359 Final

Review of C with pointers

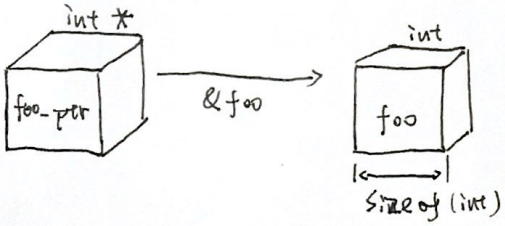
- A pointer is a memory address

`int foo;`

- 声明变量
- 占用内存（一个整数型 int，一般是32位或4个字节）
- 使用变量名 foo 来表示内存中的 bits.

`int *foo_ptr = &foo;`

- * means foo_ptr is a pointer
  - its value is a memory address
  - 不是一个整数
- & means "the address of"
  - &foo → the address of foo
  - not foo, but where foo exists in memory



- A variable is a box
  - it has a size (in bytes)
  - you put values in it
- The location of the box is its address
  - &foo
- foo_ptr is a box with a variable
  - it's type is "int *"
    - ↳ means it is a pointer / address for a box with an int.

习惯：

写两个指针

```
int *ptr_a;
int *ptr_b;
```

/ `int *ptr_a, *ptr_b;`

通过指针赋值

- suppose you want to assign a value to foo
  `foo = 42;`
- Do it with dereferencing
  `*foo_ptr = 42;`

↳ assign 42 to the box with the address of foo_ptr

Use * on LHS, RHS (LHS - write to box; RHS read from box)

Arrays

`int data[] = {45, 67, 89};`

- declare an array named "data"
- size is three
  - determined by initial assignment (3 values)
  - alternate `int data[3];` without initialization
- the variable "data" is a box with three int boxes in it
- 三个连续的盒子
- So array variable in C are just pointers to the first item.
- In most cases
  - `&data[0]` is the same thing as data.
- Decaying == `&data` == `&data[0]`

Pointer arithmetic

```
int *data_ptr = data;
printf("first element: %d\n", *(data_ptr++));
printf("second element: %d\n", *(data_ptr++));
printf("third element: %d\n", *(data_ptr++));
```

output: first element: 45
second element: 67
third element: 89

- note post-decrement operator
  - increment after use 先后++
- for each increment
  - actually added four to the pointer
  - because sizeof(int) == 4

Indexing 索引

`printf("%d\n", data[0]);`

- This prints
  45
- [] dereferences pointers with an offset to element value
- data[1] is the same as - *(data + 1)

```
int data[] = {45, 67, 89};
int *p = &data[1]
printf("%d\n", p[1])
```

prints 89

指针

4个内容：①指针的类型
②指针所指向的类型
③指针的值（指针所指向的内存区）
④指针本身所占据的内存区

例一：

```
(1) int *ptr;
(2) char *ptr;
(3) int **ptr;
(4) int (*ptr)[3];
(5) int *(*ptr)[4];
```

①指针的类型

（从语法的角度，只要把指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具的类型）

```
(1) int *ptr;      // 指针类型：int*
(2) char *ptr;     // 指针类型：char*
(3) int **ptr;     // 指针类型：int**
(4) int (*ptr)[3]; // 指针类型：int(*)[3]
(5) int *(*ptr)[4]; // 指针类型：int*(*)[4]
```

②指针所指向的类型

（当你通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。）（从语法上，只要把指针名和指针左边的*去掉，剩下的就是指针所指向的类型）

```
(1) int *ptr;      // int
(2) char *ptr;     // char
(3) int **ptr;     // int*
(4) int (*ptr)[3]; // int()[3]
(5) int *(*ptr)[4]; // int*()[4]
```

③指针的值 — 或者叫指针所指向的内存区或地址

指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。在32位程序里，所有类型的指针的值都是一个32位整数，因为32位程序里内存地址全都是32位长。指针所指向的内存区就是从指针的值所代表的那个内存地址开始，长度为sizeof(指针所指向的类型)的一块内存区。以后，我们说一个指针的值是XX，就相当于说该指针指向了以XX为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。指针所指向的类型是两个完全不同的概念。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指的内存区是不存在的，或说是无意义的。

④指针本身所占据的内存区

指针本身占了多大内存？你只要用函数sizeof(指针的类型)测一下就知道了。在32位平台里，指针本身占据了4个字节的长度。指针本身占据的内存这个概念在判断一个指针表达式是否是左值时很有用。

Structures and Unions

```
struct foo {
    size_t size;
    char name[64];
    int answer_to_ultimate_question;
    unsigned shoe_size;
};

struct foo my_foo;
my_foo.size = sizeof(struct foo);

struct *foo_ptr = &my_foo;
foo_ptr -> size = sizeof( struct foo);
```

- for access to members of struct
  - use "."
- if we have a pointer to a struct use
  - "->"
  - does pointer arithmetic to member

Multiple indirection

```
int a = 3;
int *b = &a;
int **c = &b;
int ***d = &c;
***d == c;
**d == *c == b;
***d == **c == *b == a == 3;
```

- & adds asterisks 加星号
  - increase pointer level
- * -> [] remove asterisks
  - decrease pointer level

Strings

```
char str[] = "I am the Walrus";
```

- C string is an array of char
  - terminated with null character. '\0000'
  - sentinel for end of string
- C string library
  - implemented on pointers to null-terminated arrays of char.

C library strlen() 长度.

```
int strlen( char *str ){
    int len = 0;
    while (*(str++)) ++len;
    return len;
}
```

```
int strlen( char *str) {
    int i;
    for(i=0; str[i]; ++i);
    return i;
}
```

- two ways to implement strlen()
  - first uses pointer arithmetic and dereferencing.
  - second uses array indexing.

C type casting

- Change one data type into another
  - int to float
  - float to int
  - ~~float~~ int to int*
  - void** to int*

```
#define REG_ADDRESS 0x3f000000
    int a=5;
    float b = 3.5;
    // int to float
    float f = (float)a;
    // float to int
    int i = (int)b;
    // convert int to pointer
    int *reg = (int*) REG_ADDRESS;
    // convert pointer types
    void *p = (void*) reg;
```

Memory mapped I/O

```
int *gpio;
gpio = base address of Pi GPIO
*gpio = 5;
gpio[5] = 3;
int val = gpio[6];
```

Allocating memory

- malloc allocates a block of memory of requested size, return pointer to that memory
  malloc 分配一个请求大小的内存块，返回指向该内存的指针.
- calloc
  - same but uses item size
- free returns the memory to OS when done.

