1 AES-128 optimized for Speed

Throughout the following section, this paper will introduce an C implementation of a AES-128 cryptographic-environment optimized for speed. This implementation will ground in knowledge about loop unrolling, in-line computations and in memory (RAM) table lookups.

From previous chapters the AES-128 algorithm and specification is described in further details, thus this section will only point out the implementation on each step.

1.1 Summarise

The below table 1, summarises the performance of the implementation, using the C optimisation standard `-O0`, which in none. By allowing the C compiler to optimise further on the code, using for instance `-O1`, a RAM and Program decrease of $(1-(284/316)\cdot 100 = 10\%$ and $(1-(852/2594)\cdot 100 = 67\%$ respectively was achieved.

Optimization dimention	Value	Configuration Level
Program Memory	2594 Bytes	-O0
RAM Consumption	316 Bytes	-O0
Clock Cycles	21499 Clocks	-O0

Table 1: Optimisation for speed on an ATmega16 Micro Controller using -O0 configuration

1.2 Round key generation

From Figure 2 the round key implementation is illustrated. This function is created as a static function, which takes the round number as argument ie: $0, 1, \cdots, 10$, and modifies the Round Key accordingly:

1 If round number is 0 then set round key equal to the key

2 If round number is not 0 then modify the previous round key according to algorithm summary **??**.

1.3 Pre-whitening

The pre-whitening operation is performed, as a set of in-line operations in the main method, in order to waste clock- cycles on function calls. The in addition the encryption of the 128 bit message with the 128 bit round key is carried out without looping through the byte array[]. The implementation is viewable from figure **??**

1.4 The n-1 rounds

The main part of this AES algorithm is the n-1, in this case 9, rounds. Each round follows the previously described algorithm, see algorithm summary **??**, Once again, the loops is unrolled and performed in-line, in order to minimise the clock count. In general the implemented algorithm consists of two parts, see below, the implementation is visible from figure 3.

```
static void roundKeyGeneration(uint8_t i)
{
  if(i == 0)
  {
    roundKey[0] = key[0];
    roundKey[1] = key[1];
    roundKey[2] = key[2];
    roundKey[3] = key[3];
    roundKey[4] = key[4];
    roundKey[5] = key[5];
    roundKey[6] = key[6];
    roundKey[7] = key[7];
    roundKey[8] = key[8];
    roundKey[9] = key[9];
    roundKey[10] = key[10];
    roundKey[11] = key[11];
    roundKey[12] = key[12];
    roundKey[13] = key[13];
    roundKey[14] = key[14];
    roundKey[15] = key[15];
  }
  else
  {
    uint8_t temp[4];
    temp[0] = sBox[roundKey[13]] ^ rCon[i];
    temp[1] = sBox[roundKey[14]];
    temp[2] = sBox[roundKey[15]];
    temp[3] = sBox[roundKey[12]];
    roundKey[0] ^= temp[0];
    roundKey[1] ^= temp[1];
    roundKey[2] ^= temp[2];
    roundKey[3] ^= temp[3];
    roundKey[4] ^= roundKey[0];
    roundKey[5] ^= roundKey[1];
    roundKey[6] ^= roundKey[2];
    roundKey[7] ^= roundKey[3];
    roundKey[8] ^= roundKey[4];
    roundKey[9] ^= roundKey[5];
    roundKey[10] ^= roundKey[6];
    roundKey[11] ^= roundKey[7];
    roundKey[12] ^= roundKey[8];
    roundKey[13] ^= roundKey[9];
    roundKey[14] ^= roundKey[10];
    roundKey[15] ^= roundKey[11];
  }
}
```

Figure 1: Round key implementation optimized for speed.

1 The first part is the Subbytes and Shiftrows operations, which is both performed at the same time, for each byte in the message. That is, the bytes are shifted and looked up in the S-Box in the same step.

2 The second part of the algorithm, is the mixcolumn and the add round key operations, which is likewise performed together for each message byte. Note that this part utilises 4 temporary registers, in order to enforce data consistency throughout the calculations.

