

# Imagine to Invent a Language v13 minimized

As basis we define a piece of real hardware and will implement it just using the PC – replace all of the hardware necessary with Keyboard – PC – Screen. The minimized version just uses the PW LED of the ones shown.

The real hardware consists of a few LEDs and a few Push-Button Switches. 1 LED PW on its own, 4 LEDs represent OUTPUTs, 4 LEDs represent INPUT levels, and 4 LEDs more for Additional Functions; each “pin” can be either HIGH or LOW, 1 or 0 - and we end up with the header line for function and a line for the logic level. The 4 O, I and A values are set in a way to represent binary values, so you can count up and down later using the same display.



|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| PW | T3 | T2 | T1 | O3 | O2 | O1 | O0 | I3 | I2 | I1 | I0 | A3 | A2 | A1 | A0 |
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  |

For now, we just want to set and reset these bits to 0 or 1. As programming exercise

But describing, how this should be programmed, is often more difficult than starting, thinking, designing and going through the process. So, let’s get started and “imagine to define” a new computer language. At least the basics.

We want to have it done in a very easy way to be easily understood, so no difficult constructs in our language yet.

All normal ASCII characters can be used and are allowed to define our commands.

But we still have to separate these commands from each other, and we do it as in a normal writing – so a SPACE will do this. And from now on we call these commands WORDS as in normal writing.

## SPACE - 1 is as such the first word as just defined

Programming is one activity we have to do, but we have to make sure that we can add comments as well, explaining what the program does. So, a wall between the two areas is a good idea to divide program and explanations. The keyboard gives us for example two options \ and /. As dividing needs the /, the answer is clear: \ - the back slash will be used as indicator to tell the computer, that the rest of the line is for comments and ignored by the computer.

## SPACE \ - word 2 after SPACE

There are many aspects of programming we have to cover. For example, we have to set up a way to send ASCII characters to the display. Putting this text within 2 “ quotes “ would be a good way to define the text. And we have to define a command to send characters to the screen. We will just use in general the . (full stop) for this. And combined with the two quotes we know now already how to send the usual HELLO WORLD to the screen: .” HELLO WORLD “. Try it: enter . “HELLO WORLD “, but nothing happens yet. It is actually very clear why: the computer does not know that you have finished; entering ENTER/<cr> is used for this. Push the ENTER key after the .” HELLO WORLD “<cr> and do not forget the separating SPACES. Now the computer answers HELLO WORLD. We can do already what the other languages show off with. And the . on its own we deal with later.

## SPACE \ . ” xx “ - now 5 words

It would nice, if we could define a way to combine what we know and call it something new. Like a paragraph we can relate to. We have to tell the computer though, how that construct/“paragraph” starts and where it ends.

A : will start the new definition and a ; ends it.

## SPACE \ . ” xx “ : ; - now 7 words

Let us use the HELLO WORLD as an example. Let us define our first new word, give it the name DISPLAYIT: : DISPLAYIT .” HELLO WORLD “ ; when we finish with the <cr> at the end, the computer will answer OK, which means understood. If we enter now DISPLAYIT <cr>, the HELLO WORLD appears on the screen. Enter the same DISPLAYIT, and another HELLO WORLD is stuck onto the first one.

It would be nice if the new HELLO WORLD could appear underneath the first one. A word CR does the job when we normally type, so we use it here as well as a new command.

## SPACE \ . ” xx “ : ; CR - now 8 words

We modify our : DISPLAYIT .” HELLO WORLD “ ; to : DISPLAYITCR .” HELLO WORLD “ CR ; <cr> Job done. The computer answers OK, understood. And every time we enter DISPLAYITCR <cr>, the same block is displayed, one underneath the other now as we wanted.

As well it would be nice to be able to clean the display page, and the word PAGE does the job: all of the text is deleted and we end up with a clean screen page – nothing on it. Our cursor at the top left position.

## SPACE \ . ” xx “ : ; CR PAGE - now 9 words

We have to put the LED status somewhere, and we have to be able to change it, so we call them VARIABLES and store information there. We need 16 of them as we store a bit per variable / LED for now. We use a new word ! to store info into them and use @ to get the contents back from a VARIABLE.

