Modern Assembly Language Programming with the ARM processor

Chapter 14: Running Without an Operating System

No Process or Thread Management

- When running under Linux, the operating system provides mechanisms for running programs (processes) to share the CPU.
- When running under Linux, the operating system provides mechanisms (`clone()`, etc.) for processes to split the work among multiple threads and utilize multiple CPUs.
- When running under Linux, the operating system provides mechanisms (`fork()`, `exec()`, etc.) for processes to start and manage child processes.

On bare metal, the is no support for multiple thread or multiple processes unless the programmer provides them.

All process/thread management is the responsibility of the programmer.
Threads require "context switching."
How can that be done without using Assembly language?

No I/O Functions

- When running under Linux, the operating system provides drivers for all of the I/O devices and system calls to use them. e.g. `open()`, `read()`, `write()`, `close()`, etc.
- The C standard library provides higher-level functions for accessing the system services. e.g. `fopen()`, `fprintf()`, `fscanf()`, `fclose()`, etc.

On bare metal, there are no drivers and no system calls, unless the programmer provides them.

No Stack

- When running under Linux, the operating system sets the initial contents of the `sp` register.

On bare metal, the programmer must choose a memory location for the stack, and initialize the stack pointer.

How can that be done without using Assembly language?

No Dynamic Variables

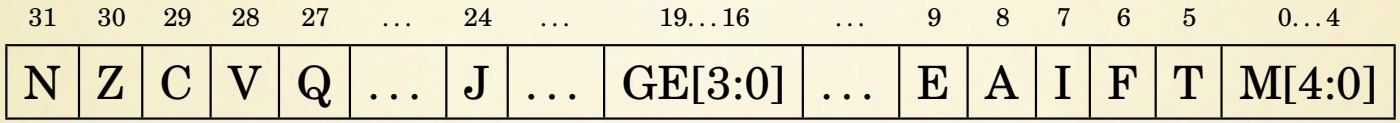
- When running under Linux, the operating system provides a virtual memory map for each running program, and the `sbrk()` system call allows the program to request or release memory.
- The C standard library provides higher-level functions for accessing dynamic memory, e.g. `malloc()`, `free()`, etc.

On bare metal, the is no virtual memory and no dynamic memory management functions, unless the programmer provides them.

All memory management is the responsibility of the programmer.

Arm Processor Modes

Operating system code has special privileges



M[4:0]	Mode	Name	Register Set
10000	usr	User	R0-R14, CPSR, PC
10001	fiq	Fast Interrupt	R0-R7, R8_fiq-R14_fiq, CPSR, SPSR_fiq, PC
10010	irq	Interrupt Request	R0-R12, R13_irq, R14_irq, CPSR, SPSR_irq, PC
10011	svc	Supervisor	R0-R12, R13_svc R14_svc CPSR, SPSR_irq, PC
10111	abt	Abort	R0-R12, R13_abt R14_abt CPSR, SPSR_abt PC
11011	und	Undefined Instruction	R0-R12, R13_und R14_und, CPSR, SPSR_und PC
11111	sys	System	R0-R14, CPSR, PC

