Verification of a Synthesisable Reed-Solomon ECC Core

David Banks
Publishing Systems and Solutions Laboratory
HP Laboratories Bristol
HPL-2001-125

E-mail: dmb@hplb.hpl.hp.com

Reed-Solomon, error correction, verilog, synthesis, synthesisable, hardware, ECC, Galois field

In this report we describe the verification of a Reed-Solomon error correction core that supports errors and erasures decoding. In a second report HPL-2001-124 we describe the design of this core.

The verification was performed using both simulation and prototyping.

The simulation environment consisted of automatic test vector (codeword) generation for a variety of tests, unit delay simulation of a gate-level netlist in Verilog-XL, and comparison of the simulation results against an independently developed Reed-Solomon ECC model written in C. The prototyping environment consisted of a Xilinx FPGA containing the ECC block with a flexible pattern generator, together with circuitry for adding errors and erasures, and circuitry for accumulating test results. Tests were configured using a C program (running under Linux), which communicated with the hardware under test using a standard parallel port interface.

Overall, we ran 1,147,000 vectors through the simulation, and 10,176,000,000 random vectors through the prototype. No failures were detected.

© Copyright Hewlett-Packard Company 2001 Approved for External Publication

1 INTRODUCTION......4

2 VERILOG SIMULATION......5

2.1 METHODOLOGY......5

2.1.1 Overall block diagram......5

2.1.2 Vector generation......6

2.1.3 Verilog simulation......9

2.1.4 Gadiel's reference encoder and decoder......11

2.1.5 Status codes......12

2.1.6 Comparing results......13

2.1.7 Top level shell script......15

2.1.8 File system organization......16

2.2 SPECIFIC TESTS......17

2.2.1 Test A - Random error and erasure combinations (32)......18

2.2.2 Test A - Random errors and erasure combinations (20)......19

2.2.3 Test B - Realistic data (32)......20

2.2.4 Test C - All error and erasure combinations (32)......21

2.2.5 Test C - All error and erasure combinations (20)......22

2.2.6 Test D - Uncorrectable error detection (32)......23

2.2.7 Test D - Uncorrectable error detection (20)......24

2.2.8 Test E - Sensitivity to erasure positions (32)......25

2.2.9 Test F - Sensitivity to gaps between codewords (32)......26

3 FPGA PROTOTYPE......27

3.1 METHODOLOGY......27

3.1.1 Overall prototype block diagram......27

3.1.2 Hardware design......28

3.1.3 Hardware implementation......41

3.1.4 Test software......45

3.2 SPECIFIC TESTS......50

3.2.1 Test A – Random error and erasure combinations......50

3.2.2 Test B – Realistic data......52

3.2.3 Test C – All error and erasure combinations......53

3.2.4 Test D – Uncorrectable error detection......54

3.2.5 Test E – Sensitivity to erasure positions......56

3.2.6 Test F – Random error and erasure combinations (limited maxerasures)......57

4 VERIFICATION OF OTHER REED-SOLOMON CODES......60

4.1 CODES IN GF (2^8)......60

4.1.1 Simulation......60

4.1.2 Prototype......60

4.2 CODES IN FIELDS OTHER THAN GF (2^8)......60

4.2.1 Simulation......60

4.2.2 Prototype......61

5 REFERENCES......62

6 APPENDIX A – MISCORRECT PROBABILITIES......63

7 APPENDIX B – `CALC_SYMBOL_PROBS()`......64

8 APPENDIC C – `CHANNEL_FIXED_INIT()` 65

Figure 1 – Overall simulation block diagram 5
Figure 2 – Overall prototype block diagram 27
Figure 3 - FPGA block diagram.. 28
Figure 4 - Clumped erasures – number of erasures distribution.................................. 35
Figure 5 - Clumped erasures - symbol error distribution 35
Figure 6 - The EVALXC2SVE-HQ240 prototyping poard.. 42
Figure 7 - The PWR3 power module 42
Figure 8 – Custom parallel cable construction 44
Figure 9 – Graph of expected verses observed miscorrection probabilities 59

1 Introduction

This document describes the verification of the Reed-Solomon ECC block designed by HP Labs Bristol. For further details on the design of the ECC block, please refer to [1]. The verification was performed using both simulation and prototyping.

The simulation environment consisted of automatic test vector generator for a variety of tests, unit delay simulation of a gate-level netlist in Verilog-XL, and comparison of the simulation results against an independently developed Reed-Solomon ECC model written in C.

The prototyping environment consisted of an FPGA containing the ECC block with a flexible pattern generator, together with circuitry for adding errors and erasures, and circuitry for accumulating test results. Tests are configured by a C program (running under Linux), which communicates with the hardware under test using a standard parallel port interface.

The remainder of this document is structured as follows. Section 2 describes in detail the simulation methodology and the specific tests run. Section 3 describes in detail the prototyping methodology and the specific tests run. Finally, in section 4 we discuss the issues that may arise in re-using this verification infrastructure for Reed-Solomon codes other than the RS(160,128,t=16) code of interest to us.

2 Verilog simulation

2.1 Methodology

2.1.1 Overall block diagram

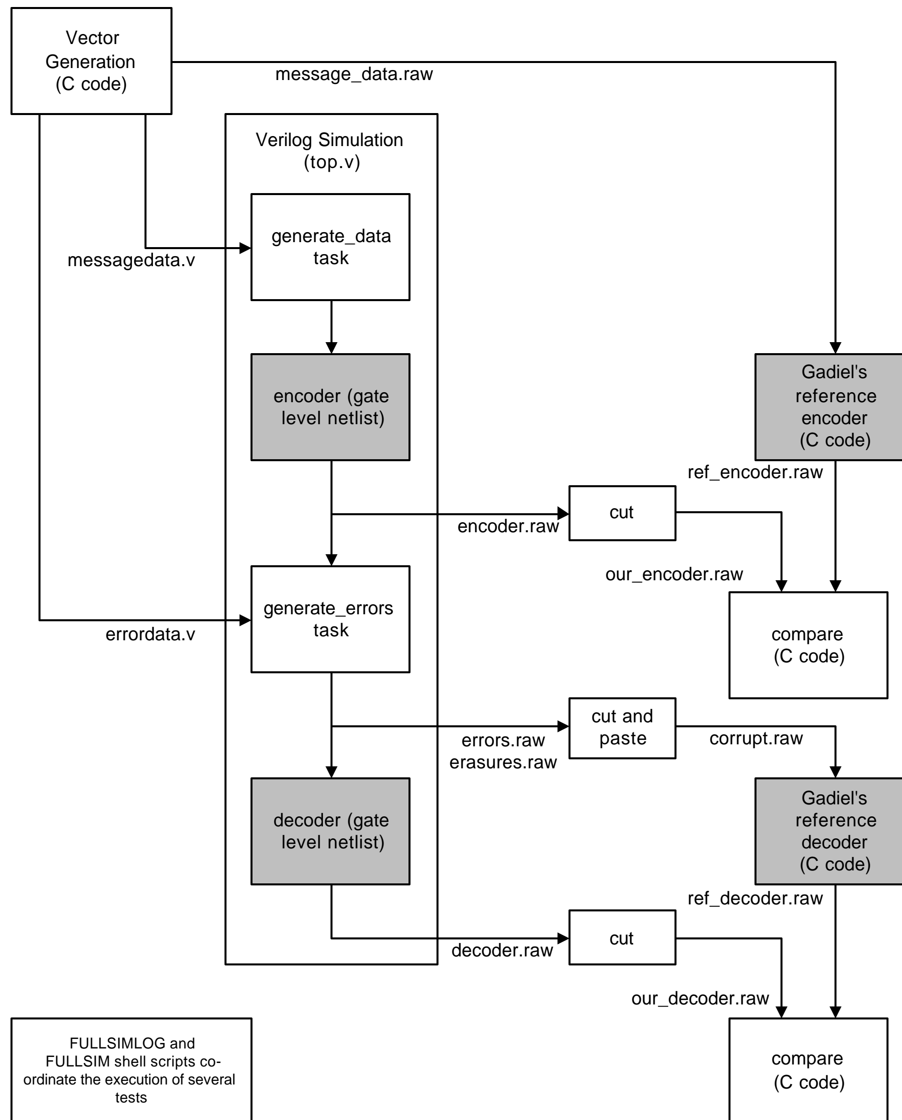

Figure 1 – Overall simulation block diagram

The overall simulation methodology is illustrated in Figure 1. The `vectors` C program can generate test vectors for several tests (testa, testb, … etc), detailed later in this section. The file names shown in Figure 1 are actually prefixed with the test name, so that the results of previous simulations are not overwritten.

For each named test, a verilog simulation is run which includes two external files (test_messagedata.v and test_errordata.v) generated by the `vectors` C program; one specifies the 128-byte information blocks to be encoded (and the gaps between them). The other specifies the 160-byte error patterns to apply to the encoded data. The

simulation generates four files: the output of the encoder (encoder.raw) the corrupted data and erasures positions (errors.raw and erasures.raw) and finally the output of the decoder (decoder.raw).

The results of the verilog simulation are checked against an independently coded Reed-Solomon encoder and decoder, provided by Gadiel Seroussi of HP Labs Palo Alto (for more details on this design see [2], which is also available directly from Gadiel). The same data set is run through Gadiel's encoder and decoder, and a simple C program is used to compare the output status codes and corrected data.

In order to simplify the execution of several back-to-back tests, we have provided a simple shell script (FULLSIM), which iterates through the different tests, renaming files where necessary, etc.

On a Pentium III 700 MHz Linux box, running Cadence's Verilog-XL 3.11.p001, we achieve a simulation performance of one vector every two seconds. We found gate-level simulation considerably (3x-6x) faster than RTL simulation, as long as it was done in unit delay mode (the +delay_mode_unit) flag. This was also considerably faster (3x-6x) than using ModelSim 3.4a on the same platform.

The remainder of this sub-section describes each of the components of the simulation environment in greater detail, giving specific examples where these are helpful.

2.1.2 Vector generation

The vector generation phase generates a specified number of vectors for each of six possible tests (testa, testb, testc, testd, teste and testf), which are detailed in the next section. To generate a set of vectors, the `vectors` C program must be run. The command line option syntax for `vectors` is:

```
vectors [-a <num>] [-b <num>] [-c <base num>]
        [-d <base num>] [-e <num>] [-f <num>]
        [-s <scale factor>] [-r <random seed>] [-v <verbosity>]
        [-B <code block size>]
        [-T <code correction capability>]
        [-W <code symbol width>]
```

-a The number of vectors to generate for testa (default 100).

-b The number of vectors to generate for testb (default 100).

-c The base number of vectors to generate for testc. The actual number of vectors will be 289 times the base for the RS (160,128,t=16) code, since this test iterates through every correctable combination of errors and erasures (default 1).

-d The base number of vectors to generate for testd. The actual number of vectors will be 45.8 times the base for the RS (160,128,t=16) code, since this test iterates through all possible numbers of erasures (default 10).

-e The number of vectors to generate for teste (default 100).

-f The number of vectors to generate for testf (default 100).

Thus, the default number of vectors generated is:

100 + 100 + 289 + 458 + 100 + 100 = 1,147

A typical simulation run of this length will take about 40 minutes.

-s A scale factor to apply to the number of vectors for each test (default 1).

Thus, if –s1000 was used, a total of 1,147,000 vectors would be generated. A simulation run of this length would take about a week.

-r A random seed, so that (if required) different vector sets can be generated (default 21011967).

-v The verbosity (0,1,2 or 3) when generating vectors (default 1)

The Reed Solomon code to be tested is specified using the following options:

-B The block size of the Reed-Solomon code to be tested (default 160).

-T The number of errors the Reed-Solomon code can correct (default 16).

-W The symbol width (in bits) of the Reed-Solomon code (default 8). Currently only values of 4 and 8 are supported.

The default code is an RS (160, 128, T=16) in GF (2^8).

The result of executing the `vectors` (with all the default options) would be the following files written to the current directory

```
% ./vectors
Running testa for 100 vectors
Running testb for 100 vectors
Running testc for 1 vectors per combination
Running testd for 10 vectors per erasure
Running teste for 100 vectors
Running testf for 100 vectors
Total number of vectors = 1147
% ls -l test*
-rw-r--r--   1 dmb      users         952 Mar  8 11:41 testa.log
-rw-r--r--   1 dmb      users       51100 Mar  8 11:41 testa_errordata.v
-rw-r--r--   1 dmb      users       51300 Mar  8 11:41 testa_messagedata.raw
-rw-r--r--   1 dmb      users       28200 Mar  8 11:41 testa_messagedata.v
-rw-r--r--   1 dmb      users         941 Mar  8 11:41 testb.log
-rw-r--r--   1 dmb      users       51100 Mar  8 11:41 testb_errordata.v
-rw-r--r--   1 dmb      users       51300 Mar  8 11:41 testb_messagedata.raw
-rw-r--r--   1 dmb      users       28200 Mar  8 11:41 testb_messagedata.v
-rw-r--r--   1 dmb      users        2973 Mar  8 11:41 testc.log
-rw-r--r--   1 dmb      users      147679 Mar  8 11:41 testc_errordata.v
-rw-r--r--   1 dmb      users      148257 Mar  8 11:41 testc_messagedata.raw
-rw-r--r--   1 dmb      users       81498 Mar  8 11:41 testc_messagedata.v
-rw-r--r--   1 dmb      users        5805 Mar  8 11:41 testd.log
-rw-r--r--   1 dmb      users      234038 Mar  8 11:41 testd_errordata.v
-rw-r--r--   1 dmb      users      234954 Mar  8 11:41 testd_messagedata.raw
-rw-r--r--   1 dmb      users      129156 Mar  8 11:41 testd_messagedata.v
-rw-r--r--   1 dmb      users         890 Mar  8 11:41 teste.log
-rw-r--r--   1 dmb      users       51100 Mar  8 11:41 teste_errordata.v
-rw-r--r--   1 dmb      users       51300 Mar  8 11:41 teste_messagedata.raw
```

```
-rw-r--r--   1 dmb      users       28200 Mar  8 11:41 teste_messagedata.v
-rw-r--r--   1 dmb      users        1090 Mar  8 11:41 testf.log
-rw-r--r--   1 dmb      users       51100 Mar  8 11:41 testf_errordata.v
-rw-r--r--   1 dmb      users       51300 Mar  8 11:41 testf_messagedata.raw
-rw-r--r--   1 dmb      users       28341 Mar  8 11:41 testf_messagedata.v
```

To understand what is in each file, we can use `vectors` to generate a very short run comprising of just three vectors for testa:

```
./vectors -a 3 -b 0 -c 0 -d 0 -e 0 -f 0
Running testa for 3 vectors
Total number of vectors = 3
```

The ".log" file comprises:

<vector number> <nerrors> <nerasures> <gap following this vector>

```
dbanks:dmb:[180] % cat testa.log
0 0 38 0
1 1 0 0
2 0 0 0
```

The "_messagedata.v" file is intended to be included directly in the verilog simulation using the `include directive. This file specifies the input to the encoder, and consists of a call to the task `generate_data (<message>, <gap>)` for each vector. In the below example, each vector is 1024 bits (i.e. 128 symbols) and there are no gaps between vectors.

```
dbanks:dmb:[181] % cat testa_messagedata.v
generate_data(1024'h9516f4223bc535bf7b4c8d18179d458bd43717bf4f1cf59999ba49c3ab16f9402c
ed6267b297262de4b345fb508a8624c19ee311bad8aa5393f3173e09107e35fde09caf77c2dc5b762257c6
acddeb6e7bce7f36a729893a1da07826b0f65cadd6f85c4dbb39a9315b00f707dde27559b1f48f581e1892
3bb90a616900bd17d6, 0);
generate_data(1024'hd4581ad411cea2ec7453bdc05ca925697c438203b5b5e7f7872b357162f40a364c
240b5ef2ad4a660007275db14cc62d8f483045fd173c844371f5a565ffdcb223e71016945a7c9461a3f112
efb8407f0070c4fe880082cb717870d6774c889b3398b1c7f22d5c54d14d66c005a63f061703049f03866a
74feda4b7627d3115a, 0);
generate_data(1024'hef7eb2c0cb1981d1bfc0d7d6c4db75c761df3c60ba87d6e15ae73bc6a95d2598db
d759a7f0da78b09a4f865e2afc268bdb62eb95e9c17643a8b20a510f2feaeb064392f71d0aa7b7592d1683
293c0e059efa9a87bb11ca64c3d4b5d2039fbd0ae24f01ff59a8b7b2d5cd35ff094404a73e9e2ef9aff85d
72cd1345d0b202da95, 0);
```

