

Optimizing BLUE MIDNIGHT WISH for size

Daniel Otte
(daniel.otte@rub.de) *

May 2010

Abstract

This document shows how an implementation of BLUE MIDNIGHT WISH can be tuned towards small code size. Examples are given in the C programming language for the AVR but the ideas can also be implemented in other environments.

An implementation based on the examples shown in this document achieves a code size of 4447 byte. An assembly version which implements the shown techniques achieves a code size as low as 1536 byte.

1 About this document

The intend of this document is to show how BLUE MIDNIGHT WISH [2] can be implemented for size constrained devices like microcontrollers. We use C language examples to outline how this can be turned into concrete code. As reference platform we use Atmels AVR CPU, which forms the base for the popular ATmega and ATtiny microcontrollers.

It may be useful to employ environment specific optimizations as well. One example hereby is storing constants into flash memory. Also at some points it may be beneficial to know the underlying architecture, for example the fact that the AVR has only three pointer registers.

This paper discusses only the implementation of the core functions since padding and organization is similar to those of other hash functions and optimized implementation highly depends on the API to implement.

Implementations for the AVR and their properties are available at [1].

*This document was developed during my stay at the Q2S Center of excellence at the NTNU, Norway. I wouldz like to thank everyone there for their great help.

2 The role of size optimization

The importance of size optimization is often underestimated. Usually, the main motivation of optimizing cryptographic algorithms is to improve the speed of such algorithms. This is due to nature of the mostly considered applications as well as hardware platforms. The latter offers usually enough space (for example in the caches) for implementations. Therefore most often space constrains are of minor importance and the implementation is optimized for speed.

Apart from this there are some applications where the size of the executable code is very important or even critical. These applications are mostly based on very constrained devices like microcontrollers for smart-cards or RFID tickets.

While the requirements of data memory can be estimated relatively easily for a given algorithm, the code memory requirements are difficult to estimate. They depend very much on the target hardware, compiler and optimization level.

3 Reasons for using the AVR platform

We chose the AVR for showing size optimizations due to the following reasons:

- availability: the AVR core is available in a number of differently parameterized microcontrollers
- price: AVR microcontrollers are quite popular in the low cost segment
- development tools: there is a very good open source tool-chain available for AVRs including the GCC

4 About the AVR

The AVR is an eight bit RISC machine with 32 eight-bit general purpose registers having a relative orthogonal instruction set. It is based on the Harvard-Architecture, according to that it has two separate buses for code (flash) and data (SRAM). Additionally it is a Load-and-Store architecture. The only instructions operating on data in RAM are the `load` and `store` instructions (except for a few instructions which use the stack). Most AVR instructions are 16 bits wide and a few are 32-bits wide. AVR instructions are mostly executed in one cycle but some may take up to three cycles.

4.1 Registers

The AVR core consists of 32 eight bit wide general purpose registers (denoted `r0` to `r31`), a flag register, a stack pointer register, and the program counter register (`PC`). Six of the general purpose registers (`r26` to `r31`) can be combined to three 16-bit pointer registers (`X`, `Y` and `Z`).

4.2 Memory access

Access to the memory is provided through two instructions: load and store. Both instructions are available in two addressing modes.

Direct-addressing, where the address to load from, or to store to, is directly coded into the instruction. This mode is suitable for accessing global variables with static addresses.

Indirect-addressing where the destination is determined by one of the three pointer registers. In indirect-addressing mode it is possible to post-increment or pre-decrement the pointer register. Also the `Z` or `Y` pointers can be used with a displacement in the range from 0 to 63.

It is also possible to load data from the flash memory into registers. This is done by the `lpm` instruction, which loads the byte addressed by the `Z` pointer and optionally increments the `Z` pointer.

4.3 Special instructions

The instruction set of the AVR is similar to those of most other micro-controllers. A few special capabilities and limits are described here. An important fact when implementing BLUE MIDNIGHT WISH is that rotations in both directions can only be done by one bit at a time.

4.3.1 16-bit moves

Modern AVR's are capable of transferring the contents of a register pair into another register pair. A register pair is formed by an even register and its next register.

