Optimized Implementation of SHA-512 for 16-bit MSP430 Microcontrollers

Christian Franck and Johann Großschädl

University of Luxembourg, Department of Computer Science,
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{christian.franck,johann.groszschaedl}@uni.lu

Abstract. The enormous growth of the Internet of Things (IoT) in the recent past has fueled a strong demand for lightweight implementations of cryptosystems, i.e. implementations that are efficient enough to run on resource-limited devices like sensor nodes. However, most of today's widely-used cryptographic algorithms, including the AES or the SHA2 family of hash functions, were already designed some 20 years ago and did not take efficiency in restricted environments into account. In this paper, we introduce implementation options and software optimization techniques to reduce the execution time of SHA-512 on 16-bit MSP430 microcontrollers. These optimizations include a novel register allocation strategy for the 512-bit hash state, a fast "on-the-fly" message schedule with low RAM footprint, special pointer arithmetic to avoid the need to copy state words, as well as instruction sequences for multi-bit rotation of 64-bit operands. Thanks to the combination of all these optimization techniques, our hand-written MSP430 Assembler code for the SHA-512 compression function reaches an execution time of roughly 40.6k cycles on an MSP430F1611 microcontroller. Hashing a message of 1000 bytes takes slightly below 338k clock cycles, which corresponds to a hash rate of about 338 cycles/byte. This execution time sets a new speed record for hashing with 256 bits of security on a 16-bit platform and improves the time needed by the fastest C implementations by a factor of 2.3. In addition, our implementation is extremely small in terms of code size (roughly 2.1k bytes) and has a RAM footprint of only 390 bytes.

Keywords: IoT security, lightweight cryptography, cryptographic hash function, MSP430 architecture, software optimization.

1 Introduction

A cryptographic hash function is an algorithm that maps data of arbitrary size and form to a fixed-size bit-string, typically between 160 and 512 bits, which is (under idealized assumptions) unique and can be seen as a "digest" or "digital fingerprint" of the data. Such algorithms play a crucial role in IT security and are used for a broad range of purposes, e.g. to verify the integrity of data, to serve as digest of data for digital signature schemes, to verify passwords, or to implement a proof-of-work for digital currencies [10]. In addition to these basic

applications, modern hash functions can also be used to construct e.g. Message Authentication Codes (MACs), eXtensible Output Functions (XOFs), Pseudo-Random Number Generators (PRNGs), and even stream ciphers. Amongst the most important and widely-used hash functions are the members of the SHA2 family, which have been adopted by the NIST and many other standardization bodies all around the world [7]. The SHA2 family consists of six hash functions altogether, which vary with respect to the digest lengths (ranging from 224 to 512 bits) and, consequently, provide different levels of security. SHA-512 is the "biggest" member of the family and especially important since it is part of the popular Edwards Curve Digital Signature Algorithm (EdDSA) [5].

The SHA-512 algorithm is based on the carefully-analyzed Merkle-Damgård structure [6,3] and uses a Davies-Meyer compression function [9] that consists of only Boolean operations (i.e. AND, OR, XOR, NOT), modular additions, as well as shifts and rotations. All these operations are applied to a 512-bit state arranged in the form of eight 64-bit words called *working variables*. Arithmetic and logical operations on 64-bit words are extremely efficient on modern high-end X64 processors, but can introduce a significant performance-bottleneck on 8 and 16-bit microcontrollers with a small register space and slow shift/rotate operations. Such resource-constrained platforms can only hold a fraction of the 512-bit state in registers (but never the entire state), which necessitates a large number of load and store operations to transfer working variables between the register file and RAM. In addition, all small 8 and 16-bit microcontrollers can only shift or rotate the content of a register by a single bit at a time, i.e. shifts or rotations by n bits take (at least) n clock cycles. The cycle count increases further when the data word to be shifted or rotated is too large to fit into one register, which is always the case when SHA-512 is implemented on processors with a word-size of less than 64 bits. Furthermore, most C compilers for small microcontrollers are not good at optimizing arithmetic or logical operations on 64-bit words because operands of such a length have hardly any application on 8/16-bit platforms apart from cryptography.

