Novel Implementation of a Keyless Concurrent Codes Spread-Spectrum (CCSS) Jam-Resistant Method in GNURadio

James Morrison
Department of Electrical and Computer Engineering
United States Air Force Academy
c23james.morrison@afacademy.af.edu

Neil Rogers
Department of Electrical and Computer Engineering
United States Air Force Academy
neil.rogers@afacademy.af.edu

William Bahn
Department of Computer and Cyber Sciences
United States Air Force Academy
william.bahn@afacademy.af.edu

***Abstract*—Key-sharing to secure wireless transmissions can be seriously degraded or denied in the presence of intentional or unintentional jamming, and public communications protocols (like GPS or ADSB/Mode-S) cannot leverage jam-resistant capabilities afforded by keys. Given the proliferation of inexpensive Software Defined Radios (SDRs), robust and secure encryption is more important than ever. This project implements a proof-of-concept for the "BBC" codec – a keyless, concurrent codes spread-spectrum approach to jam-resistant communications. Using GNU Radio as a testbed, one may assess BBC's preservation of information availability under a variety of modulation and attack strategies. Development is underway to provide a BBC CGRAN library for broader use within the GNU Radio community.**

***Keywords*—concurrent codes, jam-resistant, wireless, RF, communications, BBC, GNURadio, SDR**

I. Background

Modems that support protected (i.e., private) omnidirectional wireless connections leverage keys for two primary functions; traffic Encryption Keys (TEKs) protect the content of traffic, while Transmission Security Keys (TSKs) protect the physical-layer signal. Historically, these keys were all symmetric, having to be exchanged in advance under ideal conditions. Thus, protected networks—especially those relying on omnidirectional broadcasts—had frustrating scalability and use-case limitations. However, the advent of asymmetric keys and Public Key Infrastructure (PKI) allowed public keys to be sent without concern, effectively solving the key-sharing problem for TEKs. Even the low bit-rates of asymmetric algorithms such as Diffie-Hellman key exchange and RSA have been outperformed by hybrid approaches where symmetric keys are shared during an asymmetrically-keyed connection, allowing higher-rate symmetric algorithms like 3DES and AES to encrypt sustained connections. [**?**]

Despite these advancements, protected connections are still contingent on symmetric TSKs for jam-resistance. That is, asymmetric keys fail to facilitate modern omnidirectional transmission security strategies like frequency hopping spread spectrum (FHSS), direct-sequence spread spectrum (DSSS), and time-permutation. Thus, protected networks still have scalability and use concerns. After careful, yet prolonged distribution of symmetric TEKs, it is likely that one or more keys may become compromised. With a compromised TEK, a TSK transmitted at the beginning of a broadcast may be intercepted. As a result, the signal may be jammed in the physical layer, even if the TEK encryption holds. This danger is exacerbated because the difficulty of key-redistribution grows with the size of the network. Furthermore, without a TSK's physical-layer advantages, encrypted connections cannot exchange a TSK when jamming is present. Such a broadcast is susceptible to degradation or even denial.

Furthermore, public networks (i.e., those without an enumerated list of authorized users) cannot leverage symmetrically-keyed jam-resistance. Since, in effect, everyone has the TSK, adversaries could predict the spectrum usage of the signal and easily jam it. As the technology industry's reliance on large-scale networks like Global Navigation Satellite Systems grows, jam-resistance becomes increasingly crucial to preserve proper functionality.

The communications community has been making efforts toward these ends. Pöpper, Strasser, and Čapkun were able to achieve linear time complexity with Uncoordinated Direct-Sequence Spread Spectrum (UDSSS), which did not rely on shared secrets. However, this strategy required a delay between the message transmission and the jamming transmission to establish physical-layer jam-resistance [**?**]. While a robust methodology for many use-cases, the ability to establish a connection under adverse conditions has yet to be demonstrated.

