Imagine to Invent a Language **v13a minimized** **- Juergen Pintaske – ExMark 24 July 2017**

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 for now. More will be used in the 35 Word example.
The real hardware consists of a few LEDs showing 1 LED PW on its own and 3 Push-Button Switches, 4 LEDs represent OUTPUTs, 4 LEDs show 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 invent" 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 - WORD 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. An added SPACE before ; would separate it.

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 current 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 later 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 constructs and defining how it all works together continues in PART 2.

```
\ Type this code directly into a Forth, or use INCLUDE when available / understood
\ IMAGINE TO INVENT A LANGUAGE in 2 pages, do a small application v12  ExMark July 2017
\ 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/l
\ 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 - here just limited for now to the display of PW
: TEXT0  ." INVENT A LANGUAGE, do a samll application v4  ExMark July 2017 " 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 variable PW
\ to switch 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 TEXT1 and TEXT2

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

\ Just an example of how to give an already defined Word a new name
: BITPW   DISBIT ;

\ And now to send the text line TEXT1 to the screen
: Display   CR TEXT1 DISBIT ;

\ Use a new word MS, which adds a delay before continuing.
\ The time as number put before MS, e.g. 1000 MS as 1000 milliseconds.
\ This new word ONOFF will switch LED ON, wait 1000 millisecons, then switch LED OFF
: 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 \ is ignored by the computer to end of the line
\ .        Print a value to the screen
\ ." xx "  Print the ASCII string string xx to the screen, ended by ". Space after ." !
\ :        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 XX into a VARIABLE: XX PW !
\ @        Read the value back from a VARIABLE and print it using the . :   PW @ .
\ MS       MS defines a delay and the number before MS sets the length: e.g. 1234 MS
\ End of Part 1a: success to switch the LED On / Off, display the result

\ NOTE: When using easyFORTH, the ms does not exist and has to be replaced by sleep

\ Part 2 to follow with more

End of Part 1a
```

Imagine to Invent a Language Part 2 v12

In Part 1 we "invented" the basic Words 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 then no brackets and other constructs are used to make data good to look and more understandable visually, but not really ready to process using a conveyer belt.

Imagine the desk you are probably sitting at now: 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 activities 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 as 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 is needed, 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.

There are other memory areas we will have to deal with later on as another lot of Pidgeon holes. Or used as arrays.

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: **ITEMn ITEMn-1 ITEM2 ITEM1 - DSTACK - RSTACK - VARIABLES - MEMORY - OUTn OUTn-1 OUTn-2 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, 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 top "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 shuffeling 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 .   ;
: 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  ;
```

BUT: we are putting our hands basically into a running machine, so we have to be very careful, and as such:
all of the Debugger activities have to be written as one word between : and ; .
Here, no combined words as shown are allowed, no <cr> to make it more easily readable.

Every time from now on, when we type ????? <cr> or use ????? in a program, the internal status will be printed out.
The complete code, including the the ones from Part 1:

```
\ INVENT A LANGUAGE, do a small application v10  ExMark Oct2016
\ INCLUDE C:\VFXTESTAPP\TESTAPPv10.f

\ Display these 2 lines
\ PW_T3_T2_T1  O3_O2_O1_O0  I3_I2_I1_I0  A3_A2_A1_A0 set   XXh
\ 1  0  0  0   0  0  0  0   1  1  1  1   0  0  0  0  reset XXl

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

\ Define some text lines to display
: TEXT0 ." INVENT A LANGUAGE, do a samll application v1  ExMark Oct2016 " CR ;
: TEXT1 ." PW_T3_T2_T1  O3_O2_O1_O0  I3_I2_I1_I0  A3_A2_A1_A0 XXh / XXl " CR ;

\ Define the 16 variables - one per LED
VARIABLE PW      VARIABLE T3      VARIABLE T2      VARIABLE T1
VARIABLE O3      VARIABLE O2      VARIABLE O1      VARIABLE O0
VARIABLE I3      VARIABLE I2      VARIABLE I1      VARIABLE I0
VARIABLE A3      VARIABLE A2      VARIABLE A1      VARIABLE A0