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t numemb, size_t size);
```

Tut

Input / Output

- Output printf("text")
- Input scanf("%d", &n); //where n is a 变量.
- Concatenation
  - signed integer: printf("Age is %d", age)
  - Decimal: printf("Price is %f", price)
  - String: printf("Name is %s", fname)

Pointers

Pointers are used to map to a specific address of anther variable.

Review of basic electrical circuits.

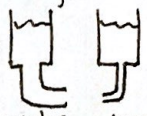
Voltage 电压

- electromotive force (EMF)
  electric pressure

- The electrical pressure that pushes current through an electrical system
  推动电流通过电气系统的电压力.

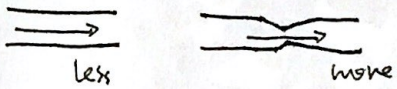
Current 电流

- The amount of charge flowing through a wire
  通过导线的电荷量.



Resistance 电阻

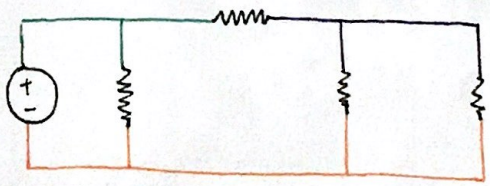
- How a wire resists (or allows) current to flow given voltage (EMF) 电流在给定电压下流动
- Inverse of conductivity
  - might be easier to understand this way
  - but ~~we~~ normally refer to resistance



Nodes

- lines in the schematic 原理图
- roughly equivalent to a wire
- same node if no other component in connected lines
- voltage same everywhere in a node

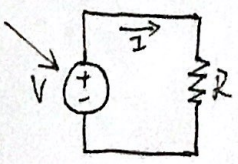
※ lines with same color are the same node※.



Ideal voltage source

↳ doesn't exists as a real, physical device

- represent with a circle
  ↳ +/- terminal (sometimes leave out '-'
- keep voltage constant across two terminals
- units - Volts (V)

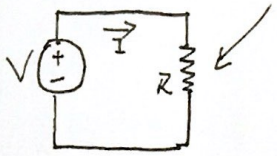


Current conventions 惯例

- electrical current is (usually) the flow of electrons.
- ~~electrons~~ electrons flow away from -ve terminal of voltage source
  - ↳ towards +ve
- But
  by convention we consider the current to be flow of positive charge
- Makes no difference to most analyses
- Summary
  - ↳ positive flow away from/out of plus terminal of voltage source
  - ↳ into negative terminal of voltage source.

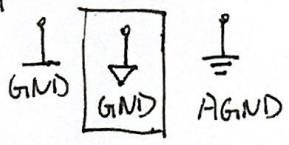
Resistor

- these can be real, physical devices
  - or integral part of real device
- resist the flow of current when voltage across them
- zigzag line 锯齿形线.
- units - Ohms (Ω)



Ground

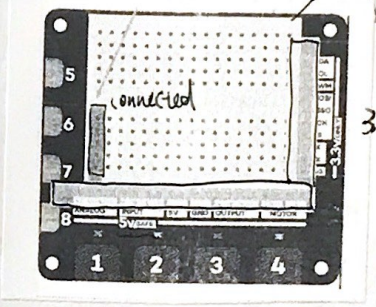
- Provides voltage reference
  - ↳ voltage must be across components
  - ↳ implies that when a node has a voltage it is across node to ground
- closes circuit



Physical connections to RPi

Wiring checklist

- Voltage level
  - 5V or 3.3
- Polarity of devices
  - ↳ power source
  - ↳ semi-conductors / ICs
  - ↳ electrolytic capacitors
- Output versus input versus bi-directional
- pull-up resistors
  - ↳ and pull-down



Separate voltages on Explorer Hat

- Two parts
  - 5V
  - 3.3V
- RPi use 3.3V
  - ↳ connecting 5V to RPi 3.3V can damage RPi
- 5V side has buffers to convert 5V to 3.3V logic

How to make connections

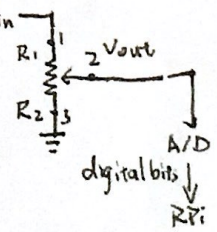
- Make connections with jumper wire
  - forms nodes in circuit
- Female pins connecting to RPi on bottom and right
  - labeled 在底部和右侧连接到RPi的引脚
- set of 5 female pins on breadboard are connected

Potentiometer 电位计

↳ 电位计构成分压器
Potentiometer forms a voltage divider

↳ Sweeping the middle contact varies V_out between V_in and 0V

↳ A/D connected via Explorer Hat
- allows RPi to read V_out



Analog-to Digital on Explorer

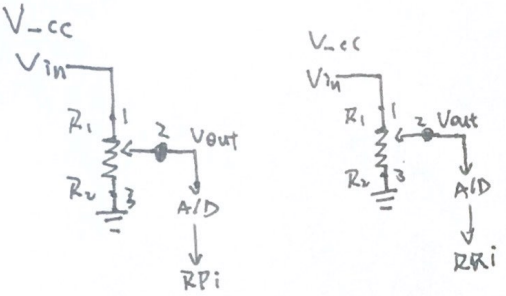
- 4 A/D inputs on Explorer Hat Pro.
- 4 pins on left-hand side of 5V section
- Labeled 1.2.3.4
- Connected to analog inputs of an ADS1015 IC

Adafruit joystick 操纵杆

↳ Has two potentiometers
- x axis / y axis

↳ one pushbutton

inferred schematic (x, y)



x axis　　y axis

Pushbutton:
- Connect to digital input
- Circuit board says
  - "select shorts to ~~group~~ ground when pressed"

Floating inputs 浮动输入

- Inputs that are not connected to anything are said to 没有连接到任何东西的输入被称为
  - float
- That is, then have no specific voltage
  - Input is arbitrary
- The solution is to use either
  - pull-up resistor 上拉电阻
  - pull-down resistor 下拉电阻

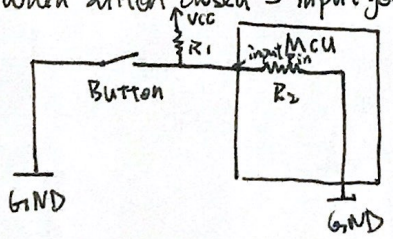
Pull-up resistors

↳ Pull-up connects floating input to logic high through the pull-up resistor 上拉通过上拉电阻将浮动输入连接到逻辑高电平.

↳ Note the ## Thevenin equivalent MCU ???

↳ when button open - voltage divider

↳ when button closed - input goes to 0V - logic low



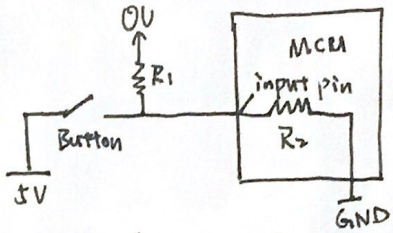
Pull-down

↳ Connect floating input to ground through pull-down resistor

↳ Need to put other end of button to logic high

↳ when button open - 0V, logic low

↳ when button closes - 5V, logic high



!. Uses pull-down for digital inputs
so need other end of push button to go to 5V.

Physical connections to RPi ⑤

ADXL 345 - Accelerometer 加速度计

↳ 3 axis accelerometer

↳ Adafruit mounts it on a breakout board
- provides essential peripheral circuitry 提供必要的外围电路
- pins for breadboard.

Breakout features

↳ 5V / 3.3V operation
- ADXL 345 is 3.3V
- added voltage regulateor

↳ x

Interface to ADXL345

- Two ways
  - I²C — Inter-Integrated circuit / I-squared C or I-2-C
  - SPI — Serial Peripheral Interface bus.
- Same functionality
- Both in wide use
- Both supported will by RPi

~~Well~~

很下面 ->

Memory mapped IO

内存映射输入/输出

CPU怎样访问外设存储空间？

一种方式，CPU用单独的指令访问。这个单独的指令需要一个要操作的外设存储空间的地址，这就是I/O端口（I/O port）

另一种方式，CPU不用单独指令访问外设存储空间，用通常的访问存储器的方式访问外设存储空间。这就是I/O存储器（memory-mapped）

memory比port好在它不受固定大小的，只受系统配置的常规存储器大小和系统地址空间大小的限制。

Memory versus IO

- conceptually
  - registers/memory are the same thing
- In practice
  - flip flop registers not efficient for main memory
    - too many transistors - space on silicon
    - too much power
- but, flip-flop register can connect to other devices.

Key concept 关键概念

- use combinational logic to decode address bus
  - some addresses connect to memory
  - other connect to registers
- registers connect to electronics that interfaces with the physical world
  - digital io pins
  - uarts
  - dma ...

I/O read/write from pointer ref

- write
  - ↳ write contents of variable (CPU register) to memory or output register
- read
  - ↳ read contents of memory or input register into variable (CPU register)

GPIO

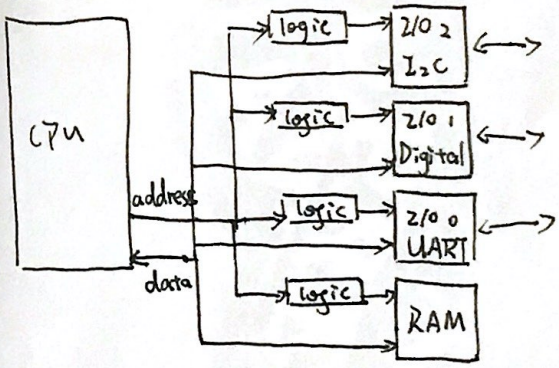
General Purpose Input/Output

通用型之输入/出

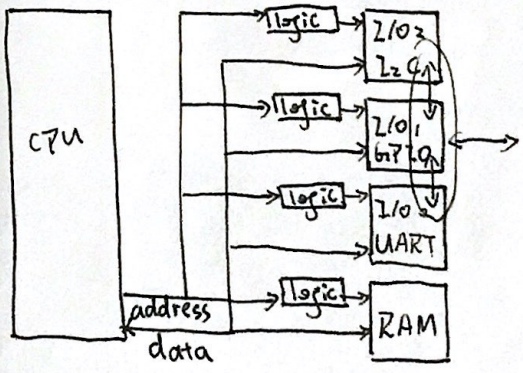
※在嵌入式系统中，经常需要控制许多结构简单的外部设备或者电路，这些设备有的需要通过CPU控制，有的需要CPU提供输入信号。对设备的控制，使用传统的串口或者并口就显得比较复杂，所以，在嵌入式微处理上通常提供一种"通用~~由~~可编程I/O端口"，也就是GPIO port，计算机与外界通讯称为IO

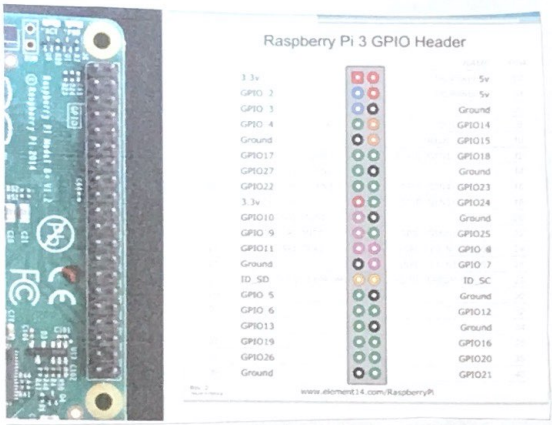
- Famous with microcontrollers
  - ↳ Provide extended functions without changing circuitry
- Has 54 lines
  - Only a subset is available for the Rpi
    - 26 lines (40 on Rpi 3)

Memory Mapped IO



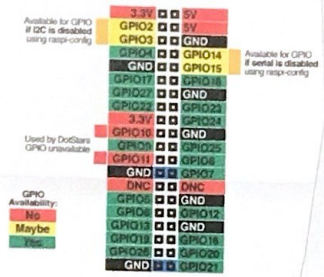
Memory Mapped IO with GPIO





RPi GPIO pins

- The Rpi GPIO has 54pins
  - not all brought out for users
  - header shows what you have to work with
  - green ones for general use
  - others have specific functions
- Since they are not plug-and-play, some work is need to initialize, write, and read
- Each of these pins can have up to 6 functions
  - For example function 1 is write & 0 is read
    - From core's point of view



GPIO Registers

Address	Field Name	Description	Size	Read/ Write	
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W	Function Select
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W	
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W	
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W	
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W	
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W	
0x 7E20 0018		Reserved			
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W	Set (Write 1)
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W	
0x 7E20 0024		Reserved			
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W	Clear (Write 0)
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W	
0x 7E20 0030		Reserved			
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R	Level (Read)
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R	
0x 7E20 003C		Reserved			

More info refer to: BCM2835 ARM Peripherals (available on BB)

GPIO Registers

- There are more registers
- Replace 7E in the virtual address by 3f to obtain the physical address.

Function Select Registers

0x3f200014	GPFSEL5	For pins 50 to 53
0x3f200010	GPFSEL4	For pins 40 to 49
0x3f20000C	GPFSEL3	For pins 30 to 39
0x3f200008	GPFSEL2	For pins 20 to 29
0x3f200004	GPFSEL1	For pins 10 to 19
→ 0x3f200000	GPFSEL0	For pins 0 to 9

Base GPIO address

有54个管脚

Accessing GPIO Registers 访问GPIO寄存器

- GPIO uses "pseudo" memory-mapped I/O
- Accessed as if accessing memory locations

