M209 converter
Two code implementations

Michael Josefsson
v1.0 2019

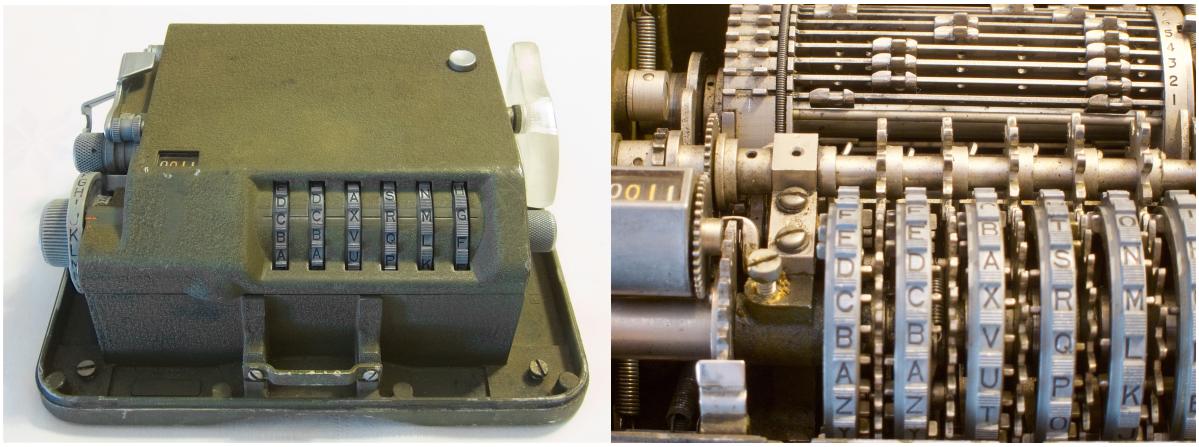
Abstract: This document describes two software implementations in C for the M209 converter. A first implementation is presented as a proof of concept. A second implementation is then developed by parallellizing the algorithm and preprocessing the internal key settings. The resulting optimized algorithm hits a processing speed of some 100 Mchars/second on a 2.8 GHz Intel Core i7 processor. The code is developed in Xcode but also compiles on gcc.

Contents

1.1 Introduction

The M209 cipher machine, also known as a *converter*, stems from Swedish inventor Boris Hagelin's earlier work on machines C-36 and C-38 and was manufactured in the US during WWII in some 140 000 units. It is a purely mechanical device with several mechanical analogs to boolean algebra.

Cryptographically it implements a six bit random generator and some scrambling for selecting a substitution alphabet for each character, `A` to `Z`, coded. Input and output are these characters only.



The M209. The left image shows the user's normal view of it. With the lid open the machines internal settings can be manipulated. The left image shows the major components. From bottom to top there are the wheels, lever linkage *and* cage.

There are several M209 descriptions on the web, I will not repeat them here. This text only describe the parts needed for translating the mechanics into C. A full understanding of the mechanism is of course neccessary for a software implementation.

A software analog of the mechanism is suitably performed in four parts:

1. the wheels, pins and their settings,
2. the cage with 27 bars each with two lugs,
3. the link mechanism of six levers,
4. finally a subtraction unit for selecting an offset alphabet.

The six front wheels have a cycle of $26 \cdot 25 \cdot 23 \cdot 21 \cdot 19 \cdot 17 = 101405850$ steps. This ensures a long and non-recurrent cycle that works as a generator for the rest of the machine. Each individual wheel's pins settings establishes a unique pattern of the same length and this pattern engages lugs on the cage, via a link mechanism, to create a sum that is used for selecting one of 26 alphabets. The wheels and pins effectively produces a six bit random number.

The link mechanism with its six levers conveys this six bit random number to a 'chord' that controls the cage's lugs. As there are six levers thay can present $2^6 = 64$ 'chords' in for each character to be coded. The cage can at most create an offset of 27, which of course is one more than the number of alphabets. This overflow is inconvenient when cracking codes but is of no consequence for a software implementation.

1.2 First implementation

This document describes two main versions of processing code. In each version a message will be enciphered/deciphered and a goal is to reduce processing time for each character so that an efficient algorithm is reached. Another goal is to get a thorough appreciation of how the M209 actually works.

As far as possible has the resulting outputs been checked against a real physical M209.

Actual code is attached and is suggested reading parallell to studying the rest of this document.

First out is a *proof of concept*-version mainly to find out what speeds one can expect and also discover the mechanism. The first version is naturally without any bells and whistles a *correct function* is the major goal.

We begin by preparing the *inner settings*.

1. Initial wheel settings One part of the initial message setup is the starting letter position of the wheels. The wheels are cogged, from left to right, 26, 25, 23, 21, 19 and 17 so that the rightmost wheel will make full revolution after 17 steps, its neighbour will do the same after 19 steps and so on. A *step* is identical to one enciphered character, or turn of the converter's outside handle.

Each letter on each wheel has an associated pin which can be in *active* or *inactive* state. Two irregularities appear

- The wheels have slightly different alphabets on them
  - the 17-wheel is lettered `A..Q`
  - the 19-wheel is lettered `A..S`
  - the 21-wheel is lettered `A..S`
  - the 23-wheel is lettered `A..X` omitting `W`
  - the 25-wheel is lettered `A..Z` omitting `W`
  - the 26-wheel is lettered `A..Z`
- An active pin engages the levers at different times
  - the 17-wheel engages at `H` (+7 steps)
  - the 19-wheel engages at `I` (+8 steps)
  - the 21-wheel engages at `J` (+9 steps)
  - the 23-wheel engages at `K` (+10 steps)
  - the 25-wheel engages at `L` (+11 steps)
  - the 26-wheel engages at `L` (+11 steps)

