematic codes [6], [30], [47], there are no separate data and check fields. Data is encoded in a unified field, which ensures delay-insensitivity. Common examples include dual-rail (i.e., 1-of-2, 1-of-4, and the general class of \( m \)-of-\( n \) codes [47]. For the simplest cases (e.g., dual-rail), data extraction is trivial, through a simple mapping.

C. Error Classification

There are three main classes of errors that can be handled [10]: symmetric, unidirectional, and asymmetric. Symmetric errors may include both \( 0 \rightarrow 1 \) and \( 1 \rightarrow 0 \) bit corruptions simultaneously within the same codeword [10], [24]. Unidirectional errors can involve either \( 0 \rightarrow 1 \) or \( 1 \rightarrow 0 \) type bit corruptions, but only one type may occur within any given codeword; the type need not be known in advance [10], [12], [13], [27]. Asymmetric errors involve only one error type for all code words (\( 0 \rightarrow 1 \) or \( 1 \rightarrow 0 \)), where the type must be known in advance [17], [24], [33]. The focus of this project is on handling the most general class, symmetric errors.

One widely used type of error-correcting code is the Hamming code. These codes have two fields: data (information) and check (parity) bits. The Hamming code is an example of a weight-based code, where each bit position is assigned a weight. The information bits are assigned non-power-of-two weights, and the check field bits are assigned power-of-two weights. The weights assigned to the information field are called the weight set. Each parity bit provides error coverage for a unique subset of information bits, called a parity group. A parity bit is set to the appropriate value which makes its parity group even.

D. BERGER CODE

In telecommunication, a Berger code is a unidirectional error detecting code, named after its inventor, J. M. Berger. Berger codes can detect all unidirectional errors. Unidirectional errors are errors that only flip ones into zeroes or only zeroes into ones, such as in asymmetric channels. The check bits of Berger codes are computed by summing all the zeroes in the information word, and expressing that sum in natural binary. If the information
word consists of \( n \) bits, then the Berger code needs \( k = \lceil \log_2(n + 1) \rceil \) "check bits", giving a Berger code of length \( k+n \). (In other words, the \( k \) check bits are enough to check up to \( 2^n - 1 \) information bits).

Berger codes can detect any number of one-to-zero bit-flip errors, as long as no zero-to-one errors occurred in the same code word. Berger codes can also detect any number of zero-to-one bit-flip errors, as long as no one-to-zero bit-flip errors occur in the same code word. Berger codes cannot correct any error. Like all unidirectional error detecting codes, Berger codes can also be used in delay-insensitive circuits.

i) Unidirectional error detection

As stated above, Berger codes detect any number of unidirectional errors. For a given code word, if the only errors that have occurred are that some (or all) bits with value 1 have changed to value 0, then this transformation will be detected by the Berger code implementation. To understand why, consider that there are three such cases:

1. Some 1s bits in the information part of the code word have changed to 0s.
2. Some 1s bits in the check (or redundant) portion of the code word have changed to 0s.
3. Some 1s bits in both the information and check portions have changed to 0s.

For case 1, the number of 0-valued bits in the information section will, by definition of the error, increase. Therefore, our Berger check code will be lower than the actual 0-bit-count for the data, and so the check will fail.

For case 2, the numbers of 0-valued bits in the information section have stayed the same, but the value of the check data has changed. Since we know some 1s turned into 0s, but no 0s have turned into 1s (that's how we defined the error model in this case), the encoded binary value of the check data will go down (e.g., from binary 1011 to 1010, or to 1001, or 0011). Since the information data has stayed the same, it has the same number of zeros it did before, and that will no longer match the mutated check value.

For case 3, where bits have changed in both the information and the check sections, notice that the number of zeros in the information section has gone up, as described for case 1, and the binary value stored in the check portion has gone down, as described for case 2. Therefore, there is no chance that the two will end up mutating in such a way as to become a different valid code word.