ARM System Mode Registers

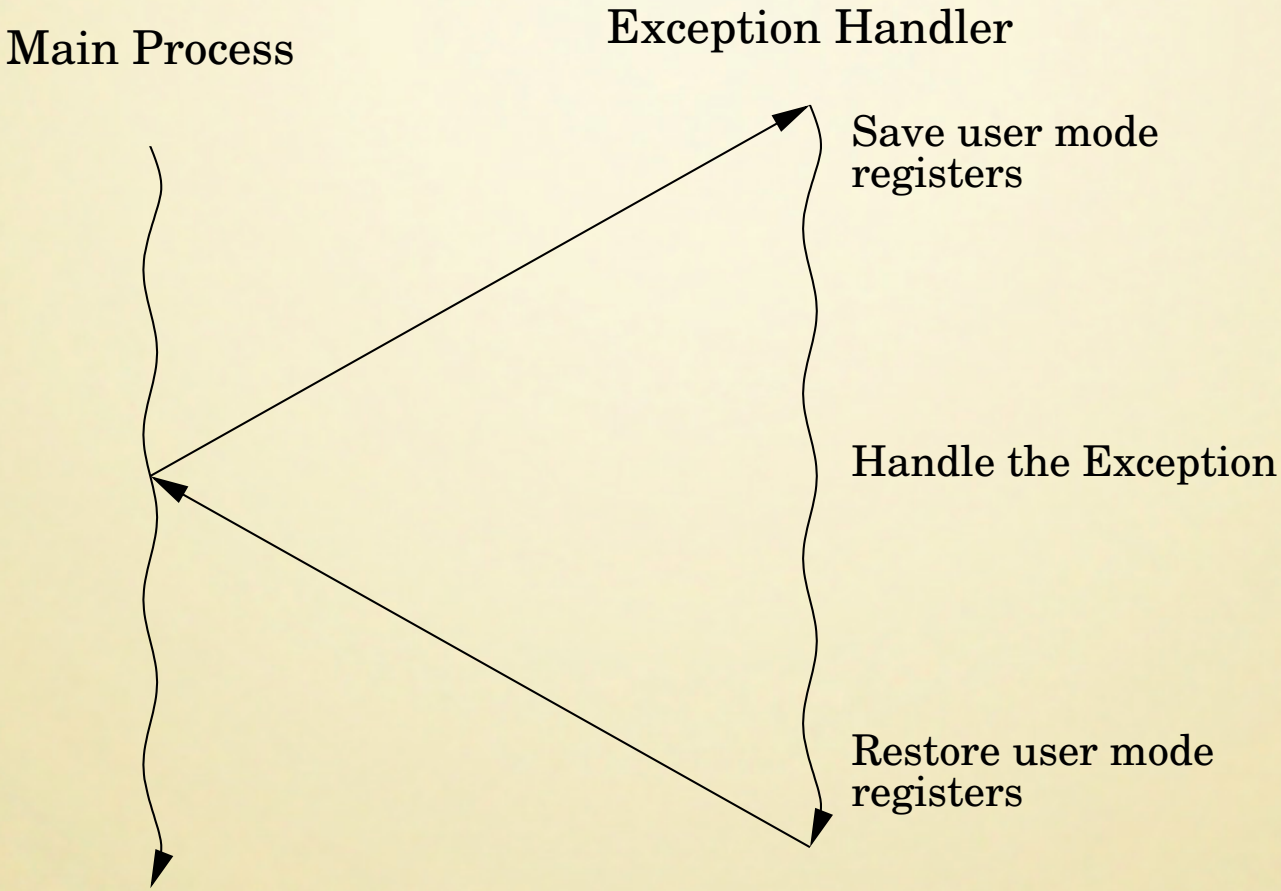
Privileged Modes Have Private Registers

usr sys	svc	abt	und	irq	fiq
r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8					r8_fiq
r9					r9_fiq
r10					r10_fiq
r11 (fp)					r11_fiq
r12 (ip)					r12_fiq
r13 (sp)	r13_svc	r13_abt	r13_und	r13_irq	r13_fiq
r14 (lr)	r14_svc	r14_abt	r14_und	r14_irq	r14_fiq
r15 (pc)					

