Homearticles **glossaire** SitemapESP32FacebookGithubcontact me

The ternary logic

published: 24 September 2020 / updated 7 October 2020

Lire cette page en français

Preamble

Let's get directly into the subject. Binary logic is two values: **TRUE** and **FALSE**. That's all.

It is the base from which are built all the binary logic through the operators **OR**, **AND**, **NOT** and the resulting functions in electronics, **NOR**, **NAND**, **XOR**....

So a ternary logic, why do it, if the binary logic responds so perfectly needs to?

Manage unknown cases

We will illustrate the ternary logic with two extremely concrete cases.

The first case is that of a sliding door, of this type of door that we find in all shopping centers:

Let's just take a clapper, the one on the right for example. We can consider that this clapper has two states:

- leaf closed: state **FALSE**
- leaf open: status **TRUE**

To confirm each of these states, the manufacturer has fitted each leaf with a contactor limit switch confirming the state of the leaves. These contactors act on the motorization to stop this motorization as soon as one of these binary states is reached.

But the state of a leaf cannot be linked to these contactors alone. It exists the case where each contactor indicates **TRUE**, which is physically impossible for a leaf: it cannot be opened AND closed! ... except in quantum physics..

On the other hand, there is one case that happens all the time. When is it the two contactors indicate **FALSE**: the leaves are neither open, nor closed.

Binary logic cannot describe the state of a leaf which in this case is neither open nor closed.

In automatic mode, sequential logic knows how to manage perfectly without having to take into account a third state, ie a **UNKNOW** state.

The ternary logic of the SQL language

Here are two examples of the action of the logical operator **AND** in SQL language:

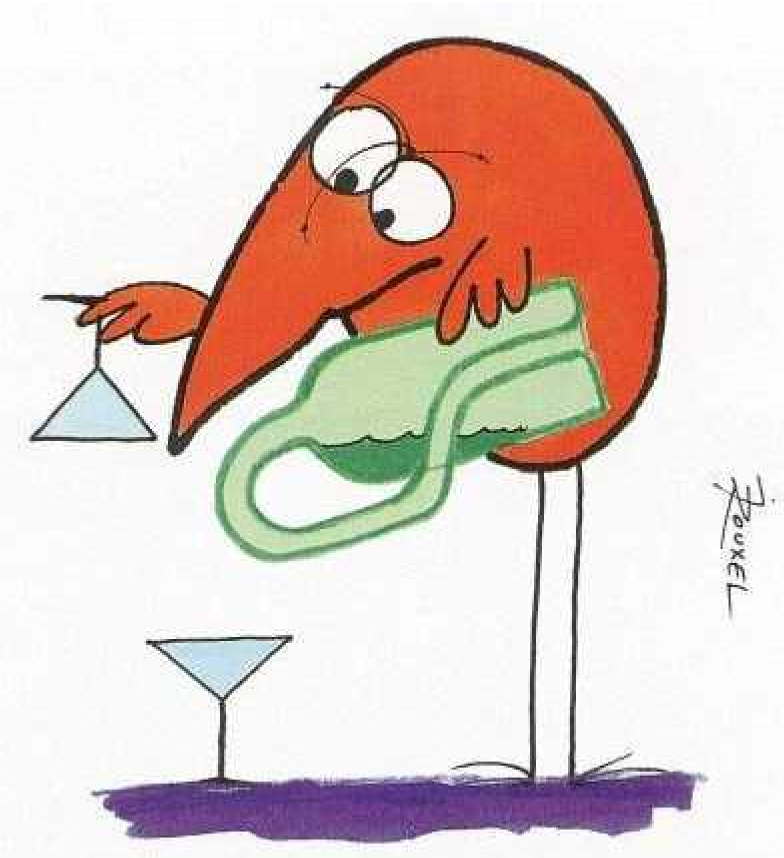
```
SELECT TRUE AND TRUE
SELECT TRUE AND FALSE
```

The SQL language includes a third state: **NULL** , which is similar to the state **UNKNOW** of our door leaf.

Still in SQL language, here are all the ternary logical combinations with the logical operator **OR**:

```
SELECT TRUE OR TRUE;     -- result: 1
SELECT TRUE OR NULL;     -- result: 1
```

```
SELECT TRUE OR FALSE;   -- result: 1
SELECT NULL OR TRUE;    -- result: 1
SELECT NULL OR NULL;    -- result: NULL
SELECT NULL OR FALSE;   -- result: NULL
SELECT FALSE OR TRUE;   -- result: 1
SELECT FALSE OR NULL;   -- result: NULL
SELECT FALSE OR FALSE;  -- result: 0
```



if there is no solution,
there is no problem.

