Graz University of Technology
IAIK
Institute for Applied Information Processing and Communications
Inffeldgasse 16a
A–8010 GRAZ

Bachelor Thesis

Efficient Implementation of the Grøstl-256 Hash Function on an ATmega163 Microcontroller

Günther A. Roland
(g.roland@student.tugraz.at)

Supervisor:
Martin Schläffer
(martin.schlaeffer@iaik.tugraz.at)

June, 2009

http://www.iaik.tugraz.at/content/teaching/

Efficient Implementation of the Grøstl-256 Hash Function on an ATmega163 Microcontroller

Günther A. Roland

June 16, 2009

Abstract

This work presents efficient implementations of Grøstl-256 using different memory footprints on an ATmega163 microcontroller. Due to the limited resources on the microcontroller we present different tradeoffs between memory usage and speed. We have implemented three different versions, all for Grøstl with a hash size of 256 bits. The high speed version using 994 bytes of SRAM runs at a speed of 456 cycles per byte. The low memory versions use 226 bytes of SRAM at a speed of 517 cycles per byte and 164 bytes of SRAM at 738 cycles per byte.
Keywords: Grøstl, efficient, ATmega163, 8-bit, microcontroller, implementation

1 Introduction

In this work we present an efficient implementation of the SHA-3 candidate hash function Grøstl-256 [GKM+08] for the 8-bit AVR microcontroller ATmega163. Grøstl-256 is an iterated hash function using 512-bit message blocks and a 512-bit chaining value. Grøstl is based on similar design principles as the block cipher AES [DR02] and uses two permutations, each operating on a 512-bit state. The output of Grøstl-256 is a 256-bit hash value, generated by an output transformation.

The ATmega163 is an 8-bit microcontroller with 32 8-bit multi-purpose registers, 1024 Bytes of SRAM and 16K of flash memory, so a speed and memory efficient algorithm is the target for this platform. Because of the RISC architecture the ATmega163 is fast when operating on registers, but the data transfer to and from the SRAM is a rather slow operation. To achieve high speed implementations we have to minimize memory access and use values loaded in registers as efficiently as possible.

Our target is to get a high speed version to perform better than 500 cycles per byte without a limitation of SRAM usage, and a low memory version using significantly less SRAM with still reasonable speed, by optimizing the permutations and especially MixBytes as the most computing intensive part.

The thesis is organised as follows. In Section 2, we will give a short introduction to Grøstl. In Section 3, we present possible ways to increase the speed of the algorithm on an

8-bit microcontroller and show implementation details of Grøstl-256. In Section 4, we will show a performance analysis of the implementations. Finally, we conclude in Section 5.

2 Specification of Grøstl

In this section, we will explain briefly how Grøstl-256 works. For a more detailed definition and for other variants see [GKM+08]. In the following document all algorithms and occurences of "Grøstl" refer to Grøstl-256 only.

Generally Grøstl-n is the variant returning an n-bit hash value. Grøstl-256 will therefore return a 256-bit hash value. Grøstl-256 iterates a compression function that maps two 512-bit input values to a 512-bit output. The input of the compression function is a message block $m_{i=0,...,t}$[1] and a 512-bit chaining value initialized with the initialization vector $IV = \{00, ..., 00, 01, 00\}$. These message blocks will then be processed sequentially along with the chaining value:

$$h_{i+1} \leftarrow f(h_i, m_i) \text{ for } i = 0, ..., t-1 \text{ with } h_0 = IV = \{00, ..., 00, 01, 00\}.$$

After all message blocks have been processed, an output transformation Ω (see Figure 2.1) is applied:

$$H(M) = \Omega(h_t),$$

where the output of Ω is the final 256-bit hash value.

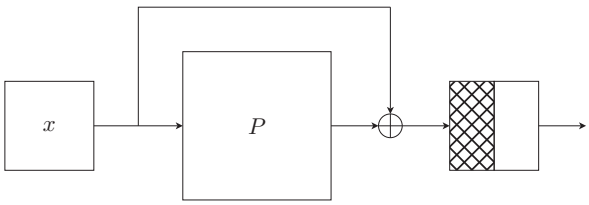

Figure 2.1: The output transformation Ω computes $P(x) \oplus x$ and then returns only the last 256 bits. P will be described in Section 2.2.

2.1 Compression Function

The compression function f is defined as follows:

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h,$$

where P and Q are 512-bit permutations as described in Section 2.2. Figure 2.2 illustrates the construction of f.

[1]The message is padded and split into 512-bit message blocks.

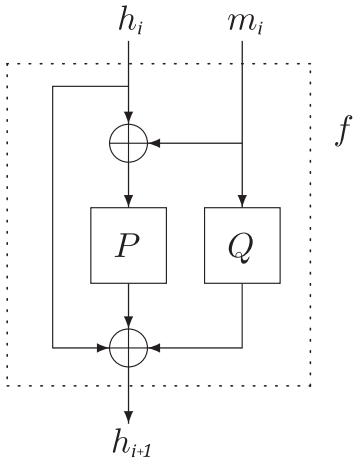

Figure 2.2: The compression function f.

