Modern Assembly Language Programming
with the
ARM processor
Chapter 14: Running Without an Operating System
1 Introduction

2 Challenges

3 ARM Privileged Modes

4 Interrupts and Exceptions
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, there 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?
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 \texttt{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, there 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

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

Privileged Modes Have Private Registers

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

<table>
<thead>
<tr>
<th>CPSR</th>
<th>SPSR_svc</th>
<th>SPSR_abt</th>
<th>SPSR_und</th>
<th>SPSR_irq</th>
<th>SPSR_fiq</th>
</tr>
</thead>
</table>
Exception and Interrupt Processing

When an exception occurs, the processor goes into privileged mode.

- **Main Process**
- **Exception Handler**
  - Save user mode registers
  - Handle the Exception
  - Restore user mode registers
Task Switching

User Process

Exception Handler
- Save user registers in process state structure
- Handle the exception
- If a device is involved then call a device driver function

Device Driver Function
- Examine device state
- Transfer data to/from device or perform any other action needed

Scheduler
- Select a user process to run

Restore user registers from process state structure

If a device is involved then call a device driver function
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.

<table>
<thead>
<tr>
<th>Address</th>
<th>Exception</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>Reset</td>
<td>svc</td>
</tr>
<tr>
<td>0x00000004</td>
<td>Undefined Instruction</td>
<td>und</td>
</tr>
<tr>
<td>0x00000008</td>
<td>Software Interrupt</td>
<td>svc</td>
</tr>
<tr>
<td>0x0000000c</td>
<td>Prefetch Abort</td>
<td>abt</td>
</tr>
<tr>
<td>0x00000010</td>
<td>Data Abort</td>
<td>abt</td>
</tr>
<tr>
<td>0x00000014</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x00000018</td>
<td>Interrupt Request</td>
<td>irq</td>
</tr>
<tr>
<td>0x0000001c</td>
<td>Fast Interrupt Request</td>
<td>fiq</td>
</tr>
</tbody>
</table>

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 privileged 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

```assembly
@@@ FILE: handlers.S
.text
.align 2

@@@ On reset, jump to startup function. The CPU must not
@@@ be in usr or sys mode! (add some code to check it)
.global reset_handler
reset_handler:
    b _start

@@

.global irq_handler
irq_handler: @ must subtract 4 from lr
    stmfd  sp!,{r0-r7, lr}
    @ handler body goes here
    ldmfd sp!,{r0-r7, lr}
    subs pc, lr, #4

@@

.global undef_handler
undef_handler: @ lr holds exact return address
    stmfd  sp!,{r0-r7, lr}
    @ handler body goes here
    ldmfd sp!,{r0-r7, lr}
    movs pc,lr

@@

.global swi_handler
swi_handler: @ lr holds exact return address
    stmfd  sp!,{r0-r7, lr}
    @ handler body goes here
```
Exception Handler Stubs

```assembly
ldmfd  sp!,{r0-r7,  lr}
movs   pc,lr

@@@  
.globl pAbort_handler
pAbort_handler: @ must subtract 4 from lr
    stmfd  sp!,{r0-r7,  lr}
    @@ handler body goes here
    ldmfd  sp!,{r0-r7,  lr}
    subs   pc,lr,#4

@@@  
.globl dAbort_handler
dAbort_handler: @ must subtract 8 from lr
    stmfd  sp!,{r0-r7,  lr}
    @@ handler body goes here
    ldmfd  sp!,{r0-r7,  lr}
    subs   pc,lr,#8

@@@  
.globl reserved_handler
reserved_handler: @ this will never be called
    stmfd  sp!,{r0-r7,  lr}
    @@ handler body goes here
    ldmfd  sp!,{r0-r7,  lr}
    movs   pc,lr

@@@  
.globl fiq_handler
fiq_handler: @ must subtract 4 from lr
    stmfd  sp!,{r0-r7,  lr}
```
Startup Code

@@@ FILE: start.S

.include "modes.S"

@@@ Stack locations
@@ uncomment one of the following two lines
@ .equ stack_top, 0x10000000 @ Raspberry pi only
@ .equ stack_top, 0x50000000 @ pcDuino only

.equ fiq_stack_top, stack_top
.equ irq_stack_top, stack_top - 0x1000
.equ abt_stack_top, stack_top - 0x2000
.equ und_stack_top, stack_top - 0x3000
.equ mon_stack_top, stack_top - 0x4000
.equ svc_stack_top, stack_top - 0x5000
.equ sys_stack_top, stack_top - 0x6000

@@@ -----------------------------------------------------------
@@@ The startup code should be loaded by the boot loader.
@@@ The entry point is _start which performs initialization of
@@@ the hardware, then calls a C function.
@@@ .section .text.boot
@@@ .global _start
@@@ .func _start
_start: @@ On reset, we should be in SVC mode.
@@ Switch to FIQ mode with interrupts disabled
msr CPSR_c,#FIQ_MODE|I_BIT|F_BIT
ldr sp,=fiq_stack_top @ set the FIQ stack pointer
@@ Switch to IRQ mode with interrupts disabled
msr  CPSR_c,#IRQ_MODE|I_BIT|F_BIT
1dr  sp,=irq_stack_top @ set the IRQ stack pointer