1.5 The final round

The final round is actually implemented as the first part in the (n-1) round loop. That is, the Subbytes and Shiftrows is calculated in the same step. However, this time we need to add the round key as well. Thus yielding the implementation of Figure 4

```
roundKeyGeneration(round);
message[0] ^= roundKey[0];
message[1] ^= roundKey[1];
message[2] ^= roundKey[2];
message[3] ^= roundKey[3];
message[4] ^= roundKey[4];
message[5] ^= roundKey[5];
message[6] ^= roundKey[6];
message[7] ^= roundKey[7];
message[8] ^= roundKey[8];
message[9] ^= roundKey[9];
message[10] ^= roundKey[10];
message[11] ^= roundKey[11];
message[12] ^= roundKey[12];
message[13] ^= roundKey[13];
message[14] ^= roundKey[14];
message[15] ^= roundKey[15];
```

Figure 2: Pre-whitening implementation optimized for speed.

```
while(round < 10)
{
  // Subbytes + Shiftrows
  message[0] = sBox[message[0]];
  message[4] = sBox[message[4]];
  message[8] = sBox[message[8]];
  message[12] = sBox[message[12]];

  temp = message[1];
  message[1] = sBox[message[5]];
  message[5] = sBox[message[9]];
  message[9] = sBox[message[13]];
  message[13] = sBox[temp];

  temp = message[2];
  temp1 = message[6];
  message[2] = sBox[message[10]];
  message[6] = sBox[message[14]];
  message[10] = sBox[temp];
  message[14] = sBox[temp1];

  temp = message[11];
  message[11] = sBox[message[7]];
  message[7] = sBox[message[3]];
  message[3] = sBox[message[15]];
  message[15] = sBox[temp];

  // MixColumns + AddRowKey
  roundKeyGeneration(round);
  temp = message[0];
  temp1 = message[1];
  temp2 = message[2];
  temp3 = message[3];
  message[0] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[0];
  message[1] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[1];
  message[2] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[2];
  message[3] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[3];

  temp = message[4];
  temp1 = message[5];
  temp2 = message[6];
  temp3 = message[7];
  message[4] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[4];
  message[5] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[5];
  message[6] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[6];
  message[7] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[7];

  temp = message[8];
  temp1 = message[9];
  temp2 = message[10];
  temp3 = message[11];
  message[8] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[8];
  message[9] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[9];
  message[10] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[10];
  message[11] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[11];

  temp = message[12];
  temp1 = message[13];
  temp2 = message[14];
  temp3 = message[15];
  message[12] = (x2(temp) + (x2(temp1) ^ temp1) + temp2 + temp3) ^ roundKey[12];
  message[13] = (temp + x2(temp1) + (x2(temp2) ^ temp2) + temp3) ^ roundKey[13];
  message[14] = (temp + temp1 + x2(temp2) + (x2(temp3) ^ temp3)) ^ roundKey[14];
  message[15] = ((x2(temp) ^ temp) + temp1 + temp2 + x2(temp3)) ^ roundKey[15];

  // Prepare next round
  round = round + 1;
}
```

Figure 3: Implementation of the (n-1)-round loop, optimized for speed.

```
roundKeyGeneration(round);
message[0] = (sBox[message[0]]) ^ roundKey[0];
message[4] = (sBox[message[4]]) ^ roundKey[4];
message[8] = (sBox[message[8]]) ^ roundKey[8];
message[12] = (sBox[message[12]]) ^ roundKey[12];

temp = message[1];
message[1] = (sBox[message[5]]) ^ roundKey[1];
message[5] = (sBox[message[9]]) ^ roundKey[5];
message[9] = (sBox[message[13]]) ^ roundKey[9];
message[13] = (sBox[temp]) ^ roundKey[13];

temp = message[2];
temp1 = message[6];
message[2] = (sBox[message[10]]) ^ roundKey[2];
message[6] = (sBox[message[14]]) ^ roundKey[6];
message[10] = (sBox[temp]) ^ roundKey[10];
message[14] = (sBox[temp1]) ^ roundKey[14];

temp = message[11];
message[11] = (sBox[message[7]]) ^ roundKey[11];
message[7] = (sBox[message[3]]) ^ roundKey[7];
message[3] = (sBox[message[15]]) ^ roundKey[3];
message[15] = (sBox[temp]) ^ roundKey[15];
```

Figure 4: Implementation of the (n) iteration, optimized for speed.