ARM Architecture and Programming

2

ARM (ARM Architecture 2016) is a family of Reduced Instruction Set Computing (RISC) microprocessors developed specifically for mobile and embedded computing environments. Due to their small sizes and low power requirements, ARM processors have become the most widely used processors in mobile devices, e.g. smart phones, and embedded systems. Currently, most embedded systems are based on ARM processors. In many cases, embedded system programming has become almost synonymous with ARM processor programming. For this reason, we shall also use ARM processors for the design and implementation of embedded systems in this book. Depending on their release time, ARM processors can be classified into classic cores and the more recent (since 2005) Cortex cores. Depending on their capabilities and intended applications, ARM Cortex cores can be classified into three categories (ARM Cortex 2016).

.The Cortex-M series: these are microcontroller-oriented processors intended for Micro Controller Unit (MCU) and System on Chip (SoC) applications.
.The Cortex-R series: these are embedded processors intended for real-time signal processing and control applications.
.The Cortex-A series: these are applications processors intended for general purpose applications, such as embedded systems with full featured operating systems.

The ARM Cortex-A series processors are the most powerful ARM processors, which include the Cortex-A8 (ARM Cortex-A8 2010) single core and the Cortex A9-MPcore (ARM Cortex A9 MPcore 2016) with up to 4 CPUs. Because of their advanced capabilities, most recent embedded systems are based on the ARM Cortex-A series processors. On the other hand, there are also a large number of embedded systems intended for dedicated applications that are based on the classic ARM cores, which proved to be very cost-effective. In this book, we shall cover both the classic and ARM-A series processors. Specifically, we shall use the classic ARM926EJ-S core (ARM926EJ-ST 2008, ARM926EJ-ST 2010) for single CPU systems and the Cortex A9-MPcore for multiprocessor systems. A primary reason for choosing these ARM cores is because they are available as emulated virtual machines (VMs). A major goal of this book is to show the design and implementation of embedded systems in an integrated approach. In addition to covering the theory and principles, it also shows how to apply them to the design and implementation of embedded systems by programming examples. Since most readers may not have access to real ARM based systems, we shall use emulated ARM virtual machines under QEMU for implementation and testing. In this chapter, we shall cover the following topics.

. ARM architecture
. ARM instructions and basic programming in ARM assembly
. Interface assembly code with C programs
. ARM toolchains for (cross) compile-link programs
. ARM system emulators and run programs on ARM virtual machines
. Develop simple I/O device drivers. These include UART drivers for serial ports and LCD driver for displaying both images and text.

K.C. Wang, *Embedded and Real-Time Operating Systems*,
DOI 10.1007/978-3-319-51517-5_2

2.1 ARM Processor Modes

The ARM processor has seven (7) operating modes, which are specified by the 5 mode bits [4:0] in the Current Processor Status Register (CPSR). The seven ARM processor modes are:

USR mode: unprivileged User mode
SYS mode: System mode using the same set of registers as User mode
FIQ mode: Fast interrupt request processing mode
IRQ mode: normal interrupt request processing mode
SVC mode: Supervisor mode on reset or SWI (Software Interrupt)
ABT mode: data exception abort mode
UND mode: undefined instruction exception mode

2.2 ARM CPU Registers

The ARM processor has 37 registers in total, all of which are 32-bits wide. These include

1 dedicated program counter (PC)
1 dedicated current program status register (CPSR)
5 dedicated saved program status registers (SPSR)
30 general purpose registers

The registers are arranged into several banks. The accessibility of the banked registers is governed by the processor mode. In each mode the ARM CPU can access

A particular set of R0-R12 registers
A particular R13 (stack pointer), R14 (link register) and SPSR (saved program status)
The same R15 (program counter) and CPSR (current program status register)

2.2.1 General Registers

Figure 2.1 shows the organization of general registers in the ARM processor.

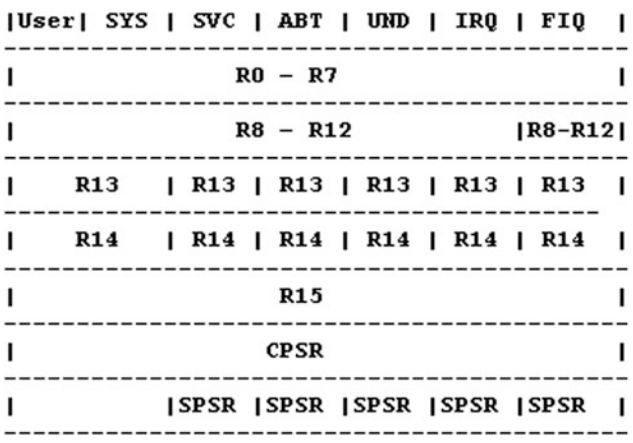


Fig. 2.1 Register banks in ARM processor

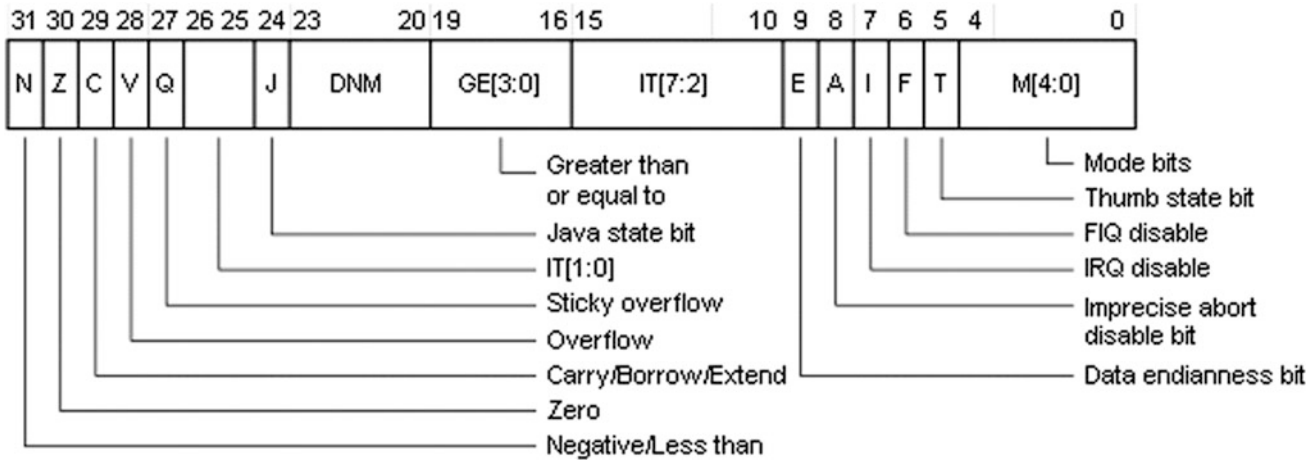


Fig. 2.2 Status register of ARM processor

User and System modes share the same set of registers. Registers R0-R12 are the same in all modes, except for the FIQ mode, which has its own separate registers R8-R12. Each mode has its own stack pointer (R13) and link register (R14). The Program Counter (PC or R15) and Current Status Register (CPSR) are the same in all modes. Each privileged mode (SVC to FIQ) has its own Saved Processor Status Register (SPSR).

2.2.2 Status Registers

In all modes, the ARM processor has the same Current Program Status Register (CPSR). Figure 2.2 shows the contents of the CPSR register.

In the CPSR register, NZCV are the condition bits, I and F are IRQ and FIQ interrupt mask bits, T = Thumb state, and M[4:0] are the processor mode bits, which define the processor mode as

```
USR:  10000  (0x10)
FIQ:  10001  (0x11)
IRQ:  10010  (0x12)
SVC:  10011  (0x13)
ABT:  10111  (0x17)
UND:  11011  (0x1B)
SYS:  11111  (0x1F)
```

2.2.3 Change ARM Processor Mode

All the ARM modes are privileged except the User mode, which is unprivileged. Like most other CPUs, the ARM processor changes mode in response to exceptions or interrupts. Specifically, it changes to FIQ mode when a FIQ interrupt occurs. It changes to IRQ mode when an IRQ interrupt occurs. It enters the SVC mode when power is turned on, following reset or executing a SWI instruction. It enters the Abort mode when a memory access exception occurs, and it enters the UND mode when it encounters an undefined instruction. An unusual feature of the ARM processor is that, while in a privileged mode, it can change mode freely by simply altering the mode bits in the CPSR, by using the MSR and MRS instructions. For example, when the ARM processor starts or following a reset, it begins execution in SVC mode. While in SVC mode, the system initialization code must set up stack pointers of other modes. To do these, it simply changes the processor to an appropriate mode, initialize the stack pointer (R13_mode) and the saved program status register (SPSR) of that mode. The following code segment shows how to switch the processor to IRQ mode while preserving other bits, e.g. F and I bits, in the CPSR.

```
MRS  r0, cpsr          // get cpsr into r0
BIC  r1, r0, #01F     // clear 5 mode bits in r0
ORR  r1, r1, #0x12    // change to IRQ mode
MSR  cpsr, r1         // write to cpsr
```

If we do not care about the CPSR contents other than the mode field, e.g. during system initialization, changing to IRQ mode can be done by writing a value to CPSR directly, as in

```
MSR cpsr, #0x92       // IRQ mode with I bit=1
```

A special usage of SYS mode is to access User mode registers, e.g. R13 (sp), R14 (lr) from privileged mode. In an operating system, processes usually run in the unprivileged User mode. When a process does a system call (by SWI), it enters the system kernel in SVC mode. While in kernel mode, the process may need to manipulate its User mode stack and return address to the User mode image. In this case, the process must be able to access its User mode sp and lr. This can be done by switching the CPU to SYS mode, which shares the same set of registers with User mode. Likewise, when an IRQ interrupt occurs, the ARM processor enters IRQ mode to execute an interrupt service routine (ISR) to handle the interrupt. If the ISR allows nested interrupts, it must switch the processor from IRQ mode to a different privileged mode to handle nested interrupts. We shall demonstrate this later in Chap. 3 when we discuss exceptions and interrupts processing in ARM based systems.

