Input and Output Block Conventions for AES Encryption Algorithms

By Dr. B. R. Gladman

Introduction

1. Cryptographic algorithms operate in computer memory and transform memory blocks for inputs, outputs and keys. When these values need to be calculated consistently in many different processor environments it is necessary to have precise definitions of how they are to be interpreted and used. This has been a source of difficulty in the AES effort so far [1, 2] and has also been the subject of proposals for improvement in the first round comments [3].

2. When considered at the lowest level, the memory used in computers is made up of binary bits that are typically grouped into larger entities such as 8-bit bytes and 32-bit words, entities that will be referred to here as units. Although processors can be designed to interpret these units in many different ways, it is common practice to treat them as numbers in which the different bits represent powers of 2.

3. When integers are being represented it is common to number the bits within each unit from 0 upwards, with bit number 'n' being used to represent 2^n. In such 'integer' representations, the numeric value (significance) associated with bits increases with bit number from the 'bottom' or 'rightmost' end of a unit. But it is equally valid, though less common, to view units as representing fractions in which bit 'n' represents $2^{-(n+1)}$ so that bits decrease in numeric significance with increasing bit number. In such 'fractional' representations bits are numbered starting at the 'top' or 'leftmost' end of a unit.

4. This issue also arises when bytes or words are combined to represent larger numbers. When arrays of bytes (8-bit units) are used to represent integers it is common to use byte 'n' to represent multiples of 256^n so that higher numbered bytes are given higher numeric significance. Again, however, such byte arrays can be considered as fractional values where byte 'n' represents multiples of $256^{-(n+1)}$, with higher numbered bytes gaining lower numeric significance. These two styles of number representation have become known as 'big-endian' and 'little-endian' and processors are often designed to directly process numbers in only one of these two forms.

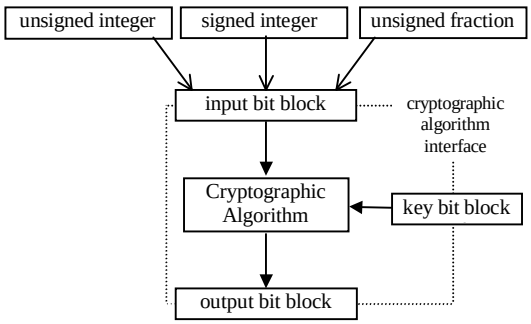
5. It is important to recognise that this issue only arises when we consider groups of bits in memory as higher level entities, for example, representations of numbers. When these bits are simply viewed as arrays of bits without semantic significance we do not have to worry about these matters. In practice, however most algorithms impose some form of order on bits and this means that we cannot avoid dealing with such issues.

6. In considering how this impacts on cryptographic algorithms, we have to decide whether our algorithms should act on the bits without considering their meaning or whether we want our algorithms to act on the higher level abstractions such as, for example, numbers.

7. In practice, however, sequences of bits in processor memory can be used to represent a huge variety of things: unsigned integers, signed integers (one's or two's complement), fractions, binary coded decimals, floating point numbers (in many different formats), sequences of characters (using different character sets) and any of an almost infinite variety of user defined abstractions.

8. Numbers are hence just one subset of an effectively infinite set of possible abstractions that can be imposed on these bits. And if algorithms act on these higher level abstractions rather than on the underlying bits, it is then necessary to specify, for each such abstraction, how the bits are being used to represent it. This would be an impossible task and this clearly suggests that this approach would not be a sensible one to adopt.

9. In consequence it seems most sensible to impose the minimum possible semantics on algorithm inputs and outputs. By doing this we allow users and designers to each attach their own meanings to these interface objects without having to share these with each other. Hence while a user might see the interface objects as numbers or character strings, an algorithm designer may choose to define them as elements in a finite field.



10. For these reasons it has become common practice to specify algorithm input, output and key blocks as groups of bits with no specified semantics as illustrated in the diagram above. This is reasonable for hardware but most processors are designed to use groups of bits rather than individual bits as their basic processing entities. Hence for software a bit level view is not the most convenient. Fortunately, however, almost all

processors offer ways of handling 8-bit units – bytes – and these are not much higher in abstract terms than the bit level definitions that have been employed in the past.

11. Hence by using bytes as convenient groups of 8-bits, and by using their order in memory, we can obtain a fairly universal, low level interface to cryptographic algorithms without imposing a particular semantic meaning either on external users or on algorithm designers.

The AES Algorithm Interface

12. The NIST specified AES algorithm interface is considered both within the source code of the NIST tool-kit and in the specification of test vectors. Within the source code the input, output and key blocks are specified as character arrays in a way that makes it clear that characters are simply convenient groups of 8 bits without semantic content. This is fully consistent with the earlier considerations.