2.2 Permutation Functions P and Q

The Permutation functions P and Q are the most computing intensive part of the Grøstl hash function. Both P and Q consist of 10 rounds. Similar to AES in each round the following four round transformations will be performed:

- AddRoundConstant
- SubBytes
- ShiftBytes
- MixBytes

All transformations operate on an 8x8 state matrix, which is four times the size of an AES state matrix (4x4). The matrix is mapped to a 64-byte (512-bit) sequence as:

$$\underbrace{\{00, 01, ..., 3e, 3f\}}_{64-byte-sequence} \Leftrightarrow \begin{pmatrix} 00 & 08 & 10 & 18 & 20 & 28 & 30 & 38 \\ 01 & 09 & 11 & 19 & 21 & 29 & 31 & 39 \\ 02 & 0a & 12 & 1a & 22 & 2a & 32 & 3a \\ 03 & 0b & 13 & 1b & 23 & 2b & 33 & 3b \\ 04 & 0c & 14 & 1c & 24 & 2c & 34 & 3c \\ 05 & 0d & 15 & 1d & 25 & 2d & 35 & 3d \\ 06 & 0e & 16 & 1e & 26 & 2e & 36 & 3e \\ 07 & 0f & 17 & 1f & 27 & 2f & 37 & 3f \end{pmatrix}$$

2.2.1 AddRoundConstant

AddRoundConstant adds (exclusive-or) a constant to the state matrix A. P and Q have different, round-dependent constants. The constants used in P and Q are

$$C_P[i] = \begin{pmatrix} i & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \end{pmatrix} \text{ and } C_Q[i] = \begin{pmatrix} 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ 00 & 00 & ... & 00 \\ i \oplus \text{0xFF} & 00 & ... & 00 \end{pmatrix},$$

where $i = 0, ..., 9$ is the number of the current round. AddRoundConstant updates the state A as

$$A \leftarrow A \oplus C[i].$$

2.2.2 SubBytes

SubBytes substitutes each byte in the state matrix by the corresponding value of the Rijndael (AES) S-box S. The S-box is a table of 256 bytes as shown in Table A.1. This substitution is the only non-linear transformation of the permutation P and Q.

$$a_{i,j} \leftarrow S(a_{i,j}),\ 0 \leq i < 8,\ 0 \leq j < 8.$$

2.2.3 ShiftBytes

For Grøstl-256 ShiftBytes moves all bytes in row i of the state matrix to the left by i positions. See Figure 2.3.

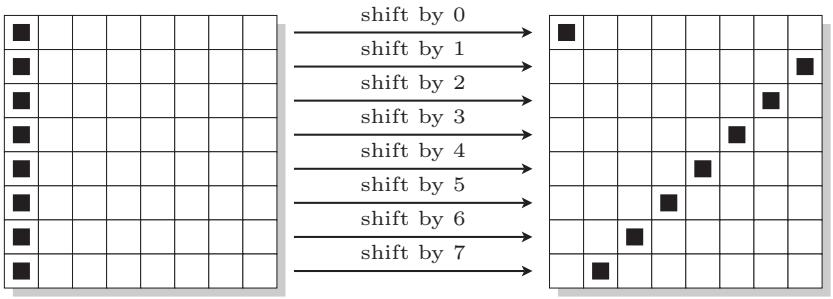

Figure 2.3: ShiftBytes. [GKM+08]

2.2.4 MixBytes

The MixBytes transformation is a matrix multiplication performed on the state matrix A as follows:

$$A \leftarrow B \times A,$$

where B is a circulant MDS matrix specified as $B = circ(02, 02, 03, 04, 05, 03, 05, 07)$ or as matrix:

$$B = \begin{pmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{pmatrix}.$$

See Figure 2.4. Multiplying with an maximum distance matrix creates diffusion in the

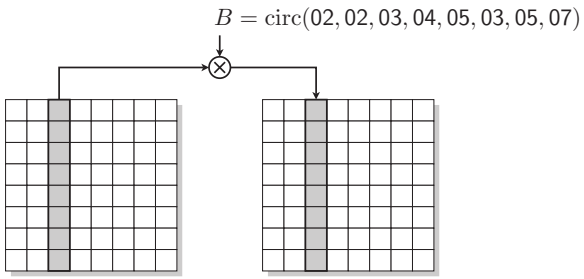

Figure 2.4: MixBytes left-multiplies each column of the state matrix by the circulant MDS matrix B. [GKM+08]

state matrix, *e.g.* a change in only one byte will result in a change of all 8 bytes of the column after MixBytes.

The multiplication is performed in a finite field $\mathbb{F}_{256}$ with the polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ (0x11B). This means that *e.g.* a multiplication by 2 can be implemented as follows:

- value is shifted left by one byte (*2)
- if an overflow occurs (carry set) we have to reduce with the polynomial (0x11B) to get a value inside the finite field
- we only need to XOR with 0x1B because we operate on 8-bit values
- $x = a \ll 1$
  if carry is 1: $x = x \oplus$ 0x1B

3 Implementation Details

In this section we will show how we efficiently implement Grøstl on the 8-bit microcontroller ATmega163. There will be three versions, one high speed and two with a low memory footprint.

3.1 The ATmega163

The ATmega163 is an 8-bit microcontroller with 32 8-bit multi-purpose registers, 1024 Bytes of SRAM and 16K of flash memory. The multi-purpose registers can be used to manipulate data. The controller needs 2 cycles to read from and write to the SRAM and 3 cycles to read from flash memory. Six of the 8-bit registers are used as 16-bit address registers X, Y and Z, thus they can usually not be used for computations.

The instructions we refer to in this work are listed with description and needed computational time in Table 3.1. Some of these instructions, *i.e.* **LDI**, can only operate on the upper 16 registers. Therefore we try to use the lower 16 registers only for simple calculations, as exclusive or.

Command	Description	Cycles
LD	load from SRAM to register	2
LDD	load from SRAM with offset to register	2
LDI	load value directly to regsiter	1
LPM	load from flash memory to register	3
ST	store from register to SRAM	2
STD	store from register to SRAM with offset	2
EOR	Exclusive OR 2 registers	1
MOV	copy between registers	1
LSL	shift left, over carry	1
IN	load from port	1
SBRC	skip if bit in register cleared	1/2/3

Table 3.1: Assembler commands used in this work.