2.3 Instruction Pipeline

The ARM processor uses an internal pipeline to increase the rate of instruction flow to the processor, allowing several operations to be undertaken simultaneously, rather than serially. In most ARM processors, the instruction pipeline consists of 3 stages, FETCH-DECODE-EXECUTE, as shown below.

```
PC      FETCH    Fetch instruction from memory
PC-4    DECODE   Decode registers used in instruction
PC-8    EXECUTE  Execute the instruction
```

The Program Counter (PC) actually points to the instruction being fetched, rather than the instruction being executed. This has implications to function calls and interrupt handlers. When calling a function using the BL instruction, the return address is actually PC-4, which is adjusted by the BL instruction automatically. When returning from an interrupt handler, the return address is also PC-4, which must be adjusted by the interrupt handler itself, unless the ISR is defined with the **__attribute__((interrupt))** attribute, in which case the compiled code will adjust the link register automatically. For some exceptions, such as Abort, the return address is PC-8, which points to the original instruction that caused the exception.

2.4 ARM Instructions

2.4.1 Condition Flags and Conditions

In the CPSR of ARM processors, the highest 4 bits, NZVC, are condition flags or simply condition code, where

```
N = negative,  Z = zero,  V = overflow,  C = carry bit out
```

Condition flags are set by comparison and TST operations. By default, data processing operations do not affect the condition flags. To cause the condition flags to be updated, an instruction can be postfix with the S symbol, which sets the S bit in the instruction encoding. For example, both of the following instructions add two numbers, but only the **ADDS** instruction affects the condition flags.

```
      ADD r0, r1, r2   ; r0 = r1 + r2
.     ADDS r0, r1, r2  ; r0 = r1 + r2 and set condition flags
```

In the ARM 32-bit instruction encoding, the leading 4 bits [31:28] represent the various combinations of the condition flag bits, which form the condition field of the instruction (if applicable). Based on the various combinations of the condition flag bits, conditions are defined mnemonically as EQ, NE, LT, GT, LE, GE, etc. The following shows some of the most commonly used conditions and their meanings.

```
0000 : EQ  Equal                        (Z set)
0001 : NE  Not equal                    (Z clear)
0010 : CS  Carry set                    (C set)
0101 : VS  Overflow set                 (V set)
1000 : HI  Unsigned higher              (C set and Z clear)
1001 : LS  Unsigned lower or same       (C clear or Z set)
1010 : GE  Signed greater than or equal (C  = V)
1011 : LT  Signed less than             (C != V)
1100 : GT  Signed greater than          (Z=0 and N=V)
1101 : LE  Signed less than or equal    (Z=1 or N!=V)
1110 : AL  Always
```

A rather unique feature of the ARM architecture is that almost all instructions can be executed conditionally. An instruction may contain an optional condition suffix, e.g. EQ, NE, LT, GT, GE, LE, GT, LT, etc. which determines whether the CPU will execute the instruction based on the specified condition. If the condition is not met, the instruction will not be executed at all without any side effects. This eliminates the need for many branches in a program, which tend to disrupt the instruction pipeline. To execute an instruction conditionally, simply postfix it with the appropriate condition. For example, a non-conditional ADD instruction has the form:

```
ADD r0, r1, r2    ; r0 = r1 + r2
```

To execute the instruction only if the zero flag is set, append the instruction with EQ.

```
ADDEQ r0, r1, r2  ; If zero flag is set then r0 = r1 + r2
```

Similarly for other conditions.

2.4.2 Branch Instructions

Branching instructions have the form

```
  B{<cond>} label           ; branch to label
BL{<cond>} subroutine       ; branch to subroutine with link
```

The Branch (B) instruction causes a direct branch to an offset relative to the current PC. The Branch with link (BL) instruction is for subroutine calls. It writes PC-4 into LR of the current register bank and replaces PC with the entry address of the subroutine, causing the CPU to enter the subroutine. When the subroutine finishes, it returns by the saved return address in the link register R14. Most other processors implement subroutine calls by saving the return address on stack. Rather than saving the return address on stack, the ARM processor simply copies PC-4 into R14 and branches to the called subroutine. If the called subroutine does not call other subroutines, it can use the LR to return to the calling place quickly. To return from a subroutine, the program simply copies LR (R14) into PC (R15), as in

```
MOV PC, LR  or  BX LR
```

However, this works only for one-level subroutine calls. If a subroutine intends to make another call, it must save and restore the LR register explicitly since each subroutine call will change the current LR. Instead of the MOV instruction, the MOVS instruction can also be used, which restores the original flags in the CPSR.

2.4.3 Arithmetic Operations

The syntax of arithmetic operations is:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

The instruction performs an arithmetic operation on two operands and places the results in the destination register Rd. The first operand, Rn, is always a register. The second operand can be either a register or an immediate value. In the latter case, the operand is sent to the ALU via the **barrel shifter** to generate an appropriate value.
Examples:

```
ADD r0, r1, r2   ; r0 = r1 + r2
SUB r3, r3, #1   ; r3 = r3 - 1
```

2.4.4 Comparison Operations

CMP: operand1—operand2, but result not written
TST: operand1 AND operand2, but result not written
TEQ: operand1 EOR operand2, but result not written

Comparison operations update the condition flag bits in the status register, which can be used as conditions in subsequent instructions. Examples:

```
CMP   r0, r1     ; set condition bits in CPSR by r0-r1
TSTEQ r2, #5     ; test r2 and 5 for equal and set Z bit in CPSR
```

2.4.5 Logical Operations

```
AND:  operand1 AND operand2          ; bit-wise AND
EOR:  operand1 EOR operand2          ; bit-wise exclusive OR
ORR:  operand1 OR operand2           ; bit-wise OR
BIC:  operand1 AND (NOT operand2)    ; clear bits
```

Examples:

```
AND    r0, r1, r2    ; r0 = r1 & r2
ORR    r0, r0, #0x80 ; set bit 7 of r0 to 1
BIC    r0, r0, #0xF  ; clear low 4 bits of r0
EORS   r1, r3, r0    ; r1 = r3 ExOR r0 and set condition bits
```

2.4.6 Data Movement Operations

```
MOV    operand1, operand2
MVN    operand1, NOT operand2
```

Examples:

```
MOV    r0, r1        ; r0 = r1 : Always execute
MOVS   r2, #10       ; r2 = 10 and set condition bits Z=0 N=0
MOVNEQ r1, #0        ; r1 = 0 only if Z bit != 0
```

2.4.7 Immediate Value and Barrel Shifter

The Barrel shifter is another unique feature of ARM processors. It is used to generate shift operations and immediate operands inside the ARM processor. The ARM processor does not have actual shift instructions. Instead, it has a barrel shifter, which performs shifts as part of other instructions. Shift operations include the conventional shift left, right and rotate, as in

```
MOV r0, r0, LSL #1  ; shift r0 left by 1 bit (multiply r0 by 2)
MOV r1, r1, LSR #2  ; shift r1 right by 2 bits (divide r1 by 4)
MOV r2, r2, ROR #4  ; swap high and low 4 bits of r2
```

Most other processors allow loading CPU registers with immediate values, which form parts of the instruction stream, making the instruction length variable. In contrast, all ARM instructions are 32 bits long, and they do not use the instruction stream as data. This presents a challenge when using immediate values in instructions. The data processing instruction format has 12 bits available for operand2. If used directly, this would only give a range of 0–4095. Instead, it is used to store a 4-bit rotate value and an 8-bit constant in the range of 0–255. The 8 bits can be rotated right an even number of positions (i.e. RORs by 0, 2, 4,…,30). This gives a much larger range of values that can be directly loaded. For example, to load r0 with the immediate value 4096, use

```
MOV r0, #0x40, 26   ; generate 4096 (0x1000) by 0x40 ROR 26
```

To make this feature easier to use, the assembler will convert to this form if given the required constant in an instruction, e.g.

```
MOV r0, #4096
```

The assembler will generate an error if the given value can not be converted this way. Instead of MOV, the LDR instruction allows loading an arbitrary 32-bit value into a register, e.g.

```
LDR rd, =numeric_constant
```

If the constant value can be constructed by using either a MOV or MVN, then this will be the instruction actually generated. Otherwise, the assembler will generate an LDR with a PC-relative address to read the constant from a literal pool.