4.3.2 loading/ORing/ANDing of constants

Registers `r16` and up can be used and loaded with an immediate value – a value coded directly into the instruction.

4.3.3 16-bit immediate addition/subtraction

The four register pairs formed by the registers `r24` and up can have a value in the range from 0 to 63 added or subtracted. The addition of zero to such a pair can be used to easily determine if the 16-bit value is zero.

4.3.4 Skip instructions

Some AVR instructions have the capability of skipping the next instruction. These instructions are very useful if only a single instruction should be executed conditionally or to skip a jump to exit a loop.

The `cpse` instruction compares two registers and skips the next instruction if the contents are equal (compare and skip if equal).

The `sbrs/sbrc` instructions allow to skip the next instruction if a bit in a register is set or cleared.

5 Why we chose Blue Midnight Wish

We chose BLUE MIDNIGHT WISH to illustrate possible size optimizations due to different reasons:

1. BLUE MIDNIGHT WISH has a quite complex code structure and so has a huge potential for optimizations
2. BLUE MIDNIGHT WISH is the fastest algorithm on the NIST reference platform
3. The BLUE MIDNIGHT WISH team offered valuable support for implementing an optimized version

For this report we will limit ourselves to only analyzing BLUE MIDNIGHT WISH-256, but most ideas can also be applied to other BLUE MIDNIGHT WISH variants.

6 Helper functions

For implementing BLUE MIDNIGHT WISH optimized for size some helper functions may be useful for performing tasks appearing in different parts of the hash function.

We are defining some functions in this section which are used in the C-examples in the remaining paper.

6.1 Shifts and Rotates

BMW256 make extensive use of 32-bit shifts and rotates. Due to the fact that we consider implementing it on an 8-bit CPU, which requires multiple instruction to perform those operations. Here, the code size can be reduced by implementing this shifts and rotates in dedicated functions.

```
1 uint32_t shift32_left(uint32_t a, int8_t shift){
2     if(shift < 0)
3         return(shift32_right(a, shift));
4     return a<<shift;
5 }
6
7 uint32_t shift32_right(uint32_t a, uint8_t shift){
8     return a>>shift;
9 }
10
11 uint32_t rotate32_left(uint32_t a, uint8_t rotate){
12     return (a<<rotate)|(a>>(32-rotate));
13 }
14
15 #define rotate32_right(a,n) rotate32_left((a), 32-(n))
```

Listing 1: shift and rotate examples

6.2 memxor

BLUE MIDNIGHT WISH makes use of the XOR function at many places. Often this xoring can be performed by directly xoring memory regions. To exploit this we use a dedicated function which xors one memory region into another.

```
1 void memxor(void* dest, void* src, size_t n){
2     while(n--){
3         *((uint8_t*)dest) ^= *((uint8_t*)src);
4         dest = (uint8_t*)dest + 1;
5         src = (uint8_t*)src + 1;
6     }
7 }
```

Listing 2: memxor examples

6.3 Using flash for constants

By default the GCC-Compiler places all data tables in RAM even constant ones. To reduce RAM usage it is useful to place the data only in flash. Due to lacking support for different memory regions in GCC, usage of special functions to load data from flash is required.

The keyword `PROGMEM` is used to place a object only in flash. Data is loaded from flash with the functions `pgm_read_byte()`, `pgm_read_word()` and `pgm_read_dword()`, which load a byte, a 16-bit word and a 32-bit word respectively. All these functions take a pointer to the object in flash as a single parameter and return the read value.