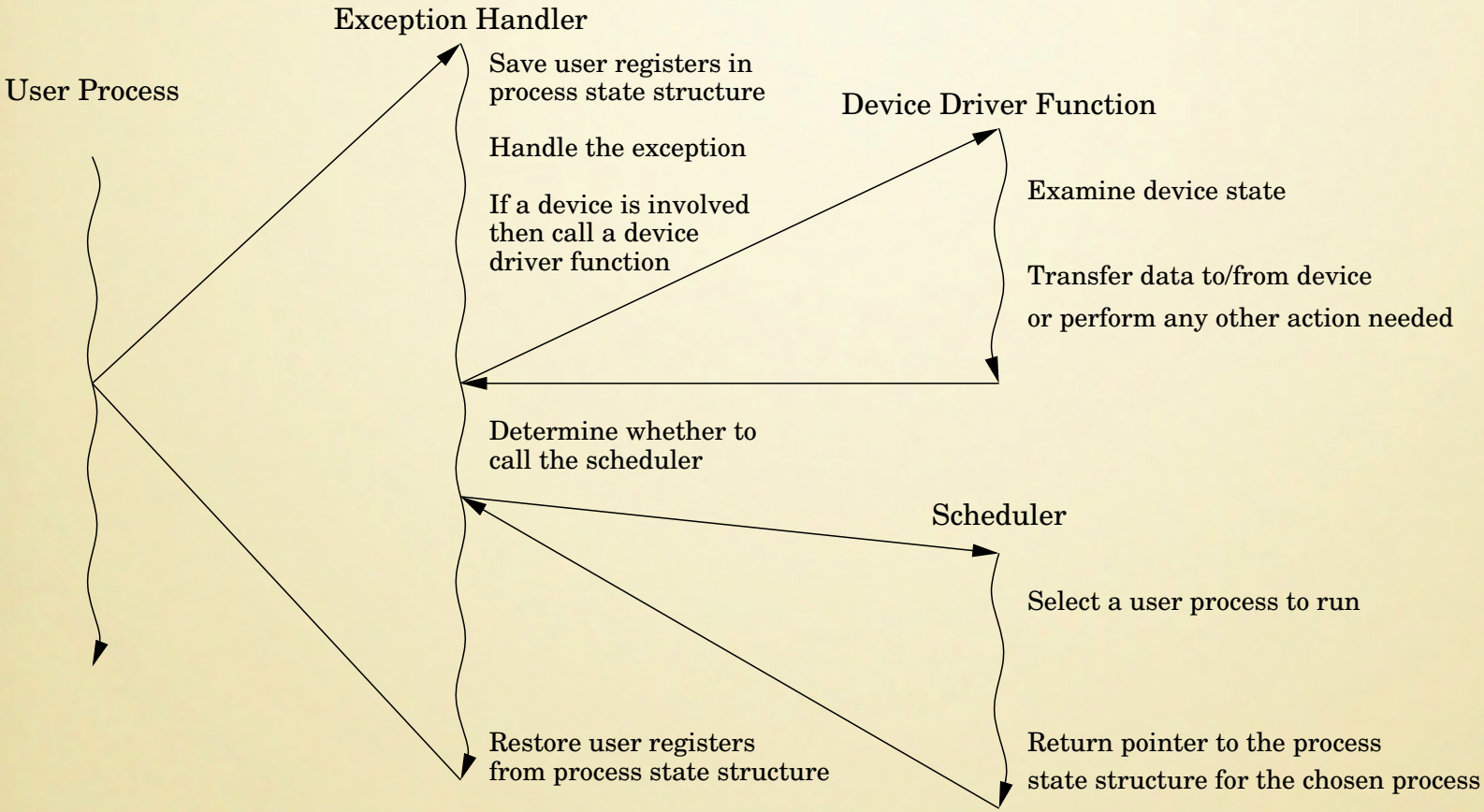
CPSR					
	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Exception and Interrupt Processing

When an exception occurs, the processor goes into privileged mode



Task Switching



ARM Interrupt Vector Table

On most ARM processors, the Interrupt Vector Table (IVT) is at address 0x0, but some versions allow it to be moved to other locations.
The IVT is a table of instructions. When an interrupt or exception occurs, the CPU changes mode and executes one instruction from the IVT.

Address	Exception	Mode
0x00000000	Reset	svc
0x00000004	Undefined Instruction	und
0x00000008	Software Interrupt	svc
0x0000000C	Prefetch Abort	abt
0x00000010	Data Abort	abt
0x00000014	Reserved	
0x00000018	Interrupt Request	irq
0x0000001C	Fast Interrupt Request	fiq

The IVT usually contains branch instructions.

Exception Processing

1. The CPSR is copied into the SPSR for the mode corresponding to the type of exception that has occurred.
2. the CPSR mode bits are changed, switching the CPU into the appropriate priviliged mode.
3. The banked registers for the new mode become active.
4. The I bit of the CPSR is cleared, which disables interrupts.
5. If the exception was an FIQ, or if a reset has occurred, then the FIQ bit is cleared, disabling fast interrupts.
6. The program counter is copied to the link register for the new mode.
7. The program counter is loaded with the address in the vector table corresponding with the exception that has occurred.
8. The processor then fetches the next instruction using the program counter as usual. However, the program counter has been set so that in loads an instruction from the vector table.

Startup Code

Write a function (or set of functions) that will be executed when the system starts.
The startup code will:

- initialize the stack pointer(s),
- initialize the `.bss` section,
- configure CPU and critical systems,
- set up memory management (optional),
- set up process and/or thread structures (optional),
- initialize devices (optional),
- set up interrupt handling (optional),
- call `main()`.

The startup code requires intimate knowledge of the target platform.
Some of the startup code can only be written in Assembly.

