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

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

Neil Rogers (iD)
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—**Current omnidirectional jam-resistant technology relies on shared secrets. This forces private networks to exchange keys before jamming is present and leaves public networks (like GPS or ADSB/Mode-S) unprotected. Given the proliferation of inexpensive Software Defined Radios (SDRs), robust jam-resistant options are more important than ever. This project implements the "BBC" codec–a keyless, concurrent codes spread-spectrum (CCSS) strategy–in GNU Radio. Initial testing of the BBC codec confirms theoretical performance estimates and suggests that BBC is a viable option to preserve information availability. Development is underway to provide an out-of-tree (OOT) module for broader use within the GNU Radio community.**

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

I. Background

Modems that support protected (i.e., private) omni-directional 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 omni-directional 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. [1]

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. In [2], 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. Though this is 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 [3]. 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 some 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; the codeword density, μ, is approaching the limit of the decoder. Since there are 6 marks and 16 bits in the 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 [1]. However, there likely will be hallucinations.

To reverse the encoder's mark placement, the decoder must rebuild 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', 10011100000001010_2.
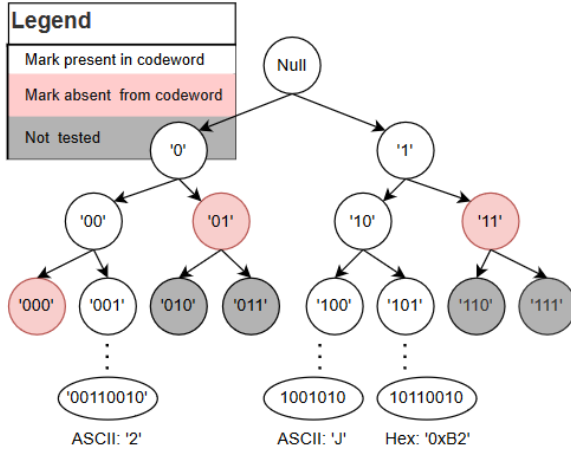

Fig. 1. Partial Decoder Tree

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 recieve-end 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 packet density 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

In [4], Baird, Bahn, Carlisle, and Smith designed a hash to optimally place marks and reduce overlap. This "Glowworm" hash was originally written as C macro functions but has been translated to Python to support object-oriented integration in GNU Radio. 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, codeword, and checksum lengths (Appendix 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* (Appendix A.2). 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.

To decode a BBC codeword, an iterative model is used, rather than the recursive model proposed in [1]. 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* (Appendix 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* (Appendix A.3, line 75).

V. IMPLEMENTATION OF BBC IN GNU RADIO

To facilitate SDR testing with GNU Radio, the *Encoder* and *Decoder* are implemented as synchronous and interpolating *Python Blocks*, respectively (Appendices B.2-B.3). Each has a *blk* constructor that instantiates the respective python class, enabling the *work* method to call BBC functions. Since the message and codeword lengths must be pre-determined and consistent between encoder and decoder, they are defined as top-level variables in the flowgraph (Appendix B.1). Stream-to-vector and vector-to-stream blocks manage the data flow appropriately.

Adjusting these top-level variables, however, currently causes a compilation error. GNU Radio does not yet support dynamic argument assignment inside the python block constructor. That is, the actual values of the top-level variable are ignored in favor of the corresponding default arguments. As a result, the message and codeword lengths had to be hard-coded in both blocks each time a different configuration was desired. This is still an open issue in the core GNU Radio source code, with a partial work-around in [5].

Initial verification consisted of a simulation (Appendix C.1). This test demonstrated the BBC codec's ability to encode multiple messages into a single codeword (i.e., a "packet") and recover both. In GNU Radio, two encoder blocks each encoded a message. The resulting codewords were passed to a native "OR," which simulated the

channel. Both messages were decoded from the packet with no hallucinations.

Furthermore, the recommended 50% packet density limit was validated with two separate tests in GNU Radio. First, using a (2^{10}) bit message (128 bytes), codeword lengths were varied from 2^{20} down to 2^{11} bits [3]. At 2^{11} bits, the codeword was twice the size of the message, and thus had an expected codeword density of 50%. As predicted, the decoder broke exceeded linear time and produced many hallucinations.