The actual setting of the pins as either active *or* inactive *is paramount for the security of the system operating instructions indicate that some 40 % to 60 % of the pins should be engaged in actual use. Too few, or too many, upsets the randomness and should be avoided. This setting is tabulated in advance and distributed by safe channels.*

An array, `char wheel_pin[][]`, contains the settings and a mapping to `int active_pin[][]` sets the pins that are active in their correct relative position:

```
char wheel_pin[NUM_WHEELS][26] = {
   "ABDHKNOQ",        // 17
   "BDEFHIMNPS",      // 19
   "CEFHIMNPSTU",     // 21
   "ABGHJRSTUVX",     // 23
   "DEGJKLORSUX",     // 25
   "ABDHIKMSTVW"};    // 26

   int active_pin[NUM_WHEELS][26];
   int c, i, n;
   for( i = 0; i != NUM_WHEELS; i++)
      for( c = 0; c != 26; c++){
         active_pin[i][c] = 0;
      }
   // wheel 0/17, 1/19, 2/21, 3/23, 4/25, 5/26
   for( i = 0; (c = wheel_pin[0][i++]) != '\0'; active_pin[0][((c-'A') + 7) % 17] = 1);
   for( i = 0; (c = wheel_pin[1][i++]) != '\0'; active_pin[1][((c-'A') + 8) % 19] = 1);
   for( i = 0; (c = wheel_pin[2][i++]) != '\0'; active_pin[2][((c-'A') + 9) % 21] = 1);
   for( i = 0; (c = wheel_pin[3][i++]) != '\0';) // or -'B' if c=='X'
      if(c == 'X')
         active_pin[3][((c-'B') + 10) % 23] = 1;
      else
         active_pin[3][((c-'A') + 10) % 23] = 1;
   for( i = 0; (c = wheel_pin[4][i++]) != '\0';) // or -'B' if c in 'X','Y','Z'
      if(c == 'X' || c == 'Y' || c == 'Z')
         active_pin[4][((c-'B') + 11) % 25] = 1;
      else
         active_pin[4][((c-'A') + 11) % 25] = 1;
   for( i = 0; (c = wheel_pin[5][i++]) != '\0'; active_pin[5][((c-'A') + 11) % 26] = 1);
```

2. Initial cage settings The cage contains 27 bars, each with two lugs that can be, again, either *active* or *inactive*. In this software the cage initial setting is also an array, `static int cage_bar[][]`. For historical reasons[1] the columns are in reverse order as to an actual machine. Wheel 17 corresponds to `cage_bar[n][0]` etc.

```
static int  cage_bar[27][6]= {
//  6  5  4  3  2  1 <-- actual machine markings i.e. backwards!
   {1, 1, 0, 0, 0, 0},  //0
   {1, 0, 0, 0, 0, 0},  //1
   {0, 0, 1, 0, 0, 0},  //2
   {0, 1, 1, 0, 0, 0},  //3
   {1, 0, 1, 0, 0, 0},  //4
   {0, 1, 1, 0, 0, 0},  //5
   {0, 1, 0, 1, 0, 0},  //6
   {0, 0, 0, 1, 0, 0},  //7
   {0, 0, 0, 1, 0, 0},  //8
   {0, 0, 0, 1, 0, 0},  //9
   {0, 0, 0, 1, 0, 0},  //10
   {0, 0, 0, 0, 1, 0},  //11
   {0, 0, 0, 0, 1, 0},  //12
   {0, 0, 0, 0, 1, 0},  //13
   {0, 0, 0, 0, 1, 0},  //14
   {0, 0, 0, 0, 1, 0},  //15
   {0, 0, 0, 0, 1, 0},  //16
   {0, 0, 0, 0, 1, 0},  //17
   {0, 0, 0, 0, 0, 1},  //18
   {0, 0, 0, 0, 0, 1},  //19
   {0, 0, 0, 0, 0, 1},  //20
   {0, 0, 0, 0, 0, 1},  //21
   {0, 0, 0, 0, 0, 1},  //22
   {0, 0, 0, 0, 0, 1},  //23
   {0, 0, 0, 0, 0, 1},  //24
   {0, 0, 0, 0, 0, 1},  //25
   {0, 0, 0, 0, 0, 1}   //26
};
```

The lug settings greatly affects the amount of substitution alphabets used and their setting is, as the pin settings above, crucial for the randomness of the output. In the extreme case that no lug is active the cipher output is not enciphered at all. If only one column is active then each character is encoded with only a few substitution alphabets.

[1] Oh, how useful isn't that phrase!

The wheels move stepwise as each letter is enciphered/deciphered, but *not* as an odometer, they all step forward in parallell at once:

```
AAAAAA
BBBBBB
CCCCCC
::::::
QQQQQQ
RRRRRA
SSSSSB
TTTTAC
UUUUBD
::::::
::::::
AAAAAA    <-- after 26*25*23*21*19*17 = 101 405 850 steps
```

a function which is achieved on the six wheels in `int wheel[]` by this code where a wheel's circumference is defined in `w_len[]` as

```
static char w_len[NUM_WHEELS] = { 17, 19, 21, 23, 25, 26};

for (int n = 0; n < NUM_WHEELS; n++){
   wheel[n]++;
   if(wheel[n] == w_len[n]) wheel[n] = 0;
}
```

3. Active levers If a pin is active it will affect the lever linkage mechanism and make that lever active as well. For each new wheel position the active levers must be re-evaluated. The corresponding code acts on a zeroized array.

```
char active_lever[6]={ 0, 0, 0, 0, 0, 0}; // connecting levers inactive

// E.g. if pin 5 is active on wheel then wheel[n] is 5, and active_lever = 1
for(char n = 0; n < NUM_WHEELS; n++){
   active_lever[n]= active_pin[n][wheel[n]];
}
```

