Binary Mathematics (Bits and Bytes)

A Tutorial

Using Binary for Configuring CV's in Hornby's RailMaster

A further Tutorial

Issue 4

© April 2016

The purpose of this document is to explain the mathematical relationship between Binary and Decimal numbering schemes and how this relationship is applied to the application of CV's (Configuration Variables) for DCC (Digital Command Control) control of digital model railway decoders.

Particular focus is made here for the DCC CV29 variable although the principle for any CV within RailMaster is exactly the same. Similarly, as decoders are NMRA compliant, the same goes for most decoders on the market.

~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~

Bits & Bytes

Preamble - The Basics

- What is a Bit
- What is a Byte

Before answering those two questions it is wise to perhaps explain what a computer can understand and what it cannot. To some the following may come as a surprise, while to others it is common knowledge. The computer (shall we now just use the term 'processor' for arguments sake as it is the processor that does the legwork), compared to our brain can understand virtually nothing. That's right, nothing… well, almost nothing. That is there are two things that the processor, any computer, large or small, can understand. These are the numbers 0 and 1. While zero is not technically a number (the Romans never had a zero, for example) it is still a value. It is that value of 0 and the number 1 that the humble and wonderful technological device you are reading this on now understands (unless you have printed this). That's it.

So, a simple 0 and 1. So, how does that technology get to do super-fast and massive calculations in seconds or less? This document will explain without the need for quantum theory computing that understands multiple 'states' and will tell you in simple terms how all this works and how you can apply Binary to your daily needs as a model rail operator. Your average Joe processor only understands two 'states'. An 'on' and an 'off'.

Bit

In computer terms, the 'off' state is represented by the value of '0' and the 'on' state by the value of '1'. These values of '0' and '1' are termed 'Bits'.

Bits on their own are great for a processor because, as you will see shortly, if the computer as a whole just looked at a massive string of 0's and 1's it would be in its element if it could display emotions et al. But, to us humans, a string of those numbers would be a nightmare to understand. So we had the programmers make up a new term which grouped a small amount of these Bits together and termed it a Byte.

Byte

A Byte is the computing term given to a group of eight Bits. How does that work then? Easy. You now know that a processor can only handle two states or numbers so what we need to do now is offer the processor more of them so it can calculate bigger sums. If I give you a coin in one hand and no coin in the other, you are effectively given a count of one. If I now give a second coin you have a count of two. Or do you? In Decimal, our normal counting system, yes you do have two coins. In Binary, you have a count of 1 plus 1. As the state can only be 0 or 1 we have to come up with a slightly different way of looking at things. This is where the Base value arrives.

Base

In the normal, Decimal, counting system we use a Base of ten or ‘10’. As we count from zero to 9 we have no other numbers to use so we have to invent something that expands our counting system as, in life, we invariably have more than 9 of everything. So what we do is add 1 to 9 and use the starting 0 again but with a ‘carried’ 1. By moving the 1 to the left we can always find the single unit count on the right hand side. The left hand number becomes a multiple of the single right hand number. So we have 1 multiple unit of ten and no single units thus, one, zero, represented by ‘10’. The next single unit added makes one, one thus ‘11’ and so on until one, nine or ‘19’. We named that nineteen just for recognition purposes. When a further single unit is added we get two, zero or ‘20’. Easy as that.

Going back to the ***Byte*** scenario we use a Base of two. Therefore, when 1 is added to 1 we cannot go past ‘1’ so the next logical step is to make it one full unit thus making ‘10’. Adding one to that each time gives… ‘11’, ‘100’, ‘101’, ‘110’, ‘111’, ‘1000’, ‘1001’ and so on. So you can see how each addition of a coin is represented in this scenario by adding one to each hand if the hand cannot hold more than one item. You would just need lots of hands as numbers would grow really fast and it would become impossible to read.

By this score we could easily use a Base of 7 or 3 or 6 or whatever you like up to 10. Base 6 would be ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘10’, ‘11’, ‘12’ etc.

Thus we have now explained the Binary system. The term ‘Binary’, an adjective, comes from “*relating to, composed of, or involving two things*”. For example: A Binary Star is a ‘star system’ consisting of **two** stars orbiting around their common centre.