II. BBC Algorithm

As proposed by Baird, Bahn, and Collins, the BBC codec is a physical-layer solution for keyless jam-resistant communication, with no inherent requirement for an uncontested synchronization step [**?**]. By modeling the wireless channel as a bitwise-OR, BBC leverages the fact that, though jamming adds unwanted information to transmitted codewords, it is unable to remove information that is already there (i.e., bits with value 1). Given that the bitwise-OR is a good approximation for variable-envelope asymmetric modulation schemes, it is useful to represent a signal as a bit vector. This simplification does not describe a signal's power distribution across frequency and time (or its phase) but is a useful heuristic and may be implemented by diverse modulation strategies.

A BBC "mark" is a binary 1 that serves as a marker for a specific substring, and a packet is a bitwise-OR of a transmitted codeword and any additional marks from intentional or unintentional interference. After receiving a packet, a decoder may do two things; it may decode the packet as a single message, or it may decode every message covered by the packet. In the former case, a transmitter and receiver pair that use a codec which encodes characters as ASCII may send a '2' across an intentionally or unintentionally jammed channel and receive a '3'; thus the packet has been corrupted by the addition of a single 1 in the codeword:

TABLE I
ASCII Single Decode after Channel OR.

Origin	ASCII Character	ASCII Byte (Codeword)
Original message (TX)	2	00110010
Jamming signal	!	00100001
Channel packet (OR)	3	00110011
Decoder (RX)	3	00110011

In the latter case, the decoder reverses the channel bitwise-OR by decoding every possibility. Using the above example, the output of the decoder includes the original message and many "hallucinations:"

Even after filtering for characters (i.e., ASCII bytes which are between the decimal values 32 and 126 inclusive), there are 7 incorrect messages, or "hallucinations"; 87.5% of the decoded packet is erroneous. Evidently, the computational efficiency of the decoder drops as more codewords are covered by the packet—that is, as there are more 1s than 0s in the packet. Furthermore, consistently selecting the original message from the hallucinations is impractical.

This toy example informs the primary goal of the BBC codec: to reduce the number of hallucinations introduced by a bitwise-OR of its codewords. Toward this end, BBC

TABLE II
ASCII Iterative Decode after Channel OR.

Origin	ASCII Character	ASCII Byte (Codeword)
Original message (TX)	2	00110010
Jamming signal	!	00100001
Channel packet (OR)	3	00110011
Decoder (RX)	SOH	00000001
	STX	00000010
	ETX	00000011
	DLE	00010000
	DC1	00010001
	DC2	00010010
	DC3	00010011
	Space	00010000
	!	00100001
	"	00100010
	#	00100011
	0	00110000
	1	00110001
	2	00110010
	3	00110011

leverages the theory of concurrent codes (and thus is an implementation of concurrent codes spread-spectrum, CCSS).

A concurrent code is a set of bit vectors such that it is improbable that an individual element "is a subset of a bitwise-OR of a small number of the others." A concurrent code also requires that such a vector be decoded efficiently, preferably in linear time. The BBC algorithm defines its concurrent code by leveraging a type of multi-hot encoding. For a given n-bit message, the encoder will place n marks inside a vector of zeros with length greater than $2n$. The location of each mark is given as a vector index by successively hashing substrings of the original message using the Glowworm hash (Fig. 2-4). The resulting vector is the BBC codeword. Below is a toy example of the BBC encoding process for an ASCII '2' (00110010 in binary), assuming a codeword length of 2n:

TABLE III
BBC Encode of ASCII '2'.

Message substring	Mark index from hash	Codeword after placing mark
''		00000000 00000000
'0'	5	00000100 00000000
'00'	0	10000100 00000000
'001'	14	10000100 00000010
'0011'	2	10100100 00000010
'00110'	0	10100100 00000010
'001100'	0	10100100 00000010
'0011001'	3	10110100 00000010
'00110010'	12	10110100 00001010

Though indices for 8 marks have been generated by the hash function, 3 of the substrings' mark indices were

the same: 0. This overlap suggests that the arbitrary length of $2n = 2(8) = 16$ bits may be too small; our codeword density, μ, is approaching the limit of the decoder. Since we have 6 marks and 16 bits in our codeword, $\mu = \frac{6}{16} = 0.375$. The receiver codec will still decode the packet in linear time since the codeword density is under 0.5 [**?**]. However, there likely will be hallucinations.

