A Specification for Rijndael, the AES Algorithm

1. Notation and Conventions

1.1 Rijndael Inputs and Outputs

The input, output and cipher key for Rijndael are sequences containing 128, 160, 192, 224 or 256 bits, where input and output sequences have the same length (called the block size). Here a bit is a binary digit (0 or 1) and ‘length’ refers to the number of elements in a sequence (in this case bits). In general the the block size and the key length can be any of the five allowed values but for the Advanced Encryption Standard (AES) the block size is fixed at 128 bits and the key length can only be 128, 192 or 256 bits.

Sequences and sub-sequences will be enumerated from zero to one less than the number of elements in the sequence. The number associated with a sequence element is called its index and sequences will be presented with lower numbered elements to the left. Unless specified, the enumeration order of sub-sequences will match that of the sequence from which they are derived. For Rijndael input, output and key sequences, the number associated with a bit will hence be in one of the five ranges 0 ≤ i < 128, 0 ≤ i < 160, 0 ≤ i < 192, 0 ≤ i < 224 or 0 ≤ i < 256.

1.2 Bytes

Bit sequences can be represented as one-dimensional arrays of 8-bit sub-sequences called bytes where byte n consists of the sub-sequence 8n to 8n+7. In such an array, denoted by a, the n’th byte will be referred to as either \( a_n \) or \( a[8n] \), where n is in one of the ranges 0 ≤ n < 16, 0 ≤ n < 20, 0 ≤ n < 24, 0 ≤ n < 28 or 0 ≤ n < 32. The order i of a bit within a byte has a value 7 – k, where k is the bit’s index, and the bit with order i in a byte b will be denoted by \( b_i \). Internally bytes represent finite field elements using:

\[
b_0 \cdot x^7 + b_1 \cdot x^6 + b_2 \cdot x^5 + b_3 \cdot x^4 + b_4 \cdot x^3 + b_5 \cdot x^2 + b_6 \cdot x^1 + b_7 \cdot x^0 = \sum_{i=0}^{7} b_i \cdot x^i
\]  

(1.2.1)

in a polynomial representation. Externally bytes have their normal meanings. The external integer byte value \( 0x01 \) is mapped through the interface (i.e. without translation) to the internal finite field value \{01\} (this notation is described in section 1.3 below).

1.3 Byte Literals

The values of bytes will be presented as a concatenation of their bits between braces with higher numbered bits to the left. Hence \{011000011\} identifies a specific finite field element. It is also convenient to denote byte values using hexadecimal notation, with each of two groups of four bits being denoted by a character as follows.

<table>
<thead>
<tr>
<th>bit pattern</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>a</td>
</tr>
<tr>
<td>1011</td>
<td>b</td>
</tr>
<tr>
<td>1100</td>
<td>c</td>
</tr>
<tr>
<td>1101</td>
<td>d</td>
</tr>
<tr>
<td>1110</td>
<td>e</td>
</tr>
<tr>
<td>1111</td>
<td>f</td>
</tr>
</tbody>
</table>

Hence the value \{011000011\} can also be written as \{63\}, where the character denoting the 4-bit group containing the higher numbered bits is again to the left. Some finite field operations add an extra bit \( b_8 \) to the left of an 8-bit byte. Where this bit is present it will appear immediately to the left of the left brace as, for example, in \{1b\}.

1.4 The Rijndael State

Internally Rijndael operates on a two dimensional array of bytes called the state that contains 4 rows and \( Nc \) columns, where \( Nc \) is the input sequence length divided by 32. In this state array, denoted by the symbol \( s \), each individual byte has two indexes: its row
number \( r \), in the range \( 0 \leq r < 4 \), and its column number \( c \), in the range \( 0 \leq c < Nc \), hence allowing it to be referred to either as \( s_{r,c} \) or \( s[r, c] \). For AES the range for \( c \) is \( 0 \leq c < 4 \) since \( Nc \) has a fixed value of 4.

At the start (end) of an encryption or decryption operation the bytes of the cipher input (output) are copied to (from) this state array in the order shown in Figure 1.

![Figure 1 – Input to, and output from, the cipher state array](image)

Hence at the start of encryption or decryption the input array \( in \) is copied to the state array according to the scheme:

\[
s[r, c] = in[r + 4c] \quad \text{for } 0 \leq r < 4 \text{ and } 0 \leq c < Nc \quad (1.4.1)
\]

and when the cipher is complete the state is copied to the output array \( out \) according to:

\[
out[r + 4c] = s[r, c] \quad \text{for } 0 \leq r < 4 \text{ and } 0 \leq c < Nc \quad (1.4.2)
\]

1.5 Arrays of 32-bit Words

The four bytes in each column of the state can be thought of as an array of four bytes indexed by the row number \( r \) or as a single 32-bit word (bytes within all 32-bit words will always be enumerated using the index \( r \)). The state can hence be considered as a one-dimensional array of words for which the column number \( c \) provides the array index.

The key schedule for Rijndael, described fully in Section 4, is an array of 32-bit words, denoted by the symbol \( k \), with the lower elements initialised from the cipher key input so that byte \( 4i+r \) of the key is copied into byte \( r \) of key schedule word \( k[i] \). The cipher iterates through a number of cycles, called rounds, each of which uses \( Nc \) words from this key schedule. Hence the key schedule can also be viewed as an array of round keys, each of which consists of an \( Nc \) word sub-array. Hence word \( c \) of round key \( n \), which is \( k[Nc * n + c] \), will also be referred to using two dimensional array notation as either \( k[n,c] \) or \( k_{n,c} \). Here the round key for round \( n \) as a whole, an \( Nc \) word sub-array, will sometimes be referred to by replacing the second index with ‘-’ as in \( k[n,-] \) and \( k_{n,-} \).

2. Finite Field Operations

2.1 Finite Field Addition

The addition of two finite field elements is achieved by adding the coefficients for corresponding powers in their polynomial representations, this addition being performed in \( GF(2) \), that is, modulo 2, so that \( 1 + 1 = 0 \). Consequently, addition and subtraction are both equivalent to an exclusive-or operation on the bytes that represent field elements.

Addition operations for finite field elements will be denoted by the symbol \( \oplus \). For example, the following expressions are equivalent:

\[
(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) \equiv x^7 + x^6 + x^4 + x^2 \quad \text{(polynomial notation)}
\]

\[
\{01010111\} \oplus \{10000011\} \equiv \{11010100\} \quad \text{(binary notation)}
\]

\[
\{57\} \oplus \{83\} \equiv \{d4\} \quad \text{(hexadecimal notation)}
\]
2.2 Finite Field Multiplication

Finite field multiplication is more difficult than addition and is achieved by multiplying the polynomials for the two elements concerned and collecting like powers of \( x \) in the result. Since each polynomial can have powers of \( x \) up to 7, the result can have powers of \( x \) up to 14 and will no longer fit within a single byte.

This situation is handled by replacing the result with the remainder polynomial after division by a special eighth order irreducible polynomial, which, for Rijndael, is:

\[
m(x) = x^8 + x^4 + x^3 + x + 1
\]  

(2.2.1)

Since this polynomial has powers of \( x \) up to 8 it cannot be represented by a single byte and will be written as either \( 1{00011011} \) or \( 1{1b} \) as indicated earlier. This process is illustrated in the following example of the product \( \{57\} \bullet \{83\} \equiv \{c1\} \) (where \( \bullet \) is used to represent finite field multiplication):

\[
\begin{align*}
(x^6 + x^4 + x^2 + x + 1) & \bullet (x^7 + x + 1) \Rightarrow \\
(x^6 + x^4 + x^2 + x + 1) & \bullet x = x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x \\
(x^6 + x^4 + x^2 + x + 1) & \bullet 1 = x^6 + x^4 + x^2 + x + 1
\end{align*}
\]

This intermediate result is now divided by \( m(x) \) above:

\[
\begin{align*}
(x^8 + x^4 + x^3 + x + 1) & \bullet x^5 = \\
\text{subtract to give intermediate remainder} \\
(x^8 + x^4 + x^3 + x + 1) & \bullet x^3 = \\
\text{subtract to give the final remainder}
\end{align*}
\]

Multiplication is associative, and there is a neutral element \( \{01\} \); for any binary polynomial \( b(x) \) of degree less than 8, the extended Euclidean algorithm can be used to compute polynomials \( a(x) \) and \( c(x) \), such that:

\[
b(x) \bullet a(x) \oplus m(x) \bullet c(x) = 1 \\
a(x) \bullet b(x) \mod m(x) = 1
\]

(2.2.2)

(2.2.3)

which shows that the polynomials \( a(x) \) and \( b(x) \) are mutual inverses. Furthermore:

\[
a(x) \bullet (b(x) \oplus c(x)) = a(x) \bullet b(x) \oplus a(x) \bullet c(x)
\]

(2.2.4)

It hence follows that the set of 256 byte values, with the XOR as addition and multiplication as defined above has the structure of the finite field GF(256).

2.3 Multiplication by Repeated Shifts

The finite field element \( \{00000010\} \) is the polynomial \( x \), which means that multiplying another element by this value increases all it’s powers of \( x \) by 1. This is equivalent to shifting its byte representation up by one bit so that the bit at position \( i \) moves to position \( i+1 \). If the top bit is set prior to this move it will overflow to create an \( x^8 \) term, in which case the modular polynomial is added to cancel this additional bit, leaving a result that fits within a single byte.

For example, multiplying \( \{11001000\} \) by \( x \), that is \( \{00000010\} \), the initial result is \( 1\{10010000\} \). The ‘overflow’ bit is then removed by adding \( 1\{00011011\} \), the modular polynomial, using an exclusive-or operation to give a final result of \( \{10001011\} \).
By repeating this process, a finite field element can be multiplied by all powers of $x$ from 0 to 7. Multiplication of this element by any other field element can then be achieved by adding the results for the appropriate powers of $x$. For example, Table 1 carries out this calculation for the product of the field elements $\{57\}$ and $\{83\}$ to give $\{c1\}$.

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
p & \{57\} \cdot x^p & \circ \cdot m(x) & \{57\} \cdot x^p & \{83\} \cdot \circ \cdot \text{to result} & \text{result} \\
\hline
0 & (01010111) & (01010111) & 1 & (01010111) & (01010111) \\
1 & (10101110) & (10101110) & 1 & (10101110) & (11111001) \\
2 & 1(01011100) & 1(00011011) & (01001111) & 0 & \\
3 & 1(00011110) & (10001110) & 0 & \\
4 & 1(00011110) & 1(00011110) & (00000111) & 0 & \\
5 & (00001110) & (00001110) & 0 & \\
6 & (00011110) & (00011110) & 0 & \\
7 & (00111100) & (00111100) & 1 & (00111100) & (11000001) \\
\hline
\end{array}
\]

Table 1 – Finite field multiply $\{57\} \cdot \{83\}$

2.4 Finite Field Multiplication Using Tables

When certain finite field elements (known as generators) are repeatedly multiplied to produce a list of their powers, $g^p$, they progressively generate all 255 non-zero elements in the field. When $p$ reaches 256 the original field element recurs, indicating that $g^{255}$ is equal to $\{01\}$. The $p$ values for each field element can be thought of as logarithms and provide a way of converting multiplication into addition. Hence the two elements $a = g^\alpha$ and $b = g^\beta$ have the product $a \cdot b = g^{\alpha + \beta}$. With a ‘logarithm’ table listing the power of the generator for each finite field element we can hence find the powers $\alpha$ and $\beta$ corresponding to the elements $a$ and $b$ and add these values to find the power of $g$ for the result. A reverse table can then be used to look up the product element.

Since the two initial power values can each be as high as 255, their sum may be greater than 255 but if this occurs, 255 can be subtracted from the value to bring it into the range of the tables because $g^{255} = \{01\}$. Although decimal exponents have been used in this explanation, all exponents in what follows are in hexadecimal notation.