The "_errordata.v" file is also intended to be included in the verilog simulation. This file specifies how data is corrupted between the encoder and decoder, and consists of a call to the task `generate_error (<error pattern>, <erasure flags>)`. The error pattern field specifies the magnitude of errors and erasures, and the erasure flags field contains a single bit erasure flag for each symbol position. In the below example, each error pattern is 1280 bits (i.e. 160 symbols).

```
dbanks:dmb:[182] % cat testa_errordata.v
generate_error(1280'hfcf10000e7b1000000000000000000009f000000530000000000000018000000000
0af00710300000000000000000000000000331e00007300000000000098000000d20000000000000000000
000000000000000cd000000d286000b000000ea0000000721f700000000007c00cc790000004c850000000
0aa00b4000000000000000000000d00005700ec000000da00000000000000000000f20f5d0000000000,
160'b110011000000000010001000000100000101100000000000001100100000010001000000000000000
0010001101000100011100000101100011000010100000001001001010001000000000011100000);
generate_error(1280'h00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000c20000000000000000000000000000000000000000,
160'b000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000);
generate_error(1280'h00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000,
```

```
160'b00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000);
```

The final file generated is the "_messagedata.raw" file, which contains the same message data as the "_messagedata.v" file, but in a format suitable for Gadiel's reference Reed-Solomon encoder. In the below example, which contains three vectors, each vector consists of 128 decimal values. There is no explicit separator between vectors; instead symbols are simply read in blocks of 128.

```
dbanks:dmb:[183] % cat testa_messagedata.raw
149  22 244  34  59 197  53 191 123  76 141  24  23 157  69 139 212  55  23 191  79
28 245 153 153 186  73 195 171  22 249  64  44 237  98 103 178 151  38  45 228 179  69
251  80 138 134  36 193 158 227  17 186 216 170  83 147 243  23  62   9  16 126  53
253 224 156 175 119 194 220  91 118  34  87 198 172 221 235 110 123 206 127  54 167
41 137  58  29 160 120  38 176 246  92 173 214 248  92  77 187  57 169  49  91   0 247
7 221 226 117  89 177 244 143  88  30  24 146  59 185  10  97 105   0 189  23 214
212  88  26 212  17 206 162 236 116  83 189 192  92 169  37 105 124  67 130   3 181
181 231 247 135  43  53 113  98 244  10  54  76  36  11  94 242 173  74 102   0   7
39  93 177  76 198  45 143  72  48  69 253  23  60 132  67 113 245 165 101 255 220 178
35 231  16  22 148  90 124 148  97 163 241  18 239 184  64 127   0 112 196 254 136   0
130 203 113 120 112 214 119  76 136 155  51 152 177 199 242  45  92  84 209  77 102
192   5 166  63   6  23   3   4 159   3 134 106 116 254 218  75 118  39 211  17  90
239 126 178 192 203  25 129 209 191 192 215 214 196 219 117 199  97 223  60  96 186
135 214 225  90 231  59 198 169  93  37 152 219 215  89 167 240 218 120 176 154  79
134  94  42 252  38 139 219  98 235 149 233 193 118  67 168 178  10  81  15  47 234
235   6  67 146 247  29  10 167 183  89  45  22 131  41  60  14   5 158 250 154 135
187  17 202 100 195 212 181 210   3 159 189  10 226  79   1 255  89 168 183 178 213
205  53 255   9  68   4 167  62 158  46 249 175 248  93 114 205  19  69 208 178   2
218 149
```

In all of the above files, the standard Reed-Solomon convention of presenting the most significant symbol first is followed.

2.1.3 Verilog simulation

The verilog simulation comprises a top-level test harness (top.v) that instantiates an encoder and a decoder, using the gate-level netlists generated from Synopsys. The test harness also defines and implements two tasks: generate_data and generate_error.

2.1.3.1 Generate_data task

The definition of the generate_data task is listed below:

```
task generate_data;
input [WIDTH * (B - 2 * T) - 1 : 0]
    dinval;
input [31 : 0]
    gap;
integer
    i,
    j;
begin
    for (i = B - 2 * T - 1; i >= 0; i = i - 1) begin
        @(posedge clock && clocken);
        for (j = 0; j < WIDTH; j = j + 1)
            din[j] <=  dinval[WIDTH * i + j] ;
        if (i == (B - 2 * T - 1)) begin
            load <= 1;
        end else begin
            load <= 0;
        end
    end
    @(posedge clock && clocken);
    din <= 'bx;
    for (i = 0; i < 2 * T + gap - 1; i = i + 1)
        @(posedge clock && clocken);
end
endtask
```

This task takes two parameters: dinval, which specifies the message to be encoded, and gap, which specifies the number of extra idle clock cycles to append to the codeword. Note that a minimum of 2T idle clocks must be appended, to allow space for the 2T check symbols. During this idle period, din to the encoder is set to the "don't care" state, consequently if we ever mistakenly use this data, the simulation will rapidly fail.

2.1.3.2 Generate_error task

The definition of the generate_error task is listed below:

```
task generate_error;
input [WIDTH * B - 1 : 0]
    errorvals;
input [B - 1 : 0]
    erasurevals;
integer
    i,
    j;
reg [WIDTH - 1 : 0]
    errorval;
begin
    while (encsob != 1) begin
       errorerasure <= 'bx;
       errordout <= 'bx;
       @(posedge clock && clocken);
    end
    for (i = B - 1; i >= 0; i = i - 1) begin
        for (j = 0; j < WIDTH; j = j + 1)
            errorval[j] = errorvals[WIDTH * i + j] ;
        errorerasure <= erasurevals[i];
        errordout <= encdout ^ errorval;
        @(posedge clock && clocken);
    end
end
endtask
```

This task takes two parameters: errorvals, which specifies the corruption to be added to the codeword, and erasurevals, which specifies which symbols will be tagged as erasures. The task synchronizes itself with the output of the encoder using the encsob signal, which is asserted with the first symbol of the encoded codeword. The task writes to the errorerasure and errordout global registers, which are used as the input to the decoder. During the idle period between codewords, errorerasure and errordout are set to the "don't care" state, consequently if we ever mistakenly use this data, the simulation will rapidly fail.

2.1.3.3 Initialising the simulation

The simulation is initialised in a very conservative way, using the following verilog initial block:

```
initial
    begin
        ... stuff deleted ...
        // Start by letting everything get into a bad state
        din <= 'bx;
        load <= 'bx;
        reset <= 'bx;
        maxerasures <= 'bx;

        // Wait for it to get really bad
        for (i = 0; i < 50; i = i + 1)
            @(posedge clock && clocken);

        // Blip reset for a single clock cycle
        @(posedge clock && clocken);
```

```
        reset <= 1;
        @(posedge clock && clocken);
        reset <= 0;

        // Take load to something sensible as well
        load <= 0;
        maxerasures <= `MAXERASURES;

        // Wait one more clock before starting
        @(posedge clock && clocken);

        `include "test_messagedata.v"

        for (i = 0; i < 1000; i = i + 1)
            @(posedge clock && clocken);

        theend <= 1;

        for (i = 0; i < 10; i = i + 1)
            @(posedge clock && clocken);

        $finish;
    end
```

The most important thing to note is that the input signals to the simulation will be in the worst possible state (the don't care state) for many cycles prior to the reset, and that reset is asserted for a single cycle. This validates that a single cycle reset is sufficient to reset both the encoder and decoder.

2.1.3.4 Simulation input and output files

The input to the simulation is the following two files:

i. test_messages.v – a list of calls to generate_data, as generated by `vectors`.
ii. test_errordata.v – a list of calls to generate_error, as generated by `vectors`.

The output from the simulation is the following four files:

i. encoder.raw – containing the output of the encoder.
ii. errors.raw – containing the corrupted codewords.
iii. erasures.raw – containing the erasure positions.
iv. decoder.raw – containing the output of the decoder.

2.1.4 Gadiel's reference encoder and decoder

Gadiel Seroussi, of HP Labs Palo Alto, wrote the reference Read-Solomon encoder and decoder. This code has been well used in HP over a number of years.

Slight modifications to the outer "wrapper" of this code have been made in the following areas:

i. The format in which the data files are read and written has been changed. In the original code the data files were binary. Each symbol was represented by a single byte, limiting the maximum symbol size to 8 bits. We have instead adopted the above (.raw) format, where a white-space-separated decimal number represents each symbol. This, in principle, allows codes with symbols larger than 8 bits to be supported.[1]
ii. Although the original code fully implemented erasure decoding, this functionality was not supported in the outer "wrapper". We have modified the wrapper such that the input to the decoder comprises B symbols (the corrupted

[1] The original code was retained, and can be enabled by #defining BINARY in file rs.c.

data) followed by a further B symbols flagging erasures (0 = no erasure, 1 = erasure).[2]

iii. In the original code, the decoder output the corrected data only and not the additional check symbols. We have modified the wrapper to output the complete decoded codeword (data and check symbols) plus one additional symbol to indicate whether decoding was successful (0 = OK, 1 = fail).

iv. A –V option was added to allow the maximum number of erasures to be reduced from the code's maximum (the default) to a smaller value. This mimics the behaviour of our decoder.

For the RS (160, 128, t = 16) code, the following command must be executed to run the reference encoder:

```
rs –n160 –r32 <input_file.raw> <output_file.raw>
```

The file `<input_file.raw>` should contain the messages to be encoded (128 decimal symbols per message). The resulting codewords will be written to the file `<output_file.raw>` (160 decimal symbols per codeword). An example of the output is:

```
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
100 blocks processed, 12800 symbols in, 16000 symbols out
```

For the RS (160, 128, t = 16) code, the following command must be executed to run the reference decoder:

```
rs –d –n160 –r32 –V20 <input_file.raw> <output_file.raw>
```

The file `<input_file.raw>` should contain the corrupted codewords to be decoded (160 decimal symbols per codeword). The resulting corrected codewords will be written to the file `<output_file.raw>` (160 decimal symbols plus one status symbol per codeword). An example of the output is:

```
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
100 blocks processed, 63 OK, 37 failed
Chien searches: 75
12800 symbols out, 16000 symbols in, 543 corrected
```

2.1.5 Status codes

Gadiel's reference decoder uses the following status codes:

0 Correctable.
1 Uncorrectable.

Our decoder uses the following basic status codes:

0 Correctable, no errors, no erasures.
1 Correctable, no errors, some erasures.
2 Correctable, some errors, no erasures.
3 Correctable, some errors, some erasures.
4 Uncorrectable, no erasures.
5 Uncorrectable, some erasures.

[2] This new erasures functionality is enabled by #defining WITHERASURES in file rs.c.

Due to the internal architecture of the decoder, the status code is generated after the decoder has attempted to correct the corrupted codeword, and is available when the final symbol of the corrected codeword is being output. This minimises latency. Thus, regardless of whether the error pattern is correctable, or not, the decoder will always attempt to correct it.

As an additional check, the decoder re-calculates the syndromes over each sequence of symbols output by the decoder. This check is performed by the final pipeline stage within the decoder, called the *monitor* block.

If the status code was 0 to 3, the sequence of symbols output by the decoder should always correspond to a valid codeword (i.e. the syndromes will be zero). If this is not the case, the status code is replaced with 6.

If the status code was 4 or 5, the sequence of symbols output by the decoder is unlikely to be a valid codeword (i.e. one or more of the syndromes should be non-zero). If this is not the case, the status code is replaced with 7.

The status codes 6 and 7 should always be treated as uncorrectable.

More specifically:

> 6 Uncorrectable, special case 1. This represents the case where the status code going in to the monitor block was 0 to 3 (i.e. correctable), yet for some reason the syndrome of the sequence of symbols output by the decoder was non-zero, indicating an invalid codeword. This could indicate a design error in the decoder. It could also indicate that hardware is not operating reliably, say due to incorrect power supply voltages, or excessive system noise.
>
> 7 Uncorrectable, special case 2. This represents the case where the status code going in to the monitor block was 4 or 5 (i.e. uncorrectable), yet for some reason the syndrome of the sequence of symbols output by the decoder was zero, indicating a valid codeword. This event does occur in practise, particularly if the weight of the error pattern is 2T + 1 (i.e. just above what is correctable). Usually the codeword, whilst valid, is the wrong one. The only reason we expose this behaviour outside of the decoder is because it may help, in the future, us to design more effective decoders.

2.1.6 Comparing results

The final simulation step for each test is to compare the results of our verilog simulation with that of Gadiel's reference encoder and decoder. We have written a small C program, `compare`, for this purpose. The syntax for compare is:

```
compare <test.log> <ref_file.raw> <our_file.raw>
        [<encoder flag> [<B> <T> <maxerasures>]]
```

`<test.log>` should be the log file generated from `vectors` for this test.

`<ref_file.raw>` should be the output generated from Gadiel's reference encoder or decoder.

`<our raw file>` should be the output generated from the verilog simulation.

When checking the encoding process, an <encoder flag> of 1 must be specified, informing compare that the input files contain no status code, otherwise `compare` expects a status code to be appended.

`<B>` should match the block size of the code (e.g. 160).

`<T>` should be the error correction capability of the code (e.g. 16).

`<maxerasures>` should be the maximum number of allowable erasures (e.g. 20 or 32)

The following checks are performed:

i. *Encoder data check* – the codewords output from our encoder should match the codewords output from the reference encoder for all messages.

ii. *Decoder data check* – the corrected codewords from our decoder should match the corrected codewords from the reference decoder, for all the codewords that the reference decoder was able to correctly decode (i.e. this step is omitted for uncorrectable error patterns, since in this case the data output is undefined).

iii. *Decoder status check* – the following checks are applied to the status codes generated by the decoders.
   - a status code in the range 0..3 from our decoder should match a 0 status code from the reference.
   - a status code in the range 4..7 from our decoder should match a 1 status code from the reference decoder.

iv. Decoder sanity check – using the log file generated by `vectors` we perform some sanity checks on both the reference decoder and our decoder. This is an attempt to check there is no defect that affects both the reference decoder and our decoder. More specifically,
   - if an error pattern has 2 * nerrors + nerasures ≤ 2T and nerasures ≤ maxerasures, then it should be declared correctable. We check this is the case.
   - if an error patterns has 2 * nerrors + nerasures > 2T and nerasures ≤ maxerasures, then it may be miscorrected. We track the number of times this happens.
   - if an error pattern has nerasures > maxerasures, it should always be declared uncorrectable. We check this is the case.

An example run of `compare` for the encoder is illustrated below:

```
processed 1000 blocks
checking encoders against each other
    dcheckcount=1000 dfailcount=0
```

The status and data for all 1000 vectors was checked, no failures were detected.