Exception Handler Stubs

```
1  @@@ FILE: handlers.S
2          .text
3          .align  2
4  @@@ ------------------------------------------------------------------
5  @@@ On reset, jump to startup function. The CPU must not
6  @@@ be in usr or sys mode! (add some code to check it)
7          .global reset_handler
8  reset_handler:
9          b       _start
10
11 @@@ ------------------------------------------------------------------
12         .global irq_handler
13 irq_handler:                    @ must subtract 4 from lr
14         stmfd   sp!,{r0-r7, lr}
15         @@ handler body goes here
16         ldmfd   sp!,{r0-r7, lr}
17         subs    pc, lr, #4
18
19 @@@ ------------------------------------------------------------------
20         .global undef_handler
21 undef_handler:                   @ lr holds exact return address
22         stmfd   sp!,{r0-r7, lr}
23         @@ handler body goes here
24         ldmfd   sp!,{r0-r7, lr}
25         movs    pc,lr
26
27 @@@ ------------------------------------------------------------------
28         .global swi_handler
29 swi_handler:                     @ lr holds exact return address
30         stmfd   sp!,{r0-r7, lr}
31         @@ handler body goes here
```

Exception Handler Stubs

```
1         ldmfd   sp!,{r0-r7, lr}
2         movs    pc,lr
3
4   @@@ ---------------------------------------------------------------
5         .global pAbort_handler
6   pAbort_handler:                 @ must subtract 4 from lr
7         stmfd   sp!,{r0-r7, lr}
8         @@ handler body goes here
9         ldmfd   sp!,{r0-r7, lr}
10        subs    pc,lr,#4
11
12  @@@ ---------------------------------------------------------------
13        .global dAbort_handler
14  dAbort_handler:                 @ must subtract 8 from lr
15        stmfd   sp!,{r0-r7, lr}
16        @@ handler body goes here
17        ldmfd   sp!,{r0-r7, lr}
18        subs    pc,lr,#8
19
20  @@@ ---------------------------------------------------------------
21        .global reserved_handler
22  reserved_handler:               @ this will never be called
23        stmfd   sp!,{r0-r7, lr}
24        @@ handler body goes here
25        ldmfd   sp!,{r0-r7, lr}
26        movs    pc,lr
27
28  @@@ ---------------------------------------------------------------
29        .global fiq_handler
30  fiq_handler:                    @ must subtract 4 from lr
31        stmfd   sp!,{r0-r7, lr}
```

Startup Code

```
1  @@@ FILE:  start.S
2
3          .include "modes.S"
4
5  @@@ Stack locations
6          @@ uncomment one of the following two lines
7          @ .equ    stack_top, 0x10000000 @ Raspberry pi only
8          @ .equ    stack_top, 0x50000000 @ pcDuino only
9
10         .equ    fiq_stack_top, stack_top
11         .equ    irq_stack_top, stack_top - 0x1000
12         .equ    abt_stack_top, stack_top - 0x2000
13         .equ    und_stack_top, stack_top - 0x3000
14         .equ    mon_stack_top, stack_top - 0x4000
15         .equ    svc_stack_top, stack_top - 0x5000
16         .equ    sys_stack_top, stack_top - 0x6000
17
18 @@@ ----------------------------------------------------------
19 @@@ The startup code should be loaded by the boot loader.
20 @@@ The entry point is _start which performs initialization of
21 @@@ the hardware, then calls a C function.
22         .section .text.boot
23         .global _start
24         .func   _start
25 _start: @@ On reset, we should be in SVC mode.
26
27         @@ Switch to FIQ mode with interrupts disabled
28         msr     CPSR_c,#FIQ_MODE|I_BIT|F_BIT
29         ldr     sp,=fiq_stack_top @ set the FIQ stack pointer
```

