Adding MAC Functionality to Edon80

Danilo Gligoroski and Svein J. Knapskog

“Centre for Quantifiable Quality of Service in Communication Systems”,
Norwegian University of Science and Technology, Trondheim, Norway

Summary
In this paper we show how the synchronous stream cipher Edon80 - proposed as a candidate stream cipher in Profile 2 of the eSTREAM project, can be efficiently upgraded to a synchronous stream cipher with authentication. We are achieving that by simple addition of two-bit registers into the e-transformers of Edon80 core, an additional 160-bit shift register and by putting additional communication logic between neighboring e-transformers of the Edon80 pipeline core. This upgrade does not change the produced keystream from Edon80 and we project that in total it will need not more then 1500 gates. A previous version of the paper with the same title that has been presented at the Special Workshop “State of the Art of Stream Ciphers SASC 2007”, hosted by ECRYPT at Ruhr University - Bochum, Germany, January 31 - February 1, 2007.

Key words: *Synchronous stream cipher, MAC, Edon80*

Introduction

Stream ciphers have to be accompanied by authentication techniques in order to provide security in the communication. Authentication can be achieved by using separate external authentication functions such as HMAC-MD5 or HMAC-SHA-1 [1, 2] or Wegman-Carter [3] authentication functions or by incorporating the authentication into the computational part of the stream cipher. Incorporating authentication into a stream cipher functionality is not always easy, neither in the design nor in the implementation phase. Frequently, the additional mathematical and logic operations for computing the authentication codes for the encrypted message will slow down the operating speed of the stream cipher. If the authentication part is implemented in hardware, then it may require hardware resources in the same order of magnitude as the basic stream cipher itself.

In the eSTREAM project [4], there is a special sub-category both for software and hardware stream ciphers with authentication named PROFILE 1A and PROFILE 2A. In Phase 1 of the eSTREAM project initially six submissions offered an authentication mechanism. Those submissions were: Frogbit, NLS, Phelix, SFINKS, SSS, and VEST [5-10]. Later, some of them have been broken or withdrawn, consequently in Phase 2 of eSTREAM project only three candidates remain: NLS, Phelix and VEST. At the time of writing, it seams that for NLS and Phelix some weaknesses have been found [11,12]. Although the eSTREAM project does not accept anymore any tweaks or new submissions, we think that the design of an efficient authentication techniques as a part of the internal definition of the remaining unbroken stream ciphers of Phase 2 of eSTREAM project still is an important research challenge.

Edon80 is one of the stream ciphers that has been proposed for hardware based implementations (PROFILE 2) [13]. Its present design does not contain an authentication mechanism by its own. Its initial design was projected to be around 7,500 gates and it was not the most compact proposal in its category. However, recently Kasper, Kumar, Lemke-Rust and Paar [14] have implemented Edon80 in a very compact way in less then 3,000 gates. Their design introduces several clever optimizations that make the implementation of Edon80 very compact, and it became natural for us to consider how to use the □ gfreed space□ h for adding MAC functionality to Edon80. Consequently, in this paper we propose how Edon80 can be upgraded to become a synchronous stream cipher with authentication by adding several hardware components and mostly using its internal structure. The produced MAC is of length 160 bits.

The paper is organized as follows: In Section 2 we give a very brief description of Edon80. In Section 3 we describe the proposed upgrade. In Section 4 we discuss some security aspects of the proposed MAC version of Edon80 and in Section 5 we state our conclusions.

2. Brief description of Edon80

Edon80 is a binary additive stream cipher. Detailed schematic and behavioural description of Edon80 is given in [13]. Here we will focus on the Edon80 Core that is described in figures 1 and 2. The internal structure of Edon80 can be seen as a pipelined architecture of 80 simple 2-bit transformers called e-transformers. The schematic view of a single e-transformer is shown in Figure 1. The structure that performs the operation $*_i$ in e-transformers is a quasigroup operation of order 4. We refer to an e-transformer by its quasigroup operation $*_i$. In

Manuscript received January 5, 2007
Manuscript revised January 25, 2007

Edon80 we have 80 of this e-transformers (the index i varies from 0 to 79), cascaded in a pipeline, one feeding another. The assignment of the working quasigroups is done by the following scheduling formula:

$$(Q,*_i) \leftarrow \begin{cases} (Q,\bullet_{K_i}), & 0 \le i \le 39 \\ (Q,\bullet_{K_{i-40}}), & 40 \le i \le 79. \end{cases} \quad (1)$$

where *Key* is a vector of 80 bits represented by a concatenation of 40 2-bit variables K_i i.e. $Key = K_0 K_1 \ldots K_{39}$.

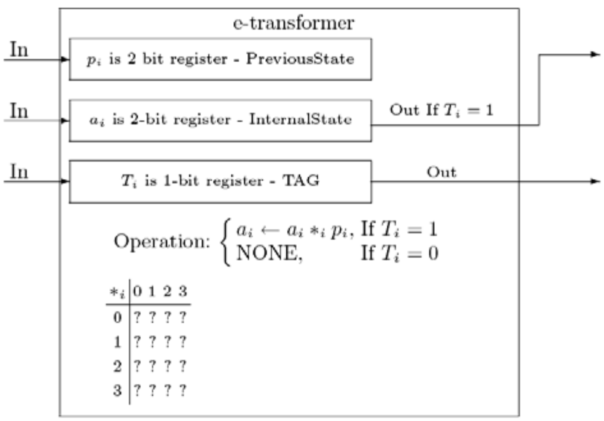

Fig. 1. Schematic representation of a single e-transformer of Edon80.

The two 2-bit registers inside every e-transformer (p_i and a_i) are used as two operands by which the new value of a_i is determined according to the defined quasigroup operation $*i$ for that e-transformer. For different e-transformers different quasigroup operations from a set of 4 predefined quasigroups of order 4 may be defined. Those 4 predefined quasigroups are described in Table 1.

Table 1: Quasigroups used in the design of Edon80

Nr. 61					Nr. 241					Nr. 350					Nr. 564				
$\bullet_0$	0	1	2	3	$\bullet_1$	0	1	2	3	$\bullet_2$	0	1	2	3	$\bullet_3$	0	1	2	3
0	0	2	1	3	0	1	3	0	2	0	2	1	0	3	0	3	2	1	0
1	2	1	3	0	1	0	1	2	3	1	1	2	3	0	1	1	0	3	2
2	1	3	0	2	2	2	0	3	1	2	3	0	2	1	2	0	3	2	1
3	3	0	2	1	3	3	2	1	0	3	0	3	1	2	3	2	1	0	3

Every e-transformer has one tag-bit T_i which controls whether the e-transformer will compute the next value of a_i or do nothing. All of the 80 e-transformers work in parallel to calculate their new value of a_i (if the tag permits that) and then pass that new value a_i to the right neighbouring register p_{i+1}. If the tag forbids the calculation of a_i, the only value that is transferred to the neighbouring element is the value of the tag T_i. Figure 2 shows the pipelined core of Edon80. Finally in this brief description of Edon80 we will describe two modes of operation: *Keystream* mode and *IVSetup* mode.

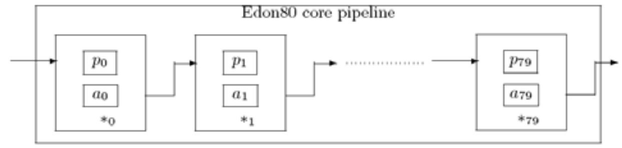

Fig. 2. Edon80 core of 80 pipelined e-transformers.

The *Keystream* mode is described in Table 2. In the first row of that table a periodic (potentially infinite) string is placed that has the shape: 01230123… 0123… . The next 80 rows in the table describe 80 e-transformations of that string by using the obtained values of a_i in *IVSetup* mode and by the quasigroups $*i$. The recurrence equations for these transformations are:

$$\begin{cases} a_{0,0} = a_0 *_0 0 & \\ a_{0,j} = a_{0,j-1} *_0 (j \bmod 4) & 1 \le j \\ a_{i,0} = a_i *_i a_{i-1,0} & 1 \le i \le 79 \\ a_{i,j} = a_{i,j-1} *_i a_{i,j-1} & 1 \le i \le 79,\ 1 \le j \end{cases} \quad (2)$$

The output of the stream cipher is every second value of the last e-transformation i.e. the *Keystream* can be described as:

$$Keystream = a_{79,1}\, a_{79,3}\, a_{79,5} \ldots a_{79,2k-1} \ldots, \quad k = 1,2,\ldots$$

So, practical implementation of the above operations by the Edon80 Core in the *Keystream* mode is as follows: The core is fed by the values 0,1,2 and 3 periodically. After a latency of 80 cycles, the keystream starts to flow from the last e-transformer i.e. from the 2-bit register a_{79}. The keystream consist of every second value that comes out from a_{79}.