The massive growth of the Internet of Things (IoT) [4] in the past 10 years has created a strong interest in the question of how cryptographic algorithms can be optimized for resource-restricted microcontrollers and what performance highly-optimized implementations can achieve. An example for such platforms is the MSP430(X) series of 16-bit ultra-low-power microcontrollers from Texas Instruments [11]. MSP430 devices were among the first embedded platforms to be equipped with Ferro-electric Random Access Memory (FRAM), which is non-volatile (like Flash) but nonetheless offers high-speed read, write, and erase accesses (similar to SRAM). In addition, MSP430 microcontrollers have several low-power modes with fine-grain control over active components, making them suitable for battery-operated devices like wireless sensor nodes. The MSP430 is based on the von-Neumann memory model, which means code and data share a unified address space, and there is a single address bus and a single data bus that connects the microcontroller core with RAM, flash/ROM, and peripheral modules. Twelve of the 16 registers (each 16 bits wide) are available for general

use; the remaining four serve a special purpose. The MSP430 architecture has a minimalist instruction set consisting of only 27 core instructions that can be divided into three categories: double-operand instructions (which overwrite one of the operands with the result), single-operand instructions, and jumps.

In this paper, we present (to the best of our knowledge) the first optimized assembler implementation of SHA-512 for the MSP430(X) platform, which we developed from scratch with the goal to achieve a reasonable trade-off between fast execution time, small code size, and low memory consumption. The main data structure of our SHA-512 software is an efficient circular buffer based on a special memory alignment method and advanced pointer arithmetic. We also explain how we optimized the rotation of 64-bit words, and how we maximized the register usage (resp. minimized the number of memory accesses) in order to speed up the computation of the compression function. Though we describe all our optimization techniques in the context of SHA-512, they also facilitate the implementation of other members of the SHA2 family, and may even be useful for applications other than cryptographic hashing. We assess the performance of our software by comparing it with a number of optimized C implementations of SHA-512. This comparison indicates that our implementation is at least 2.3 times faster and requires less code size and RAM than its competitors.

2 SHA-512

SHA-512 is a member of the SHA2 suite of hash functions, which was designed by the National Security Agency (NSA) and first published in 2001. The SHA2 suite includes six hash functions in total, with digest sizes ranging from 224 to 512 bits. After standardization by the U.S. National Institute of Standards and Technology (NIST) [7] and various other standards bodies, the SHA2 suite has become widely used in practice and is now an integral building block of modern security protocols like SSL/TLS and IPSec. Another reason for the widespread deployment of the SHA2 suite is their excellent performance in software. As its name suggests, SHA-512 produces a digest of a length of 512 bits, which makes it the "biggest" member of the SHA2 suite. It has a block size of 1024 bits and can hash data of a length of up to 2^{128} bits (i.e. 2^{125} bytes). SHA-512, like all other members of the SHA2 suite, involves a padding so that the length of the data becomes a multiple of the block size. Finding a pair of colliding messages based on the birthday paradox requires about 2^{256} evaluations of SHA-512. On the other hand, finding a preimage (i.e. a message with a given hash value) has a time complexity of 2^{512}. In other words, SHA-512 provides 256 bits of security against collision attacks and 512 bits of security against preimage attacks.

SHA-512, as well as all other members of the SHA2 suite, is a Merkle-Damgård construction, which is a well-established way of designing a hash function from a one-way *compression function* [6,3]. A hash function built according to the Merkle-Damgård approach is provably resistant against collisions when the compression function is collision-resistant and an appropriate padding scheme is used. In other words, when following the Merkle-Damgård method, the problem

of designing a collision-resistant hash function for messages of any length boils down to designing a collision-resistant compression function for short blocks. In the SHA2 suite, the compression function is based on a block cipher according to the Davies-Meyer strategy [9], which means the message block to compress is fed as key to the block cipher, while the previous hash value is the plaintext to encrypt. The ciphertext generated by the block cipher is then XORed with the plaintext to produce the next hash value. Consequently, the block cipher of the compression function of SHA-512 has a key length of 1024 bits and a block size of 512 bits.

2.1 Preprocessing

According to [7], the SHA-512 hash function consists of two core parts: preprocessing and hash computation. The former includes the padding of the message and the initialization of the working variables to fixed values. Thereafter, the actual hash computation involves a message schedule and (iteratively) produces a sequence of hash values, the last of which forms the final digest.

SHA-512 takes as input a message M of a length of $l < 2^{128}$ bits, which is processed in blocks M_i with a fixed length of 1024 bits. At first, M is padded by appending a "1" bit followed by a certain number of "0" bits such that the overall bit-length becomes congruent to 896 modulo 1024 (i.e. when k denotes the number of "0" bits, the congruence relation $l + k + 129 \equiv 0 \bmod 1024$ has to hold). Then, l is appended as unsigned 128-bit integer (most significant byte first), which means that the last block of a padded message becomes 1024 bits long. Note that padding is always added, even if the length l of the unpadded message M is already a multiple of 1024. Consequently, it can happen that the padded message becomes one block longer than the unpadded message.