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
x & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & a & b & c & d & e & f \\
\hline
0 & 00 & 19 & 01 & 32 & 02 & 1a & c6 & 4b & c7 & 1b & 68 & 33 & ee & df & 03 \\
1 & 64 & 04 & e0 & 0e & 34 & 8d & 81 & ef & 4c & 71 & 08 & c8 & f8 & 69 & 1c & c1 \\
2 & 7d & c2 & 1d & b5 & f9 & b9 & 27 & 6a & 4d & e4 & a6 & 72 & 9a & c9 & 09 & 78 \\
3 & 65 & 2f & 8a & 05 & 21 & 0f & e1 & 24 & 12 & f0 & 82 & 45 & 35 & 93 & da & 8e \\
4 & 96 & 8f & db & bd & 36 & d0 & ce & 94 & 13 & 5c & d2 & f1 & 40 & 46 & 83 & 38 \\
5 & 66 & dd & fd & 30 & bf & 06 & 8b & 62 & b3 & 25 & e2 & 98 & 22 & 88 & 91 & 10 \\
6 & 7e & 6e & 48 & c3 & a3 & b6 & 1e & 42 & 3a & 6b & 28 & 54 & fa & 85 & 3d & ba \\
7 & 2b & 79 & 0a & 15 & 9b & 9f & 5e & ca & 4e & d4 & ac & e5 & f3 & 73 & a7 & 57 \\
8 & af & 58 & a8 & 50 & f4 & ea & d6 & 74 & 4f & ef & e9 & d5 & e7 & e6 & ad & e8 \\
9 & 2c & d7 & 75 & 7a & eb & 16 & 0b & f5 & 59 & cb & 5f & b0 & 9c & a9 & 51 & a0 \\
\hline
a & 7f & 0c & f6 & 6f & 17 & c4 & 49 & ec & d8 & 43 & 1f & 2d & 4a & 76 & 7b & b7 \\
\hline
b & cc & bb & 3e & 5a & fb & 60 & b1 & 86 & 3b & 52 & a1 & 6c & a5 & 55 & 29 & 9d \\
\hline
c & 97 & b2 & 87 & 90 & 61 & be & dc & fc & bc & 95 & cf & cd & 37 & 3f & 5b & d1 \\
\hline
d & 53 & 39 & 84 & 3c & 41 & a2 & 6d & 47 & 14 & 2a & 9e & 5d & 56 & f2 & d3 & ab \\
e & 44 & 11 & 92 & d9 & 23 & 20 & 2e & 89 & b4 & 7c & b8 & 26 & 77 & 99 & e3 & a5 \\
f & 67 & 4a & ed & de & c5 & 31 & fe & 18 & 0d & 63 & 8c & 80 & c0 & f7 & 70 & 07 \\
\hline
\end{array}
\]

Table 2 – ‘Logs’ – $L$ values such that $\{xy\} = \{03\}^L$ for a given a finite field element $\{xy\}$
2.5 Polynomials with Coefficients in GF(256)

A specification for the AES algorithm, Rijndael (by Joan Daemen & Vincent Rijmen)

Multiplication is achieved by algebraically expanding the polynomial product and collecting like powers of $x$ to give:

$$ c(x) = c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 $$

where:

$$
\begin{align*}
    c_0 &= a_0 \cdot b_0 & c_4 &= a_1 \cdot b_1 \\
    c_1 &= a_1 \cdot b_0 \oplus a_0 \cdot b_1 & c_5 &= a_2 \cdot b_2 \\
    c_2 &= a_2 \cdot b_0 \oplus a_1 \cdot b_1 \oplus a_0 \cdot b_2 & c_5 &= a_3 \cdot b_3 \\
    c_3 &= a_3 \cdot b_0 \oplus a_2 \cdot b_1 \oplus a_1 \cdot b_2 \oplus a_0 \cdot b_3
\end{align*}
$$

Table 3 – ‘Antilogs’ – field elements $\{E\}$ such that $\{E\} = \{03\}^{(xy)}$ given the power $(xy)$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$f$</th>
<th>$e$</th>
<th>$d$</th>
<th>$c$</th>
<th>$b$</th>
<th>$a$</th>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
<th>$6$</th>
<th>$7$</th>
<th>$8$</th>
<th>$9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0f</td>
<td>03</td>
<td>05</td>
<td>00</td>
<td>01</td>
<td>02</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>0a</td>
<td>0b</td>
<td>0c</td>
<td>0d</td>
<td>0e</td>
<td>0f</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>5f</td>
<td>e1</td>
<td>38</td>
<td>48</td>
<td>d8</td>
<td>73</td>
<td>95</td>
<td>a4</td>
<td>f7</td>
<td>02</td>
<td>06</td>
<td>0a</td>
<td>1e</td>
<td>22</td>
<td>66</td>
<td>aa</td>
</tr>
<tr>
<td>2</td>
<td>e5</td>
<td>34</td>
<td>5c</td>
<td>e4</td>
<td>37</td>
<td>59</td>
<td>eb</td>
<td>26</td>
<td>6a</td>
<td>be</td>
<td>d9</td>
<td>70</td>
<td>90</td>
<td>9b</td>
<td>e6</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>53</td>
<td>f5</td>
<td>04</td>
<td>0c</td>
<td>14</td>
<td>3c</td>
<td>44</td>
<td>cc</td>
<td>4f</td>
<td>d1</td>
<td>68</td>
<td>b8</td>
<td>d3</td>
<td>6e</td>
<td>b2</td>
<td>cd</td>
</tr>
<tr>
<td>4</td>
<td>4c</td>
<td>d4</td>
<td>67</td>
<td>a9</td>
<td>e0</td>
<td>3b</td>
<td>4d</td>
<td>d7</td>
<td>62</td>
<td>a6</td>
<td>f1</td>
<td>08</td>
<td>18</td>
<td>28</td>
<td>78</td>
<td>88</td>
</tr>
<tr>
<td>5</td>
<td>83</td>
<td>9e</td>
<td>b9</td>
<td>d0</td>
<td>6b</td>
<td>bd</td>
<td>dc</td>
<td>7f</td>
<td>81</td>
<td>98</td>
<td>b3</td>
<td>ce</td>
<td>49</td>
<td>db</td>
<td>76</td>
<td>9a</td>
</tr>
<tr>
<td>6</td>
<td>b5</td>
<td>c4</td>
<td>5f</td>
<td>99</td>
<td>30</td>
<td>50</td>
<td>f0</td>
<td>0b</td>
<td>1d</td>
<td>27</td>
<td>69</td>
<td>bb</td>
<td>d6</td>
<td>61</td>
<td>a3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>fe</td>
<td>19</td>
<td>2b</td>
<td>7d</td>
<td>87</td>
<td>92</td>
<td>ad</td>
<td>ec</td>
<td>2f</td>
<td>71</td>
<td>93</td>
<td>ae</td>
<td>99</td>
<td>80</td>
<td>2b</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>fb</td>
<td>16</td>
<td>3a</td>
<td>4e</td>
<td>2d</td>
<td>6d</td>
<td>b7</td>
<td>c2</td>
<td>5d</td>
<td>e7</td>
<td>32</td>
<td>56</td>
<td>fa</td>
<td>15</td>
<td>3f</td>
<td>41</td>
</tr>
<tr>
<td>9</td>
<td>c3</td>
<td>5e</td>
<td>e2</td>
<td>3d</td>
<td>47</td>
<td>c9</td>
<td>40</td>
<td>c0</td>
<td>5b</td>
<td>ed</td>
<td>2c</td>
<td>74</td>
<td>9c</td>
<td>bf</td>
<td>da</td>
<td>75</td>
</tr>
<tr>
<td>a</td>
<td>9f</td>
<td>ba</td>
<td>d5</td>
<td>64</td>
<td>ac</td>
<td>ef</td>
<td>2a</td>
<td>7e</td>
<td>82</td>
<td>9d</td>
<td>bc</td>
<td>df</td>
<td>7a</td>
<td>8e</td>
<td>89</td>
<td>80</td>
</tr>
<tr>
<td>b</td>
<td>9b</td>
<td>b6</td>
<td>c1</td>
<td>58</td>
<td>e8</td>
<td>23</td>
<td>65</td>
<td>a7</td>
<td>18</td>
<td>5e</td>
<td>76</td>
<td>9f</td>
<td>b0</td>
<td>cb</td>
<td>ad</td>
<td>54</td>
</tr>
<tr>
<td>c</td>
<td>fc</td>
<td>1f</td>
<td>21</td>
<td>63</td>
<td>a5</td>
<td>f4</td>
<td>07</td>
<td>09</td>
<td>1b</td>
<td>2d</td>
<td>77</td>
<td>99</td>
<td>b0</td>
<td>cb</td>
<td>46</td>
<td>ca</td>
</tr>
<tr>
<td>d</td>
<td>45</td>
<td>cf</td>
<td>4a</td>
<td>de</td>
<td>79</td>
<td>8b</td>
<td>86</td>
<td>91</td>
<td>a8</td>
<td>e3</td>
<td>42</td>
<td>c6</td>
<td>51</td>
<td>f3</td>
<td>0e</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>12</td>
<td>36</td>
<td>5a</td>
<td>ee</td>
<td>29</td>
<td>7b</td>
<td>8d</td>
<td>8c</td>
<td>8f</td>
<td>8a</td>
<td>85</td>
<td>94</td>
<td>a7</td>
<td>f2</td>
<td>0d</td>
<td>17</td>
</tr>
<tr>
<td>f</td>
<td>39</td>
<td>4b</td>
<td>dd</td>
<td>7c</td>
<td>84</td>
<td>97</td>
<td>a2</td>
<td>fd</td>
<td>1c</td>
<td>24</td>
<td>6c</td>
<td>b4</td>
<td>c7</td>
<td>52</td>
<td>f6</td>
<td>01</td>
</tr>
</tbody>
</table>
with \( \circ \) and \( \oplus \) representing finite field multiplication and addition (XOR) respectively. This result requires six bytes to represent its coefficients but it can be reduced modulo a degree 4 polynomial to produce a result that is of degree less than 4.

In Rijndael the polynomial used is \( x^4 + 1 \) and reduction produces the following polynomial coefficients:

\[
\begin{align*}
    d_3 &= b_3 \circ b_0 \oplus a_2 \circ b_1 \oplus a_1 \circ b_2 \oplus a_0 \circ b_3 \\
    d_2 &= a_2 \circ b_0 \oplus a_1 \circ b_1 \oplus a_0 \circ b_2 \oplus a_3 \circ b_3 \\
    d_1 &= a_1 \circ b_0 \oplus a_0 \circ b_1 \oplus a_3 \circ b_2 \oplus a_2 \circ b_3 \\
    d_0 &= a_0 \circ b_0 \oplus a_3 \circ b_1 \oplus a_2 \circ b_2 \oplus a_1 \circ b_3
\end{align*}
\] (2.5.5)

If one of the polynomials is fixed, this can conveniently be written in matrix form as:

\[
\begin{bmatrix}
    d_3 \\
    d_2 \\
    d_1 \\
    d_0
\end{bmatrix} =
\begin{bmatrix}
    a_0 & a_1 & a_2 & a_3 \\
    a_3 & a_0 & a_1 & a_2 \\
    a_2 & a_3 & a_0 & a_1 \\
    a_1 & a_2 & a_3 & a_0
\end{bmatrix}
\begin{bmatrix}
    b_3 \\
    b_2 \\
    b_1 \\
    b_0
\end{bmatrix}
\] (2.5.6)

Because \( x^4 + 1 \) is not an irreducible polynomial, not all polynomial multiplications are invertible. For Rijndael, however, a polynomial that has an inverse has been chosen:

\[
a(x) = \{03\} x^3 + \{01\} x^2 + \{01\} x + \{02\}
\] (2.5.7)

\[
a^{-1}(x) = \{0b\} x^3 + \{0d\} x^2 + \{09\} x + \{0e\}
\] (2.5.8)

Another polynomial that Rijndael uses has \( a_0 = a_2 = a_3 = \{00\} \) and \( a_1 = \{01\} \), which is the polynomial \( x \). Inspection of (2.5.6) above will show that its effect is to form the output word by rotating the bytes in the input word so that \([b_3, b_2, b_1, b_0]\) is transformed into \([b_2, b_1, b_0, b_3]\), with bytes moving to higher index positions and the top byte wrapping round to the lowest position. Higher powers of \( x \) correspond to the other cyclic permutations of the four bytes within a 32-bit word. The \texttt{RotWord} function that is used in the key schedule corresponds to \( x^3 \).

### 3. The Cipher

At the start of the cipher the cipher input is copied into the internal state using the conventions described in Section 1.4. An initial round key is then added and the state is then transformed by iterating a \texttt{round function} in a number of cycles. The number of cycles \( N_n \) varies with the key length and block size. On completion the final state is copied into the cipher output using the same conventions.

The round function is parameterised using a round key which consists of an \( N_c \) word subarray from the key schedule. The latter is considered either as a one-dimensional array of 32-bit words or an array of round keys with a structure and initialisation as described in section 1.5. In general the length of the cipher input, the cipher output and the cipher state, \( N_c \), measured in multiples of 32 bits, is 4, 5, 6, 7 or 8 but the AES standard only allows a length of 4. The length of the cipher key, \( N_k \), has the same values but only lengths of 4, 6 or 8 are allowed in the AES standard.

