
The forgotten code

Gunther Zielosko

1. Introduction

Most likely anybody working with the BASIC-Tiger™ knows the different types of presenting numbers or characters. There's the binary code, the hexadecimal code, the ASCII code, the BCD code and many more to which we have more or less gotten used to. In the early years of data processing many more codes were popular, e.g. the octal code, the Hamming code, the Gray code etc. In this application note we are going to have a closer look at one of these "forgotten" codes, the Gray code. We will see that it's undeservedly living in the shadows, as it has some interesting qualities for an electronics engineer. Besides grey theory also some impulses for developments with the BASIC-Tiger™ are here.

2. The world of coded numbers

What do you need coding of numbers for? Reason for rise of all these number codes is the characteristic of computers only to be able to distinguish two different states:

on	off	
power	no power	
logical high	logical low	
high level	low level	
logical 1	logical 0	
L	0	etc.

All data exceeding that, like e.g. the digits 0 to 9, all numbers, the characters of the alphabet, but also pictures, digitized sounds and many others can only be interpreted as a combination of these two basic states. The result is e.g. the binary code, which represents this logically quite clear as sum from powers of 2 (in Tiger-Basic™ binary values are designated by a following "b"):

Number 0	$0*2^3 + 0*2^2 + 0*2^1 + 0*2^0$	binary 0000b
Number 1	$0*2^3 + 0*2^2 + 0*2^1 + 1*2^0$	binary 0001b
Number 2	$0*2^3 + 0*2^2 + 1*2^1 + 0*2^0$	binary 0010b
Number 3	$0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$	binary 0011b
Number 4	$0*2^3 + 1*2^2 + 0*2^1 + 0*2^0$	binary 0100b
Number 5	$0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$	binary 0101b
Number 6	$0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$	binary 0110b
Number 7	$0*2^3 + 1*2^2 + 1*2^1 + 1*2^0$	binary 0111b
Number 8	$1*2^3 + 0*2^2 + 0*2^1 + 0*2^0$	binary 1000b
Number 9	$1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$	binary 1001b
etc.		

Depending on the value numbers need more or less powers of 2 (Bits). The numbers 0 to 15 need 4 bits (Nibble), the numbers up to 256 need 8 bits (Byte) and the numbers up to 65535 need 16 bit (Word). Even larger numbers or decimal values with prefix resp. decimal places need even more bits. So far, so good, for most Tiger users this isn't interesting.

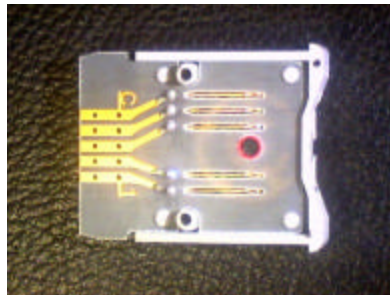
It may be news for some that it makes sense to develop and use some additional formation rules besides the one for binary numbers. The reason for development of the Gray codes is a for some applications unfavorable characteristic of the binary code.

Let's imagine a binary coded rotary switch, like it is often used for presetting limits on measuring instruments (Pictures 1 to 3). It has a numeric wheel at the front with e.g. the digits 0 to 9, on the inside 5 sliding contacts (one common tap, and 4 for the single bits) as well as a segmental wheel, with which in a clever way the corresponding binary code is created. As long as the numerical wheel is fixed exactly on the lock-in positions, there are no problems. But what happens in between two positions? If you e.g. move the wheel slowly from position 7 to 8, at first the 3 low-order bits should be set (7 = 0111b) and then all bits immediately change to (8 = 1000b). But they don't! Constructional tolerances, but also the slightest trembling of the user can lead to the bits "go berserk" in the transition phase. For example the MSB already switches to 1 and the 3 low-order bits still are at 1. This would be (15 = 1111b), a completely false value. If you want to read the switch with a logic (or the BASIC-Tiger[™]), you have to prevent such faulty combinations. But what do you do when you want to build a synchro with the same principle, which has no lock-in positions and should give correct values in between?

If there would be a code, in which from one position to the next only one bit is modified, everything would be unequivocal, even in the critical in-between positions no wrong value is created, only a "jump" to the neighbor value is possible. Such a code does exist (even several), in this application note we choose the Gray code as base for various experiments.