```
ldr   r0, =0x20200000  // Address of GPFSEL0
// To read GPFSEL0
ldr r1, [r0]
// To write GPFSEL0
str r1, [r0]
```

Pin Offset from GPIO Base Address

- A register corresponds to many GPIO pins
- For example GPFSEL0 对应 pin 0 到 9
- Each pin (0 to 9) is controlled by 3 bits in GPFSEL0
- For example, bits 0-2 set the function for GPIO pin 0
- Each pin can be given a function (from up to 6)
- In our case, it is mostly input or output functions.

Inside GPFSEL0

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL9	FSEL9 - Function Select 9 000 = GPIO Pin 9 is an input 001 = GPIO Pin 9 is an output 100 = GPIO Pin 9 takes alternate function 0 101 = GPIO Pin 9 takes alternate function 1 110 = GPIO Pin 9 takes alternate function 2 111 = GPIO Pin 9 takes alternate function 3 011 = GPIO Pin 9 takes alternate function 4 010 = GPIO Pin 9 takes alternate function 5	R/W	0
26-24	FSEL8	FSEL8 - Function Select 8	R/W	0
23-21	FSEL7	FSEL7 - Function Select 7	R/W	0
20-18	FSEL6	FSEL6 - Function Select 6	R/W	0
17-15	FSEL5	FSEL5 - Function Select 5	R/W	0
14-12	FSEL4	FSEL4 - Function Select 4	R/W	0
11-9	FSEL3	FSEL3 - Function Select 3	R/W	0
8-6	FSEL2	FSEL2 - Function Select 2	R/W	0
5-3	FSEL1	FSEL1 - Function Select 1	R/W	0
2-0	FSEL0	FSEL0 - Function Select 0	R/W	0

※寄存器 GPFSEL0 - GPFSEL5 --> 功能寄存器
指定管脚为输入/出口，每3位决定一个管脚.

Calculating the Pin Offset

- 若 pin# is single-digit
  offset = address for GPFSEL0
  (base GPIO address, 0x3f200000)
- 若 pin# 大于9 (more than one digit)
  Offset = base GPIO address + 4 × (pin# div 10)
  ↳ Address of GPFSEL1 to GPFSEL5

※. Example (pins 10 to 19):
Offset = 0x3f200000 + 4×(1) = 0x3f200004
↳ address for GPFSEL1

Getting to Appropriate Pins

- Given some pin #
- Let GPFSEL{n} be the register that controls this pin# (ie., n = pin# div 10)
- Let d be the least significant digit of pin#
  ↳ d = pin# mod 10 余数.
- The 3 bits of GPFSEL{n} that select select pin#'s function start at d*3

Example

Given Pin 32

- Register GPFSEL3, offset 0x3f20000c
  ↳ = 4x(32 div 10) + 0x3f200000
- Least significant bit of pin# is 2 = 32 mod 10
- The three bits of GPFSEL3 that control pin 32 start at bit 3*2=6
- ∴ bits 6-8 (6,7,8) select the function of pin32

GPIOFSEL{n}

30-31	27-29	24-26	21-23	18-20	15-17	12-14	9-11	6-8	3-5	0-2
reserved	pin {n}9	pin {n}8	pin {n}7	pin {n}6	pin {n}5	pin {n}4	pin {n}3	pin {n}2	pin {n}1	pin {n}0

Setting the bits in GPFSEL{n}

- Clear the appropriate bits in GPFSEL{n}
  ↳ Same as setting up for input
- Set the appropriate bits in GPFSEL{n}

Bit Bashing

x is a logical variable
  < also a bit
  also a GPIO pin state

x.0 = 0	clear bit
x.1 = x	keep bit the same
x+0 = x	keep bit the same
x+1 = 1	set bit

↳ Back to setting the bits in GPFSEL{n}

① Clear the appropriate bits in GPFSEL{n}
  ↳ same as setting up for input
  ↳ Clear means AND with 0 =D-
  ↳ must AND the bits for GPIO pin with 0
  ↳ But must preserve the other bits 保留
    . with AND, preserve with 1
  ↳ Solution - AND with integer that has 0 where I want to clear, and 1 everywhere else

② Set the appropriate bits in GPSFEL{n}
  ↳ assume starting with the bits as 0
    • should previously have cleared the bits
  ↳ setting appropriate bits mean OR with bit pattern =D
  ↳ but must preserve the other bits
    • with OR, preserve with 0
  ↳ solution - OR with integer that has bit pattern where I want it, 0 everywhere else

Clear the appropriate bits in GPFSEL{n}

```
ldr r0, =<address of GPFSEL{n}>
ldr r1, [r0]
mov r2, #7 (b0111)
lsl r2, <index of 1st bit of the corresponding 3>
bic r1, r2
str r1, [r0]
```

Set the appropriate bits in GPSFEL{n}

```
r0 = address of GPFSEL{n}
r1 = copy of GPFSEL{n}
mov r2, <function code (3 bits)>
lsl r2, <index of 1st bit of the corresponding 3>
    Example: pin#1, 1st bit is 3
    lsl r2, #3
orr r1, r2
str r1, [r0]
```

Example - Setting pin 3 to output

```
ldr r0, =0x20200000 // address for GPFSEL0
ldr r1, [r0]// copy GPFSEL0 into r1
mov r2, #7 // (b0111)
lsl r2, #9 // index of 1st bit for pin3
// r2 = 0 111 000 000 000
bic r1, r2 // clear pin3 bits
mov r3 , #1 // output function code
lsl  r3, #9 // r3 = 0 001 000 000 000
orr r1, r3 // set pin3 function in r1
str r1, [r0] // write back to GPFSEL0
```

Using C

```
#define GPIO_BASE   0x3f200000                    // 7 = b111

unsigned *gpio = mmap( ...);     // GPIO base in virtual memory

// GPIO setup macros. Always use INP_GPIO(x)
//     before using OUT_GPIO(x) or SET_GPIO_ALT(x,y)
#define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7<<(((g)%10)*3))

#define OUT_GPIO(g) *(gpio+((g)/10)) |= (1<<(((g)%10)*3))
```

b001 - means configure for output

found on the internet in many many places
original source unknown

Using C alternate

```
#define GPIO_BASE   0x3f200000

unsigned *gpio = mmap( ...);     // GPIO base in virtual memory

// GPIO setup macros. Always use INP_GPIO(x)
//     before using OUT_GPIO(x) or SET_GPIO_ALT(x,y)
#define INP_GPIO(g) gpio[(g)/10] = gpio[(g)/10] & ~(7<<(((g)%10)*3))

#define OUT_GPIO(g) gpio[(g)/10] = gpio[(g)/10] | (1<<(((g)%10)*3))
```

Preforming I/O

↳ Once a pin function is set, then pin can be used for I/O
↳ To write to a pin, use GPSET{0,1} or GPCLR{0,1}
↳ To read from a pin, use GPLEV{0,1}

GPIO Pin Output Set Registers (GPSETn)

SYNOPSIS The output set registers are used to set a GPIO pin. The SET(n) field defines the respective GPIO pin to set, writing a "0" to the field has no effect. If the GPIO pin is being used as in input (by default) then the value in the SET(n) field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations

Bit(s)	Field Name	Description	Type	Reset
31-0	SETn (n=0..31)	0 = No effect 1 = Set GPIO pin *n*	R/W	0

Table 6-8 – GPIO Output Set Register 0

Dividing a pin# by 32 determines which register must be used

Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	SETn (n=32..53)	0 = No effect 1 = Set GPIO pin *n*	R/W	0

Table 6-9 – GPIO Output Set Register 1 From: BCM2835 ARM Peripherals

Setting (Writing to) a Pin

If writing 1, then use GPSET[n], n=0..1
- gpioAddr + 28 + [4*n]

If writing 0, use GPCLR[n]
- A write of 0 to GPSET[n] is ignored
- gpioAddr + 40 + [4*n]

GPIO Pin Output Clear Registers (GPCLRn)

SYNOPSIS The output clear registers) are used to clear a GPIO pin. The CLR(n) field defines the respective GPIO pin to clear, writing a "0" to the field has no effect. If the GPIO pin is being used as in input (by default) then the value in the CLR(n) field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations.

Bit(s)	Field Name	Description	Type	Reset
31-0	CLRn (n=0..31)	0 = No effect 1 = Clear GPIO pin *n*	R/W	0

Table 6-10 – GPIO Output Clear Register 0

Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	CLRn (n=32..53)	0 = No effect 1 = Set GPIO pin *n*	R/W	0

Table 6-11 – GPIO Output Clear Register 1 From: BCM2835 ARM Peripherals

Example, writing to pin#3
- r1 = value to write {0,1}

```
r0 = #3    // pin #
ldr r2, = 0x20200000  // base GPIO reg
mov r2, #1
lsl r2, r0  // align bit for pin#3
teq r1, #0
streq r2, [r2, #40]  // GPCLR0
strne r2, [r2, #28]  // GPSET0
```

Using C (for pins 0-31)

```
unsigned int *gpio = mmap(...);
# define GPSET0  7
# define GPCLR0  10
...
if (want to set the pin)
    gpio[GPSET0] = 1 << pinNumber;
else
    gpio[GPCLR0] = 1 << pinNumber;
```

Reading a Pin

↳ Register GPLEVn is used to read a bit from a pin

↳ In the RPi, only GPLEV0 is used
- Bit n corresponds to the value of pin n

↳ Address of GPLEV0 is 0x3f200000 + 52

Inside GPLEV{n}

GPIO Pin Level Registers (GPLEVn)

SYNOPSIS The pin level registers return the actual value of the pin. The LEV(n) field gives the value of the respective GPIO pin.

Bit(s)	Field Name	Description	Type	Reset
31-0	LEVn (n=0..31)	0 = GPIO pin *n* is low 0 = GPIO pin *n* is high	R/W	0

Table 6-12 – GPIO Level Register 0

Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	LEVn (n=32..53)	0 = GPIO pin *n* is low 0 = GPIO pin *n* is high	R/W	0

Table 6-13 – GPIO Level Register 1

Example, Reading from pin #3