Why a Byte is 8 Bits long

The term Byte was first introduced in 1956 as a deliberate spelling changed to keep the confusion away from the singular Bit. Given the 2-Bit state powered by 2 gives eight (2^2) and thus the 8 Bits making up one Byte which gives a convenient count of 0 through to 255 in Binary when converting to Decimal. So in computing terms a ‘**Byte**’ contains 8 ‘**Bits**’ in a ‘**Binary**’ two state code. These Bits are defined as Bit 7, Bit 6, Bit 5, Bit 4 through Bit 0.

The following table shows the Byte and its values in Binary and Decimal.

Table 1

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Binary	‘0’ or ‘1’	‘0’ or ‘1’	‘0’ or ‘1’	‘0’ or ‘1’	‘0’ or ‘1’	‘0’ or ‘1’	‘0’ or ‘1’	‘0’ or ‘1’
Decimal	‘0’ or ‘128’	‘0’ or ‘64’	‘0’ or ‘32’	‘0’ or ‘16’	‘0’ or ‘8’	‘0’ or ‘4’	‘0’ or ‘2’	‘0’ or ‘1’

If all Binary values equalled ‘1’ then the Decimal equivalent would equal 255 by adding all Decimal values where the Binary equalled ‘1’. However, where a Binary value, anywhere along that line, equalled ‘0’ then that equivalent Decimal value is removed from the total of 255. So, ‘11101011’ would equal 235.

Thus the value of a Decimal number for a Byte never exceeds 255 but can be any value up to that. Bit ‘0’ is the equivalent of the right sided single unit in Decimal and thus counts upward to the left and thus Bit ‘1’ goes one added position to the left and so on up to Bit ‘7’.

Converting Binary 8 Bit (1 Byte) to Decimal

Table 1 is an 'aide memoir', a 'look up' table that can be used to convert Decimal to Binary and Binary to Decimal.

Converting the Decimal number 88, for example, is the summation of Decimal numbers 64 + 16 + 8 from the table. If we place a '1' in the Binary row of *Table 1* below the three Decimal numbers stated above and a '0' in the remaining boxes with no Decimal value, then the Binary row would read **0 1 0 1 1 0 0 0**.

Thus: Decimal 88 = Binary 01011000

Converting Decimal to Binary 8 Bit (1 Byte)

If we have a Binary 8 Bit number and we want to know what that number is in Decimal we populate the Binary row of *Table 1* with the '0' & '1' Bits in the correct order. Then, in the Binary row boxes where a '1' is present, we add the corresponding Decimal numbers in the Decimal row together.

Example:

Let's use Binary 8 Bit number 01001101 and consider it placed in the table. Thus, the Decimal numbers that have a matching Binary '1' are as follows: 64 + 8 + 4 + 1 = 77

Thus: Binary 01001101 = Decimal 77

Further, if someone said convert the following and just spoke to you and there were no written figures to hand you should be able to work it out… 01001011 which would be mental arithmetic at its best. Therefore, 64 + 8 + 2 + 1 would equal 75

NB. You may have noted that each time the Decimal value to the left increases, it doubles. By adding one to each sum from zero in Binary you should see and be able to work out why that is.

Try a few out for yourselves…

Table 2

Decimal	Binary Bit Decimal Value where adding the total equals that of the Decimal number to the left	Binary Equivalent
0	0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	00000000
1	0 + 0 + 0 + 0 + 0 + 0 + 0 + **1**	00000001
2	0 + 0 + 0 + 0 + 0 + 0 + **2** + 0	00000010
3	0 + 0 + 0 + 0 + 0 + 0 + **2** + **1**	00000011
4	0 + 0 + 0 + 0 + 0 + **4** + 0 + 0	00000100
5	0 + 0 + 0 + 0 + 0 + **4** + 0 + **1**	00000101
6	0 + 0 + 0 + 0 + 0 + **4** + **2** + 0	00000110
7	0 + 0 + 0 + 0 + 0 + **4** + **2** + **1**	00000111
8	0 + 0 + 0 + 0 + **8** + 0 + 0 + 0	00001000
9	0 + 0 + 0 + 0 + **8** + 0 + 0 + **1**	00001001
10	0 + 0 + 0 + 0 + **8** + 0 + **2** + 0	00001010
to	and so on to	to
255	**128** + **64** + **32** + **16** + **8** + **4** + **2** + **1**	11111111

Table 2 extends this concept into a worked example, showing the Binary Byte / Bit sequence for the Decimal numbers 0 to 10 inclusive plus 255.

