BBC Communications in GNURadio

N. Rogers and J. Morrison

United States Air Force Academy
Department of Electrical and Computer Engineering

September 2022
OVERVIEW

- BBC Overview
- GNURadio implementation
- Lessons learned
- Future Development
**WHAT IS BBC?**

- Keyless
- Concurrent-codes
- Spread spectrum
- Jam-resistant (not LPI)
- Invertable hash algorithm encoding
- Limitation: Low throughput
- Limitation: modulation type
- Limit: Mark Density

No pre-shared key
What is BBC?

- Keyless
- Concurrent-codes
- Spread spectrum
- Jam-resistant (not LPI)
- Invertable hash algorithm encoding
- Limitation: Low throughput
- Limitation: modulation type
- Limit: Mark Density

Can encode multiple messages in a single codeword
**WHAT IS BBC?**

- Keyless
- Concurrent-codes
- **Spread spectrum**
- Jam-resistant (not LPI)
- Invertable hash algorithm encoding
- Limitation: Low throughput
- Limitation: modulation type
- Limit: Mark Density

Transmitted signal size greater than message
**What is BBC?**

- Keyless
- Concurrent-codes
- Spread spectrum
- Jam-resistant (not LPI)
- Invertable hash algorithm encoding

→ Cannot remove information that exists

→ Additional information in codeword can be ignored (to an extent)
## WHAT IS BBC?

- **Keyless**
- **Concurrent-codes**
- **Spread spectrum**
- **Jam-resistant (not LPI)**
- **Invertable hash algorithm encoding**
- **Limitation: Low throughput**
- **Limitation: modulation type**
- **Limit: Mark Density**
What is BBC?

- Keyless
- Concurrent-codes
- Spread spectrum
- Jam-resistant (not LPI)
- Invertable hash algorithm encoding
  - Limitation: Low throughput
  - Limitation: modulation type
- Limit: Mark Density
**WHAT IS BBC?**

- Keyless
- Concurrent-codes
- Spread spectrum
- Jam-resistant (not LPI)
- Invertable hash algorithm encoding
- Limitation: Low throughput
- Limitation: modulation type
- Limit: Mark Density

\[ \mu = \frac{n_{\text{bits, message}}}{n_{\text{bits, codeword}}} \]

Limit $\rightarrow 0.5$
USE CASE - FHSS SEQUENCE

Start → Seed → FHSS → Dropout? → yes → Look for update → BBC Exchange

no → Happy Dance
**Encoding – The Glowworm Hash**

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example →
  - Message = 0011 0010
  - Codeword = 16 bits

---

**Encoding – the Glowworm Hash**

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example ->
  - Message = 0011 0010
  - Codeword = 16 bits

---

ENCODING – THE GLOWWORM HASH

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example →
  Message = 0011 0010
  Codeword = 16bits

---

ENCODING – THE GLOWWORM HASH

• Input is length of codeword

• Output is “sparse” codeword

• Shift register produces mark locations in codeword

• Shift register values depend on current substring

• Example →
Message = 0011 0010
Codeword = 16bits

ENCODING – THE GLOWWORM HASH

• Input is length of codeword
• Output is “sparse” codeword
• Shift register produces mark locations in codeword
• Shift register values depend on current substring

Example →
Message = 0011 0010
Codeword = 16bits
ENCODING – THE GLOWWORM HASH

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example —→ Message = 0011 0010 Codeword = 16bits

MESSAGE:

```
0 0 1 1 0 0 0 0
0 1 2 3 4 5 6 7
```

GLOWWORM HASH: 5

CODEWORD:

```
0 0 0 0 1 0 0 0 0
0 1 2 3 4 5 6 7 ...
0 0 0 0 0 0 0 0 0
8 9 1 11 12 13 14 15
```
**Encoding – the Glowworm Hash**

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring

**Example →**
Message = 0011 0010
Codeword = 16bits

**Message:**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**Glowworm Hash:** 0

**Codeword:**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>
ENCODING – THE GLOWWORM HASH

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example →
  Message = 0011 0010
  Codeword = 16bits

MESSAGE:

0 0 1 1 0 0 0 0
0 1 2 3 4 5 6 7

GLOWWORM HASH: 14

CODEWORD:

1 0 0 0 1 0 0 0
0 1 2 3 4 5 6 7 . . .
0 0 0 0 0 0 0 1
8 9 1 11 12 13 14 15

Rogers and Morrison
**Encoding – the Glowworm Hash**

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring

**Example ➔**

Message = 0011 0010
Codeword = 16bits

---

**MESSAGE:**

```
0 0 1 1 0 0 0 0
```

**GLOWWORM HASH:** 2

**CODEWORD:**

```
1 0 1 0 1 0 0 0 0
```

---

Rogers and Morrison
Encoding – The Glowworm Hash

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example →
  Message = 0011 0010
  Codeword = 16bits

Message: 0 0 1 1 0 0 0 0

Glowworm Hash: 0

Codeword: 1 0 1 0 1 0 0 0 0

0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
8 9 1 11 12 13 14 15
0 0 0 0 0 0 1 0
**ENCODING – THE GLOWWORM HASH**

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example — Message = 0011 0010
  Codeword = 16bits

