

# COS10004 Computer Systems

## Lecture 10.2 ARM Assembly - Arrays

CRICOS provider 00111D

```
.section .data
```

```
text:
```

```
.ascii "Chris McCarthy\n\0"
```

# ARRAYS

- Sometimes we want to store multiple values in an index-able list
- Most programming languages support this via arrays:
  - A collection of data stored in contiguous memory

# ARRAYS

- Example in C/C++

```
// create an array called "an_array" of 100 integers
```

```
int an_array[100];
```

```
// loop through and index each element of "an_array" , and print the value
```

```
for (int i = 0; i < 100; i++)
```

```
{
```

```
    printf("%d", an_array[i])
```

```
}
```

# ARRAYS IN ASM

- In ASM we can create arrays, but need to do a bit more of the work.
- Think about what an array requires:
  - An address in memory for the first element
  - A known constant offset to the next element (in bytes)
  - An uninterrupted contiguous block of reserved space
    - up to the predetermined size of the array (in bytes)

# ARRAYS IN ASM

- Start with a label:
- Follow with the data type and then list the array elements separated by ,

e.g.

```
myArray:  
    .int 1,2,3,4,5,6,7,8  
myName:  
    .ascii "James Hamlyn-Harris\0"  
myNum:  
    dw $F0002000
```

the label becomes the  
address/pointer to the  
array

dw = define word (32 bit)  
db = define byte (8 bit)

# ITERATING THROUGH AN ARRAY

- Get the array address and add an offset (index) to it
  - For characters offset is 1 byte
  - For integers offset is 4 bytes
- To get r0 to be set to each value in the array:

```
; use r5 as the index
; use r4 to point to the array
mov r5,#0 ;i=0
mov r4,myArray ;gets the pointer
loop1:
    ldr r0, [r4,r5]    ; in C: r0 = r4[r5]
    ;do something with r0
    add r5,#4 ;i++
b loop1
```

# USING SHIFT TO ITERATE THROUGH A BIT PATTERN (OK05)

- The SOS flashing Baking Pi tutorial stores a bit pattern (binary) as a single .int, and uses *ls* to select the bit needed.
- This code stores the "array" of bits (actually a 32-bit .int)

```
align 2    ;make next line start at a multiple of 22
pattern:   ;int pattern = 4188808
int 0b111111111101010100010001000101010
```

Padding

S (...)

O (---)

S (...)

mov needs a  
multiple of  
4

# USING SHIFT TO ITERATE THROUGH A BIT PATTERN (OK05)

```
mov r4,pattern ;store the int in register 4
mov r4,[r4] ;r4 now points to the data
mov r5,#0 ;start pattern with i = 0
loop$:
    mov r0,#16 ;Select the LED
    mov r1,#1
    lsl r1,r5 ;shift by the index to select the bit
    and r1,r4 ;r1 = 1 AND bit at r4
    bl SetGpio ;call function to write r1 to LED (0 or 1)
    mov r0,$030000 ;~0.2s
    bl Wait ;call function which waits for 1/4 sec
    add r5,#1 ;i++
    and r5,#0b11111 ;reset if we overflow
b loop$
```



# DB

```
align 4
```

```
Label:
```

```
db "ascii array"
```

initialises a char array with 12 chars (null-terminated), stores the address of the first element in Label.

# ALIGN

**align 4**

**Label:**

**db "ascii array"**

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.

# SUMMARY

- Arrays are fundamental data structures in computer programming
  - Contiguous blocks of index-able memory
- ARM assembly gives us the tools to create them:
  - Labels (address)
  - Memory allocation:
    - db, dw etc
- Indexing requires knowing:
  - The position of the element you want
  - bytes per element (i.e, size of each element)

# WRITING TO THE SCREEN

- 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.

# MAILBOXES (CHANNELS)

- 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)

Mailbox 8 seems to be more reliable than mailbox 1 for graphics.

Allows us to set up the screen and get a pointer to it.

# COMMUNICATING WITH A MAILBOX

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.

# MAILBOX 8 AND THE FRAME BUFFER

- Set Tags (key-multiple value pairs) making requests of the VC.
- Answers are written over the requests (in the same memory addresses).
- Detailed procedure here:
- <https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface#allocate-buffer>

# STRUCTS

- The message we send to the mailbox includes the memory location of our request – a large struct containing Tags and Values.
- Some of the tags will contain dummy values – to be overwritten by the answer received from the VC.
- A struct in ASM looks like this:

```
LABEL:  
Type1 value1 ;could be  
a Tag (a previously  
defined "magic number"  
constant)  
Type2 value2  
Type1 value3  
Type1 value4
```

```
FB_STRUCT:  
dw Set_Physical_Display ; Tag  
dw $00000008 ; Value  
dw $00000008 ;  
dw SCREEN_X ; constant  
dw SCREEN_Y ;  
FB_POINTER: ;pointer to screen  
dw 0 ; Value Buffer  
dw 0 ; Value Buffer
```



# TAGS WE NEED TO SEND

```

align 16
FB_STRUCT: ; Mailbox Property Interface Buffer
Structure
    dw FB_STRUCT_END - FB_STRUCT ; Buffer Size In
Bytes
    dw $00000000 ; Buffer Request/Response Code
; Sequence Of Concatenated Tags
    dw Set_Physical_Display ; Tag Identifier
    dw $00000008 ; Value Buffer Size In Bytes
    dw $00000008 ; 1 bit (MSB) Request/Response
Indicator (0=Request, 1=Response), 31 bits (LSB) Value
Length In Bytes
    dw SCREEN_X ; Value Buffer
    dw SCREEN_Y ; Value Buffer
    dw Set_Virtual_Buffer ; Tag Identifier
    dw $00000008 ; Value Buffer Size In Bytes
    dw $00000008 ;
    dw SCREEN_X ; Value Buffer
    dw SCREEN_Y ; Value Buffer
    dw Set_Depth ; Tag Identifier
    dw $00000004 ; Value Buffer Size In Bytes
    dw $00000004 ; 1 bit (MSB) Request/Response
Indicator (0=Request, 1=Response), 31 bits (LSB) Value
Length In Bytes
    dw BITS_PER_PIXEL ; Value Buffer
    dw Set_Virtual_Offset ; Tag Identifier
    dw $00000008 ; Value Buffer Size In Bytes
    dw $00000008 ; 1 bit (MSB) Request/Response