7 Implementing the logic functions

7.1 S functions

The S functions s_0 to s_4 are used in **f0** and **f1**, s_5 is used only in **f1**. Since all S functions share the same structure it is possible to implement them all in a single generic function. This function takes the number of the S function to perform as a parameter.

$$\begin{aligned} s_0(x) &= SHR^1(x) \oplus SHL^3(x) \oplus ROTL^4(x) \oplus ROTR^{13}(x) \\ s_1(x) &= SHR^1(x) \oplus SHL^2(x) \oplus ROTL^8(x) \oplus ROTR^9(x) \\ s_2(x) &= SHR^2(x) \oplus SHL^1(x) \oplus ROTL^{12}(x) \oplus ROTR^7(x) \\ s_3(x) &= SHR^2(x) \oplus SHL^2(x) \oplus ROTL^{15}(x) \oplus ROTR^3(x) \\ s_4(x) &= SHR^1(x) \oplus x \\ s_5(x) &= SHR^2(x) \oplus x \end{aligned}$$

Table 1: Definition of the S functions

```
1  const uint8_t s_table [6][4] PROGMEM = {
2    { 1, 3, 4, 13 },
3    { 1, 2, 8, 9 },
4    { 2, 1, 12, 7 },
5    { 2, 2, 15, 3 },
6    { 1, 0, 0, 0 },
7    { 2, 0, 0, 0 },
8  };
```

```

9
10 uint32_t s32(uint32_t x, uint8_t s){
11     uint32_t r;
12     uint8_t *p = (uint8_t*)s_table+4*s;
13     r = shift32_right( x, pgm_read_byte(p++))
14         ^ shift32_left( x, pgm_read_byte(p++))
15         ^ rotate32_left( x, pgm_read_byte(p++))
16         ^ rotate32_right(x, pgm_read_byte(p));
17     return r;
18 }

```

Listing 3: memxor examples

8 Optimizations on f0

The definition of **f0** is given in table 2. It is obvious that each word in H is xored together with the corresponding word in M. So instead of performing 80 xors of 32-bit values we can simply xor the complete H array with the M array. In order to save some RAM we decided to xor M directly into H and do the same again at the end of the computation of W to get H back.

As we can see in table 3, the five columns on the right are complete iterations over the T-array starting at different offsets. A good optimization for code size seems to iterate several times over the T-array. It would also be possible to use five index variables, but due to the nature of the AVR we only have three pointer registers to address memory. The handling of the signs is a bit more difficult since there is no pattern to exploit. A solution is to pack the signs in a binary structure. We decided to use a 1 for minus and 0 for plus. The transformation is illustrated in table 4.

In the code listing 4, we load the signs packed into 16 bit words and shift out bit by bit to get the individual signs. Also, we use our generic implementation of the S-functions with a parameter. The parameter is reset to 4 when going below 0, so effectively performing a "modulo 5" operation, but without using division.

```

1  const uint16_t sign_table [] =
2    { 0x0311, 0xDDB3, 0x2A79, 0x07AA, 0x51C2 };
3  const uint8_t offset_table [] =
4    { 4, 6, 9, 12, 13};
5
6  void bmw32_f0(uint32_t* q, uint32_t* h, uint32_t* m){
7    uint8_t i,row,column;
8    int8_t s_select;
9    uint16_t sign_reg;
10
11   /* xor m into h */
12   memxor(h, m, 64);
13   /* initially set q array to zero */
14   memset(q, 0, 4*16);
15   column = 4;
16   do{
17     i = 15;
18     row = offset_table[column];
19     sign_reg = hack_table[column];
20     do{
21       if(sign_reg&1){
22         q[i] -= h[row&15];
23       }else{
24         q[i] += h[row&15];
25       }
26       --row;
27       sign_reg >>= 1;
28     }while(i-- != 0);
29   }while(column-- != 0);
30   /* xor m into h again, to get the original h */
31   memxor(h, m, 64);
32   i = 15;
33   s_select = 0;
34   do{
35     q[i] = s32(q[i], s_select--) + h[(i+1)&15];
36     if( s_select == -1){
37       s_select = 4;
38     }
39   }while(i-- != 0);
40 }