4. Counting lugs For each turn of the cage the lugs corresponding to an active lever must be counted. Since there are 27 bars a loop will do the trick and the accumulated count is stored in the variable `offset`:

```
uint8_t offset = 0;

for(char i = 0; i < NUM_BARS; i++){
   for(char n = 0; n < NUM_WHEELS; n++){
      if (active_lever[n] && cage_bar[i][n]){
         offset++;   // max one count per each bar
         break;
      }
   }
}
```

Enciphering/deciphering With the `offset` from above a character from the message is processed by subtracting it from the ASCII-value of the message character. The message is defined in a character array as

```
char message[] = "HIGHER";
```

and the enciphering code is trivial:

```
u_int8_t p = message[k]-offset;

   while (p < 'A') p += 26;
   while (p > 'Z') p -= 26;
   output[k] = ZA[p-'A'];
```

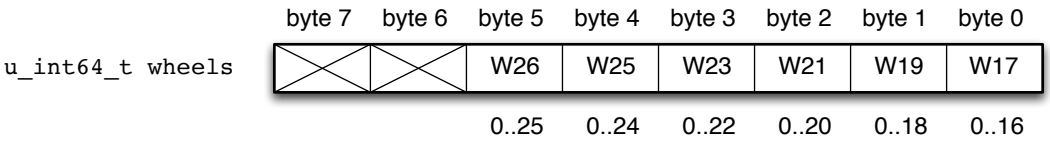
where an array `ZA[]` with a reversed alphabet is used as lookup:

```
static char ZA[26] = "ZYXWVUTSRQPONMLKJIHGFEDCBA";
```

Optimised as `-Ofast` this runs at about 10 Mchars. This far it was also noted that using `char`s is slower than using `int`s even though the individual number fit well into bytes.

1.3 Two apparent speedups

One time consuming part of the program is the repetitive bumping of the wheels one step for each enciphered character. Given that the processor available has 64 bit registers a quicker version of the wheel stepping uses one entire `u_int64_t` for *all* wheels:



Code for stepping through these wheels is equivalent to

```
#define W17    0xff
#define W19    0xff00
#define W21    0xff0000
#define W23    0xff000000UL
#define W25    0xff00000000UL
#define W26    0xff0000000000UL

wheels += 0x0000010101010101L;

int W17addr = ((wheels >> 0) & 0xff);
int W19addr = ((wheels >> 8) & 0xff);
int W21addr = ((wheels >>16) & 0xff);
int W23addr = ((wheels >>24) & 0xff);
int W25addr = ((wheels >>32) & 0xff);
int W26addr = ((wheels >>40) & 0xff);

if(W17addr == 17){ wheels = wheels & ~W17; W17addr = ((wheels >> 0) & 0xff);}
if(W19addr == 19){ wheels = wheels & ~W19; W19addr = ((wheels >> 8) & 0xff);}
if(W21addr == 21){ wheels = wheels & ~W21; W21addr = ((wheels >>16) & 0xff);}
if(W23addr == 23){ wheels = wheels & ~W23; W23addr = ((wheels >>24) & 0xff);}
if(W25addr == 25){ wheels = wheels & ~W25; W25addr = ((wheels >>32) & 0xff);}
if(W26addr == 26){ wheels = wheels & ~W26; W26addr = ((wheels >>40) & 0xff);}
```

Where the latter part can be written more efficiently as

```
if( W17addr == 17 ){ wheels &= ~W17; }
if( W19addr == 19 ){ wheels &= ~W19; }
if( W21addr == 21 ){ wheels &= ~W21; }
if( W23addr == 23 ){ wheels &= ~W23; }
if( W25addr == 25 ){ wheels &= ~W25; }
if( W26addr == 26 ){ wheels &= ~W26; }
```

The stepping is accomplished by adding `0x0000010101010101L` to `u_int64_t wheels`. Testing for each wheels maximum value is then unfortunately made in sequence, however this device was worthwhile as it was significantly faster.

The trick of assigning `int W17addr = ((wheels >> 0) & 0xff)` etc meant that this value could be re-used directly later in the code.[2]

Another time consuming part is the memory lookup in the `ZA`-array. It turns out that this is an unnecessary step as

```
u_int8_t p = message[k]-offset;

while (p < 'A') p += 26;
output[k] = 155-p;
```

will do the same for ASCII-values.[3].

[2] It was also noted that, for example,

`(wheels >> 24) & 0xff`

was faster than

`(wheels & W23) >> 24`.

Probably as a 24 bit shift can be done in one instruction, as opposed to first *and*-ing with a 64-bit constant and the shift the entire kaboodle.

[3] In an even earlier version of the code the entire alphabet was remapped `[A..Z]=[0..25]`, a hideously unnecessary contraption!

1.4 Faster version

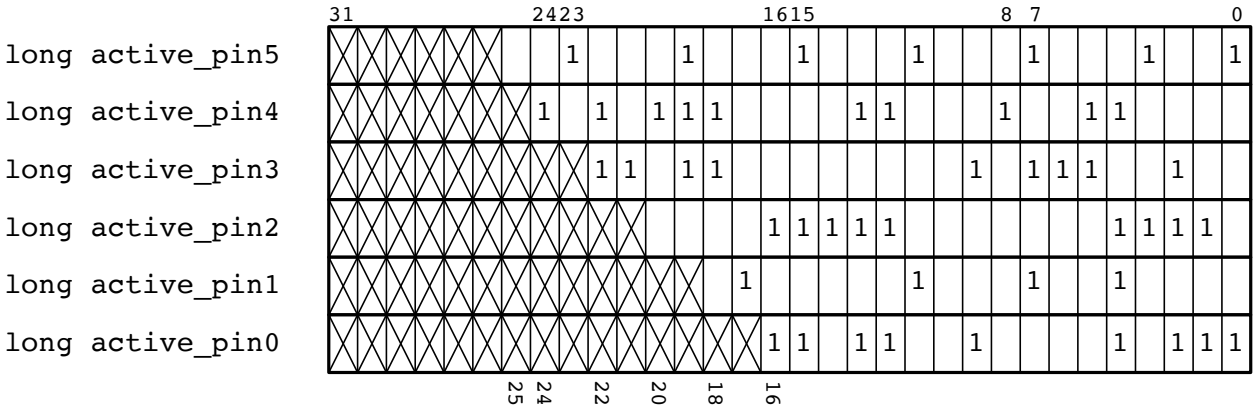
With the parallellization, where the six wheels is contained in one 64 bit number, continued work tried to build on that success. Efforts to make all stages parallell turned out to be time well spent and are used in the current version.