```

```

Indicator (0=Request, 1=Response), 31 bits (LSB) Value
Length In Bytes
FB_OFFSET_X:
    dw 0 ; Value Buffer
FB_OFFSET_Y:
    dw 0 ; Value Buffer
    dw Set_Palette ; Tag Identifier
    dw $00000010 ; Value Buffer Size In Bytes
    dw $00000010 ; 1 bit (MSB) Request/Response
Indicator (0=Request, 1=Response), 31 bits (LSB) Value
Length In Bytes
    dw 0 ; Value Buffer (Offset: First Palette Index To Set
(0-255))
    dw 2 ; Value Buffer (Length: Number Of Palette
Entries To Set (1-256))
FB_PAL:
    dw $00000000,$FFFFFF ; RGBA Palette Values
(Offset To Offset+Length-1)
    dw Allocate_Buffer ; Tag Identifier
    dw $00000008 ; Value Buffer Size In Bytes
    dw $00000008 ; 1 bit (MSB) Request/Response
Indicator (0=Request, 1=Response), 31 bits (LSB) Value
Length In Bytes
FB_POINTER: ;pointer to start of screen
    dw 0 ; Value Buffer
    dw 0 ; Value Buffer
    dw $00000000 ; $0 (End Tag)
FB_STRUCT_END: ;used to calculate length in bytes

```

# ALIGN

align 16

Any power of 2

We need this for ARM mailbox because the channel number is stored in the lowest 4 bits of the message struct. Aligning it to 16 ensure that the lowest bits are clear.

- Used when declaring arrays, structs and functions
- Forces starting address to be a multiple of (in this case 16 bytes).
- Forces padding with 0 or NOP instructions
- Improves performance by ensuring that CPU caches are efficiently used (set align to cache width).
- Default for ARM is 4
- Sometimes solves "illegal value" errors by making address of struct/array byte-aligned.

# ORG

org \$8000



Any power of 2

- Origin command
- Specifies the value of the location counter (the address to load the code).
- Useful where hardware resources are mapped to memory addresses. ORG stops them from being moved around or overwritten at run time.
- ORG lets you do crazy things like overwrite data with code, or make Unions (C language).

# ADR

- Address-relative

**adr r0, label**



destination

source (pointer)

- A pseudo-instruction which gets the address (relative to the current instruction) (pointer) of a variable/array/struct and puts it in a register.
- Useful when address of code at run time is unknown.

# SUBNE

- Subtract if not equal

**subne r1, r2 ; r1=r1-r2**

**subne r1, r1, r2**

**subne r0, r3, #4**

if only 2 parameters,  
difference is copied into  
1<sup>st</sup> one.



destination

expression

- subtract if not equal (checks APSR for result of previous cmp or xxxs)

# LDRB

- Load register with byte

`ldrb r5,[r2],#1 ;R5 = Next Text Character`



destination

source is what  $r2 + 1$  points to

- #1 can be replaced with a register containing the array index.
- Load the value into the lowest byte of a register.
- Gets the pointer in r2, adds 1 and gets the lowest byte it finds there. Puts it in r5.

# DRAWING A PIXEL

1. Define constants and magic numbers (tags)
2. Initialise the frame buffer
  - Concatenate the channel number (8), struct address to make a message.
  - Concatenate the channel number (8), BASE address, Mailbox address, write register to make a destination address.
  - STR (send message to video core)
  - LDR response (read screen pointer – if 0 repeat)
3. Calculate Pixel address and set pixel.

# THE ARM GPU MAILBOX

- The mailbox is at BASE + B880:

Address offset	Size / Bytes	Name	Description	Read or Write
0x0000	4	Read	Read mail	R
0x0010	4	Poll	Receive without retrieving.	R
0x0014	4	Sender	Sender information.	R
0x0018	4	Status	Information.	R
0x001C	4	Config.	Settings	RW
0x0020	4	Write	Send mail.	W



# INITIALISE THE FRAME BUFFER 1

- Set up the constants and magic numbers (tags)

```
BASE = $3F000000 ; 2
```

```
SCREEN_X      = 640
```

```
SCREEN_Y      = 480
```

```
BITS_PER_PIXEL = 8
```

```
;memory addresses of mailbox
```

```
MAIL_BASE = $B880 ; separate into $B800 and $0080
```

```
MAIL_WRITE = $20 ;offset for WRITE register
```

```
MAIL_TAGS = $08 ;Channel number stored in the lowest 4 bits
```

```
;memory addresses of GPU tags (key-value pairs user to program the GPU)
```

```
Allocate_Buffer_address = $00040001 ; 0 (request), returns FB
```

```
Set_Physical_Display = $00048003 ; 640,480
```

```
Set_Virtual_Buffer = $00048004 ; 640,480
```

```
Set_Depth = $00048005 ; 8 (Response: Bits Per Pixel)
```

```
Set_Virtual_Offset = $00048009 ; 0,0 (Response: X In Pixels, Y In Pixels)
```

```
Set_Palette = $0004800B ; 0,2 (first index, value)
```

- many more here:
- <https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface>

# INITIALISE THE FRAME BUFFER 2

- Set up the struct (see slide 40)
- Send the struct address to the VC

FB\_Init:

*;FB\_STRUCT is determined at run-time. If it is an illegal value, orr it into the register 1 byte at a time.*

mov r0,FB\_STRUCT and \$FF

orr r0,FB\_STRUCT and \$FF00

orr r0,FB\_STRUCT and \$FF0000

orr r0,FB\_STRUCT and \$FF000000

orr r0,MAIL\_TAGS *;send key-value pairs to GPU*

*;combine the channel number (8), mailbox address, write register address into r1*

mov r1,BASE

orr r1,MAIL\_BASE and \$00FF

orr r1,MAIL\_BASE and \$FF00

orr r1,MAIL\_WRITE

orr r1,MAIL\_TAGS

*; next: send the struct location (r0) to mailbox write register for channel 8 (r1)*

# INITIALISE THE FRAME BUFFER 3

- The mailbox returns the reply using the same struct structure and location.
- The struct we sent includes a tag for FB-POINTER, and this is populated with the address of the screen (if available).

a tag (location) in the struct

```
str r0,[r1] ;Mail Box Write
ldr r0,[FB_POINTER] ; send address of FB struct to mailbox
;mailbox delivers to GPU, sends back reply (0 fail) or pointer to screen
cmp r0,0 ; Compare Frame Buffer Pointer To Zero
beq FB_Init ; IF Zero try again
```

# MAIL FORMAT

Value																												channel			
3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0

- Writing to the Mailbox sends one parameter, which contains a channel and a value combined in one register.
- The value is the address of the FB-STRUCT but it is **aligned 16, so the lowest 4 bits MUST be zero.**
- **Add the channel number (8) – no overlap!**
- Reading from the Mailbox returns the same format.
- The value is the location of the struct, but now the output fields have been filled in.
- The FB\_POINTER field is either 0 (failed) or the pointer to the screen (top 32 bits).