```
r0 = #3   // pin#
ldr r2, = 0x20200000 // base GPIO reg
ldr r1, [r2, #52]  // GPLEV0
mov r3, #1
lsl r3, r0   // align pin 3 bit
and r1, r3   // mask everything else
teq r1, #0
moveq r4, #0   // return 0
movne r4, #1   // return 1
```

Using C (for pins 0-31)

```
unsigned int *gpio = mmap(...);
# define GPLEV0  13
...
int v;
// v = pin 3 value
v = (gpio[GPLEV0] >> 3) & 1;
```

Other GPIO Functions
- Clock signal
- UART
- JTAG
- I²C
- Etc ...

UART

Protocal

→ Universal Asynchronous Receiver / Transmitter Protocal

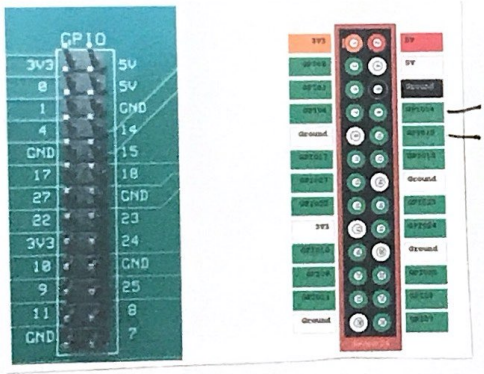
通用异步收发传输器

※ 它将要传输的资料在串行通信和并行通信之间加以转换

※ 作为把并行输入信号转成串行输出信号的芯片，UART通常被集成于其他通讯接口的连接上.

Rpi GPIO

↳ General Purpose I/O
↳ Programmable
↳ 26 header pins
↳ Provide UART and 5V power
↳ Provide others two eg. CSI (camera serial interface 相机串行接口) and DSI (Display serial interface 显示串行接口)
· Noe plug and play 不必即插即用



26

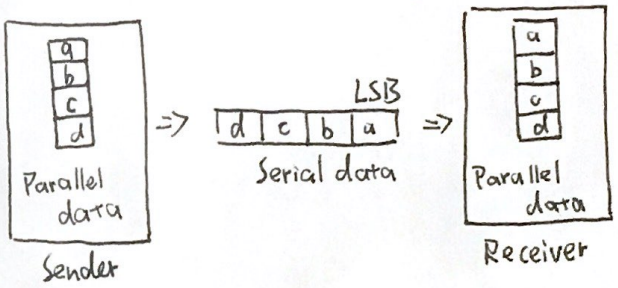
JTAG Interface

↳ Joint Test Action Group 联合测试行动组
※ 是一种国际标准测试协议 (IEEE 1149.1兼容) 主要用于芯片内部测试.

UART

↳ Low speed serial I/O protocol
↳ USB and Firewire are much faster
· But more complex
↳ Ubiquitous and highly in use 普遍和高度使用
↳ Especially useful when low-speed transfer is sufficient
↳ Inexpensive

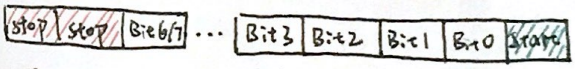
↳ Converts between parallel signal and serial signal 在并行信号和串行信号之间进行转换.



↳ Connects a core to an I/O Device
↳ 16650 UART is a famous IC Integrated Circuit Chip 芯片

UART Protocal 协议

· 1 means no data (legacy from telegraphy)
· Start bit is signal 0



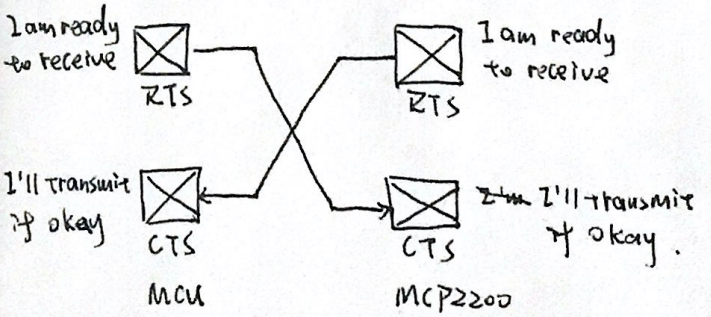
· One or more stop bits are signal 1
· Timing is important to guarantee proper communication
※ UART作为异步串口通信协议的一种，工作原理是将传输数据的每个字符一位接一位地传输.

UART Functions

· TX (Transmit to I/O device)
· RX (Receive from I/O device)

RTS & CTS

· CTS (Can TranSmit) impacts the transmitter: it does not send new bits if CTS line is 0
· RTS (Ready To Receive) impacts the receiver



RPi AUX Peripherals 外围设备

- RPi has three auxiliary peripherals:
  - ↳ Mini UART and 2 SPI masters
    (Serial Interface Peripheral 串行外设接口)
  - ↳ All three share the same area of registers
  - ↳ They share a common interrupt
  - ↳ They are controlled by the same register (ENB)

AUXENB Register

AUXENB Register (0x7E21 5004)

Synopsis: The AUXENB register is used to enable the three modules, UART, SPI1, SPI2.

Bit(s)	Field Name	Description	Type	Reset
31:3		Reserved, write zero, read as don't care		
2	SPI2 enable	If set the SPI 2 module is enabled. If clear the SPI 2 module is disabled. That also disables any SPI 2 module register access	R/W	0
1	SPI 1 enable	If set the SPI 1 module is enabled. If clear the SPI 1 module is disabled. That also disables any SPI 1 module register access	R/W	0
0	Mini UART enable	If set the mini UART is enabled. The UART will immediately start receiving data, especially if the UART1_RX line is *low*. If clear the mini UART is disabled. That also disables any mini UART register access	R/W	0

※ RPi Mini UART

↳ Is a low throughput UART intended to be used as a console
→ does not have all UART functions

↳ Some of its features:
- 7 or 8 bit operation
- 1 start bit and 1 stop bit
- No parity bit
- 8-symbol deep FIFO to receive and transmit 先进先出?
- RTS and CTS can be controlled by S/W
- 16550-like registers (but not 16550 compatible)

Baud Rate 波特率

↳ Measurement of asynchronous transmission speed
↳ Represents the number of bits sent (including control bits: start, stop and parity)
↳ Measures bits per second
↳ Measured in baud
↳ In digital system, 1 baud = 1 bit/sec

※ 波特率即调制速率，指的是有效数据信号调制载波的速率，即单位时间内载波调制状态变化的次数

Mini UART Baud Rate

↳ Baud rate = $\frac{\text{system\_clock\_frequency}}{8 * \text{baudrate\_register} + 1}$

↳ Baudrate_register = $\left(\frac{\text{system\_clock\_frequency}}{\text{baudrate} / 8}\right) - 1$

Example

Clock = 250 Mhz
Baudrate_register = 0
↳ Baud rate = 31.25 Mega baud (mbps)

Mini UART Essential Registers

Register	Function	Access	Address (RPi)
ENB	UART Enable	R/W	0x20215004
IO	Receive Data Send Data	R W	0x20215040
IER	Interrupt Enable	R/W	0x20215044
IIR FCR	Interrupt Identification FIFO Control	R W	0x20215048
LCR	Line Control	R/W	0x2021504C
MCR	Modem Control	R/W	0x20215050
LSR	Line Status	R	0x20215054
MSR	Modem Status	R	0x20215058
SCRATCH	Scratch Pad	R/W	0x2021505C

Other Registers

Register	Function	Access	Address
CONTROL	Extra control features	R/W	0x20215060
STATUS	Info about internal status of mini UART	R	0x20215064
BAUD	Access to baud counter	R/W	0x20215068

Initializing UART

① ENB = 0x00000001 ; enable UART
② IER = 0x00000000 ; enable interrupts
③ CONTROL = 0x00000000 ; [0] receive enable
[1] send enable
④ LCR = 0x00000001 ; [0] = 1 => 8-bit communication
⑤ MCR = 0x00000000 ; [1] = 0 => Request to (RTS) Transmit Signal is high!
⑥ FCR(IIR) = 0x00000006 ; 11000110 , [1] reset RCV FIFO
[2] reset snd FIFO
[7:6] enable FIFO
⑦ BAUD = 270 ; system clock freq / baud rate / 8 - 1
= 250M / 115.2k - 1 ≈ 270
⑧ Set GPIO line 14 for transmission (TXD)
⑨ set GPIO line 15 for receiving (RXD)

⑩ Disable GPIO pull-up/down
⑪ Wait (150 cycles)
⑫ Assert clock lines (14 & 15)
⑬ Wait (150 cycles)
⑭ Clear clock lines
⑮ Enable bits 0 (recv) and 1 (send) in CONTROL.

AUX_MU_LSR_REG Register (0x7E21 5054)

SYNOPSIS The AUX_MU_LSR_REG register shows the data status.

Bit(s)	Field Name	Description	Type	Reset
31:8		Reserved, write zero, read as don't care		
7		Reserved, write zero, read as don't care *This bit has a function in a 16550 compatible UART but is ignored here*		0
6	Transmitter idle	This bit is set if the transmit FIFO is empty and the transmitter is idle. (Finished shifting out the last bit).	R	1
5	Transmitter empty	This bit is set if the transmit FIFO can accept at least one byte.	R	0
4:2		Reserved, write zero, read as don't care *Some of these bits have functions in a 16550 compatible UART but are ignored here*		0
1	Receiver Overrun	This bit is set if there was a receiver overrun. That is: one or more characters arrived whilst the receive FIFO was full. The newly arrived charters have been discarded. This bit is cleared each time this register is read. To do a non-destructive read of this overrun bit use the Mini Uart Extra Status register.	R/C	0
0	Data ready	This bit is set if the receive FIFO holds at least 1 symbol.	R	0

Writing a Character

1. Test the LSR register if the transmitter is empty
   - can write if bit #5 in LSR is set
   - wait in a loop until bit #5 is set
2. Write one byte to the IO register

※. Recall
- LSR is 0x7E215054
- IO 0x7E215040

Reading a Character

1. Test the LSR register if there is something to read
   - can read if bit#0 in LSR is set
   - wait in a loop until bit#0 is set
2. Read one byte from the IO register.

Reading a Memory Mapped Registe