2.4.8 Multiply Instructions

```
MUL{<cond>}{S} Rd, Rm, Rs         ; Rd = Rm * Rs
MLA{<cond>}{S} Rd, Rm, Rs,Rn      ; Rd = (Rm * Rs) + Rn
```

2.4.9 LOAD and Store Instructions

The ARM processor is a Load/ Store Architecture. Data must be loaded into registers before using. It does not support memory to memory data processing operations. The ARM processor has three sets of instructions which interact with memory. These are:

- Single register data transfer (LDR/STR).
- Block data transfer (LDM/STM).
- Single Data Swap (SWP).

The basic load and store instructions are: Load and Store Word or Byte:

```
LDR / STR / LDRB / STRB
```

2.4.10 Base Register

Load/store instructions may use a base register as an index to specify the memory location to be accessed. The index may include an offset in either pre-index or post-index addressing mode. Examples of using index registers are

```
STR r0, [r1]       ; store r0 to location pointed by r1.
LDR r2, [r1]       ; load r2 from memory pointed to by r1.
STR r0, [r1, #12]  ; pre-index  addressing :STR r0 to [r1+12]
STR r0, [r1], #12  ; Post-index addressing :STR r0 to [r1],r1+12
```

2.4.11 Block Data Transfer

The base register is used to determine where memory access should occur. Four different addressing modes allow increment and decrement inclusive or exclusive of the base register location. Base register can be optionally updated following the data transfer by appending it with a '!' symbol. These instructions are very efficient for saving and restoring execution context, e.g. to use a memory area as stack, or move large blocks of data in memory. It is worth noting that, when using these instructions to save/restore multiple CPU registers to/from memory, the register order in the instruction does not matter. Lower numbered registers are always transferred to/from lower addresses in memory.

2.4.12 Stack Operations

A stack is a memory area which grows as new data is "pushed" onto the "top" of the stack, and shrinks as data is "popped" off the top of the stack. Two pointers are used to define the current limits of the stack.

- A base pointer: used to point to the "bottom" of the stack (the first location).
- A stack pointer: used to point the current "top" of the stack.

A stack is called **descending** if it grows downward in memory, i.e. the last pushed value is at the lowest address. A stack is called **ascending** if it grows upward in memory. The ARM processor supports both descending and ascending stacks. In addition, it also allows the stack pointer to either point to the last occupied address (**Full stack**), or to the next occupied address (**Empty stack**). In ARM, stack operations are implemented by the STM/LDM instructions. The stack type is determined by the postfix in the STM/LDM instructions:

- STMFD/LDMFD: Full Descending stack
- STMFA/LDMFA: Full Ascending stack
- STMED/LDMED: Empty Descending stack
- STMEA/LDMEA: Empty Ascending stack

The C compiler always uses **Full Descending** stack. Other forms of stacks are rare and almost never used in practice. For this reason, we shall only use **Full Descending stacks** throughout this book.

2.4.13 Stack and Subroutines

A common usage of stack is to create temporary workspace for subroutines. When a subroutine begins, any registers that are to be preserved can be pushed onto the stack. When the subroutine ends, it restores the saved registers by popping them off the stack before returning to the caller. The following code segment shows the general pattern of a subroutine.

```
STMFD sp!, {r0-r12, lr} ; save all registers and return address
 { Code of subroutine } ; subroutine code
LDMFD sp!, {r0-r12, pc} ; restore saved registers and return by lr
```

If the pop instruction has the 'S' bit set (by the '^' symbol), then transferring the PC register while in a privileged mode also copies the saved SPSR to the previous mode CPSR, causing return to the previous mode prior to the exception (by SWI or IRQ).

2.4.14 Software Interrupt (SWI)

In ARM, the SWI instruction is used to generate a software interrupt. After executing the SWI instruction, the ARM processor changes to SVC mode and executes from the SVC vector address 0x08, causing it to execute the SWI handler, which is usually the entry point of system calls to the OS kernel. We shall demonstrate system calls in Chap. 5.

2.4.15 PSR Transfer Instructions

The MRS and MSR instructions allow contents of CPSR/SPSR to be transferred from appropriate status register to a general purpose register. Either the entire status register or only the flag bits can be transferred. These instructions are used mainly to change the processor mode while in a privileged mode.

```
MRS{<cond>}  Rd, <psr>  ; Rd = <psr>
MSR{<cond>} <psr>, Rm   ; <psr> = Rm
```

2.4.16 Coprocessor Instructions

The ARM architecture treats many hardware components, e.g. the Memory Management Unit (MMU) as coprocessors, which are accessed by special coprocessor instructions. We shall cover coprocessors in Chap. 6 and later chapters.

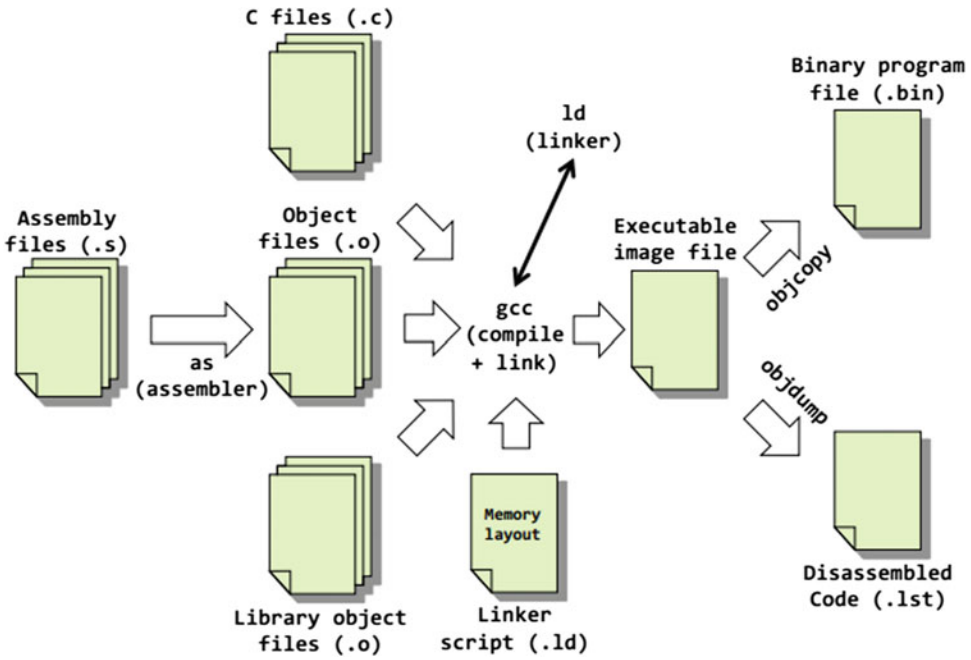


Fig. 2.3 Toolchain components

2.5 ARM Toolchain

A **toolchain** is a collection of programming tools for program development, from source code to binary executable files. A toolchain usually consists of an assembler, a compiler, a linker, some utility programs, e.g. objcopy, for file conversions and a debugger. Figure 2.3 depicts the components and data flows of a typical toolchan.

A toolchain runs on a host machine and generates code for a target machine. If the host and target architectures are different, the toolchain is called a cross toolchain or simply a **cross compiler**. Quite often, the toolchain used for embedded system development is a cross toolchain. In fact, this is the standard way of developing software for embedded systems. If we develop code on a Linux machine based on the Intel x86 architecture but the code is intended for an ARM target machine, then we need a Linux-based ARM-targeting cross compiler. There are many different versions of Linux based toolchains for the ARM architecture (ARM toolchains 2016). In this book, we shall use the arm-none-eabi toolchain under Ubuntu Linux Versions 14.04/15.10. The reader can get and install the toolchain, as well as the qemu-system-arm for ARM virtual machines, on Ubuntu Linux as follows.

```
sudo apt-get install gcc-arm-none-eabi
sudo apt-get install qemu-system-arm
```

In the following sections, we shall demonstrate how to use the ARM toolchain and ARM virtual machines under QEMU by programming examples.

2.6 ARM System Emulators

QEMU supports many emulated ARM machines (QEMU Emulator 2010). These includes ARM Integrator/CP board, ARM Versatile baseboard, ARM RealView baseboard and several others. The supported ARM CPUs include ARM926E, ARM1026E, ARM946E, ARM1136 or Cortex-A8. All these are uniprocessor (UP) or single CPU systems.

To begin with, we shall consider only uniprocessor (UP) systems. Multiprocessor (MP) systems will be covered later in Chap. 9. Among the emulated ARM virtual machines, we shall choose the ARM Versatilepb baseboard (ARM926EJ-S 2016) as the platform for implementation and testing, for the following reasons.