# WHAT CAN WE SEND TO THE FRAME BUFFER (SCREEN) ADDRESS?

A pixel:

- ~~Screen coordinates~~
- A memory address(calculated)
- A ~~colour~~
- colour number
- RGB in hi-colour format: 16 bits:

# WRITE PIXEL TO THE SCREEN

- if `ldr r0, [FB_POINTER]` returns a non-zero number, that is the pointer to the screen. We need to save this somewhere. We can use it from now on:

`; Draw Pixel`

`;r0 now contains address of screen`

`mov r7,r0 ;keep a copy for later (just in case)`

`;do some maths to find the address of a place on the screen`

`mov r1,#640 ;(screen width we asked for)`

`lsl r1,r1,5 ;2^5 = 32 = y ordinate`

`orr r1,#256 ;x ordinate`

`add r0,r1 ; Place Text At XY Position 256,32`

`;r6 is what we want to write`

`mov r6,#1 ;colour 0-1 (assumes 8-bit colour)`

`str r6,[r0],4`

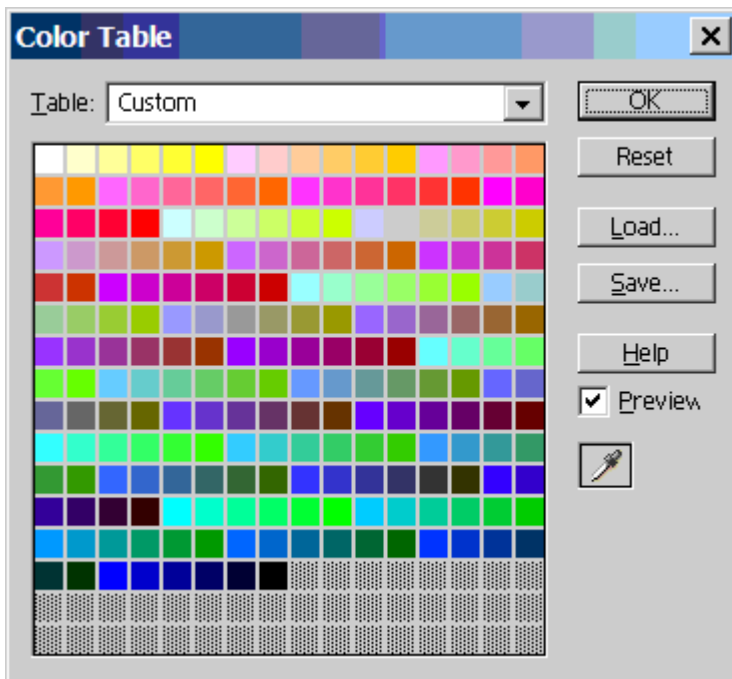
But there's a mistake here. We are writing a 4-byte register to a 1-byte pixel. What will go wrong?

# ADDING WORDS TO BYTES

- Each word (4 bytes) written to the screen will overwrite adjacent bytes for neighboring pixels.
- Need a way to only write over 1 byte (8-bit colour) or 2 bytes (16-bit colour).
- Replace STR (word) with  
STRH (half-word) or  
STRB (byte)

# 8-BIT COLOUR?

- Apparently RPi 8-bit colour is the web-safe colour palette:



<https://www.raspberrypi.org/forums/viewtopic.php?t=11682>

<http://www.codeproject.com/Articles/7124/Image-Bit-depth-conversion-from-32-Bit-to-8-Bit>

Colour 1 = white



# WRITE PIXEL TO THE SCREEN 2



# RPI 16-BIT COLOUR

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

- If we initialise the frame buffer and ask for 16 bits per pixel, we get 16-bit colour.
- It uses 5 bits for Red, 6 bits for Green and 5 bits for Blue.
- Easier to use these example values.

Red					Green						Blue				
15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

# MUL

**mul r1, r1, #16**

2<sup>nd</sup>  
parameter

register 1

value in  
decimal

literal value  
(not a  
pointer)

mul r1,\$C ;r1\*=12  
mul r1, r2, r3 ;r1=r2\*r3

- load register 1 with the product of r1 and 16.
  - Slower than **lsl** but more versatile (don't need to know multiplier at compile time).

# BLS

- branch if less than or the same

**bls label**



Does NOT set the  
link register

Could be a label or  
a function name

- Checks the ASPR for the results of the previous calculation (cmp or xxxs)

# LSR

- logical shift right

```
lsr r1, #3
```

```
lsr r1, r1, #3
```



destination

expression

- divides `r1` by 8 ( $2^3$ ) and writes the answer back into `r1`.

# USING CHANNEL 1

- Provides an alternate way to access the screen buffer.
- Works on Rpi B, B+.
- Buggy on model 2 (due to L2 cache readdressing)

“With the exception of the property tags mailbox channel, when passing memory addresses as the data part of a mailbox message, the addresses should be bus addresses as seen from the VC. These vary depending on whether the L2 cache is enabled. If it is, physical memory is mapped to start at 0x40000000 by the VC MMU; if L2 caching is disabled, physical memory is mapped to start at 0xC0000000 by the VC MMU. Returned addresses (both those returned in the data part of the mailbox response and any written into the buffer you passed) will also be as mapped by the VC MMU. In the exceptional case when you are using the property tags mailbox channel you should send and receive physical addresses (the same as you'd see from the ARM before enabling the MMU).”

<https://github.com/raspberrypi/firmware/wiki/Accessing-mailboxes>

Obvious, isn't it?

# FUNCTIONS FOR OUR OS

- *Turn on/off GPIO (gpio.s) (OK01, OK02)* 😊
- *Flash LED (flash.s) - for diagnostics* ←
- *Wait (wait.s) (OK04)* 😊
- *Draw Pixel colour, position (Screen01)* 😊
- *Draw Character (colour, char, position) (Screen04)*
- *Draw text (colour, position) (Screen04)*
- *Process input*

# DEBUGGING?

- How do you debug your code if you can't write messages to the screen?
- Write messages to the LED!