```
.equ AUX_MU_LSR_REG, 0x7E215054
ldr r2, =AUX_MU_LSR_REG
ldr r1, [r2]
tst r1 = #0x20 // 0x20 = 100000
```

Writing a Memory Mapped Register

```
.equ AUX_MU_IO_REG, 0x7E215040
ldr r2, =AUX_MU_IO_REG
ldr r0, 0x41 // ASCII A
str r0, [r2] // write ASCII A
```

Example

```
bl InitUART // initialize UART
mov r0, #0x41 // ASCII code for A
loop:
bl PutCharUART // write char
add r0, #1 // next char
cmp r0, #0x5A // ASCII for Z
ble loop // loop until Z.
```

```
void putCharUart (register charc ){
    while ((aux[AUX_MU_LSR]& 0x20)==0);
      // loop until space in mini uart fifo
    aux [AUX_MU_IO] =c ;
}
char getCharUart () {
    while ((aux [AUX_MU_LSR] & 0x01)==0);
      // loop until something in input fifo
    return (char) (aux [AUX_MU_IO] & 0x7f );
}
```

Running a UART Terminal

screen /dev/ttyusb0 225200.

Exceptions Interrupts 异常中断

Basic Concepts

↳ An exception is a system event or condition that requires action by privileged software to deal with the event 特殊软件

↳ This software is called and exception handler 异常处理程序
or interrupt handler
or interrupt service routine

↳ Polling 轮询输入输出
- program waits in a loop until ready for input or output
- cannot respond / move on while in loop
- easy
- bad

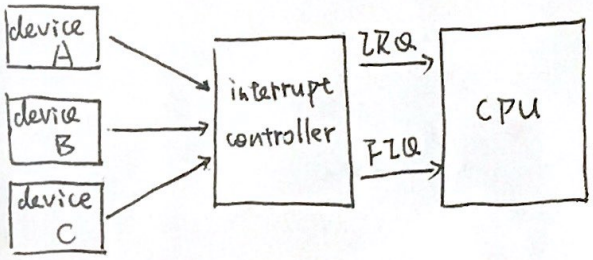
↳ interrupt driven 中断驱动
- program operates without wait loops
- handles I/O by responding to interrupts
- harder
- better

↳ Some exceptions are synchronous 同步
- the result from the execution (or attempted execution of an instruction)
- once handled, control returns to the next instruction.

↳ Some exceptions are asynchronous 异步
- called interrupts
- may occur at any time
- before handling save program state
- once handled, restore program state and continue with next instruction

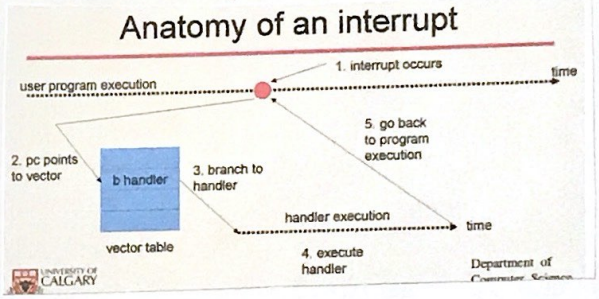
↳ Interrupts are usually triggered by external hardware
- an electrical signal applied to an input pin on the CPU
- ARM has 2
  - IRQ - normal interrupt requests
  - FIQ - fast interrupt requests.

↳ There ~~can~~ may be many source of an interrupt
- CPU has one IRQ
- an interrupt controller coordinates many sources.



↳ The controller serializes and prioritizes contending interrupts
控制器对 contending interrupts 进行串行化和排序

↳ Normally one handler for each type of exception
- CPU selects the exception handler to use with exception vector table
- or interrupt vector table



解剖图

Interrupt sources

↳ most peripherals can be configured to generate interrupts
大多数外围设备可以配置生成中断
- e.g., when edge is detected on a gpio pin
  *GPREN0 = 0x1 << 23

↳ Also have registers to record which peripherals have generated interrupts
还有寄存器来记录哪些外围设备产生了中断
- Interrupt controller
  *GPEDS0 & (0x1 << 23)

Interrupt controller

↳ has 3 registers to enable specific IRQs
*IRQ_ENABLE-IRQ_2 = (0x1 << 20)

↳ has 3 registers that record which specific IRQ is pending.

The base address for the ARM interrupt register is 0x7E00B000.

Registers overview:

Address offset'	Name	Notes
0x200	IRQ basic pending	
0x204	IRQ pending 1	
0x208	IRQ pending 2	
0x20C	FIQ control	
0x210	Enable IRQs 1	
0x214	Enable IRQs 2	
0x218	Enable Basic IRQs	
0x21C	Disable IRQs 1	
0x220	Disable IRQs 2	
0x224	Disable Basic IRQs	

p 112

The following is a table which lists all interrupts which can come from the peripherals which can be handled by the ARM.

ARM peripherals interrupts table.

#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1	system timer match 1	17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3	system timer match 3	19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9	USB controller	25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

The table above has many empty entries. These should not be enabled as they will interfere with the GPU operation.

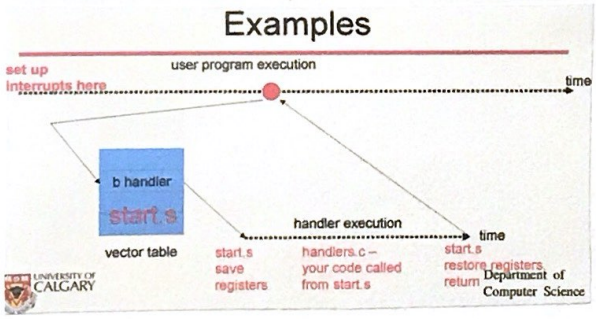
```
// Reference to the global shared value
extern unsigned int sharedValue;

void IRQ_handler()
{
    // Handle GPIO interrupts in general
    if (*IRQ_PENDING_2 == (0x1 << 20)) { // was there an interrupt from gpio[3]
        if (*GPEDS0 == (0x1 << 17)) {  // was the interrupt from pin 17
            // Clear the interrupt by writing a 1 to the GPIO Event Detect
            // Status Register at bit 17 (p. 96 of the Broadcom Manual)
            *GPEDS0 = (0x1) << 17);
            // Handle the interrupt:
            // We do this by incrementing the shared global variable
            sharedValue++;
        }
    }
    // Return to the IRQ exception handler stub
    return;
}
```

```
// Declare a global shared variable
unsigned int sharedValue;
void main() {
  unsigned int localValue

  // Init shared and local value
  localValue = sharedValue = 0;

  init_GPIO17_to_risingEdgeInterrupt();  // init input pin with interrupt enabled
  enableIRQ();  // enable interrupts for the CPU

  // Loop forever, waiting for interrupts to change the shared value
  while (1) {
    if (localValue != sharedValue) {  // has sharedValue changed - interrupt happened
      localValue = sharedValue;  // update localValue
      // Print out the shared value
      uart_puts("\nsharedValue is: "); uart_puthex(sharedValue); uart_puts( "\n");
    }
}
```

Interrupt controller

- Note: GPIO IRQs are grouped into 3 banks
  - IRQ 49: pins 0-27
  - IRQ 50: pins 28-45
  - IRQ 51: pins 46-53
  - IRQ 52: all pins

Examples



How to write a handler

- should be minimal amount of code
  - needs to be fast
  - don't want to be in handler for long
- needs to sort out where the interrupt is coming from
- use global variable to share data
- clear the interrupt before returning.

SNES

↳ Super Nintendo Entertainment System Controller

GPIO ~~SEL~~ SNES Pin - P: 2/3

DAT 9 GPIO 10 (SPI_MOSI)
LAT 21 GPIO 9 (SPI_MISO)
CLK 23 GPIO 11 (SPI_CLK)

SNES Button Numbers

Button Number/ Clock Cycle	Button
1	B
2	Y
3	Select
4	Start
5	Joy-pad UP
6	Joy-pad DOWN
7	Joy-pad LEFT
8	Joy-pad RIGHT
9	A
10	X
11	Left
12	Right
13-16	Unused – for future use (always high)

SNES Lines

↳ LATCH: latch line
  ↳ used by the core to instruct the controller to latch the state of the buttons internally
  - which buttons on the controller are pressed

↳ CLOCK: clock line
  ↳ The core sends a clock signal of 16 intervals
  - one for each button
  - Interval 1 is for Button B; 2 for Y etc...

↳ DATA: data line
  ↳ Controller sends serial signal to core indicating which buttons are pressed.

Initializing SNES

- set GPIO pin 11 (CLK) to output
- set GPIO pin 9 (LATCH) to output
- set GPIO pin 10 (DATA) to input (code 0)

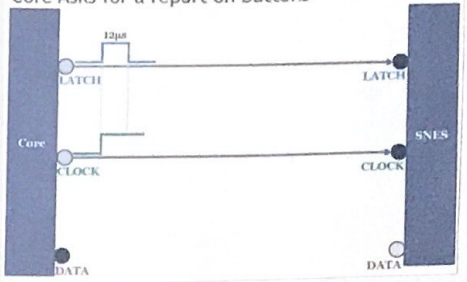
Communication Protocal 通信协议

↳ LATCH is used to instruct the SNES to sample and latch buttons
  - Which buttons are pressed
  - Latched in a register

↳ CLOCK line is used to ~~instruct the SNES to sa~~ synchronize serial signal sent from SNES to the core
时钟线用于同步从SNES发送到内核的串行信号.

↳ DATA line is used to read the serialized latched register representing button sampling.
数据线用于读取代表按钮扫描采样的串行锁存寄存器

Communication Protocol (1)
Core Asks for a report on buttons



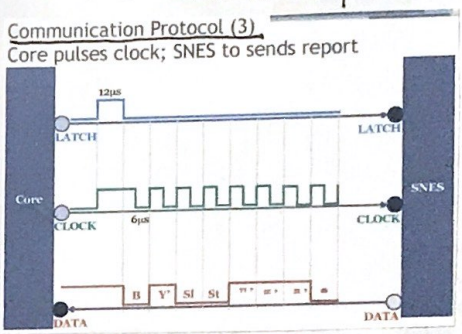
Communication Protocol (2)

SNES Samples Buttons

↳ Sampling buttons = creating a register representation of which buttons are pressed

↳ One bit for each button is sufficient
  - Example: 1111101111100
  - Three buttons are pressed (buttons 0, 1, & 7)

Communication Protocol (3)
Core pulses clock; SNES to sends report



Communication Protocol (3)

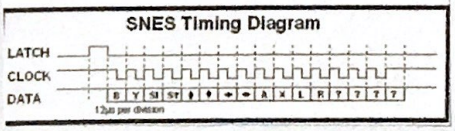
Core pulses clock; SNES to sends report

↳ Pulsing continues for 16 times

↳ SNES has 12 buttons
  - First 12 pulses sample the buttons

↳ Last 4 pulses are not used (always 1)
  - Reserved for future extensions

Communication Protocol (3)



※通讯协议，开放系统互联协议中最早的协议之一，它为连接不同操作系统和不同硬件体系结构的互联网络提供通信支持.

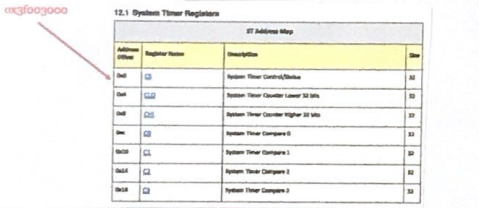
Again - ABM in C