Startup Code

```
1         @@ Switch to IRQ mode with interrupts disabled
2         msr     CPSR_c,#IRQ_MODE|I_BIT|F_BIT
3         ldr     sp,=irq_stack_top @ set the IRQ stack pointer
4
5         @@ Switch to ABT mode with interrupts disabled
6         msr     CPSR_c,#ABT_MODE|I_BIT|F_BIT
7         ldr     sp,=abt_stack_top @ set the ABT stack pointer
8
9         @@ Switch to UND mode with interrupts disabled
10        msr     CPSR_c,#UND_MODE|I_BIT|F_BIT
11        ldr     sp,=und_stack_top @ set the UND stack pointer
12
13        @@ Switch to SYS mode with interrupts disabled
14        msr     CPSR_c,#SYS_MODE|I_BIT|F_BIT
15        ldr     sp,=sys_stack_top @ set SYS/USR stack pointer
16
17        @@ Switch to SVC mode with interrupts disabled
18        msr     CPSR_c,#SVC_MODE|I_BIT|F_BIT
19        ldr     sp,=svc_stack_top @ set SVC stack pointer
20
21        @@ Clear the .bss segment to all zeros
22        @@ The __bss_start__ and __bss_end__ symbols are
23        @@ defined by the linker.
24        ldr     r1,=__bss_start__ @ load pointer to bss and
25        ldr     r2,=__bss_end__   @ to byte following bss
26        mov     r3,#0             @ load fill value (zero)
27 bssloop:cmp     r1,r2             @ Start filling
28        bge     bssdone
29        str     r3,[r1],#4
30        b       bssloop           @ loop until done
```

Startup Code

```
1         @@ Set up the vector table
2         bl      setup_vector_table
3
4         @@ Call the Main function
5         bl      main
6
7         @@ If main ever returns, cause an exception
8         swi     0xFFFFFF          @ this should never happen
9         .size   _start, . - _start
10        .endfunc
```

Initializing the Vector Table

```
1         .section .rodata        @ mark this data as read-only
2         .align        2
3         @@ All of the eight instructions in the vector table are
4         @@    ldr pc,[pc, #24]
5         @@ which loads the program counter with the program
6         @@ counter + 24. When the pc is used in this addressing
7         @@ mode, there is an 8-byte offset because of the
8         @@ pipeline (8+24=32).  The address of the corresponding
9         @@ handler will be stored 32 bytes after each entry.
10 Vector_Table:
11        ldr pc,rh
12        ldr pc,uh
13        ldr pc,sh
14        ldr pc,ph
15        ldr pc,dh
16        ldr pc,vh
17        ldr pc,ih
18        ldr pc,fh
19 rh:     .word   reset_handler
20 uh:     .word   undef_handler
21 sh:     .word   swi_handler
22 ph:     .word   pAbort_handler
23 dh:     .word   dAbort_handler
24 vh:     .word   reserved_handler
25 ih:     .word   irq_handler
26 fh:     .word   fiq_handler
27        .equ    VT_SIZE, (. - Vector_Table)
```

Initializing the Vector Table

```
1  @@@ -----------------------------------------------------------------
2          .text
3          .align          2
4          .global setup_vector_table
5  setup_vector_table:
6          @@ Cortex-A and similar: set the vector base address
7          @@ to 0x0.  (The boot loader may have changed it.)
8          mov     r0,#0
9          MCR     p15,0,r0,c12,c0,0@ Write VBAR
10         @@ This section will copy Vector_Table to address 0x0
11         ldr     r0,=Vector_Table @ pointer to table of addresses
12         ldr     r1,=0x0
13         mov     r3,#VT_SIZE      @ stop after 64 bytes
14 movit:  ldr     r2,[r0],#4
15         str     r2,[r1],#4
16         cmp     r1,r3
17         blt     movit
18         mov     pc,lr                @ return
```

Linker Script

```
/* A linker script for bare-metal on the Raspberry Pi */
ENTRY(_start)

SECTIONS
{
    /* The executable starts at 0x8000 */
    . = 0x8000;
    _start = .;
    _text_start = .;
    .text :
    {
        KEEP(*(.text.boot))
        *(.text)
    }
    . = ALIGN(4096); /* align to page size */
    _text_end = .;

    _rodata_start = .;
    .rodata :
    {
        *(.rodata)
    }
    . = ALIGN(4096); /* align to page size */
    _rodata_end = .;
```

Linker Script

```
    _data_start = .;
    .data :
    {
        *(.data)
    }
    . = ALIGN(4096); /* align to page size */
    _data_end = .;

    _bss_start = .;
    .bss :
    {
        bss = .;
        *(.bss)
    }
    . = ALIGN(4096); /* align to page size */
    _bss_end = .;
    _end = .;
}
```