Write a function which flashes an LED

The R Pi already has a POST routine:

3 flashes: start.elf not found

4 flashes: start.elf not launched

7 flashes: kernel.img not found

Persistent : kernel.img not running

8 flashes: SDRAM not recognised.

You need newer bootcode.bin/start.elf firmware.

Little  in the corner: undervoltage warning.



# FLASH.ASM

```
;flash red LED once
FLASH:
GPIO_OFFSET = $200000
mov sp,$8000
push {r0-r9}
mov r0,BASE
orr r0,GPIO_OFFSET ;Base address of GPIO
mov r1,#1
lsl r1,#15 ;B+
str r1,[r0,#12] ;enable output
mov r1,#1
lsl r1,#3
    str r1,[r0,#44] ;Turn off LED
    ;new timer
TIMER_OFFSET = $3000
mov r3,BASE
orr r3,TIMER_OFFSET ;store base address of
timer (r3)
mov r4,$70000
orr r4,$0A100
orr r4,$00020 ;TIMER_MICROSECONDS =
500,000
    ;store delay (r4)
```

```
    ldrd r6,r7,[r3,#4]
    mov r5,r6 ;store starttime (r5)
    (=currenttime (r6))
loop1:
    ldrd r6,r7,[r3,#4] ;read currenttime (r6)
    sub r8,r6,r5 ;remainingtime (8)=
currenttime (r6) - starttime (r5)
    cmp r8,r4 ;compare remainingtime (r8),
delay (r4)
    bls loop1 ;loop if LE (remainingtime <=
delay)
    str r1,[r0,#32] ;turn on LED
    ;re-use timer
    ldrd r6,r7,[r3,#4]
    mov r5,r6 ;store starttime (r5)
    (=currenttime (r6))
loop2:
    ldrd r6,r7,[r3,#4] ;read currenttime (r6)
    sub r8,r6,r5 ;remainingtime (8)=
currenttime (r6) - starttime (r5)
    cmp r8,r4 ;compare remainingtime (r8),
delay (r4)
    bls loop2 ;loop if LE (remainingtime <=
delay)
pop {r0-r9}
bx lr
```

# push

# Coffee time

How many cups of coffee ( $Q$ ) can I make from my  $N$  beans?

It takes  $D$  beans to make one cup.



# Coffee time

## Division in Assembly...

- Intel CPUs have a div operation with 2 inputs and 2 outputs
- Inputs: Numerator (dividend),  
Denominator (divisor)
- Outputs: Quotient (result),  
Remainder (modulus)
- But ARM is RISC (reduced instruction set computer).
- THERE IS NO DIV in ARM assembly!
- <http://www.info.uni-karlsruhe.de/lehre/2003SS/asm/material/arm/arm-intro.pdf>

# Wikipedia to the rescue!

[http://en.wikipedia.org/wiki/Division\\_algorithm](http://en.wikipedia.org/wiki/Division_algorithm)

# Dividing without div - pseudocode

Q,R divide(N,D)

if D = 0 then error(DivisionByZero) end

if D < 0 then (Q,R) := divide(N, -D); return (-Q, R)

end

if N < 0 then (Q,R) := divide(-N, D)

if R = 0 then return (-Q, 0)

else

return (-Q - 1, D - R) end

end

;At this point,  $N \geq 0$  and  $D > 0$

Q := 0; R := N

while R ≥ D

Q := Q + 1

R := R - D end

return (Q, R) end



It's recursive!

# Dividing without div - Assembly

```
;params: Numerator (r0), Denominator (r1)  
;returns: Quotient (r0), Remainder (r1)
```

Divide:

```
;validate (quick and dirty)  
cmp r1,#0  
ble exit ;denom must be >0  
cmp r0,#0  
ble exit ;numerator must be >0  
;At this point,  $N > 0$  and  $D > 0$   
;could do a lot more validation (previous  
slide)  
...
```

...

**;At this point,  $N > 0$  and  $D > 0$**

**;N**    **r4**

**;D**    **r5**

**;Q**    **r0**

**;R**    **r1**

**mov r4,r0**

**mov r5,r1**

**mov r0,#0**

**mov r1,r4**

**loop:**

**cmp r1,r5**

**bgt exit**

**add r0,r0,#1**

**sub r1,r1,r5        ;do a cmp**

**b loop**

**exit:**

**bx lr    ;return (Q, R)**



# Try it!

```
mov r4,r0
mov r5,r1
mov r0,#0
mov r1,r4
loop:
    cmp r1,r5
    bgt exit
    add r0,r0,#1
    sub
r1,r1,r5 ;do a cmp
b loop
exit:
```