```
#define CLK  11
#define LAT   9
#define DAT  10
```
← GPIO pins

```
#define GPSET0  7
#define GPCLR0  10
#define GPLEV0  13
```
← register offsets from GPIO base in words

```
#define HCYC  6
#define FCYC  12
```
← cycle times in micro seconds

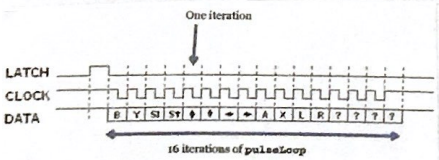
System Timer




```
sysclock = (unsigned int *)mmap( ... CLOCK_BASE );
clock = (long *)(sysclock+1);
```

```
void timedWait( int delay ) {
    long endTime;

    endTime = *clock + delay;
    while ( endTime > *clock );
}
```

```
int readSnesController() {
    int conWord = 0;
    int bitMask;
    int bit;

    gpio[GPSET0] = 1 << CLK;          // CLK high
    gpio[GPSET0] = 1 << LAT;          // Latch high
    timedWait( FCYC );                // wait for 12usec
    gpio[GPCLR0] = 1 << LAT;          // Latch low

    bitMask = 1;
    while ( bitMask < 0x10000 ) {
        timedWait( HCYC );            // wait for 6usec
        gpio[GPCLR0] = 1 << CLK;      // CLK low
        timedWait( HCYC );            // wait for 6usec
        bit = (gpio[GPLEV0] >> DAT) & 1;    // read DAT
        if ( bit ) conWord |= bitMask;      // set bit in conWord
        gpio[GPSET0] = 1 << CLK;      // CLK high
        bitMask <<= 1;
    }

    return conWord;
}
```




```
#ifndef __SNES__
#define __SNES__

int initNES();
int readSnesController();
void timedWait( int delay );

#define SNES_A          8
#define SNES_B          0
...
#define SNES_RIGHT      11

#define SNES_IS_BUTTON_DOWN(w, button)  (((w) >> (button)) & 1) == 0)

char *buttonString[] = {
    "B", "Y", "Select", "Start",
    "Up", "Down", "Left", "Right",
    "A", "X", "L", "R" };

#endif
```

Raspberry Pi System Timer

System timer

- The Broadcom system on chip has a free running 64 bit counter
- The counter is driven by a clock with frequency of 1 Mhz
  - 1 million ticks per second
  - 1 microsecond per tick

Registers for the time

↳ base address is 0x3f003000

Polling the timer

↳ Use the two 32-bit registers
  - CLO - low order bits
  - CHI - high order bits

↳ Not aligned for 64-bit read

↳ Do two 32-bit reads

How to read

- read CHI
- read CLO
- read CHI again
  - if CHI changed we know the counter turned over between bits 31/32
- if CHI changed, then read CLO again
- assemble a 64-bit count from the two 32-bit parts

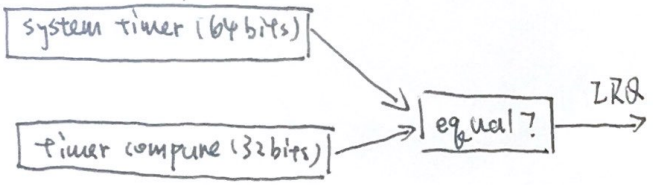
```
unsigned long getSystemTimerCounter() {
    unsigned int h=-1, l;
    // we must read MMIO area as 2 separate 32
    // bit reads
    h = *CHI;
    l = *CLO;
    // we have to repeat it if high word changed
    // during read
    if (h != *CHI) {
        h = *CHI;
        l = *CLO;
    }
    // compose long int value
    return ((unsigned long) h << 32) | l;
}
```

Polled timed wait

↳ read start time from the timer
↳ wait in busy loop until timer value reaches start time + desired delay.

```
void timedWait (int delay) {
    unsigned long endTime;
    endTime = getSystemTimerCounter() + delay;
    while (endTime > getSystemTimerCounter())
        asm volatile ("nop");
}
```

Timer interrupts



Timer compare registers

↳ 4 timer compare registers
- C0, C1, C2, C3

↳ timer status register (CS)
- bits 0-3 indicate timer match
- writing 1 to bits 0-3 clears corresponding interrupt

Interrupt control

↳ use bit 1 of
- IRQ_pending_1
- Enable_IRQs_1
- Disable_IRQs_1

↳ after that looks like GPIO interrupts.

To get an interrupt in the future

↳ read the timer
↳ add desired delay in microsecond
↳ write least sig 32 bits to C1 register
↳ enable C1 interrupts in Enable_IRQs_1
↳ enable interrupts in CPU

To handle timer interrupt

↳ Verify that interrupt came from C1
- Check bit 1 in pending_1 register

↳ clear interrupt
- write 1 to bit 1 of CS

↳ do what needs doing
↳ set up a future interrupt if needed

Example handler

```
void IRQ_handler() {
    // Handle GPIO interrupts in general
    if (*IRQ_PENDING_1 & 0x2) {
        // clear timer interrupt
        *CS = 0x2; // clear system timer match 1 (M1)
        // do some stuff
        // set C1 for interrupt 2s into future
        unsigned long t = getSystemTimerCounter();
        *C1 = (unsigned int)(t + 2000000) & 0xffffffff;
    }
}
```

※ 树莓派的计时器

与 Gpio 控制器类似，计时器也有一个地址。在这里计时器的地址是 20003000 (十六进制)

树莓派没有板载电源所以不加电的时候时间就不准，因此我们只能依靠这个计时器来计时。计时器本身有64位，与其搭配的还有32位的控制字、4个32位的比较寄存器，计时器是只读的。

计时器每一经 1μs 加 1，每次加完都会将低32位与比较寄存器进行比较，如果和其中一个匹配上了，就根据所匹配的寄存器对控制/或者叫状态寄存器进行更新

Video Programming

2D Arrays

↳ Two-dimensional arrays must be mapped onto 1D memory ↳ Can use row-major ordering
- Store each element of row 0, then row 1, etc.
- Used in most high-level languages, including C, C++.

⇆ or column-major ordering
- Store each element of column 0, then column 1, ...

Example

- Logical arrangement.

10	20	30
40	50	60

- Mapping in **row-major** order

10	20	30	40	50	60

- Mapping in **column-major** order

10	40	20	50	30	60

Indexing 2D Arrays

↳ The indices for a multidimensional array must be converted into an offset
- Added to array starting address

↳ For a 2D array list [0...n-1][0...m-1], row-major order, list[i][j] is:
((m*i)+j) · Esize

Example C Code declaration of
char array [2][3] → 2行3列
array [1][2] maps to an offset of (3*1+2)*1
((m*i)+j) · Esize

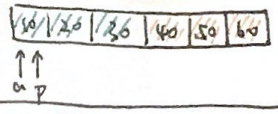
0,0	0,1	0,2
1,0	1,1	1,2

Logical view →

0	1	2
3	4	5

Physical view

In C



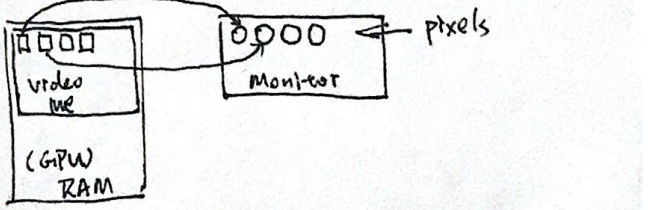

```
int a[6] = {10, 20, 30, 40, 50, 60};
int *p;
int b;

    a[3] = 5;   // assign element 3 of a
    p = a;      // p refers to elements of a
  *(p+3) = 5;   // does the same assignment
  b = a[5];     // assign b value of a[5] - 60
  b = *(p+4);   // assign b value of a[4] - 50
  // note - with C pointer arithmetic
  // C accounts for no. byte in int.
```