Table 2: Representation of quasigroup string e-transformations of Edon80 during *Keystream* mode

$*_i$		0	1	2	3	0	1	2	3	0 . .
$*_0$	a_0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$	$a_{0,8}$ . .
$*_1$	a_1	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$	$a_{1,8}$ . .
.	.	.	.	.	.	.	.	.	.	. . .
$*_{79}$	a_{79}	$a_{79,0}$	$a_{79,1}$	$a_{79,2}$	$a_{79,3}$	$a_{79,4}$	$a_{79,5}$	$a_{79,6}$	$a_{79,7}$	$a_{79,8}$ . .

The *IVSetup* mode defines the initial values of the internal states $a_0, \ldots, a_{79}$, from the values of the initial vector IV. The padded initial vector IV is a concatenation of 40 2-bit variables IV = $\tilde{o}_0\, \tilde{o}_1 \ldots \tilde{o}_{31}$ 3 2 1 0 0 1 2 3 = $\tilde{o}_0\, \tilde{o}_1 \ldots \tilde{o}_{39}$. Then 80 e-transformations are performed on the IV as described in the Table 3. All of those transformations can be described by the recurrence equations (3).

After all 80 e-transformations are performed, the values of $a_0, \ldots, a_{79}$ are initialized by the assignments (4).

$$\begin{cases} t_{0,0} = \upsilon_{39} *_0 K_0 & \\ t_{0,j} = t_{0,j-1} *_0 K_j & 1 \le j \le 39 \\ t_{0,j} = t_{0,j-1} *_0 \upsilon_{j-40} & 40 \le j \le 79 \\ t_{i,0} = \upsilon_{39-i} *_i t_{i-1,0} & 1 \le i \le 39 \\ t_{0,j} = K_{79-i} *_i t_{i-1,0} & 40 \le i \le 79 \\ t_{i,j} = t_{i,j-1} *_i t_{i-1,j} & 1 \le i \le 79,\ 1 \le j \le 79 \end{cases} \tag{3}$$

$$a_i \leftarrow t_{79,i},\ i = 0,1,\ldots,79 \tag{4}$$

The practical implementation of the above operations by the Edon80 Core in the *IVSetup* mode is as follows. Make the following assignments:

$$\begin{cases} T_i \leftarrow 0, & i = 0,\ldots,79 \\ a_{39-i} \leftarrow \upsilon_i & i = 0,\ldots,39 \\ a_{79-i} \leftarrow K_i & i = 0,\ldots,39 \end{cases}$$

Table 3: Representation of quasigroup string e-transformations of Edon80 during *IVSetup* mode

$*_i$		K_0	K_1	$\cdots$	K_{39}	υ_0	υ_1	$\cdots$	υ_{39}
$*_0$	υ_{39}	$t_{0,0}$	$t_{0,1}$	$\cdots$	$t_{0,39}$	$t_{0,40}$	$t_{0,41}$	$\cdots$	$t_{0,79}$
$*_1$	υ_{38}	$t_{1,0}$	$t_{1,1}$	$\cdots$	$t_{1,39}$	$t_{1,40}$	$t_{1,41}$	$\cdots$	$t_{1,79}$
.	.	.	.	.	.	.	.	.	
$*_{38}$	υ_1	$t_{38,0}$	$t_{38,1}$	$\cdots$	$t_{38,39}$	$t_{38,40}$	$t_{38,41}$	$\cdots$	$t_{38,79}$
$*_{39}$	υ_0	$t_{39,0}$	$t_{39,1}$	$\cdots$	$t_{39,39}$	$t_{39,40}$	$t_{39,41}$	$\cdots$	$t_{39,79}$
$*_{40}$	K_{39}	$t_{40,0}$	$t_{40,1}$	$\cdots$	$t_{40,39}$	$t_{40,40}$	$t_{40,41}$	$\cdots$	$t_{40,79}$
$*_{41}$	K_{38}	$t_{41,0}$	$t_{41,1}$	$\cdots$	$t_{41,39}$	$t_{41,40}$	$t_{41,41}$	$\cdots$	$t_{41,79}$
.	.	.	.	.	.	.	.	.	
$*_{78}$	K_1	$t_{78,0}$	$t_{78,1}$	$\cdots$	$t_{78,39}$	$t_{78,40}$	$t_{78,41}$	$\cdots$	$t_{78,79}$
$*_{79}$	K_0	$t_{79,0}$	$t_{79,1}$	$\cdots$	$t_{79,39}$	$t_{79,40}$	$t_{79,41}$	$\cdots$	$t_{79,79}$

In the first 80 cycles feed the register p_0 of Edon80 Core by the values K_0, K_1, K_2, K_3, …, K_{39} and after that by $\tilde{o}_0$, $\tilde{o}_1$, …, $\tilde{o}_{39}$. In cycle 80 set the tag T_0 to 0, and feed the content of the register a_{79} into the register a_0. In the next 79 cycles all of the e-transformers *1, …, *79 will stop consecutively. When the register *i stops, the content of the register a_{79} will be fed into the register a_i. So in total *IVSetup* mode will finish its job after 160 cycles.

Formally we can combine the assignments in formulas (3) and (4) and write the result á=$a_0a_1 \ldots a_{79}$ of the IVSetup procedure as:

$$\alpha = IVSetup(Key, IV) \tag{5}$$

In the original submission of Edon80 [13] it was conjectured that the function *IVSetup* acts as a one-way function and a major part of the security of Edon80 relies on that conjecture. As we will see in the following chapters, the security of the MACEdon80 will also rely on one-wayness of *IVSetup*.

3. MACEdon80

In this section we will describe how to compute a message authentication code by using data already present in the Edon80 core. For that purpose let us adopt the following notation:

- The plain text message of length $2k$ bits is $M=m_0m_1\ldots m_{k-1}$.
- The encrypted message E of length $2k$ bits is $E=e_0e_1\ldots e_{k-1}$.
- Message authentication code is stored in the following 160-bit string: MAC=$c_0c_1\ldots c_{79}$ where every c_i is two-bit value.
- e-transformation of the string á=$a_0a_1 \ldots a_{n-1}$ by one quasigroup $(Q,*)$ and leader $l \in Q$
  $e_{l,*}$(á) = $b_0b_1 \ldots b_{n-1}$
  where $b_i = b_{i-1} * a_i$ for each i = 0,1,…, n-1 and $b_{-1}=l$.
- e-transformation of the string á=$a_0a_1 \ldots a_{n-1}$ by a sequence of quasigroups $*_0, *_1, \ldots, *_{79}$ uniquely determined from the *Key* by the scheduling formula (1) and one leader $l \in Q$
  $e_{l,Key}$(á) = $b_0b_1 \ldots b_{n-1}$
  where $b_i = b_{i-1} *_i a_i$ for each i=0,1,…,n-1 and $b_{-1}=l$.
- Composition of $e_{l_i,Key}$ transformations, for certain sequence of k leaders $L = a_0a_1 \ldots a_{k-1}$
  $E_{L,Key}$(á)≡$(e_{l_0,Key} \circ e_{l_1,Key} \circ \ldots \circ e_{l_{k-1},Key})$(á)=
  =$e_{l_{k-1},Key}(\ldots e_{l_1,Key}(e_{l_0,Key}$(á)$)\ldots)$.