Pic 1 binary coded rotary switch



Pic 2 back view



Pic 3 sliding contact and coded printed card

3. The Gray code

The structure of the Gray code is also systematic, but it can't be that easy deduced from a mathematical relation as the binary code. At first a presentation analog to the previous chapter:

Number 0	0000	↙	only bit 3 is changed
Number 1	0001	↓	only bit 0 is changed
Number 2	0011	↓	only bit 1 is changed
Number 3	0010	↓	only bit 0 is changed
Number 4	0110	↓	only bit 2 is changed
Number 5	0111	↓	only bit 0 is changed
Number 6	0101	↓	only bit 1 is changed
Number 7	0100	↓	only bit 0 is changed
Number 8	1100	↓	only bit 3 is changed
Number 9	1101	↓	only bit 0 is changed
Number 10	1111	↓	only bit 1 is changed
Number 11	1110	↓	only bit 0 is changed
Number 12	1010	↓	only bit 2 is changed
Number 13	1011	↓	only bit 0 is changed
Number 14	1001	↓	only bit 1 is changed
Number 15	1000	↘	only bit 0 is changed

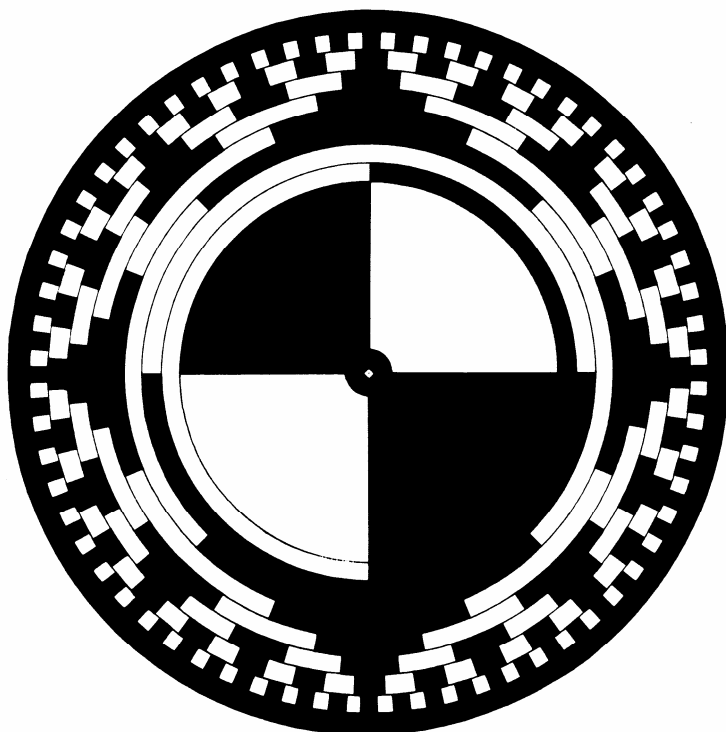
In a graphical presentation the characteristic of the Gray code to only change 1 bit per step becomes even more obvious. The “1” bits are marked red, the “0” bits white. The graphic starts on the left with the value 0. In this example only 6 bits are displayed, the Gray code of course can be realized with more bits, too.

Who wants can check on the code ruler of the Gray code in picture 4, if the result is correct. The 25th column from left (Don't forget column 0!) should show the Gray code 00010100 – and it does!

4. Some hardware

If you think about the red areas in the Gray code presentation in picture 4 as copper areas and the white as epoxy areas of a circuit board, you already have a simple electronic ruler. The copper area is connected to +5V, luckily all parts are connected (that wouldn't be the case with binary code!). Each bit gets a sliding contact, which is connected to a port of the BASIC-Tiger[™]. With an 8 bit code ruler you could acquire 256 positions. The whole can also be thought of with a disk as rotary encoder (Picture 6), here one rotation is divided into 256 single steps. Besides the simple sliding contact technology also small photoelectric barriers can be used, in this case the ruler or segment wheel consists of an exposed foil. Where this is blank, no light gets through. Ideal for such an application are miniature LED's (mostly infrared) and appropriate miniature photo transistors (Picture 7), which can be assembled in 1/10" distance. The actual photoelectric barriers should be sensitivity adjustable and have a Schmitt-Trigger behavior. Very suited for such a mini photoelectric barrier is e.g. an assembly with CD4093 (Picture 8). Build into a chassis it could then become an electronic "weathercock" or a synchro for antenna positioning.