@@ Switch to ABT mode with interrupts disabled
msr  CPSR_c,#ABT_MODE|I_BIT|F_BIT
1dr  sp,=abt_stack_top @ set the ABT stack pointer

@@ Switch to UND mode with interrupts disabled
msr  CPSR_c,#UND_MODE|I_BIT|F_BIT
1dr  sp,=und_stack_top @ set the UND stack pointer

@@ Switch to SYS mode with interrupts disabled
msr  CPSR_c,#SYS_MODE|I_BIT|F_BIT
1dr  sp,=sys_stack_top @ set SYS/USR stack pointer

@@ Switch to SVC mode with interrupts disabled
msr  CPSR_c,#SVC_MODE|I_BIT|F_BIT
1dr  sp,=svc_stack_top @ set SVC stack pointer

@@ Clear the .bss segment to all zeros
@@ The __bss_start__ and __bss_end__ symbols are
@@ defined by the linker.
1dr  r1,=__bss_start__ @ load pointer to bss and
1dr  r2,=__bss_end__ @ to byte following bss
mov  r3,#0 @ load fill value (zero)
bssloop:cmp  r3,r1 @ Start filling
  bge  bssdone
  str  r3,[r1],#4 @ loop until done
Startup Code

@@ Set up the vector table
bl setup_vector_table

@@ Call the Main function
bl main

@@ If main ever returns, cause an exception
swi 0xFFFFFFF @ this should never happen
.size _start, . - _start
.endfunc
Initializing the Vector Table

```
.section .rodata @ mark this data as read-only
.align 2
@@ All of the eight instructions in the vector table are
@@ ldr pc,[pc, #24]
@@ which loads the program counter with the program
@@ counter + 24. When the pc is used in this addressing
@@ mode, there is an 8-byte offset because of the
@@ pipeline (8+24=32). The address of the corresponding
@@ handler will be stored 32 bytes after each entry.

Vector_Table:
  ldr pc,rh
  ldr pc,uh
  ldr pc,sh
  ldr pc,ph
  ldr pc,dh
  ldr pc,vh
  ldr pc,ih
  ldr pc,fh

rh: .word reset_handler
uh: .word undef_handler
sh: .word swi_handler
ph: .word pAbort_handler
dh: .word dAbort_handler
vh: .word reserved_handler
ih: .word irq_handler
fh: .word fiq_handler
.equ VT_SIZE, (. - Vector_Table)
```
Initializing the Vector Table

```
@@@ ------------------------------------------------------------
.text
.align 2
.global setup_vector_table

setup_vector_table:
@@ Cortex-A and similar: set the vector base address
to 0x0. (The boot loader may have changed it.)
mov r0,#0
MCR p15,0,r0,c12,c0,0@ Write VBAR
@@ This section will copy Vector_Table to address 0x0
ldr r0,=Vector_Table @ pointer to table of addresses
ldr r1,=0x0
mov r3,#VT_SIZE @ stop after 64 bytes
movit: ldr r2,[r0],#4
str r2,[r1],#4
cmp r1,r3
bit movit
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)
}
.data = ALIGN(4096); /* align to page size */
_data_end = .;

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

```assembly
@@@ FILE: main.S
@@@ This program reads from three buttons connected to GPIO3-5, and
@@@ controls three leds connected to GPIO0-2. The main loop runs
@@@ continuously.

.global main
main:   stmfd   sp!,{lr}
    @@ Set the GPIO pins
    mov     r0,#0      @ Port 0
    bl      GPIO_dir_output @ set for output
    mov     r0,#1      @ Port 1
    bl      GPIO_dir_output @ set for output
    mov     r0,#2      @ Port 2
    bl      GPIO_dir_output @ set for output
                  
    mov     r0,#3      @ Port 3
    bl      GPIO_dir_input @ set for input
    mov     r0,#4      @ Port 4
    bl      GPIO_dir_input @ set for input
    mov     r0,#5      @ Port 5
    bl      GPIO_dir_input @ set for input

@@@ Main loop just reads buttons and updates the LEDs.
loop:
    @@ Read the state of the inputs and
    @@ set the outputs to the same state.
    mov     r0,#3      @ Pin 3
    bl      GPIO_get_pin @ read it
    mov     r1,r0      @ copy pin state to r1
```
Simple Main Program

