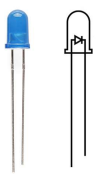
COMPUTER SYSTEM

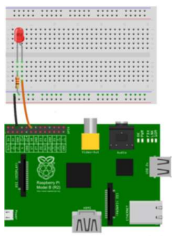
DR. NGUYEN DANG KHOA

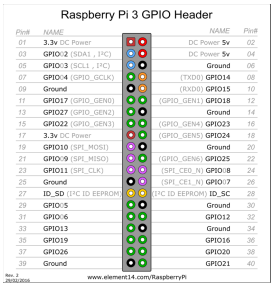
FACULTY OF ELECTRICAL AND ELECTRONIC ENGINEERING

Remind

LED control



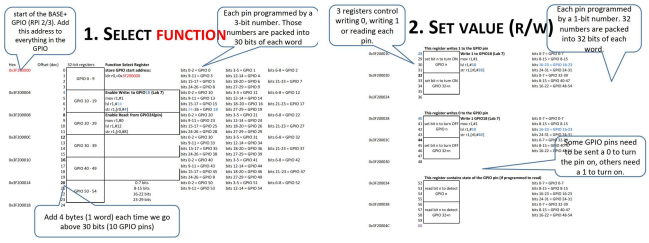




https://www.youtube.com/watch?v=Rd9kvVs1ISQ

Sample Programming: TURN ON/OFF LED

Map memory and register



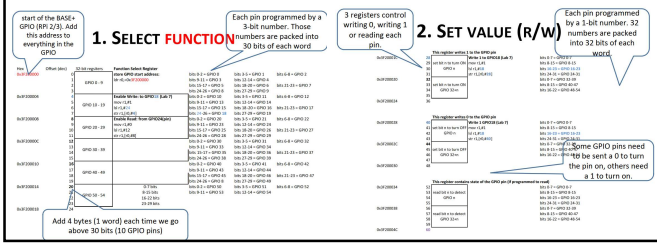
Sample Programming: TURN ON/OFF LED

How to use?

Like other microchip Arduino, PIC, AVR... We need to have 2 steps

Step 1: Configuration for GPIO in input/output mode ➔ *1. Select function* *001➔output; 000➔ input*

Step 2: Set value ON/OFF for GPIO ➔ *2. Set value (R/W)*

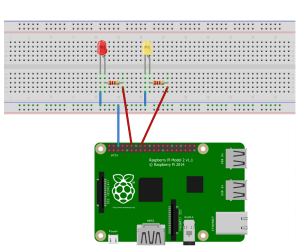


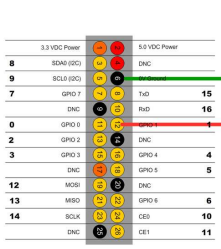
Sample Programming: TURN ON/OFF LED

Example: Blink LED in GPIO30 and GPIO36 (15 mins)

Sample Programming: TURN ON/OFF LED

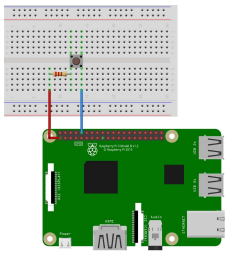
Example: Blink LED in GPIO30 and GPIO36 (15 mins)





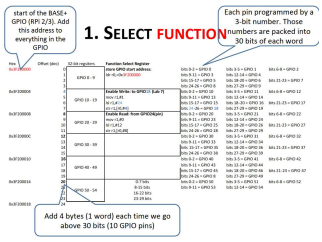
HOW ABOUT INPUT?

- Button is input?
- How to connect button to Raspberry PI? LOW active, HIGH active
- *How to get signal from button?*



