COS10004 Computer Systems

Lecture 10.3 ARM Assembly – writing to screen in bare metal ARM Assembly

.section .data
text:
.ascii "Chris McCarthy\n\0"

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`

https://en.wikipedia.org/wiki/Magic_number_(programming)

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 $00000000,$FFFFFFFF ; 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

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.

align 16

Any power of 2

- 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
```

destination

expression

if only 2 parameters, difference is copied into 1st one.

- 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 ; 2B/3B/3B+
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      = $00040001 ; 0 (request), returns FB address
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

```
; send address of FB struct t
str r0,[r1] ;Mail Box Write
ldr r0,[FB_POINTER] ;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			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

- 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)
   mul r1,#32 ;(could also do lsl r1,#5) 2^5 = 32 = y coord
   orr r1,#256 ;x ordinate added (could be add in general)
   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 resister to a 1-byte pixel. What will go wrong?

ADDING WORDS TO BYTES

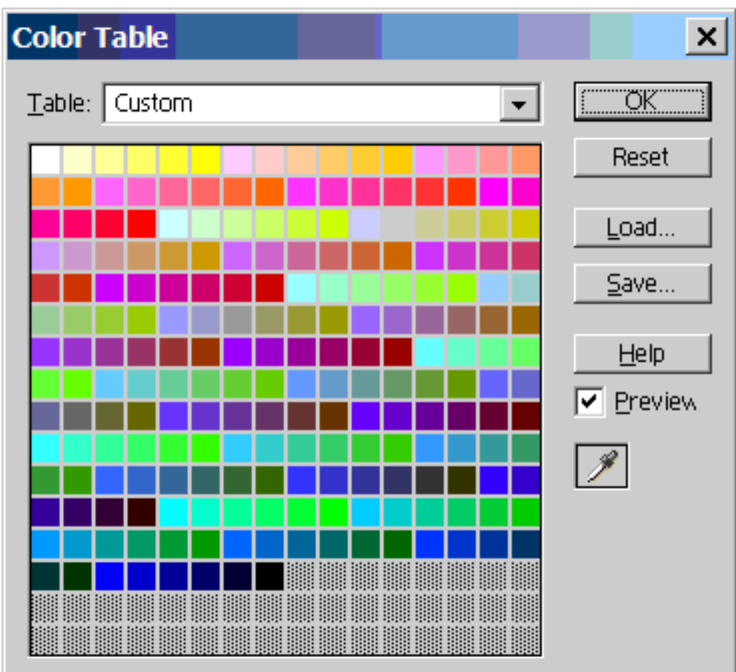
- Each word (4 bytes) written to the screen will overwrite adjacent bytes for neighboring pixels.
- Need a way to only write over 1 byte (8-bit colour) or 2 bytes (16-bit colour).
- Replace STR (word) with

  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

2nd parameter

```
mul r1,r1,#16
```

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

<u>The R Pi already has a POST routine:</u>

3 flashes: start.elf not found

4 flashes: start.elf not launched

7 flashes: kernel.img not found

Persistant  : 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))
 loopt1:
  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 loopt1 ;loop if LE (reaminingtime <=
delay)
 str r1,[r0,#32]  ;turn on LED
 ;re-use timer
 ldrd r6,r7,[r3,#4]
 mov r5,r6 ;store starttime (r5)
(=currenttime (r6))
 loopt2:
  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 loopt2 ;loop if LE (reaminingtime <=
delay)
pop {r0-r9}
 bx lr
```

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)*

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
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

THE LETTER A (8X16 DOTS)

