Computer Codes (H)

Codes are a **systematic** and **standardized** method of representing information. You use codes every day. Written and spoken language is a code. There are 3 important categories of computer codes:

- **Numeric,**
- **Character, and**
- **Error detection and correction.**

We have already studied an example of a numeric code; the BCD code. There are many other numeric code systems, but we will leave those for communication course's to look at.

Character Codes

A string of bits need not represent a number. In fact, most of the information processed by computers today is **non-numerical**. The most common non-numeric data is '**text**', strings of characters from a character set. The most common character code is **ASCII** (pronounced ASS key), **(American Standard Code for Information Interchange).**

ASCII code represents each character by a **7 bit string**, a total of 128 different characters as shown in the textbook. **In addition, there is a copy of the ASCII code on my homepage which will be included on tests if needed.**

Table 2, ASCII Character Code

	$C_6C_5C_4$							
$C_3C_2C_1C_0$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	;	J	Z	j	z
1011	VT	ESC	+	:	K	[	k	{
1100	FF	FS	`	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	S1	US	/	?	O	_	o	DEL

Let's check out the code on the following page for an example!

ASCII Example 1

Code the following phrase: **"Ditto _Head!" in ASCII:**

	binary (col) (row)	Hex form
D	100 0100	44
i	110 1001	69
t	111 0100	74
t	111 0100	74
o	110 1111	6F
space	010 0000	20
H	100 1000	48
e	110 0101	65
a	110 0001	61
d	110 0100	64
!	010 0001	21

You would write the answer as follows:

44 69 74 74 6F 20
48 65 61 64 21

ASCII Command or Action Codes

In addition to codes for letters, there are also **ASCII** codes for:

- **ACTIONS,**
- **CONDITIONS, and**
- **STATES**

Examples of such codes are:

STX	Start of text
ETX	End of text
EOT	End of transmission
ACK	Acknowledge
NAK	Negative acknowledge
CAN	Cancel
CR	Carriage return
LF	Line feed