```

Listing 4: first part of f0 example

$f_0 : \{0,1\}^{2m} \rightarrow \{0,1\}^m$	
Input: Message block $M^{(i)} = (M_0^{(i)}, M_1^{(i)}, \dots, M_{15}^{(i)})$, and the previous double pipe $H^{(i-1)} = (H_0^{(i-1)}, H_1^{(i-1)}, \dots, H_{15}^{(i-1)})$.	
Output: First part of the quadruple pipe $Q_a^{(i)} = (Q_0^{(i)}, Q_1^{(i)}, \dots, Q_{15}^{(i)})$.	
<p>1. Bijective transform of $M^{(i)} \oplus H^{(i-1)}$:</p> $ \begin{aligned} W_0^{(i)} &= (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\ W_1^{(i)} &= (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_8^{(i)} \oplus H_8^{(i-1)}) + (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) - (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\ W_2^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) + (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\ W_3^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) \\ W_4^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) - (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\ W_5^{(i)} &= (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\ W_6^{(i)} &= (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) \\ W_7^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) - (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\ W_8^{(i)} &= (M_2^{(i)} \oplus H_2^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) - (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\ W_9^{(i)} &= (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_3^{(i)} \oplus H_3^{(i-1)}) + (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{14}^{(i)} \oplus H_{14}^{(i-1)}) \\ W_{10}^{(i)} &= (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_1^{(i)} \oplus H_1^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{15}^{(i)} \oplus H_{15}^{(i-1)}) \\ W_{11}^{(i)} &= (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_0^{(i)} \oplus H_0^{(i-1)}) - (M_2^{(i)} \oplus H_2^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) + (M_9^{(i)} \oplus H_9^{(i-1)}) \\ W_{12}^{(i)} &= (M_1^{(i)} \oplus H_1^{(i-1)}) + (M_5^{(i)} \oplus H_5^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_9^{(i)} \oplus H_9^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) \\ W_{13}^{(i)} &= (M_2^{(i)} \oplus H_2^{(i-1)}) + (M_4^{(i)} \oplus H_4^{(i-1)}) + (M_7^{(i)} \oplus H_7^{(i-1)}) + (M_{10}^{(i)} \oplus H_{10}^{(i-1)}) + (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) \\ W_{14}^{(i)} &= (M_3^{(i)} \oplus H_3^{(i-1)}) - (M_5^{(i)} \oplus H_5^{(i-1)}) + (M_8^{(i)} \oplus H_8^{(i-1)}) - (M_{11}^{(i)} \oplus H_{11}^{(i-1)}) - (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) \\ W_{15}^{(i)} &= (M_{12}^{(i)} \oplus H_{12}^{(i-1)}) - (M_4^{(i)} \oplus H_4^{(i-1)}) - (M_6^{(i)} \oplus H_6^{(i-1)}) - (M_9^{(i)} \oplus H_9^{(i-1)}) + (M_{13}^{(i)} \oplus H_{13}^{(i-1)}) \end{aligned} $	
<p>2. Further bijective transform of $W_j^{(i)}, j = 0, \dots, 15$:</p> $ \begin{aligned} Q_0^{(i)} &= s_0(W_0^{(i)}) + H_1^{(i-1)}; & Q_1^{(i)} &= s_1(W_1^{(i)}) + H_2^{(i-1)}; & Q_2^{(i)} &= s_2(W_2^{(i)}) + H_3^{(i-1)}; & Q_3^{(i)} &= s_3(W_3^{(i)}) + H_4^{(i-1)}; \\ Q_4^{(i)} &= s_4(W_4^{(i)}) + H_5^{(i-1)}; & Q_5^{(i)} &= s_0(W_5^{(i)}) + H_6^{(i-1)}; & Q_6^{(i)} &= s_1(W_6^{(i)}) + H_7^{(i-1)}; & Q_7^{(i)} &= s_2(W_7^{(i)}) + H_8^{(i-1)}; \\ Q_8^{(i)} &= s_3(W_8^{(i)}) + H_9^{(i-1)}; & Q_9^{(i)} &= s_4(W_9^{(i)}) + H_{10}^{(i-1)}; & Q_{10}^{(i)} &= s_0(W_{10}^{(i)}) + H_{11}^{(i-1)}; & Q_{11}^{(i)} &= s_1(W_{11}^{(i)}) + H_{12}^{(i-1)}; \\ Q_{12}^{(i)} &= s_2(W_{12}^{(i)}) + H_{13}^{(i-1)}; & Q_{13}^{(i)} &= s_3(W_{13}^{(i)}) + H_{14}^{(i-1)}; & Q_{14}^{(i)} &= s_4(W_{14}^{(i)}) + H_{15}^{(i-1)}; & Q_{15}^{(i)} &= s_0(W_{15}^{(i)}) + H_0^{(i-1)}; \end{aligned} $	

