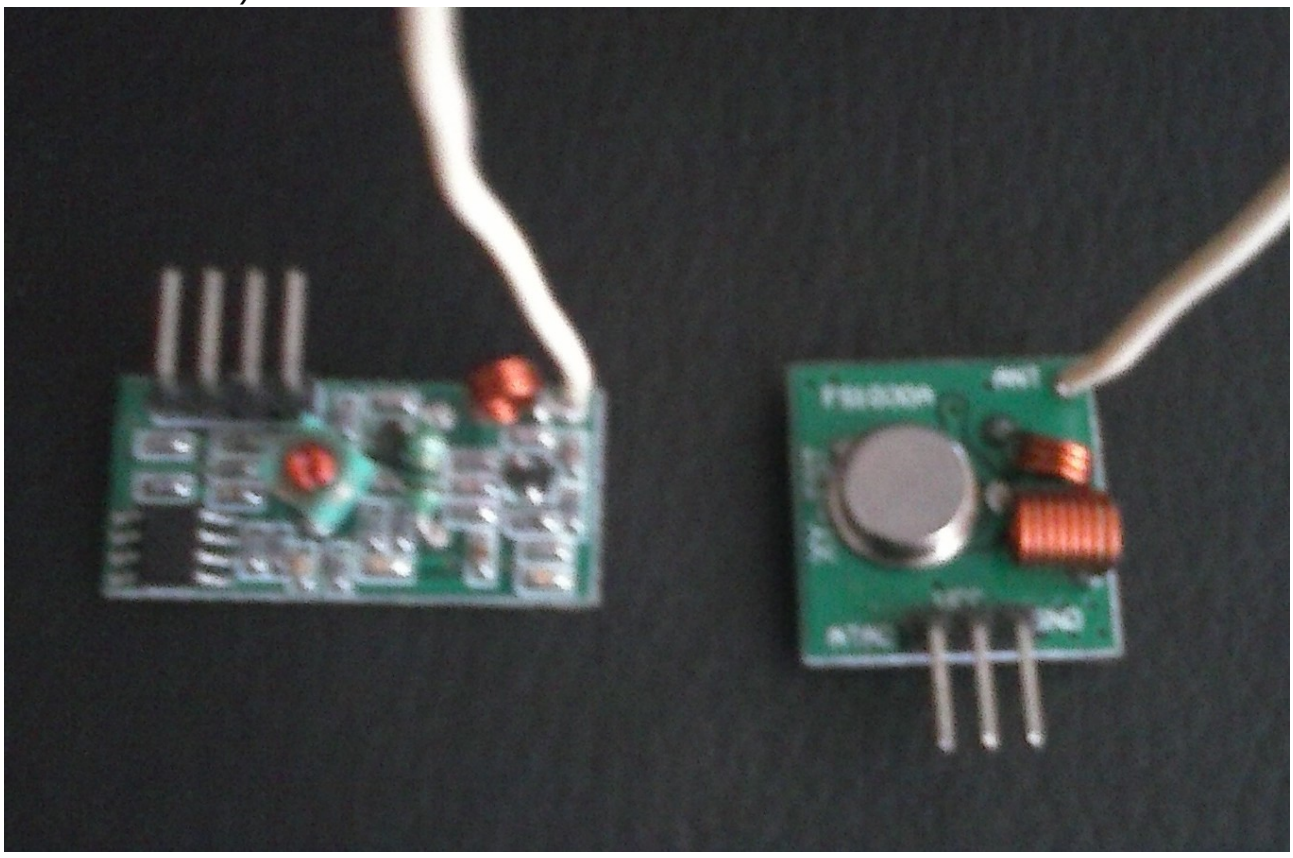
Hamming codes: Theory and Practice (width Arduino)

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

I needed a wireless communication for a personal project (an other weather station jjjj) and I didn't want to use a WIFI module because I wanted to learn about Error Correction Codes, ECC, so I bought an inexpensive wireless pair module (tx: XY-FST FS1000A; rx: XY-MK-5V)



I do not recommend this module unless you want to learn the basics of wireless (what its a ground plane and why you need it for a ¼ wave length antenna, etc...) and some about ECC jjj. This is an ASK/OOK module so it's no very reliable, this last point is why we need an ECC.