Second, the simulation was used to investigate the effect of extra marks on the decoded messages (Appendix C.2). In this test, a single byte mask was OR'd with each byte of the codeword, producing a variable number of extra marks. As expected, byte masks which contained three 1's (such as the $00011010_2 = 26_{10}$) did not significantly affect the decoding process, since the packet density was still below the 50% threshold. However, 4-bit-high masks such as $00011011_2 = 27_{10}$ produced a failure in the decoder.

Testing showed that a significant codeword-to-message ratio (such as 2^{10}) made the BBC codec resistant to a significant amount of bit-errors induced by jamming or noise. Choosing a large codeword forces the original codeword density to be low, enabling robustness against a significant number of erroneous marks.

VI. USE CASE VERIFICATION

The next logical step is to incorporate the BBC encoder/decoder framework into a representative modulation scheme. In [1], Bahn advises against using BBC in conjunction with any constant-envelope methods, because they allow added power to erase marks, invalidating the bitwise-OR assumption. That is, noise or jamming could theoretically remove a mark. Therefore, we selected on-off-keying (OOK) selected for an initial test-case (Appendices C.3-C.4).

The OOK codec and modulation/demodulation process was relatively straightforward and uses bitwise modulation of a cosine carrier. For simplicity's sake, we saved the modulator and demodulator as hierarchical blocks. Note that the BBC codeword is modulated, not the individual messages.

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 using the OOK decoder hier block to transmit a 2^{20} bit codeword. The difference in performance between the two execution times was negligible, providing reasonable confirmation of the codec's efficiency. In fact, the decoder run time was approximately 20ms, a negligible portion of the flowgraph run time.

One possible use case of the BBC codec is to facilitate key (or key parameter) sharing for another communica-

tions protocol that may be more secure and higher bandwidth, in the presence of jamming or noise. Therefore, in the flowgraph contained in C.5, frequency hopping parameters are passed through the BBC OOK transmitter and the output commands the FHSS hop sequence. In this simple case, two frequencies are sent in a single codeword, but, with a sufficiently large codeword, a large number of frequencies could be passed – essentially constituting a hop table. This flowgraph provides a proof-of-concept for a very real use case incorporating BBC communications.

VII. CONCLUSIONS AND FUTURE WORK

More work is needed to develop the robustness of and specify the use-cases for BBC. First, there are theoretical performance upgrades to the base algorithm. These include multimark BBC, codeword detection, and automatic gain control [3]. Multimark BBC reduces the number of hallucinations in a decoded packet by requiring each substring to have more than 1 associated mark. This comes at a cost, though: the addition of more marks may force the receiving decoder beyond the 50% packet density threshold. Codeword detection would enable BBC packet lengths to be greater than the selected codeword length; this is a necessary step to implement the encoder and decoder on separate clocks. Finally, automatic gain control would allow a receiver with a BBC decoder to operate in noisier environments. By adjusting the power threshold that separates the 0 and 1 symbols, the decoder could ensure that it remains below the 50% packet density threshold.

Second, statistical modeling would help specify the BBC use case. Predicted performance metrics would 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 lengths. For the decoder, they include expected packet density and decode time-complexity, as functions of codeword and message lengths and expected channel noise (Note that the ratio of message length to codeword length is a useful approximation for packet density but technically is incorrect due to mark overlaps).

Finally, rigorous and thorough testing of BBC under diverse noise and jamming strategies must be conducted to prove the algorithm's worth in any real-world use case. Reducing the probability of mark-deletion necessarily increases the probability that an erroneous mark is added, and the bit-error probabilities for BBC with OOK or other modulation strategies have not been been tested.

Shortly, an OOT module will be released via github/C-GRAN, BBC will available for widespread use and development.

REFERENCES

[1] W. Bahn, "Concurrent code spread spectrum: Theory and performance analysis of jam resistant communication without shared secrets," 2012.

[2] C. Pöpper, M. Strasser, and S. Čapkun, "Jamming-resistant broadcast communication without shared keys," 08 2009.

[3] W. L. Bahn, L. C. Baird, and M. D. Collins, "Jam resistant communications without shared secrets," 4 2008.