SHA-512 operates on a state of a length of 512 bits that holds intermediate results during the computation and also the final message digest. This state is organized in eight 64-bit working variables, usually referred to by the lowercase letters a, b, c, d, e, f, g, and h. At the beginning of the hash computation, the working variables are initialized to 64-bit integers, which are specified in [7] in big-endian format, i.e. the most-significant byte is placed at the lowest address (or leftmost byte position) of the word representing a working variable. These eight 64-bit integers were obtained by taking the first 64 bits of the fractional portions of the square roots of the first eight prime numbers.

2.2 Hash Computation

The most speed-critical part of SHA-512 is the computation of the compression function, which is an iterative process consisting of 80 rounds. Each round gets as input the set of working variables, a 64-bit word w_i that is derived from the message block to be compressed via the so-called *message schedule* (described below), and a 64-bit round constant k_i. These round constants are nothing else than the first 64 bits of the fractional portions of the cube roots of the first 80 prime numbers. At the end of each round, the set of eight working variables is

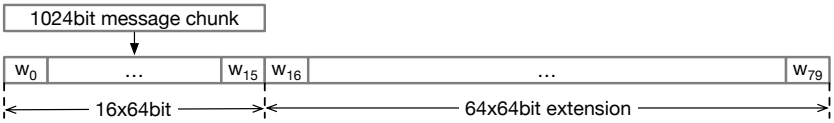


Fig. 1. SHA-512 message schedule (a 1024-bit block of the message contained in sixteen 64-bit words $w_0, \ldots, w_{15}$ is extended to 80 words $w_0, \ldots, w_{79}$).

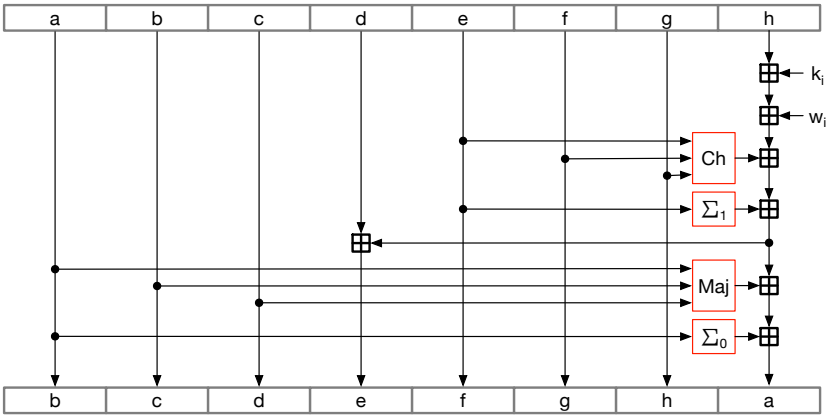


Fig. 2. Illustration of the SHA-512 round function showing how the working variables a to h are updated in every round.

updated. Following to the Davies-Meyer principle, the working variables at the end of the last (i.e. 80-th) round are XORed with the working variables at the beginning of the first round.

Message Schedule. As depicted in Fig. 1, the message schedule expands the 1024-bit message block M_i to 80 words w_i with $0 \leq i \leq 79$, each of which has a length of 64 bits. For the first 16 rounds, the 64-bit words w_0 to w_{15} are the same as the words of the 1024-bit block M_i of the message to be hashed. The remaining 64 words are computed according to the following equations.

$$
\begin{aligned}
w_i &= (\sigma_1(w_{i-2}) + w_{i-7} + \sigma_0(w_{i-15}) + w_{i-16}) \bmod 2^{64} \\
\sigma_0(w) &= (w \ggg 1) \oplus (w \ggg 8) \oplus (w \gg 7) \\
\sigma_1(w) &= (w \ggg 19) \oplus (w \ggg 61) \oplus (w \gg 6)
\end{aligned}
$$

Consequently, the word w_i for $16 \leq i \leq 79$ is derived from four preceding words of w_i, namely w_{i-2}, w_{i-7}, w_{i-15}, and w_{i-16}, whereby two of these four words are subjected to the functions $\sigma_0(.)$ and $\sigma_1(.)$. These "small sigma" functions consist of XOR operations, right-rotations (represented by the symbol $\ggg$), as well as right-shifts (represented by $\gg$).