Ternary logical operators

The idea is to reproduce, in FORTH language, the ternary equivalent of the operators **AND**, **OR** and **NOT**:

The ternary OR operator

Here is the table of ternary logic states for the operator **OR**:

A	*B*	*A* **OR** *B*
True	True	True
True	Unknow	True
True	False	True
Unknow	True	True
Unknow	Unknow	Unknow
Unknow	False	Unknow
False	True	True
False	Unknow	Unknow
False	False	False

In FORTH language, we have no logical **UNKNOW** state. We go therefore define numerical values and associate them with a logical equivalent ternary:

```
\ tfl : ternary flag
\ tfl = 0   is FALSE  flag
\ tfl = 1   is TRUE   flag
\ tfl = 2   is UNKNOW flag
```

According to this text:

- value = 0, the ternary flag is FALSE
- value = 1, the ternary flag is TRUE
- value = 2, the ternary flag is UNKNOW

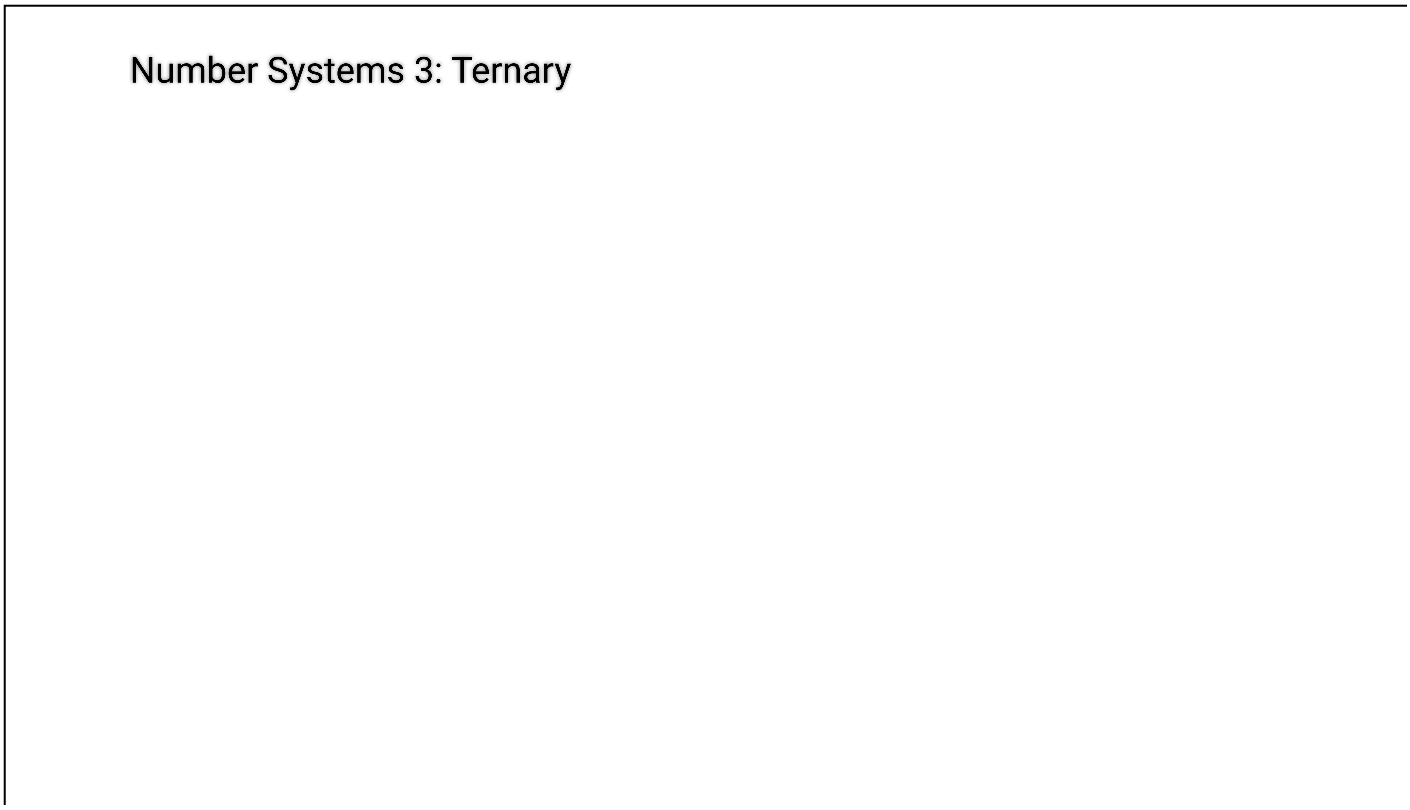
Why this choice of 0, 1 and 2?