Preprocessing the pins on the wheels was another enhancing part whereby memory fetches were significantly reduced. Each wheel is in one 64 bit variable where an active pin is a set bit and an unactive one is an unset bit.

`Active_pin0` contains wheel 17's active pins, `active_pin1` wheel 19's pins etc. They are counted from bit0 as the least significant bit.

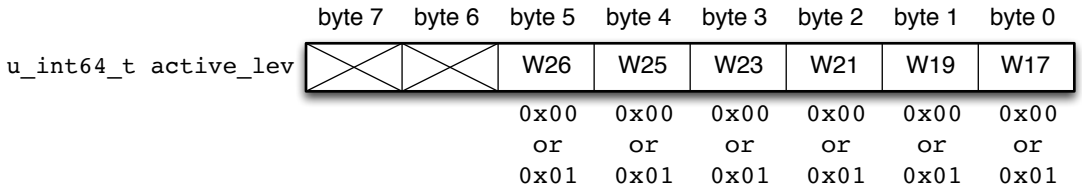
```
unsigned long active_pin0 = 0UL;
unsigned long active_pin1 = 0UL;
unsigned long active_pin2 = 0UL;
unsigned long active_pin3 = 0UL;
unsigned long active_pin4 = 0UL;
unsigned long active_pin5 = 0UL;

// map wheel[0][pin0-16] to active_pin0 bits0-16
for( i = 0; i < 17;  i++){ active_pin0 |= active_pin[0][i]<<i; };
for( i = 0; i < 19;  i++){ active_pin1 |= active_pin[1][i]<<i; };
for( i = 0; i < 21;  i++){ active_pin2 |= active_pin[2][i]<<i; };
for( i = 0; i < 23;  i++){ active_pin3 |= active_pin[3][i]<<i; };
for( i = 0; i < 25;  i++){ active_pin4 |= active_pin[4][i]<<i; };
for( i = 0; i < 26;  i++){ active_pin5 |= active_pin[5][i]<<i; };
```

Now each bit in `active_pin5..active_pin0`, when seen in parallell, e.g. bit 4 of all of them, correspond to which lever should be active. In the figure below is depicted how each active pin is a single bit.



Each wheel's setting now find the corresponding bit, either `0` or `1`, before creating the `active_lever`-setting.

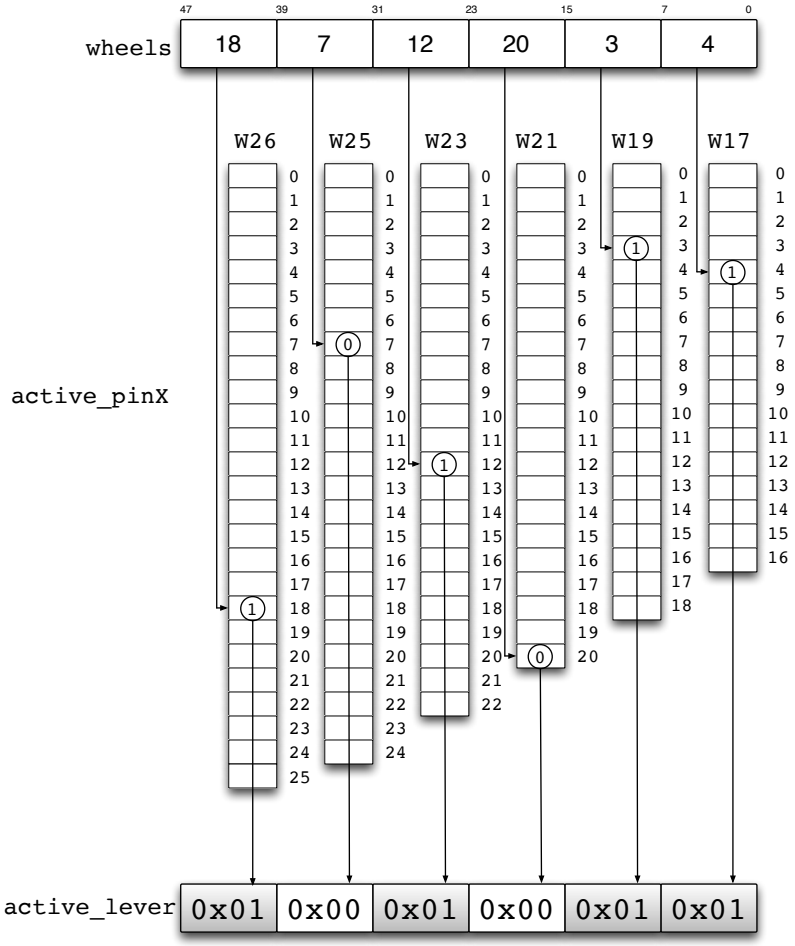
The required active levers are determined as




```
u_int64_t active_lev = 0L;

active_lev |= 0x1 & (active_pin5 >> W26addr);
active_lev <<= 8;
active_lev |= 0x1 & (active_pin4 >> W25addr);
active_lev <<= 8;
active_lev |= 0x1 & (active_pin3 >> W23addr);
active_lev <<= 8;
active_lev |= 0x1 & (active_pin2 >> W21addr);
active_lev <<= 8;
active_lev |= 0x1 & (active_pin1 >> W19addr);
active_lev <<= 8;
active_lev |= 0x1 & (active_pin0 >> W17addr);
```

This step is a nuisance as it is not as elegant as one would hope for. It is however slightly speeded up by the precalculated values `Wxxaddr`.