```
int a[6] = {10, 20, 30, 40, 50, 60};
int nr = 2, nc = 3; // n rows, n columns
int *p;
int b;
    a[1*nc+2] = 5; // assigns a[1,2]
    p = a;         // p refers to elements of a
    *(p+1*nc+2) = 5; // does the same assignment
    b = a[0*nc+2]; // assign b value of a[0,2] 30
    b = *(p+4); // assign b value of a[4] - 50
```

Frame-Buffers 帧缓冲(存储)器 (显存)

※显屏幕所显示画面由一个直接映射二维矩阵位映射图(Bit Map)来实现。
帧缓存的每一个存储单元对应屏幕上的一个像素，整个帧缓存对应一帧图象。

↳ A frame-buffer is a memory mapped arrangement for monitors 帧缓冲区是用于显示器的内存映射安排
↳ Each pixel is mapped to a memory address
↳ To write a pixel, the core simply set the corresponding value in main memory
↳ Can be implemented in a separate Video RAM or in a reserved section of RAM.



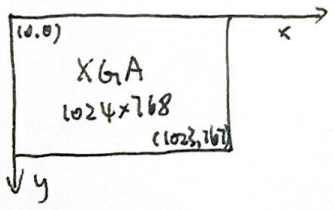
```
unsigned char *frameBuffer = (unsigned char *) 0x12345678;
int nr = 768;
int nc = 1024;
int pitch = 1024;
int i, j;

frameBuffer[i*nc + j] = 0xff  // assuming nc = pitch
frameBuffer[i*pitch + j] = 0xff; // best!
```

Drawing a Pixel

↳ To draw a single pixel, set its value in the frame buffer

- Must map from 2D logical space to 1D physical RAM
  - Must know pitch and element size for particular resolution
- Use formula: row offset = y * pitch
  - x, y are pixel's coordinates
  - Origin is upper left-hand corner
  - x range: 0 to width-1 ✓
  - y range: 0 to height-1 ✓
  - pitch is number of bytes in row
    ↳ not necessarily equal to ~~row~~ number of elements * size
- physical offset = row offset + x*element size

Frame Buffer Coordinates



Examples

- E.g. XGA (3 bytes per pixel, 24-bit color)
- pitch given by frame buffer info
  - happens to be 3*1024 = 3072
- element offset = (y*pitch) + x*3
  - Pixel (0,0)
    0*pitch + 0*0 = 0
  - Pixel (1023, 767) (lower right-hand corner)
    (767*pitch) + 1023*3 = 2359293
  - check
    (1024*768-1)*3 = 2359293.

```
static FrameBufferInfo
fbi_attribute_((aligned(16)));

void fillBoxVideo(
        unsigned char r,
        unsigned char g, unsigned char b,
        unsigned int x0, unsigned int y0,
        unsigned int w, unsigned int h ) {
  unsigned char *fb;
  int x, y;

  for( y = y0; y < y0+h; y++ ) {
    fb = (unsigned char*) ((unsigned int)fbi.fb+
            y * fbi.pitch + 3 * x0);
    for( x = x0; x < x0 + w; x++ ) {
      *fb++ = r;
      *fb++ = g;
      *fb++ = b;
    }
  }
}
```

Plot a point 점 찍기

- extrapolate from box fill

```
// NB - untested - probably needs
some type casting
// for 24-bit color
static FrameBufferInfo
fbi_attribute_((aligned(16)));

void point(
    unsigned char r,
    unsigned char g,
    unsigned char b,
    unsigned int x,
    unsigned int y ) {
    unsigned char *fb;

// add more code to check bounds if
you like
    fb = fbi.fb+y*fbi.pitch+3 * x;
    *fb++ = r;
    *fb++ = g;
    *fb++ = b;
}
```

DDA

↳ Digital Difference Analyzer

↳ a well-known algorithm for drawing lines on rasters.

↳ Bresenham's algorithm is best-known example

???

```
void line(int x0, int y0, int x1, int y1) {

    int dx = abs(x1-x0), sx = x0<x1 ? 1 : -1;
    int dy = abs(y1-y0), sy = y0<y1 ? 1 : -1;
    int err = (dx>dy ? dx : -dy)/2, e2;

    for(;;){
      setPixel(x0,y0);
      if (x0==x1 && y0==y1) break;
      e2 = err;
      if (e2 >-dx) { err -= dy; x0 += sx; }
      if (e2 < dy) { err += dx; y0 += sy; }
    }
}
```

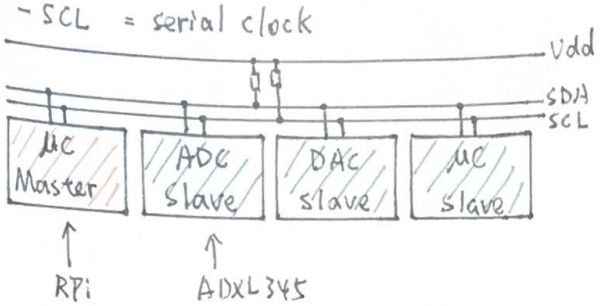
Pi 3 Video Control Mailbox ???

I²C - Inter-Integrated Circuit bus

※ I²C 是一种简单、双向二线制同步串行总线。它只需要两根线即可在连接于总线上的器件之间传送信息。

↳ Sometimes called a two-wire interface (TWI)

↳ Multiple devices chained on 2 wires
- does not include power
- SDA = serial data
- SCL = serial clock



↳ Each device must have a unique address

↳ Devices contain a set of register that control their behaviour
- these are specific to each device

↳ Use the device to writing to and reading from these registers.

↳ Write to a device register
- send device address
  - address on the bus
- send register address
  - address within the device
- send data.

↳ read from a device register
- send device address
  - address on the bus
- send register address
  - address within the device
- to listen for data to come back

Important methods

- read_byte_data(addr, cmd)
  - returns byte from register
- read_word_data(addr, cmd)
  - returns word (16-bits) from register
- read_i2c_block_data(addr, cmd, len=32)
  - returns list of len bytes from registers starting at cmd
- write_byte_data(addr, cmd, val)
  - writes val (8-bit) to register
- write_word_data(addr, cmd, val)
  - writes val (16-bits) to register
- write_i2c_block_data(addr, cmd, [vals])
  - writes list of b-bit [vals] to registers starting at cmd

ARM I²C

↳ 前面

On RPi

↳ On RPi I2C happens through the GPIO
- requires specific pin selection
- uses alternate function.

BSC (I²C)

Broadcom Serial Controller

↳ RPi has 3 BSC masters
- BSC0 0x7E20-5000
- BSC1 0x7E80-4000
- BSC2 0x7E80-5000

↳ Use BSC1
- very old Pi's may use BSC0
- BSC0 now used for EEPROM ID on hat

Two things to do

↳ GPIO
- configure GPIO pins for BSC1 pins 2 and 3
- required at initialization before you can use BSC1

↳ BSC1 registers
- use these registers to perform I²C transfers.

GPIO Registers

Set pins to BSC1 functions here

Address	Field Name	Description	Size	Read/ Write	
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W	Function Select
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W	
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W	
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W	
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W	
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W	
0x 7E20 0018		Reserved			
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W	Set (Write 1)
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W	
0x 7E20 0024		Reserved			
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W	Clear (Write 0)
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W	
0x 7E20 0030		Reserved			
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R	Level (Read)
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R	
0x 7E20 003C		Reserved			

GPIO alt functions

Every GPIO pin can carry an alternate function. Up to 6 alternate function are available but not every pin has that many alternate functions. The table below gives a quick over view.

	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5				
GPIO1	High	SCL0	SA4				
GPIO2	High	SDA1	SA3				
GPIO3	High	SCL1	SA2				
GPIO4	High	GPCLK0	SA1				ARM_TDI
GPIO5	High	GPCLK1	SA0				ARM_TDO
GPIO6	High	GPCLK2	SOE_N / SE				ARM_RTCK
GPIO7	High	SPI0_CE1_N	SWE_N / SRW_N				
GPIO8	High	SPI0_CE0_N	SD0				

GPIO initialization

↳ Need GPIO pins 2,3 initialized for Alt0 function

↳ Review how you did this previously
- clear all config bits for the pin
  - equivalent to configuring for input
- write bits for desired functions
  - output or Alt.

29-27	FSEL9	FSEL9 - Function Select 9 000 = GPIO Pin 9 is an input 001 = GPIO Pin 9 is an output 100 = GPIO Pin 9 takes alternate function 0 101 = GPIO Pin 9 takes alternate function 1 110 = GPIO Pin 9 takes alternate function 2 111 = GPIO Pin 9 takes alternate function 3 011 = GPIO Pin 9 takes alternate function 4 010 = GPIO Pin 9 takes alternate function 5	R/W	0
26-24	FSEL8	FSEL8 - Function Select 8	R/W	0
23-21	FSEL7	FSEL7 - Function Select 7	R/W	0
20-18	FSEL6	FSEL6 - Function Select 6	R/W	0
17-15	FSEL5	FSEL5 - Function Select 5	R/W	0
14-12	FSEL4	FSEL4 - Function Select 4	R/W	0
11-9	FSEL3	FSEL3 - Function Select 3	R/W	0
8-6	FSEL2	FSEL2 - Function Select 2	R/W	0
5-3	FSEL1	FSEL1 - Function Select 1	R/W	0
2-0	FSEL0	FSEL0 - Function Select 0	R/W	0

In C - map GPIO registers

```
#include ...
...
#define GPIO_BASE 0x3f200000
#define BLOCK_SIZE 4096
...
int main()
{
    // Open /dev/mem
    int fd = open( "/dev/mem", O_RDWR|
        O_SYNC );
    unsigned int *gpio = (unsigned int *)mmap(
        NULL, BLOCK_SIZE, PROT_READ |
        PROT_WRITE, MAP_SHARED, fd,
        GPIO_BASE );
// Define gpio0/1 as sda0/scl0 (alt 0)