13. In the discussion of the test vectors, however, the concepts of 'significance' and 'leftmost' are used in a way that has encouraged designers and implementers to treat these as numbers rather than blocks of bits. As a result some designers have implemented their algorithms with code to put these numeric values into big or little endian form whereas others have considered them as arrays of bytes that do not require such changes of order. The result has been some confusion and lack of interoperability between implementations when running on different processors [1,2].

14. In principle the internal semantics of algorithms are as rich as the external semantics discussed earlier. While algorithm designers must be free to decide how they want to interpret and manipulate the bits in input, output and key blocks, it is nevertheless important to have a standard notation for referencing these bits so that different implementations can each ensure that they are operating on these bits in an identical way.

Achieving a Low Level AES Interface

15. For purposes of reference it seems reasonable to take the traditional approach and hence to define all AES input, output and key blocks as sequences of bits in which the bits are numbered from 0..127, 0..191 or 0..255 as appropriate. At this primitive level bits have no defined significance, grouping or semantics.

16. But in practice, and especially within software implementations, bits within AES blocks will almost always be grouped into larger units each of which contains a number of bits which is a power of 2. Although not essential, it is desirable to have a simple mapping between such units and the underlying bits and the most simple and intuitive choice here is to group adjacently numbered bits into each unit and to define unit numbers (and bit numbers in units) using:

$$AesUnitNumber = \lfloor AesBitNumber / UnitSize \rfloor$$

$$AesBitNumberInUnit = AesBitNumber \bmod UnitSize$$

where $\lfloor x \rfloor$ is defined as "the largest integer not greater than x".

17. In practice the grouping of AES bits into 8-bit units – bytes – is an important 'canonical reference' since almost all processors offer both bytes and byte level addressing within memory. Hence a useful representation of AES blocks is as bytes numbered from 0..15, 0..23 or 0..31 as appropriate with numbering according to the conventions described earlier. This is fully consistent with the approach adopted in the NIST programming interface.

18. With these bit and byte numbering conventions at the interface it should be relatively easy for AES algorithm specifications and implementations to accurately describe mappings between AES bits, bytes (and larger units if appropriate) and those used within algorithms.

AES Test Vector Conventions

19. As discussed earlier, ambiguities in respect of byte order during the first AES round derive from uncertainties about whether the KAT test vectors are specified in terms of memory address order or by their numeric significance. In fact NIST has clearly used the latter, which was fairly certain to cause uncertainty because it is different in principle to the 'byte address order' used in the programming interface.

20. The author believes that using 'byte address ordering' for the API and 'numeric significance ordering' for the test vectors is not advisable. This subtle but important difference of approach has caused uncertainty about the true NIST intentions and hence a lack of interoperability in different implementations. Fortunately this is easy to avoid by using a consistent byte address order for both the API and the test vectors by defining test vector 'n' in a set of vectors to be one in which all bits are 0 except for bit number n, which is 1. This will be considered further later.

21. Issues of the order in which entities are printed in human readable input and output are also important. This applies, for example, to test vector printouts, where ambiguities can easily promote incorrect algorithm implementation. Since the NIST test vectors are printed in hexadecimal form the unit for output is 4-bits and printed AES blocks hence contain either 32, 48 or 64 hexadecimal digits.

22. For human readable output it is sensible to apply the following strict 'big-endian' approach:

- AES units, or sub-units within units, represented by printed symbols should have unit numbers that always increase from left to right and from top to bottom on pages;
- Where a printed symbol represents a group of bits (normally 0-9, a-f or A-F representing 4 bits) lower numbered AES bits have higher numeric significance;
- For sequences in time or within stored files (e.g. a sequence of AES encryption blocks), sequence numbers will increase with time or from the start of the file towards its end. The overall AES bit number within such sequences will be the sum of (1): the AES bit number within a block, and (2): the product of the block sequence number with the block length in bits

23. These big-endian conventions are natural for human readable output and allow test vectors to be divided into units and sub-units without changing the order of digit sequences 'on the page' as illustrated in the box opposite. They are consistent with the existing NIST test vectors.

The 128 bit block with the bits (increasing bit numbers from left to right):

000..063: 00000001001000110100010101100111100010011010101111001101111011111
064..127: 00000001001000110100010101100111100010011010101111001101111011111

is represented by:

0123456789ABCDEF0123456789ABCDEF

in hexadecimal and as

01234567 89ABCDEF 01234567 89ABCDEF

when viewed as four 32 bit numbers.

24. The only less common feature in this approach is the numbering of bits from the most to the least significant bit, the opposite of the usual processor bit numbering convention. In practice this is a small price to pay for the readability advantages that big-endian notation provides.