To reverse the encoder's mark placement, the decoder must re-build every message that may have been included by checking for marks corresponding to various substrings. In the case of a small 16-bit codeword, there are $2^{16} = 65536$ message possibilities. A depth-first search tree is employed to check for substring marks in the codeword efficiently. That is, the decoder begins by proposing that the message began with '0.' If there is a mark in the codeword corresponding to the substring '0,' this branch is taken and '00' is tested. If not, the decoder tests for the substring '1', and so on. Below is the decoding tree for the 16-bit BBC codeword for ASCII '2' (10011100 00001010).

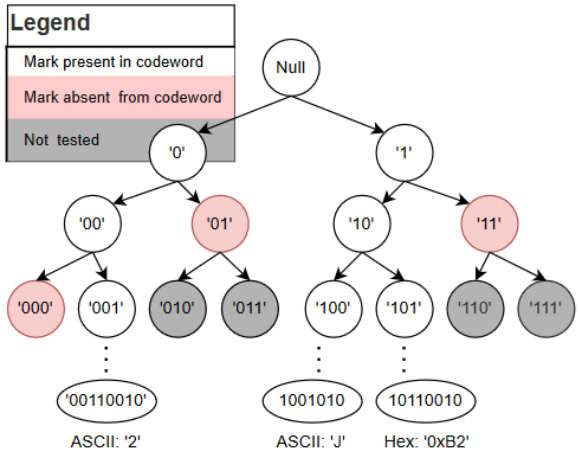


Fig. 1. Partial Decoder Tree.

In this example, at a depth of 3, the search has eliminated 5/8 of the future substrings. In other words, the decoder has already eliminated about 40,962 of the 65,536 message possibilities. While helpful to understand the efficiency of the BBC decoder, this top-down perspective suggests a breadth-first search. Note that because the algorithm is depth-first, the decoder would not discover the '1' branch of the tree until the message '00110010' had been fully decoded. The terminal nodes at the bottom of the tree show that the RX-codec decoded '2', 'J', and 0xB2. In this case, with $\mu = 0.375$, there were 2 hallucinations.

However, in the presence of jamming, the packet density is more consequential than the original codeword density; the channel bitwise-OR will introduce more marks than were in the original packet. For example, when the encoded '2' is jammed by interference that represents a BBC-encoded '!,' 4 new marks are added and packet density is $\frac{6+5}{16} = 0.6875$.

TABLE IV
CHANNEL OR.

Origin	ASCII Character	BBC Codeword
Original message (TX)	2	10011100 00001010
Jamming signal	!	01000001 10001101
Channel packet (OR)	N/A	11011101 10001111

Now over the 0.5 threshold, the decoder returns 14 possibilities (8 of which are ASCII characters)—a comparable result to the ASCII-only encoder. This example demonstrates the importance of judiciously sizing the BBC codeword. In the same jamming scenario, the decoder finds 5 message possibilities if a 32-bit codeword is used. With a 56-bit codeword, no hallucinations are present: only '2' and '!' remain.

III. The Glowworm Hash

Baird, Bahn, Carlisle, and Smith designed an optimal mark-placing hash function for BBC called Glowworm [**?**]. Originally written as C macro functions, the code has been translated to Python to support object-oriented integration in GNURadio. This hash is comprised of 3 functions: one for initializing a shift register, one to process a bit and return a mark index, and one to restore the shift register's state after processing a bit (i.e., a bit deletion). The *init()* function fills up a shift register with 32 64-bit words. The initial hash state is determined by 4096 iterations of adding the previous entry's least significant bit.

```
# Initialize Glowworm
def init(s):
    n = 0
    h = 1
    for i in range(32):
        s[i]=0
    for i in range(4096):
        h=add_bit(h&1, s)
    n = 0
```

Fig. 2. Glowworm Initialization Function.