The only special purpose register important for this work is the status register at port 0x3F, which holds information about the last instruction, such as the carry or zero flag.

3.2 Lookup Tables

For the SubBytes and MixBytes transformations we use lookup tables. As the SRAM is addressed by two registers, we have to place the lookup tables at memory positions we can easily address with the low byte set to the value to look up. Due to the fact that the SRAM address space of the ATmega163 begins at 0x60, we use the first memory blocks for the three states with 0x40 (64) bytes each, thus starting at 0x60, 0xA0 and 0xE0. The next free memory is now at 0x120. To use this address as the start address of the first lookup table, we use the **LDD** command with an offset of 0x20 instead of a simple **LD**. This way we can set the low address register to the value to look up without any further calculations, and load the value from the address + 0x20 without additional costs.

With three lookup tables in SRAM all the memory from 0x60 to 0x41F is used, leaving 64 bytes free for stack and other data.

For the high-speed version we initially move the three lookup tables to the SRAM. In the low-memory versions the lookup tables will reside in flash memory resulting in a memory usage of only 192 bytes for the states.

Instead of using lookup tables for calculating the multiplication by 2 and 4 we can use a macro using the algorithm described in section 2.2.4 for the multiplication by 2. This avoids using 512 bytes of memory for lookup tables that can be computed with little more cycles, which is interesting for low memory versions. See Listing 3.1 for the macro.

Listing 3.1 Computation of multiplication by 2. Cycles needed: 4 (see Table 3.1)

```
.macro MUL2 val, x1b, tmp
  ; val = val * 2
  ; x1b must be 0x1B
  LSL \val          ; shift left into carry
  IN  \tmp, 0x3F    ; load status register; we can't branch inside a macro
  SBRC \tmp, 0      ; check carry bit, skip if 0
  EOR \val, \x1b    ; add 0x1B if carry was 1
.endm
```

In summary, we need 2 cycles for a lookup in SRAM, 3 cycles for a lookup in flash memory and 4 cycles to compute the multiplication by 2 without lookup table.

3.3 Implementation

As the speed critical parts of Grøstl are the permutations P and Q. We will mostly cover details of their implementation in this work.

3.3.1 Permutation

The permutation consists of 10 rounds. The rounds are processed in a loop counting from 0 to 10 for P and from 255 to 245 for Q. This way we can omit adding 0xFF to the round constant for Q. To distinguish between P and Q we check if the most significant bit of the round number is set.

Because of the ShiftBytes operation, a column-wise application of MixBytes is not possible on a single state matrix. The diagonal of the state matrix will be the first column of the resulting matrix. If we now overwrite the first column of the current state matrix, the values of the remaining columns would be wrong. Therefore we use a second matrix to store the result of the first round. The second round will now read the state matrix from the temporary matrix and store the results back into the first matrix. After 10 rounds the result of the final round will be stored in the initial matrix.

To access the particular elements of the matrizes we use the **LDD** and **STD** (see Table 3.1) commands with an offset value. Both commands only take 2 cycles and can be used with the address registers **Y** and **Z**. After each round **Y** and **Z** are swapped.

The order of the transformations in this implementation is slightly different from the Grøstl design. ShiftBytes is performed before the others to have the matrix split into

columns as soon as possible. This has no effect, as AddRoundConstant, SubBytes and MixBytes are still in the correct order.

We have chosen to align the states in memory the same way as the matrix looks like, so one line in the state matrix is a continuous 8-byte block in memory, this decision does not affect the performance.

3.3.2 Message Injection

The permutations P and Q work on states that have to be initialized first. For Q the message is loaded from flash memory into SRAM as shown in Section 2.2. For P the message is loaded and H is added ($\oplus$). Thus we always need to keep the matrix H in SRAM.

The message injection could be further improved by loading the message column wise from flash before computing MixBytes instead of loading the whole block into SRAM before the permutation. Doing this is a very complex task, especially because of the padding in the last message block, and will significantly increase code size.

3.3.3 ShiftBytes

ShiftBytes is implemented by reading diagonal values from the input matrix to result in a column of the output matrix. Hereby the values loaded in the registers are already one column ready for SubBytes and MixBytes.

Listing 3.2 Principle of ShiftBytes for the first column of the output matrix.

```
LDD A0, Z+0x00   ; load value at (0,0)
LDD A1, Z+0x09   ; load value at (1,1)
LDD A2, Z+0x12   ; load value at (2,2)
LDD A3, Z+0x1B   ; load value at (3,3)
LDD A4, Z+0x24   ; load value at (4,4)
LDD A5, Z+0x2D   ; load value at (5,5)
LDD A6, Z+0x36   ; load value at (6,6)
LDD A7, Z+0x3F   ; load value at (7,7)
...              ; compute MixBytes for this column
STD Y+0x00, B0   ; store result to (0,0)
STD Y+0x08, B1   ; store result to (1,0)
STD Y+0x10, B2   ; store result to (2,0)
STD Y+0x18, B3   ; store result to (3,0)
STD Y+0x20, B4   ; store result to (4,0)
STD Y+0x28, B5   ; store result to (5,0)
STD Y+0x30, B6   ; store result to (6,0)
STD Y+0x38, B7   ; store result to (7,0)
```

3.3.4 AddRoundConstant

AddRoundConstant is done right after reading the values from SRAM. This is the only operation that differs for P and Q. For P the top-left ($a_{0,0}$) value is exclusive-ored with the current round number. For Q the bottom-left ($a_{7,0}$) value is exclusive-ored with the round number and **0xFF**. Because of that, the round numbers for Q will be decreased from **0xFF** to **0xF6** to save an additional **EOR** operation.

3.3.5 SubBytes

For SubBytes the value to be looked up in the S-box is loaded into a low address register, the high register is set to the address of the S-box in the SRAM. The resulting value can then simply be loaded with **LD** (see Table 3.1).