A similar analysis can be performed, and is perfectly valid, in the case where the only errors that occur are that some 0-valued bits change to 1. Therefore, if all the errors that occur on a specific codeword all occur in the same direction, these errors will be detected. For the next code word being transmitted (for instance), the errors can go in the opposite direction, and they will still be detected, as long as they all go in the same direction as each other.

ii) Return-to-zero (RZ)

Return-to-zero (RZ) describes a line code used in telecommunications signals in which the signal drops (returns) to zero between each pulse. This takes place even if a number of consecutive 0’s or 1’s occur in the signal. The signal is self-clocking. This means that a separate clock does not need to be sent alongside the signal, but suffers from using twice the bandwidth to achieve the same data rate as compared to non-return-to-zero format.

The "zero" between each bit is a neutral or rest condition, such as a zero amplitude impulse (PAM), zero phase shift in phase-shift keying (PSK), or mid-frequency in frequency-shift keying (FSK). That "zero" condition is typically halfway between the significant condition representing a 1 bit and the other significant condition representing a 0 bit. Although return-to-zero (RZ) contains a provision for synchronization, it still has a DC component resulting in "baseline wander" during long strings of 0 or 1 bits, just like the line code non-return-to-zero.

![Fig 1 Return-to-zero (RZ)](image)

iii) Non-return-to-zero (NRZ)

In telecommunication, a non-return-to-zero (NRZ) line code is a binary code in which 1’s are represented by one significant condition (usually a positive voltage) and 0s are represented by some other significant condition (usually a negative voltage), with no other neutral or rest condition. The pulses have more energy than a RZ code. Unlike RZ, NRZ does not have a rest state. NRZ is not inherently a self-synchronizing code, thus some additional synchronization technique (for example a run length limited constraint, or a parallel synchronization signal) must be used for avoiding bit slip.
For a given data signaling rate, i.e., bit rate, the NRZ code requires only half the bandwidth required by the Manchester code. When used to represent data in an asynchronous communication scheme, the absence of a neutral state requires other mechanisms for bit synchronization when a separate clock signal is not available. NRZ-Level itself is not a synchronous system but rather an encoding that can be used in either a synchronous or asynchronous transmission environment, that is, with or without an explicit clock signal involved. Because of this, it is not strictly necessary to discuss how the NRZ-Level encoding acts "on a clock edge" or "during a clock cycle" since all transitions happen in the given amount of time representing the actual or implied integral clock cycle. The real question is that of sampling the high or low state will be received correctly provided the transmission line has stabilized for that bit when the physical line level is sampled at the receiving end.

However, it is helpful to see NRZ transitions as happening on the trailing (falling) clock edge in order to compare NRZ-Level to other encoding methods, such as the mentioned Manchester code, which requires clock edge information (is the XOR of the clock and NRZ, actually) and to see the difference between NRZ-Mark and NRZ-Inverted.

![Fig.2 Non-return-to-zero (NRZ)](image)

**III. ZERO-SUMECU CODE**

The hardware components and system-level protocol to support the basic Zero-Sum code are now described. Fig. 3 shows the target system-level micro architecture for the Zero-Sum code. Each codeword is transmitted on the four-phase asynchronous communication channel between sender and receiver. The sender node generates the check field, which is appended to the data word to form the ECU code. The system has three key components: an encoder, a completion detector, and an error corrector unit.

A basic unoptimized encoder design for a 4-bit data word is shown in Fig. 13. It consists of a bank of selectors (i.e., multiplexers) followed by adders. There is one selector for each data field index value (3, 5, 6, 7), and each is configured either to pass the hardcoded index (if the corresponding data bit is 0) or the value 0 (if the corresponding data bit is 1). Fig. 14(a) shows an alternative unified approach where the entire encoder is designed as a single combinational logic block.