The *add_bit()* function actually hashes a given bit. In BBC, the modulo of its 64-bit return and the codeword length becomes the index of the new mark. In order to "hash a substring," each individual bit needs to be hashed in order. The shift register's modified state is responsible for '01' giving a different output than '1' or '001'; it can essentially store a substring even though it is passed via

n separate function calls, where *n* is the length of the current substring and a global variable.

```
# Enforce 64-bit word
MAX_VAL = 0xffffffffffffffff

# Hash a bit
def add_bit(b, s):
    t = (s[n % 32]^(0xffffffff if b else 0))
    t = ((t|(t>>1))^((t<<1)&MAX_VAL))
    t = (t ^ (t>>4)^(t>>8)^(t>>16)^(t>>32))
    n += 1
    s[n % 32] ^= (t&MAX_VAL)
    return s[n % 32]
```

Fig. 3. Glowworm Hash Function.

Finally, the *del_bit()* function reverses a hash state change caused by add_bit(). This enables the BBC decoder to navigate the search tree; in cases where a substring's mark is not found in the codeword, del_bit() unhashes the last bit and traverses one level up in the tree.

```
def del_bit(b, s):
    n -= 1
    add_bit(b,s),
    n -= 1
    return s[n % 32]
```

Fig. 4. Glowworm Reverse Hash Function.

IV. Implementation of BBC in Python

Python's support of object oriented programming enables the creation of a BBC codec object that can simultaneously encode messages and decode codewords. This is advantageous toward supporting full duplex wireless communication, rather than relying on procedural scripts. Furthermore, Python's built-in bytearry object supports a variable-length array that can store more information than a machine-precise 64-bit word. This is particularly helpful for creating large codewords (e.g., 2^{20} bits long). Since the bytearray object supports indexing via the *memoryview()* method, it is easy to place and search for BBC marks. The Python *Codec* object initializes both an *Encoder* and a *Decoder* for predetermined message, and codeword, and checksum lengths A.1. Note that BBC performs a pseudo-checksum by appending additional zeros to the message before encoding it, reducing the number of hallucinations contained in the codeword.

To create a BBC codeword, data is parsed from the source, interpolated to *message_length*-long bit vectors, and passed to the *Encoder*. Once its Glowworm shift register has been initialized (lines 11-14), it is free to iteratively place marks for each substring in the message (lines 27-34). Only one bit is hashed at a time, but the shift register stores the rest of the substring until bits are deleted from it A.2.

To decode a BBC codeword, an iterative model is used, rather than the recursive model proposed in [**?**]. This novel approach increases efficiency and reduces the stack size, making it computationally easier to decode larger packets/codewords. Messages that have a mark in the codeword for each substring are stored in the static variable, *message_list* A.3.

To iterate the depth-first search, the decoder uses Glowworm to check if it can append a 0 to the current message substring, which is initially empty. If Glowworm returns an index where a mark is present in the codeword, the current substring is valid, and the "explore" path is taken (lines 33-48). If the current substring is not covered by the packet, then the "backtrace" path is taken (lines 51-72). Here, the algorithm can reliably take the next "deepest" path by deleting 1's from the end of the current message substring until a 0 is encountered (reducing its depth), at which point it changes that 0 to a 1 and continues to the next iteration. After every message covered message is found, they are all returned via *message_list* (line 75) A.3.

V. Implementation of BBC in GNURadio

To support SDR testing, the *Encoder* and *Decoder* are each implemented as a GNURadio *Python Blocks*. The class implementations are similar; however, each is instantiated in the respective *blk* constructor B.2. These custom blocks along with native *Stream to Vector* and *Vector to Stream* blocks serve as a basic testbed for BBC/CCSS in GNURadio, leaving the modulation and jamming strategies open for implementation.

