1 AES-128 optimized for Code Size

The following section will illustrate tricks and exemplification upon how to optimise C-Code in the dimension of Code-Size. That is, by minimising the Program Memory consumption. Again, this optimisation is done on the AES-128 Cryptography Algorithm.

1.1 Summarise

As observed it was possible to do an astonishing $(1-(1888/2594))\cdot 100 = 27\%$ reduction in program memory compared to the Speed optimisation. However, this comes with a trade off. From table 1 it is readily seen, that the clock count of this optimisation strategy suffered great impact. If we compare this result to the speed-dimension, an $45490/21499 = 2.12$ factor difference, meaning that this programme executes more than twice as slow compared to the speed optimisation. Moreover, it is noticed that the RAM consumption merely changes between these two implementation strategies.

Optimization dimention	Value	Configuration Level
Program Memory	1888 Bytes	-O0
RAM Consumption	350 Bytes	-O0
Clock Cycles	45490 Clocks	-O0

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

1.2 Round key generation

As for the previous implementation, the round key generation of this strategy consists of two parts, see below. Moreover, the function takes the round number as input parameter. The implementation is illustrated in figure 1. From the figure, it is seen how this approach utilises the usage of loops in order to achieve the same functionality. This is done, since loops reduces code size, both visually and in terms of program memory.

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 **??**.

1.3 Pre-whitening

It really becomes visual how loops reduces the code size, when an example like the pre-whitening implementation below, figure 2, is provided. What previously consisted of 16 lines of code is combined into only 4 lines, thus removing 1/4 of the visual complexity. Furthermore, from the implementation, line 4, it is seen that the pre-whitening procedure is used to load the input message into our global static AES state variable.

```
static void roundKeyGeneration(uint8_t i)
{
  if(i == 0)
  {
    for(int i = 0; i < 16; i++)
    {
      roundKey[i] = key[i];
    }
  }
  else
  {
    uint8_t temp[4] =
    {
      sBox[roundKey[13]] ^ rCon[i], sBox[roundKey[14]], sBox[roundKey[15]], sBox[roundKey[12]]
    };

    for(int i = 0; i < len; i++)
    {
      roundKey[i] ^= (i < 4) ? temp[i] : roundKey[i-4];
    }
  }
}
```

Figure 1: Round key implementation optimized for speed.

```
roundKeyGeneration(round);
for (int i = 0; i < len; i++)
{
  message[i] = in_message[i] ^ roundKey[i];
}
```

Figure 2: Round key implementation optimized for program memory.

1.4 The n-1 rounds

From figure 3 the implementation of the AES-128, n-1 round iterations is illustrated. Again, a loop has been utilized to minimise visual and memory complexity. Moreover, a observation is made in this implementation. If the strategy did not utilise the `temp[4]` byte array to store the result from the substitution and shift row operation, and thus simply recalculated them every time in-line in the Mix Column operations, the programme memory is increased to 2660 bytes, which is even more than the space optimisation implementation. Finally, it is worthwhile noting that the encryption of the very message and the round key, in this strategy, is provided in its own function, figure 4.

```
while(round < 10)
{
  for(int i = 0; i < 4; i++)
  {
        // Subbytes + Shiftrows + MixColumns
    temp[0] = sBox[message[(i*4) % 16]];
    temp[1] = sBox[message[(i*4+5) % 16]];
    temp[1] = sBox[message[(i*4+10) % 16]];
    temp[1] = sBox[message[(i*4+15) % 16]];
    temporaryState[i*4] = (x2(temp[0]) + (x2(temp[1]) ^ temp[1]) + temp[1] + temp[1]);
    temporaryState[i*4+1] = (temp[0] + x2(temp[1]) + (x2(temp[1]) ^ temp[1]) + temp[1]);
    temporaryState[i*4+2] = (temp[0] + sBox[temp[1]] + x2(temp[1]) + (x2(temp[1]) ^ temp[1]));
    temporaryState[i*4+3] = ((x2(temp[0]) ^ temp[0]) + temp[1] + temp[1] + x2(temp[1]));
  }
  // AddRowKey
  roundKeyGeneration(round);
  encrypt();
  round = round + 1;
}
```

Figure 3: The n-1, 9, rounds of subbytes, shiftrows, mixcolumns, and addkey operations of AES-128, optimised for code size.

```
static void encrypt()
{
  for (int i = 0; i < len; i++)
  {
    message[i] = temporaryState[i] ^ roundKey[i];
  }
}
```

Figure 4: The Add Key implementation of the AES-128, optimised for code size.

1.5 The final round

Figure 5, provides the strategy of the final round, that is the 10th round of the algorithm. It is readily seen, that the implementation utilises a loop in order to minimise the code complexity. In addition, this loop iterates over four bytes at a time. This implementation is chosen since it allows for a small speed optimisation. In specific this trick allows for us to do the shiftrows and the byte substitution operation concurrently.

```
// Subbytes + Shiftrows
for(int i = 0; i < 4; i++)
{
  // Subbytes + Shiftrows
  temporaryState[i*4] = sBox[message[(i*4) % 16]];
  temporaryState[i*4+1] = sBox[message[(i*4+5) % 16]];
  temporaryState[i*4+2] = sBox[message[(i*4+10) % 16]];
  temporaryState[i*4+3] = sBox[message[(i*4+15) % 16]];
}
// AddRoundKey
roundKeyGeneration(round);
encrypt();
```

Figure 5: The final round, optimised for code size.