CONFIGURE GPIO INPUT

- 001➔ output and 000 ➔ input
- Store value 0 to the address of GPIO



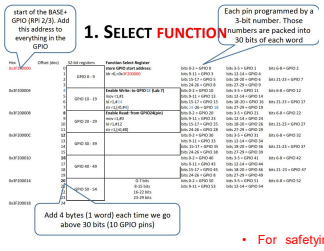
For example: GPIO10

BASE = $3F000000
GPIO_OFFSET = $200000
mov r0,BASE
orr r0,GPIO_OFFSET

mov r1,#0
str r1,[r0,#4] *;config input for GPIO10*

CONFIGURE GPIO INPUT

- 001➔ output and 000 ➔ input
- Store value 0 to the address of GPIO



For example: GPIO10 ➔ input ; GPIO15 ➔ output

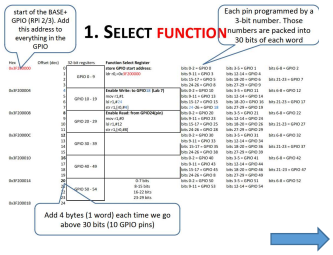
BASE = $3F000000
GPIO_OFFSET = $200000
mov r0,BASE
orr r0,GPIO_OFFSET

mov r1,#1
Lsl r1, #15
str r1,[r0,#4] *;full config*

- For safetying: we want to only change bit of GPIO10 without changing all offsets?

CONFIGURE GPIO INPUT

- For safetying.



For example: GPIO10

BASE = $3F000000
GPIO_OFFSET = $200000
mov r0,BASE
orr r0,GPIO_OFFSET ;

ldr r1,[r0,#4] ;read memory in address with offset 4
bic r1,r1,#7 ;bit clear
str r1,[r0,#4]

➔ *How does bic work?*

CONFIGURE GPIO INPUT

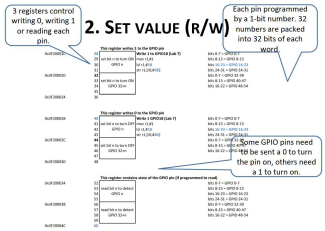
- For safetying.

bic r1,r1,#7

➔ *r1= r1 AND (NOT 7)*

bit	29	28	....	....	....	....	2	1	0
r1	0	0	1	0	1	0	1	0	1
7	0	0	0	0	0	0	1	1	1
NOT 7	1	1	1	1	1	1	0	0	0
R1 AND (NOT 7)									
r1	0	0	1	0	1	0	0	0	0

READING DATA FROM GPIO INPUT



Reading bit of GPIO in offset 52 to 56

➔ *How to read?*

➔ *ldr r9, [r0, #52]*

Compare to 0/1 to check state of button

COMMAND

- **Command: cmp**

➔ *Compare a register with a value (register...), and store result to Application Program Status Register (APSR)*
Example: cmp r1, #20

- **Command: beq**

➔ *Branch if equal*
Example:

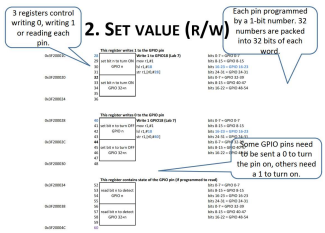
```
cmp r1, r2
beq loop$  ➔ goto loop if r1==r2. (Access APSR to get result of cmp)
```

- **Command: bne**

➔ *Branch if not equal*
Example:

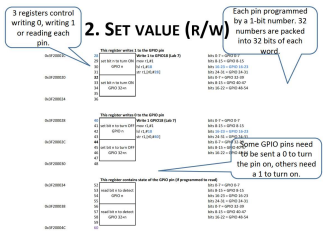
```
cmp r1, r2
beq loop$  ➔ goto loop if r1<>r2. (Access APSR to get result of cmp)
```

READING DATA FROM GPIO INPUT




```
ldr r9, [r0, #52]
cmp r9, # ????
bqe loop$          ;bne
```

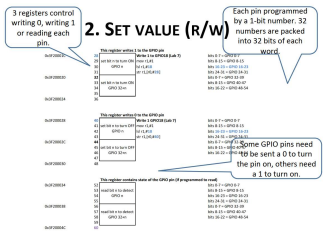
READING DATA FROM GPIO INPUT




```
;GPIO10
ldr r9, [r0, #52]
cmp r9, # 1024
bqe loop$        ;bne

;GPIO40???
```