```
mov r0,#0 @ Pin 0
bl GPIO_set_pin @ write it

mov r0,#4 @ Pin 4
bl GPIO_get_pin @ read it
mov r1,r0 @ copy pin state to r1
mov r0,#1 @ Pin 1
bl GPIO_set_pin @ write it

mov r0,#5 @ Pin 5
bl GPIO_get_pin @ read it
mov r1,r0 @ copy pin state to r1
mov r0,#2 @ Pin 2
bl GPIO_set_pin @ write it

b loop
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

```assembly
; FILE: RasPiIC.S
; Functions to manage the Interrupt Controller on the Raspberry Pi
; Address of Interrupt Controller
.equ IC, 0x7e00B000
; Register offsets
.equ IRQBP, 0x200 @ IRQ basic pending
.equ IRQP1, 0x204 @ IRQ pending 1
.equ IRQP2, 0x208 @ IRQ pending 2
.equ FIQC, 0x20C @ FIQ control
.equ IRQEN1, 0x210 @ IRQ enable 1
.equ IRQEN2, 0x214 @ IRQ enable 2
.equ IRQBEN, 0x218 @ Enable basic IRQs
.equ IRQDA1, 0x21C @ IRQ disable 1
.equ IRQDA2, 0x220 @ IRQ disable 2
.equ IRQBDA, 0x224 @ Disable basic IRQs

.text
.align 2

Initialization of the Interrupt Controller (IC)
.global IC_init
IC_init:
    @ disable all interrupts
    ldr r0,=IC
    mov r1,#0
    str r1,[r0,#IRQEN1]
    str r1,[r0,#IRQEN2]
    str r1,[r0,#IRQBEN]
    mov pc,lr
```
Configuring the Interrupt Controller

```assembly
@@@ config_interrupt (int ID, int CPU);  
@@@ On Raspberry Pi, this just enables the timer interrupt
 .global config_interrupt
config_interrupt:
  ldr   r0,=IC
  mov   r1,#1
  str   r1,[r0,#IRQBEN]
  mov   pc,lr

@@@ int get_interrupt_number();  
@@@ Get the interrupt ID for the current interrupt.
@@@ On Raspberry Pi, just read and return the pending register.
 .global get_interrupt_number
get_interrupt_number: @ Read the ICCIAR from the CPU Interface
  ldr   r0,=IC
  ldr   r0,[r0,#IRQBP]
  mov   pc,lr

@@@ void end_of_interrupt(int ID);  
@@@ Notify the IC that the interrupt has been processed.
@@@ On Raspberry Pi, this does nothing
 .global end_of_interrupt
end_of_interrupt:
  mov   pc,lr
```
Expanding the Interrupt Handler

```assembly
.global irq_handler

irq_handler:
    stmfd sp!, {r0-r12, lr}

    @@ find out which interrupt we are servicing
    bl get_interrupt_number @ returns in r0
    stmfd sp!, {r0} @ save interrupt number

    cmp r0, #54 @ is it the timer interrupt?
    breq check_timer_interrupt

    ldmfd sp!, {r0} @ retrieve interrupt number
    bl end_of_interrupt @ tell GIC we are done

    ldmfd sp!, {r0-r12, lr}
    subs pc, lr, #4 @ must subtract 4 from lr
```
Interrupt Driven Main

```
@@@ FILE: main.S
@@@ This program reads from three buttons connected to GPIO3-5, and
@@@ controls three leds connected to GPIO0-2. The main loop puts
@@@ the CPU to sleep after each iteration. A timer interrupt wakes
@@@ it up some time later.

.global main
main:    stmfd sp!,{lr}
@@ Set the GPIO pins
    mov r0,#0 @ Port 0
    bl GPIO_dir_output @ set for output
    mov r0,#1 @ Port 1
    bl GPIO_dir_output @ set for output
    mov r0,#2 @ Port 2
    bl GPIO_dir_output @ set for output

    mov r0,#3 @ Port 3
    bl GPIO_dir_input @ set for input
    mov r0,#4 @ Port 4
    bl GPIO_dir_input @ set for input
    mov r0,#5 @ Port 5
    bl GPIO_dir_input @ set for input

@@@ Main loop just reads buttons and updates the LEDs,
@@@ then puts the CPU to sleep until an interrupt occurs.
loop:    @@ Read the state of the inputs and
         @@ set the outputs to the same state.
         mov r0,#3 @ Pin 3
         bl GPIO_get_pin @ read it
         mov r1,r0 @ copy pin state to r1
         mov r0,#0 @ Pin 0
         bl GPIO_set_pin @ write it
```
Interrupt Driven Main

```assembly
mov     r0, #4           @ Pin 4
bl      GPIO_get_pin    @ read it
mov     r1, r0          @ copy pin state to r1
mov     r0, #1          @ Pin 1
bl      GPIO_set_pin    @ write it
mov     r0, #5          @ Pin 5
bl      GPIO_get_pin    @ read it
mov     r1, r0          @ copy pin state to r1
mov     r0, #2          @ pin 2
bl      GPIO_set_pin    @ write it
@@ Put CPU to sleep until an interrupt occurs
//wfi                 @ used on pcDunio
mov     r0, #0
mcr     p15, 0, r0, c7, c0, 4 @ used on Raspberry Pi
b       loop
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, multiprocessed 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.