Listing 3.3 Principle of SubBytes for one column. +0x20 as described in Section 3.2

```
LDI XH, SBOX_H   ; 1 cycle   ; load high value into address register (once!)
MOV XL, C0       ; 1 cycle   ; Lookup 1 ; load low value into address register
LDD A0, X+0x20       ; 2 cycles ; A0 = S(C0)
MOV XL, C1       ; 1 cycle   ; Lookup 2
LDD A1, X+0x20       ; 2 cycles ; A1 = S(C1)
...
```

Listing 3.4 Example of a combined ShiftBytes, SubBytes operation. This will be called 640 times per permutation.

```
LDD XL, Z+0x37   ; 2 cycles ; load value with offset into address register
LDD A6, X+0x20       ; 2 cycles ; load the corresponding S-box value
```

Listing 3.5 Example of a combined ShiftBytes, AddRoundConstant and SubBytes operation, resulting in 10 extra cycles per permutation for the round constant.

```
LDD XL, Z+0x38   ; 2 cycles ; load value with offset into address register
EOR XL, ROUND    ; 1 cycle  ; xor with round number
                            ; (this is only necessary for one of the 64 values)
LDD A7, X+0x20   ; 2 cycles ; load the corresponding S-box value
```

3.3.6 MixBytes

MixBytes is the most complex operation, and thus has the biggest speed-up potential. We implement MixBytes as a operation working on columns to fully use the registers we have. As MixBytes is a matrix multiplication, every column can be processed seperately.

The MixBytes column operation will be called 80 times in one permutation. Although the read/write operations are called 640 times per permutation we can not optimize them further (4 cycles compared to more than 100 for one MixBytes column, see Listing 3.4). Therefore we have to minimize the cycles used by the MixBytes operation as far as possible.

At first we reduce the values of the matrix B to multipliers that can be easily calculated without the need for a multiplication operator. As the highest value in B is 7 we can split into additions ($\oplus$) with multipliers 1, 2 and 4. The multiplications by 2 and 4 are stored in lookup tables.

The split-up of the MixBytes column multiplication is shown in Listing 3.6. All input values are needed for each result of the output column. Therefore we calculate the lines simultaneously, using the values we already have loaded in the registers as often as possible.

Listing 3.6 Split-up of the MixBytes column multiplication. $a_0, ..., a_7$ are the current values of the column. $b_0, ..., b_7$ are the values of the column after MixBytes. + means $\oplus$.

$$
\begin{aligned}
b_0 &= 2a_0 + 2a_1 + 3a_2 + 4a_3 + 5a_4 + 3a_5 + 5a_6 + 7a_7 \\
b_1 &= 7a_0 + 2a_1 + 2a_2 + 3a_3 + 4a_4 + 5a_5 + 3a_6 + 5a_7 \\
b_2 &= 5a_0 + 7a_1 + 2a_2 + 2a_3 + 3a_4 + 4a_5 + 5a_6 + 3a_7 \\
b_3 &= 3a_0 + 5a_1 + 7a_2 + 2a_3 + 2a_4 + 3a_5 + 4a_6 + 5a_7 \\
b_4 &= 5a_0 + 3a_1 + 5a_2 + 7a_3 + 2a_4 + 2a_5 + 3a_6 + 4a_7 \\
b_5 &= 4a_0 + 5a_1 + 3a_2 + 5a_3 + 7a_4 + 2a_5 + 2a_6 + 3a_7 \\
b_6 &= 3a_0 + 4a_1 + 5a_2 + 3a_3 + 5a_4 + 7a_5 + 2a_6 + 2a_7 \\
b_7 &= 2a_0 + 3a_1 + 4a_2 + 5a_3 + 3a_4 + 5a_5 + 7a_6 + 2a_7 \\
\Leftrightarrow & \\
b_0 &= a_2 + a_4 + a_5 + a_6 + a_7 + 2a_0 + 2a_1 + 2a_2 + 2a_5 + 2a_7 + 4a_3 + 4a_4 + 4a_6 + 4a_7 \\
b_1 &= a_0 + a_3 + a_5 + a_6 + a_7 + 2a_0 + 2a_1 + 2a_2 + 2a_3 + 2a_6 + 4a_0 + 4a_4 + 4a_5 + 4a_7 \\
b_2 &= a_0 + a_1 + a_4 + a_6 + a_7 + 2a_1 + 2a_2 + 2a_3 + 2a_4 + 2a_7 + 4a_0 + 4a_1 + 4a_5 + 4a_6 \\
b_3 &= a_0 + a_1 + a_2 + a_5 + a_7 + 2a_0 + 2a_2 + 2a_3 + 2a_4 + 2a_5 + 4a_1 + 4a_2 + 4a_6 + 4a_7 \\
b_4 &= a_0 + a_1 + a_2 + a_3 + a_6 + 2a_1 + 2a_3 + 2a_4 + 2a_5 + 2a_6 + 4a_0 + 4a_2 + 4a_3 + 4a_7 \\
b_5 &= a_1 + a_2 + a_3 + a_4 + a_7 + 2a_2 + 2a_4 + 2a_5 + 2a_6 + 2a_7 + 4a_0 + 4a_1 + 4a_3 + 4a_4 \\
b_6 &= a_0 + a_2 + a_3 + a_4 + a_5 + 2a_0 + 2a_3 + 2a_5 + 2a_6 + 2a_7 + 4a_1 + 4a_2 + 4a_4 + 4a_5 \\
b_7 &= a_1 + a_3 + a_4 + a_5 + a_6 + 2a_0 + 2a_1 + 2a_4 + 2a_6 + 2a_7 + 4a_2 + 4a_3 + 4a_5 + 4a_6
\end{aligned}
$$

In the implementation we split MixBytes into three parts, first computing all values with multiplier 1, then add ($\oplus$) all values with multiplier 2 and finally all values with multiplier 4. This way the MixBytes operation for one column takes only 113 cycles $(31 + 33 + 49)$ compared to more than 300 cycles in a straight forward implementation. To have a better view, we show the values of the source column used for the values of the target column in a table for each multiplier in Table 3.2.