[4] L. Baird, M. Carlisle, W. Bahn, and E. Smith, "The glowworm hash: Increased speed and security for bbc unkeyed jam resistance," 12 2012.

[5] Gnuradio, "Embedded python block - cannot make variable input/output length – issue #4196 – gnuradio/gnuradio." [Online]. Available: https://github.com/gnuradio/gnuradio/issues/4196

[6] L. Baird and W. Bahn, "Security analysis of bbc coding," 12 2008.

APPENDIX A
BBC PYTHON CODE

A.1 Codec Class

```
1  class Codec:
2      # The codec is comprised of an encoder and decoder, with an associated message/codeword pair
3      def __init__(self, MSG_LEN, COD_LEN, CHK_LEN):
4          self.MSG_LEN = MSG_LEN
5          self.COD_LEN = COD_LEN
6          self.CHK_LEN = CHK_LEN
7          self.encoder = Encoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)
8          self.decoder = Decoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)
9
10     # Resulting functionality should be "mycodec.encode(<message as a string>)"
11     def bbc_encode(self, message):
12         self.encoder = Encoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)
13         return self.encoder.encode(message)
14
15     # Resulting functionality should be "mycodec.decode(bytearray1)"
16     def bbc_decode(self, codeword):
17         self.decoder = Decoder(self.MSG_LEN, self.COD_LEN, self.CHK_LEN)
18         return self.decoder.decode(codeword)
```

A.2 Encoder Class

```
1  import glowworm.py as gw
2
3  class Encoder:
4      def __init__(self, MSG_LEN, COD_LEN, CHK_LEN):
5          self.shift_register = self.init_shift_register()
6          self.MSG_LEN = MSG_LEN
7          self.COD_LEN = COD_LEN
8          self.CHK_LEN = CHK_LEN
9
10     # Create a shift register using Glowworm
11     def init_shift_register(self):
12         shift_register = [0 for i in range(32)]
13         gw.init(shift_register)
14         return (shift_register)
15
16     # Force message to be correct length of bytes in ASCII
17     def parse_input(self, input):
18         input = input.encode(encoding="ASCII")
19         message = bytearray(int(self.MSG_LEN/8))
20         memoryview(message)[0:(len(input))] = input
21         return message
22
23     # Encode a message using the BBC algorithm
24     def encode(self, input):
25         message = self.parse_input(input)
26         codeword = bytearray(int(self.COD_LEN/8))
27         for i in range(self.MSG_LEN):
28             # ASCII byte to be encoded
29             element = memoryview(message)[int((i-i%8)/8)]
30             bit = ((element) >> (i%8)) & 0b1
31             # Extract bit from Byte
32             mark_loc = gw.add_bit(bit, self.shift_register) % self.COD_LEN
33             # Mark location from glowworm
34             memoryview(codeword)[int((mark_loc-mark_loc%8)/8)] |= 1<<(mark_loc%8)
35         return(codeword)
```

