TOS for the Raspberry Pi

Yeqing Yan
Abhijit Parate

Overview:

- Hardware Architecture of the Raspberry Pi
- ARM Assembler
- Boot Sequence
- Context Switch
- Exception/Interrupt Handling
- Framebuffer

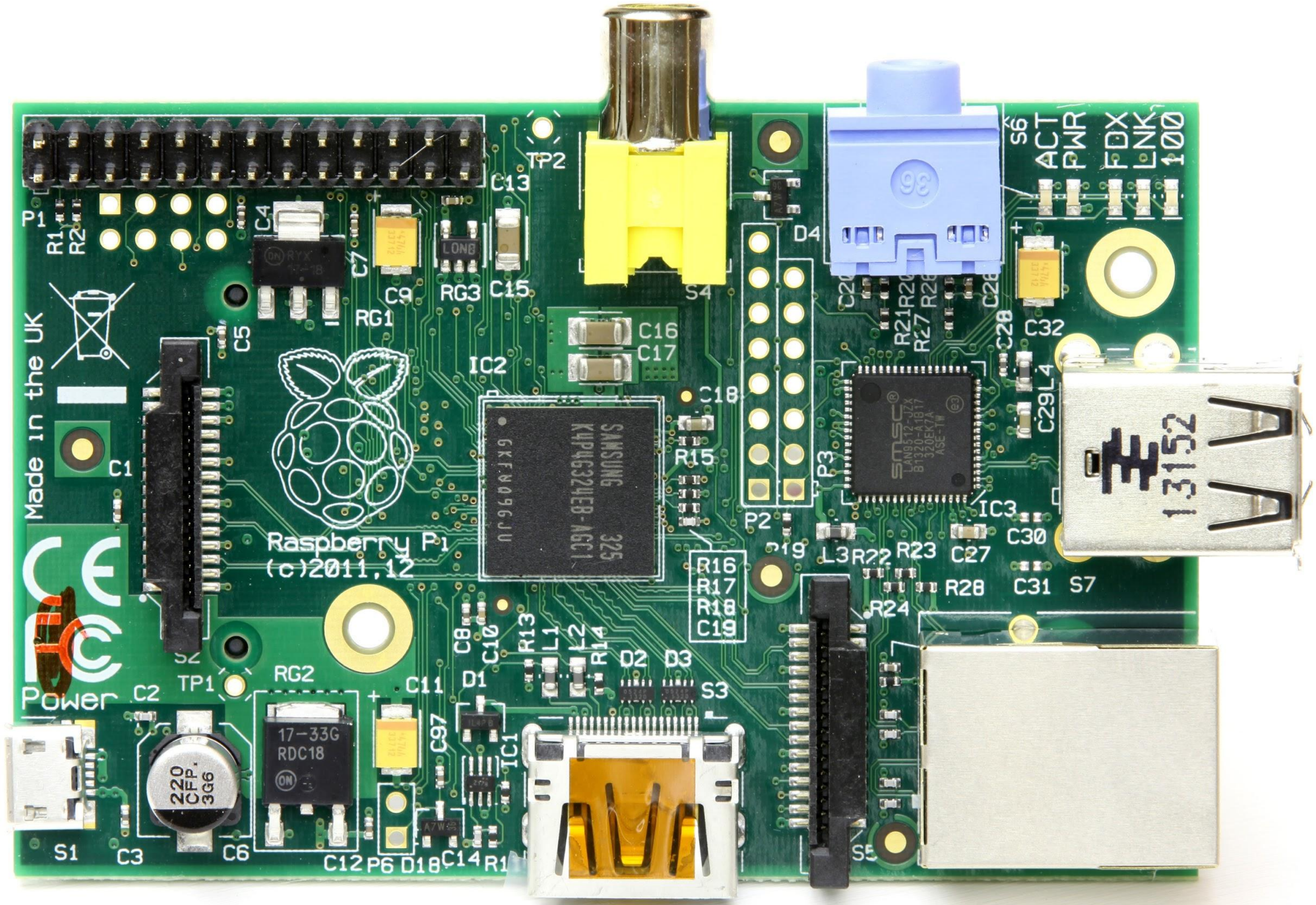
Raspberry Pi Overview

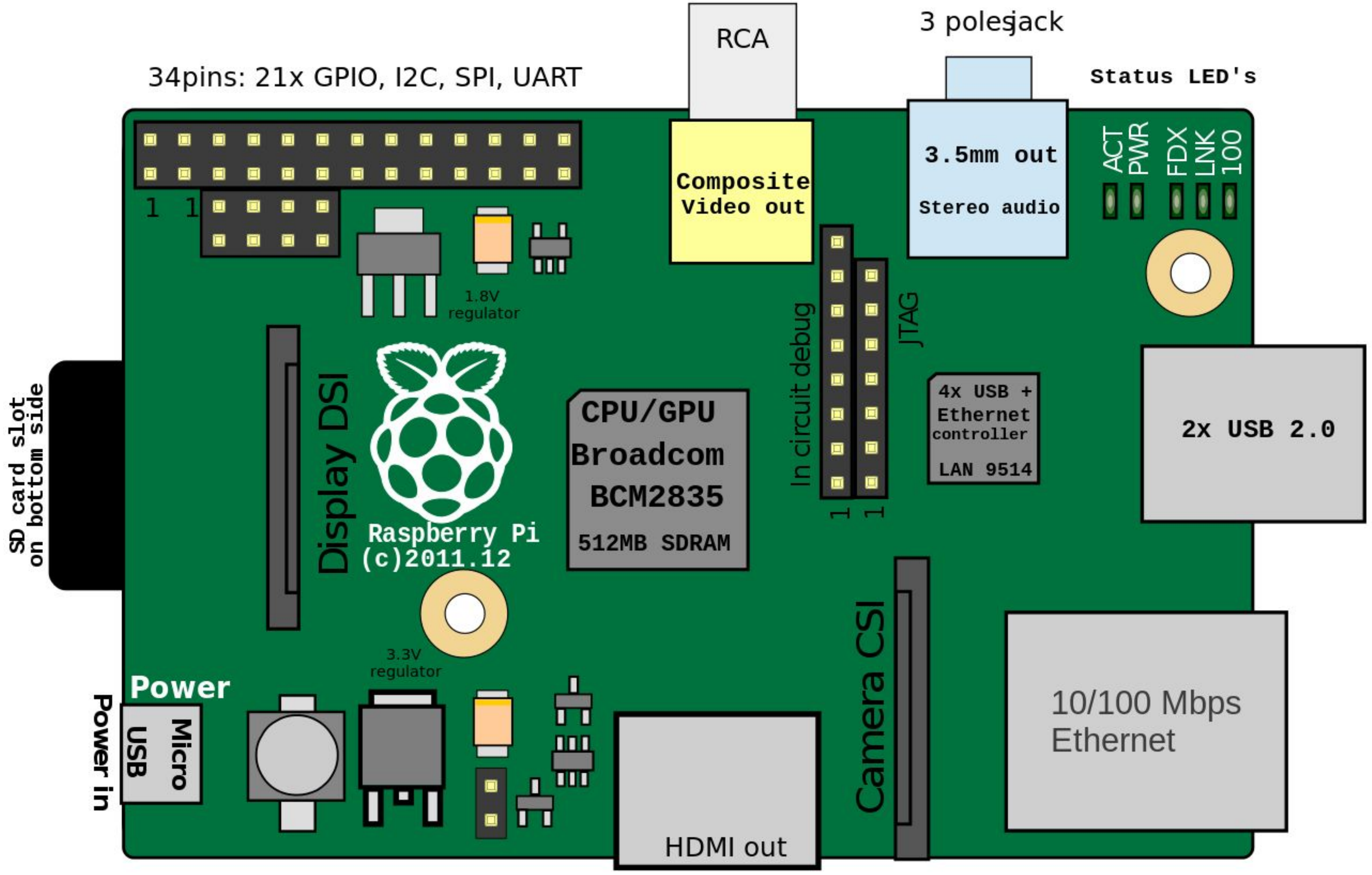
- Credit card sized computer developed by Raspberry Pi Foundation.
- Developed to promote teaching of basic computer science.
- Its very low cost ($25) and low powered.
- It comes with Broadcom SoC (System-on-Chip) which includes an ARM CPU and a GPU.
- 256MB to 1GB RAM.
- SD card to store the OS and other data.
- Few connectors to connect external devices including Internet.
- Homepage: https://www.raspberrypi.org/

Inputs & Outputs

- 10/100 Mbit/s Ethernet port (RJ45)
- 2 x USB (I/O)
- HDMI 1.4 - Audio and Video (O)
- Composite Video (O)
- 3.5mm Audio connector (O)
- 15 MIPI Camera Interface (I)
- DSI (Display Serial Interface) (O)
- Onboard Integrated Interchip Sound (I)
- 17 GPIO Pins (8 General & others multifunctionals) (I/O)

These are compact and low cost solutions developed for mobile devices by MIPI Alliance.


Raspberry Pi
(c)2011.12
Made in the UK
Power
SAMSUNG
K4P4G324EB-AGC1
smsc
LAN9512-JZX
17-33G
RDC18
ACT
PWR
FDX
LNK
100


34pins: 21x GPIO, I2C, SPI, UART
RCA
3 polesjack
Status LED's
Composite Video out
3.5mm out
Stereo audio
ACT
PWR
FDX
LNK
100
1
1
1.8V regulator
JTAG
In circuit debug
Display DSI
Raspberry Pi (c)2011.12
CPU/GPU Broadcom BCM2835
512MB SDRAM
4x USB + Ethernet controller
LAN 9514
2x USB 2.0
SD card slot on bottom side
1
1
3.3V regulator
Power
Micro USB
Power in
Camera CSI
10/100 Mbps Ethernet
HDMI out

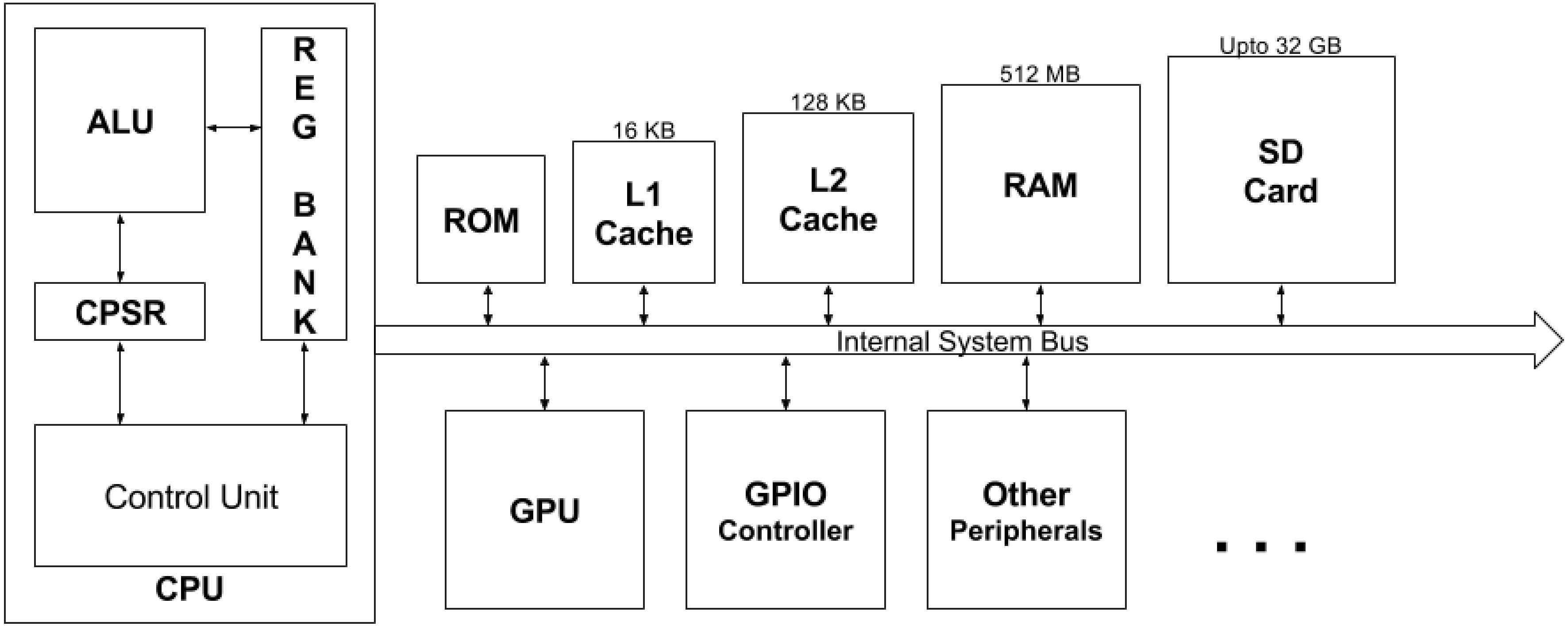
Models

Model	1 Gen				2nd Gen	ZERO	3rd Gen
	A	A+	**B**	B+			
Year	2013	2014	**2012**	2015	2015	2015	2016
Cost	$25	$20	**$35**	$25	$35	$5	$35
SoC	BCM2835				BCM2836	BCM2835	BCM2837
CPU	ARM1176JZF-S @700MHz				Cortex-A7 @900MHz	ARM1176JZF-S @1.0GHz	ARM Cortex-A53 @1.2GHz
RAM	256 MB		**512 MB**		1 GB	512 MB	1 GB
GPU	Broadcom VideoCore IV						
Others	1 USB Ethernet 8 GPIO		**2 USB** Ethernet 17 GPIO	4 USB Ethernet 17 GPIO		I Micro USB 46 GPIO	4 USB Wifi 802.11n Bluetooth 4.1 17 GPIO