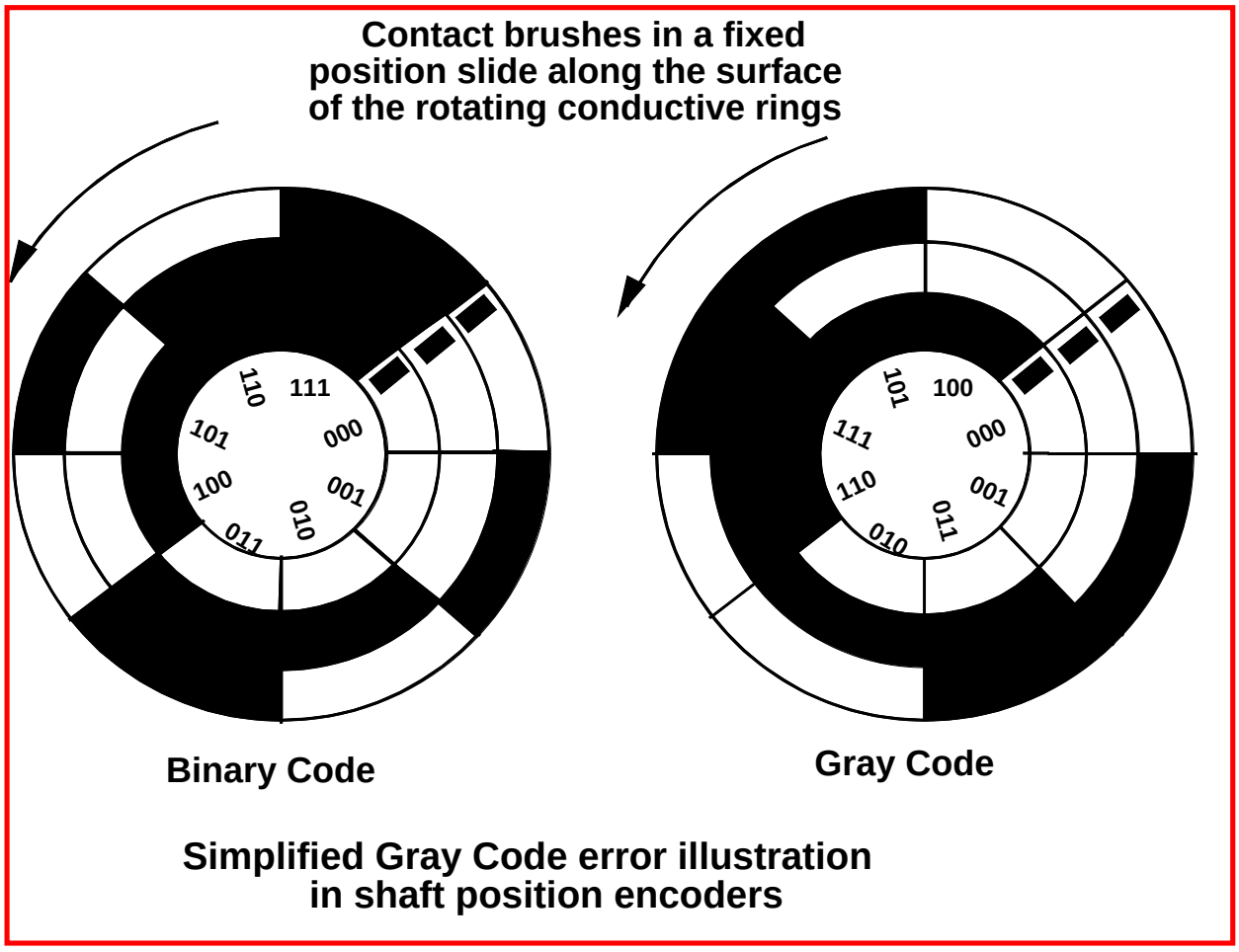
ASCII - 2

As stated before, **ASCII** is **7 bits long** which allow for **128 characters**. There is an extended **ASCII** called **ASCII** 2 which is **8 bits long**. This provides for **more command codes**. **Since both codes require 7 or 8 bits to represent a number, it's not good for numbers.**

GRAY CODE

Gray code is an example of a **cyclic code** which requires that **2 consecutive numbers can only differ in 1 bit position**. Let's look at why we use it.

Figure 1: Gray Code vs Binary Code



A copy of this graphic is included on the course homepage.

Encoder Example:

Consider three concentric conductive rings segmented into eight sectors as shown on the **previous page**. The more sectors it is divided into, the higher the position accuracy. Each sector is fixed at a high-level or a low level (magnetically or by some other method). As the rings rotate, they make contact with a brush arrangement that is "fixed" in space. The contents of the sector are read by the brushes and the data is sent to the output lines. As the shaft rotates **360°**, a **3 bit output indicates shaft position**.

Binary coded sectors

Let's look at what happens when the sectors are coded in Binary Order. When the brushes are in the shaded sectors they will output a "1" and the clear areas will output a "0". If one brush is slightly ahead of the others during the segment to segment translation, a false output will occur.

Binary Coded Example: Brushes are at "111", entering "000". If the MSB brush is slightly ahead, the position would read "011", instead of "111" or "000". Since it is impossible to maintain perfect brush alignment, LARGE errors will occur.

Gray Coded Sectors

Let's change the way we encode the sectors now. The method will be called Gray code order. The gray code assures that only one bit will change between adjacent sectors. This means that even though brushes may not be in alignment, there will NEVER be a transitional error.

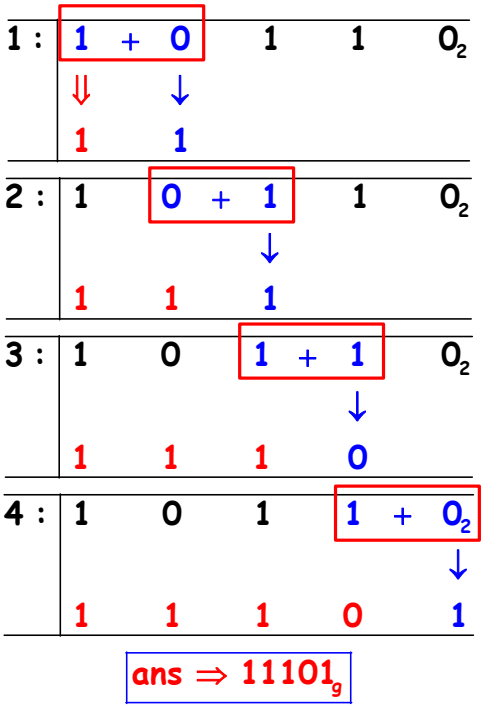
GRAY Coded Example:

Brushes again are at '111' moving into '101'. There are only 2 possible outputs, no matter if the brushes are misaligned or not! (111,101).

Conversion between Systems

Binary-to-Gray Conversion Example 1

1. The **MSB** in **Gray Code** is the **same** as the **MSB in Binary**.
2. Going from left to right, add each adjacent pair of binary bits to get the next Gray code bit. Discard all carries.