(1). It supports many peripheral devices. According to the QEMU user manual, the ARM Versatilepb baseboard is emulated with the following devices:

- ARM926E, ARM1136 or Cortex-A8 CPU.
- PL190 Vectored Interrupt Controller.
- Four PL011 UARTs.
- PL110 LCD controller.
- PL050 KMI with PS/2 keyboard and mouse.
- PL181 MultiMedia Card Interface with SD card.
- SMC 91c111 Ethernet adapter.
- PCI host bridge (with limitations).
- PCI OHCI USB controller.
- LSI53C895A PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices.

(2). The ARM Versatile board architecture is well documented by on-line articles in the ARM Information Center.
(3). QEMU can boot up the emulated ARM Versatilepb virtual machine directly. For example, we may generate a binary executable file, t.bin, and run it on the emulated ARM VersatilepbVM by

qemu-system-arm –M versatilepb –m 128M –kernel t.bin –serial mon:stdio

QEMU will load the t.bin file to 0x10000 in RAM and executes it directly. This is very convenient since it eliminates the need for storing the system image in a flash memory and relying on a dedicated booter to boot up the system image.

2.7 ARM Programming

2.7.1 ARM Assembly Programming Example 1

We begin ARM programming by a series of example programs. For ease of reference, we shall label the example programs by C2.x, where C2 denotes the chapter number and x denotes the program number. The first example program, C2.1, consists of a ts.s file in ARM assembly. The following shows the steps of developing and running the example program.

(1). **ts.s file of C2.1**

```
/************ ts.s file of C2.1 ***********/
       .text
       .global start
start:
       mov r0, #1       @  r0 = 1
       MOV R1, #2       @  r1 = 2
       ADD R1, R1, R0   // r1 = r1 + r0
       ldr r2, =result  // r2 = &result
       str r1, [r2]     /* result = r1 */
stop:  b   stop
       .data
result:   .word   0     /* a word location */
```

The program code loads the CPU registers r0 with the value 1, r1 with the value 2. Then it adds r0 to r1 and stores the result into the memory location labeled result.

Before continuing, it is worth noting the following. First, in an assembly code program, instructions are case insensitive. An instruction may use uppercase, lowercase or even mixed cases. For better readability, the coding style should be consistent, either all lowercase or all uppercase. However, other symbols, e.g. memory locations, are case sensitive. Second, as shown in the program, we may use the symbol @ or // to start a comment line, or include comments in matched pairs of /*

and */. Which kind of comment lines to use is a matter of personal preference. In this book, we shall use // for single comment lines, and use matched pairs of /* and */ for comment blocks that may span multiple lines, which are applicable to both assembly code and C programs.

(2). **The mk script file:**

A sh script, mk, is used to (cross) compile-link ts.s into an ELF file. Then it uses objcopy to convert the ELF file into a binary executable image named t.bin.

```
arm-none-eabi-as –o ts.o ts.s               # assemble ts.s to ts.o
arm-none-eabi-ld –T t.ld –o t.elf ts.o      # link ts.o to t.elf file
arm-none-eabi-nm t.elf                      # show symbols in t.elf
arm-none-eabi-objcopy –O binary t.elf t.bin # objcopy t.elf to t.bin
```

(3). **linker script file:**

In the linking step, a linker script file, t.ld, is used to specify the entry point and the layout of the program sections.

```
ENTRY(start)       /* define start as the entry address */
SECTIONS           /* program sections */
{
   . = 0x10000;    /* loading address, required by QEMU */
   .text : { *(.text) }   /* all text in .text section */
   .data : { *(.data) }   /* all data in .data section */
   .bss  : { *(.bss)  }   /* all bss  in .bss section */
   . =ALIGN(8);
    . =. + 0x1000;        /* 4 KB stack space */
   stack_top =.;  /* stack_top is a symbol exported by linker */
}
```

The linker generates an ELF file. If desired, the reader may view the ELF file contents by

```
arm-none-eabi-readelf -a t.elf # display all information of t.elf
arm-none-eabi-objdump -d t.elf # disassemble t.elf file
```

The ELF file is not yet executable. In order to execute, it must be converted to a binary executable file by objcopy, as in

```
arm-none-eabi-objcopy –O binary t.elf t.bin # convert t.elf to t.bin
```

(3) **Run the binary executable:** To run t.bin on the ARM Versatilepb virtual machine, enter the command

```
qemu-system-arm –M versatilepb –kernel t.bin –nographic –serial /dev/null
```

The reader may include all the above commands in a mk script, which will compile-link and run the binary executable by a single script command.

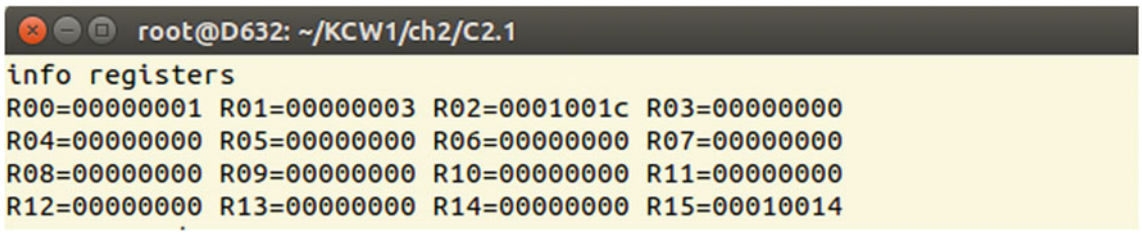


Fig. 2.4 Register contents of program C2.1

(4). **Check Results:** To check the results of running the program, enter the QEMU monitor commands:

```
info registers        :  display CPU registers
xp  /wd [address]     :  display memory contents in 32-bit words
```

Figure 2.4 shows the register contents of running the C2.1 program. As the figure shows, the register R2 contains 0x0001001C, which is the address of result. Alternatively, the command line **arm-none-eabi-nm t.elf** in the mk script also shows the locations of symbols in the program. The reader may enter the QEMU monitor command

```
xp  /wd  0x1001C
```

to display the contents of result, which should be 3. To exit QEMU, enter Control-a x, or Control-C to terminate the QEMU process.

2.7.2 ARM Assembly Programming Example 2

The next example program, denoted by C2.2, uses ARM assembly code to compute the sum of an integer array. It shows how to use a stack to call subroutine. It also demonstrates indirect and post-index addressing modes of ARM instructions. For the sake of brevity, we only show the ts.s file. All other files are the same as in the C2.1 program.

C2.2: ts.s file:

```
        .text
        .global start
start:  ldr  sp,  =stack_top // set stack pointer
        bl   sum             // call sum
stop:   b    stop            // looping

sum: // int sum(): compute the sum of an int array in Result
        stmfd sp!, {r0-r4, lr} // save r0-r4, lr on stack
        mov  r0, #0          // r0 = 0
        ldr  r1, =Array      // r1 = &Array
        ldr  r2, =N          // r2 = &N
        ldr  r2, [r2]        // r2 = N
loop:   ldr  r3, [r1], #4    // r3 = *(r1++)
        add  r0, r0, r3      // r0 += r3
        sub  r2, r2, #1      // r2—
        cmp  r2, #0          // if (r2 != 0 )
        bne  loop            //    goto loop;
        ldr  r4, =Result     // r4 = &Result
        str  r0, [r4]        // Result = r0
```

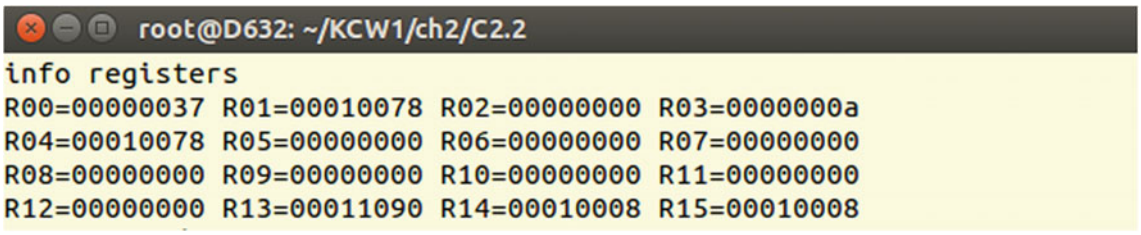


Fig. 2.5 Register contents of program C2.2

```
    ldmfd sp!, {r0-r4, pc}  // pop stack, return to caller

        .data
N:      .word 10            // number of array elements
Array:  .word 1,2,3,4,5,6,7,8,9,10
Result: .word 0
```

The program computes the sum of an integer array. The number of array elements (10) is defined in the memory location labeled N, and the array elements are defined in the memory area labeled Array. The sum is computed in R0, which is saved into the memory location labeled Result. As before, run the mk script to generate a binary executable t.bin. Then run t.bin under QEMU. When the program stops, use the monitor commands info and xp to check the results. Figure 2.5 shows the results of running the C2.2 program. As the figure shows, the register R0 contains the computed result of 0x37 (55 in decimal). The reader may use the command

arm-none-eabi-nm t.elf

to display symbols in an object code file. It lists the memory locations of the global symbols in the t.elf file, such as

```
0001004C   N
00010050   Array
00010078   Result
```

Then, use xp/wd 0x10078 to see the contents of Result, which should be 55 in decimal.