Table 4: Tabular representation of $E_{L,Key}$(á)

	$*_0$ a_0	$*_1$ a_1	$*_2$ a_2	$\cdots$	$*_{79}$ a_{79}
l_0	$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$\cdots$	$b_{0,79}$
l_1	$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$\cdots$	$b_{1,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
l_{k-2}	$b_{k-2,0}$	$b_{k-2,1}$	$b_{k-2,2}$	$\cdots$	$b_{k-2,79}$
l_{k-1}	$b_{k-1,0}$	$b_{k-1,1}$	$b_{k-1,2}$	$\cdots$	$b_{k-1,79}$

In Table 4 we give a tabular representation of the composition $E_{L,Key}$(á). The formal recurrence equations for the values in Table 4 are the following:

$$\begin{cases} b_{0,0} = l_0 *_0 a_0 & \\ b_{0,j} = b_{0,j-1} *_j a_j & 1 \le j \le 79 \\ b_{i,0} = l_i *_0 b_{i-1,0} & 1 \le i \le k-1 \\ b_{i,j} = b_{i,j-1} *_i b_{i-1,j} & 1 \le i \le k-1,\ 1 \le j \le 79 \end{cases} \tag{6}$$

We now proceed to the formal description of a MAC computation for a given message $M = m_0m_1 \ldots m_{k-1}$ that is of length $2k$ bits ($k \ge 1$). First let us define a function $f_{\alpha}: Q^+ \to Q^+$ for every string $\alpha = a_0a_1 \ldots a_{79} \in Q^{80}$ and every message $M = m_0m_1 \ldots m_{k-1}$ as following:

$$f_{\alpha}(M) = \begin{cases} M \,\|\, \alpha' \,\|\, M, & \text{if } 1 \le k < 80 \\ M_1 \,\|\, M_2 \,\|\, M_3, & \text{if } k \ge 80, \end{cases} \tag{7}$$

where $\alpha' = a_{79-k} \ldots a_0$, $M_1 = m_0m_1 \ldots m_{79}$, $M_2 = m_0m_{80} \ldots m_{k-81}m_{k-1}$, $M_3 = m_{k-80} \ldots m_{k-1}$ and the sign $\|$ denotes string concatenation.

Example 1. Let us give three examples for $k = 2$, $k = 80$ and $k = 84$.

1. If $M = m_0m_1$ then $f_{\alpha}(M) = m_0m_1a_{77}a_{76} \ldots a_0m_0m_1$
2. If $M = m_0m_1 \ldots m_{79}$ then $f_{\alpha}(M) = m_0m_1 \ldots m_{79}m_0m_1 \ldots m_{79}$
3. If $M = m_0m_1 \ldots m_{82}m_{83}$ then $f_{\alpha}(M) = m_0m_1 \ldots m_{79} \| m_0m_{80}m_1m_{81}m_2m_{82}m_3m_{83} \| m_4 \ldots m_{79}$

Finally, computation of the MACEdon80 is done by the following formula:

$$MAC(M) = E_{f_{\alpha}(M),Key}(\alpha) \tag{8}$$

where α is computed by the formula (5).

Example 2. In Table 5 we give calculations of the MACEdon80 for $M = m_0m_1$. MAC values $c_0c_1 \ldots c_{79}$ are in fact the values of the last row: $b_{81,0}b_{81,1} \ldots b_{81,79}$.

Table 5: Tabular representation of calculations for computing the MAC of a message $M = m_0m_1$

	$*_0$ a_0	$*_1$ a_1	$*_2$ a_2	$\cdots$	$*_{79}$ a_{79}
m_0	$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$\cdots$	$b_{0,79}$
m_1	$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$\cdots$	$b_{1,79}$
a_{77}	$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$\cdots$	$b_{2,79}$
a_{76}	$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$\cdots$	$b_{3,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_0	$b_{79,0}$	$b_{79,1}$	$b_{79,2}$	$\cdots$	$b_{79,79}$
m_0	$b_{80,0}$	$b_{80,1}$	$b_{80,2}$	$\cdots$	$b_{80,79}$
m_1	$b_{81,0}$	$b_{81,1}$	$b_{81,2}$	$\cdots$	$b_{81,79}$

4. Upgrading the hardware resources in the Edon80 core for computing the MAC

In this section we will explain what types of hardware resources we need to add into the Edon80 core i.e. into the basic hardware parts - the e-transformers in order to compute efficiently the MAC as it is described by the formula (8).

Let $A = A_0A_1 \ldots A_{79}$ be a 160-bit shift register described as a concatenation of 80 2-bit registers A_i, $i = 0,..,79$ and let its initial value be the same as the value of $\alpha = IVSetup(Key, IV) = a_0 \ldots a_{79}$ i.e., $A_0A_1 \ldots A_{79} \equiv a_0a_1 \ldots a_{79}$.

Additionally, into every e-transformer let us introduce a 2-bit register c_i, as it is described in Figure 3.

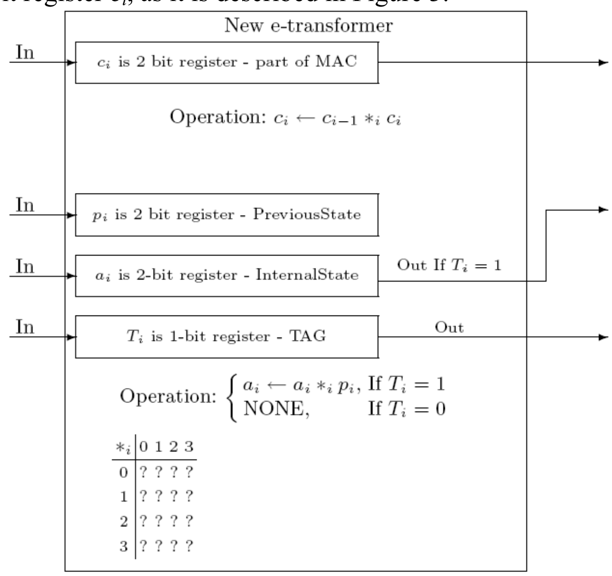

Fig. 3. Schematic representation of a new e-transformer for MACEdon80.

Since the value of the MAC depends on the plain text message M, the functionality of the MACEdon80 differs slightly in encryption and decryption mode. The encryption mode is described in Figure 4.

In the beginning let the initial values of c_i be the values of $\alpha = IVSetup(Key, IV) = a_0a_1 \ldots a_{79}$ i.e. $c_i = a_i$, $i = 0, \ldots, 79$. The main idea is to use the shift register A as a 160-bit buffer in order to implement the functionality which is formally described by the function for $f_{\alpha}(M)$ in equation (7). In the first 160 cycles when a plain text m_i enters the Edon80 core (and this is done every second cycle), before it is encrypted (xor-ed with the 2-bit value of the produced keystream), it also enters the leftmost part of A (the register A shifts right for two bits), and the value of the register c_0 is updated by $c_0 \leftarrow m_i *_0 c_0$.

When A is completely filled by the values of the message, it will start to feed the first e-transformer $*_0$ as well. So from then on, the values of c_0 and c_i, $1 \le i \le 79$ will be updated every cycle as follows:

$$\begin{cases} c_0 \leftarrow m_i *_0 c_0, & \text{if } m_i \text{ enters the core,} \\ c_0 \leftarrow A_{79} *_0 c_0, & \text{if no } m_i \text{ enters the core,} \\ c_0 \leftarrow c_{i-1} *_i c_i, & 1 \le i \le 79. \end{cases}$$

When the plain text feeding stops (the message M is completely encrypted), computation of the MAC will continue in the next 160 cycles with the remaining values in the register A. In the first 80 cycles the assignments will be:

$$\begin{cases} c_0 \leftarrow A_{79} *_0 c_0, & \\ c_0 \leftarrow c_{i-1} *_i c_i, & 1 \le i \le 79. \end{cases}$$

Then the value of c_0 will be computed as a final value for the MAC and the activity of the first e-transformer *0 will be halted. In the next 80 cycles the values of the registers c_i will be updated by $c_i \leftarrow c_{i-1} *i\ c_i$, and consecutively every e-transformer from 1 to 79 will stop its activity for computing the MAC. So completion of the MAC computation will end 160 cycles after the encryption of the message M.

The MACEdon80 in decryption mode is slightly different than in encryption mode. It is described in Figure 5. First the encrypted stream e_i is decrypted into the plain text stream m_i and those values feed the register A and the first e-transformer *0.