Model B Specs

- **SOC**: Broadcom BCM 2835
  - CPU : 700 MHz Single-core
  - GPU : Broadcom VideoCore IV @ 250 MHz
- **RAM**: 512 MB
- **Storage**: SD card up to 512GB
- **Video**
  - Out: HDMI and Composite out | In : 15 pin MIPI Camera Interface
- **Audio**
  - Out: 3.5 mm jack | In : IIS pins
- **Other connections**
  - Ethernet 10/100 Mbits/s (RJ45)
  - 17 GPIO Pins

SoC Architecture (Model B)



Processor Operating Modes

User Mode	Restricted access to special registers and other protected resources
FIRQ	Fast Interrupt Mode
IRQ*	Interrupt (Regular) Mode
Supervisor	Full control of hardware
Abort	On failure to load instructions or data from memory
Undefined	Instruction is undefined
System*	Same degree of privilege as supervisor mode

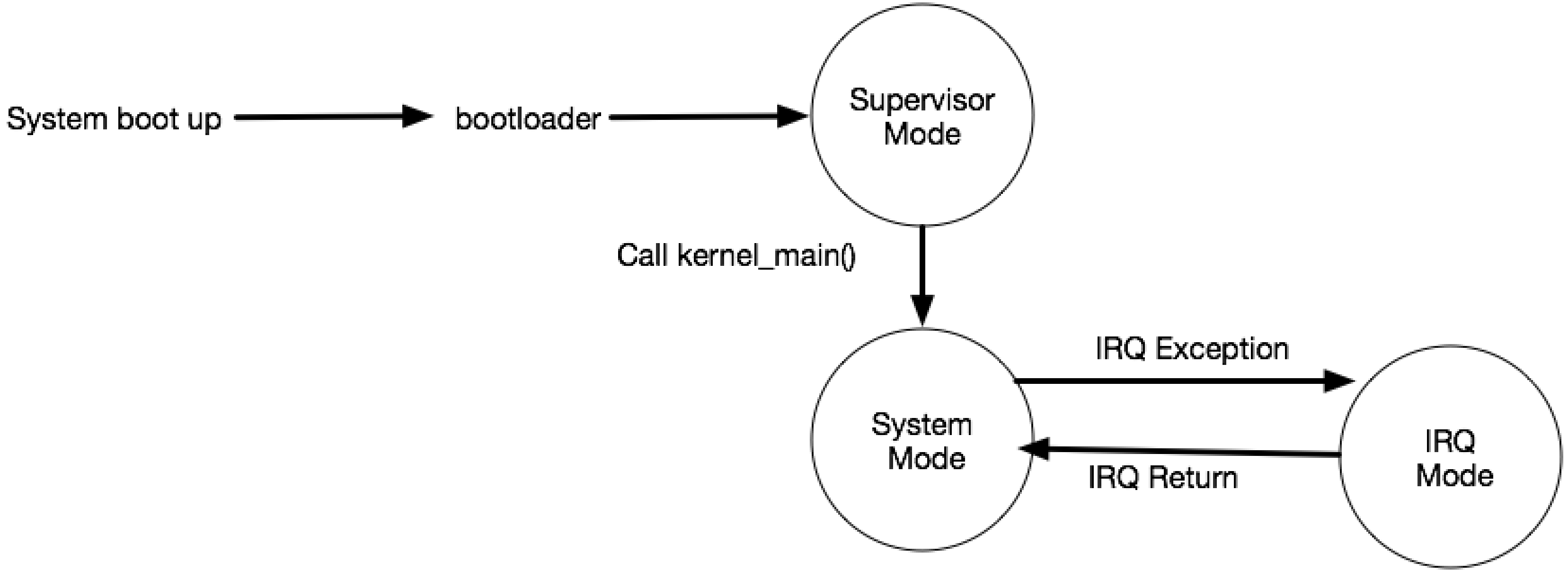
* = Used in TOS

Processor Operating Modes

- User mode: Program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur.
- The modes other than User mode are known as privileged modes, five of them are known as exception modes, they are entered when specific exceptions occur.
    - FIQ (Fast Interrupt)
    - IRQ (Regular Interrupt)
    - Supervisor
    - Abort
    - Undefined
- System mode is a privileged mode which is not entered by any exception and has exactly the same registers available as User mode, but it not subject to the User mode restrictions.
- The operating modes are controlled by using mode control bits from CPSR (see later slides)

Transitions of the CPU states used for TOS

State Machine



Registers

- The ARM processor has a total of 37 registers
  - 31 general-purpose registers, including a program counter (PC) are 32 bits wide.
  - 6 status registers. These registers are also 32 bits wide.
- Registers are arranged in partially overlapping banks, with the current processor mode controlling which bank is available. At any time, 15 general-purpose registers (R0 to R14), one or two status registers (CPSR or SPSR), and the program counter (R15, also called PC) are visible.
- The general-purpose registers R0 to R15 can be split into three groups.
- These groups differ in the way they are banked and in their special-purpose uses:
  - The unbanked registers, R0 to R7
  - The banked registers, R8 to R14
  - Register 15, the PC

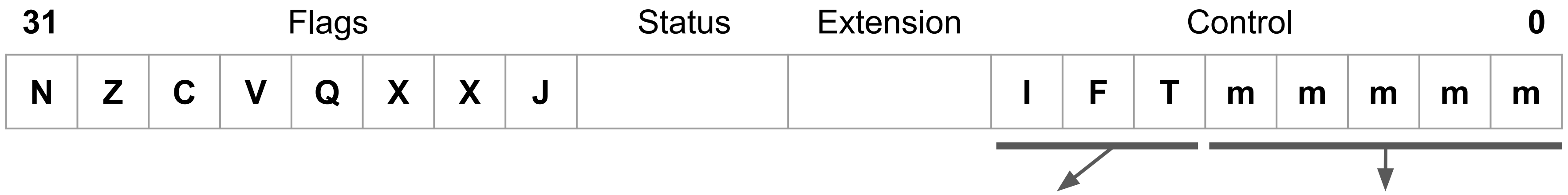
General Purpose Registers

- 64 byte register bank
- 16 registers x 32-bit each
- Can be used as 32 floating point registers
- R15 Program Counter
- R14 Link Register
- R13 Stack Pointer
- R12 Intra-Procedural scratch register
- R0 to R11 are truly general purpose registers

SP - Stack pointer
LR - Link register
PC - Program counter

R14 (LR)	R15 (PC)
R12	R13 (SP)
R10	R11
R8	R9
R6	R7
R4	R5
R2	R3
R0	R1

Current Program Status Register (CPSR)



Flags

N - Negative

Z - Zero

C - Carry

V - Overflow

Q - Saturation Flag

J - Jazzelle State Bit

Status and Extension byte are **not** used in RaspberryPi.

Control

I - Disable IRQ

F - Disable FIRQ

T - Thumb state

Mode control

10000 - User
10001 - FIQ
10010 - IRQ
10011 - Supervisor
10111 - Abort
11011 - Undefined
11111 - System

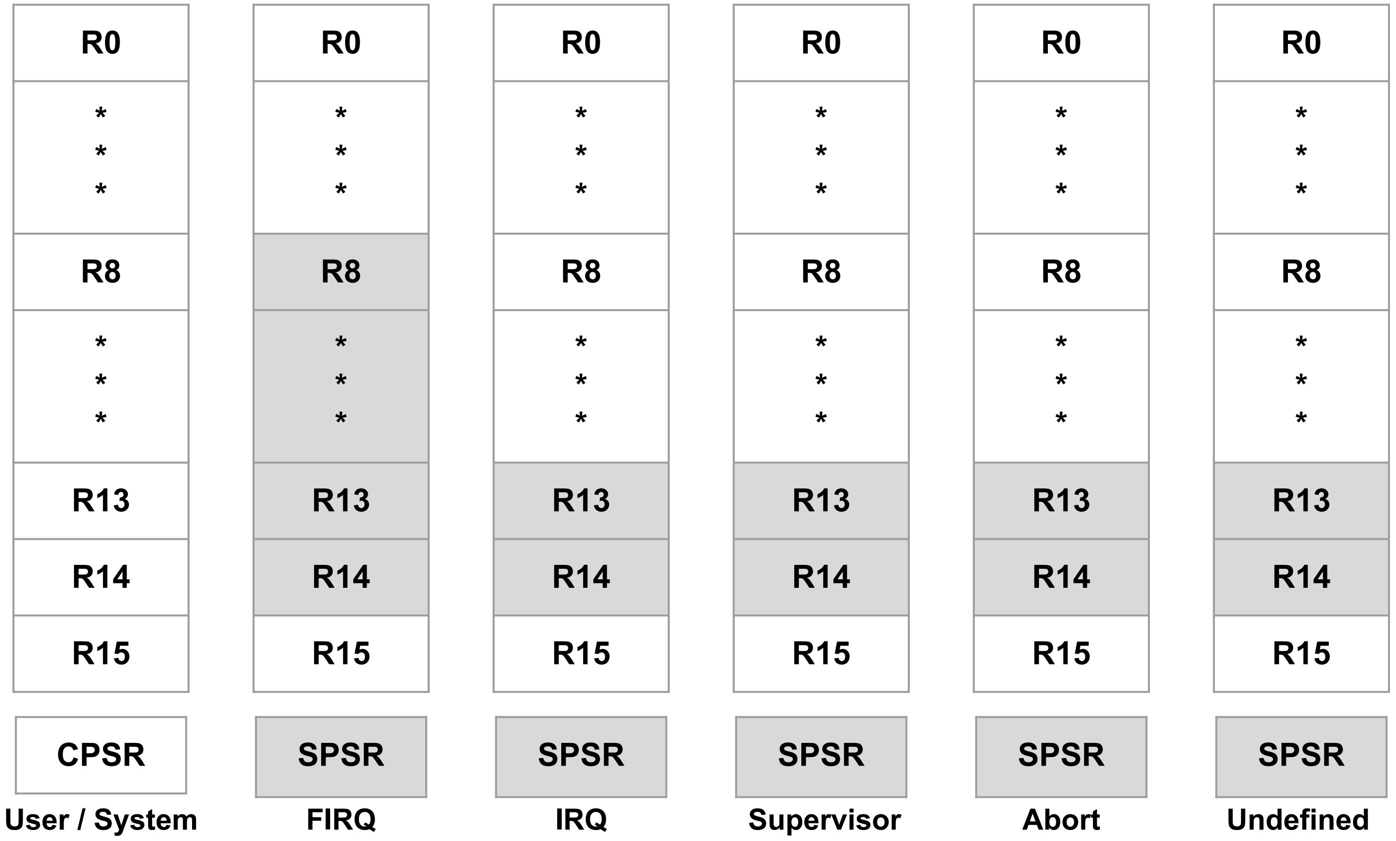
CPSR is also called **SPSR** (Saved State Program Register) in privileged modes & has same structure.

Flags

31							24
N	**Z**	**C**	**V**	**Q**	**X**	**X**	**J**

N	**Negative**	Set if the current operation results in a negative value.
Z	**Zero**	Set if the current operation results in a zero value.
C	**Carry**	Set if the current operation results in a zero value.
V	**Overflow**	Set if the current operation resulted in a carry which flipped the sign of result.
Q	**Saturation Flag**	Set if the current operation saturated the result register.
J	**Jazzelle State Bit**	If processor is in jazzelle mode, it can execute a subset of Java bytecode directly.

General Purpose Registers



CPSR -
Current Program
Status Register

SPSR -
Saved Program
State Register

Assembly Instructions used in TOS

Instruction	Usage
`add <dest>, <value1>, <value2>`	ADD adds two values. The value1 comes from a register. The value2 can be either an immediate value or a value from a register, the result is stored to dest register.
`b{l} <target_address>`	B (Branch) and BL (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow. BL also stores a return address in the link register, R14.
`cmp <value1>, <value2>`	CMP (Compare) compares two values. The first value comes from a register. The second value can be either an immediate value or a value from a register.
`mov <des>, <src>`	MOV (Move) writes a value to the destination register. The value can be either an immediate value or a value from a register.
`pop {r4, r5}`	POP (Pop Multiple Registers) loads a subset (or possibly all) of the general-purpose registers and the PC. from the stack.
`push {r4, r5}`	PUSH (Push Multiple Registers) stores a subset (or possibly all) of the general-purpose registers and the LIB to the stack.