Hamming(7,4)

The Hamming idea, in 1950, was to create a code that once received it detects the one bit position error and corrects it with and XOR (more or less).
Lets see how it works. We'll create a 7bit code with 4bits of information (this is why it's call a Hamming(7,4)). We need to identify the 7 possible positions so:

$H=\begin{bmatrix}1111000\\1100110\\1010101\end{bmatrix}$ as you can see these are the binary numbers from 7 to 1 (read the matrix by the column). Our received message will be like:
R = [b_7 b_6 b_5 b_4 b_3 b_2 b_1]

Now if we define something call Syndrome we can detect bit errors. The Syndrome is something like this:

$S=H\cdot R^T=\begin{bmatrix}S_2\\S_1\\S_0\end{bmatrix}=\begin{bmatrix}b_7+b_6+b_5+b_4\\b_7+b_6+b_3+b_2\\b_7+b_5+b_3+b_1\end{bmatrix}$ as you can see the bits b_4, b_2, b_1 appear only once so these will be the parity bits and the rest will be the message bits. So now, our received message will be like this:
R = [m_3 m_2 m_1 p_2 m_o p_1 p_0]
and S will be:

$S=H\cdot R^T=\begin{bmatrix}S_2\\S_1\\S_0\end{bmatrix}=\begin{bmatrix}m_3+m_2+m_1+p_2\\m_3+m_2+m_0+p_1\\m_3+m_1+m_0+p_0\end{bmatrix}$

If $S=\begin{bmatrix}0\\0\\0\end{bmatrix}$ = 0 there is no error and if $S=\begin{bmatrix}1\\1\\0\end{bmatrix}$ there is an error in the 6th bit, easy!!!

OK, now we need to ***C***ode the ***M***essage with this technique:

C = M x G = $[m_3m_2m_1m_0]x\begin{bmatrix}a_7a_6\dots a_1\\b_7b_6\dots b_1\\c_7c_6\dots c_1\\d_7d_6\dots d_1\end{bmatrix}=[c_7c_6\dots c_1]$

$c_7 = a_7m_3 + b_7m_2 + c_7m_1 + d_7m_0 = m_3$ so the first column of G is $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ and

if you continue you'll get $G=\begin{bmatrix}1001011\\0101010\\0011001\\0000111\end{bmatrix}$

Take a look to G and you'll notice that columns 7, 6, 5 and 3 transmit the message as is, and the other columns calculate the parity (we'll use that later...).

Hamming(8, 4)

Well, in a computer world where all is related with powers of 2 sending a 7 bit message is something weird so we're going to send instead a 8 bit message but the extra bit will not be an useless bit. With the Hamming(7, 4) we can detect and correct one bit errors, but with Hamming(8, 4) we can detect and correct one bit

error and detect a two bit error (sorry I'm not going to explain this...).
The equations are the same but now we have an other parity bit so:

C = M x G = $[m_3 m_2 m_1 m_0] x \begin{bmatrix} a_8 a_7 \dots a_1 \\ b_8 b_7 \dots b_1 \\ c_8 c_7 \dots c_1 \\ d_8 d_7 \dots d_1 \end{bmatrix} = [c_8 c_7 \dots c_1]$ so $G = \begin{bmatrix} 01001011 \\ 10101010 \\ 10011001 \\ 10000111 \end{bmatrix}$ but this time I'm going to group all the columns that send the message as is so $G = \begin{bmatrix} 0111 \\ I_4 \; 1011 \\ 1101 \\ 1110 \end{bmatrix}$ this is a way to normalize the matrix.

OK, now we need to decode the ***R***eceived message and it would have some ***E***rrors... R = C + E and this time the syndrome matrix is:
$S = HR^T = HC^T + HE^T = 0 + HE^T$ (the coded message doesn't have errors) so:

$0 = m_2 + m_1 + m_0 + p_3$ $\quad C_8 + 0 + C_6 + C_5 + 0 + C_3 + 0 + 0 = 0$
$0 = m_3 + m_2 + m_1 + p_2 = HC^T =$ ...
$0 = m_3 + m_2 + m_2 + p_1$
$0 = m_3 + m_1 + m_0 + p_0$ $\quad 0 + C_7 + 0 + C_5 + 0 + C_3 + 0 + C_1 = 0$

and finally $H = \begin{bmatrix} 10110100 \\ 01111000 \\ 01100110 \\ 01010101 \end{bmatrix}$ and as you can see this H is related closely with our previous one.

So that's it:

- To send a message we need to code it: C = M x G
- To decode the received message we use: R = C + E

<u>Lets see the Hamming codes in action</u>[1]

The message will be M = 0111

1) No error

R = C + E = C + 0 = [01111000] and S = $HR^T = \begin{bmatrix} 10110100 \\ 01111000 \\ 01100110 \\ 01010101 \end{bmatrix} x \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ so

yes... there's no error jjj

1 As an exercise lets do a 2 bit error and 3 bit error in R

2)One bit error

R = C + E = [01111000] + [00000100] = [01111100] There's one bit error at the 3rd bit. S = HR^T = $\begin{bmatrix}1\\0\\1\\1\end{bmatrix}$ ummm it detects an error in the 11 bit... but our code only has 8 bits. The trick here is that we need to use a module 8 arithmetic (in fact, if you use software like Octave or Matlab to calculate the matrices you'll get an integer values, not binary ones, but the trick it's the same, you need to use a module 2 arithmetic instead because we're using a binary code) so the bit we're talking about it's the 3rd bit. So lets XOR it and test all the parity bits (well... it's not necessary because it's the same message as the previous one jjj)

Using the Hamming(8,4) with a tiny microcontroller (like Arduino)

Maybe you have noticed instantly that all this checks, arithmetic, etc are going to consume to much time and will be hard to code. Well there's a really tiny solution too: lets precalculate this values!!!