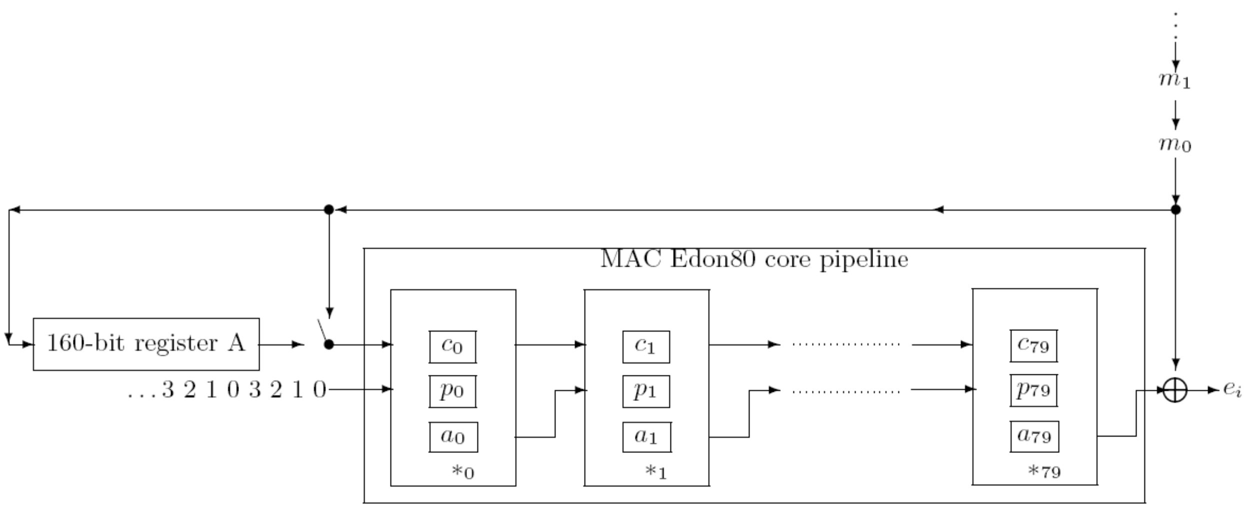

Fig. 4. MACEdon80 core in encryption mode.

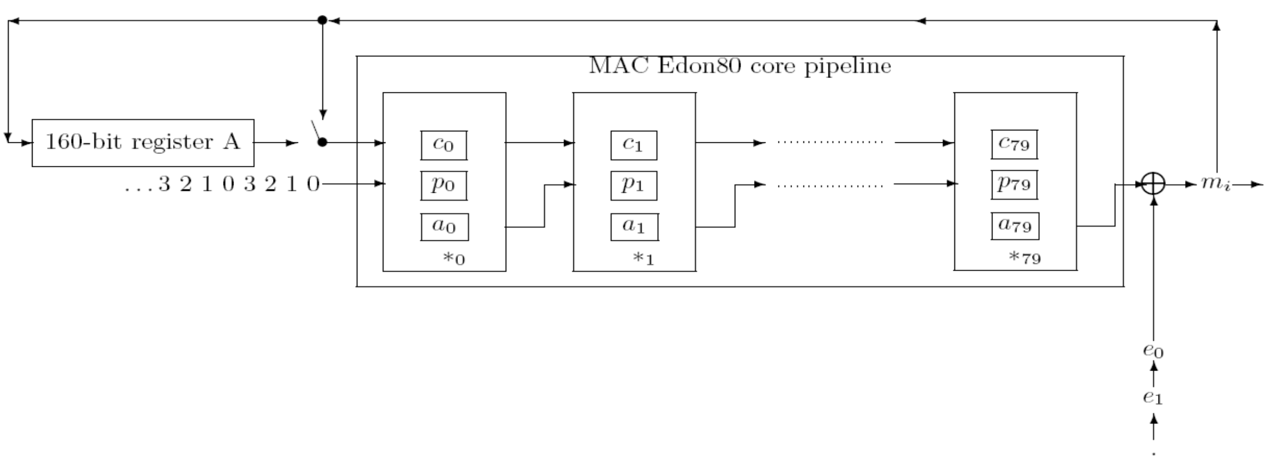

Fig. 5. MACEdon80 core in decryption mode.

At the end of this section we will give an estimate for the needed hardware resources for upgrading Edon80 to the MACEdon80. Here we will use the estimates for the compact version of Edon80 of Kasper et. al. [14]. A 160-bit shift register takes around 1000 gates. The additional communication logic and the logic when to start and when to stop computations should take no more than 500 gates. So the total upgrade will take around 1500 logic gates. Having in mind recent excellent "re-invention" or "re-packing" of Edon80 made by Kasper et. al. we do not exclude the possibility that MACEdon80 could be implemented with even less additional hardware resources.

5. Security of the MACEdon80

5.1 Resistance against key recovery for reused IVs

In the light of the latest attacks on Phelix [12] we will discuss here the resistance of MACEdon80 against a key recovery attack if an IV is used two or more times. For that purpose let us take two messages $M=m_0$ and $M'=m'_0$ ($m_0 \neq m'_0$). The table representation for the MAC computations for those two messages are represented in tables 6a and 6b.

The attacker knows the values m_0, m'_0, MAC(m_0)= $b_{80,0}\ b_{80,1}\ b_{80,2} \ldots b_{80,79}$ and MAC(m'_0)=$b'_{80,0}\ b'_{80,1}\ b'_{80,2} \ldots b'_{80,79}$. If she/he has many MAC values (for the same IV) then she/he can set up a huge system of quasigroup equations and try to solve it for the unknown values $a_0, \ldots, a_{79}$ as well as for the unknown quasigroup operations *0, *1, …, *79.

Table 6a: Tabular representation of calculations for computing MAC of the message $M=m_0$.

	$*_0$	$*_1$	$*_2$	$\cdots$	$*_{79}$
	a_0	a_1	a_2	$\cdots$	a_{79}
m_0	$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$\cdots$	$b_{0,79}$
a_{78}	$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$\cdots$	$b_{1,79}$
a_{77}	$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$\cdots$	$b_{2,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_0	$b_{79,0}$	$b_{79,1}$	$b_{79,2}$	$\cdots$	$b_{79,79}$
m_0	$b_{80,0}$	$b_{80,1}$	$b_{80,2}$	$\cdots$	$b_{80,79}$

Table 6b: Tabular representation of calculations for computing MAC of the message $M'=m'_0$.

	$*_0$	$*_1$	$*_2$	$\cdots$	$*_{79}$
	a_0	a_1	a_2	$\cdots$	a_{79}
m'_0	$b'_{0,0}$	$b'_{0,1}$	$b'_{0,2}$	$\cdots$	$b'_{0,79}$
a_{78}	$b'_{1,0}$	$b'_{1,1}$	$b'_{1,2}$	$\cdots$	$b'_{1,79}$
a_{77}	$b'_{2,0}$	$b'_{2,1}$	$b'_{2,2}$	$\cdots$	$b'_{2,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_0	$b'_{79,0}$	$b'_{79,1}$	$b'_{79,2}$	$\cdots$	$b'_{79,79}$
m'_0	$b'_{80,0}$	$b'_{80,1}$	$b'_{80,2}$	$\cdots$	$b'_{80,79}$

Having in mind that we have based the security of Edon80 on the conjectured one-way nature the *IVSetup* procedure, and the fact that MACEdon80 uses the results of *IVSetup* as a starting point of its calculations we claim that key recovery attack for MACEdon80 if the same IV value is used two (or more) times is equivalent to breaking the supposed one-way function *IVSetup*. A more in-dept analysis of these claims is definitely necessary, and this will be subject of our further research.

5.2 Computational limits for finding collisions of the MACEdon80

Since the MACEdon80 is a symmetric synchronous stream cipher with authentication, the secret *Key* is known in advance both to sender (Alice) and receiver (Bob). In this section we will discuss the collision resistance of the proposed MAC function supposing that *Key* is known to Alice and Bob. Namely, we are interested in whether we can use the MACEdon80 both as an authentication primitive and as a primitive that does not allow repudiation.

Let M=$m_0 m_1 \ldots m_{k-1}$ be the message encrypted by MACEdon80, with the key *Key* and the initial value IV, and let MAC(M) = $E_{f\acute{a}(M),\mathrm{Key}}(\acute{a})$ be the MAC value where $\acute{a}$=*IVSetup*(*Key*, *IV*).

First let us discuss the second pre-image resistance of the MACEdon80. That means that we are interested in the amount of computational effort that Alice or Bob have to perform in order to find another message M'=$m'_0 \ldots m'_{k-1}$ such that

$$\mathrm{MAC}(M) = \mathrm{MAC}(M').$$

Note that the length of both messages M and M' have to be equal since the computation of the MAC is done after the exact length k, which is known in advance.

Let us take for example $k < 80$ (the case $k \geq 80$ can be treated similarly). Then computing MAC for M and M' can be described by tables 7a and 7b.