Round Function. As shown in Fig. 2, the SHA-512 round function processes the eight working variables a, b, c, d, e, f, g, and h, using as additional inputs a word w_i of the message schedule and a round constant k_i. In each round, two of the eight working variables are updated through additions (modulo 2^{64}) in combination with the following four operations.

$$\begin{aligned} \Sigma_0(e) &= (e \ggg 28) \oplus (e \ggg 34) \oplus (e \ggg 39) \\ \Sigma_1(a) &= (a \ggg 14) \oplus (a \ggg 18) \oplus (a \ggg 41) \\ \mathrm{Ch}(e,f,g) &= (e \wedge f) \oplus (\bar{e} \wedge g) \\ \mathrm{Maj}(a,b,c) &= (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c) \end{aligned}$$

The two Σ operations ("big sigma") are very similar to the σ operations of the message schedule and consist of rotations and XORs. Ch (short for "choice") is a conditional operation where e determines whether a bit of f or a bit of g gets assigned to the output. On the other hand, Maj (short for "majority") assigns the majority of the three inputs bits a, b, c to the output, i.e. the output bit is "1" if at least two bits are "1". Finally, the values of the six working variables a, b, c, e, f, g are respectively copied to the working variables b, c, d, f, g, h.

3 Implementation and Optimization for MSP430

In the following, we explain the main design choices and optimizations that we made in order to obtain an efficient (i.e. fast) and "lightweight" (i.e. modest in terms of RAM and flash footprint) implementation of SHA-512 for MSP430.

Storage of 64-bit Words in Registers and RAM. Since the MSP430 has only 16-bit registers, the 64-bit words used by the SHA-512 algorithm have to be processed in "chunks" of 16 bits. As can be seen in Fig. 3, there are sixteen 16-bit registers in total (`R0` to `R15`), but only twelve of them (namely `R4` to `R15`) are general-purpose registers and can be freely used by the programmer. Since four 16-bit registers are necessary to store a single 64-bit word, at most three 64-bit words can be kept in registers at once, as depicted in Fig. 3. During the computation of the message expansion and the compression function, we often use the stack pointer `R1` to access 64-bit words in memory.

Depending on the used addressing mode, memory read and write operations (i.e. `MOV` instructions) can take up to seven clock cycles, which means they are relatively expensive compared to other architectures. Hence, memory accesses should be avoided as much as possible. For example, loading a 64-bit word from memory into four registers using four consecutive `POP.W` instructions requires eight clock cycles, and writing a 64-bit word from registers to memory with the help of four `PUSH.W` instructions takes even 12 cycles. But copying 64 bits from four registers to four other registers can be done in just four cycles. This makes a strong case to implement the compression function in such a way that the frequently-accessed values are kept in registers as much as possible.

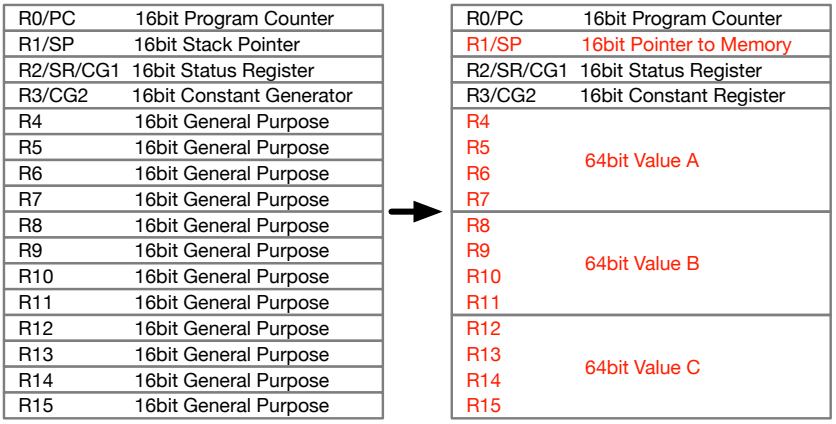


Fig. 3. The MSP430 has twelve general-purpose 16-bit registers. During the computation of the compression function, we use them to store three 64-bit words, while `R1` is used as a pointer to efficiently access 64-bit words in RAM.

On-the-Fly Message Schedule. The message schedule, which expands the 16 words $w_0, \ldots, w_{15}$ of a 1024-bit message-block to 80 words $w_0, \ldots, w_{79}$, can either be pre-computed or computed "on the fly." The former approach has the disadvantage that all 80 words need to be stored in RAM, where they consume 640 bytes in total. Since a RAM footprint of 640 bytes is not non-negligible on an MSP430, it makes sense to compute the words $w_{16}, \ldots, w_{79}$ on the fly, one word per round. Due to the fact that only the words w_{t-16}, w_{t-15}, w_{t-7}, and w_{t-2} are actually required to compute the value of w_t, a buffer containing the preceding 16 words w_{t-16} to w_{t-1} is sufficient. Therefore, when following this approach, the memory consumption is reduced from 640 to only 128 bytes. To avoid the copying of words from w_{t-15} to w_{t-16}, w_{t-14} to w_{t-15}, and so on in every round, we adopt a *circular buffer*, whereby in round t of the compression function the word w_t corresponds to the word $w_{t \bmod 16}$ in the buffer. In this way, the word w_t (i.e. $w_{t \bmod 16}$) is computed as

$$w_{t \bmod 16} \leftarrow [\sigma_1(w_{(t+14) \bmod 16}) + w_{(t+9) \bmod 16} + \sigma_0(w_{(t+1) \bmod 16}) + w_{t \bmod 16}] \bmod 2^{64}.$$