An example run of `compare` for the decoder is illustrated below:

```
processed 1000 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=6
    refstatus=0 occurred 727 times
    refstatus=1 occurred 273 times
    miscorrect table:
        nerasures=32 occurred   14 times, miscorrected    6 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=6
    ourstatus=0 occurred 103 times
    ourstatus=1 occurred 164 times
    ourstatus=2 occurred 177 times
    ourstatus=3 occurred 283 times
    ourstatus=4 occurred 55 times
    ourstatus=5 occurred 217 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 1 times
    miscorrect table:
        nerasures=32 occurred   14 times, miscorrected    6 times
checking decoders against each other
    scheckcount=1000 sfailcount=0 dcheckcount=727 dfailcount=0
```

The status of all 1000 vectors was checked, no failures were detected. The data for the 727 codewords with correctable errors according to the reference decoder were compared against our decoder. Again, no failures were detected. The miscorrect table details where miscorrects occurred. All of the miscorrects correspond to the case where nerasures was 32, which is as expected.

2.1.7 Top level shell script

A simple top-level shell script (FULLSIM) is provided to run all six tests (testa-testf):

```
#!/bin/csh

set BLOCK=160
set R=32
set T=16

foreach name (testa_32 testa_20 testb_32 testc_32 testc_20 testd_32 testd_20 teste_32
testf_32)

set test=`echo $name | cut -d_ -f1`
set maxerasures = `echo $name | cut -d_ -f2`

echo "**************************************************"
echo "Running $test with maxerasures $maxerasures"
echo "**************************************************"

# Input files to the simulation
rm -f test_messagedata.v
rm -f test_errordata.v

# Output files from the simulation
rm -f encoder.raw
rm -f errors.raw
rm -f erasures.raw
rm -f decoder.raw

ln -s vectors/${test}_messagedata.v ./test_messagedata.v
ln -s vectors/${test}_errordata.v ./test_errordata.v

#verilog top.v encoder.v decoder.v symboldelay.v erasurelist.v delay.v expander.v
scaler.v syndrome.v euclid.v polyeval.v fourney.v monitor.v

verilog +define+GATE_LEVEL +define+MAXERASURES="$maxerasures" +delay_mode_unit top.v
gatelevel/encoder.vg gatelevel/decoder.vg +libext+.v -y ~/reedsolomon/libs/perf/veri

mv verilog.log vectors/${name}_verilog.log
mv encoder.raw vectors/${name}_encoder.raw
mv errors.raw vectors/${name}_errors.raw
mv erasures.raw vectors/${name}_erasures.raw
mv decoder.raw vectors/${name}_decoder.raw
```

```
echo "Running reference encoder"
cut -d"#" -f1 < vectors/${name}_encoder.raw > vectors/${name}_ourencoder.raw
./bin/rs -n${BLOCK} -r${R} vectors/${test}_messagedata.raw
vectors/${name}_refencoder.raw
./bin/compare vectors/${test}.log vectors/${name}_refencoder.raw
vectors/${name}_ourencoder.raw 1 ${BLOCK} ${T} ${maxerasures}

if ( $? == 0 ) then
  echo "Check successful, the encoder passed the test."
else
  echo "Check failed, the encoder failed the test."
endif

echo
echo "Running reference decoder"

cut -d"#" -f1 < vectors/${name}_decoder.raw > vectors/${name}_ourdecoder.raw

cut -d"#" -f1 < vectors/${name}_errors.raw > vectors/1
cut -d"#" -f1 < vectors/${name}_erasures.raw > vectors/2
paste vectors/1 vectors/2 | tr -d "\t" > vectors/${name}_corrupted.raw

./bin/rs -d -n${BLOCK} -r${R} -V${maxerasures} vectors/${name}_corrupted.raw
vectors/${name}_refdecoder.raw
./bin/compare vectors/${test}.log vectors/${name}_refdecoder.raw
vectors/${name}_ourdecoder.raw 0 ${BLOCK} ${T} ${maxerasures}

if ( $? == 0 ) then
  echo "Check successful, the decoder passed the test."
else
  echo "Check failed, the decoder failed the test."
endif

end
```

The only point of note is that this shell script also controls the value of maxerasures into both the reference decoder and our decoder. Generally all simulation is done with maxerasures set to 32. We do however re-run three of the tests with maxerasures set to 20.

2.1.8 File system organization

The following files and directory hierarchy is required for the simulation:

```
bin/rs
bin/compare
gatelevel/decoder.vg
gatelevel/encoder.vg
vectors
vectors/testa_errordata.v
vectors/testa.log
vectors/testa_messagedata.raw
vectors/testa_messagedata.v
vectors/testb_errordata.v
vectors/testb.log
vectors/testb_messagedata.raw
vectors/testb_messagedata.v
vectors/testc_errordata.v
vectors/testc.log
vectors/testc_messagedata.raw
vectors/testc_messagedata.v
vectors/testd_errordata.v
vectors/testd.log
vectors/testd_messagedata.raw
vectors/testd_messagedata.v
vectors/teste_errordata.v
vectors/teste.log
vectors/teste_messagedata.raw
vectors/teste_messagedata.v
vectors/testf_errordata.v
vectors/testf.log
vectors/testf_messagedata.raw
```

```
vectors/testf_messagedata.v
vectors/testjim_errordata.v
vectors/testjim.log
vectors/testjim_messagedata.raw
vectors/testjim_messagedata.v
top.v
params.v
```

2.2 Specific tests

The tests described in this section were first described in [3].

The overall process for vector generation is common to all of the tests. A single vector comprises three elements:

i. A randomly generated message (e.g. 128 random bytes).
ii. A randomly generated gap following the message.
iii. A randomly generated pattern of errors and erasures, whose characteristics differ between the different tests.

A further point worth noting is that an error (by definition) must have a non-zero magnitude, or it would not be an error. An erasure, however, may be flagged on a symbol that is actually correct. Thus, when generating an erasure we do not exclude this possibility.

In the below descriptions, B refers to the block size of the code, and T to it's error correcting capability. For example, an RS (160, 128, t=16) code would yield a value of 160 for B and 16 for T.

2.2.1 Test A - Random error and erasure combinations (32)

The purpose of this test is to validate the decoder operates correctly when successive vectors contain wildly different error characteristics.

We generate successive vectors ensuring the no errors case, the errors only case, the erasures only case and the errors and erasures case each occurring frequently. We also want include some uncorrectable codewords, to validate uncorrectable error pattern detection. Randomly selected valid codewords will be corrupted according to the following error distribution:

1/10 no errors
2/10 errors only
2/10 erasures only
5/10 errors and erasures

The weight (nerrors * 2 + nerasures) of the error pattern in each case will be chosen at random from the range 0 to 3T. Error patterns with a weight > 2T are generally uncorrectable. However, there is a small probability that some of these corrupted codewords will be "within the ball" of a different codeword, and thus will miscorrect. The proportion of these heavily corrupted codewords that miscorrect should correspond to the mathematical model (see table in Appendix A)

The following results were obtained for the run generated from `vectors -s10`:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 128000 symbols in, 160000 symbols out
processed 1000 blocks
checking encoders against each other
    dcheckcount=1000 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 690 OK, 310 failed
Chien searches: 717
128000 symbols out, 160000 symbols in, 7122 corrected
processed 1000 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=8
    refstatus=0 occurred 690 times
    refstatus=1 occurred 310 times
    miscorrect table:
        nerasures=26 occurred   11 times, miscorrected    1 times
        nerasures=30 occurred    6 times, miscorrected    3 times
        nerasures=32 occurred    8 times, miscorrected    4 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=8
    ourstatus=0 occurred 111 times
    ourstatus=1 occurred 137 times
    ourstatus=2 occurred 160 times
    ourstatus=3 occurred 282 times
    ourstatus=4 occurred 64 times
    ourstatus=5 occurred 245 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 1 times
    miscorrect table:
        nerasures=26 occurred   11 times, miscorrected    1 times
        nerasures=30 occurred    6 times, miscorrected    3 times
        nerasures=32 occurred    8 times, miscorrected    4 times
checking decoders against each other
    scheckcount=1000 sfailcount=0 dcheckcount=690 dfailcount=0
Check successful, the decoder passed the test.
```

Within this run, 8 miscorrects occurred, generally when the number of erasures was large. This correlates well with the expected probabilities.

2.2.2 Test A - Random errors and erasure combinations (20)

This is identical to the previous test, except that maxerasures has been reduced to 20. This causes some (previously) correctable error patterns to be declared uncorrectable, but has the advantage of reducing the probability of miscorrection.

For the run generated from vectors -s10, the following results were obtained:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 128000 symbols in, 160000 symbols out
processed 1000 blocks
checking encoders against each other
    dcheckcount=1000 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 613 OK, 387 failed
Chien searches: 616
128000 symbols out, 160000 symbols in, 5088 corrected
processed 1000 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    refstatus=0 occurred 613 times
    refstatus=1 occurred 387 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    ourstatus=0 occurred 111 times
    ourstatus=1 occurred 80 times
    ourstatus=2 occurred 160 times
    ourstatus=3 occurred 262 times
    ourstatus=4 occurred 64 times
    ourstatus=5 occurred 245 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 78 times
checking decoders against each other
    scheckcount=1000 sfailcount=0 dcheckcount=613 dfailcount=0
Check successful, the decoder passed the test.
```

It can be seen that reducing maxerasures from 32 to 20 has removed eliminated the miscorrected codewords, but at the expense of failing to correct some previously correctable error patterns. In particular, 77 codewords that were previously corrected are now declared as uncorrectable (these 77 are now declared as status code 7, implying a valid codeword was still output).

2.2.3 Test B - Realistic data (32)

The purpose of this test is to exercise the decoder with error patterns similar to those expected in the target application.

For each vector, the number of erasures is randomly selected from the range 0 to 2T * 5/8 - 1, and the number of errors is selected randomly from the range 0 to T * 3/8 - 1. For T=16, this corresponds to 0 to 20 and 0 to 6 respectively. All error patterns should be corrected.

The following results were obtained for the run generated from `vectors -s10`:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 128000 symbols in, 160000 symbols out
processed 1000 blocks
checking encoders against each other
    dcheckcount=1000 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 1000 OK, 0 failed
Chien searches: 991
128000 symbols out, 160000 symbols in, 13073 corrected
processed 1000 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    refstatus=0 occurred 1000 times
    refstatus=1 occurred 0 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    ourstatus=0 occurred 9 times
    ourstatus=1 occurred 131 times
    ourstatus=2 occurred 44 times
    ourstatus=3 occurred 816 times
    ourstatus=4 occurred 0 times
    ourstatus=5 occurred 0 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 0 times
checking decoders against each other
    scheckcount=1000 sfailcount=0 dcheckcount=1000 dfailcount=0
Check successful, the decoder passed the test.
```

Every codeword was declared correctable, and no data mis-matches between the decoders were observed.

2.2.4 Test C - All error and erasure combinations (32)

The purpose of this test is to exhaustively test every correctable combination of number of errors and number of erasures.

More specifically, we check all combinations where nerrors * 2 + nerasures = weight, where weight varies between 0 and 2T. For each weight value, nerrors can range from 0 to (weight/2). For the RS (160, 128, t=16) code, the number of combinations works out at 289. For each combination we generate `<base_num>` vectors. All error patterns should be corrected.

The following results were obtained for the run generated from `vectors -s10`:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
2890 blocks processed, 369920 symbols in, 462400 symbols out
processed 2890 blocks
checking encoders against each other
    dcheckcount=2890 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
2890 blocks processed, 2890 OK, 0 failed
Chien searches: 2880
369920 symbols out, 462400 symbols in, 46240 corrected
processed 2890 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    refstatus=0 occurred 2890 times
    refstatus=1 occurred 0 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    ourstatus=0 occurred 10 times
    ourstatus=1 occurred 320 times
    ourstatus=2 occurred 160 times
    ourstatus=3 occurred 2400 times
    ourstatus=4 occurred 0 times
    ourstatus=5 occurred 0 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 0 times
checking decoders against each other
    scheckcount=2890 sfailcount=0 dcheckcount=2890 dfailcount=0
Check successful, the decoder passed the test.
```

Every codeword was declared correctable, and no data mis-matches between the decoders were observed.

2.2.5 Test C - All error and erasure combinations (20)

This is identical to the previous test, except that maxerasures has been reduced to 20. This causes some (previously) correctable error patterns to be declared uncorrectable, but has the advantage of reducing the probability of miscorrection.

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
2890 blocks processed, 369920 symbols in, 462400 symbols out
processed 2890 blocks
checking encoders against each other
    dcheckcount=2890 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
2890 blocks processed, 2470 OK, 420 failed
Chien searches: 2460
369920 symbols out, 462400 symbols in, 35110 corrected
processed 2890 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    refstatus=0 occurred 2470 times
    refstatus=1 occurred 420 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    ourstatus=0 occurred 10 times
    ourstatus=1 occurred 200 times
    ourstatus=2 occurred 160 times
    ourstatus=3 occurred 2100 times
    ourstatus=4 occurred 0 times
    ourstatus=5 occurred 0 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 420 times
checking decoders against each other
    scheckcount=2890 sfailcount=0 dcheckcount=2470 dfailcount=0
Check successful, the decoder passed the test.
```

Compared to the previous run, 420 codewords are now declared as uncorrectable. This should be the number of codewords with between 21 and 32 erasures. This turns out to be the case.[3]

[3] 60 + 60 + 50 + 50 + 40 + 40 +30 + 30 + 20 + 20 + 10 + 10 = 420

2.2.6 Test D - Uncorrectable error detection (32)

The purpose of this test is to confirm that the observed probability of miscorrection, given a specific number of erasures, correlates with the mathematical model. The expected probabilities derived from the model are listed in Appendix A.

As it is only feasible to simulate a relatively small number of vectors, the results will be statistically significant only in the cases where the probability of miscorrection is large. This same test can be performed on the FPGA prototype with more than 10^5 times as many vectors in a given time. Thus, more statistically significant results can be obtained in this way.

A large number (B) of errors will be added, to essentially randomise the codeword. Following this, the number of erasures is varied from 0 to B. For nerasures ≤ 2T we run 10x as many vectors as for nerasures > 2T.

The following results were obtained for the run generated from `vectors -s10`:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
4580 blocks processed, 586240 symbols in, 732800 symbols out
processed 4580 blocks
checking encoders against each other
    dcheckcount=4580 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
4580 blocks processed, 166 OK, 4414 failed
Chien searches: 1704
586240 symbols out, 732800 symbols in, 5223 corrected
processed 4580 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=166
    refstatus=0 occurred 166 times
    refstatus=1 occurred 4414 times
    miscorrect table:
        nerasures=26 occurred  100 times, miscorrected    1 times
        nerasures=28 occurred  100 times, miscorrected   21 times
        nerasures=30 occurred  100 times, miscorrected   43 times
        nerasures=31 occurred  100 times, miscorrected    1 times
        nerasures=32 occurred  100 times, miscorrected  100 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=166
    ourstatus=0 occurred 0 times
    ourstatus=1 occurred 101 times
    ourstatus=2 occurred 0 times
    ourstatus=3 occurred 65 times
    ourstatus=4 occurred 100 times
    ourstatus=5 occurred 4300 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 14 times
    miscorrect table:
        nerasures=26 occurred  100 times, miscorrected    1 times
        nerasures=28 occurred  100 times, miscorrected   21 times
        nerasures=30 occurred  100 times, miscorrected   43 times
        nerasures=31 occurred  100 times, miscorrected    1 times
        nerasures=32 occurred  100 times, miscorrected  100 times
checking decoders against each other
    scheckcount=4580 sfailcount=0 dcheckcount=166 dfailcount=0
Check successful, the decoder passed the test.
```

Comparing these results to the probabilities in appendix A shows a good match.

2.2.7 Test D - Uncorrectable error detection (20)

This is identical to the previous test, except that maxerasures has been reduced to 20.