Simple Main Program

```
1  @@@ FILE: main.S
2  @@@ This program reads from three buttons connected to GPIO3-5, and
3  @@@ controls three leds connected to GPIO0-2. The main loop runs
4  @@@ continuously.
5          .global main
6  main:   stmfd   sp!,{lr}
7          @@ Set the GPIO pins
8          mov     r0,#0           @ Port 0
9          bl      GPIO_dir_output @ set for output
10         mov     r0,#1           @ Port 1
11         bl      GPIO_dir_output @ set for output
12         mov     r0,#2           @ Port 2
13         bl      GPIO_dir_output @ set for output
14
15         mov     r0,#3           @ Port 3
16         bl      GPIO_dir_input  @ set for input
17         mov     r0,#4           @ Port 4
18         bl      GPIO_dir_input  @ set for input
19         mov     r0,#5           @ Port 5
20         bl      GPIO_dir_input  @ set for input
21 @@@ Main loop just reads buttons and updates the LEDs.
22 loop:
23         @@ Read the state of the inputs and
24         @@ set the ouputs to the same state.
25         mov     r0,#3           @ Pin 3
26         bl      GPIO_get_pin    @ read it
27         mov     r1,r0           @ copy pin state to r1
```

Simple Main Program

```
1         mov     r0,#0           @ Pin 0
2         bl      GPIO_set_pin    @ write it
3
4         mov     r0,#4           @ Pin 4
5         bl      GPIO_get_pin    @ read it
6         mov     r1,r0           @ copy pin state to r1
7         mov     r0,#1           @ Pin 1
8         bl      GPIO_set_pin    @ write it
9
10        mov     r0,#5           @ Pin 5
11        bl      GPIO_get_pin    @ read it
12        mov     r1,r0           @ copy pin state to r1
13        mov     r0,#2           @ pin 2
14        bl      GPIO_set_pin    @ write it
15
16        b       loop
17        ldmfd   sp!,{pc}
```

Building an Image

```
lpyeatt@pcDuino$ make
gcc  -c main.S -o main.o
gcc  -c pcDuino_GPIO.S -o pcDuino_GPIO.o
gcc  -c start.S -o start.o
gcc  -c vectab.S -o vectab.o
gcc  -c handlers.S -o handlers.o
ld main.o pcDuino_GPIO.o start.o vectab.o handlers.o
    -Tbare_metal.ld -o bare.elf
objcopy bare.elf -O binary kernel.img
mkimage -A arm -T kernel -a 40008000 -C none \
          -n "bare metal" -d kernel.img uImage
Image Name:   bare metal
Created:      Tue Oct 13 13:38:20 2015
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    4240 Bytes = 4.14 kB = 0.00 MB
Load Address: 40008000
Entry Point:  40008000
lpyeatt@pcDuino$
```

Configuring the Interrupt Controller

```
1  @@@ FILE: RasPiIC.S
2  @@@ Functions to manage the Interrupt Controller on the
3  @@@ Raspberry Pi
4          @@ Address of Interrupt Controller
5          .equ    IC,     0x7e00B000
6          @@ Register offsets
7          .equ    IRQBP,  0x200 @ IRQ basic pending
8          .equ    IRQP1,  0x204 @ IRQ pending 1
9          .equ    IRQP2,  0x208 @ IRQ pending 2
10         .equ    FIQC,   0x20C @ FIQ control
11         .equ    IRQEN1, 0x210 @ IRQ enable 1
12         .equ    IRQEN2, 0x214 @ IRQ enable 2
13         .equ    IRQBEN, 0x218 @ Enable basic IRQs
14         .equ    IRQDA1, 0x21C @ IRQ disable 1
15         .equ    IRQDA2, 0x220 @ IRQ disable 2
16         .equ    IRQBDA, 0x224 @ Disable basic IRQs
17 @@@ ----------------------------------------------------------------
18         .text
19         .align 2
20 @@@ ----------------------------------------------------------------
21 @@@ Initialization of the Interrupt Controller (IC)
22         .global IC_init
23 IC_init:
24         @@ disable all interrupts
25         ldr     r0,=IC
26         mov     r1,#0
27         str     r1,[r0,#IRQEN1]
28         str     r1,[r0,#IRQEN2]
29         str     r1,[r0,#IRQBEN]
30         mov     pc,lr
```