// clear gpio2
gpio[0] = gpio[0] & ~(0b111 << (SDA1 * 3));
// alt0 in gpio2
gpio[0] = gpio[0] | (0b100 << (SDA1 * 3));
// clear gpio3
gpio[0] = gpio[0] & ~(0b111 << (SCL1 * 3));
// alt0 in gpio3
gpio[0] = gpio[0] | (0b100 << (SCL1 *3));
...
```

Use BSC1 registers for transfers

↳ Registers listed on p28-29 . . .

↳ There are 8 registers for each BSC

↳ They are all the same, but correspond to different pins / buses.

offset in bytes
remember pointer arithmetic

I2C Address Map

Address Offset	Register Name	Description	Size
0x0	C	Control	32
0x4	S	Status	32
0x8	DLEN	Data Length	32
0xc	A	Slave Address	32
0x10	FIFO	Data FIFO	32
0x14	DIV	Clock Divider	32
0x18	DEL	Data Delay	32
0x1c	CLKT	Clock Stretch Timeout	32

first in first out

C – Control

- Used to
  - enable interrupts
  - clear the FIFO
  - define R/W operation

Bit(s)	Field Name	Description	Type	Reset
31:16		*Reserved - Write as 0, read as don't care*		
15	I2CEN	I2C Enable 0 = BSC controller is disabled 1 = BSC controller is enabled	RW	0x0
14:11		*Reserved - Write as 0, read as don't care*		
10	INTR	INTR Interrupt on RX 0 = Don't generate interrupts on RXR condition. 1 = Generate interrupt while RXR = 1.	RW	0x0
9	INTT	INTT Interrupt on TX 0 = Don't generate interrupts on TXW condition. 1 = Generate interrupt while TXW = 1.	RW	0x0
8	INTD	INTD Interrupt on DONE 0 = Don't generate interrupts on DONE condition. 1 = Generate interrupt while DONE = 1.	RW	0x0
7	ST	ST Start Transfer 0 = No action. 1 = Start a new transfer. One shot operation. Read back as 0.	RW	0x0
6		*Reserved - Write as 0, read as don't care*		
5:4	CLEAR	CLEAR FIFO Clear 00 = No action. x1 = Clear FIFO. One shot operation. 1x = Clear FIFO. One shot operation. If CLEAR and ST are both set in the same operation, the FIFO is cleared before the new frame is started. Read back as 0. Note: 2 bits are used to maintain compatibility to previous version.	RW	0x0
3:1		*Reserved - Write as 0, read as don't care*		
0	READ	READ Read Transfer 0 = Write Packet Transfer. 1 = Read Packet Transfer.	RW	0x0

S – Status

- Status of BSC

Bit(s)	Field Name	Description	Type	Reset
31:10		*Reserved - Write as 0, read as don't care*		
9	CLKT	CLKT Clock Stretch Timeout 0 = No errors detected. 1 = Slave has held the SCL signal low (clock stretching) for longer and that specified in the I2CCLKT register Cleared by writing 1 to the field.	RW	0x0
8	ERR	ERR ACK Error 0 = No errors detected. 1 = Slave has not acknowledged its address. Cleared by writing 1 to the field.	RW	0x0
7	RXF	RXF - FIFO Full 0 = FIFO is not full. 1 = FIFO is full. If a read is underway, no further serial data will be received until data is read from FIFO.	RO	0x0
6	TXE	TXE - FIFO Empty 0 = FIFO is not empty. 1 = FIFO is empty. If a write is underway, no further serial data can be transmitted until data is written to the FIFO.	RO	0x1
5	RXD	RXD - FIFO contains Data 0 = FIFO is empty. 1 = FIFO contains at least 1 byte. Cleared by reading sufficient data from FIFO.	RO	0x0
4	TXD	TXD - FIFO can accept Data 0 = FIFO is full. The FIFO cannot accept more data. 1 = FIFO has space for at least 1 byte.	RO	0x1
3	RXR	RXR - FIFO needs Reading (full) 0 = FIFO is less than full and a read is underway. 1 = FIFO is or more full and a read is underway. Cleared by reading sufficient data from the FIFO.	RO	0x0
2	TXW	TXW - FIFO needs Writing (full) 0 = FIFO is at least full and a write is underway (or sufficient data to send). 1 = FIFO is less than full and a write is underway. Cleared by writing sufficient data to the FIFO.	RO	0x0
1	DONE	DONE Transfer Done 0 = Transfer not completed. 1 = Transfer complete. Cleared by writing 1 to the field.	RW	0x0
0	TA	TA Transfer Active 0 = Transfer not active. 1 = Transfer active.	RO	0x0

DLEN – Data Length

- specifies length of data transfer

15:0	DLEN	Data Length. Writing to DLEN specifies the number of bytes to be transmitted/received. Reading from DLEN when TA = 1 or DONE = 1, returns the number of bytes still to be transmitted or received. Reading from DLEN when TA = 0 and DONE = 0, returns the last DLEN value written. DLEN can be left over multiple packets.	RW	0x0

higher order bits – write as 0, read as don't care

A – slave address

A Register

Synopsis The slave address register specifies the slave address and cycle type. The address register can be left across multiple transfers
The ADDR field specifies the slave address of the I2C device.

Bit(s)	Field Name	Description	Type	Reset
31:7		***Reserved*** *- Write as 0, read as don't care*		
6:0	ADDR	Slave Address.	RW	0x0

FIFO

FIFO Register

Synopsis The Data FIFO register is used to access the FIFO. Write cycles to this address place data in the 16-byte FIFO, ready to transmit on the BSC bus. Read cycles access data received from the bus.
Data writes to a full FIFO will be ignored and data reads from an empty FIFO will result in invalid data. The FIFO can be cleared using the I2CC.CLEAR field.
The DATA field specifies the data to be transmitted or received.

Bit(s)	Field Name	Description	Type	Reset
31:8		***Reserved*** *- Write as 0, read as don't care*		
7:0	DATA	Writes to the register write transmit data to the FIFO. Reads from register reads received data from the FIFO.	RW	0x0

In C – map the registers

```
#include ...
#define BSC1_BASE 0x3f804000
#define BLOCK_SIZE 4096
...
int main()
{
   // Open /dev/mem
   int fd = open("/dev/mem", O_RDWR|
      O_SYNC);
   ...
   unsigned int *bsc1 = (unsigned int *)mmap(
      NULL, BLOCK_SIZE, PROT_READ|
      PROT_WRITE, MAP_SHARED, fd,
      BSC1_BASE );
```

Do read_byte_data

```
// read device id from ADXL345
// start with writing the address register, then
start write cycle
...
bsc1[3] = 0x53;// ADXL345 address
bsc1[2] = 1;// data length 1 (register address)
bsc1[4] = 0x00;//register address in FIFO to write
bsc1[1]=(1<<9)|(1<<8)|(1<<1);
//clearCLKT,ERR,DONE
bsc1[0]=(1<<15)|(1<<7); //I2CEN,ST

// wait till done
 while( !(bsc1[1] & 0x02) ) {// 0x02 masks DONE
bit
   usleep(1000);
 }
...
```

```
// get the response from the register
bsc1[2] = 1;
bsc1[1]=(1<<9)|(1<<8)|(1<<1);
 //clearCLKT,ERR,DONE
bsc1[0] = (1 << 15) | (1 << 7) | (1 << 4) | 1;
 // I2CEN, ST, CLEAR, READ

// wait till done
 while( !(bsc1[1] & 0x02) ) {  // 0x02 masks
DONE bit
   usleep(1000);
 }
 unsigned int devId = bsc1[4];
 printf( "device ID: 0x%02x\n", devId );
 munmap(bsc1, BLOCK_SIZE);
 munmap(gpio, BLOCK_SIZE);
 close(fd);
}
```

Serial Peripheral Interface Bus

Some I2C examples

- environmental breakout
- GPS
- spectral sensor
- ...

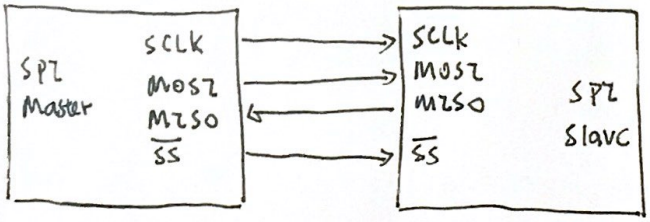
Serial Peripheral Interface Bus

- SPI bus 串行外围接口总线
- very common
  - next only to I2C
- compared to I2C
  - faster
  - more wires required
  - less flexible hardware interface

some SPI Examples

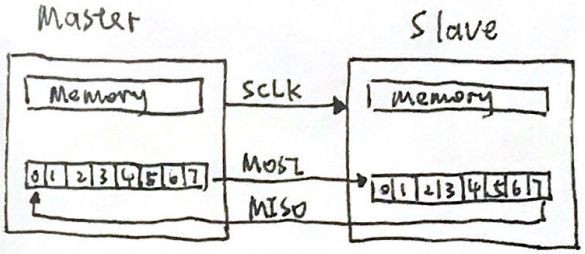
- ~~some~~ TFT display
- accelerometer
- real time clock

Basics



- Defines Master and Slave
  - master is micocontroller (CPU)
  - slave is the peripheral (sensor, display, ...)
- Two serial lines
  - MISO - master in slave out
  - MOSI - master out slave in
  - tri-state
    - high impedance when SS is high
- SCLK - serial clock
  - master provides clock signal to time serial communications
- SS
  - enables ~~sets~~ slave device
  - active low
  - separately controlled SS lines allow multiple slave devices on a ~~sig~~ single master.
    多可控制SS线允许在一个总线上有多个从设备.

~~The~~
The protocol



Actuators 致动器

↳ An actuator is a component of a machine that is responsible for moving or controlling a mechanism or system, for example by actuating (opening or closing) a valve; in simple terms, it is a "mover"

↳ Need actuators in any computer system that makes things move

  ↳ 3D printer

  ...

Current / Voltage / Power

↳ Actuators require a lot of power when compared to a logic gate

↳ Means more
  - current / voltage / power

↳ Almost always require external circuits to handle power.

Motors 发动机

↳ Focus on motors
  - most (not all) actuators have a motor
  α - most of what I have to show you as example

→ Brushed DC Motor 刷直流电机.

操作 ...