iteration	N (r4)	D (r5)	Q (r0)	R (r1)	R>D
0					
1					
2					
3					
4					

pop

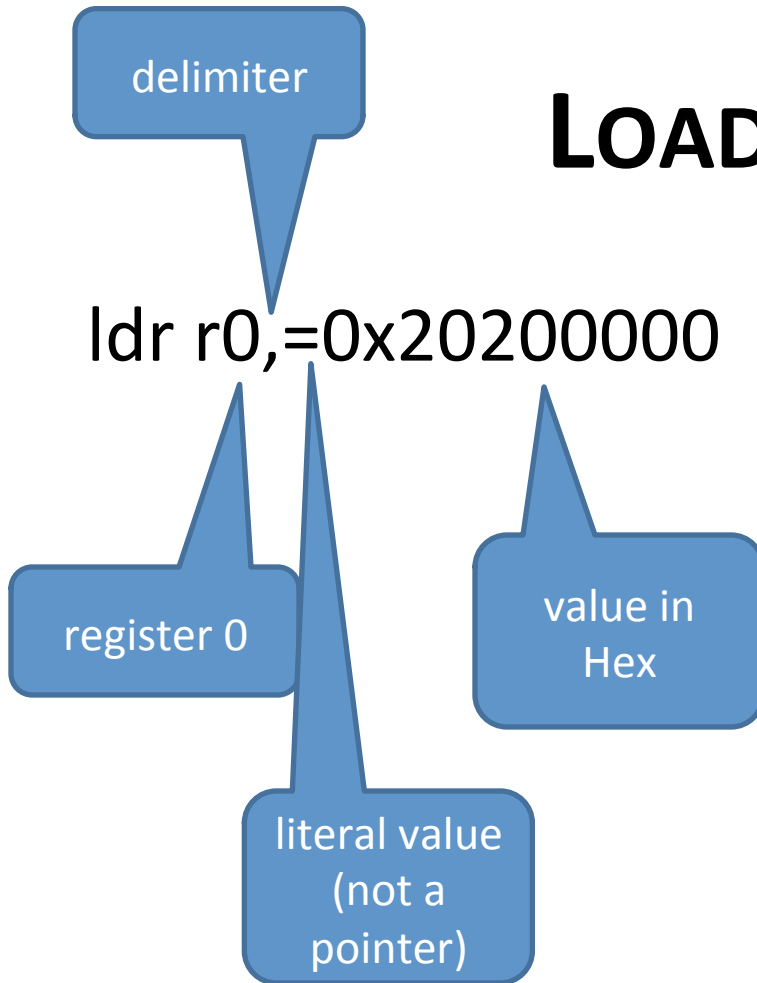
# FUNCTIONS FOR OUR OS

- *Turn on/off GPIO (gpio.s) (OK01, OK02)* 😊
- *Flash LED (flash.s) - for diagnostics* 😊
- *Wait (wait.s) (OK04)* 😊
- *Draw Pixel colour, position (Screen01)* 😊
- *Draw Character (colour, char, position)* ←
- *Draw text (colour, position) (Screen03)* ←
- *Get Char (Input01)*
- *Process command (Input02)*

# SECOND ASSIGNMENT (PART A)

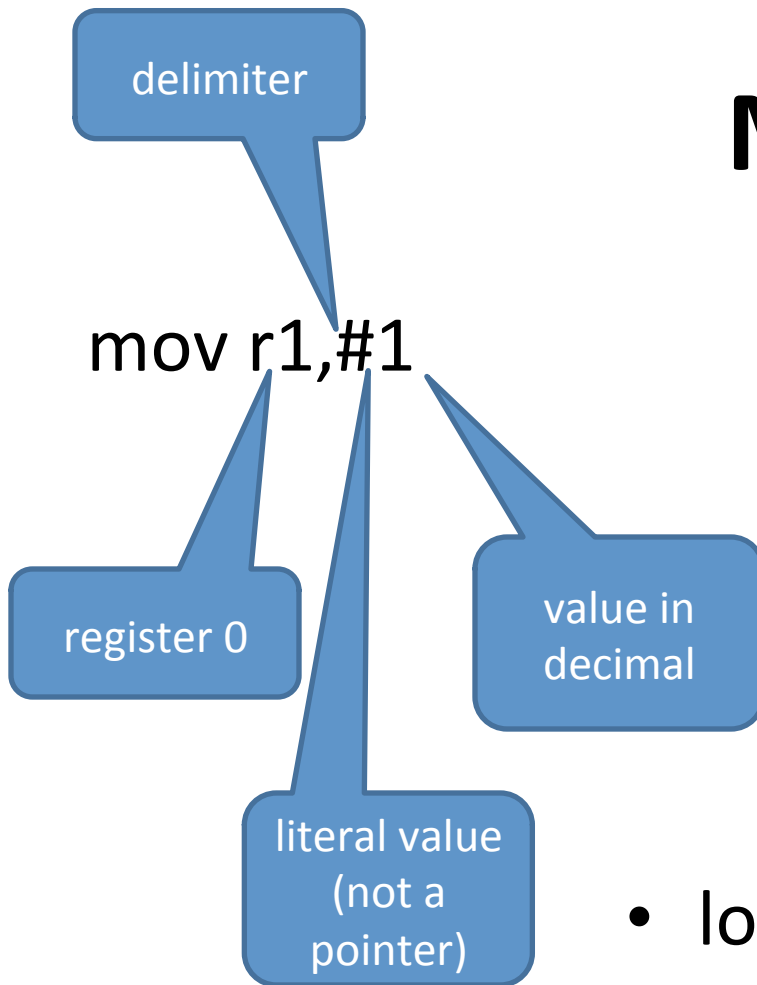
- ASM commands
- Part A:  
Write a manual covering every ASM command we have used this semester
  - The command syntax
  - What it does (incl. what registers it changes)
  - Example code
  - Special stuff (e.g. must use an even register)
- Something like these (next 3 slides) but in a more compact format...

# LOAD REGISTER



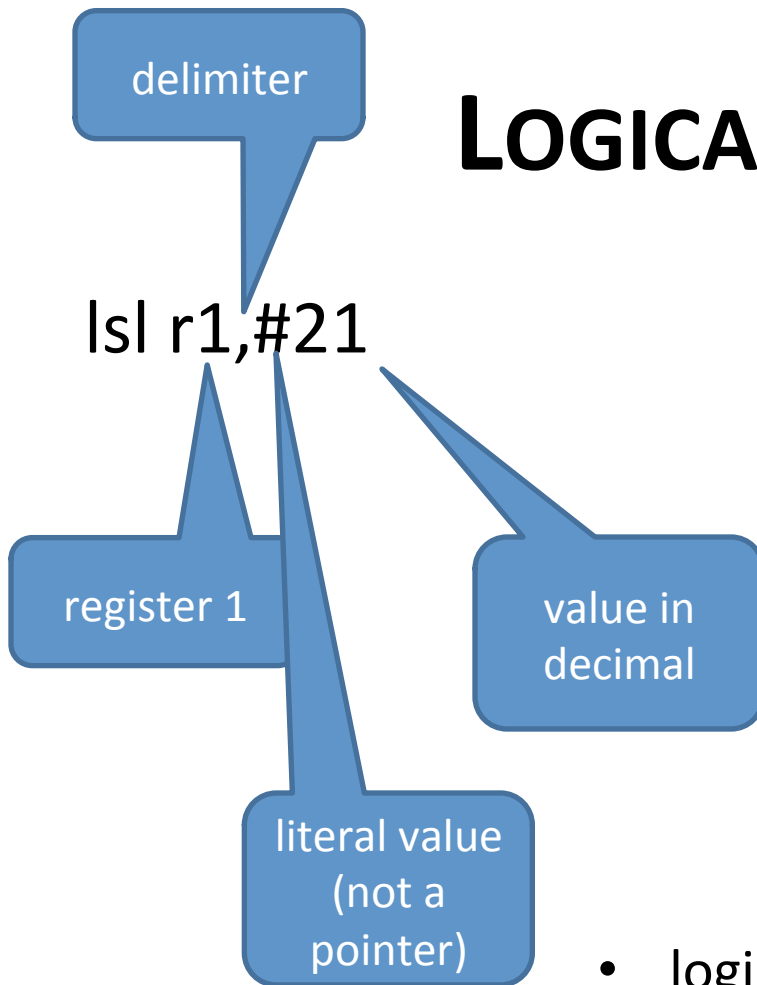
- load register 0 with the value 0x20200000 (538968064)

# MOVE



- load register 1 with the value 1
- Faster than `ldr` but only works with numbers with 8 consecutive 0s in the binary representation.