Initial verification consisted of a simulation, as shown in Figure 5. Due to the ambivalence of the BBC codec to extra marks, it is capable of encoding multiple messages into a single codeword; this flowgraph demonstrates this capability by encoding the two messages separately and performing a bitwise "OR" on the two resulting codewords. The final codeword is then translated and decoded with no errors. In this implementation, the codeword length and message length are known by the encoder and decoder. The BBC encoder and encoder expect vectors of the same length as the message and codeword as inputs/outputs, respectively. Stream-to-vector and vector-to-stream blocks manage the data flow appropriately. In fact, one of the main challenges of developing the encoder and decoder block is to dynamically scale the size of the input and output vectors for the encoder and decoders to match arbitrary message and codeword lengths. At one point, a pull request existed to implement this functionality within the core GNURadio source code, but it was closed without remedy [**?**].

As was previously mentioned, it is crucial to correctly size the codeword to remain under 50% mark density. Testing of our GNURadio implementation proved this limit. We performed two separate tests to verify this limit.

First, using a 128-byte (2^7) message and various codeword lengths from 2^{17} to 2^8 bytes, the original message was recovered until the codeword reached 2^9 bytes. At this point, the decoder produced a 1-byte error, in which the final character was not correctly decoded. Reducing the codeword length further to 2^8 bytes resulted in complete failure of the codeword to produce a message.

Secondly, we used the simulation to investigate the effect of extra marks on the decoded messages. Figure 6 illustrates the flowgraph. In this case, a single byte is OR'd with the codeword to produce extra marks wtihin each byte of the codeword. As expected, byte values containing three 1's (such as the $00011010_2 = 26_{10}$) did not significantly affect the decoding process, since the mark density was still below the 50% threshold. However, values such as $00011011_2 = 27_{10}$ produced a failure in the decoder.

Use of a significant codeword-to-message ratio (such as 2^{10}) negates any significant errors induced by jamming or noise, as the mark density of the original codeword will be so small as to allow for a significant number of fake marks

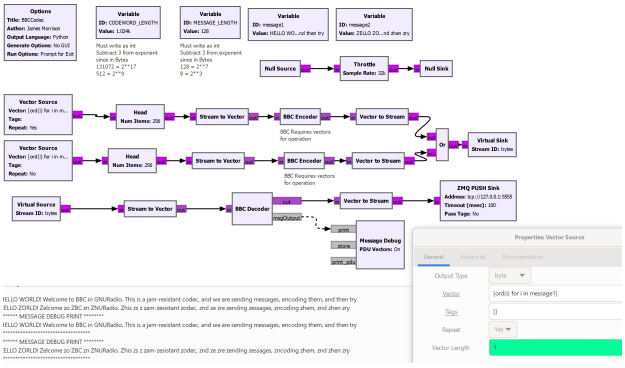


Fig. 5. The generic BBC decoder simulation as implemented in GNURadio.

VI. Towards Hardware Verification

The next logical step is to incorporate the BBC encoder/decoder framework into a representative modulation scheme. Bahn [**?**] advises against any constant envelope modulation schemes, as such jamming in schemes do not act as bitwise OR functions. Therefore, we opted for a straightforward OOK modulation scheme, as shown in Figures 7 and 8.

Although most of the blocks are standard GRC blocks, the OOK decoder is a custom python blocks, which

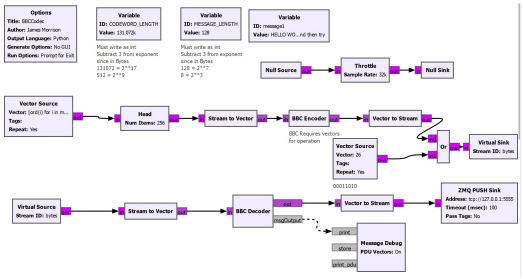


Fig. 6. The generic BBC decoder simulation modified for injecting fake marks into a codeword.