```
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
```

```
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

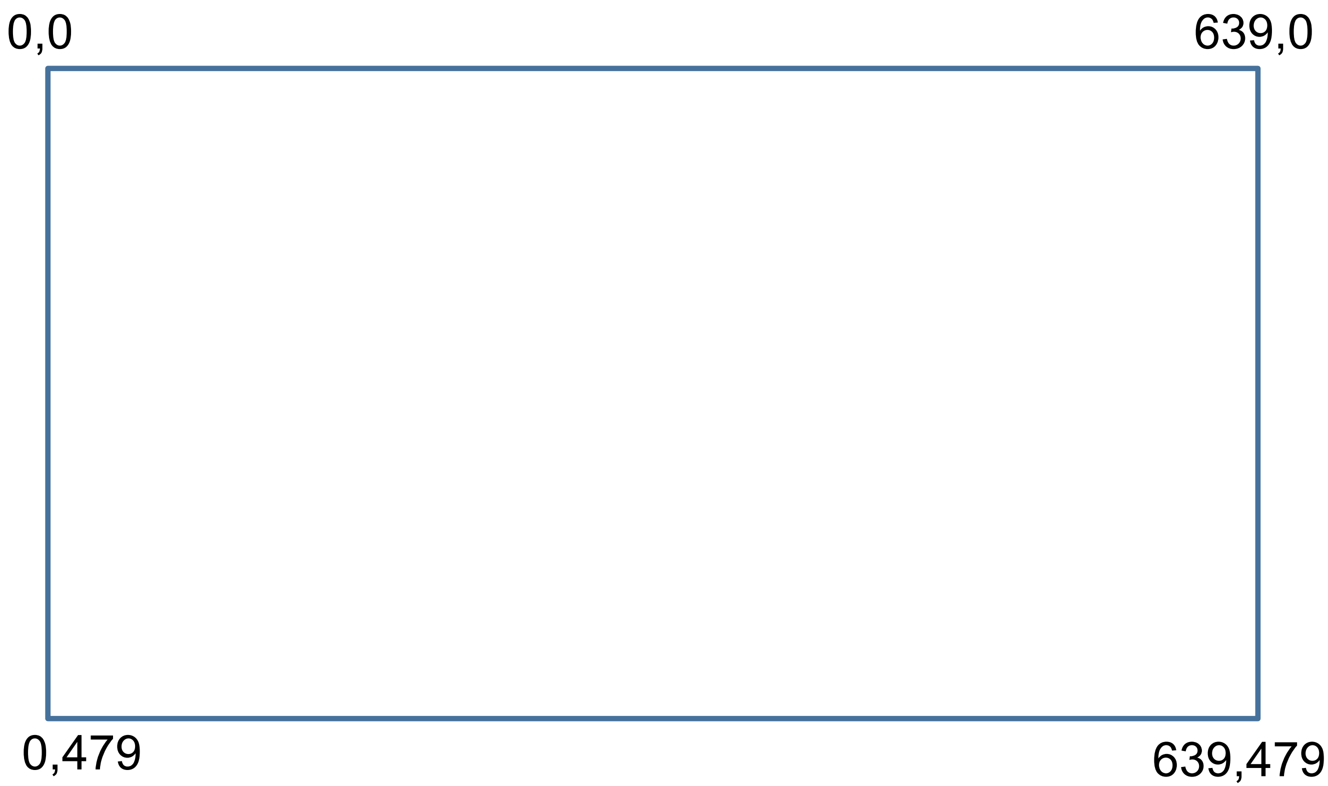
HOW DID WE PRINT THE CHARS?