To save computing time we look for combinations of values that occur more than once. In Table 3.2 those values are already marked with numbers. For each of these numbers the combined value is computed once and then added to all target values containing it. The values marked with letters can be added after using the numbered combinations. See Listing 3.7.

For the computation of the values with multiplier 2 we use the fact that the matrix for multiplier 2 in Table 3.2 is a shifted version of the matrix for multiplier 1. We can see that

	$1 * a_0$	$1 * a_1$	$1 * a_2$	$1 * a_3$	$1 * a_4$	$1 * a_5$	$1 * a_6$	$1 * a_7$
$b_{0,1}$	-	-	$\bullet^{0}$	-	$\bullet^{2}$	$\bullet^{0}$	$\bullet^{2}$	$\bullet$
$b_{1,1}$	$\bullet^{1}$	-	-	$\bullet^{1}$	-	$\bullet$	$\bullet^{a}$	$\bullet$
$b_{2,1}$	$\bullet^{b}$	$\bullet^{3}$	-	-	$\bullet^{2}$	-	$\bullet^{2}$	$\bullet^{3}$
$b_{3,1}$	$\bullet^{b}$	$\bullet^{3}$	$\bullet^{0}$	-	-	$\bullet^{0}$	-	$\bullet^{3}$
$b_{4,1}$	$\bullet^{1}$	$\bullet$	$\bullet$	$\bullet^{1}$	-	-	$\bullet^{a}$	-
$b_{5,1}$	-	$\bullet^{3}$	$\bullet$	$\bullet$	$\bullet$	-	-	$\bullet^{3}$
$b_{6,1}$	$\bullet^{1}$	-	$\bullet^{0}$	$\bullet^{1}$	$\bullet$	$\bullet^{0}$	-	-
$b_{7,1}$	-	$\bullet$	-	$\bullet$	$\bullet^{2}$	$\bullet$	$\bullet^{2}$	-

(a) Multiplier 1.

	$2 * a_0$	$2 * a_1$	$2 * a_2$	$2 * a_3$	$2 * a_4$	$2 * a_5$	$2 * a_6$	$2 * a_7$	$=$
$b_{0,2}$	$\bullet$	$\bullet$	$\bullet$	-	-	$\bullet$	-	$\bullet$	$2 * b_{3,1}$
$b_{1,2}$	$\bullet$	$\bullet$	$\bullet$	$\bullet$	-	-	$\bullet$	-	$2 * b_{4,1}$
$b_{2,2}$	-	$\bullet$	$\bullet$	$\bullet$	$\bullet$	-	-	$\bullet$	$2 * b_{5,1}$
$b_{3,2}$	$\bullet$	-	$\bullet$	$\bullet$	$\bullet$	$\bullet$	-	-	$2 * b_{6,1}$
$b_{4,2}$	-	$\bullet$	-	$\bullet$	$\bullet$	$\bullet$	$\bullet$	-	$2 * b_{7,1}$
$b_{5,2}$	-	-	$\bullet$	-	$\bullet$	$\bullet$	$\bullet$	$\bullet$	$2 * b_{0,1}$
$b_{6,2}$	$\bullet$	-	-	$\bullet$	-	$\bullet$	$\bullet$	$\bullet$	$2 * b_{1,1}$
$b_{7,2}$	$\bullet$	$\bullet$	-	-	$\bullet$	-	$\bullet$	$\bullet$	$2 * b_{2,1}$

(b) Multiplier 2.

	$4 * a_0$	$4 * a_1$	$4 * a_2$	$4 * a_3$	$4 * a_4$	$4 * a_5$	$4 * a_6$	$4 * a_7$
$b_{0,4}$	-	-	-	$\bullet^{0}$	$\bullet^{1}$	-	$\bullet^{0}$	$\bullet^{1}$
$b_{1,4}$	$\bullet^{2}$	-	-	-	$\bullet^{1}$	$\bullet^{2}$	-	$\bullet^{1}$
$b_{2,4}$	$\bullet^{2}$	$\bullet^{3}$	-	-	-	$\bullet^{2}$	$\bullet^{3}$	-
$b_{3,4}$	-	$\bullet^{3}$	$\bullet^{4}$	-	-	-	$\bullet^{3}$	$\bullet^{4}$
$b_{4,4}$	$\bullet^{5}$	-	$\bullet^{4}$	$\bullet^{5}$	-	-	-	$\bullet^{4}$
$b_{5,4}$	$\bullet^{5}$	$\bullet^{6}$	-	$\bullet^{5}$	$\bullet^{6}$	-	-	-
$b_{6,4}$	-	$\bullet^{6}$	$\bullet^{7}$	-	$\bullet^{6}$	$\bullet^{7}$	-	-
$b_{7,4}$	-	-	$\bullet^{7}$	$\bullet^{0}$	-	$\bullet^{7}$	$\bullet^{0}$	-

(c) Multiplier 4.

Table 3.2: Column-wise MixBytes operation for each multiplier. $a_0, ..., a_7$ are source values, $b_i = b_{i,1} \oplus b_{i,2} \oplus b_{i,4}$ are target values.

e.g. the line b_0 for multiplier 1 is the same as b_5 for multiplier 2. So we multiply b_0 by 2 and get the value we have to add to b_5. To avoid using 8 registers for the multiplied values we copy a value (*e.g.* b_0) into the address register to multiply it into a temporary register, then copy the target value (*e.g.* b_5) into the address register so we can add the temporary value to the register of the target value (*e.g.* b_5). See Listing 3.8 for an illustration.