The cipher is described in the following pseudo code with the individual transformations and the key schedule described subsequently. Here the key schedule is treated as an array of \( N_n + 1 \) individual round keys, each of which is itself an array of \( N_c \) words.
Cipher(byte in[4*Nc], byte out[4*Nc], word k[Nn+1,Nc], Nc, Nn)

Begin

byte state[4,Nc] // The notation k[Nn+1,Nc] above indicates that
// the array k contains Nn + 1 individual round
state = in // keys that are each arrays of Nc words
XorRoundKey(state, k[0,-], Nc) // k[0,-] = k[0..Nc-1]

for round = 1 step 1 to Nn – 1
SubBytes(state, Nc)
ShiftRows(state, Nc)
MixColumns(state, Nc)
XorRoundKey(state, k[round,-], Nc) // k[round*Nc..(round+1)*Nc-1]
end for

SubBytes(state, Nc)
ShiftRows(state, Nc)
XorRoundKey(state, k[Nn,-], Nc) // k[Nn*Nc..(Nn+1)*Nc-1]

out = state
end

The number of rounds for the cipher (Nn) varies with the block length and the key length as shown in the following table. Remember that for AES the block size, Nc, is fixed at 4 and the key, Nk, can only have the lengths 4, 6 or 8.

<table>
<thead>
<tr>
<th>Nn</th>
<th>The larger of Nc or Nk</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 4 – The number of rounds as a function of block and key size

3.1 The SubBytes Transformation

The SubBytes transformation is a non-linear byte substitution that acts on every byte of the state in isolation to produce a new byte value using an S-box substitution table. The action of this transformation is illustrated in Figure 2 for a block size of 6.

\[
\begin{array}{cccccc}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} & S_{0,4} & S_{0,5} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} & S_{1,5} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} & S_{2,5} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} & S_{3,5} \\
\end{array}
\rightarrow
\begin{array}{cccc}
S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\
S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\
S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\
S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \\
\end{array}
\rightarrow
\begin{array}{cccc}
S_{r,c} & S_{r,c} & S_{r,c} & S_{r,c} \\
S_{r,c} & S_{r,c} & S_{r,c} & S_{r,c} \\
S_{r,c} & S_{r,c} & S_{r,c} & S_{r,c} \\
S_{r,c} & S_{r,c} & S_{r,c} & S_{r,c} \\
\end{array}
\]

Figure 2 – SubBytes acts on every byte in the state in isolation

This substitution, which is invertible, is constructed by composing two transformations:

1. First the multiplicative inverse in the finite field described earlier (with element \{00\} mapped to itself).
2. Second the affine transformation over GF(2) defined by:

\[
b_i' = b_i \oplus b_{(i+4)\text{mod}8} \oplus b_{(i+5)\text{mod}8} \oplus b_{(i+6)\text{mod}8} \oplus b_{(i+7)\text{mod}8} \oplus c_i
\] (3.1.1)

for \(0 \leq i < 8\) where \(b_i\) is bit \(i\) of the byte and \(c_i\) is bit \(i\) of a byte \(c\) with the value \{63\} or \{01100011\}. Here and elsewhere a prime on a variable on the left of an equation indicates that its value is to be updated with the value on the right.

In matrix form the latter component of the S-box transformation can be expressed as:
The final result of this two stage transformation is given in the following table.

<table>
<thead>
<tr>
<th>hex</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 63</td>
<td>7c 77</td>
</tr>
<tr>
<td>1 ca</td>
<td>82 c9</td>
</tr>
<tr>
<td>2 b7</td>
<td>fd 93</td>
</tr>
<tr>
<td>3 04</td>
<td>c7 23</td>
</tr>
<tr>
<td>4 09</td>
<td>83 2c</td>
</tr>
<tr>
<td>5 53</td>
<td>d1 00</td>
</tr>
<tr>
<td>6 d0</td>
<td>ef aa</td>
</tr>
<tr>
<td>7 51</td>
<td>a3 40</td>
</tr>
<tr>
<td>8 cd</td>
<td>0c 13</td>
</tr>
<tr>
<td>9 60</td>
<td>81 4f</td>
</tr>
<tr>
<td>a e0</td>
<td>32 3a</td>
</tr>
<tr>
<td>b e7</td>
<td>c8 37</td>
</tr>
<tr>
<td>c ba</td>
<td>78 25</td>
</tr>
<tr>
<td>d 70</td>
<td>3e b5</td>
</tr>
<tr>
<td>e e1</td>
<td>f8 98</td>
</tr>
<tr>
<td>f 8c</td>
<td>a1 89</td>
</tr>
</tbody>
</table>

Table 5 – The Substitution Table – Sbox[xy] (in hexadecimal)

The pseudo code for this transformation is as follows.

SubBytes(byte state[4,Nc], Nc)
begin
  for r = 0 step 1 to 3
    for c = 0 step 1 to Nc - 1
      state[r,c] = Sbox[state[r,c]]
  end for
end for

3.2 The ShiftRows Transformation

The ShiftRows transformation operates individually on each of the last three rows of the state by cyclically shifting the bytes in the row such that:

\[ s_{r,c} = s_{(c + h(r, Nc)) \mod Nc} \quad \text{for } 0 \leq c < Nc \text{ and } 0 < r < 4 \]  

(3.2.1)

where the shift amount \( h(r, Nc) \) depends on row number \( r \) and block length as follows:

<table>
<thead>
<tr>
<th>( h(r, Nc) )</th>
<th>row (r)</th>
<th>( Nc )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, 5, 6</td>
<td>1 2 3</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1 2 4</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 6 – Shift offsets for different rows and block lengths

Note that the AES block size, \( Nc \), is fixed at 4.
This has the effect of moving bytes to lower positions in the row except that the lowest bytes wrap around into the top of the row (note that a prime on a variable indicates an updated value). The action of this transformation is illustrated in Figure 3 for a cipher block size of 6.

The pseudo code for this transformation is as follows.

```plaintext
ShiftRows(byte state[4,Nc], Nc) begin
  byte t[Nc]
  for r = 1 step 1 to 3
    for c = 0 step 1 to Nc - 1
      t[c] = state[r, (c + h(r,Nc)) mod Nc]
    end for
    for c = 0 step 1 to Nc - 1
      state[r,c] = t[c]
    end for
  end for
end
```

3.3 The MixColumns Transformation

The MixColumns transformation acts independently on every column of the state and treats each column as a four-term polynomial as described in Section 2.6.

In matrix form the transformation used given in equation (3.3.1), where all the values are finite field elements as discussed in Section 2.

\[
\begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix}
= \begin{bmatrix}
  02 & 01 & 01 & 03 \\
  03 & 02 & 01 & 01 \\
  01 & 03 & 02 & 01 \\
  01 & 01 & 03 & 02
\end{bmatrix}
\begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix}
\] for \(0 \leq c < Nc\) (3.3.1)

The action of this transformation is illustrated in Figure 4 for a cipher block size of 6.

The pseudo code for this transformation is as follows, where the function \(\text{FFmul}(x, y)\) returns the product of two finite field elements \(x\) and \(y\).
MixColumns(byte state[4,Nc], Nc)
begin
  byte t[4]
  for c = 0 step 1 to Nc - 1
    for r = 0 step 1 to 3
      t[r] = state[r,c]
    end for
    for r = 0 step 1 to 3
      state[r,c] = FFmul(0x02, t[r]) xor
                    FFmul(0x03, t[(r + 1) mod 4]) xor
                    t[(r + 2) mod 4] xor t[(r + 3) mod 4]
    end for
  end for
end

3.4 The XorRoundKey Transformation

In the XorRoundKey transformation $N_c$ words from the key schedule (the round key described later) are each added (XOR'd) into the columns of the state so that:

$$[b'_{3c}, b'_{2c}, b'_{1c}, b'_{0c}] = [b_{3c}, b_{2c}, b_{1c}, b_{0c}] \oplus [k_{round,c}] \text{ for } 0 \leq c < N_c$$

(3.4.1)

where the round key words $k_{round,c}$ (shortened to $k_c'$ in the diagram below) will be described later. The round number, round, is in the range $0 \leq round \leq N_n$, with the value of 0 being used to denote the initial round key that is applied before the round function.

![Figure 5 – Words from the key schedule are XOR’d into columns in the state](image)

The action of this transformation is illustrated in Figure 5 for a cipher block size of 6. The byte address within each word of the key schedule is that described in Section 1.

The pseudo code for this transformation is as follows, where xbyte(r, w) extracts byte r from word w.

XorRoundKey(byte state[4,Nc], word k[round,-], Nc)
Begin
  for c = 0 step 1 to Nc - 1
    for r = 0 step 1 to 3
      state[r,c] = state[r,c] xor xbyte(r, k[round,c])
    end for
  end for
end

4. The Key Schedule

The round keys are derived from the cipher key by means of a key schedule with each round requiring $N_c$ words of key data which, with an additional initial set, makes a total of $N_c(N_n + 1)$ words, where $N_n$ is the number of cipher rounds. This key schedule is considered either as a one dimensional array $k$ of $N_c(N_n + 1)$ 32-bit words with an index $i$ in the range $0 \leq i < N_c(N_n + 1)$ or as a two dimensional array $k[n,c]$ of $N_n + 1$ round keys, each or which individually consists of a sub-array of $N_c$ words.

The expansion of the input key into the key schedule proceeds according to the following pseudo code. The function SubWord(x) gives an output word for which the S-box substitution has been individually applied to each of the four bytes of its input $x$. The
function \text{RotWord}(x) \text{ converts an input word } [b_3, b_2, b_1, b_0] \text{ to an output } [b_0, b_1, b_2, b_3]. \text{ The word array } \text{Rcon}[i] \text{ contains the values } [0, 0, 0, x^{i-1}] \text{ with } x^{i-1} \text{ being the powers of } x \text{ in the field } \text{GF}(256) \text{ discussed in section 2.3 (note that the index } i \text{ starts at 1).} \\

\text{KeyExpansion(byte key}[4*Nk], \text{ word } k[Nn+1,Nc], \text{ Nc, Nk, Nn)} \text{ begin} \\
i = 0 \\
while (i < Nk) \\
\quad k[i] = \text{word} [\text{key}[4*i+3], \text{key}[4*i+2], \text{key}[4*i+1], \text{key}[4*i]] \\
i = i + 1 \\
end while \\
i = Nk \\
while (i < Nc * (Nn + 1)) \\
\quad \text{word temp = } k[i - 1] \\
\quad \text{if (i mod Nk = 0)} \\
\qquad \text{temp = } \text{SubWord(RotWord(temp)) xor Rcon[i / Nk]} \\
\quad \text{else if ((Nk > 6) and (i mod Nk = 4))} \\
\qquad \text{temp = SubWord(temp)} \\
\quad \text{end if} \\
\quad k[i] = k[i - Nk] xor temp \\
i = i + 1 \\
end while \\
end \\

\text{Note that this key schedule, which is illustrated in Figure 6 for } Nk = 4 \text{ and } Nc = 6, \text{ can be generated ‘on-the fly’ if necessary using a buffer of max}(Nc, Nk) \text{ words. It can also be split into separate, somewhat simpler, key schedules for } Nk \leq 6 \text{ and } Nk > 6 \text{ respectively.} \\

\begin{center}
\begin{array}{cccccccccccccccc}
k_0 & k_1 & k_2 & k_3 & k_4 & k_5 & k_6 & k_7 & k_8 & k_9 & k_{10} & k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} & k_{17} \\
\end{array}
\end{center}

\text{Figure 6 – The key schedule and round key selection for } Nk = 4 \text{ and } Nc = 6 \\

5. The Inverse Cipher

\text{The inversion of the cipher code presented in section 3 is straightforward and provides the following pseudo code for the inverse cipher.}