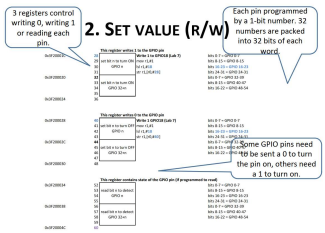
READING DATA FROM GPIO INPUT




```
ldr r9, [r0, #52]
cmp r9, # 1024
bqe loop$        ;bne

;GPIO40
ldr r9, [r0, #56]
cmp r9, # 256
bqe loop$        ;bne
```

READING DATA FROM GPIO INPUT

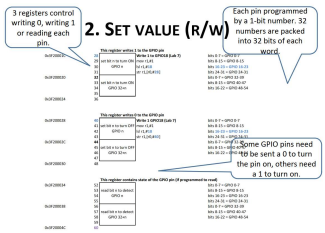


Other method:

```
ldr r9, [r0, #52]
tst r9, # 1024     ;similar to cmp
beq loop$          ;bne
```

What's the difference between the CMP and TST instructions in ARM?

READING DATA FROM GPIO INPUT



Other method:

```
ldr r9, [r0, #52]
tst r9, # 1024      ;similar to cmp
beq loop$           ;bne
```

TST R1, R2 computes the bitwise AND of R1 and R2 and then discards the result while **CMP** R1, R2 subtracts the two. TST is mainly useful on ARM for finding out if a given bit is set in a number.

READING DATA FROM GPIO INPUT

Example: *Let's connect GPIO10 and 11 to button. GPIO12 connects to LED*

- *If GPIO10=1 then turn on LED*
- *If GPIO11=1 then turn off LED*

(15 mins) 2:25PM

HOW TO PRESENT ARRAY IN ASSEMBLY

```
//C, C++
int an_array[100];

for (int i = 0; i < 100; i++)
{
        printf("%d", an_array[i])
}
```

HOW TO PRESENT ARRAY IN ASSEMBLY

```
//ASM
myArray:
.int 1,2,3,4,5,6,7,8        → integer

myName:
.ascii "James Hamlyn-Harris\0"   → string

myNum:
dw $F0002000                → Number
                              dw: 32 bit
                              db: 8 bit
```

HOW TO PRESENT ARRAY IN ASSEMBLY

```
//C, C++                    //ASM
int test[100];              myArray:
                            .int 1,2,3,4,5,6,7,8

test[0]=10;                 mov r5,#0          ;index of array
                            mov r4, myArray
                            Mov r0, #10
                            str r0, [r4, r5]   ; r4[r5]=r0

int a= test[0]+5;           mov r5, #0         ;index of array
                            mov r4, myArray
                            ldr r0, [r4, r5]   ;r0=r4[r5]
                            add r0, r0,#5
```

HOW TO PRESENT ARRAY IN ASSEMBLY

```
//C, C++                    //ASM
int test[100];              myArray:
                            .int 1,2,3,4,5,6,7,8

test[0]=10;                 mov r5,#0          ;index of array
                            mov r4, myArray    ; should  change to adr r4, myarray
                            Mov r0, #10
                            str r0, [r4, r5]   ; r4[r5]=r0

int a= test[0]+5;           mov r5, #0         ;index of array
                            mov r4, myArray    ; should  change to adr r4, myarray
                            ldr r0, [r4, r5]   ;r0=r4[r5]
                            add r0, r0,#5
```

The **ADR** instruction loads an address within a certain range, without performing a data load

HOW TO PRESENT ARRAY IN ASSEMBLY

```
 adr r4, my array
ldr r2, [r4,#8]  ;r2 = r4[2] = 3

;enable for writing
mov r1,#1
lsl r1,#24
str r1,[r0,#4]

loop$:
;turn on LED
mov r1,#1
lsl r1,#18
str r1,[r0,#28]

mov r4,$80000
ldrd r6,r7,[r3,#4]
mov r5,r6
    loop1:
     ldrd r6,r7,[r3,#4]
     sub r8,r6,r5
     cmp r8,r4
    bls loop1
```

```
mov r1,#1
lsl r1,#18
str r1,[r0,#40]

mov r4,$80000
ldrd r6,r7,[r3,#4]
mov r5,r6
    loop2:
     ldrd r6,r7,[r3,#4]
     sub r8,r6,r5
     cmp r8,r4
    bls loop2

sub r2,#1
cmp r2,#0
bne loop$

myArray:
dw 1,2,3
```

DB

align 4

Label:

db "ascii array"

- A char array with 12 chars (nullterminated), stores the address of the first element in Label.
- This directive ensures the next memory declared/used in code (i.e, Label) starts at a byte address divisible by 4 (or any power of 2 provided).
- **Ensures array indexing of words is aligned with multiple of 4 addresses**.