The approach we use to implement this circular buffer is based on a dedicated memory alignment and pointer masking. More precisely, the buffer is aligned in memory on a 256-byte boundary, so that it starts at a memory address of the form $a =$ `0x..00`. As the buffer is 128 byte long, it ranges up to $a +$ `0x7f`. The computation of w_t is then carried out by using the register `R1` as pointer into this buffer. To access w_{t-16}, w_{t-15}, w_{t-7}, and w_{t-2}, the pointer is incremented successively so that it moves in relative steps from one iteration of the message schedule to the next. The circular behaviour of the buffer is guaranteed by the application of a bit-mask to `R1` (e.g. via `AND.W #0xff7f, R1`) every time it has been incremented so that `R1` always stays within the valid address range.

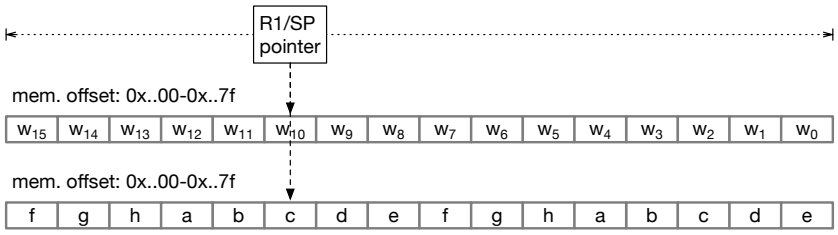


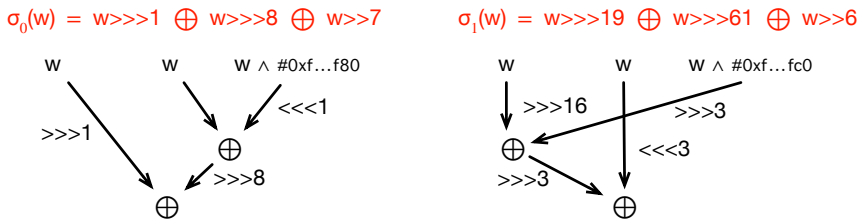
Fig. 4. The 128-byte buffers are 256-byte aligned so that they range from an address of form `0x..00` (i.e. the last two hex-digits are 0) to an address of form `0x..7f`.

.

An alternative approach to implement a circular buffer without the need to copy 64-bit words in every round was introduced in [2]. This approach utilizes a "sliding window" of 16 message words w_i in a double-sized buffer of 32 words (256 bytes), so that the words only need to be copied once every 16 rounds. The computational cost of copying these words every 16 rounds is only slightly more than the cost of masking the pointer `R1` after every increment that we perform in our approach. So, regarding the message schedule part, the main advantage of our approach based on pointer masking compared to [2] is a reduction of the RAM consumption by half since we do not need a double-sized buffer.

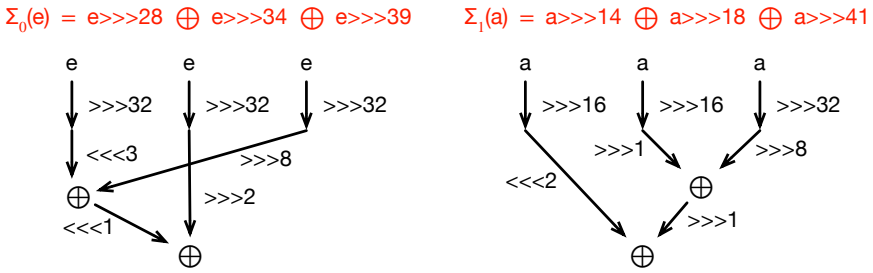
Avoiding Word-Wise State Rotation. An ordinary implementation of the compression function described in Subsect. 2.2 that directly follows the steps as specified in [7] would not be very efficient since it involves a word-wise rotation of the state, i.e. the working variables have to be copied from g to h, from f to g, and so on in every round. Similar to the message schedule, we can minimize the execution time through a circular buffer using the memory alignment and pointer masking described before. This buffer for the eight working variables is adjusted in a way that allows for fast switching between the message schedule and the compression function. As depicted in Fig. 4, the words of the message schedule are stored in reverse order, and the buffer for the compression function contains every word twice. The words are rotated so that e.g. working variable e aligns with the position of word w_0, which eliminates the need to mask the pointer `R1` when switching from the message-word buffer to the buffer with the working variables for the computation of the round function.