For the computation of the values with multiplier 4 we return using the scheme of

multiplier 1. We compute combined values, multiply them by 4 and add them to the target values. In contrast to the values of multiplier 1 we do not have any single values not covered by the combinations, as there are always 4 source values for one target value. See Listing 3.9.

The cycles needed for each of the multipliers is as follows: 31 cycles for $\cdot 1$, 33 cycles for $\cdot 2$ and 49 cycles for $\cdot 4$. As we can see, the multiplier 4 is the most computing intensive part, because the possible combinations do not improve the speed. For a speed improvement a combination has to use at least three source or target values.

The approach, first multiplying the column values by 2 and then again by 2 to get the values multiplied by 4 is not as fast as the presented way to compute MixBytes. The computation of multiplier 1 would be the same, needing 31 cycles, then 25[2] cycles are needed to compute $2 * a_{0,\ldots,7}$, then again 31 cycles for the computation of multiplier 2, 25 cycles for $2 * (2 * a_{0,\ldots,7})$ and finally 32 cycles for the computation of multiplier 4. So a total of 144 cycles would be needed.

The register alignment of the values of MixBytes is shown in Table 3.3. $c_{0,\ldots,6}$ are used to store the diagonale for the next round.

[2] 1 cycle for setting the high byte of the lookup table, 8 times: 1 cycle for setting the low byte, 2 cycles for loading the table entry.

register	function
0	c_0
1	c_1
2	a_0
3	a_1
4	a_2
5	a_3
6	a_4
7	a_5
8	a_6
9	a_7
10	b_0
11	b_1
12	b_2
13	b_3
14	b_4
15	b_5
16	b_6
17	b_7
18	round constant
19	temp
20	c_2
21	c_3
22	c_4
23	c_5
24	c_6
25	temp
26	address register XL
27	address register XH
28	address register YL
29	address register YH
30	address register ZL
31	address register ZH

Table 3.3: Register alignment for MixBytes.

Listing 3.7 Computation of MixBytes for multiplier 1. $A0, ..., A1$ are input, $B0, ..., B1$ output, as described in Table 3.2. Cycles needed: 31 (see Table 3.1)

```
MOV B0, A2   ; calculate (0) / initial b0
EOR B0, A5

MOV B1, A0   ; calculate (1) / initial b1
EOR B1, A3

MOV B2, A4   ; calculate (2) / initial b2
EOR B2, A6

MOV B3, A1   ; calculate (3) / initial b3
EOR B3, A7

MOV B6, B0   ; copy (0) to b6
EOR B6, B1   ; add  (1) to b6
EOR B6, A4   ; add remaining a4 to b6

EOR B1, A6   ; add (a) to (1)

MOV B4, B1   ; copy (1) to b4
EOR B4, A1   ; add remaining a1 and a2 to b4
EOR B4, A2

EOR B1, A5   ; add remaining a5 and a7 to b1
EOR B1, A7

MOV B5, B3   ; copy (3) to b5
EOR B5, A2   ; add remaining a2, a3 and a4 to b5
EOR B5, A3
EOR B5, A4

MOV B7, B2   ; copy (2) to b7
EOR B7, A1   ; add remaining a1, a3 and a5 to b7
EOR B7, A3
EOR B7, A5

EOR B0, B2   ; add (2) to b0
EOR B0, A7   ; add remaining a7 to b0

EOR B3, A0   ; add (b) to (3)

EOR B2, B3   ; add (3) to b2

EOR B3, A2   ; add remaining a2 and a5 to b3
EOR B3, A5
```

Listing 3.8 Computation of MixBytes for multiplier 2. This is the continuation to Listing 3.7. Cycles needed: 33 (see Table 3.1)

```
LDI ZH, MUL2_H      ; load the high address of the mul2 lookup table

MOV ZL, B0          ; load b0 into the low address
LDD TMP1, Z+0x20    ; load from lookup table
                    ; tmp1 = b0 * 2
MOV ZL, B5
LDD TMP2, Z+0x20    ; tmp2 = b5 * 2
EOR B5, TMP1        ; b5 XOR tmp1

MOV ZL, B2
LDD TMP1, Z+0x20    ; tmp1 = b2 * 2
EOR B2, TMP2        ; b2 XOR tmp2

MOV ZL, B7
LDD TMP2, Z+0x20    ; tmp2 = b7 * 2
EOR B7, TMP1        ; b7 XOR tmp1

MOV ZL, B4
LDD TMP1, Z+0x20    ; tmp1 = b4 * 2
EOR B4, TMP2        ; b4 XOR tmp2

MOV ZL, B1
LDD TMP2, Z+0x20    ; tmp2 = b1 * 2
EOR B1, TMP1        ; b1 XOR tmp1

MOV ZL, B6
LDD TMP1, Z+0x20    ; tmp1 = b6 * 2
EOR B6, TMP2        ; b6 XOR tmp2

MOV ZL, B3
LDD TMP2, Z+0x20    ; tmp2 = b3 * 2
EOR B3, TMP1        ; b3 XOR tmp1

EOR B0, TMP2        ; b0 XOR tmp2
```