The process this far can be depicted as



The final offsets can now be evaluated by running through all 27 bars:

```
u_int8_t offset = 0;

for(n = 0; n < NUM_BARS; n++){   // run through all 27 bars
   if (active_lev & cage[n]) offset++;
}
```

Preprocessing the cage was also a speed enhancer. One notices that the lugs come in three variants:

1. Lugs that are in resting position '0' and never used
2. Lugs where only *one* is active on a bar. These lugs increment `offset` by 1.
3. Lugs where *two* are active on a bar. One each bar only *one* is used for incrementing the `offset` by 1.

From each of the observations above one can conclude:

1. These do not need to be processed at all.
2. These can conveniently be precalculated and add only *once*. If all lugs are of this 'single' category evaluation of the offset is done in more or less zero time.
3. These can be activated by any (of two, of course) `active_lever` in the correct position. So these need to be checked one by one.

Preprocessing, i.e. preprocessing once only for each new cage setting, is straightforward in code:

```
u_int64_t cage[27];

for( int i = 0; i < NUM_BARS;i++){
   cage[i] = 0UL;
   for(int n = NUM_WHEELS-1; n > -1; n--){
      cage[i]<<=8;
      if(cage_bar[i][n]){ cage[i] |= 0xff; } else { cage[i] |= 0x00;}
   }
}
```

The new cage is contained in its entirety in 27 64-bit `ints` in the array `cage[]`.

Further preprocessing reduces the number of bars that need to be evaluated at runtime.

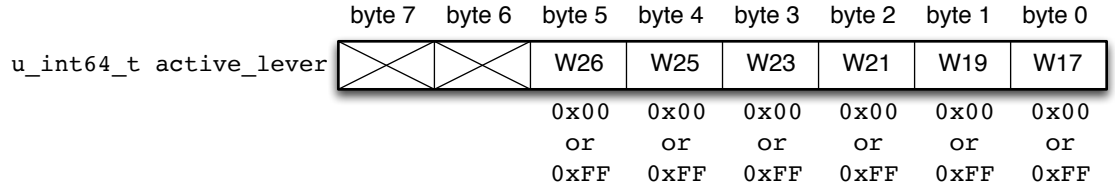
```
// single_bar: column sum of single cage lugs
u_int64_t single_bar = 0UL;
// double_bar: array of cage bars with double lugs, at most all 27
u_int64_t double_bar[NUM_BARS] = { 0UL, 0UL, 0UL, ... all 27 of them ... 0UL, 0UL };
int sum;

// find singles and doubles
int j = 0;      // index in double_bar
for( i = 0; i < NUM_BARS; i++ ){ // for each bar
   sum = 0;
   for( n = 0; n < NUM_WHEELS; n++ ){
      sum += cage_bar[i][n];  // sum lugs on that bar
   }
   // only sum == 1 or sum == 2 possible
   switch( sum ){
      case 1:  // was just one 1, then safe to add it
         for( n = 0; n < NUM_WHEELS ; n++){
            single_bar += (u_int64_t)cage_bar[i][n] << n*8;
         };
         break;
      case 2:  // copy this bar to list of double_bars
         double_bar[j++] = cage[i] ;
         break;
      default:
         printf("ERROR! Excess lugs in cage!\n");
         exit(1);
   }
}

int double_bar_size = j;
```

Now we have two arrays:

- `single_bar[]` with sum of single lugs for each column e.g. `single_bar[] = {9, 7, 4, 1, 0, 1};`
- `double_bar[0..j]` with index of `cage_bar[]` with two lugs active

Also note that all `cage_bars` with NO lugs are conveniently discarded and ignored from now on.

With these two arrays the time and complexity of the evaluation step is drastically reduced to practically only the bars that are of type 3. For this to be quick it is also beneficial to calculate the `active_levers` as values `0xFF` instead as before or `0x01`, or or `0x00`:



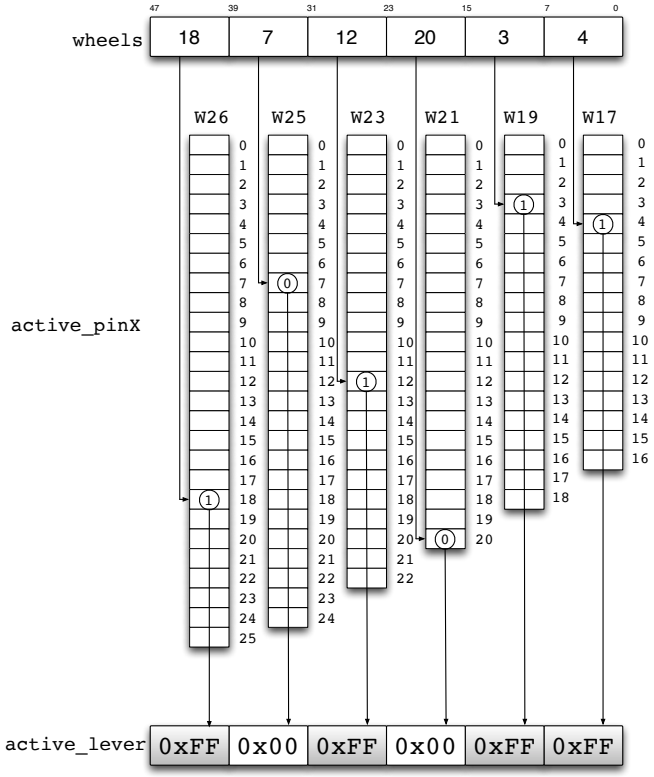
which is accomplished with code according to

```
u_int64_t active_lever = 0L;

active_lever |=  (0x01 & active_pin5>>W26addr) ? 0xFF : 0x00;
active_lever <<= 8;
active_lever |=  (0x01 & active_pin4>>W25addr) ? 0xFF : 0x00;
active_lever <<= 8;
active_lever |=  (0x01 & active_pin3>>W23addr) ? 0xFF : 0x00;
active_lever <<= 8;
active_lever |=  (0x01 & active_pin2>>W21addr) ? 0xFF : 0x00;
active_lever <<= 8;
active_lever |=  (0x01 & active_pin1>>W19addr) ? 0xFF : 0x00;
active_lever <<= 8;
active_lever |=  (0x01 & active_pin0>>W17addr) ? 0xFF : 0x00;
```