The following results were obtained for the run generated from `vectors -s10`:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
4580 blocks processed, 586240 symbols in, 732800 symbols out
processed 4580 blocks
checking encoders against each other
    dcheckcount=4580 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
4580 blocks processed, 0 OK, 4580 failed
Chien searches: 1102
586240 symbols out, 732800 symbols in, 0 corrected
processed 4580 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    refstatus=0 occurred 0 times
    refstatus=1 occurred 4580 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    ourstatus=0 occurred 0 times
    ourstatus=1 occurred 0 times
    ourstatus=2 occurred 0 times
    ourstatus=3 occurred 0 times
    ourstatus=4 occurred 100 times
    ourstatus=5 occurred 4300 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 180 times
checking decoders against each other
    scheckcount=4580 sfailcount=0 dcheckcount=0 dfailcount=0
Check successful, the decoder passed the test.
```

It can be seen all codewords are declared uncorrectable, and that no miscorrections are observed.

2.2.8 Test E - Sensitivity to erasure positions (32)

The purpose of this test is to validate the operation of the erasurelist block within the decoder.

We generate codewords where the erasures are clustered at the beginnings and ends of codewords. This is a particularly stressful case for the erasurelist block; the test in effect checks that erasures are always associated with the correct codeword. All error patterns should be corrected. More specifically:

3/10 of the time a codeword will have 0, 1 or 2 erasures
4/10 of the time a codeword will have between 3 and 2T-3 erasures
3/10 of the time a codeword will have 2T-2, 2T-1 or 2T erasures

The erasures be will distributed within the codeword as follows:

3/10 of time clustered at the start
3/10 of time clustered at the end
4/10 of time clustered at the start and end

The clustering algorithm will set an erasure in a location with a probability of 0.9, moving in from the start or end of the codeword until the required number of erasures has been marked.

The following results were obtained for the run generated from `vectors -s10`:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 128000 symbols in, 160000 symbols out
processed 1000 blocks
checking encoders against each other
    dcheckcount=1000 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 1000 OK, 0 failed
Chien searches: 896
128000 symbols out, 160000 symbols in, 15721 corrected
processed 1000 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    refstatus=0 occurred 1000 times
    refstatus=1 occurred 0 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=0
    ourstatus=0 occurred 104 times
    ourstatus=1 occurred 896 times
    ourstatus=2 occurred 0 times
    ourstatus=3 occurred 0 times
    ourstatus=4 occurred 0 times
    ourstatus=5 occurred 0 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 0 times
checking decoders against each other
    scheckcount=1000 sfailcount=0 dcheckcount=1000 dfailcount=0
Check successful, the decoder passed the test.
```

Every codeword was declared correctable, and no data mis-matches between the decoders were observed.

2.2.9 Test F - Sensitivity to gaps between codewords (32)

The purpose of this test is to verify that gaps between codewords are of no significance to the operation of the encoder or decoder.

This test replicates test A, but adds gaps between the codewords. The length of the gap is chosen at random from between 1 and 4B clock cycles from the following distribution:

4/10 between 1 and 2T
3/10 between 2T + 1 and B
1/10 between B + 1 and 2B
1/10 between 2B + 1 and 3B
1/10 between 3B + 1 and 4B

The following results were obtained for the run generated from `vectors -s10`:

```
Running reference encoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 128000 symbols in, 160000 symbols out
processed 1000 blocks
checking encoders against each other
    dcheckcount=1000 dfailcount=0
Check successful, the encoder passed the test.
Running reference decoder
Field params: m = 8 poly = 11d
RS code params: n = 160 r = 32 k = 128 L = 1 redund = 25.00% over k
1000 blocks processed, 727 OK, 273 failed
Chien searches: 743
128000 symbols out, 160000 symbols in, 7550 corrected
processed 1000 blocks
checking reference decoder against expected behaviour:
    #failures to correct=0, #miscorrects=6
    refstatus=0 occurred 727 times
    refstatus=1 occurred 273 times
    miscorrect table:
        nerasures=32 occurred   14 times, miscorrected    6 times
checking our decoder against expected behaviour:
    #failures to correct=0, #miscorrects=6
    ourstatus=0 occurred 103 times
    ourstatus=1 occurred 164 times
    ourstatus=2 occurred 177 times
    ourstatus=3 occurred 283 times
    ourstatus=4 occurred 55 times
    ourstatus=5 occurred 217 times
    ourstatus=6 occurred 0 times
    ourstatus=7 occurred 1 times
    miscorrect table:
        nerasures=32 occurred   14 times, miscorrected    6 times
checking decoders against each other
    scheckcount=1000 sfailcount=0 dcheckcount=727 dfailcount=0
Check successful, the decoder passed the test.
```

These results show the operation of the encoder and decoder is independent of gaps between codewords. The reason the results are not identical to test A is that drawing additional random numbers to determine the length of gap affects the precise error patterns.

3 FPGA prototype

3.1 Methodology

3.1.1 Overall prototype block diagram

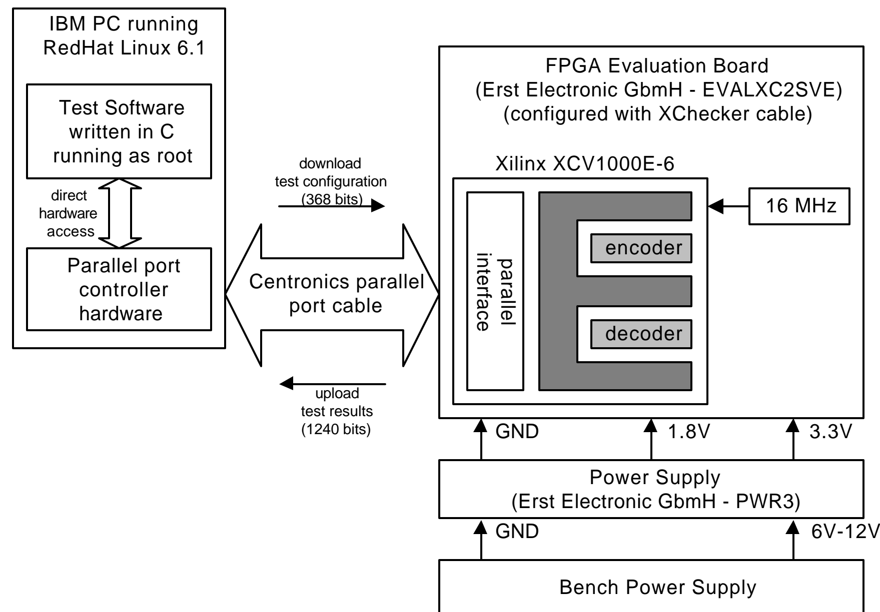

Figure 2 – Overall prototype block diagram

At the core of the prototype are the Reed-Solomon encoder and decoder, surrounded by some hardware for generating test messages, adding error and erasure patterns, and capturing results. All of this functionality is implemented using an off-the-shelf FPGA prototyping board containing a single Xilinx XCV1000E FPGA. Also implemented in this FPGA is a parallel port interface. This allows test configurations to be downloaded and test results to be uploaded. An individual test comprises up to 2^32 vectors, and with a 16 MHz clock 10^5 vectors can be run every second.

The hardware is controlled from a PC running Linux. The controlling test software is written in C, and is able to access the hardware registers of the parallel port controller directly. The advantage of this approach is that no kernel drivers are required; the disadvantage is that the test software must be run as root.

The default behaviour of the test software is to first validate the connection to the hardware (using some simple loopback modes implemented by the parallel port interface in the FPGA). Once this is done, a large number of individual tests are run sequentially. For each test, a configuration is downloaded which specifies the type and number of messages and the required distribution of errors and erasures. The test is started, and the hardware polled for until a done flag is seen. Once the test is complete, the results are uploaded and compared against the expected results.

3.1.2 Hardware design

3.1.2.1 FPGA block diagram

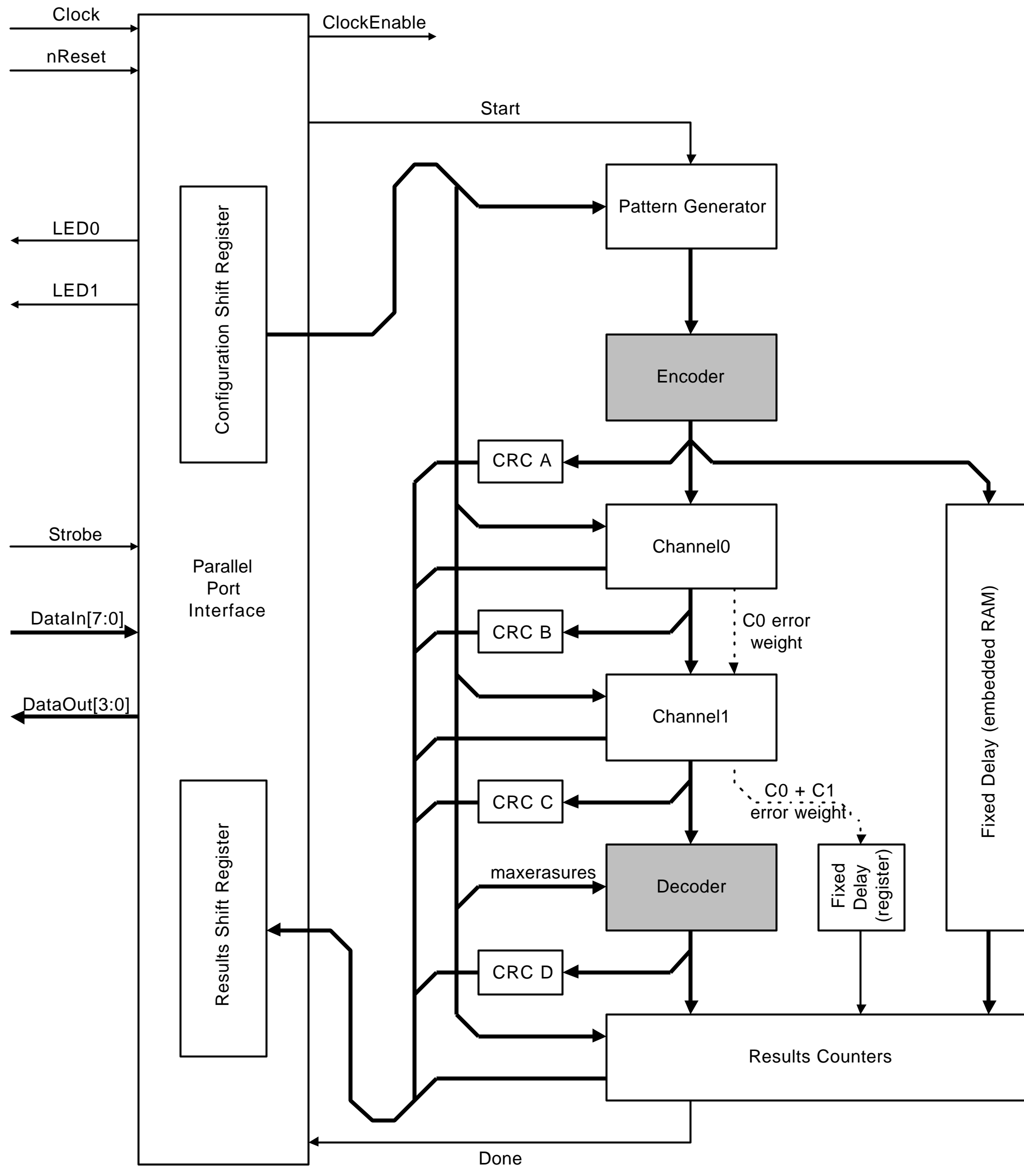

Figure 3 - FPGA block diagram

The FPGA contains the encoder and decoder, together with a number of additional blocks that are required to generate messages, add corruptions, and accumulate statistics on the results. These blocks will be described in detail in subsequent sections.

The general approach for validating the decoder is to check that output of the decoder matches the original output from the encoder. The purpose of the fixed delay block is to delay the encoded data, to allow such a comparison to be done. As an extra

precaution, we also compute signatures (CRCs) on the symbol data at four strategic points. As long as there are no unrecoverable errors, the value in CRCD at the end of a test run should match the value in CRCA.

The following global configuration registers exist:

name	*width*	*description*
r_seed	33	The seed for the random number generators. Within the FPGA there are several random number generators. The seed for each of them is derived from this value.
n	32	The number of codewords to run the test for.
erasureinX	1	The value to apply to the erasurein input of the decoder between codewords.
maxerasures	8	The value to apply to the maxerasures input of the decoder.

The following global results registers exist:

name	*width*	*description*
code_signatureA	32	The output of CRCA, which calculates a signature over the symbol data after the encoder.
code_signatureB	32	The output of CRCB, which calculates a signature over the symbol data after channel 0 has added corruptions.
code_signatureC	32	The output of CRCB, which calculates a signature over the symbol data after channel 1 has added corruptions.
code_signatureD	32	The output of CRCD, which calculates a signature over the symbol data after the decoder

3.1.2.2 Pattern generator

The purpose of the pattern generator block is to generate 128 byte messages followed by variable length gaps, to use as test data to be encoded.

Within the pattern generator, the following configuration registers exist:

name	*width*	*description*
pg_mode	1	The message data source (i.e. what each 8-bit symbol is) 0: random – bits 7 to 0 of a random number generator. 1: counter – bits 15 to 8 of the counter.
pg_cstart	16	The start value for the counter.
pg_climit	16	The limit value for the counter.
pg_cincrement	16	The increment value for the counter.
pg_i0fixed	12	The fixed part of interval i0.
pg_i0mask	12	The random interval mask for i0.
pg_i1fixed	12	The fixed part of interval i1.
pg_i1mask	12	The random interval mask for i1.
pg_iprob	8	The probability of selecting the interval between codewords from i0 rather than i1 (0 to 128).

The value for each 8-bit symbol of the message is taken either from bits 7 to 0 of a random number generator or from bits 15 to 8 of the counter, based on the `pg_mode` register.

At the beginning of a test run, the counter is initialised with `pg_cstart`.

With each symbol of each message, the counter is updated as follows:

```
if (counter + pg_cincrement < pg_climit)
      counter <= counter + pg_cincrement
else
      counter <= counter + pg_cincrement - (pg_climit - pg_cstart)
```

After a 128 byte message, 32 idle are always inserted to allow time for the encoder to append the 32 check symbols.

After this, a further gap is inserted whose length is selected using the following formulae:

```
if (prbs7() < pg_iprob)
      gap <= pg_i0fixed + (prbs12() & pg_i0mask)
else
      gap <= pg_i1fixed + (prbs12() & pg_i1mask)
```

where prbsN() returns a random number in the range 0 to 2^N-1, and

In the test software, some specific configurations of the pattern generator are used.

This configuration (called *random_nogaps*) results in a gap of zero:

```
pg_mode = 0
pg_i0fixed = 0
pg_i0mask = 0
pg_iprob = 128
```

This configuration (called *random_smallgaps*) results in a gap in the range 0 to 15:

```
pg.mode = 0
pg_i0fixed = 0
pg_i0mask = 15
pg_iprob = 128
```

This configuration (called *random_largegaps*) results in a gap in the range 0 to 511:

```
pg_mode = 0
pg_i0fixed = 0
pg_i0mask = 511
pg_iprob = 128
```

This configuration (called *random_variablegaps*) results in a small (0 to 15) gap 75% of the time, and a large (0 to 511) gap 25 % of the time:

```
pg_mode = 0
pg_i0fixed = 0
pg_i0mask = 511
pg_i1fixed = 0
pg_i1mask = 15
pg_iprob = 32
```

The pattern generator does not contain any results registers.

3.1.2.3 Channel model

The purpose of the channel model block is to corrupt the encoded data, by adding either errors or erasures according to a programmable distribution.

An instance of the channel model block can be configured to add either errors or erasures. The channel model block is replicated twice, thus allowing both errors and erasures to be added.