Listing 3.9 Computation of MixBytes for multiplier 4. This is the continuation to Listing 3.8. Cycles needed: 49 (see Table 3.1)

```
LDI ZH, MUL4_H ; load the high address of the mul4 lookup table

MOV ZL, A4          ; compute (1)
EOR ZL, A7
LDD TMP1, Z+0x20 ; multiply (1) by 4
EOR B0, TMP1        ; add (1) to b0
EOR B1, TMP1        ; add (1) to b1

MOV ZL, A3          ; compute (0)
EOR ZL, A6
LDD TMP1, Z+0x20 ; multiply (0) by 4
EOR B0, TMP1        ; add (0) to b0
EOR B7, TMP1        ; add (0) to b7

MOV ZL, A0          ; compute (2)
EOR ZL, A5
LDD TMP1, Z+0x20 ; multiply (2) by 4
EOR B1, TMP1        ; add (2) to b1
EOR B2, TMP1        ; add (2) to b2

MOV ZL, A1          ; compute (3)
EOR ZL, A6
LDD TMP1, Z+0x20 ; multiply (3) by 4
EOR B2, TMP1        ; add (3) to b2
EOR B3, TMP1        ; add (3) to b3

MOV ZL, A2          ; compute (4)
EOR ZL, A7
LDD TMP1, Z+0x20 ; multiply (4) by 4
EOR B3, TMP1        ; add (4) to b3
EOR B4, TMP1        ; add (4) to b4

MOV ZL, A0          ; compute (5)
EOR ZL, A3
LDD TMP1, Z+0x20 ; multiply (5) by 4
EOR B4, TMP1        ; add (5) to b4
EOR B5, TMP1        ; add (5) to b5

MOV ZL, A1          ; compute (6)
EOR ZL, A4
LDD TMP1, Z+0x20 ; multiply (6) by 4
EOR B5, TMP1        ; add (6) to b5
EOR B6, TMP1        ; add (6) to b6

MOV ZL, A2          ; compute (7)
EOR ZL, A5
LDD TMP1, Z+0x20 ; multiply (7) by 4
EOR B6, TMP1        ; add (7) to b6
EOR B7, TMP1        ; add (7) to b7
```

4 Performance Analysis

In this section we will show how the implementations perform. This is the first implementation of Grøstl-256 on the 8-bit microcontroller ATmega163, therefore we can only analyze the different implemented versions, comparing speed and memory usage.

As there is very limited memory on the ATmega163 we have to find a tradeoff between memory usage and computation time. Therefore we decided to implement various versions with different memory requirements.

4.1 High Speed Version

The High Speed version uses three lookup tables. All lookup tables are copied from Flash into SRAM in the initialization phase. The Permute operation works on one state using a temporary state to allow directly integrating the ShiftBytes operation. For each round the values are loaded from the diagonale of one state and stored as column in the other. This way, as we still need to keep a state in memory for h. In total we need 192 bytes of SRAM for the states and additional 768 bytes for the three lookup tables.

To improve speed, everything, except the 10 rounds, is hard coded as macros, thus resulting in larger code size.

4.2 Low Memory (192) Version

The first of the two low memory versions uses 192 Bytes of SRAM, working completely the same way as the High Speed version except for the lookup tables that are now kept in flash memory all the time. For very small message sizes (1 block), this results in an even higher speed than the high speed version, because of less overhead for copying the lookup tables into SRAM.

4.3 Low Memory (128) Version

The version with the lowest memory footprint differs much more from the other two versions:

- permute works without a temporary state, doing ShiftBytes seperated from MixBytes
- $\cdot 2$ operation is implemented as macro, thus avoiding two of the three lookup tables (see Listing 3.1)
- the MixBytes column operation is implemented as function not as macro

With these memory improvements the code size reduces to 2080 bytes, SRAM usage to 128 bytes and Flash memory usage for the lookup table only 256 bytes.

4.4 Lower Memory Versions

In theory, versions using less than 128 bytes of SRAM are possible. To achieve such low memory requirements it is necessary to write states back to EEPROM or flash memory during runtime. With an estimated duration of the write process to EEPROM of 16000 cycles for 64 bytes, a Grøstl-256 implementation using only 64 bytes of SRAM would have an estimated performance of over 2000 cycles per byte, writing 4 states back to EEPROM for each message block. Due to the limited write/erase cycles of the EEPROM (100,000) this solution would only be capable of hashing about 1.6 megabytes before the EEPROM reaches end of life.

4.5 Comparison

In Table 4.1 we show the memory usage of the three implementations for SRAM and flash memory. As all implementations are based on a C program flow, we need to store registers on the stack in SRAM, adding a little SRAM usage for all versions.

	High Speed		Low Memory (192)		Low Memory (128)	
	SRAM	flash memory	SRAM	flash memory	SRAM	flash memory
Code		3460		3402		2080
Lookup Tables	768	768		768		256
States	192		192		128	
Stack	34		34		36	
Total	**994**	**4228**	**226**	**4170**	**164**	**2336**

Table 4.1: Memory usage of the AVR 8-bit Grøstl-256 implementations.

Speed was measured using three different messages: a short message with 447 bits, a long message with 5009 bits and a very long message with 34178 bits. These message sizes result in 1, 10 and 67 blocks. Using such messages shows how the different parts of Grøstl; initialization, output transformation, and the split-up into blocks influences the overall cycles per byte performance. Table 4.2 shows the performance of the different implementations for the three messages, including a plain C implementation without optimization for comparison. Figure 4.1 shows the same data in a chart, the sawtooth form shows that there an almost empty block takes the same time to be processed as a full block.

Message size	High Speed	Low M. (192)	Low M. (128)	plain C	Unit
447 bits	51019	50529	71717	307253	cycles
	911	902	1281	5487	c./byte
5009 bits	310847	344914	492250	2084586	cycles
	496	550	785	3325	c./byte
34178 bits	1956665	2209612	3155860	13364141	cycles
	462	522	745	3154	c./byte
Calculated (overhead)	24994	21045	29623	125505	cycles
Calculated (per byte)	**456**	**517**	**738**	3125	c./byte
Speed at 8MHz	**17547**	**15487**	**10842**	2560	bytes/s.
Permutation	13202	15122	22130	87074	cycles
	413	473	692	2721	c./byte

Table 4.2: Speed of the AVR 8-bit Grøstl-256 implementations.