- 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

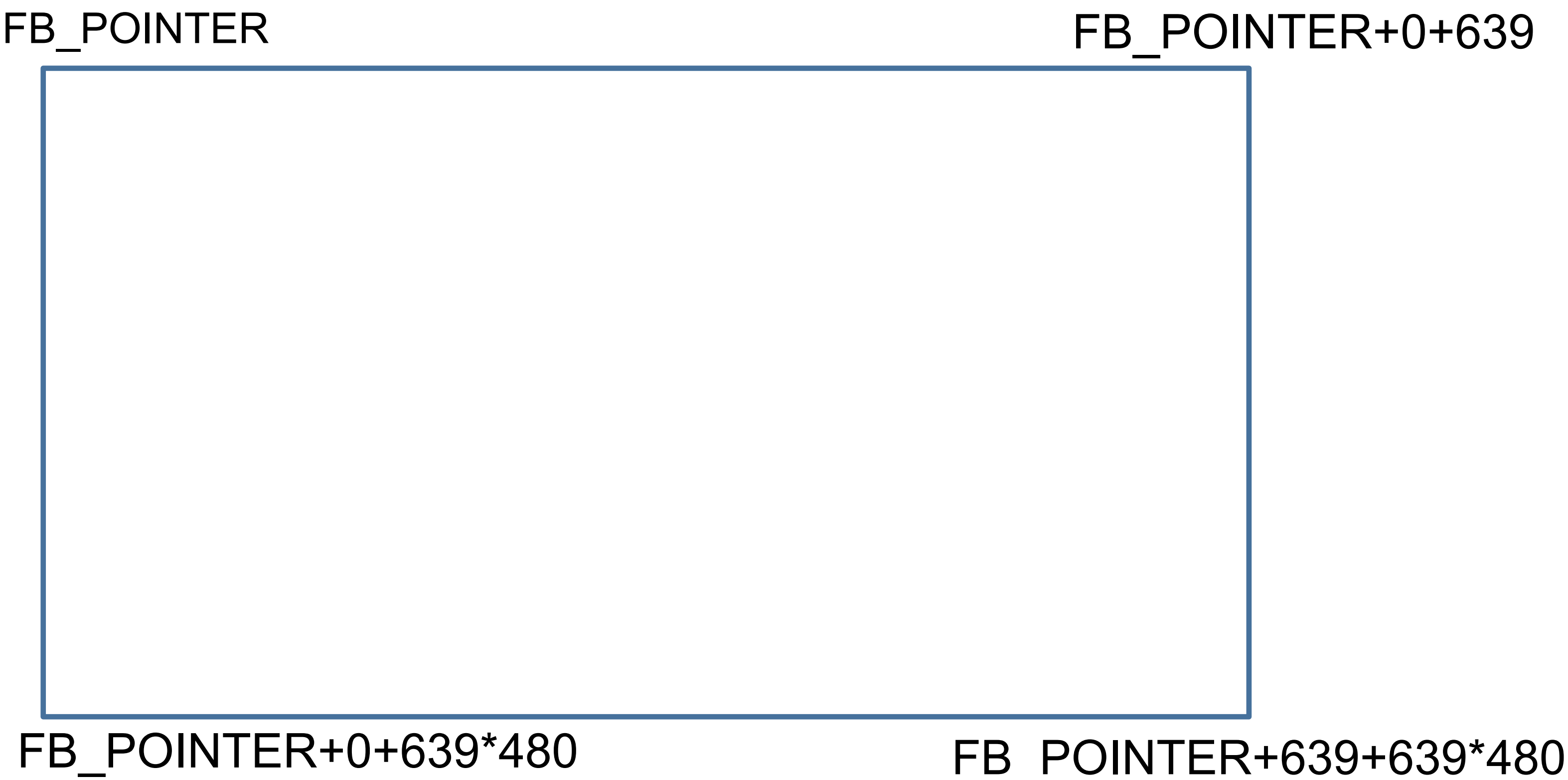
- Need to call DrawChar for each char in an array.
- Read the array N times (r3).
- Alternatively we could detect when we get to the null ‘\0’ character.
- Read/process other special characters
  - e.g. ‘\n’ means x=0 and y+=8