Within each instance of the channel model, the following configuration registers exist:

name	*width*	*description*
`mode`	2	0 – add errors, but add only if erasurein = 0. 1 – add errors regardless. 2 – add erasures, but add only if erasurein = 0. 3 – add erasures regardless.
`prob_enable`	8	The probability of the block being enabled for a given codeword (0 – 128).
`prob_symbol_fixed`	8	The fixed part of the prob_symbol distribution.
`prob_symbol_mask`	8	The random mask for the prob_symbol distribution.
`start_fixed`	8	The fixed part of the start distribution.
`start_mask`	8	The random mask for start distribution.
`step_prob`	8	The probability of selecting the step value from step0 rather than step1 (0 to 128).
`step0_fixed`	8	The fixed part of the step0 distribution.
`step0_mask`	8	The random mask for the step0 distribution.
`step1_fixed`	8	The fixed part of the step1 distribution.
`step1_mask`	8	The random mask for the step1 distribution.
`maximum`	8	The maximum number of corruptions in any one codeword.

Bit 1 of the `mode` register determines the type of corruption this channel model block will add. If bit 1 is 0 then errors will be added, otherwise erasures will be added.

Bit 0 of the `mode` register determines whether the block avoids erasures added by the previous channel model block. If bit 0 is 0 then corruptions will only be added in positions not already marked as erasures. If bit 0 is 1 then corruptions may be added anywhere.

The `prob_enable` register determines the probability that this block will be enabled for a given codeword. The probability value can range from 0 to 128, with 0 corresponding to "never enabled" and 128 corresponding to "always enabled":

```
/* Choose whether to corrupt this codeword */
if (rand7() < prob_enabled)
   enabled = 1
else
   enabled = 0
```

Corruptions are added using the following algorithm:

```
/* Initialization at the start of a codeword */
prob_symbol = prob_symbol_fixed + (rand() & prob_symbol_mask);
target = start_fixed + (rand() & start_mask);
count = 0;

/* Iterate through the symbols in the codeword */
for (i = 0; i < 160; i = i + 1) {
  /* Choose whether to corrupt this symbol */
  if ((count < maximum)                  &&
      (target == i)                      &&
      (RAND7 < prob_symbol)              &&
      ((mode & 1) || (erasure[i] == 0)) &&
      (enabled == 1)) {
    /* Choose corruption value, exclude 0 from errors */
    corruption = RAND8;
    if ((corruption == 0) && ((mode & 2) == 0))
      corruption = 255;
    /* Do the corruption, flagging erasure if required */
    symbol[i] = symbol[i] ^ corruption;
    if (mode & 2)
      erasure[i] = 1;
    /* Increment count of corruptions added */
    count = count + 1;
  }
  /* Select the next corruption target */
  if (target == i) {
    if (RAND7 < step_prob)
      target = target + step0_fixed + (rand() & step0_mask);
    else
      target = target + step1_fixed + (rand() & step1_mask);
  }
}
```

At the start of a codeword, `prob_symbol` is chosen from the distribution specified by `prob_symbol_fixed` and `prob_symbol_mask`. This represents the probability that a candidate location will be corrupted.

The `start_fixed`, `start_mask`, `step0_fixed`, `step0_mask`, `step1_fixed`, `step1_mask` and `step_prob` registers are then used to select a sequence of candidate locations within the codeword, for possible corruption.

At each candidate location, the probability of actually adding a corruption is determined by `prob_symbol`. In certain modes, locations that are already flagged as erasures will be avoided. This will skew the probability distribution slightly, which we do not correct for. An upper bound on the number of corruptions is provided by `maximum`.

In the test software, some specific configurations of the channel model are used:

This configuration (called *off*) prevents the channel model from adding corruptions:

```
mode = 0
prob_enable = 0
prob_symbol_fixed = 0
prob_symbol_mask = 0
step_prob = 0
start_fixed = 0
start_mask = 0
```

```
step0_fixed = 0
step0_mask = 0
step1_fixed = 0
step1_mask = 0
maximum = 0
```

This configuration (called *randomize*) causes every symbol to be corrupted as an error:

```
mode = 0
prob_enable = 128
prob_symbol_fixed = 128
prob_symbol_mask = 0
step_prob = 0
start_fixed = 0
start_mask = 0
step0_fixed = 0
step0_mask = 0
step1_fixed = 1
step1_mask = 0
maximum = 160
```

This configuration (called *random errorss*) causes on average *max / 2*, and at most *max*, errors to be added to a codeword. See Appendix B for a definition of the `calc_symbol_prob()` function.

```
mode = 0
prob_enable = 128
symbol_prob_fixed = calculated by calc_symbol_prob(max / 2)
symbol_prob_mask = calculated by calc_symbol_prob(max / 2
step_prob = 0
start_fixed = 0
start_mask = 0
step0_fixed = 0
step0_mask = 0
step1_fixed = 1
step1_mask = 0
maximum = max
```

This configuration (called *random erasures*) causes on average *max / 2*, and at most *max*, erasures to be added to a codeword. See Appendix B for a definition of the `calc_symbol_prob()` function.

```
mode = 3
prob_enable = 128
symbol_prob_fixed = calculated by calc_symbol_prob(max / 2)
symbol_prob_mask = calculated by calc_symbol_prob(max / 2)
step_prob = 0
start_fixed = 0
start_mask = 0
step0_fixed = 0
step0_mask = 0
step1_fixed = 1
step1_mask = 0
maximum = max
```

This configuration (called *fixed errors)* causes exactly *num* errors to be added. The distribution of these errors throughout the codeword is fairly random. See Appendix C for a definition of the `channel_init_fixed()` function.

```
mode = 0
prob_enable = 128
```

```
prob_symbol_fixed = 128
prob_symbol_mask = 0
step_prob = 0
start_fixed = 0
start_mask = calculated by channel_init_fixed()
step0_fixed = 0
step0_mask = 0
step1_fixed = 1
step1_mask = calculated by channel_init_fixed()
maximum = num
```

This configuration (called *fixed erasures)* causes exactly *num* erasures to be added. The distribution of these erasures throughout the codeword is fairly random. See Appendix C for a definition of the `channel_init_fixed()` function:

```
mode = 3
prob_enable = 128
prob_symbol_fixed = 128
prob_symbol_mask = 0
step_prob = 0
start_fixed = 0
start_mask = calculated by channel_init_fixed()
step0_fixed = 0
step0_mask = 0
step1_fixed = 1
step1_mask = calculated by channel_init_fixed()
maximum = num
```

This configuration (called *clumped erasures)* causes at most 32 erasures to be added, in such a way that they more likely to occur at the start or end of the codeword than in the middle.

```
mode = 3
prob_enable = 112
prob_symbol_fixed = 1
prob_symbol_mask = 127
step_prob = 4
start_fixed = 0
start_mask = 3
step0_fixed = 128
step0_mask = 7
step1_fixed = 1
step1_mask = 0
maximum = 32
```

The distributions obtained for *clumped erasures* are illustrated below:

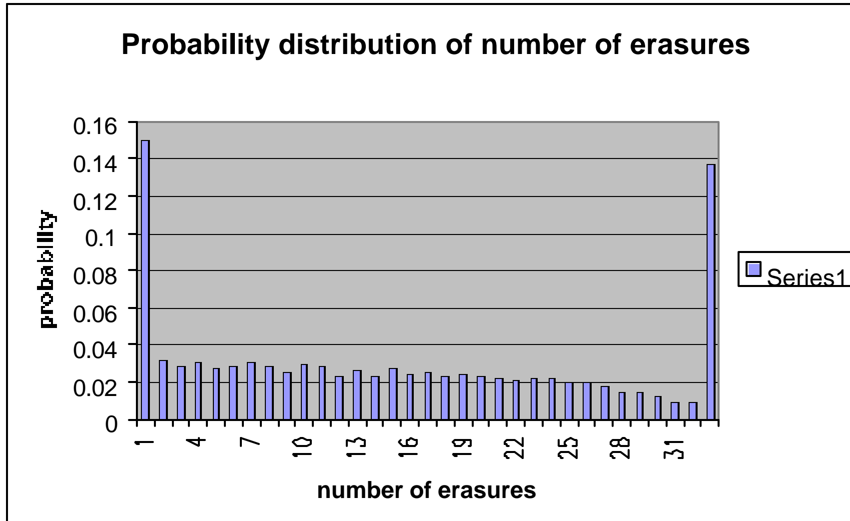

Figure 4 - Clumped erasures – number of erasures distribution

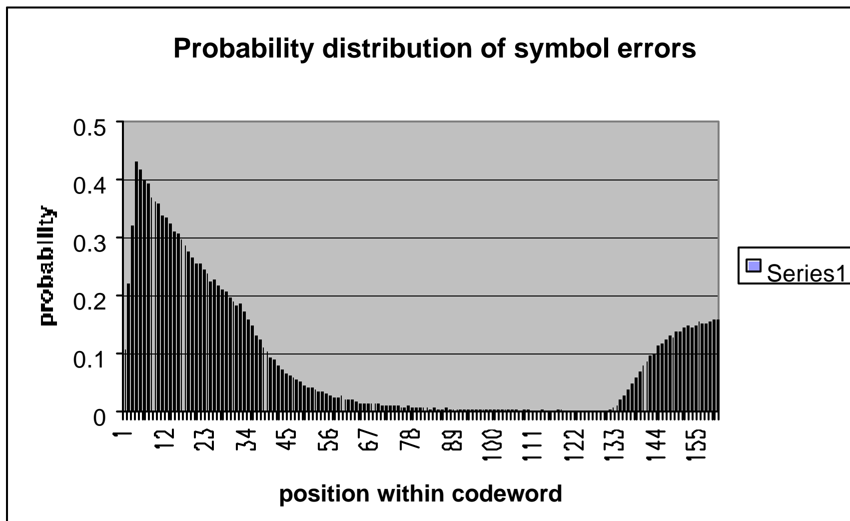

Figure 5 - Clumped erasures - symbol error distribution

Within each instance of the channel model, the following results registers exist:

name	*width*	*description*
`total_errors`	40	A cumulative total of the number of corruptions added.
`total_framingerrors`	40	A cumulative total of the number of framing errors detected.

The `total_errors` register counts the number of corruptions added by this block during the test run.

The `total_framingerrors` register counts the number of times the SOB (start of block), EOB (end of block) and ACTIVE control signals are incorrectly generated by the encoder.

3.1.2.4 Results counters

The purpose of the results counters block is to accumulate statistics from the decoder during a test run.

Within the results counters block, the following configuration register exists:

name	width	description
`matchn`	8	The value of *number of erasures* for which the matchn_status0123 and matchn_status4567 registers will accumulate results.

Within each instance of the results counters block, the following results registers exist:

name	width	description
`total_erasures`	40	A cumulative total of the number of erasure counted by the decoder.
`total_errors`	40	A cumulative total of the number of errors counted by the decoder.
`total_diffs`	32	The number of codewords where there were some differences between the symbols output from the encoder and the symbols output from the decoder, over all codewords.
`significant_diffs`	32	The number of codewords where there were some differences between the symbols output from the encoder and the symbols output from the decoder, for codewords where the error pattern had weight 2T or less, and which decoder claimed as correctable.
`framingerrors`	40	A cumulative total of the number of decoder framing errors detected.
`status0_diff0`	32	The number of codewords for which the decoder output a status of 0, and for which no symbol differences between the encoder and decoder were detected.
`status0_diffN0`	32	The number of codewords for which the decoder output a status of 0, and for which some symbol differences between the encoder and decoder were detected.
`status1_diff0`	32	The number of codewords for which the decoder output a status of 1, and for which no symbol differences between the encoder and decoder were detected.
`status1_diffN0`	32	The number of codewords for which the decoder

		output a status of 1, and for which some symbol differences between the encoder and decoder were detected.
status2_diff0	32	The number of codewords for which the decoder output a status of 2, and for which no symbol differences between the encoder and decoder were detected.
status2_diffN0	32	The number of codewords for which the decoder output a status of 3, and for which some symbol differences between the encoder and decoder were detected.
status3_diff0	32	The number of codewords for which the decoder output a status of 3, and for which no symbol differences between the encoder and decoder were detected.
status3_diffN0	32	The number of codewords for which the decoder output a status of 3, and for which some symbol differences between the encoder and decoder were detected.
status4_diff0	32	The number of codewords for which the decoder output a status of 4, and for which no symbol differences between the encoder and decoder were detected.
status4_diffN0	32	The number of codewords for which the decoder output a status of 4, and for which some symbol differences between the encoder and decoder were detected.
status5_diff0	32	The number of codewords for which the decoder output a status of 5, and for which no symbol differences between the encoder and decoder were detected.
status5_diffN0	32	The number of codewords for which the decoder output a status of 5, and for which some symbol differences between the encoder and decoder were detected.
status6_diff0	32	The number of codewords for which the decoder output a status of 6, and for which no symbol differences between the encoder and decoder were detected.
status6_diffN0	32	The number of codewords for which the decoder output a status of 6, and for which some symbol differences between the encoder and decoder were detected.
status7_diff0	32	The number of codewords for which the decoder output a status of 7, and for which no symbol differences between the encoder and decoder were detected.
status7_diffN0	32	The number of codewords for which the decoder output a status of 7, and for which some symbol

		differences between the encoder and decoder were detected.
overall_cc	32	The number of codewords with an error pattern of weight 2T or less, for which the decoder output a status of 0, 1, 2 or 3.
overall_cu	32	The number of codewords with an error pattern of weight 2T or less, for which the decoder output a status of 4, 5, 6 or 7. This can occur routinely if maxerasures is less than 2T.
overall_uc	32	The number of codewords with an error pattern of weight 2T + 1 or more, for which the decoder output a status of 0, 1, 2 or 3. This represents a miscorrection.
overall_uu	32	The number of codewords with an error pattern of weight 2T + 1 or more, for which the decoder output a status of 4, 5, 6 or 7.
matchn_status0123	32	The number of codewords for which the decoder output a status of 0, 1, 2 or 3, and for which the number of erasures matched `matchn`.
matchn_status4567	32	The number of codewords for which the decoder output a status of 0, 1, 2 or 3, and for which the number of erasures matched `matchn`.
first_diff	32	The codeword number where the `total_diffs` was first incremented.
last_diff	32	The codeword number where the `total_diffs` was last incremented.

3.1.2.5 Parallel Port Interface

The purpose of the parallel port interface block is to allow the configuration and results registers described in the preceding sections to be accessed using a standard PC equipped with a centronics parallel port. In addition, this interface is used to start a test running, and to poll for completion.

The parallel port interface implements a command based interface, where a command is sent by placing an 8-bit value on signals D0 to D7 and then taking STROBE low and back high again. The command value is latched shortly after the rising edge of STROBE. The following commands are implemented:

command opcode	*command name*	*description*	*value of DOUT when command complete*
00	STOP	Stop the current test run.	undefined
01	START	Start a new test run.	undefined
02	TRANSFER	Transfer test results to the results register.	undefined
03	SHIFT RESULTS	Shift the results register left 4 by bits.	Top 4 bits of the results register prior to shift
04	POLL	Poll for test completion.	0 if test still running 1 if test completed
05	LOOPBACK	Copy the configuration register directly to the results register.	undefined
10-1F	ECHO	Echo opcode[3:0] to DOUT.	opcode[3:0]

20-2F	SHIFT CONFIG	Shift the configuration register left 4 bits so that opcode[3:0] is loaded into the bottom 4 bits..	undefined
30-3F	LED	Load opcode[3:0] into the LED register.	undefined
40-4F	CEPROB	Defines the probability of ClockEnable being asserted on a given cycle. The probability is (opcode[3:0] + 1) / 16. For example, opcode 40 would set the probability to 1/16, opcode 4F would set it to 16/16.	undefined

Some commands return 4 bit results (DOUT) using the following parallel port status signals:

~DOUT[3] returned on signal BusyAck (inversion transparent to software).
DOUT[2] returned on signal Acknowledge.
DOUT[1] returned signal PaperOut.
DOUT[0] returned signal Select.