25. In any event, even though big-endian conventions are being used, it is bit numbers that are important, not the numeric significance of bits (the only place that numeric significance is employed is in the conversion of 4-bit units into printed hexadecimal digits).

26. An area of departure from the existing NIST definitions is that bit numbers (and any sequence numbers generally) start at zero rather than one. Without this change the mapping of bit numbers within units becomes more complex.

27. Once this bit numbering scheme (or any other) is adopted it will then be incumbent on all designers (in practice this means AES finalists) to set out fully within their specifications the correspondence between the entities they use and the bits within AES input, output and key blocks numbered in this way. Ideally these specifications will actually employ the standard numbering scheme but where this is not possible it will be essential to set out a precise relationship between the two sets of definitions.

28. At the same time implementers of algorithms should always make it clear how the processor bit and unit numbering relates to AES bit numbering.

29. These steps should help to reduce errors in algorithm implementation and interoperability.

Algorithm Internals

30. As already discussed, in addition to unit order at the algorithm interface it is also necessary to specify any impact that order has on the internal processing that an algorithm employs.

31. Some processor instructions that act on larger units, e.g. 32-bit words, are independent of internal order and these can often be used without the need to consider the placement of bits within larger units (provided that this is consistent). Examples of such instructions are those commonly known as AND, OR and XOR which act on the corresponding bits in each of the units involved.

32. As already discussed, when units are processed as numbers bit and unit order matter and need to be specified. This is also true of other instructions as well. For example, shift and rotate instructions are widely used in cryptographic algorithms and these generally require sub-unit order within larger units to be specified unless rotations with special count values are used.

33. In a bit level view of AES blocks, it is reasonable to view a right shift as one that moves a bit from a lower numbered position to a higher numbered one (this aligns with our normal ideas of increasing counts from left to right). In most processors, however, the opposite applies and a right shift moves bits to positions with lower numeric indices. It is hence easy to become confused about left and right in such situations.

34. In practice one concept for order is not sufficient for all purposes – we need different orders in different situations. In a little-endian environment, for example, we have one order for memory, another for numeric significance and a third for (big-endian) output.

35. Marcus Kuhn [3] has suggested a notation for handling unit order within specifications. This proposal has advantages but it would involve a considerable use of subscripts and superscripts where these are already in heavy use. While the

notation will be valuable when formality is required, the resulting proliferation of subscripts and superscripts seems unlikely to help in avoiding the errors that are typically made in handling such issues at a practical implementation level.

36. For this reason the author is more inclined to seek a consistent use of 'byte memory order' as described here as a 'reference order' and to seek specifications that clearly define a mapping from this canonical view of AES blocks onto the internal variables used in algorithms.

The Main AES Algorithm Tests

37. While the use of a bit/byte level interface removes many difficulties in handling endian issues, it does not eliminate them entirely because many AES algorithms treat their internal values as larger units such as 32-bit words. Because of this it is still possible to make mistakes in implementation by using the wrong order conventions.

38. Useful features that help in finding such errors are test vectors that have the same values when read in big and little endian order. Unfortunately the current variable text and variable key test vector sets do not have such 'endian-neutral' vectors and this reduces their value in finding such errors. The author hence believes that these test vector sets should be extended to include 'all zero' vectors.

39. In pseudo code the variable text and variable key tests would hence become:

```
// ECB Encryption Known Answer Test
for KeyLength = (128, 192, 256)
  Key{0..KeyLength-1} = 0
  print KeyLength, Key
  for TestNo = 0 to 128
    Ptext{0..127} = 0
    if TestNo > 0 set Ptext{TestNo-1} = 1
    encrypt(Ptext, Ctext, Key)
    print TestNo, Ptext, Ctext
  end for
end for

// ECB Decryption Known Answer Test
for KeyLength = (128, 192, 256)
  Ctext{0..127} = 0
  print KeyLength, Ctext
  for TestNo = 0 to KeyLength
    Key{0..KeyLength-1} = 0
    if TestNo > 0 set Key{TestNo-1} = 1
    decrypt(Ctext, Ptext, Key)
    output TestNo, Key, Ptext
  end for
end for
```

where the notation should be fairly self evident. The notation X{a..b} specifies an inclusive bit range (or a single bit) for a variable X. Where an operation is specified on such variables it is performed in parallel on the bits at the corresponding ***positions*** within each respective range.