To fully understand the contents of *Table 2*, it may be helpful to look at the detail in conjunction with *Table 1*.

Looking at the Binary column of *Table 2*, we can see that there is a sequential progressive pattern in the way that the Binary '0's and '1's' increment as the corresponding Decimal number increases by Decimal 1. Thus, every possible combination of 'Binary Byte' Bit patterns gives 256 possible combinations. That is to say, Decimal 0 to 255.

<u>Working backwards.</u>

<u>Deriving a Binary 8 Bit Byte sequence from a Decimal number.</u>

As well as using *Table 1* to derive the Binary Byte Bits from the Decimal number, we can achieve the same result in two further ways. As a worked example let us consider the Decimal number **57**. This time we start working from the left hand side of the 8 Bit Byte (Bit 7). This time each 'remaining' number is the next to be divided:

Can we divide	57	by	128	answer is	NO	thus Bit 7 = 0
Can we divide	57	by	64	answer is	NO	thus Bit 6 = 0
Can we divide	57	by	32	answer is	YES	thus Bit 5 = 1 (25 remaining)
Can we divide	25	by	16	answer is	YES	thus Bit 4 = 1 (9 remaining)
Can we divide	9	by	8	answer is	YES	thus Bit 3 = 1 (1 remaining)
Can we divide	1	by	4	answer is	NO	thus Bit 2 = 0
Can we divide	1	by	2	answer is	NO	thus Bit 1 = 0
Can we divide	1	by	1	answer is	YES	thus Bit 0 = 1 (**0 remaining***)

***** If you don't end up with zero remaining in the last row, then you have gone wrong somewhere with a previous division.

Thus, Decimal 57 = Binary 00111001

The same basic sequential ‘maths’ process can be used to work back any Decimal number between 0 & 255 into a Binary 8 Bit Byte.

As an alternative to the method above, there is a ‘divide by 2 methods’ that can do the same task.

In the ‘divide by 2 methods’ the Decimal number is divided by 2. Where the Decimal number is ‘even’ the remainder left over after division will be ‘0’. Where the Decimal number is ‘odd’ the remainder left over after division will be ‘1’. These ‘0’ & ‘1’ remainders define the Binary Bits.

*Example with the same Decimal **57** used in first method above:*

57	divided by 2	=	28	with	1	remaining
28	divided by 2	=	14	with	0	remaining
14	divided by 2	=	7	with	0	remaining
7	divided by 2	=	3	with	1	remaining
3	divided by 2	=	1	with	1	remaining
1	divided by 2	=	**0**	with	1	remaining

Once the division integer* is zero (**0**), the division calculations stop.

* Integer, a positive (or negative) whole number or zero.

Now taking the ‘0’ & ‘1’ remaining column and writing the numbers from the ‘**Bottom Up**’ we get:

1 1 1 0 0 1

But this Binary string is not a full 8 Bit Byte because it has only 6 Bits in it. To make it an 8 Bit Byte we need to add additional ‘**padding**’ to make a total of 8 Bits in the form of leading zeros.

So the final Binary number is:

0 0 **1 1 1 0 0 1**

This we can see is exactly the same result as that obtained in the first method above, that is to say:

Decimal 57 = Binary 00111001

With practice, this alternative method can be quicker to do, but we have to remember to write the Binary number in the results from the ‘**Bottom Up**’ and add as many additional leading zero **padding** Bits as is necessary to create an 8 Bit Byte.

Another example using this method:

Decimal 6

6	divided by 2	=	3	with	0	remaining
3	divided by 2	=	1	with	1	remaining
1	divided by 2	=	0	with	1	remaining

Writing the ‘0’ & ‘1’ column from the ‘**Bottom Up**’ we get:

1 1 0

We can see that only three Bits have been calculated, so five leading zeros of **padding** need to be added, thus we get:

Decimal 6 = Binary 0 0 0 0 0 **1 1 0**

See the ‘Decimal 6’ entry in *Table 2* for confirmation of this result.