\text{InvCipher(byte in}[4*Nc], \text{ byte out}[4*Nc], \text{ word } k[Nn+1,Nc], \text{ Nc, Nn)} \text{ begin} \\
\text{byte state}[4,Nc] \\
\text{state = in} \\
\text{XorRoundKey(state, k[Nn,-], Nc) // k[Nn*Nc..(Nn+1)*Nc-1]} \\
\text{for round = Nn - 1 step -1 to 1} \\
\qquad \text{InvShiftRows(state, Nc)} \\
\qquad \text{InvSubBytes(state, Nc)} \\
\qquad \text{XorRoundKey(state, k[round,-], Nc) // k[round*Nc..(round+1)*Nc-1]} \\
\qquad \text{InvMixColumns(state, Nc)} \\
\text{end for} \\
\text{InvShiftRows(state, Nc)} \\
\text{InvSubBytes(state, Nc)} \\
\text{XorRoundKey(state, k[0,-], Nc) // k[0..Nc-1]} \\
\text{out = state} \\
end
5.1 The Inverse ShiftRows Transformation

The $\text{InvShiftRows}$ transformation operates individually on each of the last three rows of the state cyclically shifting the bytes in the row such that:

$$s_{r,c+h(r,Nc) \mod Nc} = s_{r,c}, \text{ for } 0 \leq c < Nc \text{ and } 0 < r < 4$$

(5.1.1)

where the cyclic shift values $h(r, Nc)$ are given in Table 6. The pseudo code for this transformation is as follows.

$$\text{InvShiftRows(byte state}[4,Nc], Nc)$$

begin
  byte t[Nc]
  for $r = 1$ step 1 to 3
    for $c = 0$ step 1 to $Nc - 1$
      $t[(c + h(r,Nc)) \mod Nc] = state[r,c]$
    end for
  for $c = 0$ step 1 to $Nc - 1$
    $state[r,c] = t[c]$
  end for
end

5.2 The Inverse SubBytes Transformation

The inverse S-box table needed for the inverse $\text{InvSubBytes}$ transformation is given in Section 3.1. The pseudo code for this transformation is as follows:

$$\text{InvSubBytes(byte state}[4,Nc], Nc)$$

begin
  for $r = 0$ step 1 to 3
    for $c = 0$ step 1 to $Nc - 1$
      $state[r,c] = \text{InvSbox}[state[r,c]]$
    end for
  end for
end

Table 7 gives the full inverse S-box, the inverse of the affine transformation (3.1.1) being:

$$b_i = b_{(i+2)\mod 8} \oplus b_{(i+5)\mod 8} \oplus b_{(i+7)\mod 8} \oplus d_i, \text{ where byte } d = \{05\}$$

(5.2.1)

<table>
<thead>
<tr>
<th>hex</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>52</td>
<td>09</td>
<td>6a</td>
<td>d5</td>
<td>30</td>
<td>36</td>
<td>a5</td>
<td>38</td>
<td>bf</td>
<td>40</td>
<td>a3</td>
<td>9e</td>
<td>81</td>
<td>f3</td>
<td>d7</td>
<td>fb</td>
</tr>
<tr>
<td>1</td>
<td>7c</td>
<td>e3</td>
<td>39</td>
<td>82</td>
<td>9b</td>
<td>2f</td>
<td>ff</td>
<td>87</td>
<td>34</td>
<td>8e</td>
<td>43</td>
<td>44</td>
<td>c4</td>
<td>de</td>
<td>e9</td>
<td>cb</td>
</tr>
<tr>
<td>2</td>
<td>54</td>
<td>7b</td>
<td>94</td>
<td>32</td>
<td>a6</td>
<td>c2</td>
<td>23</td>
<td>3d</td>
<td>ee</td>
<td>4c</td>
<td>95</td>
<td>0b</td>
<td>42</td>
<td>fa</td>
<td>c3</td>
<td>4e</td>
</tr>
<tr>
<td>3</td>
<td>08</td>
<td>2e</td>
<td>a1</td>
<td>66</td>
<td>28</td>
<td>d9</td>
<td>24</td>
<td>b2</td>
<td>76</td>
<td>5b</td>
<td>a2</td>
<td>49</td>
<td>6d</td>
<td>8b</td>
<td>d1</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>f8</td>
<td>f6</td>
<td>64</td>
<td>86</td>
<td>68</td>
<td>98</td>
<td>16</td>
<td>d4</td>
<td>a4</td>
<td>5c</td>
<td>cc</td>
<td>5d</td>
<td>b6</td>
<td>92</td>
<td>e9</td>
</tr>
<tr>
<td>5</td>
<td>6c</td>
<td>70</td>
<td>48</td>
<td>50</td>
<td>fd</td>
<td>ed</td>
<td>b9</td>
<td>da</td>
<td>5e</td>
<td>15</td>
<td>46</td>
<td>57</td>
<td>a7</td>
<td>8d</td>
<td>9d</td>
<td>84</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
<td>d8</td>
<td>ab</td>
<td>00</td>
<td>8c</td>
<td>bc</td>
<td>d3</td>
<td>0a</td>
<td>f7</td>
<td>e4</td>
<td>58</td>
<td>05</td>
<td>b8</td>
<td>b3</td>
<td>45</td>
<td>06</td>
</tr>
<tr>
<td>7</td>
<td>d0</td>
<td>2c</td>
<td>1e</td>
<td>8f</td>
<td>ca</td>
<td>3f</td>
<td>0f</td>
<td>02</td>
<td>c1</td>
<td>af</td>
<td>bd</td>
<td>03</td>
<td>01</td>
<td>13</td>
<td>8a</td>
<td>6b</td>
</tr>
<tr>
<td>8</td>
<td>3a</td>
<td>91</td>
<td>11</td>
<td>41</td>
<td>4f</td>
<td>67</td>
<td>dc</td>
<td>ea</td>
<td>97</td>
<td>f2</td>
<td>cf</td>
<td>ce</td>
<td>f0</td>
<td>b4</td>
<td>e6</td>
<td>73</td>
</tr>
<tr>
<td>9</td>
<td>96</td>
<td>ac</td>
<td>74</td>
<td>22</td>
<td>e7</td>
<td>ad</td>
<td>35</td>
<td>85</td>
<td>e2</td>
<td>f9</td>
<td>37</td>
<td>e8</td>
<td>1c</td>
<td>75</td>
<td>df</td>
<td>6e</td>
</tr>
<tr>
<td>a</td>
<td>47</td>
<td>f1</td>
<td>1a</td>
<td>71</td>
<td>1d</td>
<td>29</td>
<td>c5</td>
<td>89</td>
<td>6f</td>
<td>b7</td>
<td>62</td>
<td>0e</td>
<td>aa</td>
<td>18</td>
<td>be</td>
<td>1b</td>
</tr>
<tr>
<td>b</td>
<td>fc</td>
<td>56</td>
<td>3e</td>
<td>4b</td>
<td>c6</td>
<td>d2</td>
<td>79</td>
<td>20</td>
<td>9a</td>
<td>db</td>
<td>c0</td>
<td>fe</td>
<td>78</td>
<td>cd</td>
<td>5a</td>
<td>f4</td>
</tr>
<tr>
<td>c</td>
<td>1f</td>
<td>dd</td>
<td>a8</td>
<td>33</td>
<td>88</td>
<td>07</td>
<td>c7</td>
<td>31</td>
<td>bl</td>
<td>12</td>
<td>10</td>
<td>59</td>
<td>27</td>
<td>80</td>
<td>ec</td>
<td>5f</td>
</tr>
<tr>
<td>d</td>
<td>60</td>
<td>51</td>
<td>7f</td>
<td>a9</td>
<td>19</td>
<td>b5</td>
<td>4a</td>
<td>0d</td>
<td>2d</td>
<td>e5</td>
<td>7a</td>
<td>9f</td>
<td>93</td>
<td>c9</td>
<td>9c</td>
<td>ef</td>
</tr>
<tr>
<td>e</td>
<td>a0</td>
<td>e0</td>
<td>3b</td>
<td>4d</td>
<td>ae</td>
<td>2a</td>
<td>f5</td>
<td>b0</td>
<td>c8</td>
<td>eb</td>
<td>bb</td>
<td>3c</td>
<td>83</td>
<td>53</td>
<td>99</td>
<td>61</td>
</tr>
<tr>
<td>f</td>
<td>17</td>
<td>2b</td>
<td>04</td>
<td>7e</td>
<td>ba</td>
<td>77</td>
<td>d6</td>
<td>26</td>
<td>e1</td>
<td>69</td>
<td>14</td>
<td>63</td>
<td>55</td>
<td>21</td>
<td>0c</td>
<td>7d</td>
</tr>
</tbody>
</table>

Table 7 – The Inverse Substitution Table – InvSbox[xy] (in hexadecimal)

5.3 The Inverse XorRoundKey Transformation

The $\text{XorRoundKey}$ transformation is its own inverse.
5.4 The Inverse MixColumns Transformation

The InvMixColumns transformation acts independently on every column of the state and treats each column as a four-term polynomial as described in Section 2.6. In matrix form the transformation used given in equation (5.4.1), where all the values are finite field elements as discussed in Section 2.

\[
\begin{bmatrix}
    s_{0c} \\
    s_{1c} \\
    s_{2c} \\
    s_{3c}
\end{bmatrix}
= \begin{bmatrix}
    0e & 09 & 0d & 0b \\
    0b & 0e & 09 & 0d \\
    0d & 0b & 0e & 09 \\
    09 & 0d & 0b & 0e
\end{bmatrix}
\begin{bmatrix}
    s_{0c} \\
    s_{1c} \\
    s_{2c} \\
    s_{3c}
\end{bmatrix}
\quad \text{for } 0 \leq c < Nc
\]  

(5.4.1)

The pseudo code for this transformation is as follows, where the function \( \text{FFmul}(x, y) \) returns the product of two finite field elements \( x \) and \( y \).

\[
\text{InvMixColumns}(\text{byte } \text{block}[4, Nc], Nc)
\begin{align*}
\text{begin} \\
\text{byte } t[4] \\
\text{for } c = 0 \text{ step 1 to } Nc - 1 \\
\quad \text{for } r = 0 \text{ step 1 to 3} \\
\quad \quad t[r] = \text{block}[r, c] \\
\quad \text{end for} \\
\quad \text{for } r = 0 \text{ step 1 to 3} \\
\quad \quad \text{block}[r, c] = \\
\quad \quad \quad \text{FFmul}(0x0e, t[r]) \text{ xor} \\
\quad \quad \quad \text{FFmul}(0x0b, t[(r + 1) \mod 4]) \text{ xor} \\
\quad \quad \quad \text{FFmul}(0x0d, t[(r + 2) \mod 4]) \text{ xor} \\
\quad \quad \quad \text{FFmul}(0x09, t[(r + 3) \mod 4]) \\
\quad \text{end for} \\
\text{end for} \\
\text{end}
\end{align*}
\]

5.5 The Equivalent Inverse Cipher

The inverse cipher uses the same key schedule as the forward cipher (in reverse) but its form is different. However a series of transformations can be applied to transform the inverse cipher to match the form of the forward cipher. This is possible because the order of some operations in the inverse cipher can be changed without changing the final result.

For example the order of the \text{SubBytes} and \text{ShiftRows} transformations does not matter because \text{SubBytes} changes the value of bytes without changing their positions whereas \text{ShiftRows} does the exact opposite. Moreover, the order of the \text{XorRoundKey} and \text{InvMixColumns} operations can be inverted to put the forward and inverse ciphers in the same form provided that an adjustment is made to the key schedule. The order of round key addition and column mixing can be changed because the column mixing operation is linear with respect to the column input so that:

\[
\text{InvMixColumns}(\text{state xor } k) = \text{InvMixColumns}(\text{state}) \text{ xor } \text{InvMixColumns}(k)
\]

where \( k \) represents a round key in the form of a state array. Hence, provided that an inverse column mixing operation is performed on appropriate words (columns) of the decryption key schedule, the order of these transformations can be reversed during decryption. Note, however, that this operation is not be performed on the first and last round keys (the first and last \( Nc \) words of the key schedule) since these do not operate in association with the column-mixing step.