A.3 Decoder Class

```python
1  from math import ceil
2  import glowworm.py as gw
3
4  class Decoder:
5      # Store configuration parameters as static variables
6      def __init__(self, MSG_LEN, COD_LEN, CHK_LEN):
7          # <Constructor here>
8
9
10     # Allocate memory for and initialize the shift register
11     def init_shift_register(self):
12         shift_register = [0 for i in range(32)]
13         gw.init(shift_register)
14         return (shift_register)
15
16     # Use novel iterative approach to decoding a packet
17     def decode(self, packet):
18
19         # Initialize variables
20         self.message_list = []
21         message = bytearray(ceil((self.MSG_LEN + self.CHK_LEN)/8))
22
23         while True:
24             # Find the proposed bit from previous execution, or a 0 if initial iteration
25             prop_bit = (memoryview(message)[int((self.n - self.n%8)/8)]>>(self.n%8)) & 0b1
26
27             # Find the corresponding mark location from glowworm
28             val = gw.add_bit(prop_bit, self.shift_register) % (self.COD_LEN)
29
30             # Logical AND to determine if present in packet/codeword
31             bit = (memoryview(packet)[int((val-val%8)/8)]>>(val%8)) & 0b1
32
33             # If the mark is present, explore
34             if bit==1:
35                 # Message is complete, write to buffer
36                 if self.n == (self.MSG_LEN + self.CHK_LEN - 1):
37                     self.message_list.append(bytes(memoryview(message)\
38                                                 [0:self.MSG_LEN - 1 - self.CHK_LEN]))
39                     bit = 0
40
41                 # Message is incomplete, continue assuming next bit is 0
42                 elif self.n < (self.MSG_LEN + self.CHK_LEN - 1):
43                     self.n += 1
44                     memoryview(message)[int((self.n - self.n%8)/8)] &= (0xff ^ (1<<self.n%8))
45                     continue
46
47             # If the mark is not present, backtrace
48             if bit!=1:
49                 # Delete checksum bits
50                 while self.n >= self.MSG_LEN:
51                     gw.del_bit(0, self.shift_register)
52                     self.n -= 1
53
54                 # Delete 1's until a 0 is encountered
55                 while self.n >=0 and (((memoryview(message)[int((self.n - self.n%8)/8)]>>\
56                                                     (self.n%8)) & 0b1 )==1):
57                     gw.del_bit(1, self.shift_register)
58                     memoryview(message)[int((self.n - self.n%8)/8)] &= (0xff ^ (1<<self.n%8))
59                     self.n -= 1
60
61                 # Packet is fully decoded, proceed with next packet
62                 if self.n < 0:
63                     break
64
65                 # Move over to the 1 branch of current search
66                 else:
67                     gw.del_bit(0, self.shift_register)
68                     memoryview(message)[int((self.n - self.n%8)/8)] |= (1<<self.n%8)
69
70         return self.message_list
```

APPENDIX B
GNU RADIO IMPLEMENTATION

B.1 Top-Level Chart
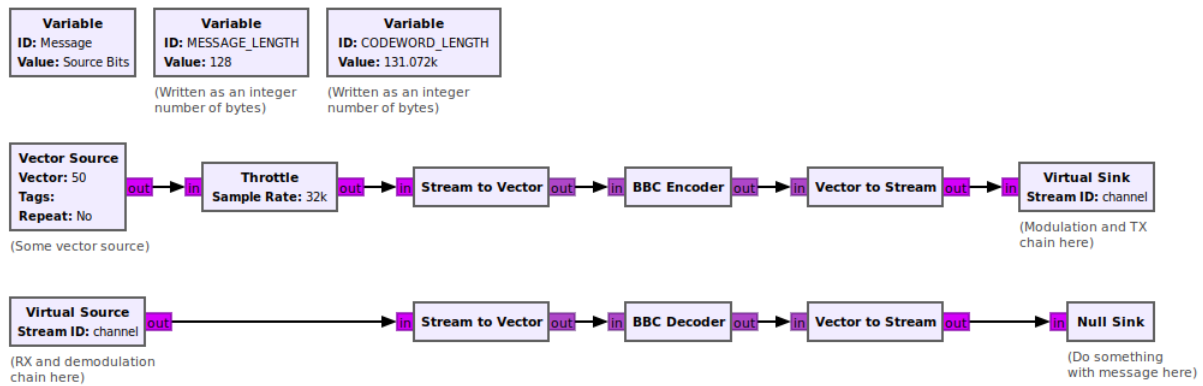
B.2 GNU Radio Encoder Block Class

```python
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 GNU Radio Decoder Block Class