ARM Subroutines

```
Addr    Machine Code        Assembly
-------------------------------------------
0x0:    e3 a0 01 00             mov r1, #0
0x4:    eb 00 00 00             bl L2
0x8:    ea ff ff fe         L1: b  L1
0xC:    e2 81 10 01         L2: add r1, r1, #1
0x10:   e1 a0 f0 0e             mov pc, lr
```

```
PC 0x0
LR
R1 0x0
```

```
PC 0x4
LR
R1 0x0
```

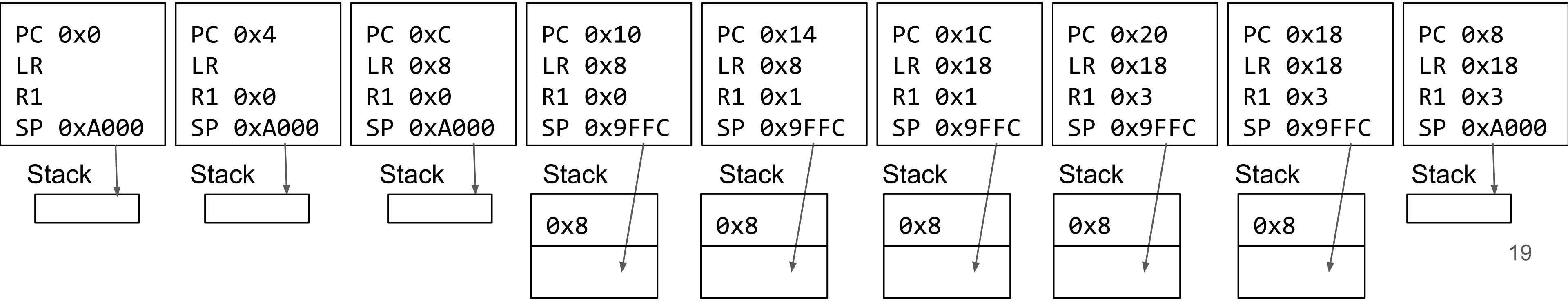
```
PC 0xC
LR 0x8
R1 0x0
```

```
PC 0x10
LR 0x8
R1 0x1
```

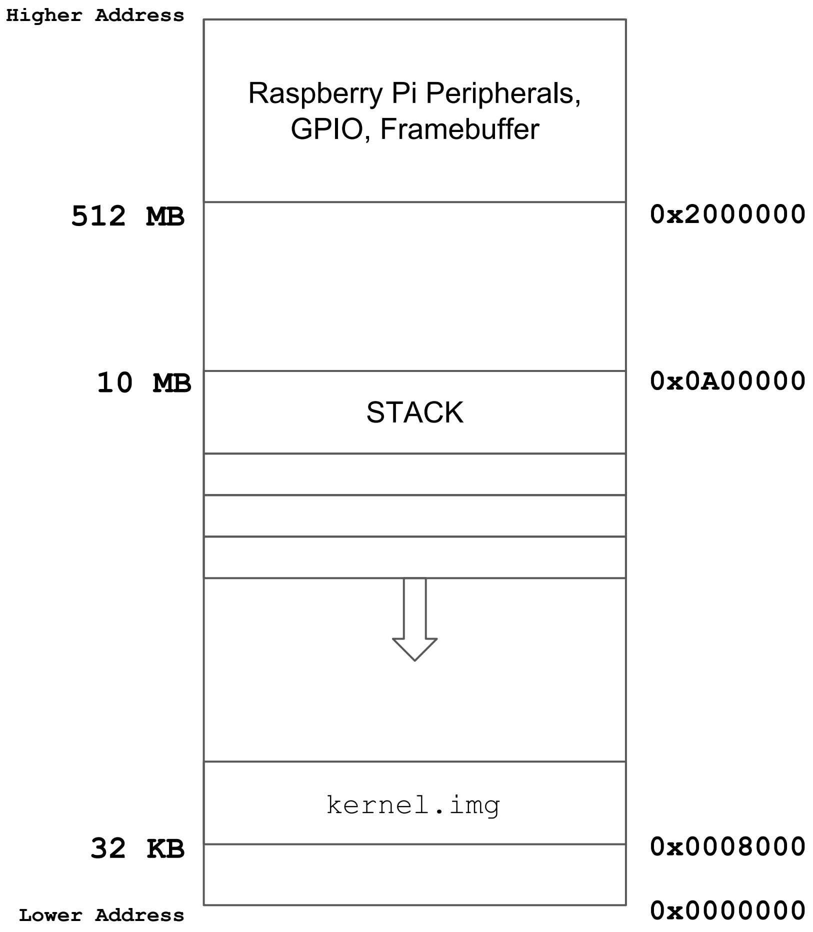
```
PC 0x8
LR 0x8
R1 0x1
```

ARM Nested Subroutine Calls

```
Addr    Machine Code        Assembly
0x0     e3 a0 10 00         mov r1, #0
0x4     eb 00 00 00         bl L2
0x8     ea ff ff fe     L1: b L1
0xC     e5 2d e0 04     L2: push {lr}
0x10    e2 81 10 01         add r1, r1, #1
0x14    eb 00 00 00         bl L3
0x18    e4 9d f0 04         pop {pc}
0x1c    e2 81 10 02         add r1, r1, #2
0x20    e1 a0 f0 0e     L3: mov pc, lr
```



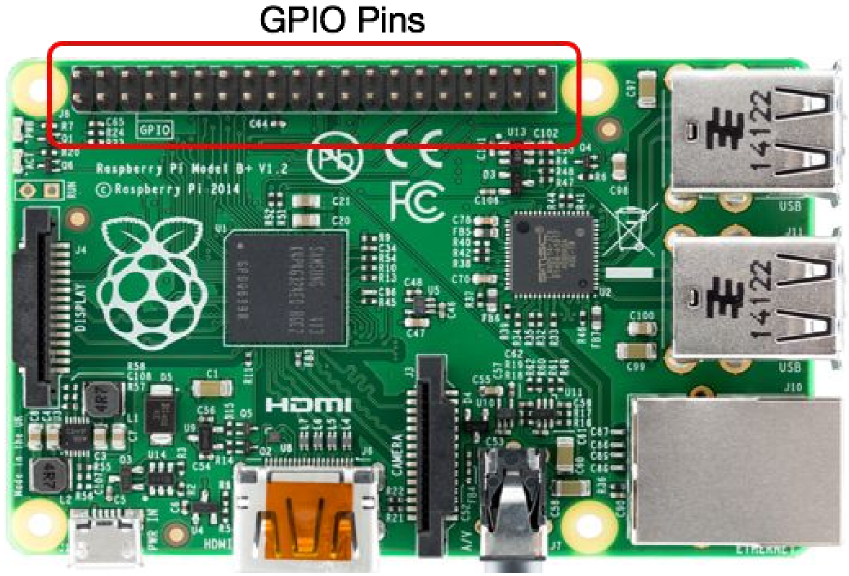
Memory Layout



GPIO

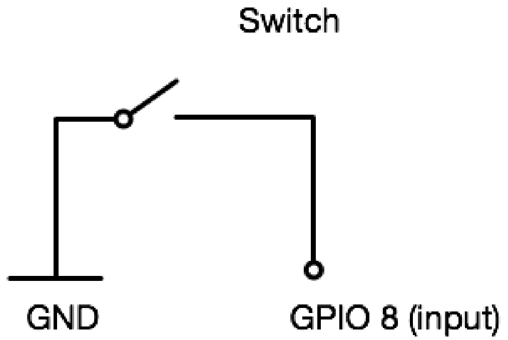
- General-Purpose I/O are generic pins on the Raspberry Pi that can be controlled by the user at run time.

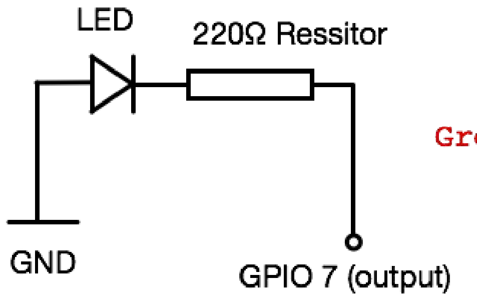
- Can be configured for input or output.
- Input:
  - Button, temperature, accelerometer, carbon-monoxide, etc.
- Output:
  - LED, relais, motor, etc.

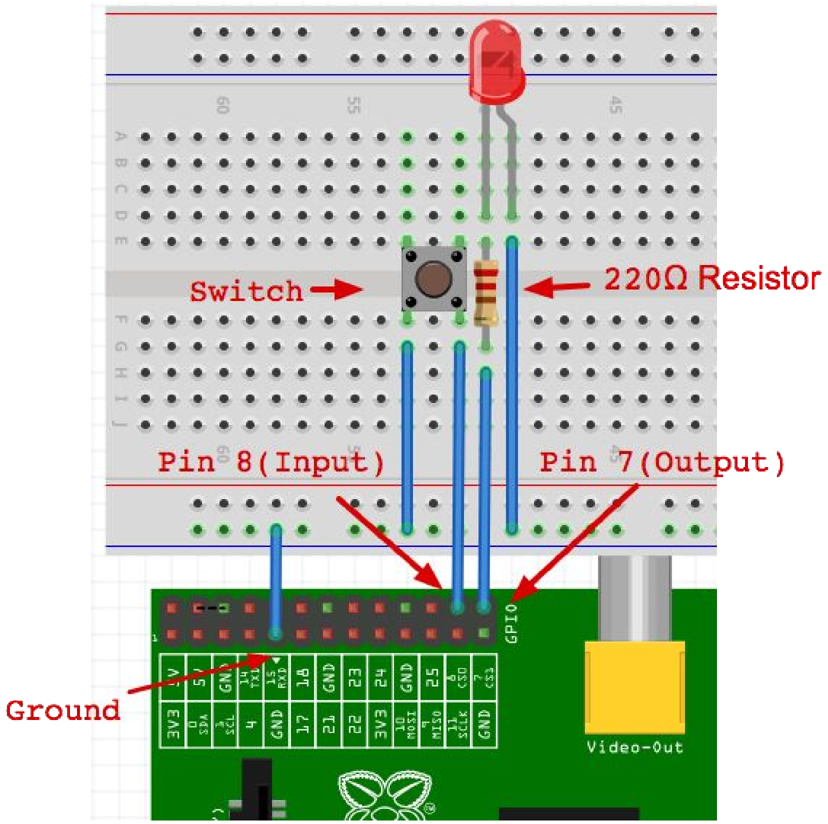


GPIO Hardware side (LED)

- Using resistors to reduce the current flow, high current may damage the LED.







GPIO software side

```c
typedef struct {
    unsigned volatile int pud      : 2;
    unsigned volatile int unused   : 30;
} GPPUD_REG;

typedef struct {
    unsigned volatile int pin0_pin7   : 8;
    unsigned volatile int pin8        : 1;
    unsigned volatile int pin9_pin31 : 23;
} GPPUDCLK_REG;

typedef struct {
    unsigned volatile int pin0_pin6 : 21;
    unsigned volatile int pin7      : 3;
    unsigned volatile int pin8      : 3;
    unsigned volatile int pin9      : 5;
} GPFSEL_REG;


typedef struct {
    unsigned volatile int pin0_pin7   : 8;
    unsigned volatile int pin8        : 1;
    unsigned volatile int pin9_pin31 : 23;
} GPLEV_REG;


typedef struct {
    unsigned volatile int pin0_pin6   : 7;
    unsigned volatile int pin7        : 1;
    unsigned volatile int pin8_pin31 : 24;
} GPSET_GPCLR_REG;
```

```c
// GPIO Base address
int* gpio_addr = (int *) 0x20200000;
const int PIN_IN=8, PIN_OUT=7;
const int INPUT_FUNC = 0, OUTPUT_FUNC = 1;
// SET_OFFSET = 0x1C CLR_OFFSET = 0x28 ... etc
const int SET_OFFSET=7, CLR_OFFSET=10, GPLEV_OFFSET=13;
const int GPPUD_OFFSET=37, GPPUDCLK_OFFSET=38;

void wait_150_cycles() {
    int i;
    for (i=0; i < 150; i++) {
          asm volatile("nop");
    }
}
```

GPIO software side

```c
int kernel_main() {
    // Step 1. Set default input pin 8 to high.
    // GPIO Pull-up/down register: Set bit 1-0 to 10 to enable default input to HIGH
    ((GPPUD_REG *)(gpio_addr + GPPUD_OFFSET))->pud = 2;
    wait_150_cycles();      // According BCM2835 manual, wait 150 cycles before next step.
    // GPIO Pull-up/down clock register: Write the previous control value to Pin 8
    ((GPPUDCLK_REG *)(gpio_addr + GPPUDCLK_OFFSET))->pin8 = 1;
    wait_150_cycles();
    // Cleanup two registers
    ((GPPUD_REG *)(gpio_addr + GPPUD_OFFSET))->pud = 0;
    ((GPPUDCLK_REG *)(gpio_addr + GPPUDCLK_OFFSET))->pin8 = 0;

    // Step 2. Enable input for pin 8, output for pin 7
    ((GPFSEL_REG *) gpio_addr)->pin7 = 1;
    ((GPFSEL_REG *) gpio_addr)->pin8 = 0;

    // Step 3. Using switch to turn on/off led.
    while (1) {
        if (((GPLEV_REG *)(gpio_addr + GPLEV_OFFSET))->pin8 == 0) { // Check Pin 8's level
            ((GPSET_GPCLR_REG *)(gpio_addr + SET_OFFSET))->pin7 = 1; // If Pin 8 is high, turn Pin 7 on to turn on LED
        } else {
            ((GPSET_GPCLR_REG *)(gpio_addr + CLR_OFFSET))->pin7 = 1; // If Pin 8 is low, turn Pin 7 off to turn off LED
        }
    }
}
```

Boot Sequence

Root folder

```
/boot
    |- bootcode.bin
    |- cmdline.txt
    |- config.txt
    |- kernel.img
    |- start.elf
    |- ...
```

bootcode.bin:

Contains ARM code. Is 2nd stage bootloader that enables RAM. It is a Broadcom proprietary image.

cmdline.txt:

Contains command line parameters for kernel.img.

config.txt:

Contains hardware configuration details. Used to alter the default values of video mode, alter system clock speeds, voltages, etc. at boot time.

start.elf:

GPU binary firmware image.

kernel.img:

The image of the OS to load into RAM, i.e., the TOS image. It has ARM executable code.

Step I

- On power on GPU runs first.
- Main CPU is off and is held in inactive state.
- GPU hardwired to read instructions from ROM on boot.
- GPU starts executing instructions from ROM (Also called as 1st stage bootloader)
- ROM has instructions to activate the SD Card.
- GPU searches for ‘bootcode.bin’ on SD Card and copies it in the cache memory (L2 Cache).

Step II

- ‘bootcode.bin’ is a stage 2 bootloader.
- ‘bootcode.bin’ has enough intelligence to activate the RAM.
- GPU the searches for ‘start.elf’ which is stage 3 bootloader.
- GPU turns on the the RAM and copies ‘start.elf’ to 0x00000000.
- ‘start.elf’ has default Raspberry Pi configurations and settings.
- GPU reads other files such as ‘config.txt’ which has configuration settings to override the default configuration of the Raspberry Pi.

NOTE:
Much of the Broadcom and Raspberry Pi functionality is proprietary and only available under an NDA.