Configuring the Interrupt Controller

```
1  @@@ ----------------------------------------------------------------
2  @@@ config_interrupt (int ID, int CPU);
3  @@@ On Raspberry Pi, this just enables the timer interrupt
4          .global config_interrupt
5  config_interrupt:
6          ldr     r0,=IC
7          mov     r1,#1
8          str     r1,[r0,#IRQBEN]
9          mov     pc,lr
10 @@@ ----------------------------------------------------------------
11 @@@ int get_interrupt_number();
12 @@@ Get the interrupt ID for the current interrupt.
13 @@@ On Raspberry Pi, just read and return the pending register.
14         .global get_interrupt_number
15 get_interrupt_number: @ Read the ICCIAR from the CPU Interface
16         ldr     r0,=IC
17         ldr     r0,[r0,#IRQBP]
18         mov     pc,lr
19 @@@ ----------------------------------------------------------------
20 @@@ void end_of_interrupt(int ID);
21 @@@ Notify the IC that the interrupt has been processed.
22 @@@ On Raspberry Pi, this does nothing
23         .global end_of_interrupt
24 end_of_interrupt:
25         mov     pc,lr
```

Expanding the Interrupt Handler

```
1  @@@ ------------------------------------------------------------
2          .global irq_handler
3  irq_handler:
4          stmfd   sp!,{r0-r12,lr}
5
6          @@ find out which interrupt we are servicing
7          bl      get_interrupt_number @ returns in r0
8          stmfd   sp!,{r0}        @ save interrupt number
9
10         cmp     r0,#54          @ is it the timer interrupt?
11         bleq    check_timer_interrupt
12
13         ldmfd   sp!,{r0}        @ retrieve interrupt number
14         bl      end_of_interrupt@ tell GIC we are done
15
16         ldmfd   sp!,{r0-r12,lr}
17         subs    pc, lr, #4      @ must subtract 4 from lr
```

Interrupt Driven Main

```
1  @@@ FILE: main.S
2  @@@ This program reads from three buttons connected to GPIO3-5, and
3  @@@ controls three leds connected to GPIO0-2. The main loop puts
4  @@@ the CPU to sleep after each iteration.  A timer interrupt wakes
5  @@@ it up some time later.
6          .global main
7  main:   stmfd   sp!,{lr}
8          @@ Set the GPIO pins
9          mov     r0,#0           @ Port 0
10         bl      GPIO_dir_output @ set for output
11         mov     r0,#1           @ Port 1
12         bl      GPIO_dir_output @ set for output
13         mov     r0,#2           @ Port 2
14         bl      GPIO_dir_output @ set for output
15
16         mov     r0,#3           @ Port 3
17         bl      GPIO_dir_input  @ set for input
18         mov     r0,#4           @ Port 4
19         bl      GPIO_dir_input  @ set for input
20         mov     r0,#5           @ Port 5
21         bl      GPIO_dir_input  @ set for input
22 @@@ Main loop just reads buttons and updates the LEDs,
23 @@@ then puts the CPU to sleep until an interrupt occurs.
24 loop:   @@ Read the state of the inputs and
25         @@ set the ouputs to the same state.
26         mov     r0,#3           @ Pin 3
27         bl      GPIO_get_pin    @ read it
28         mov     r1,r0           @ copy pin state to r1
29         mov     r0,#0           @ Pin 0
30         bl      GPIO_set_pin    @ write it
```

Interrupt Driven Main

```
1         mov     r0,#4             @ Pin 4
2         bl      GPIO_get_pin      @ read it
3         mov     r1,r0             @ copy pin state to r1
4         mov     r0,#1             @ Pin 1
5         bl      GPIO_set_pin      @ write it
6         mov     r0,#5             @ Pin 5
7         bl      GPIO_get_pin      @ read it
8         mov     r1,r0             @ copy pin state to r1
9         mov     r0,#2             @ pin 2
10        bl      GPIO_set_pin      @ write it
11        @@ Put CPU to sleep until an interrupt occurs
12        //wfi                     @ used on pcDunio
13        mov     r0,#0
14        mcr     p15,0,r0,c7,c0,4@ used on Raspberry Pi
15        b       loop
16        ldmfd   sp!,{pc}
```

ARM Versions

Almost 20 major versions of the ARMv7 architecture.
Targeted for everything from smart sensors to desktops and servers.
Three major *profiles*:

ARMv7-A Applications processors are capable of running a full, multiuser, virtual memory, multiprocessing operating system.

ARMv7-R: Real-time processors are for embedded systems that may need powerful processors, cache, and/or large amounts of memory.

ARMv7-M: Microcontroller processors only execute Thumb instructions and are intended for use in very small cost-sensitive embedded systems. They provide low cost, low power, and small size, and may not have hardware floating point or other high-performance features.