```
1  from gnuradio import gr
2  import numpy as np
3
4
5  class blk(gr.interp_block):
6
7      # Configure port sizes
8      def __init__(self, MESSAGE_LENGTH=2**7, CODEWORD_LENGTH=2**17):
9          self.msg_len = MESSAGE_LENGTH
10         self.cod_len = CODEWORD_LENGTH
11
12         # Use an interpolation block
13         gr.sync_block.__init__(self,
14             name='BBC Decoder',
15             in_sig =[(np.byte, self.cod_len)],
16             out_sig=[(np.byte, self.msg_len)])
17
18         # Initialize the interpolation rate to synchronous case
19         self.set_relative_rate(1)
20
21         # Convert from Bytes to bits
22         self.myDecoder = Decoder(self.msg_len*8, self.cod_len*8, DEFAULT_CHECKSUM)
23
24
25     # Use BBC to decode the incoming codeword vectors
26     def work(self, input_items, output_items):
27
28         # Pull packet from the queue
29         packet = input_items[0][:][0]
30
31         # Check for a nonzero codeword
32         if sum(packet >0):
33
34             result = self.myDecoder.decode(packet)
35             interp = len(result)
36
37             #Check for a decoded message
38             if interp > 0:
39
40                 # Definre the number of outputs
41                 self.set_relative_rate(interp)
42
43                 # Assign outputs iteratively
44                 for j in range(interp):
45                     try:
46                         output_items[0][j][:] = result[j]
47                     except:
48                             raise RuntimeError("BBC Encoder output assignment failed")
49         return len(output_items)
```

Appendix C
GNU Radio BBC Channel Simulation

C.1 Multi-Codeword BBC

Options
Title: BBCCodec
Author: James Morrison
Output Language: Python
Generate Options: No GUI
Run Options: Prompt for Exit

Variable
ID: CODEWORD_LENGTH
Value: 1.024k

Must write as int
Subtract 3 from exponent
since in Bytes
131072 = 2**17
512 = 2**9

Variable
ID: MESSAGE_LENGTH
Value: 128

Must write as int
Subtract 3 from exponent
since in Bytes
128 = 2**7
8 = 2**3

Variable
ID: message1
Value: HELLO WO...nd then try

Variable
ID: message2
Value: ZELLO ZO...nd zhen zry

Null Source → Throttle Sample Rate: 32k → Null Sink

Vector Source — Vector: [ord(i) for i in m... Tags: Repeat: Yes → Head Num Items: 256 → Stream to Vector → BBC Encoder → Vector to Stream → Or → Virtual Sink Stream ID: bytes

BBC Requires vectors for operation

Vector Source — Vector: [ord(i) for i in m... Tags: Repeat: No → Head Num Items: 256 → Stream to Vector → BBC Encoder → Vector to Stream

BBC Requires vectors for operation

Virtual Source Stream ID: bytes → Stream to Vector → BBC Decoder (out / msgOutput) → Vector to Stream → ZMQ PUSH Sink Address: tcp://127.0.0.1:5555 Timeout (msec): 100 Pass Tags: No

Message Debug PDU Vectors: On (print / store / print_pdu)

Properties: Vector Source
General | Advanced | Documentation
Output Type: byte
Vector: [ord(i) for i in message1]
Tags: []
Repeat: Yes
Vector Length: 1

...ELLO WORLD! Welcome to BBC in GNURadio. This is a jam-resistant codec, and we are sending messages, encoding them, and then try
...ELLO ZORLD! Zelcome zo ZBC zn ZNURadio. Zhis zs z zam-zesistant zodec, znd ze zre zending zessages, zncoding zhem, znd zhen zry
****** MESSAGE DEBUG PRINT ********
...ELLO WORLD! Welcome to BBC in GNURadio. This is a jam-resistant codec, and we are sending messages, encoding them, and then try
****
****** MESSAGE DEBUG PRINT ********
...ELLO ZORLD! Zelcome zo ZBC zn ZNURadio. Zhis zs z zam-zesistant zodec, znd ze zre zending zessages, zncoding zhem, znd zhen zry
****

C.2 Noise Injection

Options
Title: BBCCodec
Author: James Morrison
Output Language: Python
Generate Options: No GUI
Run Options: Prompt for Exit

Variable
ID: CODEWORD_LENGTH
Value: 131.072k