Step III

- GPU reads ‘cmdline.txt’ if available which contains the command line parameters for the kernel that is to be loaded.
- GPU loads ‘kernel.img’ at 0x8000 and turns on main CPU
- Main CPU starts executing the code from 0x8000.
- ‘kernel.img’ contains the OS image (e.g., Raspbian, TOS)

config.txt

- It is a simple text file with ‘.txt’ extension placed in root folder in the SD card. It is used to configure the Raspberry Pi during the boot process.
- **Memory Division options** - Allows partition of memory among CPU and GPU. Default is 64MB for GPU and rest for CPU.
- **Audio** - disable or enable 3.5mm audio jack.
- **Video Mode** - to change the default display settings and output.
- **Overclocking Options** - alter default frequencies of CPU, GPU, RAM and Voltages.
- **Conditional Filters** - provide options to filter setting for different models of raspberry pi and serial numbers.

Sample config.txt

```
# Uncomment if you get no picture on HDMI for a default "safe" mode
#hdmi_safe=1

# Uncomment this if your display has a black border of unused pixels visible
# and your display can output without overscan.
#disable_overscan=1

# Uncomment the following to adjust overscan. Use positive numbers if console
# goes off screen, and negative if there is too much border.
overscan_left=20
overscan_right=20
overscan_top=20
overscan_bottom=20

# Uncomment to force a console size. By default it will be display's size minus
# overscan.
#framebuffer_width=1280
#framebuffer_height=720
.
.
.
```

TOS Compile Process (Object files)

- Generate object files (eg. main.o)

```
arm-none-eabi-gcc -Wall -nostdinc -I./include -g -fomit-frame-pointer
-fno-defer-pop -mcpu=arm1176jzf-s -c source/main.c -o build/main.o
```

Option	Description
`-g`	Enable debug information
`-Wall`	Show warning messages
`-nostdinc`	Do not search standard system directories for header files
`-fomit-frame-pointer`	Do not keep frame pointer in a register
`-fno-defer-pop`	Always pop the arguments to each function call as soon as that function returns
`-mcpu=arm1176jzf-s`	Raspberry Pi 1 model B uses CPU ARM1176JZF-S

TOS Compile Process (kernel.img)

- ARM does not have assembly instructions for division (/) and mod (%) operations.
- We need arm-none-eabi-gcc to provide the library which implements division and mod, then pass the link option to ld through -Wl option.
- Highlighted options are the options send to linker.
- Kernel image starts at 0x8000, stack address start at 0xA00000
- -nostartfiles means do not use standard system startup files when linking

```
arm-none-eabi-gcc -nostartfiles build/process.o build/main.o -o
build/output.elf -Wl,--section-start,.init=0x8000,--section-start,.
stack=0xA00000,-Map=kernel.map
```

Debug (QEMU side)

- QEMU allows us to debug the kernel. Run the following command to start QEMU:

```
qemu-system-arm -s -S -M raspi -cpu arm1176 -kernel kernel.img
```

QEMU Options	Description
`-s`	Shorthand for -gdb tcp::1234, open a gdbserver on TCP port 1234
`-S`	Do not start CPU at startup(Wait for attach gdb)
`-M raspi`	Select Raspi as Emulated Machine
`-cpu arm1176`	Select CPU
`-kernel kernel.img`	Select kernel image

Debug (GDB side)

- Run follow command to start gdb and connect to QEMU:

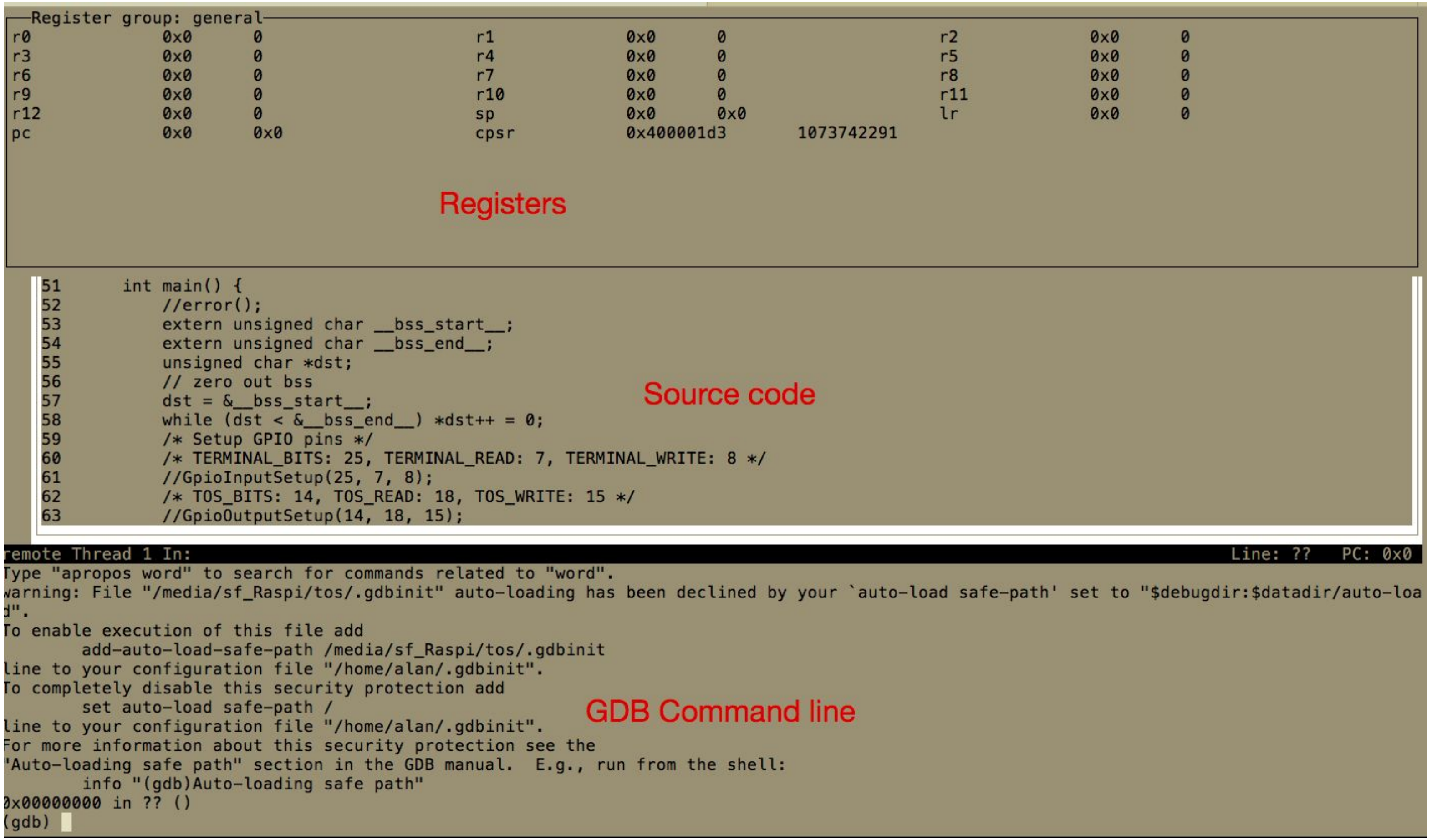
```
arm-none-eabi-gdb -tui -x gdb_cmd
```

GDB Options	Description
`-tui`	Start gdb text UI
`-x gdb_cmd`	Execute gdb command from file gdb_cmd

- Commands in file “gdb_cmd” to setup debug environment:

```
target remote localhost:1234     // Connect to QEMU
symbol-file build/output.elf     // Select symbol file in our tos build folder
layout src                       // Show source code layout
layout regs                      // Show registers
```

GDB text user interface



```
Register group: general
r0             0x0      0                  r1             0x0      0                  r2             0x0      0
r3             0x0      0                  r4             0x0      0                  r5             0x0      0
r6             0x0      0                  r7             0x0      0                  r8             0x0      0
r9             0x0      0                  r10            0x0      0                  r11            0x0      0
r12            0x0      0                  sp             0x0      0x0                lr             0x0      0
pc             0x0      0x0                cpsr           0x400001d3       1073742291
```

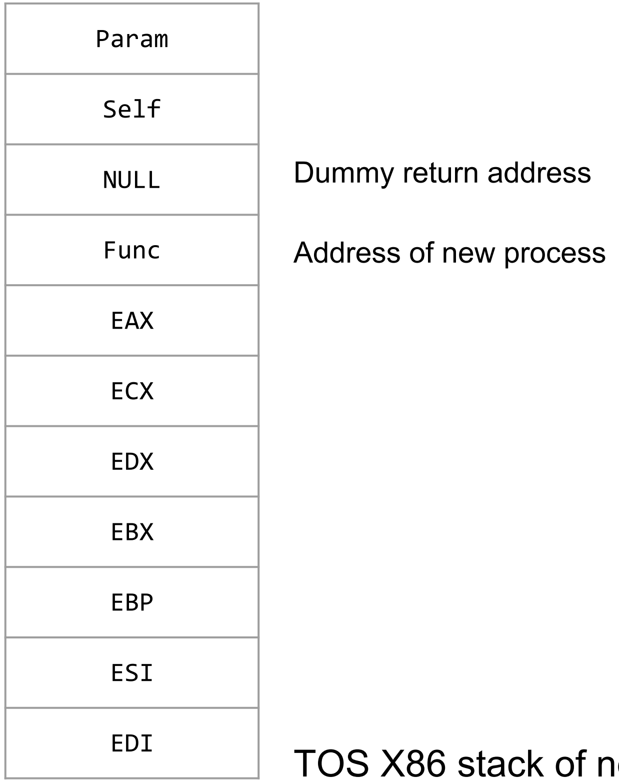
```
        int main() {
            //error();
            extern unsigned char __bss_start__;
            extern unsigned char __bss_end__;
            unsigned char *dst;
            // zero out bss
            dst = &__bss_start__;
            while (dst < &__bss_end__) *dst++ = 0;
            /* Setup GPIO pins */
            /* TERMINAL_BITS: 25, TERMINAL_READ: 7, TERMINAL_WRITE: 8 */
            //GpioInputSetup(25, 7, 8);
            /* TOS_BITS: 14, TOS_READ: 18, TOS_WRITE: 15 */
            //GpioOutputSetup(14, 18, 15);
```

```
remote Thread 1 In:                                                                                  Line: ??   PC: 0x0
Type "apropos word" to search for commands related to "word".
warning: File "/media/sf_Raspi/tos/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-loa
d".
To enable execution of this file add
        add-auto-load-safe-path /media/sf_Raspi/tos/.gdbinit
line to your configuration file "/home/alan/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/alan/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
0x00000000 in ?? ()
(gdb)
```

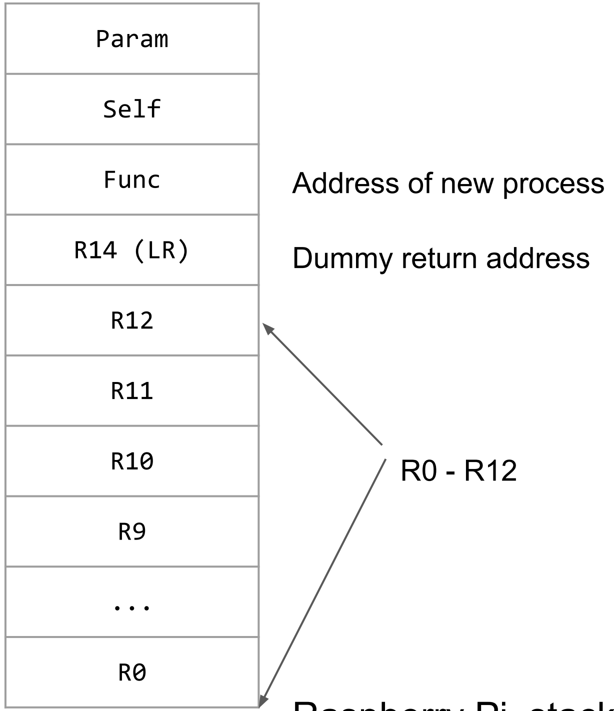
Context Switch

create_process()

- Stack frames look similar between TOS on x86 and TOS for the Raspberry Pi



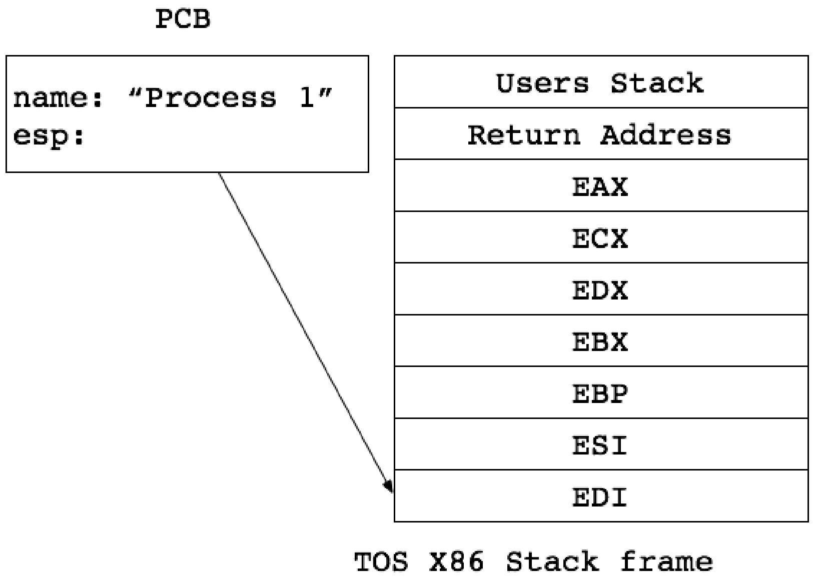
TOS X86 stack of new process

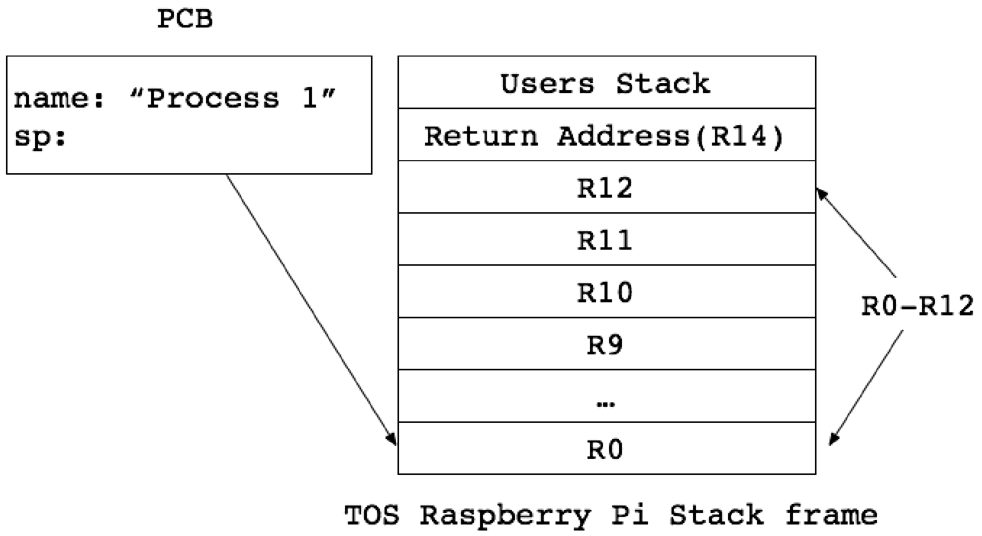


Raspberry Pi stack of new process

resign()

- Stack frame looks similar between TOS on x86 and TOS on Raspberry Pi.
- On Raspberry Pi, the stack layout will change when supporting interrupts.