Binary-to-Gray Conversion Example 2

Problem: Convert 11000110_2 into gray code.

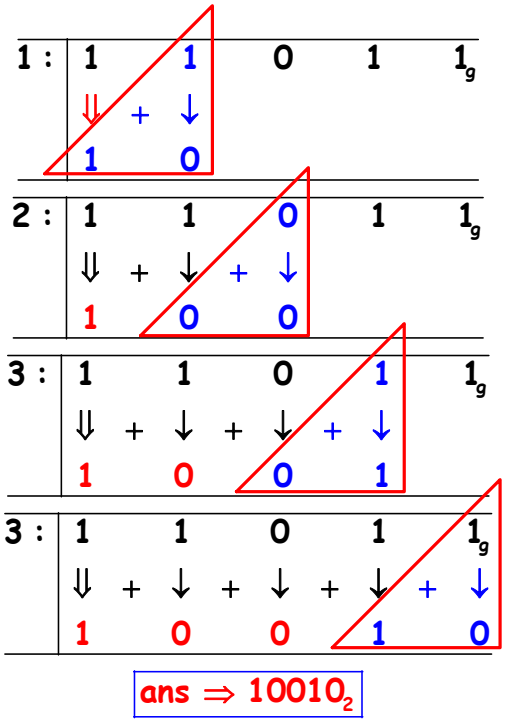
1	1	0	0	0	1	1	0_2
1	0	1	0	0	1	0	1_g

Gray-to-Binary *Conversion Example 1*

I. The **MSB's** are the **same**.

II. Add each binary code bit generated to the gray bit in the next adjacent position.

Discard all carries.



Gray-to-Binary Conversion Example 2

Problem: Convert 10101111_g into binary code.

$$\begin{array}{cccccccc} 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1_g \\ \hline 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0_2 \end{array}$$

Gray code is normally used with **asynchronous systems**. If you don't want to use gray code but you still want accuracy, there is one way to improve the accuracy of the Binary coded wheel. With the addition of a **strobe signal** which allows the reading of the wheel only in the **middle of a sector**, it would take a pretty big misalignment of the brushes to cause the error to be noticed.

ERROR DETECTION &/OR CORRECTION CODES

Parity codes

A **parity bit (P)** is added to an information string. If **EVEN parity** is desired, the **total # of 1's** INCLUDING the parity bit **must be EVEN**. So, if the total # of 1's before the parity is added is already even, then **P=0, else P=1**. We can follow the same thought process for ODD parity. The parity bit may be added at the beginning of the word or at the end of the word. **The only requirement is that all the equipment using the code must understand how the code is set up.** The book places its parity bits at the end of the word to the right of the LSB. I tend to follow the industry standard and place the parity bit at the beginning of the word to the left of the MSB. If the parity/word combination is going to be shifted as discussed earlier, the parity bit is left out of the shift. The shift itself may cause an overflow condition or a discard of a "1" in the word which will show up as a parity error.

Let's take a look at a data string which is the **ASCII code for T. We desire an ODD parity.**

(p) 0	1	0	1	0	1	0	0_{asc}

We see that the **number of 1's** was already **odd** so we make the **parity bit equal to 0**.

With the parity system, **single** error detection is easy. But you have to know ahead of time whether it is **ODD** or **EVEN** parity and where the parity bit is located. Communications protocols require that in order for the receiver to be synced up with the transmitter, both systems must know the protocol ahead of time. When an ODD parity protocol data stream is received, the 1's are counted and if it turns out to be an EVEN number, then there is an error. The problem here is that **you don't know which bit has the error.** Also, **what happens if there are 2 errors? In this case, you wouldn't even see the error and it would be sent on through.**

One way of being able to locate an error in a group of words is to use vertical and horizontal parity redundancy checking. In this case, each word has a parity bit. Then the group of words has a parity bit for each column of bits. In the example below, **each row** has **EVEN parity** and **each column** has **odd parity**.

Multiple Parity Example

This example has **5 words of data** and **1 word of parity bits. Each word**, including the parity word has **a parity bit as the left most bit.** You should be able to see that **the error** is in the **second row, the third bit from the right.**

	even						
	0	**1**	**0**	**1**	**0**	**1**	**1**
	1	**1**	**1**	**0**	**0**	**0**	**0**
	1	**1**	**0**	**0**	**0**	**0**	**0**
	1	**0**	**1**	**1**	**1**	**1**	**1**
	0	**0**	**0**	**0**	**1**	**0**	**1**
odd	**0**	**0**	**1**	**1**	**0**	**1**	**0**