All the possible coded messages are 16 and all the possible received messages are 256 (almost a ¼ KB!!) Event this amount of space is ridiculous, we'll put this data in the flash memory not in the RAM (because Ardunio/Genuino UNO has 2Kb of SRAM and 32Kb of flash memory...)

The table of possible coded messages is:

0x00, 0x1e, 0x2d, 0x33, 0x4b, 0x55, 0x66, 0x78, 0x87, 0x99, 0xaa, 0xb4, 0xcc, 0xd2, 0xe1, 0xff

And the possible received ones is:

```
    0x00, 0x00, 0x00, 0x10, 0x00, 0x10, 0x10, 0x08, 0x00, 0x10,
0x10, 0x04, 0x10, 0x02, 0x01, 0x10,
    0x00, 0x10, 0x10, 0x03, 0x10, 0x05, 0x01, 0x10, 0x10, 0x09,
0x01, 0x10, 0x01, 0x10, 0x01, 0x01,
    0x00, 0x10, 0x10, 0x03, 0x10, 0x02, 0x06, 0x10, 0x10, 0x02,
0x0a, 0x10, 0x02, 0x02, 0x10, 0x02,
    0x10, 0x03, 0x03, 0x03, 0x0b, 0x10, 0x10, 0x03, 0x07, 0x10,
0x10, 0x03, 0x10, 0x02, 0x01, 0x10,
    0x00, 0x10, 0x10, 0x04, 0x10, 0x05, 0x06, 0x10, 0x10, 0x04,
0x04, 0x04, 0x0c, 0x10, 0x10, 0x04,
    0x10, 0x05, 0x0d, 0x10, 0x05, 0x05, 0x10, 0x05, 0x07, 0x10,
0x10, 0x04, 0x10, 0x05, 0x01, 0x10,
    0x10, 0x0e, 0x06, 0x10, 0x06, 0x10, 0x06, 0x06, 0x07, 0x10,
0x10, 0x04, 0x10, 0x02, 0x06, 0x10,
    0x07, 0x10, 0x10, 0x03, 0x10, 0x05, 0x06, 0x10, 0x07, 0x07,
0x07, 0x10, 0x07, 0x10, 0x10, 0x0f,
```

```
    0x00, 0x10, 0x10, 0x08, 0x10, 0x08, 0x08, 0x08, 0x10, 0x09,
0x0a, 0x10, 0x0c, 0x10, 0x10, 0x08,
    0x10, 0x09, 0x0d, 0x10, 0x0b, 0x10, 0x10, 0x08, 0x09, 0x09,
0x10, 0x09, 0x10, 0x09, 0x01, 0x10,
    0x10, 0x0e, 0x0a, 0x10, 0x0b, 0x10, 0x10, 0x08, 0x0a, 0x10,
0x0a, 0x0a, 0x10, 0x02, 0x0a, 0x10,
    0x0b, 0x10, 0x10, 0x03, 0x0b, 0x0b, 0x0b, 0x10, 0x10, 0x09,
0x0a, 0x10, 0x0b, 0x10, 0x10, 0x0f,
    0x10, 0x0e, 0x0d, 0x10, 0x0c, 0x10, 0x10, 0x08, 0x0c, 0x10,
0x10, 0x04, 0x0c, 0x0c, 0x0c, 0x10,
    0x0d, 0x10, 0x0d, 0x0d, 0x10, 0x05, 0x0d, 0x10, 0x10, 0x09,
0x0d, 0x10, 0x0c, 0x10, 0x10, 0x0f,
    0x0e, 0x0e, 0x10, 0x0e, 0x10, 0x0e, 0x06, 0x10, 0x10, 0x0e,
0x0a, 0x10, 0x0c, 0x10, 0x10, 0x0f,
    0x10, 0x0e, 0x0d, 0x10, 0x0b, 0x10, 0x10, 0x0f, 0x07, 0x10,
0x10, 0x0f, 0x10, 0x0f, 0x0f, 0x0f
```

Notice that I've marked the impossible messages to correct with 0x10 because the biggest correct message is 0x0F

Of course I've wrapped all this stuff in a class so it's really easy to use this Hamming(8,4) in our Arduino. As a practical example I'm using the class with a RF module but you can use it with all kind of byte communication.

To see all the class code and some examples see the Hamming library.

Happy arduining ;-)

Code: https://bitbucket.org/bullakio/hamming-8-4-arduino