2.7.3 Combine Assembly with C Programming

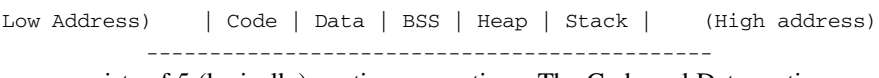
Assembly programming is indispensable, e.g. when accessing and manipulating CPU registers, but it is also very tedious. In systems programming, assembly code should be used as a tool to access and control low-level hardware, rather than as a means of general programming. In this book, we shall use assembly code only if absolutely necessary. Whenever possible, we shall implement program code in the high-level language C. In order to integrate assembly and C code in the same program, it is essential to understand program execution images and the calling convention of C.

2.7.3.1 Execution Image

An executable image (file) generated by a complier-linker consists of three logical parts.

Text section:	also called Code section containing executable code
Data section:	initialized global and static variables, static constants
BSS section:	un-initialized global and static variables. (BSS is not in the image file)

During execution, the executable image is loaded into memory to create a run-time image, which looks like the following.



A run-time image consists of 5 (logically) contiguous sections. The Code and Data sections are loaded directly from the executable file. The BSS section is created by the BSS section size in the executable file header. Its contents are usually cleared to zero. In the execution image, the Code, Data and BSS sections are fixed and do not change. The Heap area is for dynamic memory allocation within the execution image. The stack is for function calls during execution. It is logically at the high (address) end of the execution image, and it grows downward, i.e. from high address toward low address.

2.7.3.2 Function Call Convention in C

The function call convention of C consists of the following steps between the calling function (the caller) and the called function (the callee).

---------- Caller ----------

(1). load first 4 parameters in r0-r3; push any extra parameters on stack
(2). transfers control to callee by BL callee

---------- Callee ----------

(3). save LR, FP(r12) on stack, establish stack frame (FP point at saved LR)
(4). shift SP downward to allocate local variables and temp spaces on stack
(4). use parameters, locals, (and globals) to perform the function task
(5). compute and load return value in R0, pop stack to return control to caller

---------- Caller ----------

(6). get return value from R0
(7). Clean up stack by popping off extra parameters, if any.

The function call convention can be best illustrated by an example. The following t.c file contains a C function func(), which calls another function g().

```
/****************** t.c file ********************/
extern int g(int x, in y);  // an external function

int func(int a, int b, int c, int d, int e, int f)
{
  int x, y, z;              // local variables
  x = 1; y =2; z = 3;       // access locals
  g(x, y);                  // call g(x,y)
  return a + e;             // return value
}
```

Use the ARM cross compiler to generate an assembly code file named t.s by

arm-none-eabi-gcc **–S** –mcpu=arm926ej-s **t.c**

The following shows the assembly code generated by the ARM GCC compiler, in which the symbol sp is the stack pointer (R13) and fp is the stack frame pointer (R12).

```
      .global func           // export func as global symbol
func:
(1). Establish stack frame
      stmfd sp!, {fp, lr}    // save lr, fp in stack
      add   fp, sp, #4       // FP point at saved LR
(2). Shift SP downward 8 (4-byte) slots for locals and temps
      sub   sp, sp, #32
(3). Save r0-r3 (parameters a,b,c,d) in stack at –offsets(fp)
      str   r0, [fp, #-24]   // save r0 a
      str   r1, [fp, #-28]   // save r1 b
      str   r2, [fp, #-32]   // save r2 c
      str   r3, [fp, #-36]   // save r3 d
(4). Execute x=1; y=2; z=3; show their locations on stack
      mov   r3, #1
      str   r3, [fp, #-8]    // x=1 at  -8(fp);
      mov   r3, #2
      str   r3, [fp, #-12]   // y=2 at -12(fp)
      mov   r3, #3
      str   r3, [fp, #-16]   // z=3 at -16(fp)
(5). Prepare to call g(x,y)
      ldr   r0, [fp, #-8]    // r0 = x
      ldr   r1, [fp, #-12]   // r1 = y
      bl    g                // call g(x,y)
(6). Compute a+e as return value in r0
      ldr   r2, [fp, #-24]   // r2 = a (saved at -24(fp)
      ldr   r3, [fp, #4]     // r3 = e        at  +4(fp)
      add   r3, r2, r3       // r3 = a+e
      mov   r0, r3           // r0 = return value in r0
(7). Return to caller
      sub   sp, fp, #4       // sp=fp-4 (point at saved FP)
      ldmfd sp!, {fp, pc}    // return to caller
```

When calling the function func(), the caller must pass (6) parameters (a, b, c, d, e, f) to the called function. The first 4 parameters (a, b, c, d) are passed in registers r0–r3. Any extra parameters are passed via the stack. When control enters the called function, the stack top contains the extra parameters (in reverse order). For this example, the stack top contains the 2 extra parameters e and f. The initial stack looks like the following.

```
High       SP                    low
----|---|---|--------------------
    | f | e |
----|exParam|--------------------
```

(1). Upon entry, the called function first establishes the stack frame by pushing LR, FP into stack and letting FP (r12) point at the saved LR.
(2). Then it shifts SP downward (toward low address) to allocate space for local variables and temporary working area, etc. The stack becomes

```
                  SP                                  SP
High       |-push->|---- subtract SP by #32 ------>|----- Low
---|+8 |+4 | 0 |-4 |-8 |-12|-16|-20|-24|-28|-32|-36|
   | f | e |LR |FP |   |   |   |   |   |   |   |   |
---|exParam|-|-|---| local variables, working space|-----
             FP
```

In the diagram, the (byte) offsets are relative to the location pointed by the FP register. While execution is inside a function, the extra parameters (if any) are at [fp, +offset], local variables and saved parameters are at [fp, –offset], all are referenced by using FP as the base register. From the assembly code lines (3) and (4), which save the first 4 parameters passed in r0-r3 and assign values to local variables x, y, z, we can see that the stack contents become as shown in the next diagram.

```
High                                                SP  Low
----|+8 |+4 | 0 |-4 |-8 |-12|-16|-20|-24|-28|-32|-36|---
    | f | e |LR |FP | x | y | z | ? | a | b | c | d |
----|exParam|-|-|---|- locals --|---| saved r0-r3 --|---
             FP
    |< ----------- stack frame of func ----------- >|
```

Although the stack is a piece of contiguous memory, logically each function can only access a limited area of the stack. The stack area visible to a function is called the **stack frame** of the function, and FP (r12) is called the **stack frame pointer**.

At the assembly code lines (5), the function calls g(x, y) with only two parameters. It loads x into r0, y into r1 and then BL to g.

At the assembly code lines (6), it computes a + e as the return value in r0.

At the assembly code lines (7), it deallocates the space in stack, pops the saved FP and LR into PC, causing execution return to the caller.

The ARM C compiler generated code only uses r0–r3. If a function defines any register variables, they are assigned the registers r4–r11, which are saved in stack first and restored later when the function returns. If a function does not call out, there is no need to save/restore the link register LR. In that case, the ARM C compiler generated code does not save and restore the link register LR, allowing faster entry/exit of function calls.

2.7.3.3 Long Jump

In a sequence of function calls, such as

```
main() -> A() -> B()->C();
```

when a called function finishes, it normally returns to the calling function, e.g. C() returns to B(), which returns to A(), etc. It is also possible to return directly to an earlier function in the calling sequence by a long jump. The following program demonstrates long jump in Unix/Linux.

```
/***** longjump.c demonstrating long jump in Linux *****/
  #include <stdio.h>
  #include <setjmp.h>
jmp_buf env;       // for saving longjmp environment
main()
 {
   int r, a=100;
   printf("call setjmp to save environment\n");
   if ((r = setjmp(env)) == 0){
      A();
      printf("normal return\n");
   }
   else{
```

```
      printf("back to main() via long jump, r=%d a=%d\n", r, a);
   }
}
int A()
{  printf("enter A()\n");
    B();
    printf("exit A()\n");
}
int B()
{
  printf("enter B()\n");
  printf("long jump? (y|n) ");
  if (getchar()=='y')
     longjmp(env, 1234);
  printf("exit B()\n");
}
```

In the above longjump.c program, the main() function first calls setjmp(), which saves the current execution environment in a jmp_buf structure and returns 0. Then it proceeds to call A(), which calls B(). While in the function B(), if the user chooses not to return by long jump, the functions will show the normal return sequence. If the user chooses to return by longjmp(env, 1234), execution will return to the last saved environment with a nonzero value. In this case, it causes B() to return to main() directly, bypassing A().