which, with after several runs, turns out faster if written as

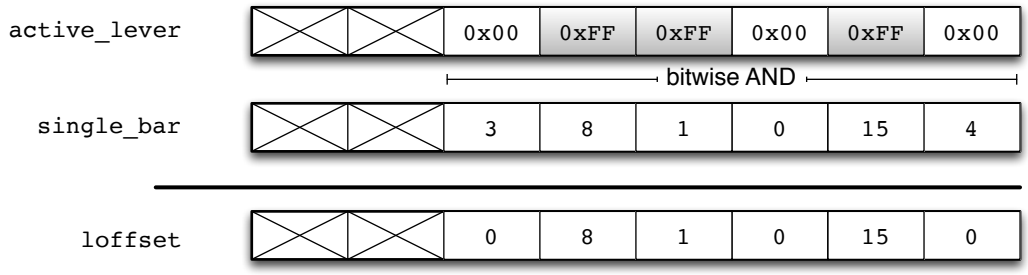
```
u_int64_t active_lever = 0L;

if((0x01 & active_pin5>>((wheels >> 40) & 0xff))) active_lever |= 0xff;
active_lever <<= 8;
if((0x01 & active_pin4>>((wheels >> 32) & 0xff))) active_lever |= 0xff;
active_lever <<= 8;
if((0x01 & active_pin3>>((wheels >> 24) & 0xff))) active_lever |= 0xff;
active_lever <<= 8;
if((0x01 & active_pin2>>((wheels >> 16) & 0xff))) active_lever |= 0xff;
active_lever <<= 8;
if((0x01 & active_pin1>>((wheels >>  8) & 0xff))) active_lever |= 0xff;
active_lever <<= 8;
if((0x01 & active_pin0>>((wheels >>  0) & 0xff))) active_lever |= 0xff;
```

The above remarks can be concluded in the figure below:



This turns out to be clever as it also reduces the computation needed, as finding all singles now is a mere bitwise logical `AND`-operation. Along the same logic the preprocessed list of double-lugged bars is analyzed and offset is incremented when the bitwise `AND` between one such and `active_lever` is not equal to zero. The increment is performed in the lowest byte of `loffset` which is okay as the accumulated number cannot overflow in a byte.

With `active_lever` thusly prepared `loffset` is calculated in one operation.



With corresponding code as:

```
u_int64_t loffset = 0UL;

// add single bars by anding 0xff on each number
// each bars individual contribution is spread in
// the lower NUM_WHEELS bytes of loffset.

loffset = single_bar & active_lever;


//then add each double bar:


for( i = 0; i < double_bar_size ; i++ ){
   if ( active_lever & double_bar[i] ){ // one hit is enough
      loffset++;
   }
}
```

The final step adds all the individual terms in `loffset`'s six lowest bytes to the final `offset` used in the enciphering step. Again no overflow can occur.

```
// now add the offsets for each wheel together to
// wheel 17's place
offset = 0;
for( i = 0; i < NUM_WHEELS; i++ ){
   offset += (char)( loffset & 0xff );
   loffset >>= 8;
}
```

Optimised as `-Ofast` this runs at about 100 Mchars/sec, some 8 times faster than the initial approach and the entire key space of the wheels is completed in about one second. This speedup is mainly due to the parallellization and the preprocessing of the cage.

C-code for the final code version is supplied in the following pages.