Optimized Rotations. The rotations of 64-bit words that are carried out as part of the functions $\sigma_0(w)$, $\sigma_1(w)$, $\Sigma_0(e)$, $\Sigma_1(a)$ are slower on MSP430 than on more sophisticated processors due to the lack of a fast barrel shifter capable to shift/rotate a register by several bits at once. Instead, the MSP430 provides instructions for shifts/rotations by only a single bit [11]. However, one can still reduce the overall execution time by carefully optimizing each function. Special base cases where a 64-bit word (held in four registers) is rotated by 1, 8, or 16



(a) Computation of $\sigma_0(w)$. (b) Computation of $\sigma_1(w)$.

Fig. 5. Optimized computation of the rotations for the message schedule. Shift operations (i.e. $w \gg 7$ and $w \gg 6$) are computed using a masking operation (to set the appropriate bits to 0) followed by a rotation.



(a) Computation of $\Sigma_0(e)$. (b) Computation of $\Sigma_1(a)$.

Fig. 6. Optimized computation of the rotations for the compression function.

bits have the following costs. A simple rotation by 1 bit can be implemented to execute in only five cycles (see Listing 1 and 2). When taking advantage of the byte-swap instruction `SWPB`, a rotation by 8 bits can be done via the sequence in Listing 3 so that it only takes 16 cycles instead of the 40 it would take when the operand was rotated eight times by 1 bit. Finally, a rotation by a multiple of 16 bits is basically free as one can simply "re-order" the registers, e.g. when a 64-bit word is held in the four registers (`R4`, `R5`, `R6`, `R7`), it can be implicitly rotated by 16 bits by accessing it in the order (`R7`, `R4`, `R5`, `R6`).

Figure 5 illustrates how we optimized the functions $\sigma_0(w)$ and $\sigma_1(w)$ of the message schedule. Note that the shift operations are transformed to rotations and logical ANDs ($\wedge$) with a mask to ensure that the appropriate bits are all set to 0. These functions can, therefore, be re-written as

$$\sigma_0(w) = (w \ggg 1) \oplus ((w \oplus ((w \wedge \texttt{0xf...f80}) \lll 1)) \ggg 8)$$
$$\sigma_1(w) = ((((w \wedge \texttt{0xf...fc0}) \ggg 3) \oplus (w \ggg 16)) \ggg 3) \oplus (w \lll 3).$$

Figure 6 shows how we optimized the functions $\Sigma_0(w)$ and $\Sigma_1(w)$ of the compression function. These functions can be re-written as

Table 1. Execution time of the four sigma functions.

Function	Rot+XOR	Loads	Total
$\sigma_0(w)$	36 cycles	21 cycles	57 cycles
$\sigma_1(w)$	55 cycles	16 cycles	71 cycles
$\Sigma_0(e)$	54 cycles	26 cycles	80 cycles
$\Sigma_1(a)$	44 cycles	26 cycles	70 cycles

$$\Sigma_0(e) = ((((e \ggg 32) \lll 3) \oplus ((e \ggg 32) \ggg 8)) \lll 1) \oplus ((e \ggg 32) \ggg 2)$$
$$\Sigma_1(a) = ((a \ggg 16) \lll 2) \oplus ((((a \ggg 16) \ggg 1) \oplus ((a \ggg 32) \ggg 8)) \ggg 1).$$

Besides executing the actual rotations, the 64-bit words also have to be loaded from memory into the registers and XORed ($\oplus$) twice, the latter of which takes eight cycles. The detailed costs of the rotations are summarized in Table 1.

Choice and Majority Function. The Choice (Ch) and Majority (Maj) function both take three 64-bit operands as input. Unlike for the rotations, one can perform these operations on 16-bit chunks in such a way that there is no need to load the entire 64-bit words from memory at once. Using this approach, we start with three pointers to the 64-bit operands, and then progressively execute the whole operation on 16-bit chunks (e.g. we start at the lowest 16 bits of the words, then continue with the next higher 16 bits, and so on). But since these functions are not really complex, there is only little space for optimization.