Table 7a: Calculations for computing MAC of the message $M=m_0 \ldots m_{k-1}$, for $1 \leq k < 80$

	$*_0$	$*_1$	$*_2$	$\cdots$	$*_{79}$
	a_0	a_1	a_2	$\cdots$	a_{79}
m_0	$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$\cdots$	$b_{0,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
m_{k-1}	$b_{k-1,0}$	$b_{k-1,1}$	$b_{k-1,2}$	$\cdots$	$b_{k-1,79}$
a_{79-k}	$b_{k,0}$	$b_{k,1}$	$b_{k,2}$	$\cdots$	$b_{k,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_0	$b_{79,0}$	$b_{79,1}$	$b_{79,2}$	$\cdots$	$b_{79,79}$
m_0	$b_{80,0}$	$b_{80,1}$	$b_{80,2}$	$\cdots$	$b_{80,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
m_{k-2}	$b_{79+k-1,0}$	$b_{79+k-1,1}$	$b_{79+k-1,2}$	$\cdots$	$b_{79+k-1,79}$
m_{k-1}	c_0	c_1	c_2	$\cdots$	c_{79}

Table 7b: Calculations for computing MAC of the message $M'=m'_0 \dots m'_{k-1}$, for $1 \le k < 80$

	$*_0$	$*_1$	$*_2$	$\cdots$	$*_{79}$
	a_0	a_1	a_2	$\cdots$	a_{79}
m'_0	$b'_{0,0}$	$b'_{0,1}$	$b'_{0,2}$	$\cdots$	$b'_{0,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
m'_{k-1}	$b'_{k-1,0}$	$b'_{k-1,1}$	$b'_{k-1,2}$	$\cdots$	$b'_{k-1,79}$
a_{79-k}	$b'_{k,0}$	$b'_{k,1}$	$b'_{k,2}$	$\cdots$	$b'_{k,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_0	$b'_{79,0}$	$b'_{79,1}$	$b'_{79,2}$	$\cdots$	$b'_{79,79}$
m'_0	$b'_{80,0}$	$b'_{80,1}$	$b'_{80,2}$	$\cdots$	$b'_{80,79}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
m'_{k-2}	$b'_{79+k-1,0}$	$b'_{79+k-1,1}$	$b'_{79+k-1,2}$	$\cdots$	$b'_{79+k-1,79}$
m'_{k-1}	c_0	c_1	c_2	$\cdots$	c_{79}

The irregularity of quasigroup operations and their lack of properties such as associativity, commutativity or neutral elements make us believe that attempts to set up a system of equations from tables 7a and 7b and to solve that system on unknown variables $m'_0, \dots, m'_{k-1}$ will require computational efforts in the range of 2^{160}.

The situation for collision resistance is very similar to the one for un-keyed authentication primitives (hash functions). Namely, if Alice or Bob start randomly to choose messages *M* and *M'* in order to find a collision using the birthday paradox attack, she/he will need approximately 2^{80} attempts to find a collision since the total length of MAC is 160 bits. At the time of writing, we do not know any faster attack than that. Again, the irregularity of the quasigroup operations and the lack of algebraic properties of quasigroups as previously mentioned make us believe that the birthday attack is the most effective attack for finding MAC collisions when *Key* is known both to Alice and Bob.

6. Conclusion

We have designed an upgraded version of Edon80 called MACEdon80 that besides the keystream production which is the same as in Edon80, computes a message authentication code MAC for the messages that are encrypted (decrypted).

The total length of computed the MAC is 160 bits. It takes 160 cycles after the encryption (decryption) of the message to compute the MAC. We estimate that the additional hardware resources for this upgrade are in the range of 1500 gates.

References

[1] The Federal Information Processing Standards Publication Series of the National Institute of Standards and Technology (NIST), "The Keyed-Hash Message Authentication Code (HMAC)", FIPS PUB 198, March 6, 2002, http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf

[2] J. Kim, A. Biryukov, B. Preneel and S. Hong, "On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1", Proceedings of SCN 2006 (Moti Yung ed.), LNCS, Springer-Verlag, 2006.

[3] M.N.Wegman and J.L.Carter, "New hash functions and their use in authentication and set equality", J. Comp. Sys. Sci. 22 (1981), 265-279.

[4] eSTREAM - ECRYPT - Stream Cipher Project, http://www.ecrypt.eu.org/stream

[5] T. Moreau, "The Frogbit Cipher, a Data Integrity Algorithm", Report 2005/009, [4], 2005.

[6] G. Rose, P. Hawkes, M. Paddon and M. W. de Vries, "Primitive Specification for NLS", Report 2005/019, [4], 2005.

[7] D. Whiting, B. Schneier, S. Lucks and F. Muller, "Phelix - Fast Encryption and Authentication in a Single Cryptographic Primitive", Report 2005/020, [4], 2005.

[8] A. Braeken, J. Lano, N. Mentens, B. Preneel and I. Verbauwhede, "SFINKS : A Synchronous Stream Cipher for Restricted Hardware Environments", Report 2005/026, [4], 2005.

[9] G. Rose, P. Hawkes, M. Paddon and M. W. de Vries, "Primitive Specification for SSS", Report 2005/028, [4], 2005.

[10] S. O'Neil, B. Gittins, H. Landman, "VEST – Hardware Dedicated Stream Ciphers", Report 2005/032, [4], 2005.

[11] J. Y. Cho and J. Pieprzyk, "Crossword Puzzle Attack on NLSv2", Report 2006/051, [4], 2006.

[12] H. Wu and B. Preneel, "Differential Attacks against Phelix", Report 2006/056, [4], 2006.

[13] D. Gligoroski, S. Markovski, L. Kocarev and M. Gusev, "Edon80", Report 2005/007, [4], 2005.

[14] M. Kasper, S. Kumar, K. Lemke-Rust and C. Paar, "A Compact Implementation of Edon80", Report 2006/057, [4], 2006.

Danilo Gligoroski received his PhD degree in Computer Science from Institute of Informatics, Faculty of Natural Sciences and Mathematics, at University of Skopje – Macedonia in 1997. His research interests are Cryptography, Computer Security, Discrete algorithms and Information Theory and Coding. Currently he is PostDoc at Q2S – Centre for Quantifiable Quality of Service in Communication Systems at Norwegian University of Science and Technology - Trondheim, Norway.

Prof. Svein Johan Knapskog is the head of the “Centre for Quantifiable Quality of Service in Communication Systems – Q2S”, at the Norwegian University of Science and Technology, Trondheim, Norway. His research interests are Network Security, Cryptography, and Security Standards.