A test is configured by downloading a configuration into the configuration register (4 bits at a time) using the SHIFT CONFIG command. The test can then be started using the START command. Periodically, the completion status of the test can be polled using the POLL command. Once the test is complete, the TRANSFER command is used to load the results register with the test results. Finally the results of the test are uploaded (again, 4 bits at a time) using the SHIFT RESULTS command.

The individual configuration registers described in the preceding sections are mapped into the 368-bit configuration register in the following sequence:

name	*width*	*start position in configuration register*
pg_mode	1	367
c0_mode	2	365
c1_mode	2	363
erasureinX	1	362
spare	1	361
r_seed	33	328
n	32	296
pg_cstart	16	280
pg_climit	16	264
pg_cincrement	16	248
pg_i0fixed	12	236
pg_i0mask	12	224
pg_i1fixed	12	212
pg_i1mask	12	200
pg_iprob	8	192
c0_prob_enable	8	184
c0_prob_symbol_fixed	8	176
c0_prob_symbol_mask	8	168
c0_step_prob	8	160
c0_start_fixed	8	152
c0_start_mask	8	144
c0_step0_fixed	8	136

c0_step0_mask	8	128
c0_step1_fixed	8	120
c0_step1_mask	8	112
c0_maximum	8	104
c1_prob_enable	8	96
c1_prob_symbol_fixed	8	88
c1_prob_symbol_mask	8	80
c1_step_prob	8	72
c1_start_fixed	8	64
c1_start_mask	8	56
c1_step0_fixed	8	48
c1_step0_mask	8	40
c1_step1_fixed	8	32
c1_step1_mask	8	24
c1_maximum	8	16
matchn	8	8
maxerasures	8	0

The `SHIFT CONFIG` command shifts the configuration register left 4 bits and then replaces the LS 4 bits with opcode[3:0].

The first 4-bit chunk to be shifted in would set:

```
pg_mode      = opcode[3]
c0_mode[1]   = opcode[2]
c0_mode[0]   = opcode[1]
c1_mode[1]   = opcode[0]
```

The final 4-bit chunk to be shifted in would set:

```
maxerasure[3] = opcode[3]
maxerasure[2] = opcode[2]
maxerasure[1] = opcode[1]
maxerasure[0] = opcode[0]
```

The individual results registers described in the preceding sections are mapped into the 1240-bit results register in the following sequence:

name	*width*	*start position in results register*
code_signatureA	32	1208
code_signatureB	32	1176
code_signatureC	32	1144
code_signatureD	32	1112
c0_totalerrors	40	1072
c0_framingerrors	40	1032
c1_totalerrors	40	992
c1_framingerrors	40	952
total_erasures	40	912
total_errors	40	872
total_diffs	32	840
significant_diffs	32	808
framingerrors	40	768
status0_diff0	32	736
status0_diffN0	32	704
status1_diff0	32	672
status1_diffN0	32	640

status2_diff0	32	608
status2_diffN0	32	576
status3_diff0	32	544
status3_diffN0	32	512
status4_diff0	32	480
status4_diffN0	32	448
status5_diff0	32	416
status5_diffN0	32	384
status6_diff0	32	352
status6_diffN0	32	320
status7_diff0	32	288
status7_diffN0	32	256
overall_cc	32	224
overall_cu	32	192
overall_uc	32	160
overall_uu	32	128
matchn_status0123	32	96
matchn_status4567	32	64
first_diff	32	32
last_diff	32	0

The `SHIFT RESULTS` command shifts sets DOUT to the MS 4 bits of the results register, then shifts the results register left 4 bits

The first 4-bit chunk to be shifted out would have:

```
DOUT[3] = code_signatureA[31]
DOUT[2] = code_signatureA[30]
DOUT[1] = code_signatureA[29]
DOUT[0] = code_signatureA[28]
```

The final 4-bit chunk to be shifted out would have:

```
DOUT[3] = last_diff[3]
DOUT[2] = last_diff[2]
DOUT[1] = last_diff[1]
DOUT[0] = last_diff[0]
```

3.1.3 Hardware implementation

3.1.3.1 Prototyping board

The prototype is implemented using an off-the-shelf development board from ErSt Electronic GmbH (their website is http://www.erst.ch). The board used was the EVALXC2SVE-HQ240 containing a single Xilinx XCV1000E-HQ240-6 FPGA.

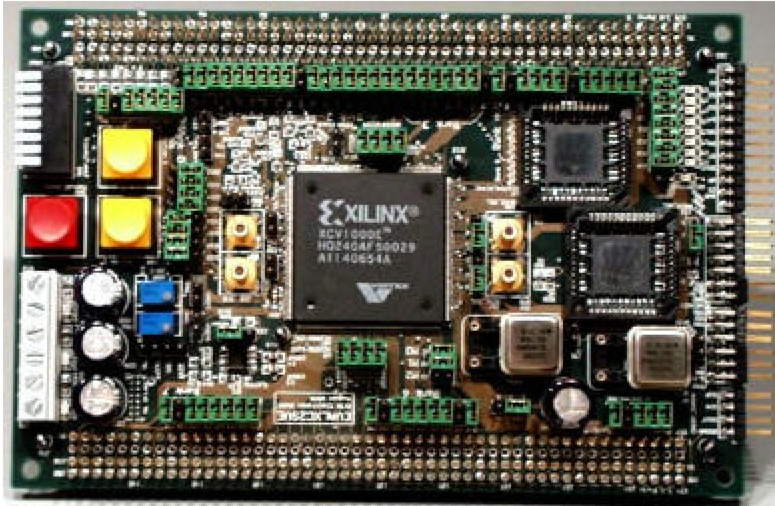

Figure 6 - The EVALXC2SVE-HQ240 prototyping poard

More information on the prototyping board can be found at:
http://www.erst.ch/english/evalxc2sve/evalxc2sve.html

The board is supplied with a power supply called the PWR3 power module, which generates the 1.8V and 3.3V supplies required by the Xilinx FPGA using switching regulators. The PWR3 power module requires a single DC supply of between 6V and 12V.

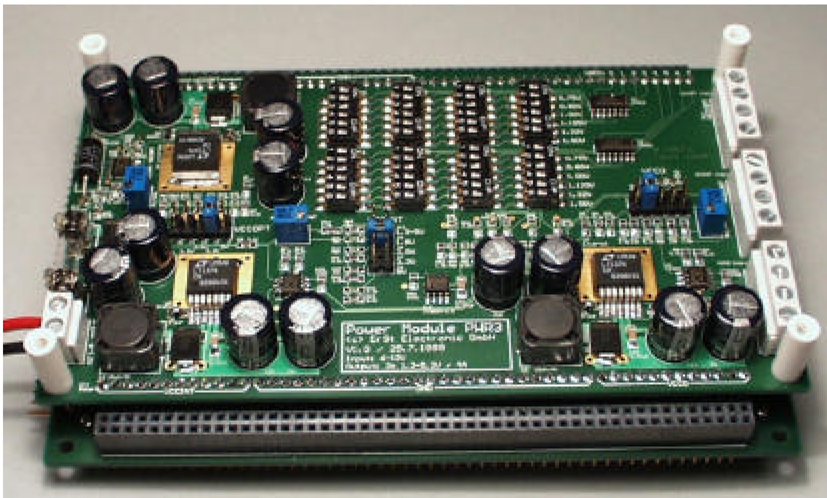

Figure 7 - The PWR3 power module

More information on the power module can be found at:
http://www.erst.ch/english/evalxcv/evalxcvhq240.html#pwr3

The prototyping board and power module together cost $2,500.

3.1.3.2 Jumper configuration

The prototype board is highly configurable using jumpers. As some of these jumpers select power supply voltages, incorrect settings may cause permanent damage. Check everything carefully before applying power!!

The jumper settings used for our design are listed below:

jumper ID	*jumper name*	*jumper position*
J1	M 0	off
J2	M 1	off
J3	M 2	off
J4	Connector for Xchecker cable	n/a
J5	Connector for JTAG cable	n/a
J6	Connector for SelectMap cable	n/a
J7	LED D9	on
J8	LED D8	on
J9-J14	LEDs D7-D2	off
J15-J18	Connectors for external clocks	n/a
J19-J22	Ground Points	n/a
J23	Reset	link 1-2
J24	VCCO selection	link 2-3
J25	Connector for daisy chain	n/a
J26	GCLK0_SEL	link 2-3
J27	GLCK1_SEL	link 2-3
J28	GCLK2_SEL	off
J29	GCLK3_SEL	off
J30	VCO0	off
J31	VCO1	off
J32	promsel	off
J33	connector for external power	n/a
J34	solder bridge	factory set
J35-J101	VREF selection	off
J102-J105	VREF selection	link 2-3
J106-J109	VREF selection	link 1-2
J110-J111	VREF Measurement points	n/a
J112-J113	Ground Points	n/a
J114-J121	VCCO selection	on
J122	Linear Burst	off
J123	Connector for SRAM JTAG	n/a
J124	FPGA Clock	off

3.1.3.3 Download cable

To download the FPGA PROM file to the FPGA we used a Xilinx Xchecker serial download cable, connected to a Windows PC running the Xilinx software. A download speed of 115 Kbps works reliably. The Xchecker cable is connected to J4 on the prototyping board as follows:

Xchecker signal	*colour*	*J4 pin num*	*J4 pin name*
VCC	red	6	3.3V
GND	black	7	GND
CCLK	yellow	4	CCLK
D/P	blue	2	DONE
DIN	green	5	DATA
PROG	orange	1	PROG
INIT	white	3	INIT
RST	purple	n/c	n/c

Note that the old Xchecker cables use a 5V VCC, but will generally run perfectly well from 3.3V.

3.1.3.4 Parallel cable

A custom cable needs to be constructed to connect the parallel port on the PC to the development board. The diagram below illustrates how this cable is made up.

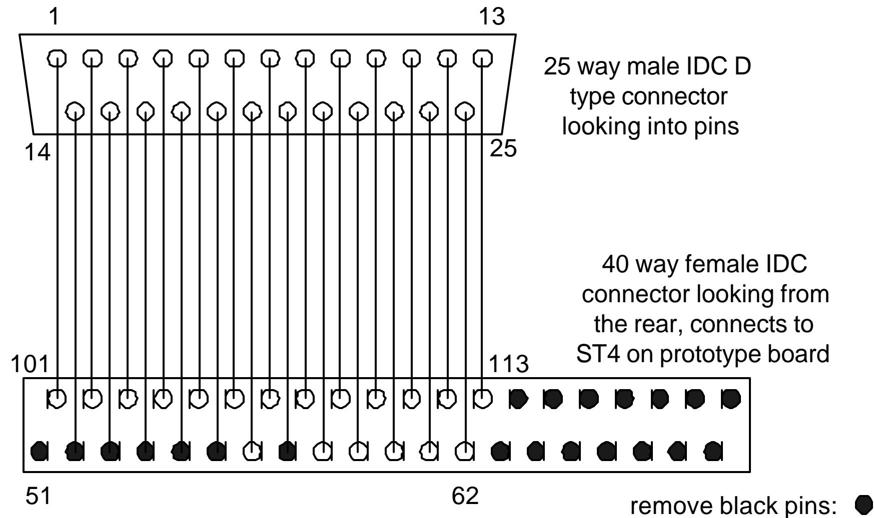

Figure 8 – Custom parallel cable construction

3.1.3.5 FPGA Usage

The development board contains a XCV1000E Xilinx FPGA. The utilization of this part (as reported by the Xilinx mapper) is shown below:

```
Target Device  : xv1000e
Target Package : hq240
Target Speed   : -6
Mapper Version : virtexe -- D.26
Mapped Date    : Wed Mar 28 14:48:08 2001

Design Summary
--------------
   Number of errors:      0
   Number of warnings:   84
   Number of Slices:            12,286 out of 12,288   99%
   Number of Slices containing
      unrelated logic:              38 out of 12,286    1%
   Number of Slice Flip Flops:   10,662 out of 24,576   43%
   Total Number 4 input LUTs:    19,076 out of 24,576   77%
      Number used as LUTs:                   18,843
      Number used as a route-thru:               73
      Number used as Shift registers:           160
   Number of bonded IOBs:           16 out of    158   10%
   Number of GCLKs:                  1 out of      4   25%
   Number of GCLKIOBs:               1 out of      4   25%
Total equivalent gate count for design:  230,312
Additional JTAG gate count for IOBs:  816
```

It can be seen that the current design occupies approx 99% of the available slices. This figure is slightly misleading, since that mapper does not start placing unrelated logic within the same slice until the whole device has been occupied. However, we

are still pretty close to the limit. It would not be feasible to prototype a longer code in within this device.

3.1.4 Test software

The test software is written in C, and is designed to run under Linux. It will not, in its current form, run under HP-UX or Windows. This is because it uses a Linux specific system call - `ioperm`() - to map the PC's parallel port interface register into user space, thus enabling direct access to the hardware without a driver. The name of test software executable is `tester`. The software is currently compiled for a parallel port at IO address 0x378, which is the default for LPT1.

3.1.4.1 Command line options

The command line option syntax for `tester` is:

```
tester  [-a <base num>] [-b <base num>] [-c <base num>]
        [-d <base num>] [-e <base num>] [-f <base num>]
        [-x <num>] [-y <num>] [-z <num>]
        [-s <scale factor>] [-r <random seed>] [-v <verbosity>]
        [-C <clock enabble probability>]
```

-a The base number of vectors to execute testa for. The actual number of vectors will be four times the base, since this test is run with four different pattern generator configurations (default 1,000,000).

-b The base number of vectors to execute testb for. The actual number of vectors will be 132 times the base, since this test is run for four different pattern generator configurations and 33 different error and erasure combinations (default 100,000).

-c The base number of vectors to execute testc for. The actual number of vectors will be 1156 times the base, since this test is run for four different pattern generator configurations and 289 different error and erasure combinations (default 10,000).

-d The base number of vectors to execute testd for. The actua l number of vectors will be 33 times the base, since this test is run for 33 different erasure combinations (default 1,000,000).

-e The base number of vectors to execute teste for. The actual number of vectors will be 8 times the base, since this test is run for four different pattern generator configurations and 2 different erasureinX combinations (default 1,000,000).

-f The base number of vectors to execute testf for. The actual number of vectors will be 33 times the base, since this test is run for 33 different erasure combinations (default 1,000,000).

Thus, the default number of vectors generated is:

4 * 1000000 + 132 * 100000 + 1156 * 10000
33 * 1000000 + 8 * 100000 + 33 * 1000000 = 101,760,000

A typical run of this length will take about 22 minutes.

-s A scale factor to apply to the number of vectors for each test (default 1).

Thus, if just –s 100 was given, a total of 10,176,000,000 vectors would be generated. A run of this length would take about 37 hours.

-r A random seed, so that (if required) different vector sets can be generated (default 21011967).

-v The verbosity (0 to 15) when generating vectors (default 1).

-x The number of iterations of the flashing LED test (default 10).

-y The number of iterations of the parallel port echo test (default 1000).

-z The number of iterations of the parallel port loopback test (default 1000).

-C The probability of asserting clock enable (0 – 15), with 0 equating to a probability of 1/16th, and 15 equating to a probability of 16/16th (i.e. 1). The results of a run should be identical, regardless of the clock enable probability, at the complete test system is stalled when clock enable is low.

3.1.4.2 Results checking

After each iteration of each test, the test software applies several checks to the results counters to check for correct operation of the encoder and decoder.

There are two checking modes: CHK_DIFFS and CHK_NORMAL.

The CHK_DIFFS mode is the most conservative, and should be used when all error patterns are expected to be correctable. This mode will only pass if every codeword is corrected back to the original.