Stack frame of Process 1 after it call resign()

Resign() implementation

```
/* Use dispatcher_impl() helper function, so all code in resign() is in assembly
 * GCC compile won't add push/pop code around the function.
 */
void dispatcher_impl() {
    active_proc = dispatcher();
}


void resign() {
    asm("push {r0-r12, r14}");
    /* Set active process */
    asm("mov %[old_sp], %%sp" : [old_sp] "=r"(active_proc->sp) :);
    asm("bl dispatcher_impl");
    asm("mov %%sp, %[new_sp]" : : [new_sp] "r"(active_proc->sp));
    asm("pop {r0-r12, pc}");
}
```

Exceptions/Interrupts

ARM Exceptions

- Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction.
- The processor state just before handling the exception is normally preserved so that the original program can be resumed when the exception routine has completed. More than one exception can happen at the same time.
- The ARM architecture supports seven types of exceptions. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of the exception. These fixed addresses are called the **exception vectors.**

Exception Types

Exception type	Mode	Address	Description
Reset	Supervisor	`0x00000000`	When the Reset input is asserted on the processor, the ARM processor immediately stops execution of the current instruction.
Undefined Instruction	Undefined	`0x00000004`	If an attempt is made to execute an instruction that is UNDEFINED, an Undefined Instruction exception occurs.
Software Interrupt (SWI)	Supervisor	`0x00000008`	The Software Interrupt instruction (SWI) enters Supervisor mode to request a particular supervisor (operating system) function.
Prefetch Abort	Abort	`0x0000000C`	Fetch an instruction from an illegal address, the instruction is flagged as invalid.
Data Abort	Abort	`0x00000010`	A data transfer instruction attempts to load or store data at an illegal address.
IRQ (interrupt)	IRQ	`0x00000018`	The IRQ exception is generated externally by asserting the IRQ input on the processor.
FIQ (fast interrupt)	FIQ	`0x0000001C`	The FIQ exception is generated externally by asserting the FIQ input on the processor.

Exception Vectors structure

Exception Vectors need to be copied to address 0x00000000

```
.section .text
_vectors:
        ldr pc, reset_addr      // 0x0
        ldr pc, undef_addr      // 0x4
        ldr pc, swi_addr        // 0x8
        ldr pc, prefetch_addr   // 0xC
        ldr pc, abort_addr      // 0x10
        ldr pc, reserved_addr   // 0x14
        ldr pc, irq_addr        // 0x18
        ldr pc, fiq_addr        // 0x1C

reset_addr:     .word reset_handler
undef_addr:     .word undef_handler
swi_addr:       .word swi_handler
prefetch_addr:  .word prefetch_handler
abort_addr:     .word abort_handler
reserved_addr:  .word reset_handler
irq_addr:       .word master_isr
fiq_addr:       .word fiq_handler
_endvectors:
```

Can not use "ldr pc, reset_handler" here, since function reset_handler() might be far away from here that exceed the offset limitation.

Store 32-bit function address

Setup Exception Vectors

```
.section .init
       // After kernel.img loaded, processor is in supervisor mode
        ldr r0, =_vectors   // Exception Vectors start address in previous slides
        mov r1, #0x0000     // Destination address 0x0

       // Copy Exception vectors (64 Byte) to memory start from 0x00000000
        ldmia r0!, {r2-r9}  // Load 32 bytes starting from _vectors into register r2-r9
        stmia r1!, {r2-r9}  // Store register r2-r9 to memory 0x00000000.
        ldmia r0!, {r2-r9}  // Repeat for next 32 bytes
        stmia r1!, {r2-r9}

        cpsid i, #0x1F           // Change CPSR to System mode (0x1F)
        mov sp, #0xA00000        // Set up Stack pointer for SYS mode

        b kernel_main            // call kernel main function
```

Exceptions

- When an exception happens, CPU will enter corresponding mode by changing 5 bits in the CPSR[4:0] register.
- Many exceptions are synchronous events related to instruction execution in the OS kernel. However, asynchronous events cause the following exception to occur:
  - Reset
  - Interrupts (FIQ, IRQ, software interrupts)

Exception Priorities

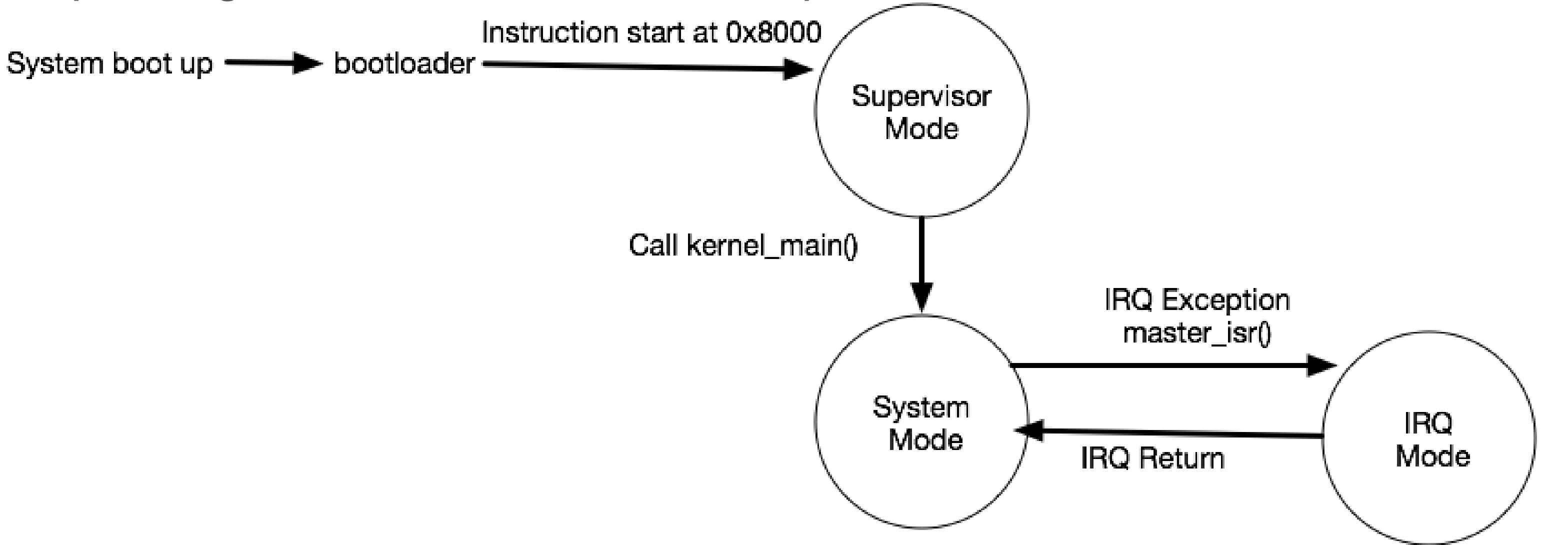
- If two or more exceptions occur simultaneously, the exceptions are dealt with in the fixed order given in the table.
- Reset is the only non-maskable event in ARM architecture.
- FIQ has a higher priority than IRQ, it used in a way analogous to the non-maskable (NMI) interrupt found on other processors like x86, although FIQ can be masked in ARM.

Priority	Exception
Highest: 1	Reset
2	Data Abort
3	FIQ
4	IRQ
6	Prefetch Abort
Lowest:7	Undefined instruction. Software interrupt (SWI)

- ARM instructions do not support division, gcc provide library to do the division, when divide-by-zero happened, it returns 2^{31}-1 rather than raise an exception.

Interrupts in ARM

- Interrupt (IRQ) is an type of exception in ARM.
- All exceptions in ARM are handled by different functions. reset_handler() for Reset exception, master_isr() for IRQ exception, etc...
- All IRQs will go through master_isr() at address 0x00000018.
- We need to implement master_isr() by ourselves, in master_isr(), we check IRQ pending bits to determine which irq subroutine we need to execute.



Interrupts/ISR initialize

```c
/* Initialize Interrupts */
void init_interrupts(void) {
    int i;

    for (i = 0; i < INTERRUPTS_NUMBER; i++) {
        interrupts_table[i] = NULL;
    }
    init_timer();

    // Enable IRQ bit in CPSR
    asm("mrs r0, cpsr");
    asm("bic r0, r0, #0x80");
    asm("msr cpsr, r0");
}
```

```c
#define TIMER_IRQ            0
#define ARM_TIMER_BASE       0x2000B400
#define ENABLE_BASIC_IRQS    0x2000B218
#define INTR_TIMER_CTRL_32BIT  (1 << 1) // Use 32-bit counter
#define INTR_TIMER_CTRL_ENABLE (1 << 7) // Timer Enabled
#define INTR_TIMER_CTRL_INT_ENABLE (1 << 5) // Enable Timer interrupt
#define INTR_TIMER_CTRL_PRESCALE_1 (0 << 2) // Pre-scal is clock/1

/* Init timer */
void init_timer(void) {
     int* enable_basic_irqs = (int *) (ENABLE_BASIC_IRQS);
     int* arm_timer_load = (int *) (ARM_TIMER_BASE);
     int* arm_timer_ctrl = (int *) (ARM_TIMER_BASE + 0xC);

    /* Setup timer interrupt service routine (ISR) */
    interrupts_table[TIMER_IRQ] = isr_timer;

    // Enable receive timer interrupt IRQ
    *(enable_basic_irqs) = 1 << (TIMER_IRQ);

    // Get Timer register address, based on BCM2835 document 14.2
    // Setup Timer frequency around 1kHz
    // Get timer load to 1024
    *(arm_timer_load) = 0x400;

    // Enable Timer, send IRQ, no-prescale, use 32-bit counter
    *(arm_timer_ctrl) = INTR_TIMER_CTRL_32BIT |INTR_TIMER_CTRL_ENABLE
            |INTR_TIMER_CTRL_INT_ENABLE | INTR_TIMER_CTRL_PRESCALE_1;
}
```

Interrupt Controller

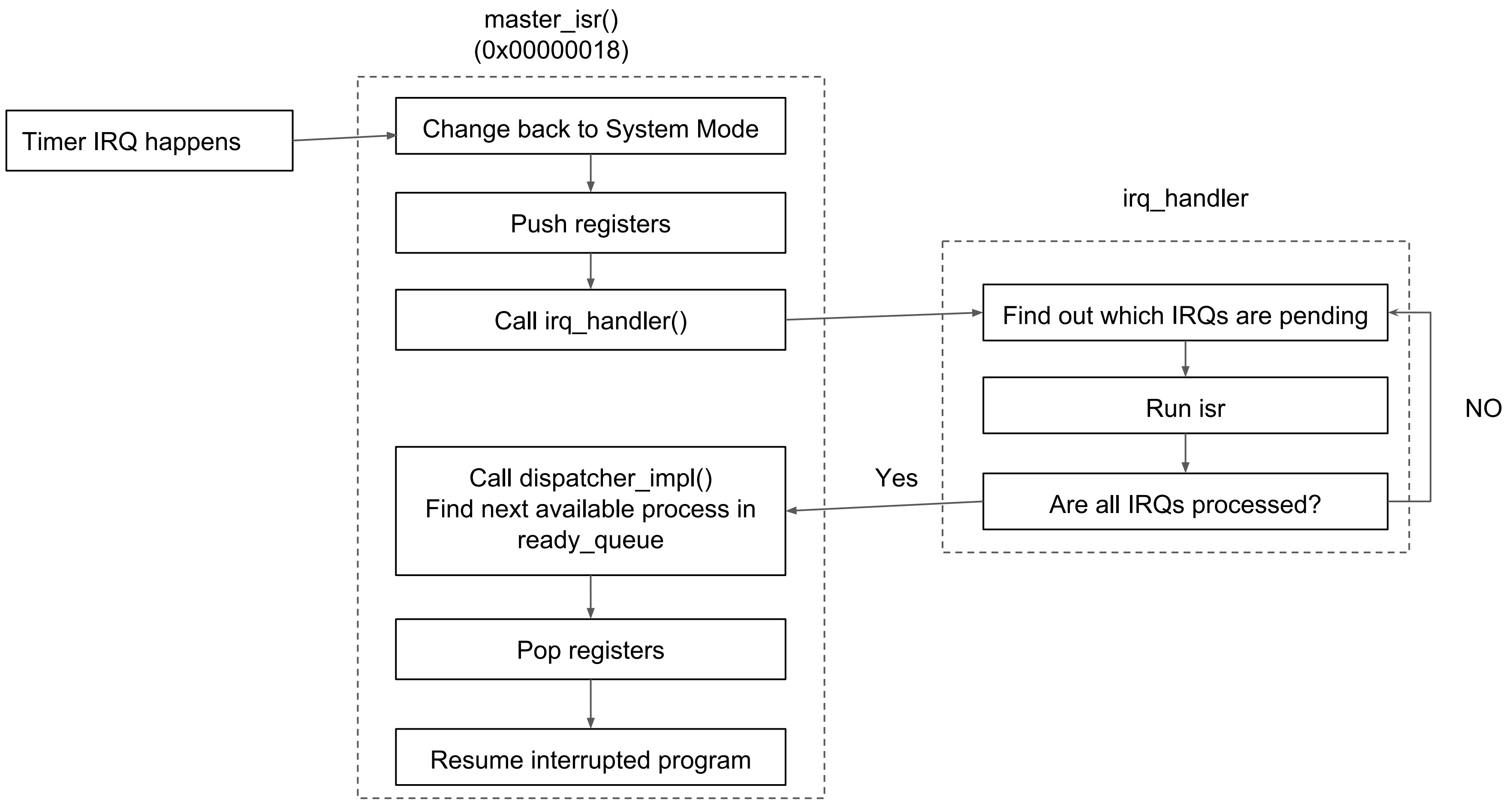
- ARM architecture does not have I/O ports, all communication with hardware is done via memory-mapped I/O.
- Interrupts are handled by an external Broadcom chip (BCM 2835)
- To communicate with the ARM interrupt controller, read/write to Base Address `0x2000B000 + Offset` (see BCM 2835 Manual Section 7.5)
- When an IRQ happens, the interrupt pending registers in memory mapped address (0x2000B200-0x2000B208) show which IRQ happened.
- E.g., bit 0 off address 0x2000B200 shows the IRQ0 (Timer). In the interrupt handler, if the bit 0 is 1, we need to run isr_timer().