40. The Electronic Code Book (ECB) Monte Carlo Tests are fairly easy to describe in the notation used here:

```
// ECB Encryption Monte Carlo Test
for KeyLength = (128, 192, 256)
  Key{0..KeyLength-1} = KEY0{0..KeyLength-1}
  Ptext{0..127}       = PLAINTEXT0{0..127}
  for TestNo = 0 to 399
    print TestNo
    print Key              // KEY
    print Ptext            // PLAINTEXT
    for Count = 0 to 4999
      encrypt(Ptext, Ctext{0..127}, Key)
      Ptext{0..127} = Ctext{0.127}
      encrypt(Ptext, Ctext{128..255}, Key)
      Ptext{0..127} = Ctext{128..255}
    end for
    print Ctext{128..255}  // CIPHERTEXT
    Ptext{0..127} = Ctext{128..255}
    Key{0..KeyLength-1} ^=
                Ctext{256-KeyLength..255}
  end for
end for
```

The decryption test is similar except that decryption is used and the roles of **`Ptext`** and **`Ctext`** are reversed.

41. The Cipher Block Chaining tests are somewhat more complex to describe as follows:

```
// CBC Monte Carlo (Encryption) Test
for KeyLength = (128, 192, 256)
  Key{0..KeyLength-1} = KEY0{0..KeyLength-1}
  Block{0..127}       = PLAINTEXT0{0..127}
  Block{128..255}     = IV0{0..127}
  for TestNo = 0 to 399
    print TestNo, Key
    print Block{128..255}  // IV
    print Block{0..127}    // PLAINTEXT
    for Count = 0 to 4999
      Block{0..127} ^= Block{128..255}
      encrypt(Block{0..127},
                Block{0..127}, Key)
      Block{128..255} ^= Block{0..127}
      encrypt(Block{128..255},
                Block{128..255}, Key)
    end for
    print Block{128..255}  // CIPHERTEXT
    Key{0..KeyLength-1} ^=
                Block{256-KeyLength..255}
  end for
end for

// CBC Monte Carlo (Decryption) Test
for KeyLength = (128, 192, 256)
  Key{0..KeyLength-1} = KEY0{0..KeyLength-1}
  Block{0..127}       = IV0{0..127}
  Block{128..255}     = CIPHERTEXT0{0..127}
  for TestNo = 0 to 399
    print TestNo, Key
    print Block{0..127}    // IV
    print Block{128..255}  // CIPHERTEXT
    for TestNo = 0 to 4999
      decrypt(Block{128..255},
                Dtext{0..127}, Key)
      Block{0..127} ^= Dtext{0..127}
      decrypt(Block{0..127},
                Dtext{0..127}, Key)
      Block{128..255} ^= Dtext{0..127}
    end for
    print Block{128..255}  // PLAINTEXT
    Key{0..KeyLength-1} ^=
                Block{256-KeyLength..255}
  end for
end for
```

42. In these pseudo code test descriptions a single 10,000 cycle loop has often been replaced by

a 5000 cycle double loop with a double length output block so that the last two 128-bit blocks are available in the form needed to compute the next key when the key length is either 192 or 256 bits. The order of these blocks is important since the new key will be different depending on the order in which the 128-bit blocks are placed into the 256-bit block. Here NIST intentions in this respect are interpreted to mean that the last block will be placed so that so that bit 127 of the penultimate block is followed by bit 0 of the last block.

Conclusions

43. The interface between AES algorithms and the external world should be defined at the lowest possible level so that issues of abstract representation within and external to algorithms are separated. The input, output and key blocks used by AES algorithms should be considered as blocks of bits without an imposed ordering convention. In practice the lowest level of interface likely to be available is that offered by arrays of bytes and it makes sense to employ this as the interface standard in the way that NIST has done. This interface should offer the minimum possible semantics in order to allow both users and designers the freedom to decide such issues for themselves. In practice the grouping of bits into bytes and the use of an order defined by byte memory addresses as a reference for this interface should provide a sound basis for consistent specification and implementation.

44. Where an algorithm design handles larger units consisting of groups of bytes it is incumbent on the designer to specify how bytes are combined to form these larger units, either by specifying big or little endian order or by providing a defined mapping between AES bit, byte or unit numbers and the entities employed within the algorithm.

Acknowledgement

45. I am most grateful to Markus Kuhn and Michael Roe of the Cambridge Computer Laboratory, Cambridge, UK and to Shai Halevi of IBM for a helpful exchange of views on the issues of unit order within cryptographic algorithms.

References

1. Brian R. Gladman: *Implementation Experience with AES Candidate Algorithms.* Second AES Conference, Rome, Italy, March 22-23, 1999.
2. Serge Vaudenay et al: *Report on the AES Candidates.* Second AES Conference, Rome, Italy, March 22-23, 1999.
3. Markus G. Kuhn: *A Bit Naming Convention for Cryptographic Algorithms.* AES First Round Comments.