1.5 Code for second version

```
//
//  main.c
//  M209
// Speedier version with bits in variables instead of ints in arrays
//  With cage preprocessing for peaks of 100 Mchar/sec
//  Created by Michael Josefsson on 2018-12-25.
//  Copyright 2018 Michael Josefsson. All rights reserved.
//

#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
```

```
#define NUM_WHEELS   6
#define NUM_BARS     27
#define WHEEL_SPACE  (26*25*23*21*19*17)

// below select NONE or ONE of SPEEDTEST and KEYSPEED
// If NONE defined normal coding is performed

// SPEEDTEST for running RUNLEN number of codings
// used for timing efficient coding
#define SPEEDTEST

// KEYSPEED for running RUNLEN number of key schedules
// used for timing efficient key scheduling
// Key scheduling is much more rare than coding
//#define KEYSPEED

#ifdef KEYSPEED
#define RUNLEN 1000000L
#endif

#ifdef SPEEDTEST

#define RUNLEN WHEEL_SPACE //1000000UL
char output[RUNLEN];
char message[] = "H";
char coded_output[] = "Y";

#else
// These are matching texts!
char message[] ="THEZMESSAGE";
char coded_output[] = "MWRASWBZHBO";
char output[100];
#endif

// KEY SETTING ========== START EDITING HERE

char wheel_pin[NUM_WHEELS][26] = {
   "ABDHKNOQ",       // 17
   "BDEFHIMNPS",     // 19
   "CEFHIMNPSTU",    // 21
   "ABGHJRSTUVX",    // 23
   "DEGJKLORSUX",    // 25
   "ABDHIKMSTVW"};   // 26

// cage_bars, note backwardity!
// max 2 active lugs in each row
// cage_bar[3][1] is bar 4, column 5
// column '6' is for wheel 17
static int  cage_bar[27][6]= {
   //  6  5  4  3  2  1 <-- actual machine markings i.e. backwards!
   {1, 1, 0, 0, 0, 0}, //0
   {1, 0, 0, 0, 0, 0}, //1
   {0, 0, 1, 0, 0, 0}, //2
   {0, 1, 1, 0, 0, 0}, //3
   {1, 0, 1, 0, 0, 0}, //4
   {0, 1, 1, 0, 0, 0}, //5
   {0, 1, 0, 1, 0, 0},  //6
   {0, 0, 0, 1, 0, 0},  //7
   {0, 0, 0, 1, 0, 0},  //8
   {0, 0, 0, 1, 0, 0},  //9
   {0, 0, 0, 1, 0, 0},  //10
   {0, 0, 0, 0, 1, 0},  //11
   {0, 0, 0, 0, 1, 0},  //12
   {0, 0, 0, 0, 1, 0},  //13
   {0, 0, 0, 0, 1, 0},  //14
   {0, 0, 0, 0, 1, 0},  //15
   {0, 0, 0, 0, 1, 0},  //16
   {0, 0, 0, 0, 1, 0},  //17
   {0, 0, 0, 0, 0, 1},  //18
   {0, 0, 0, 0, 0, 1},  //19
   {0, 0, 0, 0, 0, 1},  //20
   {0, 0, 0, 0, 0, 1},  //21
   {0, 0, 0, 0, 0, 1},  //22
   {0, 0, 0, 0, 0, 1},  //23
   {0, 0, 0, 0, 0, 1},  //24
   {0, 0, 0, 0, 0, 1},  //25
   {0, 0, 0, 0, 0, 1}   //26
};

// KEY SETTING ========== STOP EDITING HERE
int main(int argc, const char * argv[]) {

   // Dont move the key setting to a function!
   // The call overhead slows it by 10x
   //KEY SCHEDULE SETTING START==========================================
#ifdef KEYSPEED
   clock_t keystart = clock();
   for( int speed = 0; speed < RUNLEN; speed++ ){
```

```
#endif
      // re-map letters above to actual pins
      // active_pin[i] = { 1, 0, 1, ... } <=> AC.. but with wheel offset
      // clear all before populating
      int active_pin[NUM_WHEELS][26];
      int c, i, n;
      for( i = 0; i != NUM_WHEELS; i++ )
         for( c = 0; c != 26; c++ ){
            active_pin[i][c] = 0;
         }
      //#wheel 0/17, 1/19, 2/21, 3/23, 4/25, 5/26
      for( i = 0; ( c = wheel_pin[0][i++] ) != '\0';  active_pin[0][ ((c-'A') + 7) % 17 ] = 1 );
      for( i = 0; ( c = wheel_pin[1][i++] ) != '\0';  active_pin[1][ ((c-'A') + 8) % 19 ] = 1 );
      for( i = 0; ( c = wheel_pin[2][i++] ) != '\0';  active_pin[2][ ((c-'A') + 9) % 21 ] = 1 );
      for( i = 0; ( c = wheel_pin[3][i++] ) != '\0';) // or -'B' if c=='X'
         if(c == 'X')
            active_pin[3][ ((c-'B') + 10) % 23 ] = 1;
         else
            active_pin[3][ ((c-'A') + 10) % 23 ] = 1;
      for( i = 0; (c = wheel_pin[4][i++] ) != '\0';) // or -'B' if c in 'X','Y','Z'
         if(c == 'X' || c == 'Y' || c == 'Z')
            active_pin[4][ ((c-'B') + 11) % 25 ] = 1;
         else
            active_pin[4][ ((c-'A') + 11) % 25 ] = 1;
      for( i = 0; ( c = wheel_pin[5][i++] ) != '\0';  active_pin[5][ ((c-'A') + 11) % 26 ] = 1 );

      // For new structure, map the arrays above to individual bit-patterns
      // active_pinX={b31, b30, b29, b28,..., b4, b3, b2, b1, b0}
      // where a '1'-bit is in position Y as in active_pin[X][Y]=1 above
      u_int64_t active_pin0 = 0UL;
      u_int64_t active_pin1 = 0UL;
      u_int64_t active_pin2 = 0UL;
      u_int64_t active_pin3 = 0UL;
      u_int64_t active_pin4 = 0UL;
      u_int64_t active_pin5 = 0UL;

      // map wheel[0][pin0-16] to active_pin0 bits0-16
      for( i = 0; i < 17; i++ ){ active_pin0 |= active_pin[0][i] << i; };
      for( i = 0; i < 19; i++ ){ active_pin1 |= active_pin[1][i] << i; };
      for( i = 0; i < 21; i++ ){ active_pin2 |= active_pin[2][i] << i; };
      for( i = 0; i < 23; i++ ){ active_pin3 |= active_pin[3][i] << i; };
      for( i = 0; i < 25; i++ ){ active_pin4 |= active_pin[4][i] << i; };
      for( i = 0; i < 26; i++ ){ active_pin5 |= active_pin[5][i] << i; };

      // preprocess cage, expand into register.
      // Array above is turned into
      // new versions in u_int64_t cage[27]
      // W17 is LSByte
      u_int64_t cage[27];

      for( int i = 0; i < NUM_BARS; i++ ){
         cage[i] = 0UL;
         for( int n = NUM_WHEELS-1; n>-1; n-- ){
            cage[i]<<=8;