The CHK_NORMAL mode is a sub-set of CHK_DIFFS, and should be used where some of the error patterns are expected to be uncorrectable.

In the CHK_NORMAL mode, the following checks are made:

name	*description*	*return code failure bit*
framing	The `c0_framingerrors`, `c1_framingerrors` and `framing_errors` counters should be zero. A failure implies the position of the SOB, EOB and ACTIVE signals from the encoder or decoder is incorrect.	0
status6	The `status6_diff0` and `status6_diffN0` counters should be zero. A failure implies the monitor block has trapped a case where the decoder claims to have corrected	1

	a codeword, yet the resultant codeword is invalid.	
missing codewords	The sum of the values of the sixteen `status` counters should equal n. The sum of the values of the four `overall` counters should equal n. A failure implies the decoder has silently discarded a codeword, or (somehow) generated extra ones.	2
significant diffs	The `significant_diffs` counter should be zero. A failure implies the input to the decoder was correctable, the decoder output a status of correctable, then failed to correct properly.	3
failure to correct	If `maxerasures` = 32, then the `overall_cu` counter should be zero A failure implies the input to the decoder was correctable, yet the decoder output a status of uncorrectable. If maxerasures is reduced below 32, this case may occur, hence we only perform this check if maxerasures is 32.	4
miscorrection consistency	The `overall_uc` counter should equal the sum of the four `status[0123]_diffN0` counters. A failure implies something other than a miscorrection caused the decoder to output a status of correctable then failed to correct properly.	5
correction consistency	The `overall_cc` counter should equal the sum of the four `status[0123]_diff0` counters. A failure implies one of two things: 1) The input to the decoder was correctable, the decoder output a status of correctable, yet failed to correct properly. This will also be recorded in the `significant_diffs` counter. 2) The input to the decoder was uncorrectable, yet somehow (magic?) the decoder managed to correct back to the original codeword.	6
C0 erasures	If channel 0 is set to add erasures, then `total_erasures` should equal `c0_total_errors`. A failure implies the decoder has incorrectly counted the number of erasures.	12
C1 erasures	If channel 1 is set to add erasures, then `total_erasures` should equal `c1_total_errors`. A failure implies the decoder has incorrectly counted the number of erasures.	13

In the CHK_DIFFS mode, the following **additional** checks are made:

name	*description*	*return code failure bit*
all correctable	The sum of the `overall_cc` and `overall_cu` counters should equal n. A failure implies the test included some uncorrectable error patterns, and the CHK_DIFFS mode should not be used. This is a failure of the test software,	7

	not the decoder.	
all corrected	The sum of the four `status[0123]_diff0` counters should be n. The `overall_cc` counter should equal n. A failure implies some codewords were not properly corrected.	8
some diffs	The `total_diffs` counter should be zero. A failure implies some codewords were not properly corrected.	9
incorrect signature	The `code_signatureD` register should equal the `code_signatureA` register. A failure implies some codewords were not properly corrected.	10
	undefined	11
C0 errors	If channel 0 is set to add errors, then `total_errors` should equal `c0_total_errors`. A failure implies the decoder has incorrectly counted the number of errors.	14
C1 errors	If channel 1 is set to add errors, then `total_errors` should equal `c1_total_errors`. A failure implies the decoder has incorrectly counted the number of errors.	15

3.1.4.3 Expected miscorrection probability

With Reed-Solomon codes, if the weight of the error pattern exceeds the error correction capability of the code, there is a finite probability that the corrupted codeword will lie *within the ball* of a different codeword. In this case, the decoder will correct to the different codeword, and a miscorrection is said to have occurred.

Miscorrection can happen with any Reed-Solomon decoder, regardless of how it is implemented. Without prior knowledge of the error pattern, it is not detectable by the decoder. Hence, it is a serious failure mode.

The probability of miscorrection depends heavily on the number of erasures; as the number of erasures approaches 2T, the probability of miscorrection approaches unity. Appendix A lists the probability of miscorrection for a RS (160,120,T=16) code, as the number of erasures varies from 0 to 32.

It is possible to trade off correction capability for detection capability, by placing an explicit limit of the number of erasures that may occur. If this limit is exceeded, the codeword is declared uncorrectable. Some otherwise correctable codewords will now be declared as uncorrectable. However, the probability of miscorrection is reduced, since this decreases with number of erasures.

If a particular test contains just uncorrectable error patterns with a fixed number of erasures, the expected probability of miscorrection can be obtained directly from the table in Appendix A.

If, however, the test contains either a mixture of correctable and uncorrectable error patterns, or a variable number of erasures, then the probabilities listed in Appendix A must be weighted, according to the distributions of error and erasures.

Determining the precise distributions of error and erasure for a given channel configuration is hard to do analytically. Instead, the test software runs a short simulation of the channel, using the `calibrate_miscorrects()` function.

The simulation models the errors and erasures added by channel 0 and channel 1 over a run of 1,000,000 codewords. Counts are maintained (by number of erasures) for the number of correctable and uncorrectable error patterns. These are scaled using the probabilities listed in Appendix A, to give the expected probability of miscorrection for this specific configuration as a whole. A table is generated, showing the how the expected probabilities reduce as maxerasures is reduced.

This approach is used to derive the expected miscorrection probabilities for tests A and F.

3.2 Specific tests

3.2.1 Test A – Random error and erasure combinations

The purpose of this test is to validate the decoder operates correctly when successive vectors contain wildly different error characteristics.

The test iterates through the four gap configurations: *nogaps, smallgaps, largegaps* and *variablegaps*. This checks for any sensitivity to gaps between codewords. Maxerasures is set to 32.

The pattern generator is configured to generate randomly selected valid codewords.

Channel 0 is configured such that 50% of the codewords have an average of 18 and a maximum of 36 erasures added (the *random erasures* configuration).

Channel 1 is configured such that 50% of the codewords have an average of 9 and a maximum of 18 errors added (the *random errors* configuration).

The test includes some uncorrectable error patterns, and so the checking mode is set to CHK_NORMAL.

The test was run using `tester` –s100 –v15, which generates a total of:

4 * 1,000,000 * 100 = 400,000,000 vectors.

All checks defined in CHK_NORMAL mode passed.

The `calibrate_miscorrects()` function generated the following table:

maxerasures	*prob(cu)*	*prob(uc)*
32	0.000000e+00	3.677533e-03
31	1.540000e-03	2.037533e-03
30	3.880000e-03	2.029369e-03
29	6.720000e-03	6.888828e-04
28	1.052000e-02	6.819536e-04
27	1.600000e-02	1.465440e-04
26	2.215000e-02	1.438288e-04
25	2.976000e-02	2.278844e-05
24	3.869000e-02	2.220337e-05
23	4.969000e-02	2.669433e-06
22	6.209000e-02	2.586809e-06
21	7.581000e-02	2.288010e-07
20	9.090000e-02	2.195126e-07
19	1.085300e-01	1.599571e-08
18	1.264200e-01	1.517070e-08
17	1.447200e-01	9.695390e-10
16	1.640000e-01	9.128643e-10
15	1.841300e-01	5.484194e-11
14	2.047500e-01	5.150709e-11
13	2.258600e-01	2.372961e-12
12	2.461500e-01	2.187200e-12

11	2.670300e-01	1.048965e-13
10	2.869200e-01	9.498990e-14
9	3.078800e-01	3.282340e-15
8	3.275200e-01	2.928969e-15
7	3.459400e-01	6.267958e-17
6	3.602500e-01	5.239942e-17
5	3.715700e-01	9.351424e-19
4	3.787400e-01	7.784026e-19
3	3.830300e-01	8.250100e-20
2	3.849100e-01	8.051137e-20
1	3.855400e-01	7.007313e-20
0	3.856700e-01	7.007313e-20

The observed probability of miscorrections ranged from 0.00354 to 0.00357. This correlated well with the theoretical probability of 0.003678, as maxerasures is set to 32.

3.2.2 Test B – Realistic data

The purpose of this test is to exercise the decoder with error patterns similar to those expected in the target application.

The test iterates through *nerasures* values from 0 to 32. For each value of *nerasures*, *nerrors* is calculated so that *nerrors* + 2 * *nerasures* = 32. This ensures the error pattern is always correctable. For each *nerrors-nerasures* combination, the test iterates through the four gap configurations: *nogaps, smallgaps, largegaps* and *variablegaps.* This checks for any sensitivity to gaps between codewords.

The pattern generator is configured to generate randomly selected valid codewords.

Channel 0 is configured such that 50% of the codewords have an average of *nerasures* / 2 and a maximum of *nerasures* erasures added (*the random erasures* configuration).

Channel 1 is configured such that 50% of the codewords have an average of *nerrors* / 2 and a maximum of *nerrors* errors added (*the random errors* configuration).

The test includes no uncorrectable error patterns, and so the checking mode is set to CHK_DIFFS.

The test was run with `tester` –s100 –v15, which generates a total of:
4 * 33 * 100,000 * 100 = 1,320,000,000 vectors.

All checks defined in CHK_NORMAL and CHK_DIFFS modes passed.

There were no miscorrections, since all error patterns were correctable.

3.2.3 Test C – All error and erasure combinations

The purpose of this test is to exhaustively test every correctable combination of number of errors and number of erasures.

In the previous test, the number of errors and erasures in a particular codeword was chosen randomly from a specific distribution. In this test, the number of errors and erasures are fixed for a particular iteration of the test.

The test iterates through all combinations of *nerrors* and *nerasures* where *nerrors* * 2 + *nerasures* = *weight*, where weight is between 0 and 32. For each *weight* value, *nerrors* can range from 0 to (*weight* / 2). There are 289 such *nerrors-nerasures* combinations. For each combination, the test iterates through the four gap configurations: *nogaps, smallgaps, largegaps* and *variablegaps.* This checks for any sensitivity to gaps between codewords. Maxerasures is set to 32.

The pattern generator is configured to generate randomly selected valid codewords.

Channel 0 is configured such that all of the codewords have an exactly *nerasures* erasures added (*the fixed erasures* configuration).

Channel 1 is configured such that all of the codewords have an exactly *nerrors* errors added (*the fixed errors* configuration).

The test includes no uncorrectable error patterns, and so the checking mode is set to CHK_DIFFS.

The test was run with `tester` –s100 –v15, which generates a total of:

4 * 289 * 10,000 * 100 = 1,156,000,000 vectors.

All checks defined in CHK_NORMAL and CHK_DIFFS modes passed.

There were no miscorrections, since all error patterns were correctable.

3.2.4 Test D – Uncorrectable error detection

The purpose of this test is to confirm that the observed probability of miscorrection, given a specific number of erasures, correlates with the mathematical model. The expected probabilities derived from the model are listed in appendix A.

The test iterates through *nerasures* values from 0 to 32. Maxerasures is set to 32.

The pattern generator is configured to generate randomly selected valid codewords with no gaps between codewords (the *random nogaps* configuration).

Channel 0 is configured such that all of the codewords have 160 errors added, effectively replacing the codeword with random data (the *randomize* configuration).

Channel 1 is configured such that all of the codewords have a *nerasures* erasures added (the *fixed erasures* configuration).

The test includes just uncorrectable error patterns, and so the checking mode is set to CHK_NORMAL.

The test was run using `tester` –s100 –v15, which generates a total of:

33 * 1,000,000 * 100 = 3,300,000,000 vectors.

All checks defined in CHK_NORMAL mode passed.

The observed probability of miscorrection is calculated by dividing the `overall_uc` counter by the N, the number of vectors per iteration.

The results are shown below:

number of erasures	*expected number of miscorrects*	*observed number of miscorrects*	*expected probability of miscorrects*	*observed probability of miscorrects*	*Z*
0	0	0	1.121170e-17	0.000000e+00	-0.00
1	0	0	1.125540e-16	0.000000e+00	-0.00
2	0	0	2.609560e-16	0.000000e+00	-0.00
3	0	0	2.487040e-17	0.000000e+00	-0.00
4	0	0	5.799180e-15	0.000000e+00	-0.00
5	0	0	5.224660e-16	0.000000e+00	-0.00
6	0	0	1.225340e-13	0.000000e+00	-0.00
7	0	0	1.038400e-14	0.000000e+00	-0.00
8	0	0	2.449820e-12	0.000000e+00	-0.00
9	0	0	1.941600e-13	0.000000e+00	-0.00
10	0	0	4.608420e-11	0.000000e+00	-0.02
11	0	0	3.392670e-12	0.000000e+00	-0.01
12	0	0	8.102350e-10	0.000000e+00	-0.09
13	0	0	5.495870e-11	0.000000e+00	-0.02
14	0	0	1.320810e-08	0.000000e+00	-0.36
15	0	0	8.173650e-10	0.000000e+00	-0.09
16	2	3	1.977010e-07	3.000000e-07	0.73
17	0	0	1.102620e-08	0.000000e+00	-0.33
18	27	26	2.684530e-06	2.600000e-06	-0.16
19	1	0	1.328510e-07	0.000000e+00	-1.15
20	326	302	3.256270e-05	3.020000e-05	-1.31
21	14	14	1.400970e-06	1.400000e-06	-0.00
22	3457	3342	3.457490e-04	3.342000e-04	-1.96
23	126	130	1.257590e-05	1.300000e-05	0.38
24	31254	31583	3.125430e-03	3.158300e-03	1.86
25	923	920	9.228220e-05	9.200000e-05	-0.09
26	230993	230558	2.309930e-02	2.305580e-02	-0.92
27	5192	5196	5.191610e-04	5.196000e-04	0.06
28	1309070	1306989	1.309070e-01	1.306989e-01	-1.95
29	19912	20015	1.991150e-03	2.001500e-03	0.73
30	5058440	5058529	5.058440e-01	5.058529e-01	0.06
31	39062	39272	3.906250e-03	3.927200e-03	1.06
32	10000000	10000000	1.000000e+00	1.000000e+00	0.00

***** data is actually from –s10, replace with –s100 *****

THE Z COLUMN IS CALCULATED USING THE EXPECTED AND OBSERVED PROBABILITIES WITH FOLLOWING STATISTICAL TEST:

$$Z = \frac{\text{OBSERVED} - \text{EXPECTED}}{\sqrt{\text{expected} * (1 - \text{expected}) / n}}$$

The observed probability of miscorrections is within 95% confidence bounds given by $-1.96 \leq Z \leq +1.96$.

For further details on this statistical test, see [4].

3.2.5 Test E – Sensitivity to erasure positions

The purpose of this test is to validate the operation of the erasurelist block within the decoder.

The test iterates through the four gap configurations: *nogaps, smallgaps, largegaps* and *variablegaps.* This checks for any sensitivity to gaps between codewords. For each gap configuration, the test iterates through erasureinX = 0 and erasureinX = 1. Maxerasures is set to 32.

Channel 0 is disabled (the *off* configuration).

CHANNEL 1 IS CONFIGURED TO ADD AT MOST 32 ERASURES, WITH MOST OF THE ERASURES ARE THE START AND END OF THE CODEWORDS (THE *CLUMPED ERASURES* CONFIGURATION).

The test includes no uncorrectable error patterns, and so the checking mode is set to CHK_DIFFS.

The test was run with `tester` –s100 –v15, which generates a total of:
4 * 2 * 1,000,000 * 100 = 800,000,000 vectors.

All checks defined in CHK_NORMAL and CHK_DIFFS modes passed.

There were no miscorrections, since all error patterns were correctable.

3.2.6 Test F – Random error and erasure combinations (limited maxerasures)

The purpose of this test is to validate that reducing maxerasures reduces the probability of miscorrection, at the cost of failing to correct some previously correctable error patterns

The test iterates through *maxerasures* values from 0 to 32.

The pattern generator is configured to generate randomly selected valid codewords with no gaps between codewords (the *random nogaps* configuration).

Channel 0 is configured such that 50% of the codewords have an average of 18 and a maximum of 36 erasures added (the *random erasures* configuration).