4 Experimental Results

To assess the performance of our software we compared it with various C implementations that are usable (and optimized) for embedded platforms such as the MSP430. More concretely, we looked at SHA-512 implementations from

- the paper of Cheng et al. [2],
- the CycloneCRYPTO library [8],
- the Noise-C protocol [12], and
- the RELIC toolkit [1].

The version of Cheng et al. we benchmarked is a plain C implementation of an approach that uses a double-length buffer to avoid copying of working variables in every round. CycloneCRYPTO is a cryptographic library specifically tuned for use in embedded systems. Noise-C is a plain C implementation of the Noise framework for building security protocols. Finally, RELIC is a research-oriented cryptographic meta-toolkit with emphasis on efficiency and flexibility.

These implementations have been compiled and benchmarked with version 7.21 of IAR Embedded Workbench for MSP430 using an MSP430F1611 as the target device. The optimization level of the C compiler was set to medium, and Common Subexpression Elimination as well as Code Motion were enabled. We determined the stack memory consumption using a simple stack canary.

Table 2. Execution times of SHA-512 implementations on an MSP430F1611.

Implementation	Type	Hash 3 byte	Hash 1000 byte	Compr. only
Our software	C & Asm	42351 cycles	337736 cycles	40582 cycles
Cheng et al. [2]	pure C	100354 cycles	792951 cycles	97597 cycles
Cyclone [8]	pure C	102026 cycles	795323 cycles	97698 cycles
Noise-C [12]	pure C	97297 cycles	758898 cycles	94468 cycles
RELIC [1]	pure C	123466 cycles	1084390 cycles	118420 cycles

Table 3. Memory requirements of SHA-512 implementations.

Implementation	Type	Code size	RAM size
Our software	C & Asm	2104 bytes	390 bytes
Cheng et al. [2]	pure C	2642 bytes	408 bytes
Cyclone [8]	pure C	2840 bytes	318 bytes
Noise-C [12]	pure C	7436 bytes	966 bytes
RELIC [1]	pure C	3624 bytes	990 bytes

Performance. The execution times of the five SHA-512 implementations are summarized in Table 2. We measured the number of cycles needed to compute the SHA-512 digest of a 3-byte and a 1000-byte message, respectively, as well as the number of cycles for a single execution of the compression function. These results show that our implementation is the fastest; concretely, it is

- 2.30 to 2.91 times faster to compute the digest of a 3-byte message,
- 2.35 to 3.21 times faster to compute the digest of a 1000-byte message, and
- 2.33 to 2.92 times faster to execute the compression function.

The fastest "pure" C implementation is the one from Noise-C, closely followed by that of Cheng et al. and the one from CycloneCRYPTO. RELIC, which is not particularly optimized for speed on embedded devices, is between 21% and 26% slower than the other three C implementations.

Code Size and RAM Footprint. Table 3 shows the results for the code size and RAM consumption. Our implementation has the smallest binary code size (only 2104 bytes), followed by the ones of Cheng et al., CycloneCRYPTO, and RELIC. The code size of Noise-C exceeds the size of all other implementations by a factor of more than two, which is because it unrolls eight rounds to avoid the copying of working variables in each round. Regarding RAM footprint, the CycloneCRYPTO library is the most efficient one since it needs only 318 bytes of RAM. Our software follows with 390 bytes, and then Cheng et al's with 408 bytes. The implementations with the largest RAM footprint are the ones from Noise-C and RELIC with respectively 966 and 990 bytes. This is mainly due to the fact that, in these two implementations, all 80 words w_i from the message schedule are pre-computed and stored in an array in RAM.

5 Concluding Remarks

SHA-512 is a standardized and well-established hash function whose use cases range from signature schemes (e.g. EdDSA) to all kinds of security protocols (e.g. IPSec). In this paper, we presented a highly-optimized assembly-language implementation of SHA-512 for 16-bit MSP430 microcontrollers. We explained how we handle 64-bit words, how we minimize RAM usage by performing the message schedule on the fly, and how we avoid the copying of working variables during the round function. Further, we discussed the efficient implementation of circular buffers through memory alignment and pointer masking. Finally, we tackled the the problem of performing multi-bit rotations on the MSP430, and presented fast implementations of the functions $\sigma_0(w)$, $\sigma_1(w)$, $\Sigma_0(e)$, $\Sigma_1(a)$.

Our experiments show that our implementation compares very favorably to the three C implementations we benchmarked, which means it is (at least) 2.3 times faster than the best C implementation. In addition, it has a smaller code size, and is also among the most efficient implementations with respect to the RAM footprint. Our work can be directly used to improve the speed and code size of SHA-512-based cryptosystems (resp. security protocols) on the MSP430 platform, and we hope that the presented optimization techniques will also be useful to increase the efficiency of other members of the SHA2 family.