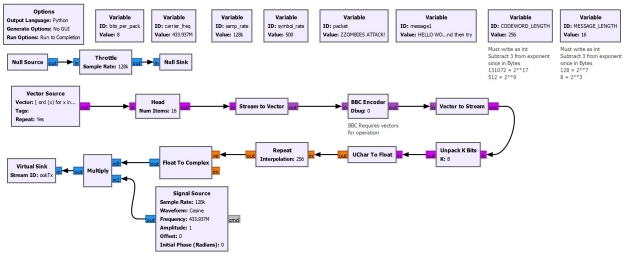


Fig. 7. The BBC encoder coupled with an OOK transmitter.

serves to decode the received and filtered OOK signal. Since each bit in the codeword is repeated to allow for modulation of a full period of the carrier, the received and demodulated signal must be carefully sampled for correct decoding. In this case, the received samples are counted and a correlated into the appropriate bit value (1 or 0). This particular block takes advantage of the PMT data type to output codewords; this is simply for convenience and will be updated in the near future. The detected bits are sent as individually bytes using the PMT.inter() method. In the future, these bytes will be sent over a standard output port.

To We modified the BBC decoder from the simulation stage to receive the individual bytes and compile the codeword, which is fed into the glowworm decoding algorithm.

In order to assess the performance of the BBC OOK implementation, the transmission time was compared with an OOK simulation containing no BBC blocks and the same custom OOK decoder, transmitting a 512 byte message. The difference in performance between the two execution times was negligible.

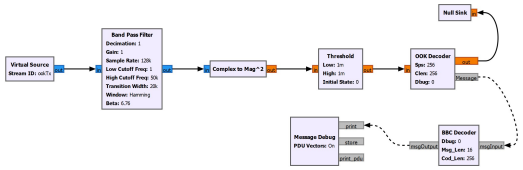


Fig. 8. The BBC decoder coupled with an OOK receiver/decoder.

VII. Conclusions and Future Work

Lessons Learned -

-

More work is needed to develop the robustness of and use-cases for BBC real-world integration. First, there are theoretical performance upgrades to the base algorithm. These include multimark BBC, codeword detection, checksum implementation, and active statistic thresholding.

Secondly, further modeling would help improve performance and efficiency. Predicted performance metrics would help inform message, codeword, and checksum length selection for variable data-rates and expected jamming/noise powers. Specific to the encoder, these metrics include codeword density and overlap count, as functions of codeword and message legnths. For the decoder, they include expected packet density and decode time-complexity, as functions of codeword and message lengths and expected channel noise.

Finally, rigorous and thorough testing of BBC under diverse noise and jamming strategies must be conducted to prove the algorithm's worth. From a communications perspective, reducing the probability of mark-deletion necessarily increases the probability that an erroneous mark is added– the extent of this change has not been tested and is dependent on the selected modulation technique.

APPENDIX A
BBC PYTHON CODE

A.1 Codec Class

```
class Codec:
    # The codec is comprised of an encoder and decoder, with an associated message/codeword pair
    def __init__(self, MSG_LEN, COD_LEN, CHK_LEN):
        self.MSG_LEN = MSG_LEN
        self.COD_LEN = COD_LEN
        self.CHK_LEN = CHK_LEN
        self.encoder = Encoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)
        self.decoder = Decoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)

    # Resulting functionality should be "mycodec.encode(<message as a string>)"
    def bbc_encode(self, message):
        self.encoder = Encoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)
        return self.encoder.encode(message)

    # Resulting functionality should be "mycodec.decode(bytearray1)"
    def bbc_decode(self, codeword):
        self.decoder = Decoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)
        return self.decoder.decode(codeword)
```