            if( cage_bar[i][n] ){ cage[i] |= 0x01; } else { cage[i] |= 0x00; }
         }
      }

      // increased speed
      // single_bar: column sum of single cage lugs
      u_int64_t single_bar = 0UL;
      // double_bar: array of cage bars with double lugs, at most all 27
      u_int64_t double_bar[NUM_BARS];

      for( i = 0; i < NUM_BARS; i++ ) double_bar[i] = 0UL;

      int sum;

      // find singles and doubles
      int j = 0;     // index in double_bar
      for( i = 0; i < NUM_BARS; i++ ){ // for each bar
         sum = 0;
         for( n = 0; n < NUM_WHEELS; n++ ){
            sum += cage_bar[i][n];  // sum lugs on that bar
         }
         // only sum == 1 or sum == 2 possible
         switch( sum ){
            case 1:  // was just one 1, then safe to add it
               for( n = 0; n < NUM_WHEELS ; n++){
                  single_bar += (u_int64_t)cage_bar[i][n] << n*8;
               };
               break;
            case 2:  // copy this bar to list of double_bars
               double_bar[j++] = cage[i] ;
               break;
            default:
               printf("ERROR! Excess lugs in cage!\n");
               exit(1);
```

```
         }
      }

      int double_bar_size = j;
      // Now we have two arrays:
      // single_bar = {9, 7, 4, 1, 0, 1}; with sum of single lugs for each column (9 is W26)
      // double_bar[0..j] = index of (at most all) cage_bar with two lugs active
      // cage_bars with NO lugs are discarded from now on

      // KEY SCHEDULE END


#ifdef KEYSPEED
   }
   clock_t keyend = clock();
   float keysecs = (float)(keyend - keystart) / CLOCKS_PER_SEC;
   printf("\nTime: %f ms, ==> %.0f keyssched/sec\n",1000*keysecs, RUNLEN/keysecs);
#endif

#ifndef KEYSPEED
   // CODING / DECODING
   u_int64_t wheels = 0x00UL;
   u_int8_t offset;

   clock_t start = clock();

#ifdef SPEEDTEST
   for( u_int64_t k = 0; k < RUNLEN; k++){
#else
      for( u_int64_t k = 0UL; k < strlen(message); k++){
#endif

         // use one u_int64_t for all wheels '--AAAAAA', '--AAAAAB', '--AAAAAC'...

         // masks for wheels 00|00|26|25|23|21|19|17
#define W17    0xffUL
#define W19    0xff00UL
#define W21    0xff0000UL
#define W23    0xff000000UL
#define W25    0xff00000000UL
#define W26    0xff0000000000UL


   //bump wheels
   wheels += 0x0000010101010101L;

   int W17addr = ((wheels >> 0) & 0xff);
   int W19addr = ((wheels >> 8) & 0xff);
   int W21addr = ((wheels >>16) & 0xff);
   int W23addr = ((wheels >>24) & 0xff);
   int W25addr = ((wheels >>32) & 0xff);
   int W26addr = ((wheels >>40) & 0xff);

   if( W17addr == 17 ){ wheels &= ~W17; }
   if( W19addr == 19 ){ wheels &= ~W19; }
   if( W21addr == 21 ){ wheels &= ~W21; }
   if( W23addr == 23 ){ wheels &= ~W23; }
   if( W25addr == 25 ){ wheels &= ~W25; }
   if( W26addr == 26 ){ wheels &= ~W26; }

   // For the letters of the wheel above after one step forward
   // find out if each wheel's pins are in active position or not
   // Active position means active_lever = 1
   // active_lever is this wheel's index
   // E.g. if pin 5 is active on wheel then wheel[n] is 5, and active_lever = 1
   u_int64_t active_lever = 0L;

   if((0x01 & active_pin5>>((wheels >> 40) & 0xff))) active_lever |= 0xff;
   active_lever <<= 8;
   if((0x01 & active_pin4>>((wheels >> 32) & 0xff))) active_lever |= 0xff;
   active_lever <<= 8;
   if((0x01 & active_pin3>>((wheels >> 24) & 0xff))) active_lever |= 0xff;
   active_lever <<= 8;
   if((0x01 & active_pin2>>((wheels >> 16) & 0xff))) active_lever |= 0xff;
   active_lever <<= 8;
   if((0x01 & active_pin1>>((wheels >>  8) & 0xff))) active_lever |= 0xff;
   active_lever <<= 8;
   if((0x01 & active_pin0>>((wheels >>  0) & 0xff))) active_lever |= 0xff;

   u_int64_t loffset = 0UL;

   // add single bars by anding 0xff on each number
   // each bars individual contribution is spread in
   // the lower NUM_WHEELS bytes of loffset.

   loffset = single_bar & active_lever;
```

```
   //then add each double bar:

   for( i = 0; i < double_bar_size ; i++ ){
      if ( active_lever & double_bar[i] ){ // one hit is enough
         loffset++;
      }
   }

   // now add the offsets for each wheel together to
   // wheel 17's place
   offset = 0;
   for( i = 0; i < NUM_WHEELS; i++ ){
      offset += (char)( loffset & 0xff );
      loffset >>= 8;
   }


#ifdef SPEEDTEST
   u_int8_t p = message[0]-offset;
   while (p < 'A') p += 26;
   output[0] = 155-p;
#else
   u_int8_t p = message[k]-offset;
   while (p < 'A') p += 26;
   output[k] = 155-p;
#endif
   }
   clock_t end = clock();
   float secs = (float)( end - start ) / CLOCKS_PER_SEC;

#ifdef SPEEDTEST
   printf("\nTime: %f ms\t\tSpeed %.0f chars/sec", 1000*secs, RUNLEN/secs);
   printf("\nMessage length = %llu chars\n", (u_int64_t)RUNLEN);

#else
   printf("CODE OUTPUT =");

   for(i = 0; i < strlen(output); i++){
      //if (i % 5 == 0) printf(" ");
      printf("%c",output[i]);
   }
   printf("\nTime: %f ms\t\tSpeed %.0f chars/sec",1000*secs, strlen(message) /secs);
   printf("\nMessage length = %lu chars\n", strlen(message));
   printf("Strcmp says %s", strcmp(coded_output, output)==0 ? "MATCH\n" : "NOMATCH!\n");
#endif

#endif
   return 0;
}
```

-o–Ö–o-