Another example using this method:

Decimal 222

222	divided by 2	=	111	with	0	remaining
111	divided by 2	=	55	with	1	remaining
55	divided by 2	=	27	with	1	remaining
27	divided by 2	=	13	with	1	remaining
13	divided by 2	=	6	with	1	remaining
6	divided by 2	=	3	with	0	remaining
3	divided by 2	=	1	with	1	remaining
1	divided by 2	=	0	with	1	remaining

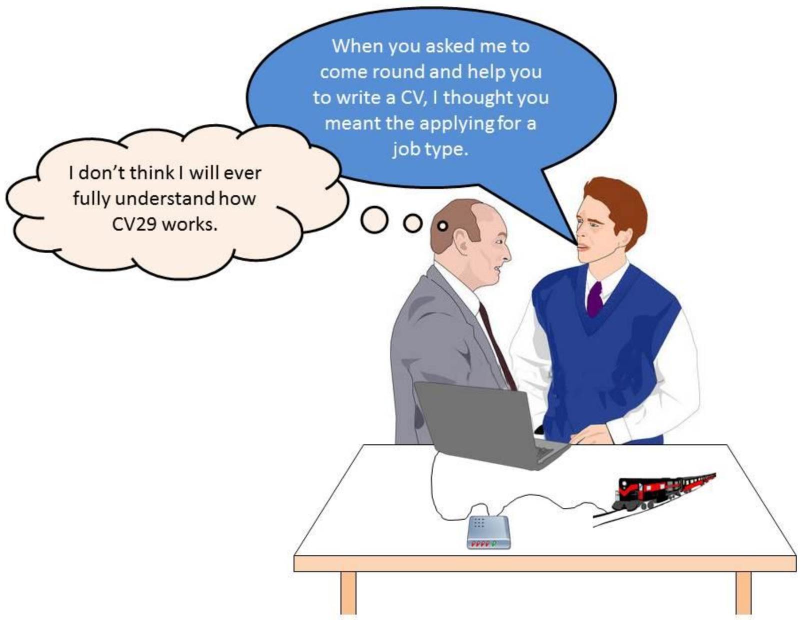
Writing the ‘0’ & ‘1’ column from the ‘**Bottom Up**’ we get:

1 1 0 1 1 1 1 0

We can see that all 8 Bits have been calculated, so **no** leading zeros of **padding** need to be added, thus we get:

Decimal 222 = Binary **1 1 0 1 1 1 1 0**

How to Implement Binary to any Decoder CV on a Model Railway


When you asked me to come round and help you to write a CV, I thought you meant the applying for a job type.
I don't think I will ever fully understand how CV29 works.

Example: applying the previously discussed principles to DCC CV29.

In CV29 there are up to eight features that can be either enabled or disabled by setting the appropriate 'Bit' in the 'Byte'. These 'Bits' are like 'on / off' switches to enable or disable a feature.

The eight Bits are:

Table 3

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Binary	'0' or '1'	'0' or '1'	'0' or '1'	'0' or '1'	'0' or '1'	'0' or '1'	'0' or '1'	'0' or '1'
Decimal	'0' or '128'	'0' or '64'	'0' or '32'	'0' or '16'	'0' or '8'	'0' or '4'	'0' or '2'	'0' or '1'
Feature	Reserved for Custom Manufacturer use	Reserved for Custom manufacturer use	Long Loco Address	Complex Speed Curve	Railcom	DC Operation	28/128 Speed Steps	Reverse Direction

To enable a specific feature in the table above, the corresponding 'Bit' in the eight Bit Byte is set to 1.

To disable the specific feature, the corresponding 'Bit' in the eight Bit Byte is set to 0.

Some examples of common Decimal values of CV29 are:

Decimal 2, 6 and 38

In Binary these would be:

Table 4

Binary	Decimal	Enabled Features
00000010	2	28 / 128 Speed Steps
00000110	4+2=6	DC Operation + 28 / 128 Speed Steps
00100110	32+4+2=38	Long Loco Addresses + DC Operation + 28 / 128 Speed Steps

When looking at *Table 4*, relate the entries contained within back to the contents of *Table 3*.