The principle of long jump is very simple. When a function finishes, it returns by the (callerLR, callerFP) in the current stack frame, as shown in the following diagram.

```
          ------------------------------------------
          |params|callerLR|callerFP|.............|
          ------------|--------------------------|---
                   CPU.FP                      CPU.SP
```

If we replace (callerLR, callerFP) with (savedLR, savedFP) of an earlier function in the calling sequence, execution would return to that function directly. For example, we may implement setjmp(int env[2]) and longjmp(int env[2], int value) in assembly as follows.

```
        .global setjmp, longjmp
setjmp:  // int setjmp(int env[2]); save LR, FP in env[2]; return 0
         stmfd sp!, {fp, lr}
         add   fp, sp, #4
         ldr   r1, [fp]        // caller's return LR
         str   r1, [r0]        // save LR in env[0]
         ldr   r1, [fp, #-4]   // caller's FP
         str   r1, [r0, #4]    // save FP in env[1]
         mov   r0, #0          // return 0 to caller
         sub   sp, fp, #4
         ldmfd sp!, {fp, pc}

longjmp: // int longjmp(int env[2], int value)
         stmfd sp!, {fp, lr}
         add   fp, sp, #4
         ldr   r2, [r0]         // return function's LR
         str   r2, [fp]         // replace saved LR in stack frame
         ldr   r2, [r0, #4]     // return function's FP
         str   r2, [fp, #-4]    // replace saved FP in stack frame
         mov   r0, r1           // return value
         sub   sp, fp, #4
```

```
        ldmfd sp!, {fp, pc}    // return via REPLACED LR and FP
```

Long jump can be used to abort a function in a calling sequence, causing execution to resume to a known environment saved earlier. In addition to the (savedLR, savedFP), setjmp() may also save other CPU registers and the caller's SP, allowing longjmp() to restore the complete execution environment of the original function. Although rarely used in user mode programs, long jump is a common technique in systems programming. For example, it may be used in a signal catcher to bypass a user mode function that caused an exception or trap error. We shall demonstrate this technique later in Chap. 8 on signals and signal processing.

2.7.3.4 Call Assembly Function from C

The next example program C2.3 shows how to call assembly function from C. The main() function in C calls the assembly function sum() with 6 parameters, which returns the sum of all the parameters. In accordance with the calling convention of C, the main() function passes the first 4 parameters a, b, c, d in r0–r3 and the remaining parameters e, f, on stack. Upon entry to the called function, the stack top contains the parameters e, f, in the order of increasing addresses. The called function first establish the stack frame by saving LR, FP on stack and letting FP (r12) point at the save LR. The parameters e and f are now at FP + 4 and FP + 8, respectively. The sum function simply adds all the parameters in r0 and returns to the caller.

```
(1)./************  t.c file of Program C2.3 ***********/
int g;                          // un-initialized global
int main()
{
    int a, b, c, d, e, f;       // local variables
    a = b = c =d =e = f = 1;  // values do not matter
    g = sum(a,b,c,d,e,f);     // call sum(), passing a,b,c,d,e,f
}

(2)./*********** ts.s file of Program C2.3 **********/
       .global start, sum
start:  ldr sp, =stack_top
        bl  main           // call main() in C
stop:   b stop

sum:  // int sum(int a,b,c,d,e,f){ return a+b+e+d+e+f;}
// upon entry, stack top contains e, f, passed by main()in C
// Establish stack frame
      stmfd sp!, {fp, lr}  // push fp, lr
      add   fp, sp, #4     // fp -> saved lr on stack
// Compute sum of all (6) parameters
      add   r0, r0, r1     // first 4 parameters are in r0-r3
      add   r0, r0, r2
      add   r0, r0, r3
      ldr   r3, [fp, #4]   // load e into r3
      add   r0, r0, r3     // add to sum in r0
      ldr   r3, [fp, #8]   // load f into r3
      add   r0, r0, r3     // add to sum in r0
// Return to caller
      sub   sp, fp, #4     // sp=fp-4 (point at saved FP)
      ldmfd sp!, {fp, pc}  // return to caller
```

It is noted that in the C2.3 program, the sum() function does not save r0–r3 but use them directly. Therefore, the code should be more efficient than that generated by the ARM GCC compiler. Does this mean we should write all programs in assembly? The answer is, of course, a resounding NO. It should be easy for the reader to figure out the reasons.

(3). **Compile-link t.c and ts.s into an executable file**

```
arm-none-eabi-as –o ts.o ts.s              # assemble ts.s
arm-none-eabi-gcc –c t.c                   # compile t.c into t.o
arm-none-eabi-ld –T t.ld –o t.elf t.o ts.o  # link to t.elf file
arm-none-eabi-objcopy –O binary t.elf t.bin # convert t.elf to t.bin
```

(4). **Run t.bin and check results on an ARM virtual machine as before.**

2.7.3.5 Call C Function from Assembly

Calling C functions with parameters from assembly is also easy if we follow the calling convention of C. The following program C2.4 shows how to call a C function from assembly.

```
/*********** t.c file of Program 2.4 ****************/
int sum(int x, int y){ return x + y; }   // t.c file

/*********** ts.s file of Program C2.4 *************/
     .text
     .global start, sum
start:
      ldr sp,  = stack_top // need a stack to make calls
      ldr r2, =a
      ldr r0, [r2]       // r0 = a
      ldr r2, =b
      ldr r1, [r2]       // r1 = b
      bl sum             // c = sum(a,b)
      ldr r2, =c
      str r0, [r2]       // store return value in c
stop: b   stop

     .data
a:   .word 1
b:   .word 2
c:   .word 0
```

2.7.3.6 Inline Assembly

In the above examples, we have written assembly code in a separate file. Most ARM tool chains are based on GCC. The GCC compiler supports inline assembly, which is often used in C code for convenience. The basic format of inline assembly is

```
__asm__("assembly code");  or simply  asm("assembly code");
```

If the assembly code has more than one line, the statements are separated by \n\t; as in

```
asm("mov %r0, %r1\n\t;  add %r0,#10,r0\n\t");
```

Inline assembly code can also specify operands. The template of such inline assembly code is

```
asm ( assembler template
  : output operands
  : input operands
  : list of clobbered registers
  );
```

The assembly statements may specify output and input operands, which are referenced as %0, %1. For example, in the following code segment,

```
int a, b=10;
asm("mov %1,%%r0; mov %%r0,%0;"  // use %%REG for registers
     :"=r"(a)                    // output MUST have =
     :"r"(b)                     // input
     :"%r0"                      // clobbered registers
   );
```

In the above code segment, %0 refers to a, %1 refers to b, %%r0 refers to the r0 register. The constraint operator "r" means to use a register for the operand. It also tells the GCC compiler that the r0 register will be clobbered by the inline code. Although we may insert fairly complex inline assembly code in a C program, overdoing it may compromise the readability of the program. In practice, inline assembly should be used only if the code is very short, e.g. a single assembly instruction or the intended operation involves a CPU control register. In such cases, inline assembly code is not only clear but also more efficient than calling an assembly function.

2.8 Device Drivers

The emulated ARM Versatilepb board is a virtual machine. It behaves just like a real hardware system, but there are no drivers for the emulated peripheral devices. In order to do any meaningful programming, whether on a real or virtual system, we must implement device drivers to support basic I/O operations. In this book, we shall develop drivers for the most commonly used peripheral devices by a series of programming examples. These include drivers for UART serial ports, timers, LCD display, keyboard and the Multimedia SD card, which will be used later as a storage device for file systems. A practical device driver should use interrupts. We shall show interrupt-driven device drivers in Chap. 3 when we discuss interrupts and interrupts processing. In the following, we shall show a simple UART driver by polling and a LCD driver, which does not use interrupts. In order to do these, it is necessary to know the ARM Versatile system architecture.

2.8.1 System Memory Map

The ARM system architecture uses memory-mapped-I/O. Each I/O device is assigned a block of contiguous memory in the system memory map. Internal registers of each I/O device are accessed as offsets from the device base address. Table 2.1 shows the (condensed) memory map of the ARM Versatile/926EJ-S board (ARM 926EJ-S 2016). In the memory map, I/O devices occupy a 2 MB area beginning from 256 MB.

2.8.2 GPIO Programming

Most ARM based system boards provide General Purpose Input-Output (GPIO) pins as I/O interface to the system. Some of the GPIO pins can be configured for inputs. Other pins can be configured for outputs. In many beginning level embedded system courses, the programming assignments and course project are usually to program the GPIO pins of a small embedded system board to interface with some real devices, such as switches, sensors, LEDs and relays, etc. Compared with other I/O devices, GPIO programming is relatively simple. A GPIO interface, e.g. the LPC2129 GPIO MCU used in many early embedded system boards, consists of four 32-bit registers.