\ Define the 32 words to switch the 16 bits to High or Low
: PWH 1 PW ! ;   : T3H 1 T3 ! ;    : T2H 1 T2 ! ;    : T1H 1 T1 ! ;
: PWL 0 PW ! ;   : T3L 0 T3 ! ;    : T2L 0 T2 ! ;    : T1L 0 T1 ! ;
: O3H 1 O3 ! ;   : O2H 1 O2 ! ;    : O1H 1 O1 ! ;    : O0H 1 O0 ! ;
: O3L 0 O3 ! ;   : O2L 0 O2 ! ;    : O1L 0 O1 ! ;    : O0L 0 O0 ! ;
: I3H 1 I3 ! ;   : I2H 1 I2 ! ;    : I1H 1 I1 ! ;    : I0H 1 I0 ! ;
: I3L 0 I3 ! ;   : I2L 0 I2 ! ;    : I1L 0 I1 ! ;    : I0L 0 I0 ! ;
: A3H 1 A3 ! ;   : A2H 1 A2 ! ;    : A1H 1 A1 ! ;    : A0H 1 A0 ! ;
: A3L 0 A3 ! ;   : A2L 0 A2 ! ;    : A1L 0 A1 ! ;    : A0L 0 A0 ! ;

\ Define a word that displays the 16 variables as shown above

\ This needs the spaces for better readability
: 2SP Space Space ;

: Nibble1  PW @ . SPACE  T3 @ . SPACE  T2 @ . SPACE  T1 @ . ;
: Nibble2  O3 @ . SPACE  O2 @ . SPACE  O1 @ . SPACE  O0 @ . ;
: Nibble3  I3 @ . SPACE  I2 @ . SPACE  I1 @ . SPACE  I0 @ . ;
: Nibble4  A3 @ . SPACE  A2 @ . SPACE  A1 @ . SPACE  A0 @ . ;

\ And all 16 bits plus the Spaces in between
: 4NIBBLES  Nibble1 2SP  NIBBLE2 2SP  NIBBLE3 2SP  NIBBLE4 ;
\ And both lines now
: Display  CR TEXT1 4NIBBLES ;

: Hello ." HELLO FORTH WORLD " ;   \ added to show the usual first program

\ SPACE   \   .   ." xx   "   :   ;   CR   PAGE   VARIABLE   !   @   >R   R>

\ Part2 Define a Debugger, display the Forth Machine V1..V4, RS0..RS2, DS0..7
: v-r-d  ." V1_V2_V3_V4    R0_R1_R2   D0_D1_D2_D3_D4_D5_D6_D7 " CR ;
: vars  PW @ . space T3 @ . space  T2 @ . space  T1 @ . space space space ; \ display variables
: RSTACK  R>  R>  R>  . space  . space . space space space ;   \ display RSTACK
: DSTACKtoRSTACK  >R >R >R >R >R >R >R >R ;  \ move 8 top parts of DSTACK to RSTACK
: DtRbackDUP1_4.  R> DUP . space  R> DUP . space  R> DUP . space  R> DUP .  space  ; \ display 4
: DtRbackDUP5_8.  R> DUP . space  R> DUP . space  R> DUP . space  R> DUP .  space  ; \ display 4
: DSTACK  DSTACKtoRSTACK  DtRbackDUP1_4. DtRbackDUP5_8. ;  \ combine the three
: ??  VARS  RSTACK  DSTACK  ;  \ combine Vars display, with RSTACK display, and DSTACK display
: RSTACKBASE 0 >R ;          \ After start, insert a bottom marker 0 to show positions later
: RSTACK12  1 >R 2 >R ;      \ After start, insert 2 more positions
: DSTACKBASE 0 ;             \ After start, insert a bottom marker 0 onto DSTACK
: DSTACK17 1 2 3 4 5 6 7 ;   \ After start, insert 7 numbers to show 7 additional positions
```

The final Debugger ????? needs all of these parts in one line between : and ; to work correctly

```
: ?????  3 spaces PW @ . T3 @ . T2 @ . T1 @ . 4 spaces R>  R>  R> DUP >R . Space DUP >R . space
DUP >R . 4 SPACES  >R >R >R >R >R >R >R >R R> DUP . R> DUP . R> DUP . R> DUP . R> DUP . R> DUP .
R> DUP . R> DUP . ;
\ ????? can be used to check the status - as well in programs or to check how words behave
```

End Part 2 - to be extended see https://wiki.forth-ev.de/doku.php/en:projects:a-start-with-forth:start0