The importance of this transformation is that the structure of the forward cipher allows the round function to be expressed in an efficient form for implementation. By transforming the inverse cipher into the same sequence of operations as the cipher itself, it can be implemented in the same way, thereby achieving this efficiency.
In this modified form the inverse cipher is as follows (with the modified decryption key
schedule in the word array $d_{k[Nn+1,Nc]}$).

```plaintext
EqInvCipher(byte in[4*Nc], byte out[4*Nc], word $d_{k[Nn+1,Nc]}$, Nc, Nn) begin
    byte state[4,Nc]
    state = in
    XorRoundKey(state, $d_{k[Nn,-]}$, Nc)    // $d_{k[Nn*Nc..(Nn+1)*Nc-1]}
    for round = Nn - 1 step -1 to 1
        InvSubBytes(state, Nc)
        InvShiftRows(state, Nc)
        InvMixColumns(state, Nc)
        XorRoundKey(state, $d_{k[round,-]}$, Nc) // $d_{k[round*Nc..(round+1)*Nc-1]}
    end for
    InvSubBytes(state, Nc)
    InvShiftRows(state, Nc)
    XorRoundKey(state, $d_{k[0,-]}$, Nc)   // $d_{k[0..Nc-1]}
    out = state
end
```

where the following pseudo code is added to the end of the key expansion step (this can
be made more efficient if encryption and decryption are not required simultaneously).

```plaintext
for round = 0 step 1 to Nn
    $d_{k[i,-]}$ = $k[i,-]$    // copy Nc words at a time
end for
for round = 1 step 1 to Nn - 1
    InvMixColumns($d_{k[round,-]}$) // note implicit change of type
end for
```

Note that, since $InvMixColumns$ operates on a two-dimensional array of bytes while the
round keys are held in an array of words, the call to $InvMixColumns$ in this pseudo code
sequence involves a change of type. This requires care with byte order conventions.

6. Implementation Issues

6.1 Implicit Assumptions

While hardware implementations of Rijndael can treat the input, output and cipher key
inputs as bit sequences, software implementations will almost always to treat these
entities as arrays of 8-bit bytes. Equally, while a hardware implementation will have to
include a description of how Rijndael inputs and outputs are interfaced, a software
implementation will often operate in an environment where Rijndael’s two key
enumerations – the enumeration of bits within 8-bit bytes and the enumeration of bytes
within arrays – are already defined.

Where the environment in which Rijndael is implemented provides both for 8-bit bytes as
addressable entities and for the enumeration of bits within bytes, it is reasonable to
assume that Rijndael inputs and outputs will comply with these conventions.

In consequence Rijndael implementations in software should either indicate that this
assumption is correct or alternatively undertake one of the following:

(a) convert inputs and outputs to (or from) these standard formats to those being used
internally;

(b) document the interface to ensure that users of the implementation know that the inputs
and outputs are in non-standard formats.
6.2 Bit Enumerations

In processing bytes to undertake finite field multiplication it is useful to define a function to multiply by $x$, an operation that involves shifting the value of a byte by one and then performing a conditional XOR operation. If by convention bit 0 is the ‘lowest’ bit in a byte (i.e. it represents a numeric value of 1) then multiplying by $x$ will correspond to a left shift. This is the most likely situation but it is not unknown for bit 0 to be designated as the ‘highest’ bit in a byte, the bit that represents a numeric value of 128 in decimal, in which case multiplication by $x$ will correspond to a right shift. When this applies, all byte values will also have their bits reversed so that $\{01100011\}$ or $\{63\}$, which in former convention would be associated with a numeric value of $0x63$ in hexadecimal, will instead be associated with a numeric value of $0xc6$. For this reason the terms ‘left’ and ‘right’ when referring to shifts have been avoided in this specification by using the terms ‘up’ and ‘down’ to refer to operations in which bytes at an index position move to higher or lower index positions respectively.

6.3 Bytes Within Words

A number of Rijndael operations involve the manipulation of the four 8-bit bytes within a 32-bit word, one such operation being the cyclic shift (rotation) of these four bytes into new positions. Whether the operation of moving bytes to higher array index positions corresponds to a cyclic left or a cyclic right shift for a 32-bit word will depend on how the bytes are organised within words.

On some (‘little-endian’) processors bytes are numbered upwards from the ‘low’ end of 32-bits words and this means that a cyclic shift of bytes to higher array index positions will correspond to a cyclic left shift. But on other (‘big-endian’) processors bytes are numbered upwards starting at the ‘high’ end of a word so that a cyclic shift to higher index positions corresponds to a cyclic right shift.

In consequence care is needed in implementing Rijndael to ensure that the right directions of shifts and rotates are employed for the processor or processors for which an implementation is being designed.

In general these issues can be tackled either by the conversion of input and output values before use or by ensuring that the conventions employed for implementation are those of the architecture on which the cipher will operate.

7. Implementation Techniques

In the pseudo code in this section the following symbols will be used:

- $\&$ bits in result are the AND of the corresponding bits in the two operands
- $|$ bits in result are the OR of the corresponding bits in the two operands
- $^\wedge$ bits in result are the XOR of the corresponding bits in the two operands
- $\gg$ right shift of left operand by amount given by right operand
- $\ll$ left shift of left operand by amount given by right operand
- $<>$ not equal
- $0x...$ hexadecimal value

7.1 Finite Field Multiplication

The basic technique for finite field multiplication is explained in Section 2.4 and is implemented as follows:
byte FFmul(const byte a, const byte b)
begin
    byte aa = a, bb = b, r = 0, t
    while (aa <> 0)
        if ((aa & 1) <> 0)
            r = r ^ bb
        endif
        t = bb & 0x80
        bb = bb << 1
        if (t <> 0)
            bb = bb ^ 0x1b // top bit of field polynomial (0x11b) is not
            // needed here since bb is an 8 bit value
        endif
        aa = aa >> 1
    endwhile
    return r
end

But this approach can be quite slow compared with table lookup using the techniques described in Section 2.5. With a 256-byte arrays from tables 2 and 3 we obtain:

byte FFlog[256] // array from table 2
byte FFpow[256] // array from table 3

byte FFmul(const byte a, const byte b)
begin
    if ((a <> 0) and (b <> 0))
        word t = FFlog[a] + FFlog[b]
        if(t >= 255)
            t = t – 255
        endif
        return FFpow[t]
    else
        return 0
    endif
end

This can be speeded up by doubling the length of the FFpow[] array and setting the values for elements 255 to 509 to the same values as elements 0 to 254 respectively so that FFmul() can be coded as:

byte FFmul(const byte a, const byte b)
begin
    if ((a <> 0) and (b <> 0))
        return FFpow[FFlog[a] + FFlog[b]]
    else
        return 0
    endif
end

In practice many compilers will allow these functions to be specified as inline code and this makes finite field multiplication very efficient.

7.2 Column Mixing

Provided that the state array is arranged appropriately in memory, each of the columns will be a single 32-bit word. If the bytes in such a word are c[0] to c[3] then the mixing operation is:

\[
\begin{align*}
c[0]' &= {02} \cdot c[0] \oplus {03} \cdot c[1] \oplus {02} \cdot c[2] \oplus {03} \cdot c[3] \\
c[1]' &= {02} \cdot c[1] \oplus {03} \cdot c[2] \oplus {02} \cdot c[3] \oplus {03} \cdot c[0] \\
c[2]' &= {02} \cdot c[2] \oplus {03} \cdot c[3] \oplus {02} \cdot c[0] \oplus {03} \cdot c[1] \\
c[3]' &= {02} \cdot c[3] \oplus {03} \cdot c[0] \oplus {02} \cdot c[1] \oplus {03} \cdot c[2]
\end{align*}
\]

(7.2.1)

where the bytes are updated with the values on the left at the end of this sequence. But since \{03\} \cdot c[0] = \{02\} \cdot c[0] \oplus c[0], this can also be written as:
\[c[0]' = c[0] \oplus t \oplus \{02\} \cdot (c[0] \oplus c[1])\]
\[c[1]' = c[1] \oplus t \oplus \{02\} \cdot (c[1] \oplus c[2])\]
\[c[2]' = c[2] \oplus t \oplus \{02\} \cdot (c[2] \oplus c[3])\]
\[c[3]' = c[3] \oplus t \oplus \{02\} \cdot (c[3] \oplus c[0])\]

where \(t = c[0] \oplus c[1] \oplus c[2] \oplus c[3]\). When the need for temporary storage is taken into account, this code sequence becomes:
\[
t = c[0] \oplus c[1] \oplus c[2] \oplus c[3]
\]
\[
u = c[0] \oplus t \oplus \text{FFmul}(0x02, c[0] \oplus c[1])
\]
\[
c[1] = c[1] \oplus t \oplus \text{FFmul}(0x02, c[1] \oplus c[2])
\]
\[
c[2] = c[2] \oplus t \oplus \text{FFmul}(0x02, c[2] \oplus c[3])
\]
\[
c[3] = c[3] \oplus t \oplus \text{FFmul}(0x02, c[3] \oplus c[0])
\]
\[
c[0] = u
\]

Moreover, multiplication by the element \{02\} is just a shift followed by a conditional exclusive-or operation.

Although this formulation is quite efficient on 8-bit processors, the operations can be speeded up considerably on processors with 32 bit words provided that there are operations that can cyclicly rotate the bytes within such words. The functions required are as follows:
\[
\text{rot1}(w) \quad \text{moves the bytes in positions 0, 1 and 2 in the word } w \text{ to positions 1, 2 and 3 respectively and moves the byte in position 3 to position 0.}
\]
\[
\text{rot2}(w) \quad \text{moves the bytes in positions 0, 1, 2 and 3 in } w \text{ to positions 2, 3, 0 and 1 respectively (or exchanges byte 0 with byte 2 and byte 1 with byte 3).}
\]
\[
\text{rot3}(w) \quad \text{moves the bytes in positions 1, 2 and 3 in } w \text{ to positions 0, 1 and 2 respectively and moves the byte in position 0 to position 3.}
\]

Using these operations on each word \(w\) of the state allows the above code sequence on individual bytes to be rewritten as one operation on each word (column) as a whole:
\[
w = \text{rot3}(w) \oplus \text{rot2}(w) \oplus \text{rot1}(w) \oplus \text{FFmulX}(w \oplus \text{rot3}(w))
\]

where the function \(\text{FFmulX}(w)\) performs a finite field multiplication of each of the four bytes in the word \(w\) by \{02\}. This itself can be coded to operate in parallel on the four bytes in the word as follows:
\[
\text{word FFmulX(const word w) begin}
    \text{word } t = w \& 0x80808080
    \text{return } ((w \oplus t) \ll 1) \oplus ((t >> 3) \mid (t >> 4) \mid (t >> 6) \mid (t >> 7))\end
\]

Here the word \(t\) extracts the highest bits from each byte within \(w\), while the term \(w\oplus t\) extracts the lower 7 bits. The four individual bytes within the latter can then be multiplied by \{02\} in parallel using a single 32-bit left shift without creating overflows from one byte to the next. The \((t >> 3) \mid (t >> 4) \mid (t >> 6) \mid (t >> 7)\) construction leaves zero bytes within \(t\) unchanged but changes the bytes whose top bits are set to 0x1b. There are several alternative ways of performing this step including, for example \(((u - (u >> 7)) \& 0x1b1b1b1b)\) or \(((u >> 7) \times 0x0000001b)\), the most efficient depending on the characteristics of the processor instruction set available for its implementation. Finally, when this value is XOR’ed into the result the effect is that required – namely, the modular polynomial is added to all bytes in which the top bits were originally set.

Similar techniques can be used to speed up the inverse column mixing operation.
7.3 Implementation Using Tables

Rijndael can be implemented very efficiently on processors with 32-bit words by transforming it in the following way.

Considering a single column (word) of the state and applying the SubBytes, ShiftRows, MixColumns and XorRoundKey transformations in turn gives:

\[
\begin{bmatrix}
    s'_{0,c} \\
    s'_{1,c} \\
    s'_{2,c} \\
    s'_{3,c}
\end{bmatrix} =
\begin{bmatrix}
    S[s_{0,c}] \\
    S[s_{1,c}] \\
    S[s_{2,c}] \\
    S[s_{3,c}]
\end{bmatrix}
\]

(7.3.1)

after SubBytes:

\[
\begin{bmatrix}
    s_{0,c} \\
    s_{1,c} \\
    s_{2,c} \\
    s_{3,c}
\end{bmatrix} =
\begin{bmatrix}
    S[s_{0,c}] \\
    S[s_{1,c}] \\
    S[s_{2,c}] \\
    S[s_{3,c}]
\end{bmatrix}
\]

(7.3.2)

after ShiftRows:

\[
\begin{bmatrix}
    s_{0,c} \\
    s_{1,c} \\
    s_{2,c} \\
    s_{3,c}
\end{bmatrix} =
\begin{bmatrix}
    S[s_{0,c}] \\
    S[s_{1,c}] \\
    S[s_{2,c}] \\
    S[s_{3,c}]
\end{bmatrix}
\]

(7.3.3)

after MixColumns:

\[
\begin{bmatrix}
    s_{0,c} \\
    s_{1,c} \\
    s_{2,c} \\
    s_{3,c}
\end{bmatrix} =
\begin{bmatrix}
    S[s_{0,c}] \\
    S[s_{1,c}] \\
    S[s_{2,c}] \\
    S[s_{3,c}]
\end{bmatrix} \oplus
\begin{bmatrix}
    k_{0,c} \\
    k_{1,c} \\
    k_{2,c} \\
    k_{3,c}
\end{bmatrix}
\]

(7.3.4)

after XorRoundKey:

where the shorthand notation \( c(r) = [c + h(r, Nc)] \mod Nc \), with \( c(0) = c \), has been used in the column index \( c \).