Table 2: Definition of \mathbf{f}_0

$T_j^{(i)} = M_j^{(i)} \oplus H_j^{(i-1)}$						
$W_0^{(i)}$	$=$	$+T_5^{(i)}$	$-T_7^{(i)}$	$+T_{10}^{(i)}$	$+T_{13}^{(i)}$	$+T_{14}^{(i)}$
$W_1^{(i)}$	$=$	$+T_6^{(i)}$	$-T_8^{(i)}$	$+T_{11}^{(i)}$	$+T_{14}^{(i)}$	$-T_{15}^{(i)}$
$W_2^{(i)}$	$=$	$+T_7^{(i)}$	$+T_9^{(i)}$	$-T_{12}^{(i)}$	$+T_{15}^{(i)}$	$+T_0^{(i)}$
$W_3^{(i)}$	$=$	$+T_8^{(i)}$	$-T_{10}^{(i)}$	$+T_{13}^{(i)}$	$+T_0^{(i)}$	$-T_1^{(i)}$
$W_4^{(i)}$	$=$	$+T_9^{(i)}$	$-T_{11}^{(i)}$	$-T_{14}^{(i)}$	$+T_1^{(i)}$	$+T_2^{(i)}$
$W_5^{(i)}$	$=$	$+T_{10}^{(i)}$	$-T_{12}^{(i)}$	$+T_{15}^{(i)}$	$-T_2^{(i)}$	$+T_3^{(i)}$
$W_6^{(i)}$	$=$	$-T_{11}^{(i)}$	$+T_{13}^{(i)}$	$-T_0^{(i)}$	$-T_3^{(i)}$	$+T_4^{(i)}$
$W_7^{(i)}$	$=$	$-T_{12}^{(i)}$	$-T_{14}^{(i)}$	$+T_1^{(i)}$	$-T_4^{(i)}$	$-T_5^{(i)}$
$W_8^{(i)}$	$=$	$+T_{13}^{(i)}$	$-T_{15}^{(i)}$	$+T_2^{(i)}$	$-T_5^{(i)}$	$-T_6^{(i)}$
$W_9^{(i)}$	$=$	$+T_{14}^{(i)}$	$+T_0^{(i)}$	$-T_3^{(i)}$	$+T_6^{(i)}$	$-T_7^{(i)}$
$W_{10}^{(i)}$	$=$	$+T_{15}^{(i)}$	$-T_1^{(i)}$	$-T_4^{(i)}$	$-T_7^{(i)}$	$+T_8^{(i)}$
$W_{11}^{(i)}$	$=$	$-T_0^{(i)}$	$-T_2^{(i)}$	$-T_5^{(i)}$	$+T_8^{(i)}$	$+T_9^{(i)}$
$W_{12}^{(i)}$	$=$	$+T_1^{(i)}$	$+T_3^{(i)}$	$-T_6^{(i)}$	$-T_9^{(i)}$	$+T_{10}^{(i)}$
$W_{13}^{(i)}$	$=$	$+T_2^{(i)}$	$+T_4^{(i)}$	$+T_7^{(i)}$	$+T_{10}^{(i)}$	$+T_{11}^{(i)}$
$W_{14}^{(i)}$	$=$	$+T_3^{(i)}$	$-T_5^{(i)}$	$+T_8^{(i)}$	$-T_{11}^{(i)}$	$-T_{12}^{(i)}$
$W_{15}^{(i)}$	$=$	$-T_4^{(i)}$	$-T_6^{(i)}$	$-T_9^{(i)}$	$+T_{12}^{(i)}$	$+T_{13}^{(i)}$