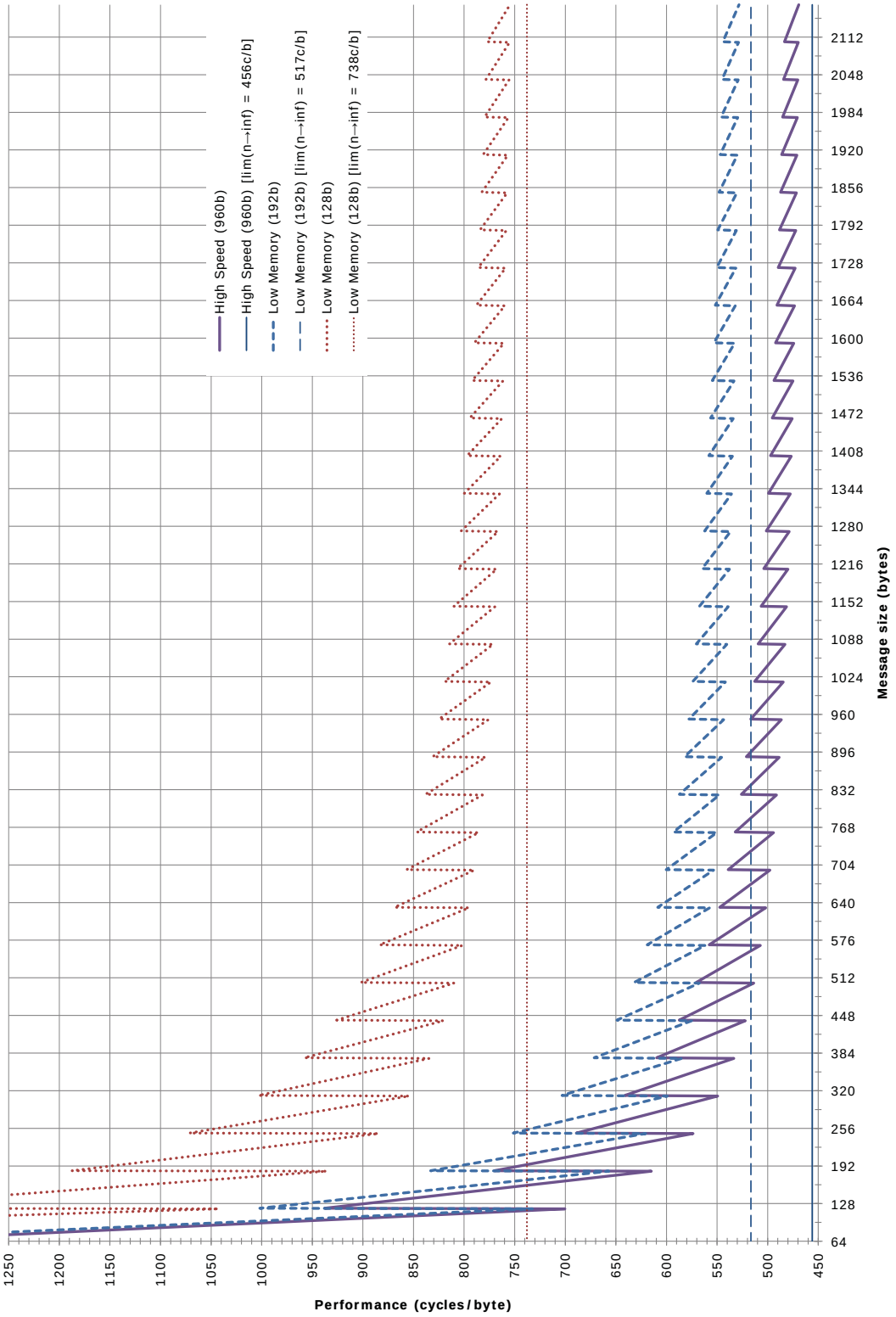

Figure 4.1: Performance of the AVR 8-bit Grøstl-256 implementations.

5 Conclusion

Of the three implemented versions, the low memory variants will be the most suitable variants to be used on a microcontroller. The limited memory is always a problem when using such small controllers, so for an hash function that will probably be used in addition to another program it is very important to keep the memory footprint as small as possible. We showed ways to do this while keeping the speed of Grøstl reasonable. For this it is very helpful that all parts of Grøstl operate on 8-bit values, as the controller does. The high speed version, using lookup tables in SRAM and macros for MixBytes, runs at a speed of 456 cycles per byte using 994 bytes of SRAM. The low memory version using lookup tables in flash runs at 517 cycles per byte using 226 bytes of SRAM. The lowest memory version uses only the s-box lookup table and performs ShiftBytes seperated from MixBytes to avoid using a temporary state im SRAM thus running at 738 cycles per byte using 164 bytes of SRAM.

A Tables

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table A.1: The Rijndael S-box.

References

[Atm03] Atmel. 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash. ATmega163. Available online at `http://www.atmel.com/`, 2003.

[BKH07] Stefan Berger, Robert Könighofer, and Christoph Herbst. Eine 8-bit Highspeed Softwareimplementierung von Whirlpool. In Patrick Horster, editor, *DACH Security 2007*, IT security & IT management, pages 459 – 470. Syssec, 2007.

[DR02] Joan Daemen and Vincent Rijmen. The Design of Rijndael: AES - The Advanced Encryption Standard. Springer, 2002.

[GKM+08] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl a SHA-3 candidate. Available online at `http://www.groestl.info/`, October 2008.