GPIODIR:	set pin direction; 0 for input, 1 for output
GPIOSET:	set pin voltage level to high (3.3 V)
GPIOCLR:	set pin voltage level to low (0 V)
GPIOPIN:	read this register returns the states of all pins

Table 2.1 Memory map of ARM versatile/ARM926EJ-S

MPMC Chip Select 0, 128 MB SRAM	0x00000000	128 MB
MPMC Chip Select 1, 128 MB expansion SRAM	0x08000000	128 MB
System registers	0x10000000	4 KB
Secondary Interrupt Controller (SIC)	0x10003000	4 KB
Multimedia Card Interface 0 (MMCID)	0x10005000	4 KB
Keyboard/Mouse Interface 0 (keyboard)	0x10006000	4 KB
Reserved (UART3 Interface)	0x10009000	4 KB
Ethernet Interface	0x10010000	64 KB
USB Interface	0x10020000	64 KB
Color LCD Controller	0x10120000	64 KB
DMA Controller	0x10130000	64 KB
Vectored Interrupt Controller (PIC)	0x10140000	64 KB
System Controller	OxlOlEOOOO	4 KB
Watchdog Interface	OxlOlElOOO	4 KB
Timer modules 0 and 1 interface	0xl01E2000	4 KB
(Timer 1 at 0xl01E2020)	0xl01E2FFF	
Timer modules 2 and 3 interface	0X101E3000	4 KB
(Timer 3 at 0xl01E3020)	0X101E3FFF	
GPIO Interface (port 0)	0X101E4000	4 KB
GPIO Interface (port 1)	0xl01E5000	4 KB
GPIO Interface (port 2)	0X101E6000	4 KB
UART 0 Interface	OxlOlElOOO	4 KB
UART 1 Interface	0X101E2000	4 KB
UART 2 Interface	0xl01F3000	4 KB
SSMC static expansion memory	0x20000000	256 MB

The GPIO registers can be accessed as word offsets from a (memory mapped) base address. In the GPIO registers, each bit corresponds to a GPIO pin. Depending on the direction setting in the IODIR, each pin can be connected to an appropriate I/O device.

As a specific example, assume that we want to use the GPIO pin0 for input, which is connected to a (de-bounced switch), and pin1 for output, which is connected to the (ground side) of an LED with its own +3.3 V voltage source and a current-limiting resistor. We can program the GPIO registers as follows.

GPIODIR: bit0=0 (input), bit1=1 (output);
GPIOSET: all bits=0 (no pin is set to high);
GPIOCLR: bit1=1 (set to LOW or ground);
GPIOPIN: read pin state, check pin0 for any input.

Similarly, we may program other pins for desired I/O functions. Programming the GPIO registers can be done in either assembly code or C. Given the GPIO base address and the register offsets, it should be fairly easy to write a GPIO control program, which

- turn on the LED if the input switch is pressed or closed, and
- turn off the LED if the input switch is released or open.

We leave this and other GPIO programming cases as exercises in the Problem section. In some systems, the GPIO interface may be more sophisticated but the programming principle remains the same. For example, on the ARM Versatile-PB board, GPIO interfaces are arranged in separate groups called ports (Port0 to Port2), which are at the base

addresses 0x101E4000-0x101E6000. Each port provides 8 GPIO pins, which are controlled by a (8-bit) GPIODIR register and a (8-bit) GPIODATA register. Instead of checking the input pin states, GPIO inputs may use interrupts. Although interesting and inspiring to students, GPIO programming can only be performed on real hardware systems. Since the emulated ARM VMs do not have GPIO pins, we can only describe the general principles of GPIO programming. However, all the ARM VMs support a variety of other I/O devices. In the following sections, we shall show how to develop drivers for such devices.

2.8.3 UART Driver for Serial I/O

Relying on the QEMU monitor commands to display register and memory contents is very tedious. It would be much better if we can develop device drivers to do I/O directly. In the next example program, we shall write a simple UART driver for I/O on emulated serial terminals. The ARM Versatile board supports four PL011 UART devices for serial I/O (ARM PL011 2016). Each UART device has a base address in the system memory map. The base addresses of the 4 UARTs are

```
UART0: 0x101F1000
UART1: 0x101F2000
UART2: 0x101F3000
UART3: 0x10090000
```

Each UART has a number of registers, which are byte offsets from the base address. The following lists the most important UART registers.

```
0x00 UARTDR   Data register: for read/write chars
0x18 UARTFR   Flag register: TxEmpty, RxFull, etc.
0x24 UARIBRD  Baud rate register: set baud rate
0x2C UARTLCR  Line control register: bits per char, parity, etc.
0x38 UARTIMIS Interrupt mask register for TX and RX interrupts
```

In general, a UART must be initialized by the following steps.

(1). Write a divisor value to the baud rate register for a desired baud rate. The ARM PL011 technical reference manual lists the following integer divisor values (based on 7.38 MHz UART clock) for the commonly used baud rates:

```
0x4 = 1152000, 0xC = 38400, 0x18 = 192000, 0x20 = 14400, 0x30 = 9600
```

(2). Write to Line Control register to specify the number of bits per char and parity, e.g. 8 bits per char with no parity.
(3). Write to Interrupt Mask register to enable/disable RX and TX interrupts

When using the emulated ARM Versatilepb board, it seems that QEMU automatically uses default values for both baud rate and line control parameters, making steps (1) and (2) either optional or unnecessary. In fact, it is observed that writing any value to the integer divisor register (0x24) would work but this is not the norm for UARTs in real systems. For the emulated Versatilepb board, all we need to do is to program the Interrupt Mask register (if using interrupts) and check the Flag register during serial I/O. To begin with, we shall implement the UART I/O by polling, which only checks the Flag status register. Interrupt-driven device drivers will be covered later in Chap. 3 when we discuss interrupts and interrupts processing. When developing device drivers, we may need assembly code in order to access CPU registers and the interface hardware. However, we shall use assembly code only if absolutely necessary. Whenever possible, we shall implement the driver code in C, thus keeping the amount of assembly code to a minimum. The UART driver and test program, C2.5, consists of the following components.

(1). ts.s file: when the ARM CPU starts, it is in the Supervisor or SVC mode. The ts.s file sets the SVC mode stack pointer and calls main() in C.

```
       .global start, stack_top  // stack_top defined in t.ld
start:
        ldr sp, =stack_top // set SVC mode stack pointer
        bl  main           // call main() in C
        b .                // if main() returns, just loop
```

(2). t.c file: this file contains the main() function, which initializes the UARTs and uses UART0 for I/O on the serial port.

```
/************* t.c file of C2.5 **************/
int v[] = {1,2,3,4,5,6,7,8,9,10}; // data array
int sum;


#include "string.c"  // contains strlen(), strcmp(), etc.
#include "uart.c"    // UART driver code file


int main()
{
  int i;
  char string[64];
  UART *up;
  uart_init();      // initialize UARTs
  up = &uart[0];    // test UART0
  uprints(up, "Enter lines from serial terminal 0\n\r");
  while(1){
     ugets(up, string);
     uprints(up, "   ");
     uprints(up, string);
     uprints(up, "\n\r");
     if (strcmp(string, "end")==0)
        break;
  }
  uprints(up, "Compute sum of array:\n\r");
  sum = 0;
  for (i=0; i<10; i++)
    sum += v[i];
  uprints(up, "sum = ");
  uputc(up, (sum/10)+'0'); uputc(up, (sum%10)+'0');
  uprints(up, "\n\rEND OF RUN\n\r");
}
```

(3). uart.c file: this file implements a simple UART driver. The driver uses the UART data register for input/output chars, and it checks the flag register for device readiness. The following lists the meaning of the UART register contents.

Data register (offset 0x00): data in (READ)/data out (WRITE)
Flag register (offset 0x18): status of UART port

```
        7    6    5    4    3     2     1     0
      | TXFE RXFF TXFF RXFE BUSY   -     -     -  |
```

where TXFE=Tx buffer empty, RXFF=Rx buffer full, TXFF=Tx buffer full,
RXFE=Rx buffer empty, BUSY=device busy.

A standard way to access the individual bits of a register is to define them as symbolic constants, as in

```
#define TXFE 0x80
#define RXFF 0x40
#define TXFF 0x20
#define RXFE 0x10
#define BUSY 0x08
```

Then use them as bit masks to test the various bits of the flag register. The following shows the UART driver code.

```
/******** uart.c file of C2.5 : UART Driver Code ********/
/*** bytes offsets of UART registers from char *base ***/
#define UDR   0x00
#define UFR   0x18
typedef volatile struct uart{
  char *base;            // base address; as char *
  int  n;                // uart number 0-3
}UART;
UART uart[4];            // 4 UART structures

int uart_init()          // UART initialization function
{
  int i; UART *up;
  for (i=0; i<4; i++){   // uart0 to uart2 are adjacent
    up = &uart[i];
    up->base = (char *)(0x101F1000 + i*0x1000);
    up->n = i;
  }
  uart[3].base = (char *)(0x10009000); // uart3 at 0x10009000
}

int ugetc(UART *up)          // input a char from UART pointed by up
{
  while (*(up->base+UFR) & RXFE);    // loop if UFR is REFE
  return *(up->base+UDR);           // return a char in UDR
}

int uputc(UART *up, charc)           // output a char to UART pointed by up
{
  while (*(up->base+UFR) & TXFF);   // loop if UFR is TXFF
  *(up->base+UDR) = c;              // write char to data register
}

int upgets(UART *up, char *s)      // input a string of chars
{
  while ((*s = ugetc(up)) != '\r') {
    uputc(up, *s);
    s++;
  }
  *s = 0;
}

int uprints(UART *up, char *s)        // output a string of chars
{
  while (*s)
    uputc(up, *s++);
}
```

(4). link-script file: The linker script file, t.ld, is the same as in the program C2.2.
(5). mk and run script file: The mk script file is also the same as in C2.2. For one serial port, the run script is

qemu-system-arm -M versatilepb -m 128M -kernel t.bin **-serial mon:stdio**

For more serial ports, add –serial /dev/pts/1 –serial /dev/pts/2, etc. to the command line. Under Linux, open xterm(s) as pseudo terminals. Enter the Linux ps command to see the pts/n numbers of the pseudo terminals, which must match the pts/n numbers in the –serial /dev/pts/n option of QEMU. On each pseudo terminal, there is a Linux sh process running, which will grab all the inputs to the terminal. To use a pseudo terminal as serial port, the Linux sh process must be made inactive. This can be done by entering the Linux sh command

sleep 1000000

which lets the Linux sh process sleep for a large number of seconds. Then the pseudo terminal can be used as a serial port of QEMU.