# LOGICAL SHIFT LEFT



- logical shift left (double) register 1 21 times
  - (multiply by  $2^{21}$ )
  - (`r1 = 2097152 = 0x200000`)
    - (bit 21 is set)
- (`00000000 00100000 00000000 00000000`)

# ASSIGNMENT 2 PART B

- 50% (optional)
- Write something awesome in ARM assembly.
- Must run on the R Pi (B+, 2 or 3).
- Can use extra hardware (within reason).
- Can use an operating system
- Demonstrate to your tutor in Week 12 lab.



# ASSIGNMENT IDEAS

- Complete other parts of the baking Pi series, with your own code - not the standard answers.
- Adapt and extend Peter Lemon's code
- Control some hardware/ electronics/ peripherals (extra LEDs? Add buttons? build and program a keypad?)
- Extend the labs - more graphics, text, inputs.
- Read the mouse (really hard).
- Program a phone keypad (22, 33, = b,e, )
- Plug-in boards.
- Games! Snake, Etch-s-sketch, Pong.

# ASSIGNMENT 2 PART B

- What is awesome?
- Awesome (like a hot dog)
- Awesome (like destroying the world)
- Marks depend on awesomeness.
- modify some existing code = mildly awesome
- new code / functions = awesome-er
- Interesting – code does something new = very awesome
- Amazing – Code does something we didn't expect = extreme awesomeness

# CONSTRAINTS

- Must be legal
- Must be ethical
- No teams (but you can collaborate - each contribution must be different)
- Must use the R Pi
- Must use ARM Assembly
- Must not destroy the world

# DRAWING TEXT

- Let's start small - draw a character
- Need to be able to represent a char as data
- Array of bits?
- One integer (32 bits should be enough)?

# THE LETTER A (8x16 DOTS)

```
00000000
00000000
00011000
00011000
00100100
00100100
00100100
01111110
01000010
01000010
10000001
00000000
00000000
00000000
00000000
```

# THE LETTER A (8x16 DOTS)

00000000	0x00	0
00000000	0x00	0
000 <b>11</b> 000	0x18	24
000 <b>11</b> 000	0x18	24
00 <b>1001</b> 00	0x24	36
00 <b>1001</b> 00	0x24	36
00 <b>1001</b> 00	0x24	36
0 <b>11111</b> 10	0x7E	126
0 <b>10000</b> 10	0x42	66
0 <b>10000</b> 10	0x42	66
<b>1000000</b> 1	0x81	129
00000000	0x00	0
00000000	0x00	0
00000000	0x00	0
00000000	0x00	0

# THE LETTER A (8x16 DOTS)

00000000	0x00	0
00000000	0x00	0
000 <b>11</b> 000	0x18	24
000 <b>11</b> 000	0x18	24
00 <b>100</b> 100	0x24	36
00 <b>100</b> 100	0x24	36
00 <b>100</b> 100	0x24	36
0 <b>11111</b> 10	0x7E	126
0 <b>10000</b> 10	0x42	66
0 <b>10000</b> 10	0x42	66
<b>1000000</b> 1	0x81	129
00000000	0x00	0
00000000	0x00	0
00000000	0x00	0
00000000	0x00	0

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

# FONT8x8.ASM

- Peter Lemon has provided an 8x8 font file. We'll use it.
- He has sample code on his GitHub site. We'll adapt it.
  - Each character takes up 8 bytes
- To get the letter 'A' (ASCII 65),
  - set up a label Font, align 4 and include the font file,
  - load the file,
  - store the address of the label (start of the array),
  - Set up some text (Text:...db...)
  - include DrawChar (Peter's nested loops) and start drawing.



```

; ...set up BASE address,
; ...call FB_Init
; Setup Characters
CHAR_X = 8
CHAR_Y = 8
mov r0,r7
mov r1,SCREEN_X
lsl r1,r1,5 ;32
orr r1,#192
add r0,r1 ; Place Text At XY Position 256,32
adr r1,Font ; R1 = Characters
adr r2,Text ; R2 = Text Offset
mov r3,#29 ; R3 = Number Of Text Characters To Print
DrawChars:
    mov r4,CHAR_Y ; 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 ;call Peter's code
    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
Loop:
    b Loop ;wait forever

include "FBinit8.asm"
include "DrawChar.asm"
Text:
    db "Hello Computer Systems World!"
align 4
Font:
    include "Font8x8.asm"

```

from Char02-channel8 – ASCII.zip

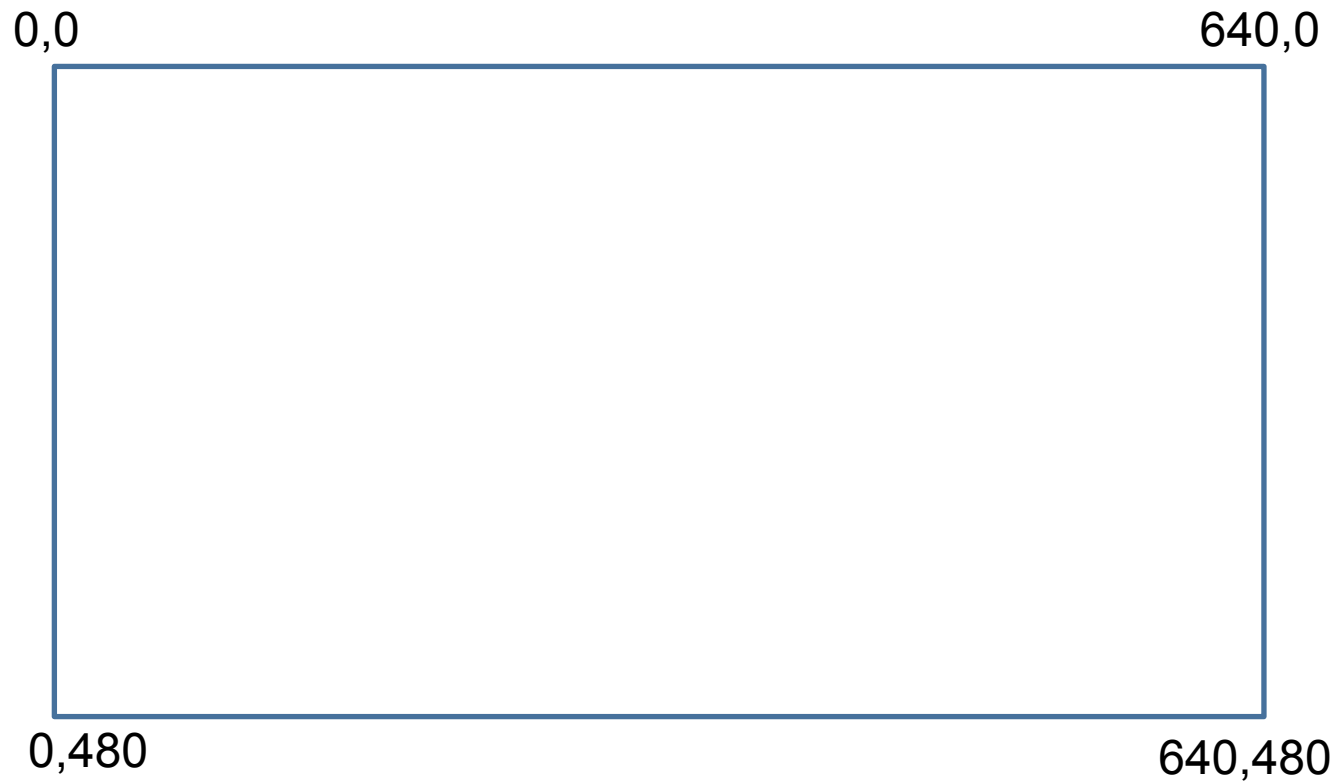
# HOW DID WE PRINT THE CHARS?