## SPACE \ . ” xx “ : ; CR PAGE VARIABLE ! @ - now 12 words, all of the Forth words needed .

This is now the end of PART 1a. More difficult constructs and defining how it works together continues in PART 2.

End of Part 1 – more at <http://preview.tinyurl.com/hfwf7y4> and <http://preview.tinyurl.com/znwssv>

```
\ IMAGINE TO INVENT A LANGUAGE in 2 pages, do a small application v12 ExMark Nov2016
\ INCLUDE C:\VFXTESTAPP\TESTAPPv10.f
```

```
\ We want to display these 2 lines on the screen and set these bits to 0 and 1.
\ PW T3 T2 T1 O3 O2 O1 O0 I3 I2 I1 I0 A3 A2 A1 A0 set XXh/1
\ 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0
```

```
\ All of the Forth Words used and in the sequence as described above here:
\ SPACE \ . " xx " : ; CR PAGE VARIABLE ! @
```

```
\ Define the text lines to display
: TEXT0 ." INVENT A LANGUAGE, do a samll application v4 ExMark Oct2016 " CR ;
: TEXT1 ." PW " CR ;
: TEXT2 ." 0 " CR ;
```

```
\ Define the variable needed for the LED to save the status in
VARIABLE PW
```

```
\ Define the 2 words to switch the 1 bit in the variablke PW
\ to 1 (HIGH) and the word to set the variable to 0 (LOW)
: PWH 1 PW ! ; \ set the variable PW to 1 (HIGH)
: PWL 0 PW ! ; \ set the variable PW to 0 (LOW)
```

```
\ Define a word to display the contents of the variable as in TEXT2 and TEXT2
```

```
\ Define a word to display the variable:
: DISBIT PW @ . SPACE ;
```

```
\
: BITPW DISBIT ;
```

```
\ And now to send the text line TEXT1 and the 16 Variable values to the screen
: Display CR TEXT1 DISBIT ;
```

```
\ use a new word MS, which adds a delay before continuing.
\ The time as number before, e.g. 1000 MS as 1000 milliseconds.
: ONOFF PWH 1000 MS PWL ;
```

```
: Hello ." HELLO FORTH WORLD " ; \ just added, the usual first program ...
```

The Forth Words used and a short explanation:

```
\ SPACE Print a SPACE to the screen
\ \ The divider - anything after the \ is ignored by the computer
\ . Print a value to the screen
\ ." xx " Print the ASCII string string xx to the screen, ended by "
\ : Start a new word definition; e.g. : Hello ." Hello " ;
\ ; And finish the new word definition shown above with ;
\ CR Send a return to the display - one line down, cursor to the left
\ PAGE Delete all of the information on screen to have a clean screen
\ VARIABLE Define a new 16 Bit VARIABLE - only the lowest bit is used here
\ ! Store a value into a VARIABLE: XX PW !
\ @ Read the value back from a VARIABLE and print it: PW @ .
\ MS MS defines a delay and the number before MS sets the length: 100 MS
\ End of Part 1a: success to switch the LED On / Off, display the result
\ Part 2 to follow with more difficult constructs
```

End of Part 1a - more at <http://preview.tinyurl.com/hfwf7y4> and <http://preview.tinyurl.com/znwssvr>

## Imagine to Invent a Language Part 2 v12

In Part 1 we “invented” the basics of our language. Now in Part 2 it is getting more complex, and we have to set up a system to ensure we can deal with all of the issues that might come up.

Rather than inventing something new, we look at examples that work. One is the office space we work in – or how cars are manufactured using a conveyer belt. Such a process will require that we have to set up the sequence of activities before we start to execute them. This will ensure as well, that all of the parts are prepared as a sequence of events. Such a definition results as well in the fact, that no brackets and other constructs are used to make data good to look, but not really ready to process using a conveyer belt.

Imagine: On the left we have the work coming in – like slices of work.