Appendix:
C source code of MACEdon80

```
/* ecrypt-sync-MACEdon80.h */
/* ecrypt-sync-MACEdon80.h
 * v1.0   December 2006
 * ECRYPT C api code
 * intervention made by:  Danilo Gligoroski
 *
 * This code is placed as a reference code for ECRYPT
 * call for Stream Ciphers.
 */

/*
 * Header file for synchronous stream ciphers without authentication
 * mechanism.
 *
 * *** Please only edit parts marked with "[edit]". ***
 */

#ifndef ECRYPT_SYNC_AE
#define ECRYPT_SYNC_AE

#include "ecrypt-portable.h"

/* ------------------------------------------------------------------------- */

/* Cipher parameters */

/*
 * The name of your cipher.
 */
#define ECRYPT_NAME "MACEdon80"    /* [edit] */

/*
 * Specify which key and IV sizes are supported by your cipher. A user
 * should be able to enumerate the supported sizes by running the
 * following code:
 *
 * for (i = 0; ECRYPT_KEYSIZE(i) <= ECRYPT_MAXKEYSIZE; ++i)
 *   {
 *     keysize = ECRYPT_KEYSIZE(i);
 *
 *     ...
 *   }
 *
 * All sizes are in bits.
 */

#define ECRYPT_MAXKEYSIZE 80               /* [edit] */
#define ECRYPT_KEYSIZE(i) (80 + (i)*8)      /* [edit] */
/*
  The design of Edon80 in principle is not restricted by the key size.
  However, for ECRYPT call for Stream Ciphers - PROFILE 2, we restrict
  Edon80 key size only on requested size of 80 bits.
*/

#define ECRYPT_MAXIVSIZE 64               /* [edit] */
#define ECRYPT_IVSIZE(i) (64 + (i)*8)      /* [edit] */
/*
  We repeat the same comment from above for IVSIZE. For ECRYPT call for
  Stream Ciphers, we restrict Edon80 key size only on requested size of 64 bits.
*/

/* ------------------------------------------------------------------------- */

/* Data structures */

/*
 * ECRYPT_ctx is the structure containing the representation of the
 * internal state of your cipher.
 */

typedef struct
{
   /* Edon80 is a Stream Cipher based on Quasigropup String Transformations.  */
   /* For the definition of Edon80 we need quasigroups of order 4     */
   u8  Q[ECRYPT_MAXKEYSIZE][4][4];
   /* Counter is internal variable that has values in the range 0 to 3 */
   u8  Counter;
   /* The working size of the key (in pairs of bits). */
   u32 keysize;
   /* The values of the Initial Vector are kept in this array. */
   u8  key[ECRYPT_MAXKEYSIZE/2];
   /* The working size of the Initial Vector (in pairs of bits). */
   u32 ivsize;
   /* The values of the Initial Vector are kept in this array. */
   u8  iv[ECRYPT_MAXKEYSIZE/2];
   /* The actual number of internal states. */
   u32 NumberOfInternalStates;
   /* All internal states are kept in this array. */
   u8  InternalState[ECRYPT_MAXKEYSIZE];
   /* The values of MAC are kept in this array. */
   u8  MAC[ECRYPT_MAXKEYSIZE];
   /* The values 160-bit shift register A are kept in this array. */
   u8  A[ECRYPT_MAXKEYSIZE];
  /*
   * [edit]
   *
   * Put here all state variable needed during the encryption process.
   */
} ECRYPT_ctx;

/* ------------------------------------------------------------------------- */

/* Mandatory functions */

/*
 * Key and message independent initialization. This function will be
 * called once when the program starts (e.g., to build expanded S-box
 * tables).
 */
void ECRYPT_init();

/*
 * Key setup. It is the user's responsibility to select the values of
 * keysize and ivsize from the set of supported values specified
 * above.
 */
void ECRYPT_keysetup(
  ECRYPT_ctx* ctx,
  const u8* key,
  u32 keysize,                /* Key size in bits. */
  u32 ivsize);               /* IV size in bits. */


/*
 * IV setup. After having called ECRYPT_keysetup(), the user is
 * allowed to call ECRYPT_ivsetup() different times in order to
 * encrypt/decrypt different messages with the same key but different
 * IV's.
 */
void ECRYPT_ivsetup(
  ECRYPT_ctx* ctx,
  const u8* iv);
```

```
/*
 * Encryption/decryption of arbitrary length messages.
 *
 * For efficiency reasons, the API provides two types of
 * encrypt/decrypt functions. The ECRYPT_encrypt_bytes() function
 * (declared here) encrypts byte strings of arbitrary length, while
 * the ECRYPT_encrypt_blocks() function (defined later) only accepts
 * lengths which are multiples of ECRYPT_BLOCKLENGTH.
 *
 * The user is allowed to make multiple calls to
 * ECRYPT_encrypt_blocks() to incrementally encrypt a long message,
 * but he is NOT allowed to make additional encryption calls once he
 * has called ECRYPT_encrypt_bytes() (unless he starts a new message
 * of course). For example, this sequence of calls is acceptable:
 *
 * ECRYPT_keysetup();
 *
 * ECRYPT_ivsetup();
 * ECRYPT_encrypt_blocks();
 * ECRYPT_encrypt_blocks();
 * ECRYPT_encrypt_bytes();
 *
 * ECRYPT_ivsetup();
 * ECRYPT_encrypt_blocks();
 * ECRYPT_encrypt_blocks();
 *
 * ECRYPT_ivsetup();
 * ECRYPT_encrypt_bytes();
 *
 * The following sequence is not:
 *
 * ECRYPT_keysetup();
 * ECRYPT_ivsetup();
 * ECRYPT_encrypt_blocks();
 * ECRYPT_encrypt_bytes();
 * ECRYPT_encrypt_blocks();
 */

void ECRYPT_encrypt_bytes(
  ECRYPT_ctx* ctx,
  const u8* plaintext,
  u8* ciphertext,
  u32 msglen);                /* Message length in bytes. */


void ECRYPT_decrypt_bytes(
  ECRYPT_ctx* ctx,
  const u8* ciphertext,
  u8* plaintext,
  u32 msglen);                /* Message length in bytes. */


/* ------------------------------------------------------------------------- */

/* Optional features */

/*
 * For testing purposes it can sometimes be useful to have a function
 * which immediately generates keystream without having to provide it
 * with a zero plaintext. If your cipher cannot provide this function
 * (e.g., because it is not strictly a synchronous cipher), please
 * reset the ECRYPT_GENERATES_KEYSTREAM flag.
 */

#define ECRYPT_GENERATES_KEYSTREAM
#ifdef ECRYPT_GENERATES_KEYSTREAM

void ECRYPT_keystream_bytes(
  ECRYPT_ctx* ctx,
  u8* keystream,
  u32 length);                /* Length of keystream in bytes. */

#endif

/* ------------------------------------------------------------------------- */

/* Optional optimizations */

/*
 * By default, the functions in this section are implemented using
 * calls to functions declared above. However, you might want to
 * implement them differently for performance reasons.
 */

/*
 * All-in-one encryption/decryption of (short) packets.
 *
 * The default definitions of these functions can be found in
 * "ecrypt-sync.c". If you want to implement them differently, please
 * undef the ECRYPT_USES_DEFAULT_ALL_IN_ONE flag.
 */
#define ECRYPT_USES_DEFAULT_ALL_IN_ONE        /* [edit] */

void ECRYPT_encrypt_packet(
  ECRYPT_ctx* ctx,
  const u8* iv,
  const u8* plaintext,
  u8* ciphertext,
  u32 msglen);

void ECRYPT_decrypt_packet(
  ECRYPT_ctx* ctx,
  const u8* iv,
  const u8* ciphertext,
  u8* plaintext,
  u32 msglen);

/*
 * Encryption/decryption of blocks.
 *
 * By default, these functions are defined as macros. If you want to
 * provide a different implementation, please undef the
 * ECRYPT_USES_DEFAULT_BLOCK_MACROS flag and implement the functions
 * declared below.
 */

#define ECRYPT_BLOCKLENGTH 4                  /* [edit] */

#define ECRYPT_USES_DEFAULT_BLOCK_MACROS      /* [edit] */
#ifdef ECRYPT_USES_DEFAULT_BLOCK_MACROS

#define ECRYPT_encrypt_blocks(ctx, plaintext, ciphertext, blocks)  \
  ECRYPT_encrypt_bytes(ctx, plaintext, ciphertext,                 \
    (blocks) * ECRYPT_BLOCKLENGTH)

#define ECRYPT_decrypt_blocks(ctx, ciphertext, plaintext, blocks)  \
  ECRYPT_decrypt_bytes(ctx, ciphertext, plaintext,                 \
    (blocks) * ECRYPT_BLOCKLENGTH)

#ifdef ECRYPT_GENERATES_KEYSTREAM

#define ECRYPT_keystream_blocks(ctx, keystream, blocks)            \
  ECRYPT_AE_keystream_bytes(ctx, keystream,                        \
    (blocks) * ECRYPT_BLOCKLENGTH)

#endif

#else

void ECRYPT_encrypt_blocks(
  ECRYPT_ctx* ctx,
  const u8* plaintext,
  u8* ciphertext,
  u32 blocks);                /* Message length in blocks. */

void ECRYPT_decrypt_blocks(
  ECRYPT_ctx* ctx,
  const u8* ciphertext,
  u8* plaintext,
  u32 blocks);                /* Message length in blocks. */

#ifdef ECRYPT_GENERATES_KEYSTREAM

void ECRYPT_keystream_blocks(
  ECRYPT_AE_ctx* ctx,
  const u8* keystream,
  u32 blocks);                /* Keystream length in blocks. */

#endif

#endif

/* ------------------------------------------------------------------------- */

#endif

******************************************************************************
******************************************************************************
******************************************************************************


/* ecrypt-sync-MACEdon80.c   v1.0   December 2006
 * Reference ANSI C code for MACEdon80
 * author:  v1.0 Danilo Gligoroski
 * Copyright 2006
 *
 */

#include "ecrypt-sync-MACEdon80.h"

/* Here is the actual definition of ECRYPT_keysetup */
void ECRYPT_keysetup( ECRYPT_ctx* ctx, const u8* key, u32 keysize, u32 ivsize)
{
   u32 i, j, m;   /* Variables i, j and m are internal counters. */
   u32 index;     /* Variable index is local temporal variable. */

   u8 Q[4][4][4]={   /* In the design phase we choose this 4 quasigroups of order 4 */
                 /* out of possible 384 good candidates (out of 64 very good candidates). */
      {
         {0, 2, 1, 3}, /* 0: Nr: 61 */
         {2, 1, 3, 0},
         {1, 3, 0, 2},
         {3, 0, 2, 1}
      },
      {
         {1, 3, 0, 2}, /* 1: Nr. 241 */
         {0, 1, 2, 3},
         {2, 0, 3, 1},
         {3, 2, 1, 0}
      },
      {
         {2, 1, 0, 3}, /* 2: Nr. 350 */
         {1, 2, 3, 0},
         {3, 0, 2, 1},
         {0, 3, 1, 2}
      },
      {
         {3, 2, 1, 0}, /* 3: Nr. 564 */
         {1, 0, 3, 2},
         {0, 3, 2, 1},