CONTROL DISPLAY IN THE MONITOR

The Raspberry Pi GPU exchanges messages with the CPU using a "postman" paradigm.

– Both "chips" share a common bus.

- Messages are placed in a "mailbox" and can be polled, read, written or sent.
- Messages are sent to the GPU (VC or video core) or the CPU. This is a common paradigm for supercomputers, GPU programming and massively parallel processing.

There are 10 mailbox channels.

- Mailbox 1 writes to the screen (frame buffer)
- Mailbox 8 can be used for getting the location of the screen buffer. Once we have that we can write directly to the screen.

0: Power management (read-only)
1: Framebuffer (write-only)
2: Virtual UART (RS232)
3: VCHIQ (camera, audio)
4: LEDs
5: Buttons
6: Touch screen
7:
8: Property tags or "Mail Tags" (ARM -> VC)
9: Property tags (VC -> ARM)

➔ Using mailbox 8 for controlling display.

➔ *How?*

CONTROL DISPLAY IN THE MONITOR

To read from a mailbox:

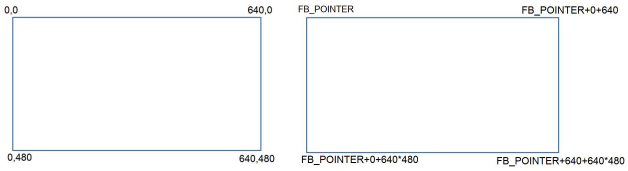
1. Read the status register until the empty flag is not set.
2. Read data from the read register.
3. If the lower four bits do not match the channel number desired then repeat from 1.
4. The upper 28 bits are the returned data.

To write to a mailbox:

1. Read the status register until the full flag is not set.
2. Write the data (shifted into the upper 28 bits) combined with the channel (in the lower four bits) to the write register.

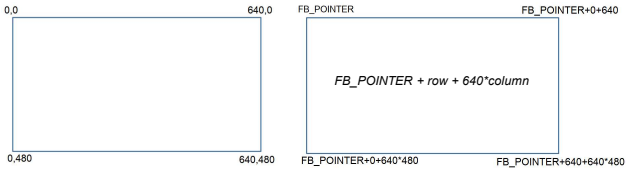
https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface

MONITOR COORDINATES



FB_POINTER to position (256, 32) ➔ FB_POINTER=????

MONITOR COORDINATES

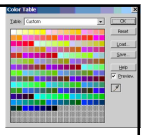


FB_POINTER to position (256, 32) ➔ x=256, y=32 ➔ 32 rows

```
Mov r1, #640
Lsl r1,r1,#5      ;20.480  = 640*32 = 640*2^5
Orr r1,#256       ; r1=r1+256
Add r0,r1         ; assume that r0=FB_POINTER
```

➡ *Insert data ➔ pixcel ➔ color*

COLOR



- *Color has 2 types*
  - *8 bit*
  - *16 bit*

8-bit Color

In 8-bit color there are $2^8 = 256$ possible tonal variations for each of the colors RGB. In a digital image, there are 3 channels that represent the colors RGB in a ***pixel*** (picture element). Each channel is 8 bits, thus we have a total of 24-bits per pixel (8 for Red, 8 for Green, 8 for Blue). The image can show a total of 16,777,216 colors (2^{24}). That is in millions of colors.