**MESSAGE:**

```
0 0 1 1 0 0 0 0
0 1 2 3 4 5 6 7
```

**GLOWWORM HASH:** 0

**CODEWORD:**

```
1 0 1 0 1 0 0 0 0
0 1 2 3 4 5 6 7 ...
0 0 0 0 0 0 1 0
8 9 1 11 12 13 14 15
```
ENCODING – THE GLOWWORM HASH

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example
  Message = 0011 0010
  Codeword = 16bits

MESSAGE:

GLOWWORM HASH: 3

CODEWORD:
ENCODING – THE GLOWWORM HASH

- Input is length of codeword
- Output is “sparse” codeword
- Shift register produces mark locations in codeword
- Shift register values depend on current substring
- Example — Message = 0011 0010
  Codeword = 16bits

MESSAGE:

GLOWWORM HASH: 12

CODEWORD:
**Encoding – the Glowworm Hash**

- Input is length of codeword

- Output is “sparse” codeword

- Shift register produces mark locations in codeword

- Shift register values depend on current substring

- Example →
  
  Message = 0011 0010
  
  Codeword = 16bits

- $\mu = \frac{b_{message}}{b_{codeword}} = \frac{6}{16} = 0.375$

---

**MESSAGE:**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**GLOWWORM HASH:**

**CODEWORD:**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
**DECODING**

- **Depth-First Search:**
  - Intuitive to implement,
  - Stack depth
  - Inefficient use of memory

- **Breadth-First Search**
  - More difficult to implement
  - Memory / CPU efficient
# JAM-RESISTANCE

Jamming acts as a bit-wise “OR”

<table>
<thead>
<tr>
<th>Origin</th>
<th>ASCII Character</th>
<th>BBC Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original message (TX)</td>
<td>2</td>
<td>10011100 00001010</td>
</tr>
<tr>
<td>Jamming signal</td>
<td>!</td>
<td>01000001 10001101</td>
</tr>
<tr>
<td>Channel packet (OR)</td>
<td>N/A</td>
<td>11011101 10001111</td>
</tr>
</tbody>
</table>

- Broadband noise attack → enough energy wins (always)
- Random marks add additional messages, don’t affect existing
- Checksum bits protect against hallucinations
HELLO WORLD! Welcome to BBC in GNURadio. This is a jam-resistant codec, and we are sending messages, encoding them, and then try ZELLO ZORLD! Welcome to BBC in GNURadio. Zhis zs z zam-zesistant zodec, znd zez zez zending zessages, znoding zhem, znd zhen zry
Details

- **Encoder**
  - Input = bytes vectors of length MESSAGE LENGTH
  - Output = bytes vectors of length CODEWORD LENGTH

- **Decoder**
  - Input = bytes vectors of length CODEWORD LENGTH
  - Output = bytes vectors of length MESSAGE LENGTH

HELLO WORLD! Welcome to BBC in GNURadio. This is a jam-resistant codec, and we are sending messages, encoding them, and then try ZELLO ZORLSD Zelcone zo ZBC in ZNURadio. Zhis zs z zam-zesistant zodec, znd ze zre zending zessages, zncoding zhem, znd zhern zry...
BASIC TESTING – PACKET DENSITY LIMIT

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 WORLD then try

Null Source -> in -> Throttle
Sample Rate: 32k -> out -> in -> Null Sink

Vector Source
Vector: [(ord(i) for i in m...]
Tags:
Repeat: Yes

Head
Num Items: 256

Stream to Vector
in -> out

BBC Encoder
out -> in

Vector to Stream
in -> out

Vector Source
Vector: 26
Tags:
Repeat: Yes

Virtual Sink
Stream ID: bytes

Virtual Source
Stream ID: bytes

Stream to Vector
in -> out

BBC Decoder
msgOutput -> out

Vector to Stream
in -> out

Message Debug
PDU Vectors: On

ZMQ PUSH Sink
Address: tcp://127.0.0.1:5555
Timeout (msec): 100
Pass Tags: No
**Basic Testing – Packet Density Limit**

### Details

- Replaced message2 with vector source
- Idea is to test $\mu = 50\%$ mark density
- $26_{10} = 00011010_2$ (hovering around limit) **PASS**
- $27_{10} = 00011011_2$ (over limit) **FAIL**
OOK Implementation

- Chose OOK for simplicity
- Complements jam-resistance of BBC
OOK Implementation

- OOK Hier blocks
- Sends codeword as bytes over message port as PMT.intern()
- BBC Decoder modified to receive bytes/re-assemble codewords
- Outputs messages
USE CASE - FHSS

Virtual Source
Stream ID: ook_bx

OOK Demodulator
samp_rate: 128k
symbol_rate: 500

Stream to Vector

BBC Decoder
frequencyCmd

Null Sink
Vector Length: 64

Message Debug
PDU Vectors: On

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

time

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

Frequency Xampling FIR Filter
Decimation: 1
Center Frequency: 0
Sample Rate: 128k

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

- out_sig can be a challenge
- Dynamic sizing of in/out vectors
- Sometimes, you just need a PMT
- Block development documentation difficult to follow (but is getting better!)
- Future goal is to output bytes to out_sig
SUMMARY & WHAT’S NEXT?

- Fix bytes output
- Package for CGAN
- Performance benchmark testing
- Hardware testing/implementation
- Active statistical thresholding
- Multimark BBC
- Codeword detection using ring buffer