A.2 Encoder Class

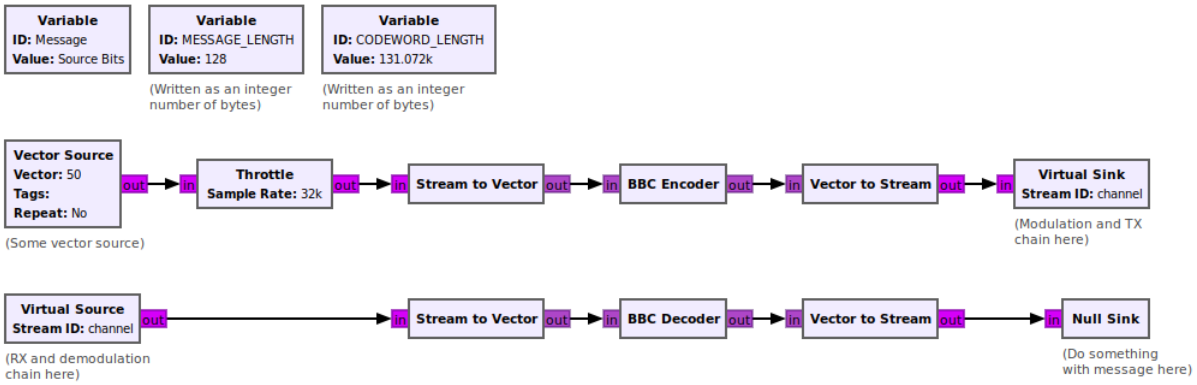
```
import glowworm.py as gw

class Encoder:
    def __init__(self, MSG_LEN, COD_LEN, CHK_LEN):
        self.shift_register = self.init_shift_register()
        self.MSG_LEN = MSG_LEN
        self.COD_LEN = COD_LEN
        self.CHK_LEN = CHK_LEN

    # Create a shift register using Glowworm
    def init_shift_register(self):
        shift_register = [0 for i in range(32)]
        gw.init(shift_register)
        return (shift_register)

    # Force message to be correct length of bytes in ASCII
    def parse_input(self, input):
        input = input.encode(encoding="ASCII")
        message = bytearray(int(self.MSG_LEN/8))
        memoryview(message)[0:(len(input))] = input
        return message

    # Encode a message using the BBC algorithm
    def encode(self, input):
        message = self.parse_input(input)
        codeword = bytearray(int(self.COD_LEN/8))
        for i in range(self.MSG_LEN):
            # ASCII byte to be encoded
            element = memoryview(message)[int((i-i%8)/8)]
            bit = ((element) >> (i%8)) & 0b1
            # Extract bit from Byte
            mark_loc = gw.add_bit(bit, self.shift_register) % self.COD_LEN
            # Mark location from glowworm
            memoryview(codeword)[int((mark_loc-mark_loc%8)/8)] |= 1<<(mark_loc%8)
        return(codeword)
```