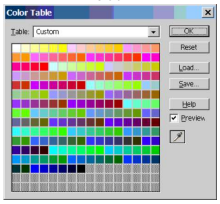
16-bit Color

In 16-bit color there are $2^{16} = 65,536$ possible tonal variations for each of the colors RGB. Each channel is 16 bits, thus we have a total of 48-bits per pixel (16 for Red, 16 for Green, 16 for Blue). The image can show a total of 281,474,976,710,656 colors (2^{48}). That is in trillions of colors.

https://www.raspberrypi.org/forums/viewtopic.php?t=11682
http://www.codeproject.com/Articles/7124/Image-Bit-depthconversion-from-32-Bit-to-8-Bit

COLOR

- *8 bit*



Example: White color=1

- *16 bit*

u16 code	color
0x0000	Black
0xFFFF	White
0xBDF7	Light Gray
0x7BEF	Dark Gray
0xF800	Red
0xFFE0	Yellow
0xFBE0	Orange
0x79E0	Brown
0x7E0	Green
0x7FF	Cyan
0x1F	Blue
0xF81F	Pink

We need to store color code to FB_POINTER ➔ str command.

str r1, [r0, #offset] ;please note that r1, r0 is 32 bit register. Our case is now 8 bit and 16 bit

COLOR

- *8 bit*

  strb r3, [r0] ;r3 stores 8 bit color, no offset in address (STRB (byte))
- *16 bit*

  strh r3, [r0] ;r3 stores 16 bit color; no offset in address (STRH (half-word))
- *Normally, we define 8 bit for 1 byte.*
- *Color has two types: 8bit/pixcel and 16bit/pixcel.*
- *If 16bit/pixcel ➔ How about solution?*

COLOR

FB_POINTER to position (256, 32) ➔ x=256, y=32 ➔ 32 rows

```
Mov r1, #640
Lsl r1,r1,#5        ;20.480  = 640*32 = 640*2^5
Orr r1,#256         ; r1=r1+256
Add r0,r1           ; assume that r0=FB_POINTER
```

Change to new code

```
    ;calc (y * 640 * BITS_PER_PIXEL / BITS PER BYTE)
    ;BITS_PER_PIXEL=8 (16) BITS PER BYTE=8
mov r1, #640
mul r1, #32  ; with r2=row; you can use Lsl r1,r1,#5
mul r1,r9  ;r9=BITS_PER_PIXEL  ➔ r1=r1*r9
lsr r8,#3  ;/8 bits per byte
add r0,r8   ;add y term
```

COLOR

Result

DRAWING TEXT

- *Char code*

```
00000000  0x00 0
00000000  0x00 0
00011000  0x18 24
00011000  0x18 24
00100100  0x24 36
00100100  0x24 36
00100100  0x24 36
01111110  0x7E 126
01000010  0x42 66
01000010  0x42 66
10000001  0x81 129
00000000  0x00 0
00000000  0x00 0
00000000  0x00 0
00000000  0x00 0
```

```
; $41: A
db 0,0,0,1,1,0,0,0
db 0,0,1,1,1,1,0,0
db 0,0,1,0,0,1,0,0
db 0,1,1,1,1,1,1,0
db 0,1,1,0,0,1,1,0
db 0,0,0,0,0,0,0,0
db 0,1,1,0,0,1,1,0
db 0,0,0,0,0,0,0,0

; $42: B
db 0,1,1,1,1,1,0,0
db 0,1,1,0,0,1,1,0
db 0,1,1,1,1,1,0,0
db 0,1,1,0,0,1,1,0
db 0,1,1,0,0,1,1,0
db 0,1,1,1,1,1,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
```

fonta:
.long
0x00,0x00,0x18,0x18,0x24,0x24,0x24,0x7E,0x42,0x42,0x81,0x00,0x00,0x00,0x00 ;/'A'

ARM Compiler armasm User Guide Version 5.04

LDR (immediate offset)

Load with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doubleword
LDRD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, doubleword
```

4.9.9 Examples

ldr r6, [r5], 4

```
STR R1,[R2,R4]!     ; Store R1 at R2+R4 (both of which are
                    ; registers) and write back address to
                    ; R2.
STR R1,[R2],R4      ; Store R1 at R2 and write back
                    ; R2+R4 to R2.
```


DRAWING TEXT

- *How to load char code*

```
; $41: A
db 0,0,0,1,1,0,0,0
db 0,0,1,1,1,1,0,0
db 0,0,1,0,0,1,0,0
db 0,1,1,1,1,1,1,0
db 0,1,1,0,0,1,1,0
db 0,1,1,0,0,1,1,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0