![Fig.3 Extended Block-level system Micro Architecture](image)

The completion detector (CD) is shown in Fig.3. Each C-element detects exactly one of the 16 distinct codeword’s, and hence only one C-element is enabled per transaction. A single multi input OR gate combines the results to produce the final ACK. Each gate can be decomposed into smaller fan in gates without affecting the hazard-freedom of the design. To reduce overhead, inputs to C-elements were removed whenever they are set to 0 in the corresponding codeword. For the 4-bit data word 0000, the C-element for the corresponding AC-element is a standard storage element, whose output is 0 (1) when all inputs are 0 (1), and which otherwise holds its value, codeword 0000 10101 has only three inputs $c_{16}$, $c_{4}$, and $c_{1}$ which correspond to the bits set to 1.

The 4-bit error corrector unit, shown in Fig.4, is divided into two parts: a syndrome generator and corrector. The syndrome generator produces the syndrome by performing the operations of comparison and subtraction. First, given the received data word, an encoder generates a new check field. Next, the syndrome is generated by finding the mathematical difference between the received and newly generated check fields. A magnitude comparator is used to perform the absolute value function. The top-most multiplexer selects the larger of the two values, while the lower multiplexer selects the smaller of the two.

The second part of the error corrector unit performs the correction operation similar to Hamming correction. A C-element and 2-input XOR gate is allocated for each bit of a codeword. The input to the C-element is the syndrome, and for each bit, a nonzero syndrome which uniquely identifies when an error occurs in that particular bit. The corresponding XOR gate corrects the faulty bit by
performing bit-inversion. Given an error, exactly one C-element and XOR gate will be enabled. A bank of latches is included to ensure a glitch-free transaction.

Fig. 4. Micro architecture of 4-bit zero-sum error-corrector unit design.

Fig 5 Basic 4-bit Encoder Design

Fig 6 4-Bit hardware components. (a) Unified Encoder. (b) CD

Fig 7. M-of-NCU design hardware blocks: (a) Encoder. (b) CD (c) Decoder.

Transmission Scenarios: There are three transmission cases: i) error-free; ii) error occurs: CD detects a valid codeword; and iii) error occurs: CD does not detect a valid codeword. Case ii) occurs if a valid codeword is traversed on the way to an illegal codeword. The valid codeword may be either the original sent codeword (i.e., error bit is last to arrive), or in some scenarios another valid codeword (i.e., error bit is not last to arrive, and a different valid codeword is traversed). In this case, there is at least one additional 1 bit (i.e., 0 → 1 error).

In both scenarios, the CD asserts its ACK high. Case iii) occurs if no valid codeword is traversed. For a single-bit error, this case occurs when a 1-bit is suppressed (i.e., 1 → 0 error). As a result, the CD’s ACK output remains low. As shown in Fig. 12, the combinational syndrome generator processes the codeword continuously as it arrives, but the corrector unit is only activated through an input D-latch register once the CD asserts its ACK high. Hence, in cases (i) and (ii), once a valid codeword is detected, correction is initiated. However, in case (iii), where the CD is not activated, a time-out mechanism is added to initiate correction.

A. ENCODER

Unlike a multiplexer that selects one individual data input line and then sends that data to a single output line or switch, a Digital Encoder more commonly called a Binary Encoder takes all its data inputs one at a time and then converts them into a single encoded output. So we can say that a binary encoder, is a multi-input combinational logic circuit that converts the logic level "1" data at its inputs into an equivalent binary code at its output. The Digital Encoder is a combinational circuit that generates a specific code at its outputs such as binary or BCD in response to one or more active inputs. There are two main types of digital encoder they are The Binary Encoder and the Priority Encoder.
The **Binary Encoder** converts one of $2^n$ inputs into an $n$-bit output. Then a binary encoder has fewer output bits than the input code. Binary encoders are useful for compressing data and can be constructed from simple AND or OR gates. One of the main disadvantages of a standard binary encoder is that it would produce an error at its outputs if more than one input were active at the same time. To overcome these problem priority encoders were developed.