         {2, 1, 0, 3}
      }
      };

   /* First we store the number of pairs of key bits into ctx->keysize. */
   ctx->keysize=keysize>>1;

   /* Then we transform the received key vector, into a vector of pairs of bits. */
   j=0;
   for (i=0; i<keysize>>3; i++)
   {
      ctx->key[j++]=key[i]>>6;           /* Upper 2 bits of the key[i] */
      ctx->key[j++]=(key[i] & 0x30)>>4;  /* Next  2 bits of the key[i] */
      ctx->key[j++]=(key[i] & 0x0c)>>2;  /* Next  2 bits of the key[i] */
      ctx->key[j++]=key[i] & 0x03;       /* Lower 2 bits of the key[i] */
   }

   /* Then we set the value of ctx->ivsize as a number of iv pairs of bits. */
   ctx->ivsize=ivsize>>1;

   /* Then we set the number of internal states. */
   ctx->NumberOfInternalStates=ECRYPT_MAXKEYSIZE;

   /* Finally we set the working quasigroups Q[][] according to the values of the ctx->key[]. */
   for (m=0; m<keysize>>1; m++)
   {
      index=ctx->key[m];
      for (i=0; i<4; i++)
         for (j=0; j<4; j++)
         {
            ctx->Q[m][i][j]=Q[index][i][j];
            ctx->Q[m+ECRYPT_MAXKEYSIZE/2][i][j]=Q[index][i][j];
         }
   }
}

/* Here is the actual definition of ECRYPT_ivsetup */
void ECRYPT_ivsetup(ECRYPT_ctx* ctx, const u8* iv)
{
   u32 i, j;       /* Variables i and j are internal counters. */
   s32 k;          /* Variable k is internal signed counter. */
   u8 Temp[ECRYPT_MAXKEYSIZE];        /* Temp is a vector of bytes that will temporarily
                              hold the values of the vector InternalStates[] */

   /* First we transform the received iv vector, into a vector of pairs of bits. */
   j=0;
   for (i=0; i<ctx->ivsize>>2; i++)
   {
      ctx->iv[j++]=iv[i]>>6;           /* Upper 2 bits of the iv[i] */
      ctx->iv[j++]=(iv[i] & 0x30)>>4;  /* Next  2 bits of the iv[i] */
      ctx->iv[j++]=(iv[i] & 0x0c)>>2;  /* Next  2 bits of the iv[i] */
      ctx->iv[j++]=iv[i] & 0x03;       /* Lower 2 bits of the iv[i] */
   }

   /* Then we copy vectors ctx.key[] and ctx.iv[] into vectors ctx->InternalState[] and Temp[] */
   for (i=0; i<ctx->keysize; i++) Temp[i]=ctx->InternalState[i]=ctx->key[i];
   for (j=0; j<ctx->ivsize; j++)
   {
      Temp[i]=ctx->InternalState[i]=ctx->iv[j];
      ++i;
```

```
    }

    /*
      Then we pad ctx->InternalState[] and Temp[] with
      16 bits: 0xE41B i.e. with 32100123 (in system of base 4)
    */
    Temp[i]=ctx->InternalState[i]=3;
    ++i;
    Temp[i]=ctx->InternalState[i]=2;
    ++i;
    Temp[i]=ctx->InternalState[i]=1;
    ++i;
    Temp[i]=ctx->InternalState[i]=0;
    ++i;
    Temp[i]=ctx->InternalState[i]=0;
    ++i;
    Temp[i]=ctx->InternalState[i]=1;
    ++i;
    Temp[i]=ctx->InternalState[i]=2;
    ++i;
    Temp[i]=ctx->InternalState[i]=3;
    ++i;

    /*  Finaly we transform the vector InternalState[], with
       Quasigroup e-transformations, with leaders that are
       initial key[] and iv[] elements (i.e. elements in Temp[]),
       ordered in reverse order and by quasigroups determined
       by the value of the key[].
    */
    for (k=i-1; k>=0; k--)
    {
       ctx->InternalState[0]=ctx->Q[k][Temp[k]][ctx->InternalState[0]];
       for (j=1; j<ctx->NumberOfInternalStates; j++)
          ctx->InternalState[j]=ctx->Q[k][ctx->InternalState[j-1]][ctx->InternalState[j]];
    }

    /* We set up the vector InternalState[] as initial MAC[] */
    for (i=0; i<ctx->NumberOfInternalStates; i++) ctx->MAC[i]=ctx->InternalState[i];

    /* We set up the vector InternalState[] as initial A[] */
    for (i=0; i<ctx->NumberOfInternalStates; i++) ctx->A[i]=ctx->InternalState[i];

    /* As a final step in iv setup we prepare the value of internal Counter to 3. */
    ctx->Counter=3;
}

/* Here is the actual definition of ECRYPT_encrypt_bytes */
void ECRYPT_encrypt_bytes(ECRYPT_ctx* ctx, const u8* plaintext, u8* ciphertext, u32 msglen)
{
    u32 i, j;            /* Variables i and j are internal counters. */
    u8 X;                /* We will store produced byte from the keystream in X. */
    u32 MACQueueLength=0; /* Variables MACQueueLength is internal counter.
                         It can be at most ctx->NumberOfInternalStates. */

    for (j=0;j<msglen;j++)
    {
       /* Variable ctx.Counter will varry from 0 to 3, periodically.
         It will be the first feed to the e-transformations by the quasigroups. */
       ctx->Counter++;
       ctx->Counter&=0x03;

       /* Obtaining the first 2 bits from the cipher */
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
       if (MACQueueLength==ctx->NumberOfInternalStates)
       {
          /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
          for (i=MACQueueLength;i>1;i--)
             ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
          ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
       }

       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

       X=(ctx->InternalState[i-1])<<6;
       /* MAC part */
       /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1 */
       MACQueueLength++;
       if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
       /* Insert plaintext two bits into the shift register A */
       for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
       ctx->A[0]=(plaintext[j]>>6);
       /* Perform operations on MAC structure. */
       for (i=MACQueueLength;i>1;i--)
          ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
       ctx->MAC[0]=ctx->Q[0][(plaintext[j]>>6)][ctx->MAC[0]];

       /* Obtaining next 2 bits from the cipher */
       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
       if (MACQueueLength==ctx->NumberOfInternalStates)
       {
          /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
          for (i=MACQueueLength;i>1;i--)
             ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
          ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
       }

       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

       X=X^(ctx->InternalState[i-1]<<4);
       /* MAC part */
       /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1. */
       MACQueueLength++;
       if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
       /* Insert plaintext two bits into the shift register A */
       for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
       ctx->A[0]=((plaintext[j]>>4)&0x03);
       /* Perform operations on MAC structure. */
       for (i=MACQueueLength;i>1;i--)
          ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
       ctx->MAC[0]=ctx->Q[0][((plaintext[j]>>4)&0x03)][ctx->MAC[0]];

       /* Obtaining next 2 bits from the cipher */
       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
       if (MACQueueLength==ctx->NumberOfInternalStates)
       {
          /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
          for (i=MACQueueLength;i>1;i--)
             ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
          ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
       }

       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

       X=X^(ctx->InternalState[i-1]<<2);
       /* MAC part */
       /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1. */
       MACQueueLength++;
       if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
       /* Insert plaintext two bits into the shift register A */
       for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
       ctx->A[0]=((plaintext[j]>>2)&0x03);
       /* Perform operations on MAC structure. */
       for (i=MACQueueLength;i>1;i--)
          ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
       ctx->MAC[0]=ctx->Q[0][((plaintext[j]>>2)&0x03)][ctx->MAC[0]];

       /* Obtaining last 2 bits from the cipher, to form a keystream byte. */
       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
       if (MACQueueLength==ctx->NumberOfInternalStates)
       {
          /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
          for (i=MACQueueLength;i>1;i--)
             ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
          ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
       }

       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

       X=X^ctx->InternalState[i-1];
       /* MAC part */
       /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1. */
       MACQueueLength++;
       if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
       /* Insert plaintext two bits into the shift register A */
       for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
       ctx->A[0]=(plaintext[j] & 0x03);
       /* Perform operations on MAC structure. */
       for (i=MACQueueLength;i>1;i--)
          ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
       ctx->MAC[0]=ctx->Q[0][(plaintext[j] & 0x03)][ctx->MAC[0]];

       /* Finally we XOR the plaintext with X */
       ciphertext[j]=plaintext[j]^X;
    }

    /* Next operations finish the feeding from the register A */
    for (j=0; j<ctx->NumberOfInternalStates; j++)
    {
       MACQueueLength++;
       if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
       /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
       for (i=MACQueueLength;i>1;i--)
          ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
       ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
       /* Shift register A for two bits. */
       for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
    }

    /* Next operations finish the journey of the MAC data trought the pipeline */
    for (j=1; j<ctx->NumberOfInternalStates; j++)
    {
       MACQueueLength++;
       if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