; $42: B
db 0,1,1,1,1,1,0,0
db 0,1,1,0,0,1,1,0
db 0,1,1,1,1,1,1,0
db 0,1,1,0,0,1,1,0
db 0,1,1,0,0,1,1,0
db 0,1,1,1,1,1,1,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
```

```
r0=[FB_POINTER] ; r5 is address of 'A'
mov r4, #8
DrawChar:
ldr r6, [r5], 4     ; Load Font Text Character 1/2 Row
str r6,[r0],4       ; Store Font Text Character 1/2 Row To Frame Buffer
ldr r6,[r5],4       ; Load Font Text Character 1/2 Row
str r6,[r0],4       ; Store Font Text Character 1/2 Row To Frame Buffer
subs r4,1           ; Decrement Character Row Counter
bne DrawChar        ; IF (Character Row Counter != 0) ;DrawChar
```

Question: How/why about load in 8 bit?

Answer: In experiment time

DRAWING TEXT

- Define text in assembly

```
Text:
  db " Open!"
align 4
Text2:
  db "Closed"
```

- Link register to text in assembly

Command: adr (Address-relative)

adr r0,label

Example:

```
adr r2,Text
```

DRAWING TEXT

```
adr r2, Text                ; load text
adr r1, Font                ; load font
mov r3,#6                   ; 6 characters
DrawChars:
 mov r4,#8                  ; R4 = Character Row Counter
 ldrb r5,[r2],1             ; R5 = Next Text Character
 add r5, r1, r5, lsl 6      ; Add Shift To Correct Position In Font (* 64)
 bl DrawChar
 subs r3,1                  ; Subtract Number Of Text Characters To Print
 subne r0,SCREEN_X * CHAR_Y ; Jump To Top Of Char
 addne r0,CHAR_X            ; Jump Forward 1 Char
 bne DrawChars
        ; IF (Number Of Text Characters != 0) Continue To Print Characters
b cont
```

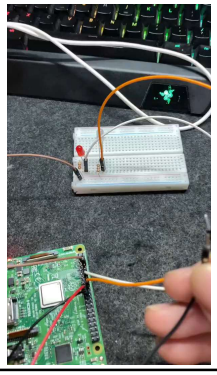
```
times ((0*)*320) db 0 ; Characters 0 to 01F

; 320: Space " "
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0

; 321: Exclamation mark "!"
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,0,0,0,0,0

; 322: Quotation mark """
db 0,1,1,0,0,1,1,0
db 0,1,1,0,0,1,1,0
db 0,1,1,0,0,1,1,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0

; 323: Cross hatch "#"
db 0,0,0,1,0,1,0,0
db 0,0,0,1,0,1,0,0
db 0,1,1,1,1,1,1,0
db 0,0,1,1,0,1,0,0
db 0,1,1,1,1,1,1,0
db 0,0,1,1,0,1,0,0
db 0,0,1,1,0,1,0,0
db 0,0,0,0,0,0,0,0
```



DRAWING TEXT

```
add r5, r1, r5, lsl 6  ; Add Shift To Correct Position In Font (* 64)
```

;r1 font

;r5 character 'O'

➔

Shift from beginning to 64 times to go to next character (8x8 codes for one character)

For example:

'A', need to shift 'A'*64 times. ➔ offset

Then add offset to first address ➔ find position of 'A' in Font

```
times ((0*)*320) db 0 ; Characters 0 to 01F

; 320: Space " "
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0

; 321: Exclamation mark "!"
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,1,1,0,0,0
db 0,0,0,0,0,0,0,0

; 322: Quotation mark """
db 0,1,1,0,0,1,1,0
db 0,1,1,0,0,1,1,0
db 0,1,1,0,0,1,1,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0

; 323: Cross hatch "#"
db 0,0,0,1,0,1,0,0
db 0,0,0,1,0,1,0,0
db 0,1,1,1,1,1,1,0
db 0,0,1,1,0,1,0,0
db 0,1,1,1,1,1,1,0
db 0,0,1,1,0,1,0,0
db 0,0,1,1,0,1,0,0
db 0,0,0,0,0,0,0,0
```

FINISH