One of the main disadvantages of standard digital encoders is that they can generate the wrong output code when there is more than one input present at logic level "1". For example, if we make inputs $D_1$ and $D_2$ HIGH at logic "1" at the same time, the resulting output is neither at "01" nor at "10" but will be at "11" which is an output binary number that is different to the actual input present. Also, an output code of all logic "0"s can be generated when all of its inputs are at "0" OR when input $D_0$ is equal to one. One simple way to overcome this problem is to "Prioritize" the level of each input pin and if there was more than one input at logic level "1" the actual output code would only correspond to the input with the highest designated priority. Then this type of digital encoder is known commonly as a Priority Encoder or P-encoder for short.

**B. PRIORITY ENCODER**

The Priority Encoder solves the problems mentioned above by allocating a priority level to each input. The priority encoders output corresponds to the currently active input which has the highest priority. So when an input with a higher priority is present, all other inputs with a lower priority will be ignored. The priority encoder comes in many different forms with an example of an 8-input priority encoder along with its truth table shown below.

Priority encoders are available in standard IC form and the TTL 74LS148 is an 8-to-3 bit priority encoder which has eight active LOW (logic "0") inputs and provides a 3-bit code of the highest ranked input at its output. Priority encoders output the highest order input first for example, if input lines "D2", "D3" and "D5" are applied simultaneously the output code would be for input "D5" ("101") as this has the highest order out of the 3 inputs. Once input "D5" had been removed the next highest output code would be for input "D3" ("011"), and so on.

**C. DECODER**

A decoder is a device which does the reverse operation of an encoder, undoing the encoding so that the original information can be retrieved. The same method used to encode is usually just reversed in order to decode. It is a combinational circuit that converts binary information from $n$ input lines to a maximum of $2^n$ unique output lines. In digital electronics, a decoder can take the form of a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs, where the input and output codes are different. e.g. $n$-to-2$^n$, binary-coded decimal decoders. Enable inputs must be on for the decoder to function, otherwise its outputs assume a single "disabled" output code word. Decoding is necessary in applications such as data multiplexing, 7 segment display and memory address decoding.

The example decoder circuit would be an AND gate because the output of an AND gate is "High" (1) only when all its inputs are "High." Such output is called as "active High output". If instead of AND gate, the NAND gate is connected the output will be "Low" (0) only when all its inputs are "High". Such output is called as "active low output".

![2-to-4 Line Single Bit Decoder](image)

A slightly more complex decoder would be the $n$-to-$2^n$ type binary decoders. These types of decoders are combinational circuits that convert binary information from `$n` coded inputs to a maximum of $2^n$ unique outputs. We say a maximum of $2^n$ outputs because in case the `$n` bit coded
information has unused bit combinations, the decoder may have less than $2^n$ outputs. We can have 2-to-4 decoder, 3-to-8 decoder or 4-to-16 decoder. We can form a 3-to-8 decoder from two 2-to-4 decoders (with enable signals).

IV. SIMULATION RESULTS

Fig10. Zero Sum Encoder

Fig11. Zero Sum Decoder
Designing Delay Insensitive Error-Correcting Unordered Codes for Robust Asynchronous Global Communication

Fig 12. Zero Sum Plus Encoder

Fig 13. Zero Sum Star Encoder

Fig 14. Zero Sum Star Element
Fig15. Four Phase Converter

Fig16. Completion Detector
V. CONCLUSION

Zero-Sum error-correcting unordered codes support the design of asynchronous global communication which combines two forms of reliability: timing robustness and fault tolerance. Several key extensions to a basic Zero-Sum code were introduced. The complete Zero-Sum code family was defined, by generalizing the weight assignment of the entire code space. Two new ECU codes, Zero-Sum+ and Zero-Sum*, were also proposed. The Zero-Sum* code, using a strategic heuristic, corrected a significant percent of 2-bit errors (52.92%) for mid to large field sizes with only modest overheads. Designs of the Zero-Sum code hardware have been implemented and technology-mapped, and have substantially lower area and moderately lower latency, for most field sizes, when compared to the most coding-efficient m-of-n ECU (Error Correcting Unordered) codes. Hence we can conclude that our new design is an efficient pipelined architecture with error correction capabilities for asynchronous communication with an improvement in error correction capability from 52.92% to 61.9%.

VI. REFERENCES