These “slices” coming in have either to be processed immediately or put somewhere for later work. A stack is a good solution for this – like a stack of plates and new parts are added on top of what is there already.

And, as we had prepared the sequence of activity before starting already – they are put onto the stack in the correct sequence as they will be needed later. This stack we call Data Stack, and of its importance **The Stack**. Sometimes we might have to interrupt our normal sequence of activities, before we return to the sequence in progress – so another stack, and we call it **Return Stack**.

We have to deal with the **Variables** as well that we had defined – and they look like Pidgeon Holes.

These Variables are like named memory addresses; but there are other memory areas we will have to deal with later as another lot of Pidgeon holes.

After processing, relevant data has to be sent out – so as we had an INPUT STREAM – there is an OUTPUT STREAM.

And this “execution machine” that we have defined now, we will use all the time from now on.

|  |  |  |     |   |  |  |    |  |      |
|--|--|--|-----|---|--|--|----|--|------|
|  |  |  | TOS |   |  |  |    |  | ADD8 |
|  |  |  | 2   |   |  |  |    |  | ADD7 |
|  |  |  | 3   |   |  |  |    |  | ADD6 |
|  |  |  | 4   |   |  |  |    |  | ADD5 |
|  |  |  | 5   |   |  |  | V4 |  | ADD4 |
|  |  |  | 6   | 1 |  |  | V3 |  | ADD3 |
|  |  |  | 7   | 2 |  |  | V2 |  | ADD2 |
|  |  |  | 8   | 3 |  |  | V1 |  | ADD1 |

**FORTH MACHINE:** ITEM<sub>n</sub> ITEM<sub>n-1</sub> ITEM2 ITEM1 - DSTACK - RSTACK - VARIABLES - MEMORY - OUT<sub>n</sub> OUT<sub>n-1</sub> OUT<sub>n-2</sub> OUT1

We actually have used exactly the same machine already in Part 1 – but without mentioning it.

And as things are getting more complicated, we need a way to see what is happening inside our execution machine. A little Debugger will show some details: some Variables – a part of the Return Stack – a part of the Data Stack – or more if we want later.

So let’s define what we want to display with our little Debugger and then define the words to achieve it:

```
: DebugDisplay ." V1_V2_V3_V4__R0_R1_R2__D0_D1_D2_D3_D4_D5_D6_D7 "
```

Like this we can see 4 Variables, the 3 levels of the return stack from the top, and the top 8 DSTACK values D0 to 7.

How to display the 4 Variables we know already: **V1 @ . V2 @ . V3@ . V4 @ .** and so a first word is

```
: vars PW @ . space T3 @ . space T2 @ . space T1 @ . space ;
```

With a bit of shuffling we can print the Return Stack information as well, but we need a few new words:

**R>** to move the data of the top item of the Return Stack to the Data Stack and then later to print.

**>R** will move the data on the top of the DSTACK to the Return Stack.

**>R>R>R** will transfer the 3 “plates” R2 R1 and R0 onto the Data Stack and print them out as R0, then R1 and R2

```
: RSTACK R> R> R> . space . space . space . ;
```

A bit of shuffling as well to show the status of the DSTACK Plates: **>R>R>R>R>R>R>R>R** to get them to the RSTACK. This transfers the plates over from the DStack to the RStack in the right sequence D7 D6 D5 D4 D3 D2 D1 D0, now on the RStack. This can be printed out now using 8 times the procedure to transfer the Rstack to the DStack and print it out. Move from R to DStack, DUP, to duplicate the value, so one is printed, one is kept to rebuild as it was before:

```
: DSTACKtoRSTACK >R >R >R >R >R >R >R >R ;
: DtRbackDUP1_4. R> DUP . space R> DUP . space R> DUP . space R> DUP . space ;
: DtRbackDUP5_8. R> DUP . space R> DUP . space R> DUP . space R> DUP . space ;
: DSTACK DSTACKtoRSTACK DtRbackDUP1_4. DtRbackDUP5_8. ;
```

The three words combined in our Debugger would send the Forth Machine State information to the screen:

```
: ?? VARS RSTACK DSTACK ;
```