- We need to draw a box (8 wide x 8 deep)
  - Fill in the pixels which correspond to set bits
- Read the bits (of the font) 1 row (byte) at a time
  - This will read the higher bits first (left-most), so we need a bit mask which starts at 7 and counts down to 0
  - On the screen, we compare the bit mask to the byte read from the font,
  - and display the pixel for it if the bit is set.

# DRAWING TEXT

- Need to call DrawChar for each char in an array.
- Read the array N times (r3).
- Alternatively we could detect when we get to the null '\0' character.
- Read/process other special characters
  - e.g. '\n' means  $x=0$  and  $y+=8$

# LOGICAL COORDINATES



# PHYSICAL ADDRESSES

- Note:  $r0 = r7 = \text{FB\_POINTER}$

FB\_POINTER

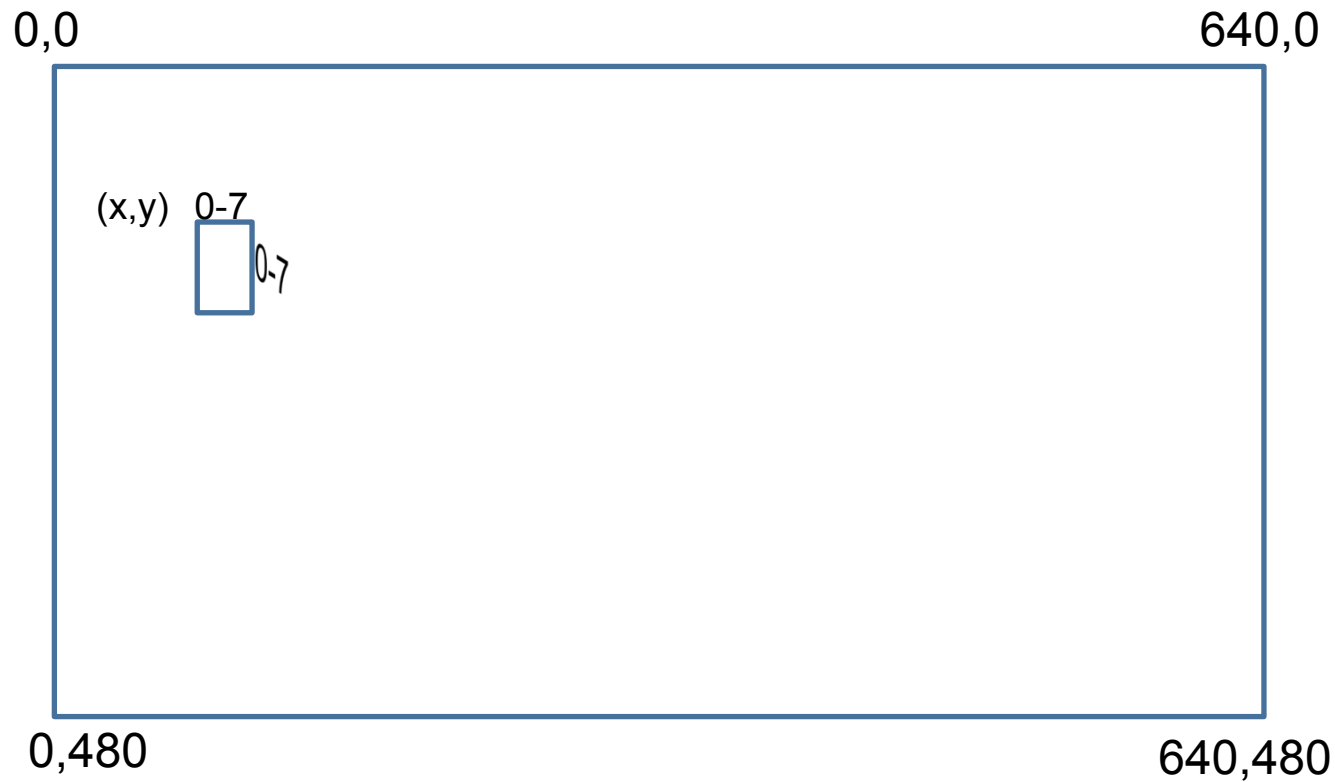
FB\_POINTER+0+640



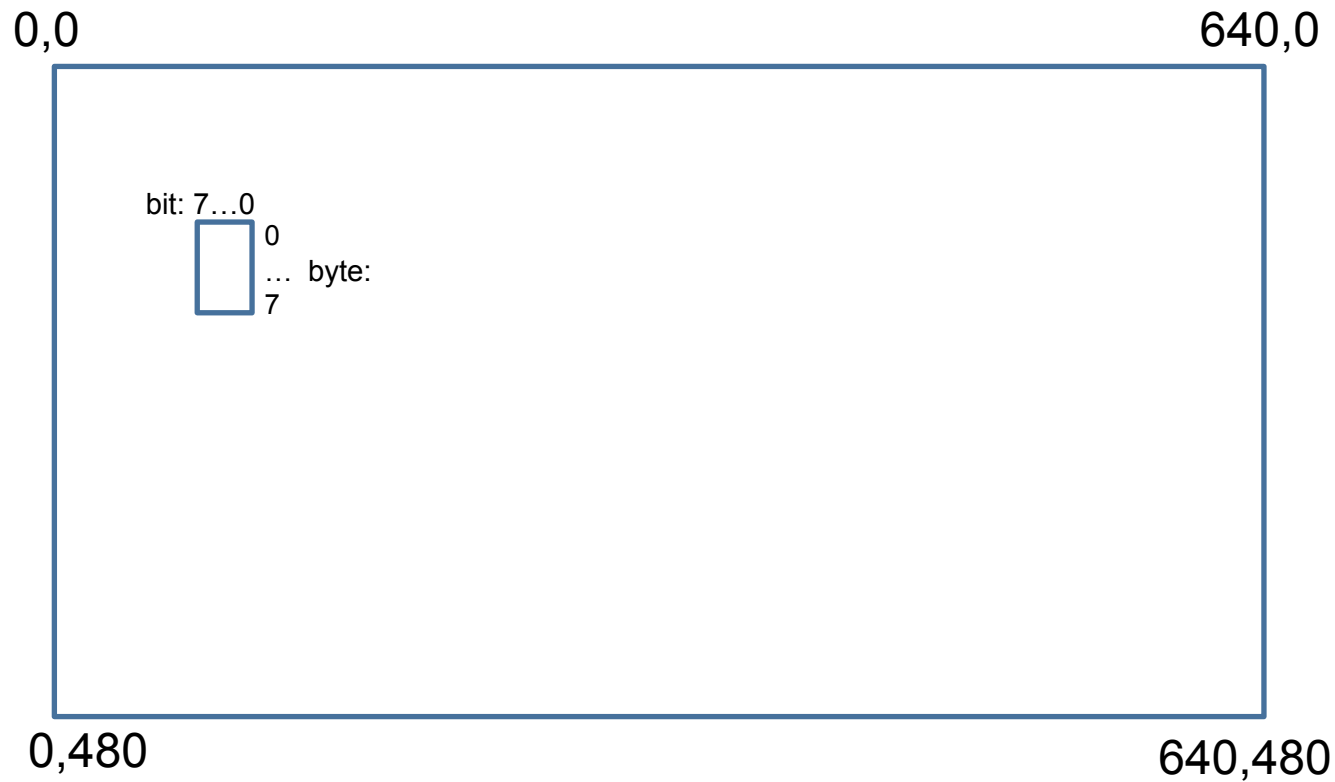
FB\_POINTER+0+640\*480

FB\_POINTER+640+640\*480

# CHAR LOCATION AND DIMENSIONS



# PIXEL LOCATION IN CHAR



# ONE LOOP, ADDING 1 TO X...



Char01-channel8 - functions - H-Line8.zip

we will need 2 loops(nested)



# FUNCTIONS FOR OUR OS

- *Turn on/off GPIO (gpio.s) (OK01, OK02)* 😊
- *Flash LED (flash.s) - for diagnostics* 😊
- *Wait (wait.s) (OK04)* 😊
- *Draw Pixel colour, position (Screen01)* 😊
- *Draw Character (colour, char, position)* 😊
- *Draw text (colour, position) (Screen03)* 😊
- *Process Input*

# PUT IT ALL TOGETHER

- We will program a GPIO header pin for input
  - Start a loop
  - read the GPIO
  - cmp to 0
  - if true, print the text “Closed”
  - else print the text “Open!”
- 
- too awesome to fit on one slide.

```

;set up GPIOs
GPIO_OFFSET = $200000
mov r10,BASE
orr r10,GPIO_OFFSET ;Base address of GPIO
ldr r8,[r10,#4] ;read function register for GPIO 10 - 19
bic r8,r8,#27 ;bit clear 27 = 9 * 3 = read access
str r8,[r10,#4];10 input
;set up input
mov r8,#1
lsl r8,#10 ;bit 10 to enable input GPIO10
mov r0,BASE