Treating this as one complex transformation (i.e. with a single prime), it can be written in column vector form as:

\[
\begin{bmatrix}
    s'_{0,c} \\
    s'_{1,c} \\
    s'_{2,c} \\
    s'_{3,c}
\end{bmatrix} =
\begin{bmatrix}
    02 & 03 & 01 & 01 \\
    01 & 02 & 03 & 01 \\
    03 & 01 & 01 & 02 \\
    03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
    S[s_{0,c}] \\
    S[s_{1,c}] \\
    S[s_{2,c}] \\
    S[s_{3,c}]
\end{bmatrix}
\]

(7.3.5)

And if four tables each of 256 32-bit words are defined (for \( 0 \leq x < 256 \)) as follows:

\[
\begin{align*}
T_0[x] &= \begin{bmatrix} 02 \cdot S[x] \\ S[x] \end{bmatrix}, & T_1[x] &= \begin{bmatrix} 03 \cdot S[x] \\ S[x] \end{bmatrix}, & T_2[x] &= \begin{bmatrix} 02 \cdot S[x] \\ S[x] \end{bmatrix}, & T_3[x] &= \begin{bmatrix} 03 \cdot S[x] \\ S[x] \end{bmatrix}, \\
03 \cdot S[x] &= \begin{bmatrix} S[x] \\ S[x] \end{bmatrix}, & 02 \cdot S[x] &= \begin{bmatrix} S[x] \\ S[x] \end{bmatrix}, & 03 \cdot S[x] &= \begin{bmatrix} S[x] \\ S[x] \end{bmatrix}
\end{align*}
\]

(7.3.6)

equation (6.3.5) can then be expressed in the form:

\[
\begin{bmatrix}
    s'_{0,c} \\
    s'_{1,c} \\
    s'_{2,c} \\
    s'_{3,c}
\end{bmatrix} = T_0[S_{0,c}(0)] \oplus T_1[S_{1,c}(1)] \oplus T_2[S_{2,c}(2)] \oplus T_3[S_{3,c}(3)] \oplus k_{round,c}
\]

(7.3.7)

where \( c(r) = [c + h(r, Nc)] \mod Nc \), \( c(0) = c \) and \( k_{round,c} \) is word \( c \) of round key \( round \).
This shows that each column in the output state can be computed using four XOR instructions involving a word from the key schedule and four words from tables that are indexed using four bytes from the input state.

Equation (6.3.7) applies to all but the last round because the latter is different in that the MixColumns step is not present. This means that different tables are required for the last round as follows:

\[
U_0[x] = \begin{bmatrix} S[x] \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad U_1[x] = \begin{bmatrix} 0 \\ S[x] \\ 0 \\ 0 \end{bmatrix}, \quad U_2[x] = \begin{bmatrix} 0 \\ 0 \\ S[x] \\ 0 \end{bmatrix}, \quad U_3[x] = \begin{bmatrix} 0 \\ 0 \\ 0 \\ S[x] \end{bmatrix}
\]  

(7.3.8)

These tables can be implemented directly or can be computed either from the S-Box table or by masking the appropriate tables for normal rounds.

The tables for the main rounds amount to 4 kbytes of table space and this is doubled if the last round tables are also implemented. However, it is worth noting that these tables are closely related since \( T_i(x) = \text{rot1}(T_{i-1}(x)) \), and this means that the table space can be reduced by a factor of four at the expense of three additional rotations in the calculation of each column of the state.

This implementation technique can also be used for the equivalent inverse cipher since it has the same form as the forward cipher. This requires another set of tables since the inverse S-Boxes have to be used in the above transformations. The byte indexing for the table values is also different for the inverse cipher – \( c(r) = [c - h(r, Nc) + Nc] \mod Nc \).

8. Acknowledgements

This specification was originally written as an input to the AES FIPS development process but has been developed further since then as a result of comments received on the original version. I would like to acknowledge and thank Joan Daemen and Vincent Rijmen for many significant inputs that they made during its development. I would also like to thank both Jim Foti and Elaine Barker of NIST for their many helpful comments and suggestions, many of which are embodied both here and in the FIPS. My thanks also go to Paulo Barreto for his cooperation in publishing the original development test vectors and to Lawrence Bassham of NIST for independently checking their correctness. In respect of version 3.4 of this document, my thanks also go to Bryan Olson for a mapping idea used in section 1.2 and to Doug Gwynn and David Hopwood for prompting me to add a byte oriented external interface definition.

9. References


10. Errors

This specification has been produced from the base document referenced in section 9 above. It has no formal status but the author would be grateful if any errors found in it could be reported to him at brg@gladman.uk.net.

Software implementations of Rijndael by the author (in C/C++) are available at:

http://fp.gladman.plus.com/cryptography_technology/rijndael/
11. An Example of Cipher Operation

The following diagram shows the hexadecimal values in the state array as the cipher progresses for a cipher input length ($N_c$) of 4 and a cipher key length ($N_k$) of 4. The notation for the following inputs is given at the start of Section 12.

Input = 3243f6a8885a308d313198a2e0370734 (pi * 2^124)
Key   = 2b7e151628aed2a6abf7158809cf4f3c (e * 2^124)

<table>
<thead>
<tr>
<th>round</th>
<th>start of round</th>
<th>subbytes</th>
<th>shiftrows</th>
<th>mixcolumns</th>
<th>round key value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32 88 31 e0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>43 5a 31 77</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6f 30 98 07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a8 8d a2 34</td>
<td>19 0a 9a e9</td>
<td>d4 0e b8 1e</td>
<td>d4 0e b8 1e</td>
<td>2b 28 ab 09</td>
</tr>
<tr>
<td>4</td>
<td>1d 86 6b 02</td>
<td>49 45 7f f7</td>
<td>49 45 7f f7</td>
<td>58 1b db 1b</td>
<td>2f 7a 59 73</td>
</tr>
<tr>
<td>5</td>
<td>9c 9f 5b 6a</td>
<td>7f 35 ea 50</td>
<td>d2 96 87 53</td>
<td>87 53 d2 96</td>
<td>ea 5a ca 60</td>
</tr>
<tr>
<td>6</td>
<td>2e 35 43 49</td>
<td>3f 71 1a 38</td>
<td>3d 89 1f 1a</td>
<td>1f 0a af 6f</td>
<td>1f 8a 19 0f</td>
</tr>
<tr>
<td>7</td>
<td>ab 61 82 68</td>
<td>ac ef 13 45</td>
<td>sc ef 13 45</td>
<td>75 20 53 db</td>
<td>3d 47 1e 0d</td>
</tr>
<tr>
<td>8</td>
<td>81 dd 42 32</td>
<td>73 c1 b5 23</td>
<td>c1 b5 23 73</td>
<td>ec 0b c0 25</td>
<td>80 16 23 7a</td>
</tr>
<tr>
<td>9</td>
<td>51 ef 4a 46</td>
<td>cf 11 db 5a</td>
<td>db sa c1 f1</td>
<td>09 63 cf 80</td>
<td>47 6e e8 88</td>
</tr>
<tr>
<td>10</td>
<td>ee 0d 38 0f</td>
<td>76 df b5 b9</td>
<td>0f 7b d5 b5</td>
<td>93 37 cc de</td>
<td>7d 3e 41 1b</td>
</tr>
<tr>
<td>11</td>
<td>4b 67 4d d6</td>
<td>52 b5 e3 f6</td>
<td>52 b5 e3 f6</td>
<td>d6 60 f6 5e</td>
<td>ef a8 6b db</td>
</tr>
<tr>
<td>12</td>
<td>6c 1d e3 5f</td>
<td>50 a4 11 cf</td>
<td>a4 11 cf 50</td>
<td>d6 31 c0 b3</td>
<td>44 52 71 0b</td>
</tr>
<tr>
<td>13</td>
<td>46 9d 1b 58</td>
<td>2f 0e 88 6a</td>
<td>c8 6a 2f 5e</td>
<td>da 38 10 13</td>
<td>ab 50 25 ad</td>
</tr>
<tr>
<td>14</td>
<td>80 0d 50 01</td>
<td>90 ba 53 7c</td>
<td>7c 9b ba 53</td>
<td>ad 68 8e 60</td>
<td>f8 87 6c bc</td>
</tr>
<tr>
<td>15</td>
<td>e0 c8 d9 85</td>
<td>e1 e8 35 97</td>
<td>e1 e8 35 97</td>
<td>25 bd b6 4c</td>
<td>d4 7c ca 11</td>
</tr>
<tr>
<td>16</td>
<td>92 63 1b 68</td>
<td>4f 3b c8 6c</td>
<td>fc 8c 6c 4f</td>
<td>d1 11 3a 4c</td>
<td>d1 83 fc 29</td>
</tr>
<tr>
<td>17</td>
<td>78 63 5b a0</td>
<td>6b 2f 96 ae</td>
<td>96 ae 2d fb</td>
<td>a9 01 3c 2b</td>
<td>c6 9d be 15</td>
</tr>
<tr>
<td>18</td>
<td>60 0f 32 0c</td>
<td>7c 8d 1f 2f</td>
<td>7c 8d 1f 2f</td>
<td>da 38 10 13</td>
<td>ab 50 25 ad</td>
</tr>
<tr>
<td>19</td>
<td>0f 6d 3f f1</td>
<td>2f d7 01 79</td>
<td>2f d7 01 79</td>
<td>2f d7 01 79</td>
<td>2f d7 01 79</td>
</tr>
<tr>
<td>20</td>
<td>5a 1e 03 1a</td>
<td>be d4 0a da</td>
<td>be d4 0a da</td>
<td>00 0b 5f 6a</td>
<td>ea b5 31 7f</td>
</tr>
<tr>
<td>21</td>
<td>41 49 e0 8c</td>
<td>3e 1b 64 1e</td>
<td>3b 64 1e 8c</td>
<td>5c 0b 3f 4c</td>
<td>d3 93 4f 1e</td>
</tr>
<tr>
<td>22</td>
<td>42 1d 19 04</td>
<td>2c 86 d4 f2</td>
<td>2c 86 d4 f2</td>
<td>2c 86 d4 f2</td>
<td>2c 86 d4 f2</td>
</tr>
<tr>
<td>23</td>
<td>b1 1f 65 0c</td>
<td>c8 0d 4d fe</td>
<td>fe 0b 4d fe</td>
<td>df 0d 4d fe</td>
<td>df 0d 4d fe</td>
</tr>
<tr>
<td>24</td>
<td>ec 0d 65 85</td>
<td>87 12 4d 97</td>
<td>87 12 4d 97</td>
<td>4d 19 3a 4c</td>
<td>ac 1a 28 67</td>
</tr>
<tr>
<td>25</td>
<td>83 45 5d 96</td>
<td>ec 6e 4c 90</td>
<td>ec 6e 4c 90</td>
<td>6e 4c 90 6c</td>
<td>7f fa d1 0c</td>
</tr>
<tr>
<td>26</td>
<td>5c 33 98 b0</td>
<td>4c a3 46 e7</td>
<td>4c a3 46 e7</td>
<td>4c a3 46 e7</td>
<td>4c a3 46 e7</td>
</tr>
<tr>
<td>27</td>
<td>f0 2c 4d c5</td>
<td>8c 0d 95 a6</td>
<td>8c 0d 95 a6</td>
<td>a6 ed a5 0b</td>
<td>f3 e3 2f 0f</td>
</tr>
<tr>
<td>28</td>
<td>eb 59 8b 1b</td>
<td>e9 cb 3d af</td>
<td>e9 cb 3d af</td>
<td></td>
<td>d0 c9 e1 b6</td>
</tr>
<tr>
<td>29</td>
<td>40 2e a1 c3</td>
<td>09 31 32 2e</td>
<td>31 32 2e 09</td>
<td></td>
<td>14 ee 3f 63</td>
</tr>
<tr>
<td>30</td>
<td>2f 38 13 42</td>
<td>89 07 1d 2c</td>
<td>7d 0c 89 07</td>
<td></td>
<td>f9 23 8c 0c</td>
</tr>
<tr>
<td>31</td>
<td>1e 84 7d d2</td>
<td>72 5f 94 b5</td>
<td>72 5f 94 b5</td>
<td></td>
<td>a8 85 c8 a6</td>
</tr>
<tr>
<td>32</td>
<td>39 02 4c 19</td>
<td>2d dc 11 6a</td>
<td>2d dc 11 6a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>29 dc 11 6a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>84 09 85 0b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>1d fb 97 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dr. Brian Gladman, v3.6, 15th April 2003
12. Rijndael Development Test Vectors

All vectors are in hexadecimal notation with each pair of characters giving a byte value
where the left and right characters of each pair provide the bit pattern for the 4 bit group
containing the higher and lower numbered bits respectively using the format explained in
Section 1.2. The array index for all bytes (groups of two hexadecimal digits) within these
test vectors starts at zero on the left and increases from left to right.