If we take any integer, 16 or 32 bits, simply hiding on bit b0, there are two possible states: 0 for an even value, 1 for an odd value.

If we mask **TRUE** (in binary 1111111111111111), we get 1 which remains **TRUE** in ternary logic. It will be the same for all other values odd integers.

We could have proceeded differently: 0 for FALSE, positive for TRUE and negative for UNKNOW.

Here is a video which explains ternary numeration (base 3). It is this video that has oriented our choice on the values 0 (FALSE), 1 (TRUE) and 2 (UNKNOW):



For the ternary mechanics to work in FORTH language, we go first create the word `n>tfl`:

```
\ convert decimal value to ternary flag
: n>tfl ( n --- tfl)
    dup  0= if            \ test if n equal 0
        drop 0
    else
        dup  1 and 0> if  \ test if n is ODD
            drop 1
        else
            drop 2        \ else n is EVEN
        then
    then
  ;
```

This word `n> tfl` is simple and efficient:

- if n is zero, stack zero
- if n is odd, stack 1
- if n is even, stack 2

If we want to test two values, we will add their ternary values like this:

```
\ sum two ternary flags
\ 0 2 on stack:
  n>tfl
  swap n>tfl
  10 * +
```

Thus, two ternary flags will merge into 9 possible values: 00, 01, 02, 10, 11, 12, 20, 21 and 22.

We can now define the ternary operator **TOR** (for Ternary OR):

```
\ ternary OR
: tOR ( n1 n2 --- tfl)
    n>tfl
    swap n>tfl
    10 * +
    dup 11 =   if  drop 1 exit  then
    dup 12 =   if  drop 1 exit  then
    dup 10 =   if  drop 1 exit  then
    dup 21 =   if  drop 1 exit  then
    dup 22 =   if  drop 2 exit  then
    dup 20 =   if  drop 2 exit  then
    dup 01 =   if  drop 1 exit  then
    dup 02 =   if  drop 2 exit  then
        00 =   if         0 exit  then
  ;
```

There are certainly much more elegant methods. We will talk about this later. The interest of this definition is to show a *readable* way of create this operator `tOR`.

The ternary AND operator

Here is the table of ternary logic states for the **AND** operator:

A	***B***	***A* AND *B***
True	True	True
True	Unknow	Unknow
True	False	False
Unknow	True	Unknow
Unknow	Unknow	Unknow
Unknow	False	False
False	True	False
False	Unknow	False
False	False	False

```
\ ternary AND
: tAND ( n1 n2 --- tfl)
    n>tfl
    swap n>tfl
    10 * +
    dup 11 =   if  drop 1 exit  then
    dup 12 =   if  drop 2 exit  then
    dup 10 =   if  drop 0 exit  then
    dup 21 =   if  drop 2 exit  then
```

```
    dup 22 =   if  drop 2 exit  then
    dup 20 =   if  drop 0 exit  then
    dup 01 =   if  drop 0 exit  then
    dup 02 =   if  drop 0 exit  then
        00 =   if         0 exit  then
  ;
```

The ternary NOT operator

Here is the table of ternary logic states for the **NOT** operator:

A	**NOT *A***
True	False
Unknow	Unknow
False	True

```
\ ternary NOT
: tNOT ( n1 --- tfl)
    n>tfl
    dup 1 =   if  drop 0 exit  then
    dup 2 =   if  drop 2 exit  then
        0 =   if  drop 1 exit  then
  ;
```

Other solutions

Gordon Charlton solution

```
: tOR ( tfl1 tfl2 -- tfl3)
    or dup 2 > if 2 - then ;
```

Branchless `tOR`:

```
: tOR
    or dup 1+ 4 / 2* - ;
```

Bruce R. McFarling solution

Though in my poor xForth, condemned to run on a 65C02, only the tOR would be faster... branching would be faster than multiplying, which is painfully slow.

Now, the only time {A,B,tAND} is not {A,B,AND} is when {A,B,+}>2 => {A,B,tAND}=2, so perhaps:

```
: tAND ( tfl1 tfl2 -- tfl3 )
  2DUP AND >R + 2 > 2 AND
  R> OR ;
```

Any computer scientist knows binary logic. Ternary logic is an extension, very little known, but which has very real applications.

WARNING

In our articles, reference is made to different versions of the language FORTH (AmForth, FlashForth, GForth ...).

All of these versions are licensed GNU. Ref: [Licence publique générale GNU]

So you can download, install, use all these versions on your computer and / or on an ARDUINO card with no transfer of fees and no special conditions.