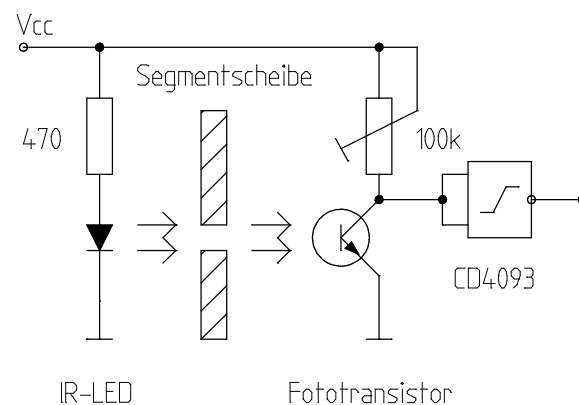
The details presented here may be sufficient for simple applications, in any case it gives impulses for own experiments. There are professional angle transmitters with much higher precision, but these are quite expensive. If you like the principle of absolute positions determination without previous reset or without initialization, you should investigate the internet, there you find rotary encoders with 8192(!) steps per rotation (<http://www.pepperl-fuchs.com>)



Pic 6 A segment wheel for an angular sensor / rotary encoder



Pic 7 A homemade phototransistor line in 1/10" raster for 8 bit



Pic 8 Circuit with CD4093, here only one channel is displayed

5. Software

Coming up to here it is evident that a program for the BASIC-Tiger™ has to follow, with which an 8-bit value coming from a port can be converted from the Gray code into a binary number. This is shown on the Tiger display in binary and also decimal format.

```
-----
' Name: GRAY01.TIG
' Converts the 8-bit Gray code present at port 8 into the appropriate
' binary number. The Gray code, the appropriate binary number as well
' as the decimal value are shown on the LC display
-----
TASK MAIN                                ' begin task MAIN
  BYTE d, e, f                            ' help variables for calculations
  LONG n, a, b                             ' n is counter variable
                                           ' a is Gray input variable
                                           ' b is binary number to calculate

  INSTALL_DEVICE #1, "LCD1.TDD"           ' install LCD driver (BASIC-Tiger)

  DIR_PORT 8,255                          ' set port 8 to input
  USING "UD<1><1> 0.0.0.0.0"              ' format for binary output

Anfang:                                  ' Label for beginning

  b = 0                                    ' preset of help variable b
  IN 8, a                                  ' read Gray code from port 8
  WAIT_DURATION 200                       ' wait a bit

  PRINT #1, "<1>Gray a = ";                ' output of read in Gray number a

                                           ' Gray code in binary presentation
  FOR n = 7 TO 0 STEP -1                  ' loop over all bits
    PRINT USING #1, BIT (a,n);            ' bit by bit output of a
  NEXT                                    ' next bit
  PRINT #1, "b"                            ' output of binary designation

                                           ' Calculation of binary value
  FOR n = 7 TO 0 STEP -1                  ' loop over all bits
    d = BIT (a, n)                        ' query the n-th bit from a
    e = BIT (b, n + 1)                   ' query the (n+1)-th bit from b
    f = d BITXOR e                        ' combine both with XOR
    IF f = 1 THEN                          ' if result is 1 (bits not equal)
      SET_BIT b, n                         ' set n-th bit of b to 1
    ELSE                                    ' if not, keep bit at 0
      SET_BIT b, n
    ENDIF
  NEXT

                                           ' binary value in bit presentation
  PRINT #1, "Bin. b = ";                  ' output of binary number b
  FOR n = 7 TO 0 STEP -1                  ' loop over all bits
    PRINT USING #1, BIT (b,n);            ' bit by bit output of b
  NEXT                                    ' next bit

  PRINT #1, "b"                            ' output of binary designation

                                           ' output decimal number
  PRINT #1, "b = ";b                      ' finally the decimal value b

  GOTO Anfang                             ' new pass from Anfang

END                                        ' end of task MAIN
```