

Rijndael.mws

```
[ restart:
```

■ Rijndael Functions

Implementation of the Rijndael cipher following the description in *AES Proposal: Rijndael* by Joan Daemen and Vincent Rijmen. This implementation is not in the least optimized with respect to performance but rather with a view on the readability in connection with the aforementioned description.

Author: Olaf Müller

Version: 1.0

Platform: Maple 6

```
[ > #F256 := GF( 2, 8, x^8 + x^4 + x^3 + x + 1 );
```

■ Nb, Nk and Nr - They can be changed using InitializeRijndael(Nb, Nk).

```
[ > Nb := 4:
  Nk := 4:
  Nr := max(Nb,Nk) + 6:
```

■ S-Box and InvSBox

```
[ > SBox := proc( a )
  local b;
  if a = F256[zero] then
    b := F256[zero];
  else
    b := F256[inverse]( a );
  fi;
  F256[ConvertIn]( Rem( F256[ConvertOut]( b ) * ( x^4 + x^3 + x^2 + x + 1 )
    + x^6 + x^5 + x + 1, x^8 + 1, x ) mod 2 );
end:
[ > InvSBox := proc( b )
  local a;
  a := F256[ConvertIn]( Rem( ( F256[ConvertOut]( b ) + x^6 + x^5 + x + 1 )
    * ( x^6 + x^3 + x ), x^8 + 1, x ) mod 2 );
  if a <> F256[zero] then
    a := F256[inverse]( a );
  fi;
  RETURN( a );
end:
```

■ ByteSub and InvByteSub

```
[ > ByteSub := proc( )
  global State;
  State := map( a -> SBox( a ), State );
end:
[ > InvByteSub := proc( )
  global State;
  State := map( b -> InvSBox( b ), State );
end:
```

■ ShiftRow and InvShiftRow

```
[ > ShiftOffsets[4] := [ 1, 2, 3 ]:
  ShiftOffsets[6] := [ 1, 2, 3 ]:
  ShiftOffsets[8] := [ 1, 3, 4 ]:
[ > ShiftRowBy := proc( i, C )
  global Nb, State;
  local NewRow, j;
  for j from 0 to Nb - 1 do
    NewRow[j] := State[ j + C mod Nb, i ];
  od;
  for j from 0 to Nb - 1 do
    State[j,i] := NewRow[j];
  od;
end:
[ > ShiftRow := proc( )
  global Nb;
  local i;
  for i from 1 to 3 do
    ShiftRowBy( i, ShiftOffsets[Nb][i] );
  od;
end:
[ > InvShiftRow := proc( )
```

```

global Nb;
local i;
  for i from 1 to 3 do
    ShiftRowBy( i, -ShiftOffsets[Nb][i] );
  od;
end:

```

MixColumn and InvMixColumn

```

> MixColumnPolynomial := F256[ConvertOut]( F256[input]( 3 ) ) * T^3
+ F256[ConvertOut]( F256[input]( 1 ) ) * T^2
+ F256[ConvertOut]( F256[input]( 1 ) ) * T
+ F256[ConvertOut]( F256[input]( 2 ) );

> MixColumn := proc( )
global State, Nb;
local j, a, b, i;
  for j from 0 to Nb - 1 do
    a := linalg[row]( matrix(State), j+1 );
    a := sum( 'F256[ConvertOut]( a[i] ) * T^(i-1)' , 'i' = 1..4 );
    b := Rem( a * MixColumnPolynomial, T^4 + 1, T ) mod 2;
    for i from 0 to 3 do
      State[j,i] := F256[ConvertIn]( coeff( b, T, i ) );
    od;
  od;
end:

> InvMixColumnPolynomial := F256[ConvertOut]( F256[input]( 11 ) ) * T^3
+ F256[ConvertOut]( F256[input]( 13 ) ) * T^2
+ F256[ConvertOut]( F256[input]( 9 ) ) * T
+ F256[ConvertOut]( F256[input]( 14 ) );

> InvMixColumn := proc( RoundKey )
local I_RoundKey, j, a, b, i;
  I_RoundKey := array( 0 .. Nb - 1 , 0 .. 3 );
  for j from 0 to Nb - 1 do
    a := linalg[row]( matrix(RoundKey), j+1 );
    a := sum( 'F256[ConvertOut]( a[i] ) * T^(i-1)' , 'i' = 1..4 );
    b := Rem( a * InvMixColumnPolynomial, T^4 + 1, T ) mod 2;
    for i from 0 to 3 do
      I_RoundKey[j,i] := F256[ConvertIn]( coeff( b, T, i ) );
    od;
  od;
  RETURN( I_RoundKey );
end:

```

AddRoundKey

```

> AddRoundKey := proc ( RoundKey )
global State;
local i, j;
  for j from 0 to Nb - 1 do
    for i from 0 to 3 do
      State[j,i] := F256[ '+' ]( State[j,i], RoundKey[j,i] );
    od;
  od;
end:

```

KeyExpansion and I_KeyExpansion

```

> RotByte := proc( word_in )
local word_out, i;
  word_out[4] := word_in[1];
  for i from 1 to 3 do word_out[i] := word_in[i+1]; od;
  RETURN( [ 'word_out[i]' $ 'i' = 1 .. 4 ] );
end:

> RC := array( 1 .. iquo( 8 * (14 + 1) - 1, 4 ) ):
RC[1] := F256[input]( 1 );
for i from 2 to iquo( 8 * (14 + 1) - 1, 4 ) do
  RC[i] := F256[ '*' ]( F256[input]( 2 ), RC[i-1] );
od:

> Rcon := proc( i )
[ RC[i], F256[input]( 0 ), F256[input]( 0 ), F256[input]( 0 ) ];
end:

> AddWords := proc( a, b )
local i, c;
  for i from 1 to 4 do
    c[i] := F256[ '+' ]( a[i], b[i] );
  od;
  [ 'c[i]' $ 'i' = 1 .. 4 ];
end:

> KeyExpansion := proc( Key )

```

```

global Nr, Nb;
local W, i, temp;
W := array( 0 .. Nb * (Nr + 1) - 1 );
for i from 0 to Nk - 1 do
  W[i] := [ Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3] ];
od;
for i from Nk to Nb * (Nr + 1) - 1 do
  temp := W[i-1];
  if i mod Nk = 0 then
    temp := AddWords( map( a -> SBox(a), RotByte( temp ) ),
                      Rcon( iquo( i, Nk ) ) );
  elif Nk > 6 and i mod Nk = 4 then
    temp := map( a -> SBox( a ), temp );
  fi;
  W[i] := AddWords( W[i-Nk], temp );
od;
RETURN( W );
end:
> RoundKey := proc( ExpandedKey, i )
global Nb;
RETURN( array( 0 .. Nb - 1, 0 .. 3,
              [ 'ExpandedKey[Nb * i + j]' $ 'j' = 0 .. Nb - 1 ] ) );
end:
> I_KeyExpansion := proc( Key )
global Nr, Nb;
local I_ExpandedKey, i, I_RoundKey, j;
I_ExpandedKey := KeyExpansion( Key );
for i from 1 to Nr - 1 do
  I_RoundKey := InvMixColumn( RoundKey( I_ExpandedKey, i ) );
  for j from Nb * i to Nb * (i + 1) - 1 do
    I_ExpandedKey[j] := convert( linalg[row]( matrix( I_RoundKey ),
                                                j mod Nb + 1), 'list' );
  od;
od;
RETURN( I_ExpandedKey );
end:

```

Round, FinalRound and I_Round, I_FinalRound

```

> Round := proc( RoundKey )
  ByteSub( );
  userinfo( 4, Round, 'State after ByteSub           :',
           StateToHexForUserinfo( State ) );
  ShiftRow( );
  userinfo( 4, Round, 'State after ShiftRow         :',
           StateToHexForUserinfo( State ) );
  MixColumn( );
  userinfo( 4, Round, 'State after MixColumn        :',
           StateToHexForUserinfo( State ) );
  userinfo( 4, Round, 'RoundKey                     :',
           StateToHexForUserinfo( RoundKey ) );
  AddRoundKey( RoundKey );
  userinfo( 4, Round, 'State after AddRoundkey      :',
           StateToHexForUserinfo( State ) );
end:
> FinalRound := proc( RoundKey )
  ByteSub( );
  userinfo( 4, FinalRound, 'State after ByteSub       :',
           StateToHexForUserinfo( State ) );
  ShiftRow( );
  userinfo( 4, FinalRound, 'State after ShiftRow      :',
           StateToHexForUserinfo( State ) );
  userinfo( 4, FinalRound, 'RoundKey                 :',
           StateToHexForUserinfo( RoundKey ) );
  AddRoundKey( RoundKey );
  userinfo( 4, FinalRound, 'State after AddRoundkey   :',
           StateToHexForUserinfo( State ) );
end:
> I_Round := proc( I_RoundKey )
global State;
  InvByteSub( );
  userinfo( 4, I_Round, 'State after InvByteSub       :',
           StateToHexForUserinfo( State ) );
  InvShiftRow( );
  userinfo( 4, I_Round, 'State after InvShiftRow     :',
           StateToHexForUserinfo( State ) );
  State := InvMixColumn( State );
  userinfo( 4, I_Round, 'State after InvMixColumn    :',

```

```

        StateToHexForUserinfo( State ) );
userinfo( 4, I_Round, 'I_RoundKey           :`,
        StateToHexForUserinfo( I_RoundKey ) );
AddRoundKey( I_RoundKey );
userinfo( 4, I_Round, 'State after AddRoundkey :`,
        StateToHexForUserinfo( State ) );
end:
> I_FinalRound := proc ( I_RoundKey )
  InvByteSub( );
userinfo( 4, I_FinalRound, 'State after InvByteSub :`,
        StateToHexForUserinfo( State ) );
  InvShiftRow( );
userinfo( 4, I_FinalRound, 'State after InvShiftRow :`,
        StateToHexForUserinfo( State ) );
userinfo( 4, I_FinalRound, 'I_RoundKey           :`,
        StateToHexForUserinfo( I_RoundKey ) );
AddRoundKey( I_RoundKey );
userinfo( 4, I_FinalRound, 'State after AddRoundkey :`,
        StateToHexForUserinfo( State ) );
end:

```

Rijndael and I_Rijndael and InitializeRijndael

```

> Rijndael := proc( CipherInput, CipherKey )
global State, Nr;
local ExpandedKey, i;
  State := copy( CipherInput );
  ExpandedKey := KeyExpansion( CipherKey );
userinfo( 4, Rijndael, 'initial State           :`,
        StateToHexForUserinfo( State ) );
userinfo( 4, Rijndael, 'I_RoundKey             :`,
        StateToHexForUserinfo( RoundKey( ExpandedKey, 0 ) ) );
AddRoundKey( RoundKey( ExpandedKey, 0 ) );
userinfo( 4, Rijndael, 'State after AddRoundkey :`,
        StateToHexForUserinfo( State ) );
  for i from 1 to Nr - 1 do
    Round( RoundKey( ExpandedKey, i ) );
  od;
  FinalRound( RoundKey( ExpandedKey, Nr ) );
  RETURN( State );
end:
> I_Rijndael := proc( CipherInput, CipherKey )
global State, Nr;
local I_ExpandedKey, i;
  State := copy( CipherInput );
  I_ExpandedKey := I_KeyExpansion( CipherKey );
userinfo( 4, I_Rijndael, 'initial State           :`,
        StateToHexForUserinfo( State ) );
userinfo( 4, I_Rijndael, 'I_RoundKey             :`,
        StateToHexForUserinfo( RoundKey( I_ExpandedKey, Nr ) ) );
AddRoundKey( RoundKey( I_ExpandedKey, Nr ) );
userinfo( 4, I_Rijndael, 'State after AddRoundkey :`,
        StateToHexForUserinfo( State ) );
  for i from Nr - 1 to 1 by -1 do
    I_Round( RoundKey( I_ExpandedKey, i ) );
  od;
  I_FinalRound( RoundKey( I_ExpandedKey, 0 ) );
  RETURN( State );
end:
> InitializeRijndael := proc( Nb_in, Nk_in )
global Nb, Nk, Nr;
  if not member( Nb_in, {4, 6, 8} ) then
    ERROR("Nb must be 4, 6 or 8.");
  elif not member( Nk_in, {4, 6, 8} ) then
    ERROR("Nk must be 4, 6 or 8.");
  fi;
  Nb := Nb_in;
  Nk := Nk_in;
  Nr := max( Nb, Nk ) + 6;
end:

```

Random generation

```

□ > GenerateRandomByte := rand( 0 .. 255 );

```

GenerateRandomKey

```

□ > GenerateRandomKey := proc( )

```

```

global Nk;
local Key, i;
  Key := array( 0 .. 4 * Nk - 1 );
  for i from 0 to 4 * Nk - 1 do
    Key[i] := F256[input]( GenerateRandomByte( ) );
  od;
RETURN( Key );
end:

```

GenerateRandomMessage

```

> GenerateRandomMessage := proc( )
global Nb;
local Key, i, j;
  Key := array( 0 .. Nb - 1 , 0 .. 3 );
  for j from 0 to Nb - 1 do
    for i from 0 to 3 do
      Key[j,i] := F256[input]( GenerateRandomByte( ) );
    od;
  od;
RETURN( Key );
end:

```

Modes of operation

StringToCipherInput

```

> BytesToCipherInput := proc( ByteList )
global Nb;
local CipherInput, i, j;
  CipherInput := array( 0 .. Nb - 1, 0 .. 3 );
  for j from 0 to Nb - 1 do
    for i from 0 to 3 do
      CipherInput[j,i] := F256[input]( ByteList[4*j+i+1] );
    od;
  od;
RETURN( CipherInput );
end:

> StringToCipherInput := proc( Text )
local ByteList, ByteListLength, NumberOfBlocks, CipherInput, i;
  ByteList := convert( Text, bytes );
  ByteListLength := nops( ByteList );
  NumberOfBlocks := ceil( ByteListLength / ( 4 * Nb ) );
  ByteList := [ op( ByteList ),
                0 $ NumberOfBlocks * 4 * Nb - ByteListLength ];
  for i from 1 to NumberOfBlocks do
    CipherInput[i] := BytesToCipherInput(
      [ 'ByteList[ 4 * Nb * (i-1) + 1 + j ]' $ 'j' = 0 .. 4 * Nb - 1 ] );
  od;
  convert( CipherInput, 'list' );
end:

> BytesToKey := proc( ByteList )
global Nk;
  array( 0 .. 4 * Nk - 1, map( x -> F256[input](x), ByteList[1..4*Nk] ) );
end:

> HexToKey := proc( HexKey )
global Nk;
local Key, i;
  Key := array( 0 .. 4 * Nk - 1 );
  for i from 0 to 4 * Nk - 1 do
    Key[i] := F256[input]( convert( HexKey[2*i+1..2*i+2],
                                   decimal, hex ) );
  od;
RETURN( Key );
end:

```

RijndaelECB

```

> RijndaelECB := proc( Text, CipherKey )
global State;
local CipherInput, NumberOfBlocks, CipherOutput, i;
  CipherInput := StringToCipherInput( Text );
  NumberOfBlocks := nops( CipherInput );
  for i from 1 to NumberOfBlocks do
    Rijndael( CipherInput[i], CipherKey );
    CipherOutput[i] := copy( State );
  od;
  convert( CipherOutput, 'list' );
end:

```

```

end:
> I_RijndaelECB := proc( CipherOutput, CipherKey )
global State;
local NumberOfBlocks, CipherInput, i;
  NumberOfBlocks := nops( CipherOutput );
  CipherInput := vector( NumberOfBlocks );
  for i from 1 to NumberOfBlocks do
    I_Rijndael( CipherOutput[i], CipherKey );
    CipherInput[i] := copy( State );
  od;
  cat( 'StateToText( CipherInput[i] )' $
      'i' = 1 .. NumberOfBlocks );
end:

```

RijndaelCBC

```

> AddStates := proc( State1, State2 )
global Nb;
local Result, j, i;
  Result := array( 0 .. Nb - 1, 0 .. 3 );
  for j from 0 to Nb - 1 do
    for i from 0 to 3 do
      Result[j,i] := F256['+']( State1[j,i], State2[j,i] );
    od;
  od;
  RETURN( Result );
end:
> RijndaelCBC := proc( Text, CipherKey )
global State;
local CipherInput, NumberOfBlocks, CipherOutput, i;
  CipherInput := StringToCipherInput( Text );
  NumberOfBlocks := nops( CipherInput );
  CipherOutput[0] := GenerateRandomMessage( );
  for i from 1 to NumberOfBlocks do
    Rijndael( AddStates( CipherOutput[i-1],
                        CipherInput[i] ), CipherKey );
    CipherOutput[i] := copy( State );
  od;
  convert( CipherOutput, 'list' );
end:
> I_RijndaelCBC := proc( CipherOutput, CipherKey )
global State;
local NumberOfBlocks, CipherInput, i;
  NumberOfBlocks := nops( CipherOutput );
  CipherInput[1] := CipherOutput[1];
  for i from 2 to NumberOfBlocks do
    I_Rijndael( CipherOutput[i], CipherKey );
    CipherInput[i] := AddStates( State, CipherOutput[i-1] );
  od;
  convert( CipherInput, 'list' );
  cat( 'StateToText( CipherInput[i] )' $
      'i' = 2 .. NumberOfBlocks );
end:
> RijndaelCBC0 := proc( Text, CipherKey )
global State;
local CipherInput, NumberOfBlocks, CipherOutput, i;
  CipherInput := StringToCipherInput( Text );
  NumberOfBlocks := nops( CipherInput );
  CipherOutput[1] := copy( Rijndael( CipherInput[1], CipherKey ) );
  for i from 2 to NumberOfBlocks do
    Rijndael( AddStates( CipherOutput[i-1],
                        CipherInput[i] ), CipherKey );
    CipherOutput[i] := copy( State );
  od;
  convert( CipherOutput, 'list' );
end:
> I_RijndaelCBC0 := proc( CipherOutput, CipherKey )
global State;
local NumberOfBlocks, CipherInput, i;
  NumberOfBlocks := nops( CipherOutput );
  CipherInput[1] := copy( I_Rijndael( CipherOutput[1], CipherKey ) );
  for i from 2 to NumberOfBlocks do
    I_Rijndael( CipherOutput[i], CipherKey );
    CipherInput[i] := AddStates( State, CipherOutput[i-1] );
  od;
  cat( 'StateToText( CipherInput[i] )' $
      'i' = 1 .. NumberOfBlocks );
end:

```

RijndaelCFB

```
> RijndaelCFB := proc( Text, CipherKey )
  global State;
  local CipherInput, NumberOfBlocks, CipherOutput, i;
  CipherInput := StringToCipherInput( Text );
  NumberOfBlocks := nops( CipherInput );
  CipherOutput[0] := GenerateRandomMessage( );
  for i from 1 to NumberOfBlocks do
    Rijndael( CipherOutput[i-1], CipherKey );
    CipherOutput[i] := AddStates( State, CipherInput[i] );
  od;
  convert( CipherOutput, 'list' );
end:
> I_RijndaelCFB := proc( CipherOutput, CipherKey )
  global State;
  local NumberOfBlocks, CipherInput, i;
  NumberOfBlocks := nops( CipherOutput );
  CipherInput[1] := CipherOutput[1];
  for i from 2 to NumberOfBlocks do
    Rijndael( CipherOutput[i-1], CipherKey );
    CipherInput[i] := AddStates( State, CipherOutput[i] );
  od;
  convert( CipherInput, 'list' );
  cat( 'StateToText( CipherInput[i] )' $
      'i' = 2 .. NumberOfBlocks );
end:
```

RijndaelOFB

```
> RijndaelOFB := proc( Text, CipherKey )
  global State;
  local CipherInput, NumberOfBlocks, CipherOutput, i;
  CipherInput := StringToCipherInput( Text );
  NumberOfBlocks := nops( CipherInput );
  CipherOutput[0] := GenerateRandomMessage( );
  State := CipherOutput[0];
  for i from 1 to NumberOfBlocks do
    Rijndael( State, CipherKey );
    CipherOutput[i] := AddStates( State, CipherInput[i] );
  od;
  convert( CipherOutput, 'list' );
end:
> I_RijndaelOFB := proc( CipherOutput, CipherKey )
  global State;
  local NumberOfBlocks, CipherInput, i;
  NumberOfBlocks := nops( CipherOutput );
  CipherInput[1] := CipherOutput[1];
  State := CipherOutput[1];
  for i from 2 to NumberOfBlocks do
    Rijndael( State, CipherKey );
    CipherInput[i] := AddStates( State, CipherOutput[i] );
  od;
  convert( CipherInput, 'list' );
  cat( 'StateToText( CipherInput[i] )' $
      'i' = 2 .. NumberOfBlocks );
end:
```

Output functions

StateTo...

```
> StateToText := proc( State )
  global Nb;
  local ByteList, i, j;
  ByteList := array(1..4*Nb);
  for j from 0 to Nb - 1 do
    for i from 0 to 3 do
      ByteList[4*j+i+1] := F256[output]( State[j,i] );
    od;
  od;
  RETURN( convert( convert( ByteList, 'list' ), bytes ) );
end:
> StateToHexForUserinfo := proc( State )
  convert( matrix(
    map( x -> cat( "0", convert( convert( F256[output]( x ),
      hex ), string ) ) [-2..-1], State ) ), 'listlist' );
end:
```

```

end:
> StateToHex := proc( State )
    linalg[transpose]( matrix(
        map( x -> convert( F256[output]( x ), hex), State ) ) );
end:
> StateToBytes := proc( State )
    linalg[transpose]( matrix(
        map( x -> F256[output]( x ), State ) ) );
end:
> StateToBits := proc( State )
    linalg[transpose]( matrix(
        map( x -> cat( "0000000", convert( F256[output]( x ), binary) )[-8..-1],
        State ) ) );
end:

```

KeyTo...

```

> KeyToHex := proc( Key )
    linalg[transpose]( matrix( Nk, 4,
        map( x -> convert( F256[output]( x ), hex), Key ) ) );
end:
> KeyToBytes := proc( Key )
    linalg[transpose]( matrix( Nk, 4,
        map( x -> F256[output]( x ), Key ) ) );
end:
> KeyToBits := proc( Key )
    linalg[transpose]( matrix( Nk, 4,
        map( x -> cat( "0000000", convert( F256[output]( x ), binary) )[-8..-1], Key
    ) ) );
end:

```

StateListToHex

```

> StateListToHex := proc( StateList )
    map( Block -> StateToHex( Block ), StateList );
end:

```

```

> save "Rijndael.m";

```

```

>

```