       /* Perform operations on MAC structure. */
       for (i=MACQueueLength;i>j;i--)
          ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
    }
}

/* Here is the actual definition of ECRYPT_decrypt_bytes */
void ECRYPT_decrypt_bytes(ECRYPT_ctx* ctx, const u8* ciphertext, u8* plaintext, u32 msglen)
{
    u32 i, j;      /* Variables i and j are internal counters. */
    u8 X;          /* We will store produced byte from the keystream in X. */
    u32 MACQueueLength=0; /* Variables MACQueueLength is internal counter.
                         It can be at most ctx->keysize. */
    u8 mm;         /* mm is temp variable */

    for (j=0;j<msglen;j++)
    {
       /* Variable ctx.Counter will varry from 0 to 3, periodically.
         It will be the first feed to the e-transformations by the quasigroups. */
       ctx->Counter++;
       ctx->Counter&=0x03;

       /* Obtaining the first 2 bits from the cipher */
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
       if (MACQueueLength==ctx->NumberOfInternalStates)
       {
          /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
          for (i=MACQueueLength;i>1;i--)
             ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
          ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
       }

       ctx->Counter++;
       ctx->Counter&=0x03;
       ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
       for (i=1;i<ctx->NumberOfInternalStates;i++)
          ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

       X=(ctx->InternalState[i-1])<<6;
       mm=(ciphertext[j]>>6)^ctx->InternalState[i-1];
       /* MAC part */
       /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1 */
       MACQueueLength++;
       if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
       /* Insert plaintext two bits into the shift register A */
       for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
       ctx->A[0]=mm;
       /* Perform operations on MAC structure. */
       for (i=MACQueueLength;i>1;i--)
          ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
       ctx->MAC[0]=ctx->Q[0][mm][ctx->MAC[0]];

       /* Obtaining next 2 bits from the cipher */
       ctx->Counter++;
       ctx->Counter&=0x03;
```

```
        ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
        for (i=1;i<ctx->NumberOfInternalStates;i++)
           ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
        if (MACQueueLength==ctx->NumberOfInternalStates)
        {
           /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
           for (i=MACQueueLength;i>1;i--)
              ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
           ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
        }

        ctx->Counter++;
        ctx->Counter&=0x03;
        ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
        for (i=1;i<ctx->NumberOfInternalStates;i++)
           ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

        X=X^(ctx->InternalState[i-1]<<4);
        mm=((ciphertext[j]>>4)&0x03)^ctx->InternalState[i-1];
        /* MAC part */
        /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1. */
        MACQueueLength++;
        if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
        /* Insert plaintext two bits into the shift register A */
        for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
        ctx->A[0]=mm;
        /* Perform operations on MAC structure. */
        for (i=MACQueueLength;i>1;i--)
           ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
        ctx->MAC[0]=ctx->Q[0][mm][ctx->MAC[0]];

        /* Obtaining next 2 bits from the cipher */
        ctx->Counter++;
        ctx->Counter&=0x03;
        ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
        for (i=1;i<ctx->NumberOfInternalStates;i++)
           ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
        if (MACQueueLength==ctx->NumberOfInternalStates)
        {
           /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
           for (i=MACQueueLength;i>1;i--)
              ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
           ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
        }

        ctx->Counter++;
        ctx->Counter&=0x03;
        ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
        for (i=1;i<ctx->NumberOfInternalStates;i++)
           ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

        X=X^(ctx->InternalState[i-1]<<2);
        mm=((ciphertext[j]>>2)&0x03)^ctx->InternalState[i-1];
        /* MAC part */
        /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1. */
        MACQueueLength++;
        if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
        /* Insert plaintext two bits into the shift register A */
        for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
        ctx->A[0]=mm;
        /* Perform operations on MAC structure. */
        for (i=MACQueueLength;i>1;i--)
           ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
        ctx->MAC[0]=ctx->Q[0][mm][ctx->MAC[0]];

        /* Obtaining last 2 bits from the cipher, to form a keystream byte. */
        ctx->Counter++;
        ctx->Counter&=0x03;
        ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
        for (i=1;i<ctx->NumberOfInternalStates;i++)
           ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];
        if (MACQueueLength==ctx->NumberOfInternalStates)
        {
           /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
           for (i=MACQueueLength;i>1;i--)
              ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
           ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
        }

        ctx->Counter++;
        ctx->Counter&=0x03;
        ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
        for (i=1;i<ctx->NumberOfInternalStates;i++)
           ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

        X=X^ctx->InternalState[i-1];
        mm=(ciphertext[j] & 0x03)^ctx->InternalState[i-1];
        /* MAC part */
        /* Variable MACQueueLength will grow from 0 to ctx->NumberOfInternalStates-1. */
        MACQueueLength++;
        if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
        /* Insert plaintext two bits into the shift register A */
        for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
        ctx->A[0]=mm;
        /* Perform operations on MAC structure. */
        for (i=MACQueueLength;i>1;i--)
           ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
        ctx->MAC[0]=ctx->Q[0][mm][ctx->MAC[0]];

        /* Finally we XOR the plaintext with X */
        plaintext[j]=ciphertext[j]^X;
     }

     /* Next operations finish the feeding from the register A */
     for (j=0; j<ctx->NumberOfInternalStates; j++)
     {
        MACQueueLength++;
        if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
        /* Perform operations on MAC structure with the value of A[MACQueueLength-1] */
        for (i=MACQueueLength;i>1;i--)
           ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
        ctx->MAC[0]=ctx->Q[0][ctx->A[MACQueueLength-1]][ctx->MAC[0]];
        /* Shift register A for two bits. */
        for (i=ctx->NumberOfInternalStates; i>1; i--) ctx->A[i-1]=ctx->A[i-2];
     }

     /* Next operations finish the journey of the MAC data trought the pipeline */
     for (j=1; j<ctx->NumberOfInternalStates; j++)
     {
        MACQueueLength++;
        if (MACQueueLength>ctx->NumberOfInternalStates) MACQueueLength=ctx->NumberOfInternalStates;
        /* Perform operations on MAC structure. */
        for (i=MACQueueLength;i>j;i--)
           ctx->MAC[i-1]=ctx->Q[i-1][ctx->MAC[i-2]][ctx->MAC[i-1]];
     }
}

/* Here is the actual definition of ECRYPT_keystream_bytes */
void ECRYPT_keystream_bytes(ECRYPT_ctx* ctx, u8* keystream, u32 length)
{
   u32 i, j;      /* Variables i and j are internal counters. */
   u8 X;          /* We will store produced byte from the keystream in X. */

   for (j=0;j<length;j++)
   {
      /* Variable ctx.Counter will varry from 0 to 3, periodically.
         It will be the first feed to the e-transformations by the quasigroups. */
      ctx->Counter++;
      ctx->Counter&=0x03;

      /* Obtaining the first 2 bits from the ciher */
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      ctx->Counter++;
      ctx->Counter&=0x03;
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      X=(ctx->InternalState[i-1])<<6;

      /* Obtaining next 2 bits from the ciher */
      ctx->Counter++;
      ctx->Counter&=0x03;
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      ctx->Counter++;
      ctx->Counter&=0x03;
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      X=X^(ctx->InternalState[i-1]<<4);

      /* Obtaining next 2 bits from the ciher */
      ctx->Counter++;
      ctx->Counter&=0x03;
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      ctx->Counter++;
      ctx->Counter&=0x03;
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      X=X^(ctx->InternalState[i-1]<<2);

      /* Obtaining last 2 bits from the ciher, to form a keystream byte. */
      ctx->Counter++;
      ctx->Counter&=0x03;
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      ctx->Counter++;
      ctx->Counter&=0x03;
      ctx->InternalState[0]=ctx->Q[0][ctx->InternalState[0]][ctx->Counter];
      for (i=1;i<ctx->NumberOfInternalStates;i++)
         ctx->InternalState[i]=ctx->Q[i][ctx->InternalState[i]][ctx->InternalState[i-1]];

      X=X^ctx->InternalState[i-1];

      /* Finally we feed the keystream with X. */
      keystream[j]=X;
   }
}
```