Must write as int
Subtract 3 from exponent
since in Bytes
131072 = 2**17
512 = 2**9

Variable
ID: MESSAGE_LENGTH
Value: 128

Must write as int
Subtract 3 from exponent
since in Bytes
128 = 2**7
8 = 2**3

Variable
ID: message1
Value: HELLO WO...nd then try

Null Source → Throttle Sample Rate: 32k → Null Sink

Vector Source — Vector: [ord(i) for i in m... Tags: Repeat: Yes → Head Num Items: 256 → Stream to Vector → BBC Encoder → Vector to Stream → Or → Virtual Sink Stream ID: bytes

BBC Requires vectors for operation

Vector Source — Vector: 26 Tags: Repeat: Yes

00011010

Virtual Source Stream ID: bytes → Stream to Vector → BBC Decoder (out / msgOutput) → Vector to Stream → ZMQ PUSH Sink Address: tcp://127.0.0.1:5555 Timeout (msec): 100 Pass Tags: No

Message Debug PDU Vectors: On (print / store / print_pdu)

C.3 BBC with On-Off Keying

Options
Title: BBCCodec
Author: James Morrison
Output Language: Python
Generate Options: No GUI
Run Options: Prompt for Exit

Variable
ID: CODEWORD_LENGTH
Value: 1.024k

Must write as int
Subtract 3 from exponent
since in Bytes
131072 = 2**17
512 = 2**9

Variable
ID: MESSAGE_LENGTH
Value: 128

Must write as int
Subtract 3 from exponent
since in Bytes
128 = 2**7
8 = 2**3

Variable
ID: samp_rate
Value: 128k

Variable
ID: symbol_rate
Value: 500

Variable
ID: carrier_freq
Value: 433.937M

Variable
ID: message2
Value: ZELLO ZO...nd zhen zry

Vector Source
Vector: [ord(i) for i in m...
Tags:
Repeat: No

Throttle
Sample Rate: 128k

Stream to Vector

BBC Encoder

BBC Requires vectors
for operation

Vector to Stream

Variable
ID: message1
Value: HELLO WO...nd then try

Vector Source
Vector: [ord(i) for i in m...
Tags:
Repeat: No

Throttle
Sample Rate: 128k

Stream to Vector

BBC Encoder

BBC Requires vectors
for operation

Vector to Stream

Or

OOK Modulator
carrier_freq: 433.937M
samp_rate: 128k
symbol_rate: 500

Virtual Sink
Stream ID: tx_rx

Virtual Source
Stream ID: tx_rx

OOK Demodulator
samp_rate: 128k
symbol_rate: 500

Stream to Vector

BBC Decoder

Vector to Stream

Null Sink

C.4 On-Off Keying Modulator and Demodulator

Options
Title: OOK Modulator
Author: James Morrison
Output Language: Python
Generate Options: Hier Block
Category: [GRC Hier Blocks]

Parameter
ID: samp_rate
Value: 128k

Parameter
ID: symbol_rate
Value: 500

Parameter
ID: carrier_freq
Value: 433.937M

Variable
ID: bits_per_pack
Value: 8

Pad Source
Label: in

Unpack K Bits
K: 8

UChar To Float

Repeat
Interpolation: 256

Float To Complex

Multiply

Pad Sink
Label: out

Signal Source
Sample Rate: 128k
Waveform: Cosine
Frequency: 433.937M
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Options
Title: OOK Demodulator
Author: James Morrison
Output Language: Python
Generate Options: Hier Block
Category: [GRC Hier Blocks]

Parameter
ID: samp_rate
Value: 128k

Parameter
ID: symbol_rate
Value: 500

Pad Source
Label: in

Band Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 128k
Low Cutoff Freq: 1
High Cutoff Freq: 50k
Transition Width: 20k
Window: Hamming
Beta: 6.76

Complex to Mag^2

Threshold
Low: 1m
High: 1m
Initial State: 0

Keep 1 in N

Float To Char
Scale: 1

Unpacked to Packed
Bits per Chunk: 1
Endianness: MSB

Pad Sink
Label: out

C.5 BBC-FHSS Seeding

Virtual Source
Stream ID: ook_tx

OOK Demodulator
samp_rate: 128k
symbol_rate: 500

Stream to Vector

BBC Decoder
frequencyCmd

Null Sink
Vector Length: 64

Message Debug
PDU Vectors: On
print
store
print_pdu

Signal Source
Sample Rate: 128k
Waveform: Cosine
Frequency: 100
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0
cmd

Rational Resampler
Interpolation: 100
Decimation: 1
Taps:
Fractional BW: 0

Frequency Xlating FIR Filter
Decimation: 1
Taps: 1
Center Frequency: 0
Sample Rate: 128k
freq

QT GUI Waterfall Sink
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 128k