ARM Interrupt Controller

- Offset 0x200 used to determine pending IRQs.
- Offset 0x218 used for enabling IRQs.
- Offset 0x224 used for disabling IRQs.

Address offset	**Usage**
0x200	IRQ basic pending
0x218	Enable Basic IRQs
0x224	Disable Basic IRQs

IRQ Handle Process Flow chart



master_isr() implementation

- When processor went into IRQ mode, it will have its own stack, since we do not want to use stack in IRQ mode, we jump back to system at the beginning of master_isr()

```c
void master_isr(void) {
    // When CPU into IRQ mode, the Link Register(R14) will have value PC+4, where PC is the address
    // of the instruction that was NOT executed because the IRQ took priority. In order to return to
    // the right address(PC), we need to minus R14 by 4
    asm("sub lr, lr, #4");

    // SRS (Store Return State) stores the LR and SPSR of the current mode (IRQ) to the stack in SYS mode
    // 5 bits (0x1f) indicate the SYS mode. "Db" (Decrement Before) suffix means CPU will decrement the stack
    // pointer before storing values onto stack. "!" means after store R14 and SPSR of the IRQ mode into SYS
    // mode stack, update the stack pointer in SYS mode.
    asm("srsdb #0x1f!");
    asm("cpsid i, #0x1f");       // Change CPSR to SYS mode, "i" means disable IRQ. "0x1f" means SYSTEM mode.
    asm("push {r0-r12, r14}");   // Save registers
    asm("mov %[old_sp], %%sp" : [old_sp] "=r"(active_proc->sp) :);    // Save Stack pointer
    asm("bl irq_handler");       // handler IRQs
    asm("bl dispatcher_impl");   // Call dispatcher()
    asm("mov %%sp, %[new_sp]" : : [new_sp] "r"(active_proc->sp));    // Get new process stack pointer

    asm("pop {r0-r12, r14}");    // Restore registers
    // RFE (Return From Exception) loads LR, SPSR on SYS stack into PC and CPSR registers, "ia" (increase after)
    // means increment stack pointer after read from stack. "!" means update value in sp after instruction.
    asm("rfeia sp!");
}
```

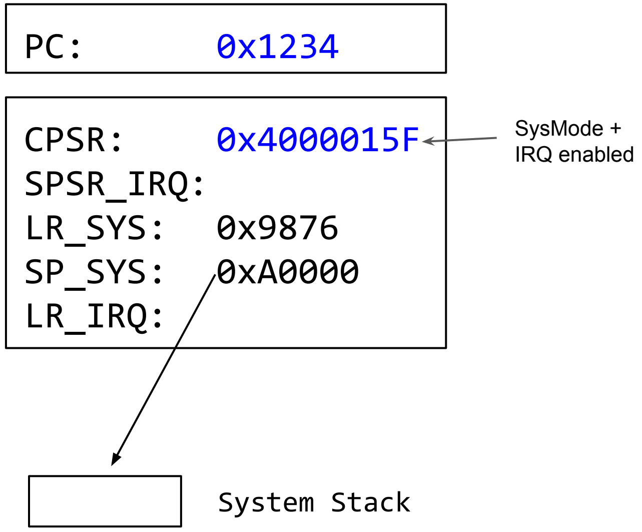
irq_handler() example

```
void irq_handler(void) {
    // IRQ Base Pending address
    volatile unsigned int* irq_address = (unsigned int *) 0x2000B200;

    // Timer is IRQ 0
    unsigned int TIMER_IRQ = 0;
    unsigned int TIMER_IRQ_BIT = 1 << TIMER_IRQ;

    // Handle Timer IRQ
    if (((*irq_address) & TIMER_IRQ_BIT == 1) &&
            (interrupt_table[TIMER_IRQ] != NULL)) {
        // Will call isr_timer(). Should clear Timer IRQ before return
        interrupt_table[TIMER_IRQ]();
     }

     // Handle other IRQs here
}
```

IRQ Handling

System Mode

```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebffffb9   bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```

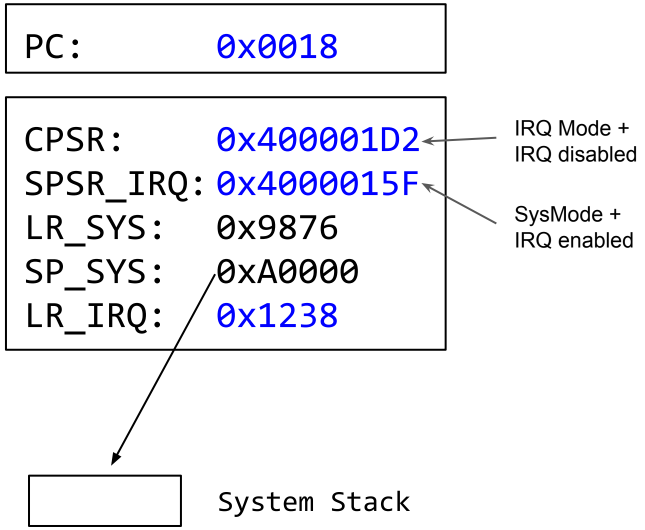


```
PC:         0x1234

CPSR:       0x4000015F
SPSR_IRQ:
LR_SYS:     0x9876
SP_SYS:     0xA0000
LR_IRQ:
```

IRQ Handling

IRQ Mode

```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebffffb9   bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```

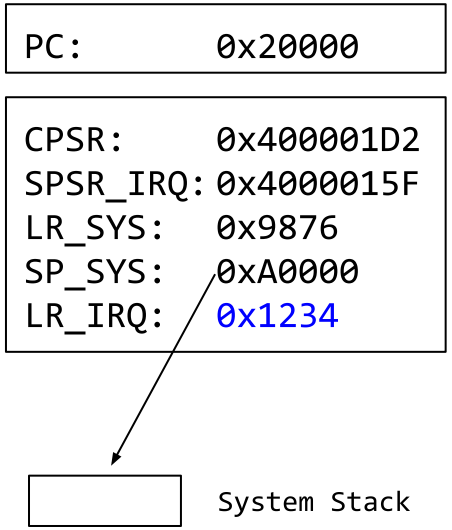


```
PC:        0x0018

CPSR:      0x400001D2
SPSR_IRQ:  0x4000015F
LR_SYS:    0x9876
SP_SYS:    0xA0000
LR_IRQ:    0x1238
```

IRQ Handling

IRQ Mode

```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebffffb9   bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```

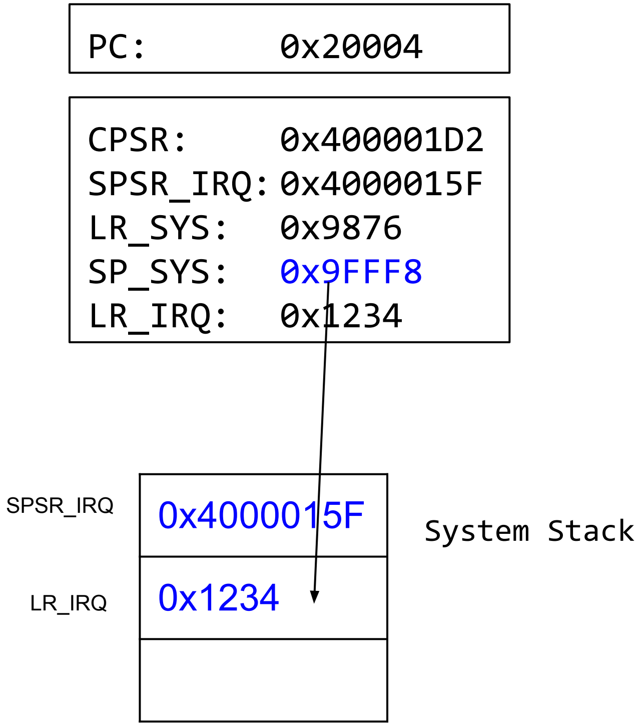


```
PC:        0x20000

CPSR:      0x400001D2
SPSR_IRQ:  0x4000015F
LR_SYS:    0x9876
SP_SYS:    0xA0000
LR_IRQ:    0x1234
```

IRQ Handling

IRQ Mode

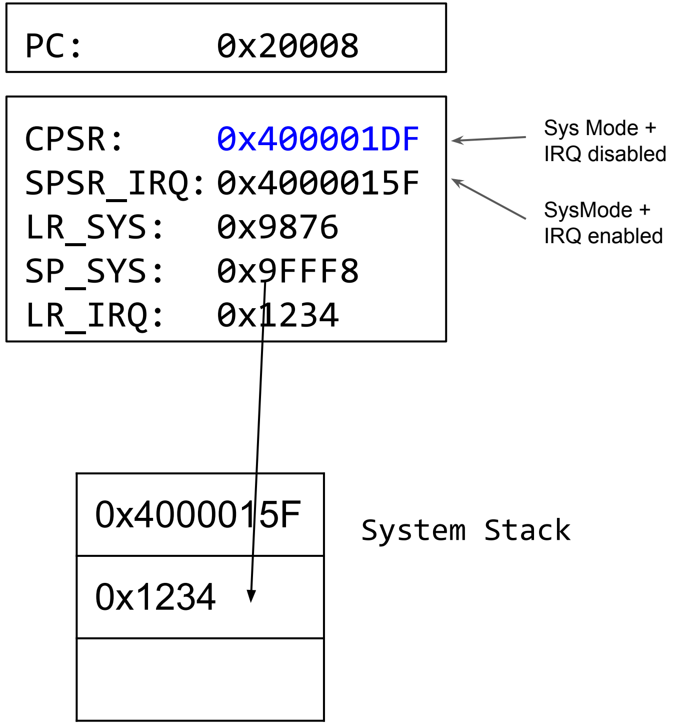
```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb  #0x1f!
20008:      f10e009f   cpsid  i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebffffb9   bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```



IRQ Handling

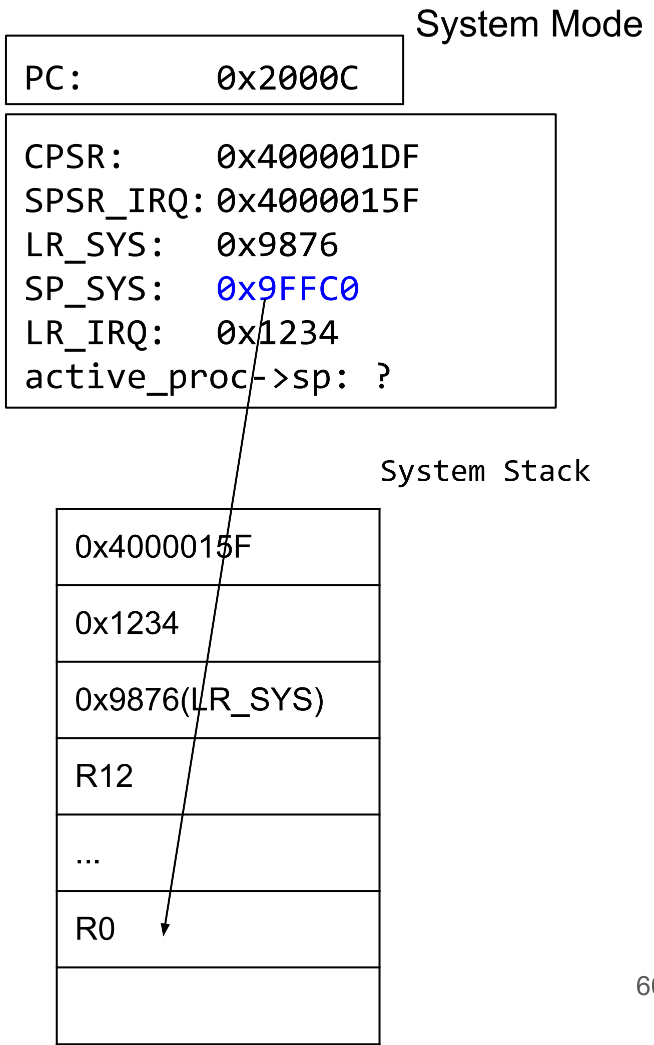
System Mode

```
00018:     e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:     00012070   .word  0x00020000

<master_isr>:
20000:     e24ee004   sub    lr, lr, #4
20004:     f96d051f   srsdb   #0x1f!
20008:     f10e009f   cpsid   i, #0x1f
2000c:     e92d5fff   push   {r0-r12, r14}

20010~2001c:          mov    active_proc->sp, r13

20020:     ebffffb9   bl     irq_handler
20024:     ebfff884   bl     dispatcher_impl

20028~20034:          mov    r13, active_proc->sp

20038:     e8bd5fff   pop    {r0-r12, r14}
2003c:     f8bd0a00   rfeia  sp!
```