References

1. D. F. Aranha, C. P. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an efficient library for cryptography. Source code, available online at `http://github.com/relic-toolkit/relic`, 2020.
2. H. Cheng, D. Dinu, and J. Großschädl. Efficient implementation of the SHA-512 hash function for 8-bit AVR microcontrollers. In J.-L. Lanet and C. Toma, editors, *Innovative Security Solutions for Information Technology and Communications — SecITC 2018*, volume 11359 of *Lecture Notes in Computer Science*, pages 273–287. Springer Verlag, 2019.
3. I. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer Verlag, 1989.
4. D. Evans. The Internet of things: How the next evolution of the Internet is changing everything. Cisco IBSG white paper, available for download at `http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, Apr. 2011.
5. S. Josefsson and I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). Internet Research Task Force, Crypto Forum Research Group, RFC 8032, Jan. 2017.
6. R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Advances in Cryptology — CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer Verlag, 1989.
7. National Institute of Standards and Technology (NIST). Secure Hash Standard (SHS). FIPS Publication 180-4, available for download at `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf`, Aug. 2015.
8. Oryx Embedded. CycloneCRYPTO: Embedded Crypto Library. Source code, available online at `http://oryx-embedded.com/products/CycloneCRYPTO.html`, 2021.

9. B. Preneel. Davies-Meyer. In H. C. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 312–313. Springer Verlag, 2nd edition, 2011.
10. W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson, 7th edition, 2016.
11. Texas Instruments Inc. MSP430 Family Architecture Guide and Module Library. TI literature number SLAUE10B, available for download at `http://www.ti.com/sc/docs/products/micro/msp430/userguid/ag_01.pdf`, 1996.
12. R. Weatherley and E. Fidler. Noise-C library. Source code, available online at `http://github.com/rweather/noise-c`, 2016.

A Optimized Rotation of 64-bit Words

The MSP430 architecture provides instructions to shift or rotate a 16-bit value by a single bit. However, unlike e.g. more powerful ARM processors, MSP430 microcontrollers are not equipped with a barrel shifter that would allow one to shift or rotate multiple bits at once. Furthermore, when a 64-bit word is to be shifted or rotated, the number of instructions and, consequently, the execution time increases accordingly. Listing 1 shows a sequence of MSP430 instructions for a 1-bit right-rotation of a 64-bit word that is held in the four registers `R4` to `R7`. These five instructions execute in five clock cycles.

Listing 1. 1-bit right-rotation of a 64-bit word held in `R4` to `R7` (5 cycles).

```
1  BIT.W  #1, R4
2  RRC.W  R7
3  RRC.W  R6
4  RRC.W  R5
5  RRC.W  R4
```

Listing 2 contains a code snipped for a 1-bit left-rotation of a 64-bit word that is held in the four registers `R4` to `R7`. These instructions have an execution time of five clock cycles on an MSP430 microcontroller.

Listing 2. 1-bit left-rotation of a 64-bit word held in `R4` to `R7` (5 cycles).

```
1  RLA.W  R7
2  RLC.W  R4
3  RLC.W  R5
4  RLC.W  R6
5  ADC.W  R7
```

Right/left-rotations by two or three bits can be simply assembled from the instruction sequence for 1-bit rotation. However, there are "shortcuts" for some rotation distances due to special MSP430 instructions. For example, a rotation by eight bits can be greatly accelerated with help of the byte-swap instruction

SWPB, which swaps the lower and upper byte of a 16-bit value. Listing 3 shows how this instruction can be used to implement a right-rotation of a 64-bit word by eight bits, whereby it is assumed that this word is held in the four registers R4 to R7. However, unlike the 1-bit rotation, the rotation by eight bits needs an additional register, namely R8, for storing a temporary value. The instruction sequence of Listing 3 executes in 16 clock cycles, which is 2.5 times faster than a naive implementation based on eight 1-bit right-rotations.

Listing 3. 8-bit right-rotation of a 64-bit word held in R4 to R7 (16 cycles).

```
1  MOV.B  R4, R8
2  XOR.B  R5, R8
3  XOR.W  R8, R4
4  XOR.W  R8, R5
5  MOV.B  R5, R8
6  XOR.B  R6, R8
7  XOR.W  R8, R5
8  XOR.W  R8, R6
9  MOV.B  R6, R8
10 XOR.B  R7, R8
11 XOR.W  R8, R6
12 XOR.W  R8, R7
13 SWPB   R4
14 SWPB   R5
15 SWPB   R6
16 SWPB   R7
```