Table 3: rearrangement of first part of **f0**

					0x0311	0xDDB3	0x2A79	0x07AA	0x51C2	
0 (msb):	+	-	+	+	+	0	1	0	0	0
1 :	+	-	+	+	-	0	1	0	0	1
2 :	+	+	-	+	+	0	0	1	0	0
3 :	+	-	+	+	-	0	1	0	0	1
4 :	+	-	-	+	+	0	1	1	0	0
5 :	+	-	+	-	+	0	1	0	1	0
6 :	-	+	-	-	+	1	0	1	1	0
7 :	-	-	+	-	-	1	1	0	1	1
8 :	+	-	+	-	-	0	1	0	1	1
9 :	+	+	-	+	-	0	0	1	0	1
10 :	+	-	-	-	+	0	1	1	1	0
11 :	-	-	-	+	+	1	1	1	0	0
12 :	+	+	-	-	+	0	0	1	1	0
13 :	+	+	+	+	+	0	0	0	0	0
14 :	+	-	+	-	-	0	1	0	1	1
15 (lsb):	-	-	-	+	+	1	1	1	0	0

Table 4: sign translation

9 Optimizations on `f1`

The purpose of the `f1` function is to generate the second half of the quadruple pipe, from `H`, the message block and the first half of the quadruple pipe. It does so by generating word by word, as each word depends on the 16 words before.

There are two different methods to compute a word for the quadruple pipe, named `expand1` and `expand2`. They both perform the `AddElement` first, and then add values based on the 16 preceding words of `Q`. Since the `AddElement` function is the same for `expand1` and `expand2`, it is useful to implement it in a dedicated function.

Implementing `f1` is straight forward. We implemented separate functions for `expand1` and `expand2` which both use the `expand_base` functions which implement nearly completely the `AddElement` function. We did not implement the xor with h_{j+7} as part of `expand_base` to avoid passing an additional parameter to the function. In `expand1` the transformation of `Q` is done by our generic `S` function implementation, while we implement the `R` function as direct rotates for `expand2`, since it is only used in `expand2`. The constants are used from a precomputed table.

```
1  static const
2  uint32_t k_lut [] PROGMEM = {
3      0x55555550L, 0x5aaaaaa5L, 0x5ffffffaL,
4      0x6555554fL, 0x6aaaaaa4L, 0x6ffffff9L,
5      0x7555554eL, 0x7aaaaaa3L, 0x7ffffff8L,
6      0x8555554dL, 0x8aaaaaa2L, 0x8ffffff7L,
7      0x9555554cL, 0x9aaaaaa1L, 0x9ffffff6L,
8      0xa555554bL };
9
10 uint32_t expand_base(uint8_t j, const void* m){
11     return ( rotate32_left (((uint32_t*)m)[j&0xf],
12                          ((j+0)&0xf)+1 )
13           + rotate32_left (((uint32_t*)m)[(j+3)&0xf],
14                          ((j+3)&0xf)+1 )
15           - rotate32_left (((uint32_t*)m)[(j+10)&0xf],
16                          ((j+10)&0xf)+1 )
17           + pgm_read_dword_near(&(k_lut[j])));
18 }
19
20 uint32_t bmw_small_expand1(uint8_t j, const uint32_t* q,
21     const void* m, const void* h){
22     uint32_t r;
23     uint8_t i;
```

```

24     r = expand_base(j, m) ^ ((uint32_t*)h)[(j+7)&0xf];
25     for(i=0; i<16; ++i){
26         r += s32(q[j+i], (i+1)%4);
27     }
28     return r;
29 }
30
31 uint32_t bmw_small_expand2(uint8_t j, const uint32_t* q,
32     const void* m, const void* h){
33     uint8_t rotates [] = { 3, 7, 13, 16, 19, 23, 27};
34     uint32_t r;
35     uint8_t i;
36     r = expand_base(j, m) ^ ((uint32_t*)h)[(j+7)&0xf];
37     for(i=0; i<14; i+=2){
38         r += q[j+i];
39         r += rotate32_left(q[j+i+1], rotates[i/2]);
40     }
41     r += s32(q[j+14], 4);
42     r += s32(q[j+15], 5);
43     return r;
44 }
45
46 void bmw_small_f1(uint32_t* q, const void* m, const void* h){
47     uint8_t i;
48     q[16] = bmw_small_expand1(0, q, m, h);
49     q[17] = bmw_small_expand1(1, q, m, h);
50     for(i=2; i<16; ++i){
51         q[16+i] = bmw_small_expand2(i, q, m, h);
52     }
53 }