IRQ Handling

```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebfffb9    bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```



System Mode

```
PC:        0x2000C

CPSR:      0x400001DF
SPSR_IRQ:  0x4000015F
LR_SYS:    0x9876
SP_SYS:    0x9FFC0
LR_IRQ:    0x1234
active_proc->sp: ?
```

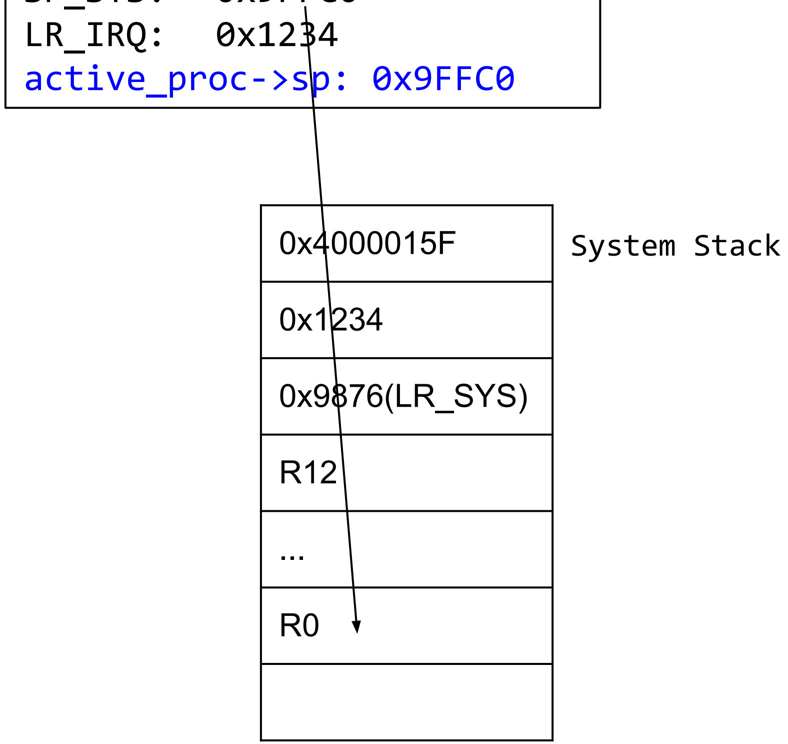
IRQ Handling

System Mode

```
PC:        0x20010~0x2001C
```

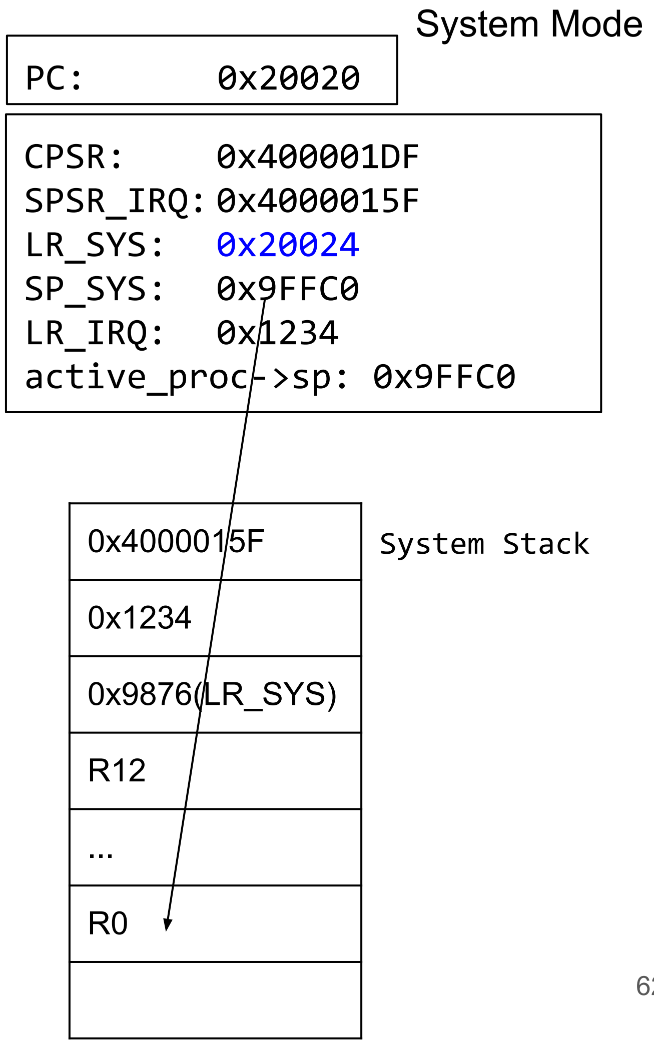
```
CPSR:      0x400001DF
SPSR_IRQ:  0x4000015F
LR_SYS:    0x9876
SP_SYS:    0x9FFC0
LR_IRQ:    0x1234
active_proc->sp: 0x9FFC0
```

```
00018:      e59ff018    ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070    .word  0x00020000

<master_isr>:
20000:      e24ee004    sub    lr, lr, #4
20004:      f96d051f    srsdb   #0x1f!
20008:      f10e009f    cpsid   i, #0x1f
2000c:      e92d5fff    push   {r0-r12, r14}

20010~2001c:            mov    active_proc->sp, r13

20020:      ebffffb9    bl     irq_handler
20024:      ebfff884    bl     dispatcher_impl

20028~20034:            mov    r13, active_proc->sp

20038:      e8bd5fff    pop    {r0-r12, r14}
2003c:      f8bd0a00    rfeia  sp!
```



IRQ Handling

```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebffffb9   bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```



System Mode

```
PC:         0x20020

CPSR:       0x400001DF
SPSR_IRQ:   0x4000015F
LR_SYS:     0x20024
SP_SYS:     0x9FFC0
LR_IRQ:     0x1234
active_proc->sp: 0x9FFC0
```

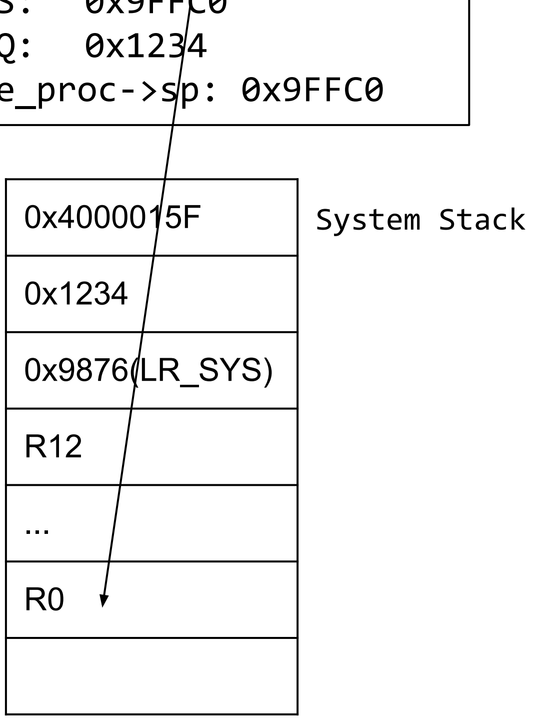
IRQ Handling

System Mode

```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebfffb9    bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```

```
PC:        0x20024
```

```
CPSR:      0x400001DF
SPSR_IRQ:  0x4000015F
LR_SYS:    0x20028
SP_SYS:    0x9FFC0
LR_IRQ:    0x1234
active_proc->sp: 0x9FFC0
```



Assumption: dispatcher_impl does not change active_proc.

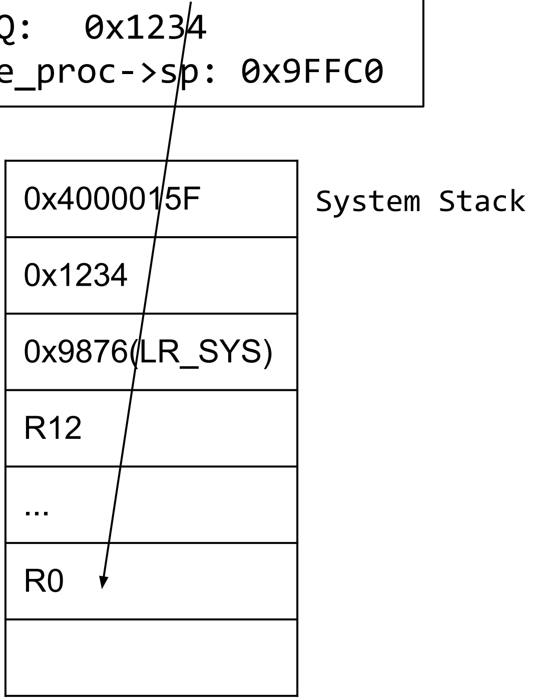
IRQ Handling

System Mode

```
00018:     e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:     00012070   .word  0x00020000

<master_isr>:
20000:     e24ee004   sub    lr, lr, #4
20004:     f96d051f   srsdb   #0x1f!
20008:     f10e009f   cpsid   i, #0x1f
2000c:     e92d5fff   push   {r0-r12, r14}

20010~2001c:          mov    active_proc->sp, r13

20020:     ebffffb9   bl     irq_handler
20024:     ebfff884   bl     dispatcher_impl

20028~20034:          mov    r13, active_proc->sp

20038:     e8bd5fff   pop    {r0-r12, r14}
2003c:     f8bd0a00   rfeia  sp!
```

```
PC:        0x20028~0x20034
```

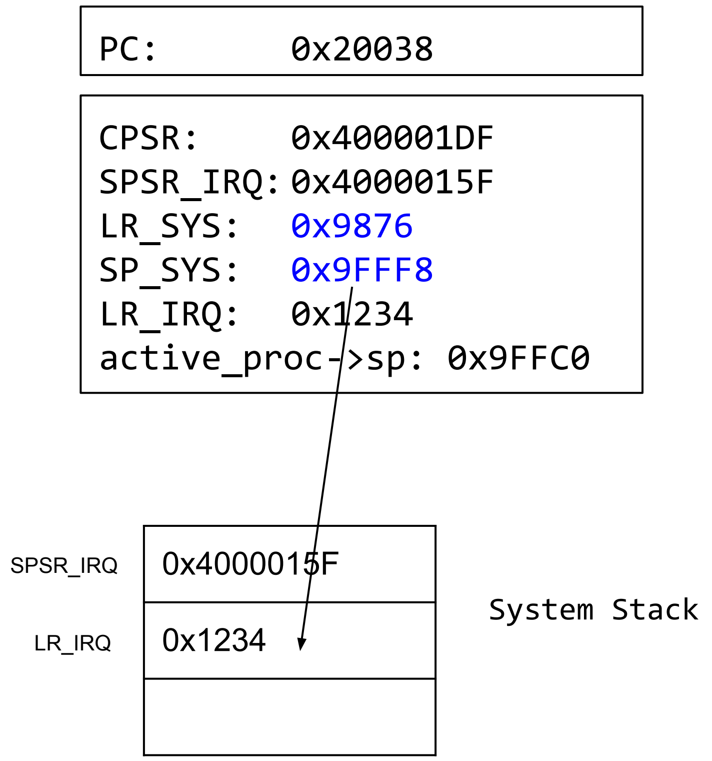
```
CPSR:      0x400001DF
SPSR_IRQ:  0x4000015F
LR_SYS:    0x20028
SP_SYS:    0x9FFC0
LR_IRQ:    0x1234
active_proc->sp: 0x9FFC0
```



IRQ Handling

System Mode

```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebffffb9   bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```

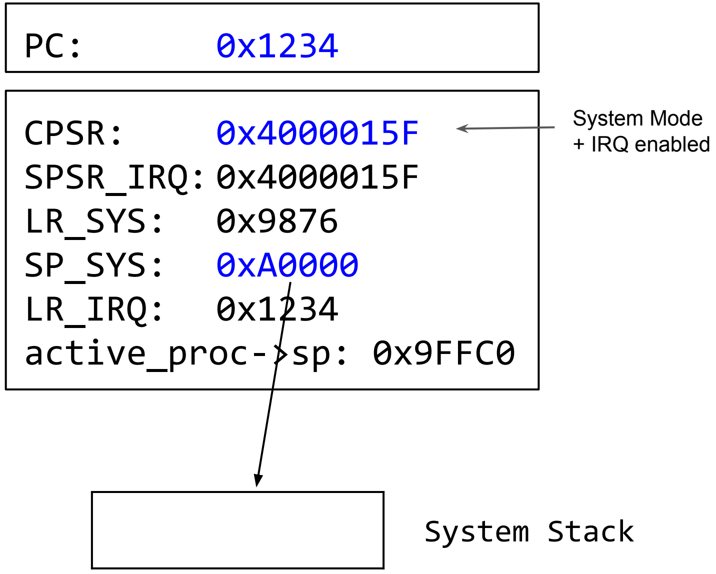


```
PC:        0x20038

CPSR:      0x400001DF
SPSR_IRQ:  0x4000015F
LR_SYS:    0x9876
SP_SYS:    0x9FFF8
LR_IRQ:    0x1234
active_proc->sp: 0x9FFC0
```