LOGICAL COORDINATES

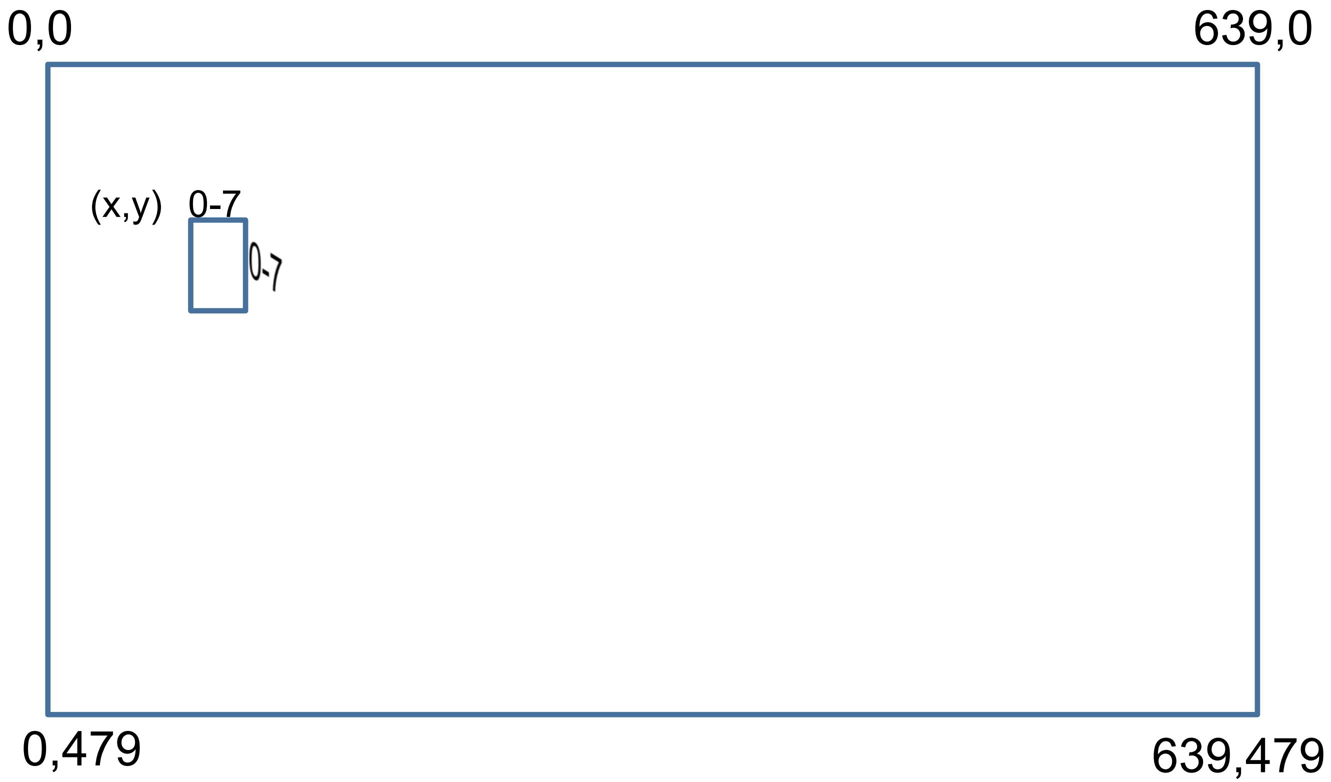


PHYSICAL ADDRESSES

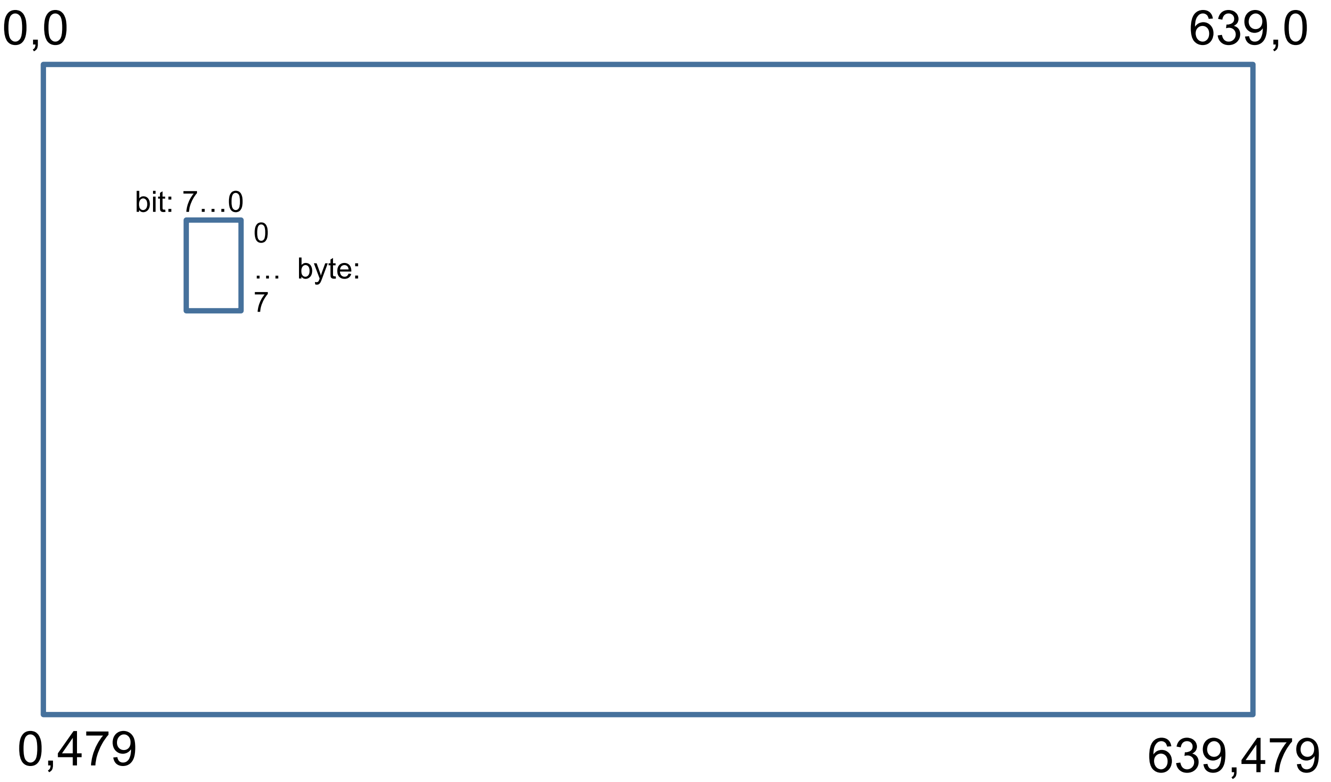
- Note: r0 = r7 = FB_POINTER



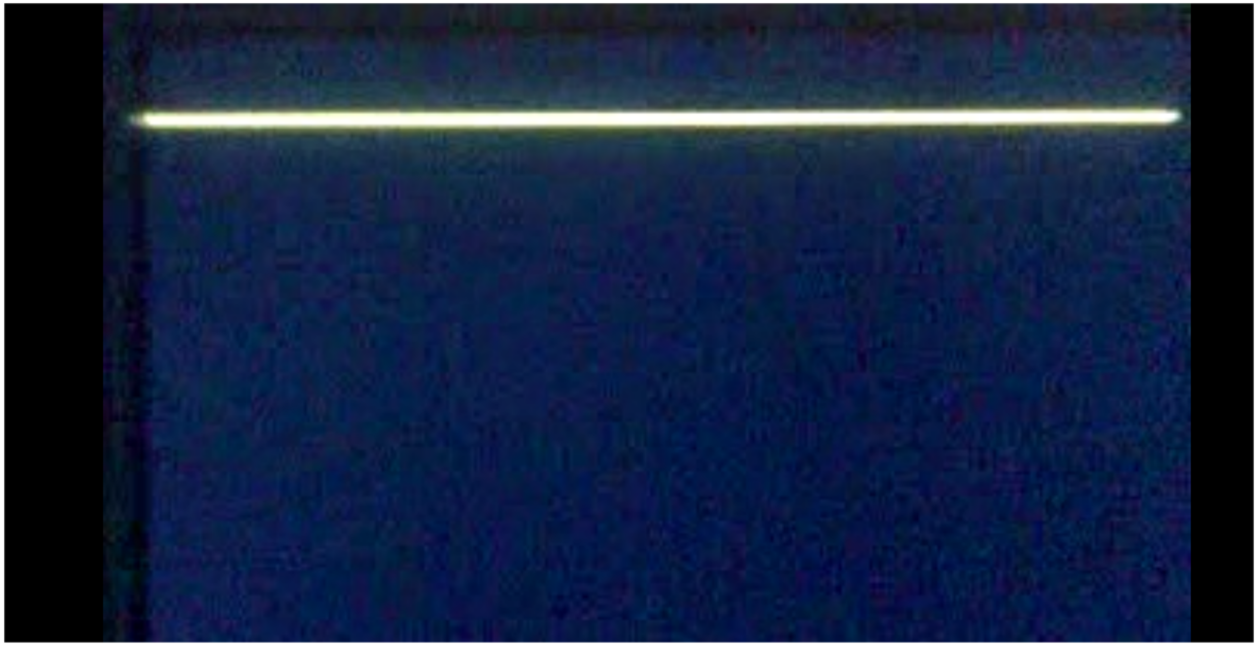
CHAR LOCATION AND DIMENSIONS



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

- In lab you 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 in slides.