Channel 1 is configured such that 50% of the codewords have an average of 9 and a maximum of 18 errors added (the *random errors* configuration).

The test includes some uncorrectable error patterns, and so the checking mode is set to CHK_NORMAL.

The test was run using `tester` –s100 –v15, which generates a total of:

33 * 1,000,000 * 100 = 3,300,000,000 vectors.

All checks defined in CHK_NORMAL mode passed.

The expected an observed probabilities of failed correction and miscorrection are shown in the below table.

maxerasures	*expected probability of failed correction*	*observed probability of failed correction*	*expected probability of miscorrection*	*observed probability of miscorrection*
32	0.000000e+00	0.000000e+00	3.666729e-03	3.561400e-03
31	1.570000e-03	1.508000e-03	2.016729e-03	2.059400e-03
30	3.920000e-03	3.547000e-03	2.008761e-03	2.065300e-03
29	6.770000e-03	6.304900e-03	6.834492e-04	6.963000e-04
28	1.054000e-02	9.827200e-03	6.764404e-04	6.861000e-04
27	1.607000e-02	1.448780e-02	1.462670e-04	1.473000e-04
26	2.227000e-02	2.022720e-02	1.435310e-04	1.535000e-04
25	2.990000e-02	2.745770e-02	2.272171e-05	2.280000e-05
24	3.884000e-02	3.616400e-02	2.213295e-05	2.490000e-05
23	4.975000e-02	4.671360e-02	2.661517e-06	2.500000e-06
22	6.221000e-02	5.855750e-02	2.579899e-06	2.600000e-06
21	7.594000e-02	7.245200e-02	2.288060e-07	4.000000e-07
20	9.102000e-02	8.735220e-02	2.195035e-07	2.000000e-07
19	1.084400e-01	1.040750e-01	1.598666e-08	0.000000e+00
18	1.262400e-01	1.214285e-01	1.516298e-08	0.000000e+00
17	1.445600e-01	1.402720e-01	9.618153e-10	0.000000e+00
16	1.638600e-01	1.594642e-01	9.055817e-10	0.000000e+00
15	1.839300e-01	1.796524e-01	5.546738e-11	0.000000e+00
14	2.046700e-01	2.000611e-01	5.215705e-11	0.000000e+00
13	2.257800e-01	2.214465e-01	2.362512e-12	0.000000e+00
12	2.460900e-01	2.424941e-01	2.179500e-12	0.000000e+00
11	2.670700e-01	2.640226e-01	1.052982e-13	0.000000e+00
10	2.869400e-01	2.856715e-01	9.542553e-14	0.000000e+00
9	3.080200e-01	3.074614e-01	3.257125e-15	0.000000e+00
8	3.275900e-01	3.277059e-01	2.905696e-15	0.000000e+00
7	3.461200e-01	3.461859e-01	6.390467e-17	0.000000e+00
6	3.605100e-01	3.614618e-01	5.362451e-17	0.000000e+00
5	3.717800e-01	3.728243e-01	9.348937e-19	0.000000e+00
4	3.789000e-01	3.801331e-01	7.781539e-19	0.000000e+00
3	3.831700e-01	3.844080e-01	8.225229e-20	0.000000e+00
2	3.850200e-01	3.861152e-01	8.051137e-20	0.000000e+00
1	3.856500e-01	3.863464e-01	7.007313e-20	0.000000e+00
0	3.858000e-01	3.866950e-01	7.007313e-20	0.000000e+00

***** data is actually from –s10, replace with –s100 *****

Drawn graphically, this data shows an excellent match between observed and expected probabilities:

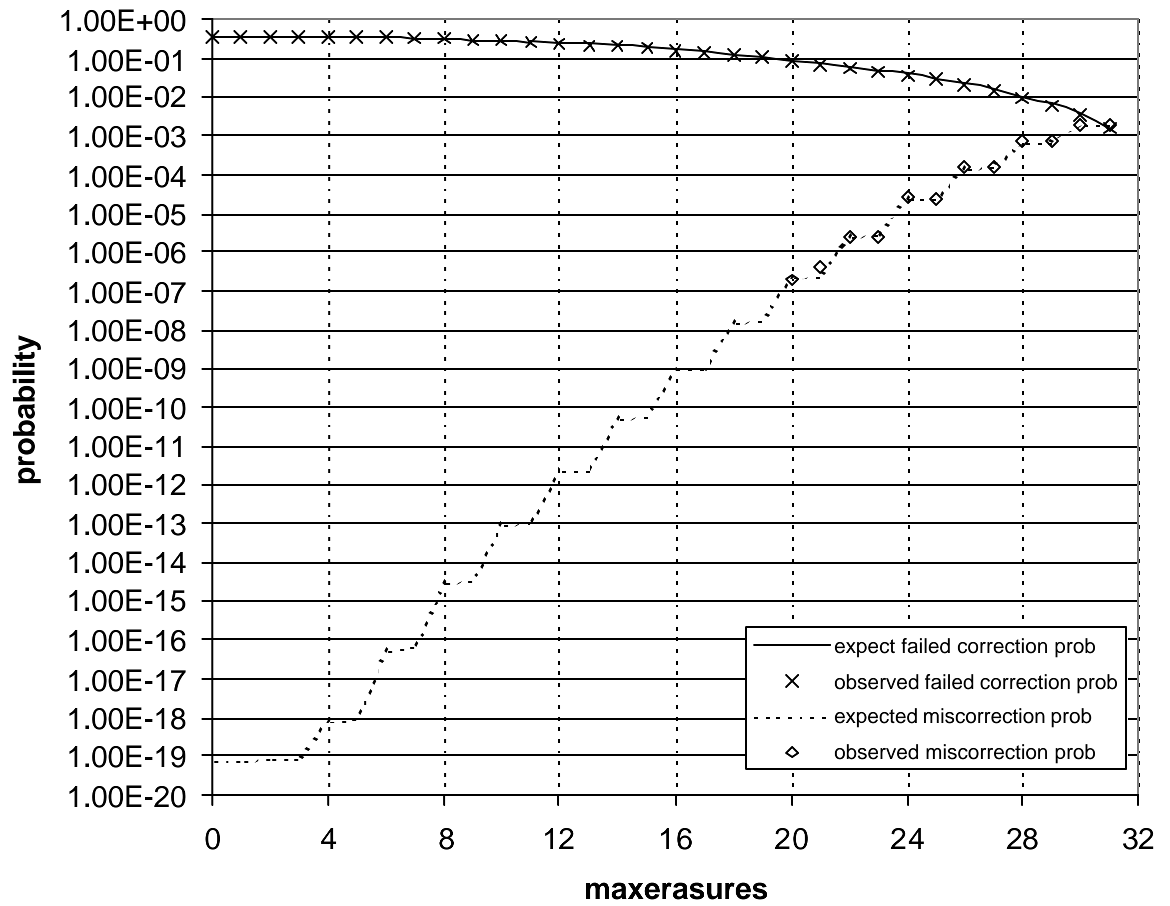

Figure 9 – Graph of expected verses observed miscorrection probabilities

4 Verification of other Reed-Solomon codes

The verilog source code for the encoder and decoder is fully parameterised, allowing encoders and decoders for different Reed-Solomon codes to be generated. This section outlines the changes that would be needed to verify such encoders and decoder.

The main code we have verified to date is the RS (160, 128, T=16) code in GF (2^8).

4.1 Codes in GF (2^8)

4.1.1 Simulation

The `params.v` file needs to be updated with the appropriate parameters for the Reed-Solomon code of interest.

A new gate-level netlist needs to be generated for encoder and decoder, using Synopsys.

Simulation vectors can be generated for a different Reed-Solomon code by setting appropriate values of B and T (using the –B and –T options to `vectors`).

This, for example, would allow codes like:

RS (152,128,T=12) in GF (2^8)
RS (144,128,T=8) in GF (2^8)
RS (136,128,T=4) in GF (2^8)

to be verified.

Simulation of both longer and shorter codes should be possible.

4.1.2 Prototype

It would be feasible to re-synthesize the prototype with a different encoder and decoder. All of the verilog that specifies the pattern generator, channel, results accumulator etc is parameterised, and so should work with different codes.

A new xsymboldelay.v file will need to be generated, using the `genshiftreg` C program. This needs to match the latency of the new decoder. In addition, the delay associated with xdelay.v may need to be changed.

The test software currently has several hard-coded constants, and changes would be needed. Ideally, this will be tidied up, allowing options like –B –T and –W to be given.

4.2 Codes in fields other than GF (2^8)

4.2.1 Simulation

As well as supporting different length codes in GF(2^8), the simulation supports codes of different field widths. This is because the .raw data file format uses white-space-separated decimal numbers to represent symbols throughout.

Simulation vectors can be generated for a different field width by setting appropriate values of W (using the –W and –T options to `vectors`).

We have (briefly) tested this on the following codes:

RS (484, 456, T=28) in GF(2^9)
RS (242, 228, T=14) in GF(2^9)
RS (120, 112, T=7) in GF(2^9)

[these were of interest to the Orca program in PSB]

4.2.2 Prototype

There should be no problems if the code field width is decreased.

The following problems will need to be addressed if the code field width is increased:

i. Some of the constants in `testparams.v` may need increasing. More specifically GEN_CWIDTH should be 2 * WIDTH, and GEN_IWIDTH should be WIDTH + 4 and WWIDTH should be wide enough to hold 3 * B.
ii. The current design uses 99% of the slices in a XCV1000E Xilinx FPGA. If the code width were increased, the design would be unlikely to fit. It may be possible to find a development board with a larger part. Alternatively, it is possible to partition the design between two FPGAs. This has been tested between two Xilinx XCV800 FPGAs using a different development board.

The test software currently has several hard-coded constants, and changes would be needed. Ideally, this will be tidied up, allowing options like –B –T and –W to be given.

5 References

[1] Design of a Synthesisable Reed-Solomon ECC Core, HPL Technical Report HPL-2001-124, David Banks, May 2001.
[2] A Hypersystolic Reed-Solomon Decoder, Elwyn Berlekamp, Gadiel Seroissi and Po Tong, published as chapter 10 of Reed-Solomon Codes and their Applications, IEEE Press, 1994.
[3] Proposed Reed-Solomon ECC Verification Plan, David Banks, 7th Feb 2001, (now superseded by this document).
[4] NIST Engineering Statistics Handbook, section 7.2.4: http://www.nist.gov/itl/div898/handbook/prc/section2/prc24.htm

6 Appendix A – Miscorrect probabilities

The below table shows the theoretical probabilities of miscorrecting a heavily corrupted codeword, as the number of flagged erasures increases.

number of erasures	*probability of miscorrect*
0	1.12117e-17
1	1.12554e-16
2	2.60956e-16
3	2.48704e-17
4	5.79918e-15
5	5.22466e-16
6	1.22534e-13
7	1.03840e-14
8	2.44982e-12
9	1.94160e-13
10	4.60842e-11
11	3.39267e-12
12	8.10235e-10
13	5.49587e-11
14	1.32081e-8
15	8.17365e-10
16	1.97701e-7
17	1.10262e-8
18	2.68453e-6
19	1.32851e-7
20	3.25627e-5
21	1.40097e-6
22	3.45749e-4
23	1.25759e-5
24	3.12543e-3
25	9.22822e-5
26	2.30993e-2
27	5.19161e-4
28	1.30907e-1
29	1.99115e-3
30	5.05844e-1
31	3.90625e-3
32	1.00000

These probabilities were generated using the following Mathematic fragment (written by Jim Davis):

```
Do[Print["n = ", 160 - j, "  ", "Minimum distance = ", 33 - j, "  ",
    "# of erasures = ", j, "  ", "Proportion of random words in a ball = ",
    N[Sum[Binomial[160 - j, i] 255^i, {i, 0,
           16 - Floor[(j + 1)/2]}] 256^(128)/256^(160 - j), 10]], {j, 0, 32}]
```

7 Appendix B – calc_symbol_probs()

```
void calc_symbol_prob(tchannelcfg *c, int average) {
  int sp;
  int delta;
  int fixed;
  int mask;

  /* Calculate symbol probability to achieve average errors */
  sp = (average * 128 + 80) / 160;

  /* Set delta to the largest power of two that is <= sp */
  delta = largest_power_of_two(sp);

  /* Now, ensure distribution of sp is centred on sp */
  fixed = sp - delta;
  mask = delta + delta - 1;

  /* Example1: Call with average = 18
   * -> sp = 14, msb = 3, delta = 8, fixed = 6, mask = 15
   * sp drawn at random from 6..21 -> mean sp = 13.5
   * mean #errors is 160 * 13.5 / 128 == 16.875
   * min #errors is 160 * 6 / 128 == 7.5
   * max #errors is 160 * 21 / 128 == 26.25
   */
  /* Example2: Call with average = 16
   * -> sp = 13, msb = 3, delta = 8, fixed = 5, mask = 15
   * sp drawn at random from 5..20 -> mean sp = 12.5
   * mean #errors is 160 * 12.5 / 128 == 15.625
   * min #errors is 160 * 5 / 128 == 6.25
   * max #errors is 160 * 20 / 128 == 25.00
   */
  /* Example3: Call with average = 6
   * -> sp = 5, msb = 2, delta = 4, fixed = 1, mask = 7
   * sp drawn at random from 1..8 -> mean sp = 4.5
   * mean #errors is 160 * 4.5 / 128 == 5.625
   * min #errors is 160 * 1 / 128 == 1.25
   * max #errors is 160 * 9 / 128 == 11.25
   */
  /* Example4: Call with average = 20
   * -> sp = 16, msb = 4, delta = 16, fixed = 0, mask = 31
   * sp drawn at random from 0..31 -> mean sp = 15.5
   * mean #errors is 160 * 15.5 / 128 == 19.373
   * min #errors is 160 * 0 / 128 == 0
   * max #errors is 160 * 31 / 128 == 38.75
   */
  /* Example5: Call with average = 32
   * -> sp = 26, msb = 4, delta = 16, fixed = 10, mask = 31
   * sp drawn at random from 10..41 -> mean sp = 25.5
   * mean #errors is 160 * 25.5 / 128 == 31.875
   * min #errors is 160 * 10 / 128 == 12.50
   * max #errors is 160 * 41 / 128 == 51.25
   */

   c->prob_symbol_fixed = fixed;
   c->prob_symbol_mask = mask;
}
```

8 Appendic C – `channel_fixed_init()`

```
void channel_init_fixed (int n, tconfig *cfg, int eprob, int num) {
  tchannelcfg *c = n ? &(cfg->c1) : &(cfg->c0);
  int stepmask;
  int startmask;
  int minend;
  int maxend;
  int minstep;
  int maxstep;
  int minstart;
  int maxstart;
  if (num == 1) {
    stepmask = 0;
  } else {
    stepmask = largest_power_of_two(159 / (num - 1)) - 1;
  }
  minstep = 1;
  maxstep = 1 + stepmask;
  startmask = largest_power_of_two(160 - ((num - 1) * maxstep)) - 1;
  minstart = 0;
  maxstart = startmask;
  minend = minstart + (num - 1) * (minstep);
  maxend = maxstart + (num - 1) * (maxstep);
  if (maxend >= 160) {
    printf("there is something wrong with channel_init_fixed()!\n");
    printf("num=%d, stepmask=%d, startmask=%d, minend=%d, maxend=%d\n",
      num, stepmask, startmask, minend, maxend);
    exit(1);
  }
  c->prob_enable = eprob;
  c->prob_symbol_fixed = 128;
  c->prob_symbol_mask = 0;
  c->step_prob = 0;
  c->start_fixed = 0;
  c->start_mask = startmask;
  c->step0_fixed = 0;
  c->step0_mask = 0;
  c->step1_fixed = 1;
  c->step1_mask = stepmask;
  c->maximum = num;
}
```