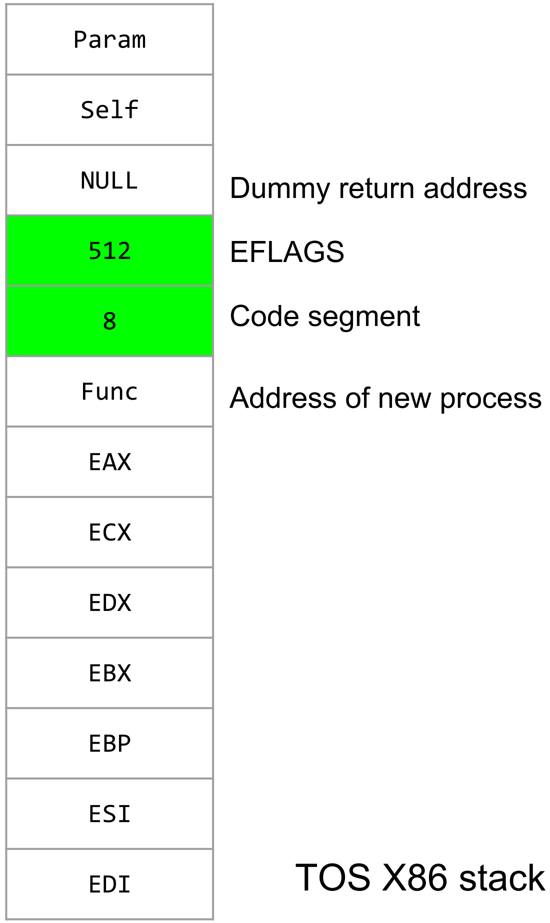
IRQ Handling

System Mode

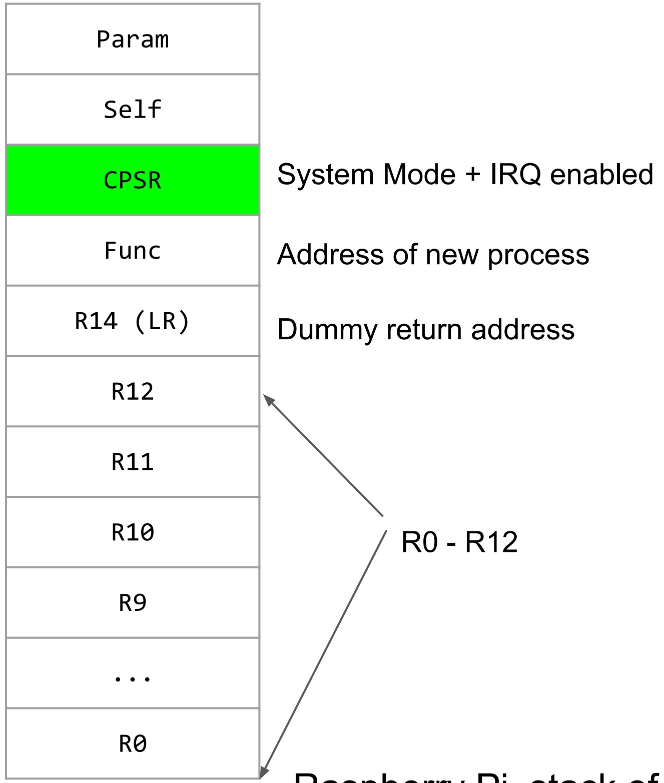
```
00018:      e59ff018   ldr    pc, irq_addr

<irq_addr>:
11e70:      00012070   .word  0x00020000

<master_isr>:
20000:      e24ee004   sub    lr, lr, #4
20004:      f96d051f   srsdb   #0x1f!
20008:      f10e009f   cpsid   i, #0x1f
2000c:      e92d5fff   push   {r0-r12, r14}

20010~2001c:           mov    active_proc->sp, r13

20020:      ebffffb9   bl     irq_handler
20024:      ebfff884   bl     dispatcher_impl

20028~20034:           mov    r13, active_proc->sp

20038:      e8bd5fff   pop    {r0-r12, r14}
2003c:      f8bd0a00   rfeia  sp!
```



create_process() - revised

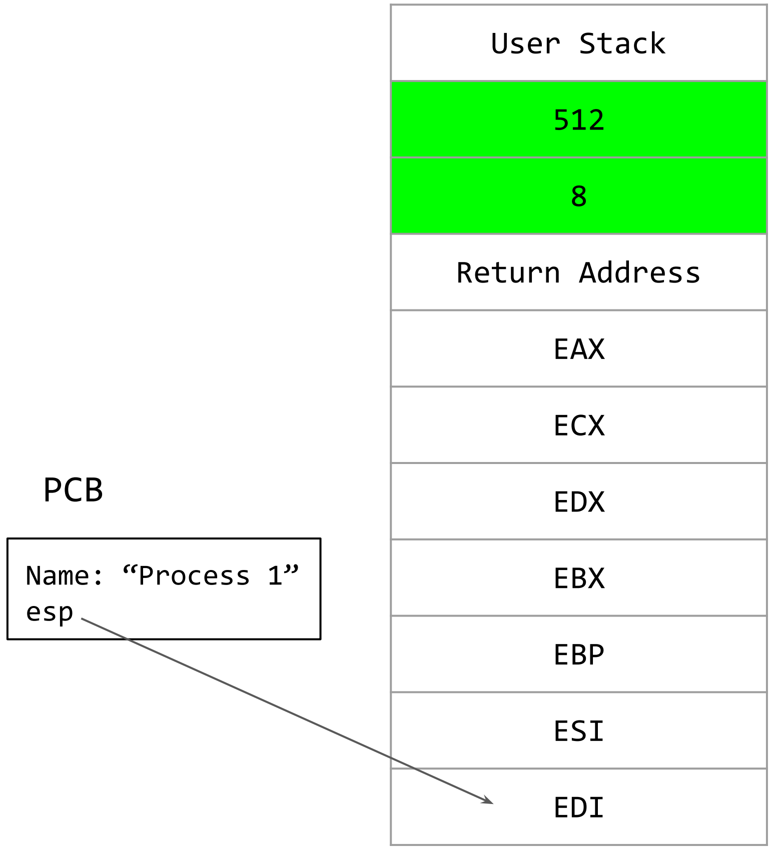


TOS X86 stack of new process

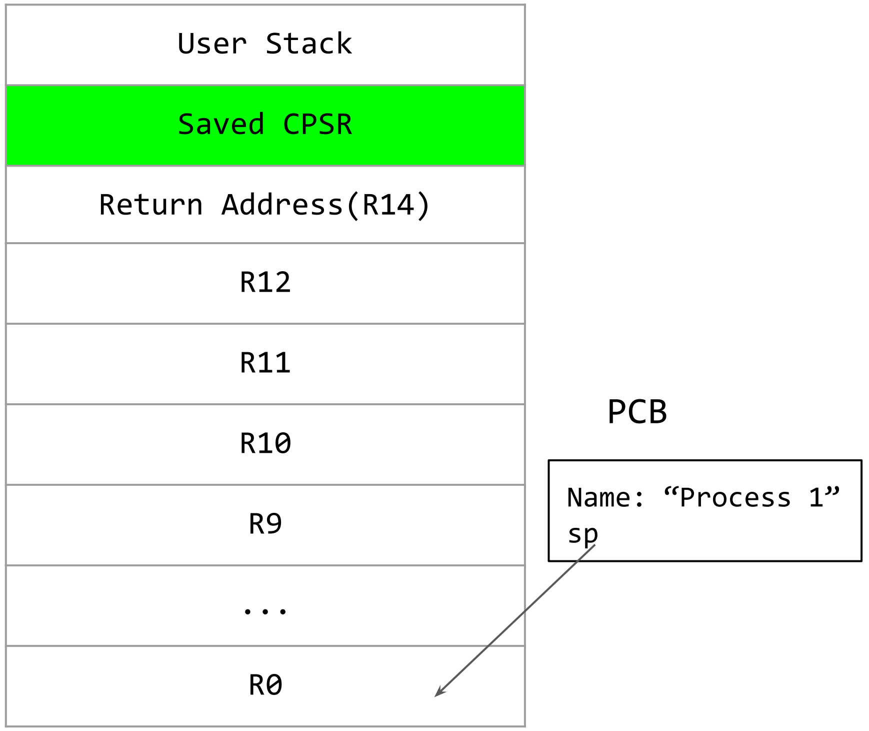


Raspberry Pi stack of new process

resign() - revised



TOS x86 Process stack frame



TOS Raspberry Pi Process stack frame

Standard Output (HDMI)

Framebuffer vs Text Mode

- Text Mode:
  - Content is represented on screen in terms of characters rather than pixels.
  - Lower memory consumption.
  - Fast screen manipulation (using hardware converting character to pixels)
- Framebuffer:
  - A framebuffer is a portion of RAM which contains a bitmap.
  - This bitmap has the complete frame to be displayed on the external display.
  - Content is represented on screen in pixels.
  - Displaying ASCII characters requires font-information.

Framebuffer

- Store each pixel in memory, the more memory we used, the more unique colors we got.

Name	**Unique Colors**	**Memory cost for 1 pixel**
Low colour	256	8 bits(1 Byte)
High colour	65536	16 bits(2 Bytes)
True colour	16777216	24 bits(3 Bytes)

Code to draw a pixel

```c
typedef struct {
    int p_width;      // Physical Width
    int p_height;     // Physical Height
    int v_width;      // Virtual Width (Framebuffer width)
    int v_height;     // Virtual Height (Framebuffer height)
    int gpu_pitch;    // GPU - Pitch
    int bit_depth;    // Bit Depth (High Colour)
    int x; // number of pixels to skip in the top left corner of the screen when copying the framebuffer to screen
    int y;
    int gpu_pointer; // Point to the frame buffer
    int gpu_size;     // GPU - Size
} FrameBufferInfo;


FrameBufferInfo* graphicsAddress;       // FramebufferInfo was initialized when TOS booted.

short foreground_color = 0xFFFF; // Foreground white color
// Draw a pixel at row y, column x. This function only work for high color(16-bit)
void draw_pixel(int x, int y) {
    int    width;
    short* gpu_pointer;      // Each pixel uses 2 bytes.

    width = graphicsAddress->p_width;       /* Get width */

    /* Compute the address of the pixel to write */
    gpu_pointer = (short *) graphicsAddress->gpu_pointer;
    *(gpu_pointer + (x + y * width)) = foreground_color;  /* Calculate pixel position in memory */
}
```

How to draw characters

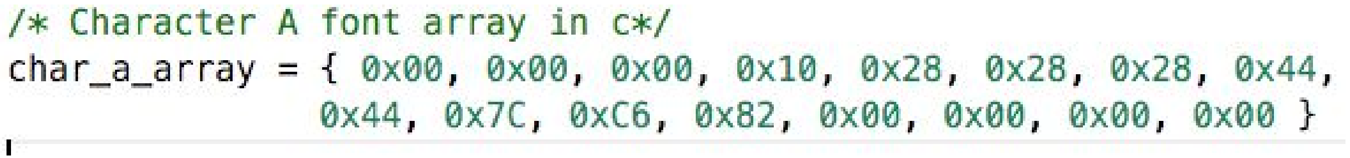
- Bitmap fonts
  - Using binary to describe a character. Just copy it to screen.
  - Easy to implement, but hard to resize for different screen.
  - We use bitmap fonts in TOS
- Vector fonts
  - Describe how to draw a character, e.g. an 'o' could be circle with radius half that of the maximum letter height
  - Perfect at any resolution
  - Hard to implement

8x16 Bitmap fonts

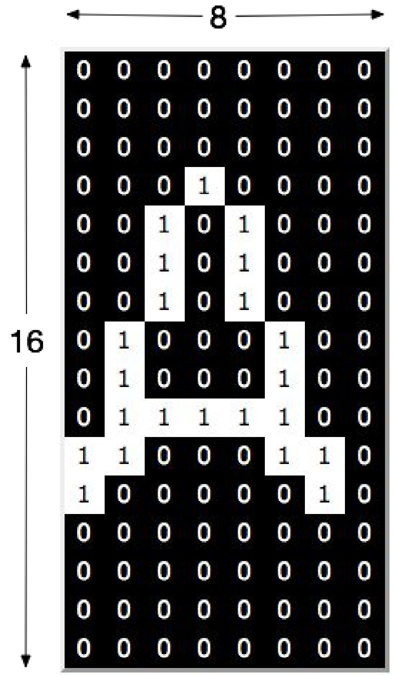
- Example: 'A' character in the monospace, monochrome, 8x16 font Bitstream

  Vera Sans Mono

- Each character cost 16 bytes in font file.
- The 16 bytes in hexadecimal:



```c
/* Character A font array in c*/
char_a_array = { 0x00, 0x00, 0x00, 0x10, 0x28, 0x28, 0x28, 0x44,
                 0x44, 0x7C, 0xC6, 0x82, 0x00, 0x00, 0x00, 0x00 }
```



```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 1 0 0 0
0 0 1 0 1 0 0 0
0 0 1 0 1 0 0 0
0 1 0 0 0 1 0 0
0 1 0 0 0 1 0 0
0 1 1 1 1 1 0 0
1 1 0 0 0 1 1 0
1 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

References

- Raspberry Pi - TOS on github
- Raspberry Pi Bare metal tutorial from University of Cambridge
- ARM Information Center
- Migrating from IA-32 to ARM
- Embedded Xinu
- OSDev
- Raspberry Pi GPIO LED examples
- Raspberry Pi on linux
- Gcc ARM options
- Baking Pi – Operating Systems Development
- Understanding the Raspberry Pi Boot Process
- Raspberry Pi Wiki
- How the Raspberry Pi boots up
- Raspberry PI bare metal programming Part 1: The Boot Process
- Bare Metal Programming in C
- config.txt Documentation

References

- Raspberry Pi ARM based bare metal examples
- Baremetal examples for Raspberry Pi
- Learn Raspberry Pi Programming
- ARM Assembly Language Programming
- ARM assembler in Raspberry Pi
- Low-level Graphics on Raspberry Pi
- Online C to Assembly converter