bl FB_Init
;r0 now contains address of screen
mov r7,r0 ;back-up a copy of the screen address

```

```

; Setup Characters
CHAR_X = 8
CHAR_Y = 8
loop$:
;read first block of GPIOs
ldr r9,[r10,#52] ;read gpios 0-31
tst r9,#1024 ; use tst to check bit 10
bne red ;if ==0
bl setup_chars
adr r2,Text ; R2 = Text Offset "Open"
DrawChars:
    mov r4,CHAR_Y ; R4 = Character Row Counter
    ldrb r5,[r2],1 ; R5 = Next Text Character
    add r5,r1,r5,ls1 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

```

```

red:
bl setup_chars
adr r2,Text2 ; R2 = Text Offset "Closed"
DrawChars2:
    mov r4,CHAR_Y ; R4 = Character Row Counter
    ldrb r5,[r2],1 ; R5 = Next Text Character
    add r5,r1,r5,ls1 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 DrawChars2 ; IF (Number Of Text Characters != 0) Continue To Print
Characters
cont:
;call timer (stop keybounce)
push {r0-r11}
mov r0,BASE
mov r1,$0A100
orr r1,$00020 ;TIMER_MICROSECONDS = 40,000
bl TIMER
pop {r0-r11}
b loop$

```

```

setup_chars:
; Setup Characters
mov r0,r7
mov r1,SCREEN_X
lsl r1,r1,5 ;32 rows
orr r1,#256 ; 256 columns
add r0,r1 ; Place Text At XY Position 256,32
adr r1,Font ; R1 = Characters
mov r3,#6 ; R3 = Number Of Text Characters To Print
bx lr
include "FBinit8.asm"
include "timer2_2Param.asm"
include "DrawChar.asm"
align 4
Text:
    db " Open!"
align 4
Text2:
    db "Closed"
align 4
Font:
    include "Font8x8.asm"

```

CHAR4-DIY\_KBD.zip

## Issues:

- Not optimised – most of the code in the DrawChars, DrawChars2 sections is the same.
- Should be pushed out to a function.

# THE LAB

- Start with Char01-channel8 - functions – XY.zip.
- Modify it so that it draws lines or a box
  - change the code in kernel7.asm
  - you will need to use loops, counters.
- Take your Pi with you.
  - And a micro-USB to USB cable.
  - And a USB to SD or micro-SD adapter
  - And an **HDMI to DVI-D** cable