Considered instead as bit sequences, with hexadecimal digits numbered from left to right
starting from 0, hexadecimal digit \(n\) gives the value of bits \(4n\) to \(4n+3\) in the sequence
using the 4-bit notation given in Section 1.2 except that lower numbered bits are now on
the left (this arises because bits in bit sequences and bits in bytes are mapped in reverse).

These test have been generated by Dr Brian Gladman using the
program aes_vec.cpp <brg@gladman.uk.net> 24th January 2001.

**LEGEND FOR ENCRYPT** (round number \(r = 0\) to 10, 12 or 14)
- **input**: cipher input
- **start**: state at start of round[\(r\)]
- **s_box**: state after s_box substitution
- **s_row**: state after shift row transformation
- **m_col**: state after mix column transformation
- **k_sch**: key schedule value for round[\(r\)]
- **output**: cipher output

**LEGEND FOR DECRYPT** (round number \(r = 0\) to 10, 12 or 14)
- **KEY SCHEDULE FOR KEY XOR FOLLOWED BY INVERSE MIX COLUMN**
  - **input**: inverse cipher input
  - **istart**: state at start of round[\(r\)]
  - **is_box**: state after inverse s_box substitution
  - **is_row**: state after inverse shift row transformation
  - **im_col**: state after inverse mix column transformation
  - **ik_sch**: key schedule value for round[\(r\)]
  - **ioutput**: cipher output

**LEGEND FOR DECRYPT (MOD)** (round number \(r = 0\) to 10, 12 or 14)
- **KEY SCHEDULE FOR INVERSE MIX COLUMN FOLLOWED BY KEY XOR**
  - **input**: inverse cipher input
  - **istart**: state at start of round[\(r\)]
  - **is_box**: state after inverse s_box substitution
  - **is_row**: state after inverse shift row transformation
  - **im_col**: state after inverse mix column transformation
  - **ik_sch**: key schedule value for round[\(r\)]
  - **ioutput**: cipher output

**PLAINTEXT**: 3243f6a8885a308d313198a2e0370734 (\(\pi \times 2^{124}\))
**KEY**: 2b7e151628aed2a6abf71758809cf4f3c (\(e\times 2^{124}\))

**ENCRYPT**

| R[0].input | 3243f6a8885a308d313198a2e0370734 |
| R[0].k_sch | 2b7e151628aed2a6abf71758809cf4f3c |
| R[1].start | 193de3bea0f4e22b9ac68d2aef9f84808 |
| R[1].s_box | d42711aee0bf98f1b8845de51e415230 |
| R[1].s_row | d4bf530e0b452ab8411f12e279e5 |
| R[1].m_col | 046861e5e0cb199a48f8d37a2806264c |
| R[1].k_sch | a0fae1788542cb123a33932a6c7605 |
| R[2].start | 49c7ff2609f352b6b5bea43026a5049 |
| R[2].s_box | 49ded28945db96f17f93817a7702533b |
| R[2].s_row | 49db87315f3953897f02d2f177de961a |
| R[2].m_col | 584dca11bd49a1b5acdb7eaa81b6bb0e5 |
| R[2].k_sch | f2c29f527a96b9435935807a7359f67f |
| R[3].start | aa8f50f361d0e3ef82d24ad26832469a |
| R[3].s_box | ac73cf7be111d1f3b56b545235ab8 |
| R[3].s_row | acc1d6b8efb55a7b1323cfd457311b5 |
| R[3].m_col | 75e0993200633353cc0f7c0b2500dc |
| R[3].k_sch | 3d80477d4716fe3e1e237e446d7a883b |
| R[4].start | 486c40ee6719d90d4de3b138dd56f58e7 |
| R[4].s_box | 52502f2885a45ed7e31c807f6cf6a94 |

Dr. Brian Gladman, v3.6, 15th April 2003
R[ 4].s_row  52a4c89485116a28e3c2f9d7f6505e07
R[ 4].m_col  0fd6daa9603138bf6f0c1016b5eb31301
R[ 4].s_box  ef4a45a18525b7f6b761253db0b0ad00
R[ 5].start  e0927fe8c86363c0d9b1355085b8be01
R[ 5].s_box  e14fd29be8fbbba35c89639376c7e0c
R[ 5].s_row  elf9670c8c8a9b356cd2ba974ff5b53
R[ 5].m_col  25d1a9adb11d16863a338e4c0cc0b0
R[ 5].k_sch  d4d1c6f87c839d87caf2b3c11f915bc
R[ 6].start  f1006f55c1924ce7c838b325db5d50c
R[ 6].s_box  a163a8fc784f29df1e08d3234d5035e0c
R[ 6].s_row  a14f32f7e803fc0d8a8f6c6329232
R[ 6].m_col  4b8686d6d62e4a89830393d4e673128d8
R[ 6].k_sch  6d88a37a110b3efddf9f8641aca093f0d
R[ 7].start  260e2e173d41bd8e86472a9f0dd82b25
R[ 7].s_box  f7ab31f02783a9ff9b3430d3545b35d3f
R[ 7].s_row  f783403f27433df09b531ff54aab9d3
R[ 7].m_col  1415b5bf46165ee27656d7342ad845
R[ 7].k_sch  4e54f70e5f5f9f34a64f2bea46d4c4f
R[ 8].start  5a4142bb11949dc1fca3e019657a8c040c
R[ 8].s_box  be832cc8d43b860oaed4dada64f2fe0c
R[ 8].s_row  be3b4f6e4e12f2c80a642cc0da88646
R[ 8].m_col  00512fd1b889ff5766dcd1a8b99ea
R[ 8].k_sch  ead27321b58dbdad2312bf5607f8d292f
R[ 9].start  ea835cf04453322d655db98a5d96b05c
R[ 9].s_box  87ec4a8cf26ed384c49659790e7a6
R[ 9].s_row  876e46a6f24ce78c4d904ad897e9c395
R[ 9].m_col  4737794ed40d4ea4a53703aa4694f2bc
R[ 9].k_sch  ac776f319fadc2128212d914575c00ef
R[10].start  eb40f2e1592e38848ba1137bc342d2
R[10].s_box  e9098792cb310753fd32794a2e2cb5
R[10].s_row  e9317db5cb322cb723d2e895fa097094
R[10].k_sch  d014f9a8c9ee2589e13f0cc86630ca6
R[10].output  3925841d02cc09fbd118597196a0b32

**DECRYPT**
R[ 0].iinput  3925841d02dc09fbdcc118597196a0b32
R[ 0].ik_sch  6d88a37a110b3efddf9f8641aca093f0d
R[ 1].start  e9317db5cb322c723d2e895fa097094
R[ 1].is_row  e9098792cb310753fd32794a2e2cb5
R[ 1].is_box  eb40f2e1592e38848ba1137bc342d2
R[ 1].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 2].start  eb40f2e1592e38848ba1137bc342d2
R[ 2].is_row  e9317db5cb322c723d2e895fa097094
R[ 2].is_box  6d88a37a110b3efddf9f8641aca093f0d
R[ 2].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 3].start  eb40f2e1592e38848ba1137bc342d2
R[ 3].is_row  e9317db5cb322c723d2e895fa097094
R[ 3].is_box  e9098792cb310753fd32794a2e2cb5
R[ 3].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 4].start  eb40f2e1592e38848ba1137bc342d2
R[ 4].is_row  e9317db5cb322c723d2e895fa097094
R[ 4].is_box  e9098792cb310753fd32794a2e2cb5
R[ 4].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 5].start  eb40f2e1592e38848ba1137bc342d2
R[ 5].is_row  e9317db5cb322c723d2e895fa097094
R[ 5].is_box  e9098792cb310753fd32794a2e2cb5
R[ 5].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 6].start  eb40f2e1592e38848ba1137bc342d2
R[ 6].is_row  e9317db5cb322c723d2e895fa097094
R[ 6].is_box  e9098792cb310753fd32794a2e2cb5
R[ 6].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 7].start  eb40f2e1592e38848ba1137bc342d2
R[ 7].is_row  e9317db5cb322c723d2e895fa097094
R[ 7].is_box  e9098792cb310753fd32794a2e2cb5
R[ 7].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 8].start  eb40f2e1592e38848ba1137bc342d2
R[ 8].is_row  e9317db5cb322c723d2e895fa097094
R[ 8].is_box  e9098792cb310753fd32794a2e2cb5
R[ 8].ik_sch  ac776f319fadc2128212d914575c00ef
R[ 9].start  eb40f2e1592e38848ba1137bc342d2
R[ 9].is_row  e9317db5cb322c723d2e895fa097094
R[ 9].is_box  e9098792cb310753fd32794a2e2cb5
R[ 9].ik_sch  ac776f319fadc2128212d914575c00ef
R[10].start  eb40f2e1592e38848ba1137bc342d2
R[10].is_row  e9317db5cb322c723d2e895fa097094
R[10].is_box  e9098792cb310753fd32794a2e2cb5
R[10].ik_sch  ac776f319fadc2128212d914575c00ef
R[10].output  3925841d02cc09fbd118597196a0b32

**Dr. Brian Gladman, v3.6, 15th April 2003**

**page 22**
R[ 7].i_add 75ec099320b63353c0cf7cba25d0dc
R[ 8].i.start acc1d6b8efb5a7b1323cfdfe457311b5
R[ 8].is_row ac73cf7beefc11df3b356b545235ab9
R[ 8].is_box aa8f5f0361dde3ef82d24ad26832469a
R[ 8].i.sch f2c295f27a96b9439538073a73596f7f
R[ 9].i.add 584da41f1b45aaacabe7e8a1b6b0e5
R[ 9].i.start 49db873b453953879f02d2f177de961a
R[ 9].is_row 49ded2845db69f17f39871a7702533b
R[ 9].is_box a49cf7ff2689f352b6b5b4a43026a5049
R[ 9].i.sch a0fafaef1788542cb123a39392a6c7605
R[ 9].i.a0 046681e5e0c1b99a48f5d37a2806264c
R[10].i.start d4bf5d300e4b5ae8841f127e98e5
R[10].is_row d42711ae0b9f8lbb45de5e145230
R[10].is_box 193ed3beb0f422b9ac68d2ae9f84080
R[10].i.sch 2bfe151628ed2a6aebf7158809cf4f3c
R[10].i.output 3243efa8885a308d313198a2e0370734