```

Listing 5: **f1** example

10 Optimizations on f2

The **f2** function compresses the previous double pipe, the quadruple pipe and the message block into the new double pipe. There are numerous ways to perform the computation of **f2**. In our case, at least the computation of XL and XH is simple and straight forward. The rest of the computation is rearranged as shown in table 5.

The second part is structured in such a way that at no time more than three pointers are needed. Additionally, the reuse of the loop constructs reduces the total amount of instructions even more.

1. The message block is copied into the double pipe
2. $H_0..H_7$ are updated (xor) with shifted values of XH and $Q_{16}..Q_{23}$
3. $Q_{24}..Q_{31}$ is xored into $Q_0..Q_7$
4. XH and $Q_{24}..Q_{31}$ are xored into $H_8..H_{15}$
5. $H_0..H_7$ are updated (add) with the xor of XL and $Q_0..Q_7$ also this new values are rotated and add to $H_8..H_{15}$
6. $Q_{16}..Q_{22}$ is xored into $Q_9..Q_{15}$ and Q_{23} is xored into Q_8
7. Finally $H_8..H_{15}$ are updated (add) with the xor of $Q_8..Q_{15}$ and shifted values of XL

```
1 int8_t shift_table_1 [] PROGMEM =
2   { 5, -5, -7, 8, -5, 5, -1, 5, -3, 0, 6, -6, -4, 6, -11, 2 };
3 int8_t shift_table_2 [] PROGMEM =
4   { 8, -6, 6, 4, -3, -4, -7, -2 };
5
6 void bmw_small_f2(uint32_t* h, uint32_t* q, const void* m){
7   uint32_t xl=0, xh;
8   uint8_t i;
9   const int8_t *ptr;
10  for(i=16;i<24;++i){
11    xl ^= q[i];
12  }
13  xh = xl;
14  for(i=24;i<32;++i){
15    xh ^= q[i];
16  }
17  // ① copy m into h
```

```

18 memcpy(h, m, 16*4);
19 ptr = shift_table_1;
20 // ②
21 for(i=0; i<8; ++i){
22     h[i] ^= shift32_left(xh, pgm_read_byte(ptr++));
23     h[i] ^= shift32_left(q[16+i], pgm_read_byte(ptr++));
24 }
25 // ③  $Q_i \leftarrow Q_i \oplus Q_{i+24} \forall i \in [0..7]$ 
26 memxor(q, q+24, 8*4);
27 for(i=0; i<8; ++i){
28     // ④
29     h[8+i] ^= xh ^ q[24+i];
30     // ⑤
31     h[8+i] += rotate32_left(h[(4+i)%8] += xl^q[(4+i)%8], i+9);
32 }
33 // ⑥  $Q_i \leftarrow Q_i \oplus Q_{i+7} \forall i \in [9..15]$ 
34 memxor(q+9, q+16, 7*4);
35 q[8] ^= q[23];
36 ptr = shift_table_2;
37 // ⑦
38 for(i=8; i<16; ++i){
39     h[i] += shift32_left(xl, pgm_read_byte(ptr++)) ^ q[i];
40 }
41 }

```

Listing 6: **f2** example

References

- [1] *AVR-Crypto-Lib*. Das LABOR e.V., Bochum, 2010. <http://www.das-labor.org/wiki/AVR-Crypto-Lib/en>.
- [2] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jorn Amundsen, and Stig Frode Mjolsnes. Cryptographic hash function blue midnight wish. http://people.item.ntnu.no/~danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf, 2009. Submission to NIST (Round 2).