So for example from *Table 4* to only enable 28 /128 Speed Steps + DC operation you would write Decimal value 6 to CV29.

<u>Changing the Decimal number of the CV changes the corresponding Binary Bit sequence.</u>
<u>Changing the Binary Bit sequence defines what features are enabled and disabled.</u>

Example: if the loco goes in the wrong direction because the motor has been wired up 'back to front' to the DCC decoder then rather than take the loco apart and simply unsolder the wires on the motor, reverse them and re-solder them, we can fix that issue in the DCC software by incrementing the value of CV29 by Decimal 1.

Refer to *Table 3* (Bit 0 column) - by incrementing the Decimal value of CV29 by Decimal 1 the Bit 0 of the 8 Bit Byte is set to Binary 1, this enables the 'Reverse Direction' CV29 feature.

So using the three examples in *Table 4* above:

The CV29 Decimal 2 value would become Decimal 3
The CV29 Decimal 6 value would become Decimal 7
The CV29 Decimal 38 value would become Decimal 39

In principle, the concept of Binary Bits in a CV being set by defining a corresponding Decimal number can be applied to any CV.

Example 1

If a manual says set Bit 4 of CV(n) to 'ON' or to '1' then looking at *Table 1* we can see that Bit 4 is represented by Decimal 16. So, by adding the Decimal value 16 to the current CV(n) Decimal value will enable (or switch 'ON' or set to '1') that particular 'Bit' in the 'Byte'. Or does it? see note[1)]

Example 2

If a manual says set Bits 3 **AND** 5 'ON' or set to '1' then looking at *Table 1* we can see that Bit 3 is represented by Decimal 8 and Bit 5 is represented by Decimal 32. So adding 8 + 32 (40) to the value of that particular CV(n) will enable (switch 'ON' or set to '1') the Bit 3 AND Bit 5 of that Byte, or does it? See ** below.

Similarly, if a feature needs to be disabled, you subtract the relevant 'Bit' Decimal value in *Table 1* from the CV(n) Decimal number. Or does it? See ** below.

** In order to minimise the risk of making errors it is essential to always 'read' the existing CV Decimal value first and derive its Binary Byte sequence before making any changes. Else you risk disabling features that have been previously enabled or enabling features that you had no intention of enabling.

An example of not deriving the Binary sequence of the CV Decimal value first before modifying those CV Decimal values.

Let's say that a decoder manual states that CV(n) should have Bit 4 set to 'ON' or '1' to enable a specific feature. Now looking at *Table 1* we can see that Bit 4 = Decimal 16.

Now let's say that the Decimal value of CV(n) is read back and it is found to be Decimal 82.

By adding the Decimal value 16 to the existing Decimal value 82 of CV(n) will result in Decimal value 98 being written back to CV(n).

To do that would be a fundamental error. Let's see why.

By converting the Decimal values back into their Binary Byte sequences we can see that:

The existing CV(n) Decimal value that was 82 = Binary 010**1**0010.

The highlighted (bold) Binary '1' Bit is '**Bit 4'**. We can now see that 'Bit' 4 had already been set to '1', thus the specific feature that the manual had referred to had already been 'enabled'.

If we now write Decimal value 98 to CV(n), because Decimal value 16 had just been added to the existing Decimal 82 value and written back to CV(n) without checking, we would be setting the new Binary Byte sequence as:

Decimal 96 = Binary 01**10**0010.

Binary Bits 5 & 4 above highlighted (bold).

Now, not only have we now set '**Bit 4**' as '0' (not '1' as required), we have also changed the value of '**Bit 5**' from '0' to '1'.

So, in conclusion, not only has the feature that was required in CV(n) been disabled (Bit 4), a different feature altogether (Bit 5) has been enabled instead.

By always converting a CV(n) Decimal number into its Binary Byte sequence, prior to writing a new value, the risk of making feature configuration errors is greatly reduced.

Hopefully this tutorial has given a deeper insight to how Binary works using Bits and Bytes on a Base Level of 2 and how these can be implemented into any decoder CV.