R[ 0].i.input 3925841d02dc09fbdcd1859716ead32
R[ 0].i.sch d014f39a8c9ee2589a1f30cc8b6630ca6
R[ 1].i.start e9317db5c32723d2e895fa9f090794
R[ 1].is_box eb2e13d259a1421e8b3cf2841b4038e7
R[ 1].is_row eb40f21e92e38848ba11371bc342d2
R[ 1].i.col 8b151ccce5150d72f9ac2481f1a30821
R[ 1].i.sch 0c7b5a6313e19aaebf039889064c6b4
R[ 2].i.start 876ee46af624784cd940a8d97eccc95
R[ 2].is_box ea4598c5045db0f06596c2d85833ad
R[ 2].is_row ea835cf0445332d65d49d8a58960bc5
R[ 2].im_col 614646a4cbb34255a944e00cf656593
R[ 2].i.sch df7d925a1f662b9d9a302626ed675324
R[ 3].i.start be3bd4fed4e1f28ca01642c0da8386d4
R[ 3].is_box 5a49190c19e004b1a38c421f7a146cc65
R[ 3].is_row 5a1412b11949dc1fa3e019637a8c040c
R[ 3].i.col e543367875cf1f3727fe30c21febe899
R[ 3].i.sch 12c07647c01ff22c7bc42d2f37555114a
R[ 4].i.start f783403f27433df09b531ff4aba9d3
R[ 4].is_box ea4598c5045db0f06596c2d85833ad
R[ 4].is_row ea835cf0445332d65d49d8a58960bc5
R[ 4].im_col 6a30a1fc2c23fc6e8a24b6b543aa38d
R[ 4].i.sch e1f9b67e8c8e9a9953c6d2ba974fbb53
R[ 5].i.start e063350c1b1beef89db87f0c85926350
R[ 5].is_box e0297f8c86363c0d9b1355085b8be01
R[ 5].i.col c22c8c8875791ec22f16e0795ed998c3e
R[ 6].i.sch 90884413d280b60a12a18241b087939
R[ 7].i.start 52a4c89485116a283c3f27f7f6505e07
R[ 7].is_box 481db1e76e3358e4d5f4ede0dd6693d8
R[ 8].i.start 486c44e6179d9dd4eb31386df58e7
R[ 8].is_box d0dec5f4adb9862d30261974ca1aeec
R[ 9].i.sch 7c1f13f74208c219c021ae480969bf7b
R[ 9].i.start acc1d6b8efb5a7b1323cfdfe457311b5
R[10].i.output aad4a961d2460382325feef688fe3d2
R[10].row aa8f5f0361dee3ef8d24ad26832469a
R[10].i.box 85ae82d07be8267fd2bbeae0b968729
R[10].i.sch cc705eb5e31d1e8e222965c19481133
R[10].i.a0 49db873b453953879f02df177de961a
R[10].i.box a49f3aa8f5b04f26b6a7f0b29c35483
R[10].i.sch a49c7ff2689f3526b5bea30426a5049
R[10].i.box ff8559712d68a047f4ace554e6587
R[10].i.sch 2b3708af7262d405bc3e0dbf4b616762
R[10].i.start d4bf5d30e0b52aeb8411f11e2799e5
R[10].is_box 19f48d08a0c648be9af832be93de22a
R[10].row 193de3beaf4e22b9ac68d2ae9f84808
R[10].i.sch 2bfe151628ed2a6aebf7158809cf4f3c
R[10].i.output 32433fa8885a308d313198a2e0370734
PLAINTEXT: 3243f6a8885a308d313198a2e0370734 (pi * 2^124)
KEY: 2b7e15162a26af7158809cf4f3c (e * 2^188)
762e7160f38b4da5
ENCRYPT
R[ 0].input  3243f6a8885a308d313198a2e0370734
R[ 0].k_sch  2b7e15162a26af7158809cf4f3c
R[ 1].start  193de3bea0f4e22b9ac6d2ae9f84808
R[ 1].s_box  d42711aeeef0bf8b45de51e145230
R[ 1].s_row  d4bf5d30e0b452aeb8411f11e27986e5
R[ 1].m_col  046681e5e0cb199a48f83d7a2a806264c
R[ 1].k_sch  762e7160f38b4da5179d131b3f33c1bd
R[ 2].start  724bf0851340543f5f6c0611735e7f1
R[ 2].s_box  40528c977d092075cf4dbaeef09694aa1
R[ 2].s_row  8026de2a2ed7a166b202c5bac2dbc8bc0
R[ 2].k_sch  762e7160f38b4da5179d131b3f33c1bd
R[ 3].start  14e20a1fb3d3ca6236272fd3a756f70
R[ 3].s_box  fa8615516dcca8c0059d67a57988066
R[ 3].s_row  fa8615516dcca8c0059d67a57988066
R[ 3].k_sch  3a83a524f4dbdb14878b33aebf817e2d
R[ 4].start  cb42fd92333f2843211fe843cbbca81a
R[ 4].s_box  2f11c39be82c15acc434292f2802b260
R[ 4].s_row  8a6ce3ed78a64f5db7862ead6a4a01
R[ 4].k_sch  2f11c39be82c15acc434292f2802b260
R[ 5].start  9499c6ebeb789412bb04097b7a97c025
R[ 5].s_box  22ee4b2856bccc29ea2f01a9588ba3f
R[ 5].s_row  22ee4b2856bccc29ea2f01a9588ba3f
R[ 5].k_sch  a0e2c727c5947af4d8a1c12f8f9b0
R[ 6].start  435ce25b977c16d9717c0ff77915e19
R[ 6].s_box  1a4a98ea88104761a31076686d4dd49
R[ 6].s_row  1a4a98ea88104761a31076686d4dd49
R[ 7].start  70b837b9aefc3b657e6722f875024aa
R[ 7].s_box  e3be25a42b95f5f8ba885eb6be17aa9
R[ 7].s_row  e3be25a42b95f5f8ba885eb6be17aa9
R[ 7].k_sch  94a2331ed2b7ed7d6c8384b9e1d1e6
R[ 8].start  516c9a5f6eb3654b562064b10f2e
R[ 8].s_box  51b0622e4b50e56a19a6556bc306
R[ 8].s_row  51b0622e4b50e56a19a6556bc306
R[ 9].start  9ee45fd543142876430d5c9c6cd37
R[ 9].s_box  522d88c5edab194e257c11fa6b08
R[ 9].s_row  522d88c5edab194e257c11fa6b08
R[ 10].start 00d8ca4a6556d242f3fcede8f9c822d7f30
R[ 10].s_box  00d8ca4a6556d242f3fcede8f9c822d7f30
R[ 10].s_row  00d8ca4a6556d242f3fcede8f9c822d7f30
R[ 11].start 197e3783af594192ed1f0f62d425b62
R[ 11].s_box  b2fc638be087d9159d106341de04f4b3
R[ 11].s_row  b2fc638be087d9159d106341de04f4b3
R[ 12].start 62132065740ef3ef14b42251223c97c
R[ 12].s_box  624042757b4c69f61e213e325
R[ 12].s_row  624042757b4c69f61e213e325
R[ 13].start 3a58b225cd5ee24a542298bced7f2b38c
R[ 13].s_box  79d0917abf194c4d0de52f6c6ed62f6d3
R[ 13].s_row  79d0917abf194c4d0de52f6c6ed62f6d3
R[ 14].start 4388b3262a7f68e8f4facc4a2a3ad4d55f
R[ 14].s_box  1a46df70268459bb444ba49e5808e6ecf
R[ 14].s_row  1a46df70268459bb444ba49e5808e6ecf
R[ 15].start 3e393601fe7324d28a3b5a838af9e5
R[ 15].output f9f9b29aefca384a250340d833b87ebc0
DECRYPT

Dr. Brian Gladman, v3.6, 15th April 2003
762e7160f38b4da56a784d9045190cfe

Dr. Brian Gladman, v3.6, 15th April 2003   page 26
R[ 1].is_row d3203dd11ac72d8b5ec47224953dfe5b
R[ 1].im_col 5c148cde9ad281737e8988b0c467d3a
R[ 1].istart 0841f147cdd229fa1b5af5dace7f6f0c
R[ 1].is_box bff82b16803e4c14442f2921aa8a0681
R[ 1].is_row bf8a291480f8062143e2eb81a2af4c16
R[ 1].im_col f3d1755fccc7f7a01d4c9d0737030ecc
R[ 1].ik_sch ac08bcdcf47d2ce0edfd00b41e25776
R[ 3].istart 5fd9e833bbadb40f024cd3b292679a
R[ 3].is_box 84e55a4149c09f7217a6889fa1740a37
R[ 3].is_row 8474487249e50a9f17c05a37a1a69f41
R[ 3].im_col 15d50dcd7f739c631903b29bc2e8b60
R[ 3].ik_sch 6a2da9b7704a7c7108e2b9c8cc52f40
R[ 4].istart 7ff8a470783ba4098e1e0090ea420
R[ 4].is_box 6be11dd0ea8fc072e2e0652936cd15d
R[ 4].is_row 6b3ce672ae11d52e28f1d5496e0cd0
R[ 4].im_col 463156e25faa20916ee417d421bd8e7
R[ 4].ik_sch b0724665875e7319822e12c1a25a7d
R[ 5].istart f64315047dfc7ade636096e09829a
R[ 5].is_box d6647cfd38ef3118610903599401137
R[ 5].is_row d6409182864113561ef7c37990031fd
R[ 5].im_col 0cc6e706a1446e65a528961ed70e027
R[ 5].ik_sch 79e580fb1a67d5c678a840ea42713db
R[ 6].istart 758bf6f1a2b32bd23dd08699b69575fcfc
R[ 6].is_box 3fc6ed62e0e3d2c93a4cece4da555
R[ 6].im_col 3fdae4320e55ec932d655e43acdb2
R[ 6].ik_sch 119ba2157575ba0e0cd3f226404685d09
R[ 7].istart bab84b33e8071a81a8f8c67bd5b97f9b36
R[ 7].is_box 7a321626bf5116b6a80c4df1b7f267f9
R[ 7].is_row 2932ff23f4704458f09eae4b5626236e
R[ 7].im_col 2926ea5f84323234bf070ff6e56e423
R[ 8].istart 8aa854fccccbf1a06eeef24e363821
R[ 8].is_box c6fffd5f275af443a5990455b655767b
R[ 9].im_col 025a3316f6ab7147a7170dc43336e26b
R[ 9].ik_sch 3e3f5d0ef56ba396014dc0119e40c61d
R[ 10].istart 3cc65e1803c0dd2da165acdd5afede477
R[ 10].im_col 72d05bb79e01aaa3af11951aba818560
R[ 10].ik_sch f8780f7852bf15b9a9ff6765f467bd4
R[ 11].istart 8a8a854fccccbf1a06eeef24e363821
R[ 11].is_box c6fffd5f275af443a5990455b655767b
R[ 11].is_row cff5043276f7655a5af7dbb69945f4
R[ 12].istart 96e9f6a6382553d62cf952cfcf85331
R[ 12].is_box 63da77112d88a684257d64f1232e37
R[ 13].im_col 393d66622da2e4f2f2887737215781a11
R[ 13].ik_sch 6dbc4534d51f75f154680b5180c1d0f
R[ 14].istart 7217e7019952c82220979b9c9e5ad302
R[ 14].is_box 72a8cca55cb549f40ef88f4263879f9d0600
R[ 15].ik_sch b51c1cb5eaa7c71c1fb4072e4598d91ac
R[ 15].is_box 8849558466ad7f590deb5e33aa840e9
R[ 16].im_col 88a8edbd6649405e9f5add5e93307d8f4
R[ 16].ik_sch b12fee6e02a546d06dfca93d40c0e54
R[ 17].ik_sch 7a8a854fccccbf1a06eeef24e363821
R[ 17].is_box 951c1cb5eaa7c71c1fb4072e4578d91ac
R[ 17].is_row c43bc5f3395d203d7b7e95c32091e
R[ 18].istart 3c65e1803c0dd2da165acdd5afede477
R[ 18].is_box 6dbc4534d51f75f154680b5180c1d0f
R[ 19].is_row 6dc08051d5bc1db5c5f1450f18467f34
R[ 19].im_col 7127e7019952c82220979b9c9e5ad302
R[ 19].ik_sch b51c1cb5eaa7c71c1fb4072e4598d91ac
R[ 20].istart 8849558466ad7f590deb5e33aa840e9
R[ 20].im_col 88a8edbd6649405e9f5add5e93307d8f4
R[ 20].ik_sch b12fee6e02a546d06dfca93d40c0e54
R[ 21].ik_sch cba3243bc9f1b84989f9c21b4c410858
R[ 21].is_box 5971a649122b9ad4e269a8445df8bf5e
R[ 22].istart 59f8a8d41271b4f4e22ba65e5d699a49
R[ 22].im_col 8687d60a6216e408c247e4509a8887f
R[ 22].ik_sch c68edd3d1fbd019f5187625a6d8a88f
R[ 23].istart 40090b377dcde59793c08c7535c22087
R[ 23].im_col 72409eb213802a8522f1f036d64854ea
R[ 24].istart 7248f0851340543f2f2809ee6ad1f21236
R[ 24].im_col 393b256561dc663873387ece7s057cfd6e
R[ 25].istart ed847858bd8343cd3f279df16f26b658b
R[ 25].im_col d4bf5d30e0b452a88411f1191e2b9e3
R[ 26].istart 19f48d08a0c648b9ef83e23b9ee22a
R[ 26].im_col 193e3eb0af4e22b9ac68d2aef8f4808
R[ 27].istart 2b7e151268aed2a6aaf7f58809cfc4f3c
R[ 27].im_col 3243f6a8885a308d313198a2e0370734

Dr. Brian Gladman, v3.6, 15th April 2003  page 29