A.3 Decoder Class

```
from math import ceil
import glowworm.py as gw

class Decoder:
    # Store configuration parameters as static variables
    def __init__(self, MSG_LEN, COD_LEN, CHK_LEN):
        # <Constructor here>


    # Allocate memory for and initialize the shift register
    def init_shift_register(self):
        shift_register = [0 for i in range(32)]
        gw.init(shift_register)
        return (shift_register)

    # Use novel iterative approach to decoding a packet
    def decode(self, packet):

        # Initialize variables
        self.message_list = []
        message = bytearray(ceil((self.MSG_LEN + self.CHK_LEN)/8))

        while True:
            # Find the proposed bit from previous execution, or a 0 if initial iteration
            prop_bit = (memoryview(message)[int((self.n - self.n%8)/8)]>>(self.n%8)) & 0b1

            # Find the corresponding mark location from glowworm
            val = gw.add_bit(prop_bit, self.shift_register) % (self.COD_LEN)

            # Logical AND to determine if present in packet/codeword
            bit = (memoryview(packet)[int((val-val%8)/8)]>>(val%8)) & 0b1

            # If the mark is present, explore
            if bit==1:
                # Message is complete, write to buffer
                if self.n == (self.MSG_LEN + self.CHK_LEN - 1):
                    self.message_list.append(bytes(memoryview(message)\
                                                   [0:self.MSG_LEN - 1 - self.CHK_LEN]))
                    bit = 0

                # Message is incomplete, continue assuming next bit is 0
                elif self.n < (self.MSG_LEN + self.CHK_LEN - 1):
                    self.n += 1
                    memoryview(message)[int((self.n - self.n%8)/8)] &= (0xff ^ (1<<self.n%8))
                    continue

            # If the mark is not present, backtrace
            if bit!=1:
                # Delete checksum bits
                while self.n >= self.MSG_LEN:
                    gw.del_bit(0, self.shift_register)
                    self.n -= 1

                # Delete 1's until a 0 is encountered
                while self.n >=0 and (((memoryview(message)[int((self.n - self.n%8)/8)]>>\
                                                            (self.n%8)) & 0b1 )==1):
                    gw.del_bit(1, self.shift_register)
                    memoryview(message)[int((self.n - self.n%8)/8)] &= (0xff ^ (1<<self.n%8))
                    self.n -= 1

                # Packet is fully decoded, proceed with next packet
                if self.n < 0:
                    break

                # Move over to the 1 branch of current search
                else:
                    gw.del_bit(0, self.shift_register)
                    memoryview(message)[int((self.n - self.n%8)/8)] |= (1<<self.n%8)

        return self.message_list
```

APPENDIX B
GNURADIO IMPLEMENTATION

B.1 Top-Level Chart



B.2 GNURadio Encoder Block Class

```
from gnuradio import gr
import numpy as np

class blk(gr.sync_block):

    # Configure port sizes
    def __init__(self, MESSAGE_LENGTH=2**7, CODEWORD_LENGTH=2**17):
        self.msg_len = MESSAGE_LENGTH
        self.cod_len = CODEWORD_LENGTH

        # Use a synchronous block
        gr.sync_block.__init__(self,
            name='BBC Encoder',
            in_sig =[(np.byte, self.msg_len)],
            out_sig=[(np.byte, self.cod_len)])

        # Convert from Bytes to bits
        self.myEncoder = Encoder(self.msg_len*8, self.cod_len*8)

    # Use BBC to encode the incoming message vectors
    def work(self, input_items, output_items):
        result = self.myEncoder.encode(input_items[0][:])
        try:
            output_items[0][:] = result
            return len(output_items[0])
        except:
            raise RuntimeError("BBC Encoder output assignment failed")
```

B.3 GNURadio Decoder Block Class

```
from gnuradio import gr
import numpy as np


class blk(gr.interp_block):

    # Configure port sizes
    def __init__(self, MESSAGE_LENGTH=2**7, CODEWORD_LENGTH=2**17):
        self.msg_len = MESSAGE_LENGTH
        self.cod_len = CODEWORD_LENGTH

        # Use an interpolation block
        gr.sync_block.__init__(self,
            name='BBC Decoder',
            in_sig =[(np.byte, self.cod_len)],
            out_sig=[(np.byte, self.msg_len)])

        # Initialize the interpolation rate to synchronous case
        self.set_relative_rate(1)

        # Convert from Bytes to bits
        self.myDecoder = Decoder(self.msg_len*8, self.cod_len*8, DEFAULT_CHECKSUM)


    # Use BBC to decode the incoming codeword vectors
    def work(self, input_items, output_items):

        # Pull packet from the queue
        packet = input_items[0][:][0]

        # Check for a nonzero codeword
        if sum(packet >0):

            result = self.myDecoder.decode(packet)
            interp = len(result)

            #Check for a decoded message
            if interp > 0:

                # Definre the number of outputs
                self.set_relative_rate(interp)

                # Assign outputs iteratively
                for j in range(interp):
                    try:
                        output_items[0][j][:] = result[j]
                    except:
                            raise RuntimeError("BBC Encoder output assignment failed")
        return len(output_items)
```