2.8.3.1 Demonstration of UART Driver

In the uart.c file, each UART device is represented by a UART data structure. As of now, the UART structure only contains a base address and a unit ID number. During UART initialization, the base address of each UART structure is set to the physical address of the UART device. The UART registers are accessed as *(up->base+OFFSET) in C. The driver consists of 2 basic I/O functions, ugetc() and uputc().

(1). int ugetc(UART *up): this function returns a char from the UART port. It loops until the UART flag register is no longer RXFE, indicating there is a char in the data register. Then it reads the data register, which clears the RXFF bit and sets the RXFE bit in FR, and returns the char.
(2). int uputc(UART *up, c): this function outputs a char to the UART port. It loops until the UART's flag register is no longer TXFF, indicating the UART is ready to transmit another char. Then it writes the char to the data register for transmission out.

The functions ugets() and uprints() are for I/O of strings or lines. They are based on ugetc() and uputc(). This is the typical way of how I/O functions are developed. For example, with gets(), we can implement an int itoa(char *s) function which converts a sequence of numerical digits into an integer. Similarly, with putc(), we can implement a printf() function for formatted printing, etc. We shall develop and demonstrate the printf() function in the next section on LCD driver. Figure 2.6 shows the outputs of running the C2.5 program, which demonstrates UART drivers.

2.8.3.2 Use TCP/IP Telnet Session as UART Port

In addition to pseudo terminals, QEMU also supports TCP/IP telnet sessions as serial ports. First run the program as

```
qemu-system-arm -M versatilepb -m 128M -kernel t.bin \
-serial telnet:localhost:1234,server
```

When QEMU starts, it will wait until a telnet connection is made. From another (X-window) terminal, enter **telnet localhost 1234** to connect. Then, enter lines from the telnet terminal.

```
Enter lines from serial termianl 0
test UART driver   test UART driver
a new test line   a new test line
end   end
Compute sum of 10 integers:
sum = 55
END OF RUN
```

Fig. 2.6 Demonstration of UART driver program

2.8.4 Color LCD Display Driver

The ARM Versatile board supports a color LCD display, which uses the ARM PL110 Color LCD controller (ARM PrimeCell Color LCD Controller PL110, ARM Versatile Application Baseboard for ARM926EF-S). On the Versatile board, the LCD controller is at the base address 0x10120000. It has several timing and control registers, which can be programmed to provide different display modes and resolutions. To use the LCD display, the controller's timing and control registers must be set up properly. ARM's Versatile Application Baseboard manual provides the following timing register settings for VGA and SVGA modes.

```
Mode  Resolution  OSC1    timeReg0    timeReg1    timeReg2
-------------------------------------------------------------
VGA   640x480    0x02C77  0x3F1F3F9C  0x090B61DF  0x067F1800
SVGA  800x600    0x02CAC  0x1313A4C4  0x0505F6F7  0x071F1800
-------------------------------------------------------------
```

The LCD's frame buffer address register must point to a frame buffer in memory. With 24 bits per pixel, each pixel is represented by a 32-bit integer, in which the low 3 bytes are the BGR values of the pixel. For VGA mode, the needed frame buffer size is 1220 KB bytes. For SVGA mode, the needed frame buffer size is 1895 KB. In order to support both VGA and SVGA modes, we shall allocate a frame buffer size of 2 MB. Assuming that the system control program runs in the lowest 1 MB of physical memory, we shall allocate the memory area from 2 to 4 MB for the frame buffer. In the LCD Control register (0x1010001C), bit0 is LCD enable and bit11 is power-on, both must be set to 1. Other bits are for byte order, number of bits per pixel, mono or color mode, etc. In the LCD driver, bits3–1 are set to 101 for 24 bits per pixel, all other bits are 0s for little-endian byte order by default. The reader may consult the LCD technical manual for the meanings of the various bits. It should be noted that, although the ARM manual lists the LCD Control register at 0x1C, it is in fact at 0x18 on the emulated Versatilepb board of QEMU. The reason for this discrepancy is unknown.

2.8.4.1 Display Image Files

As a memory mapped display device, the LCD can display both images and text. It is actually much easier to display images than text. The principle of displaying images is rather simple. An image consists of H (height) by W (width) pixels, where H<=480 and W<=640 (for VGA mode). Each pixel is specified by a 3-byte RGB color values. To display an image, simply extract the RGB values of each pixel and write them to the corresponding pixel location in the display frame buffer. There are many different image file formats, such as BMP, JPG, PNG, etc. Applications for Microsoft Windows typically use BMP images. JPG images are popular with Internet Web pages due to their smaller size. Each image file has a header which contains information of the image. In principle, it should be fairly easy to read the file header and then extract the pixels of the image file. However, many image files are often in compressed format, e.g. JPG files, which must be uncompressed first. Since our purpose here is to show the LCD display driver, rather than manipulation of image files, we shall only use 24-bit color BMP files due to their simple image format. Table 2.2 shows the format of BMP files.

A 24-bit color BMP image file is uncompressed. It begins with a 14-byte file header, in which the first two bytes are BMP file signature 'M' and 'B', indicating that it is a BMP file. Following the file header is a 40-byte image header, which contains the width (W) and height (H) of the image in number of pixels at the byte offsets 18 and 22, respectively. The image header also contains other information, which can be ignored for simple BMP files. Immediately following the image header are 3-byte BGR values of the image pixels arranged in H rows. In a BMP file, the image is stored upside down. The first row in the image file is actually the bottom row of the image. Each row contains (W*3) raised to a multiple of 4 bytes. The example program reads BMP images and displays them to the LCD screen. Since the LCD can only display 640x480 pixels in VGA mode, larger images can be displayed in reduced size, e.g. 1/2 or 1/4 of their original size.

2.8.4.2 Include Binary Data Sections

Raw image files can be included as binary data sections in an executable image. Assume that IMAGE is a raw image file. The following steps show how to include it as a binary data section in an executable image.

(1). Convert raw data into object code by objcopy

```
arm-none-eabi-objcopy -I binary -O elf32-littlearm -B arm IMAGE image.o
```

Table 2.2 BMP file format

Offset	Size	Description
---------- 14-byte file header ----------		
0	2	Signature ('MB')
2	4	Size of BMP file in bytes
6	2	Reserved (0)
8	2	Reserved (0)
10	4	Offset to start of image data in bytes
---------- 40-byte image header ----------		
14	4	Size of image header (40)
18	4	Image width in pixels
22	4	Image height in pixels
26	2	Number of image planes (1)
28	2	Number of bits per pixel (24)
---------- Other fields ----------		
50	4	Number of important colors (0)
---------- Rows of image image ----------		
54 to end of file: rows of images		

(2). Include object code as binary data sections in linker script

```
#---- linker script file t.ld -------
ENTRY(resset_start)
SECTIONS
{  . = 0x10000;
   .text : { ts.o  *( .text) }
   .data : { *(.data) }
   .bss  : { *(.bss)  }
   .data : { *(image.o) } /* include image.o as a data section */
    /* stack areas */
}
```

2.8.4.3 Programming Example C2.6: LCD Driver

The example program C2.6 implements an LCD driver which displays raw image files. The program consists of the following components.

(1). The ts.s file: Since the driver program does not use interrupts, nor tries to handle any exceptions, there is no need to install the exception vectors. Upon entry, it sets up the SVC mode stack and calls main() in C.

```
/*********** ts.s file of C2.6 *********/
      .global reset_start
reset_start:
  LDR sp, =stack_top   // set SVC stack pointer
  BL main
  B .
```

(2). The vid.c file: This is the LCD driver. It initializes the LCD registers to VGA mode of 640x480 resolutions and sets the frame buffer at 2 MB. It also includes code for SVGA mode with 800x600 resolutions but they are commented out.

```
/************** vid.c file of C2.6 ******************/
int volatile *fb;
int WIDTH = 640; // default to VGA mode for 640x480
int fbuf_init(int mode)
{
   fb = (int *)(0x200000); // at 2 MB to 4 MB
   //***************** for 640x480 VGA ***************/
    *(volatile unsigned int *)(0x1000001c) = 0x2C77;
    *(volatile unsigned int *)(0x10120000) = 0x3F1F3F9C;
    *(volatile unsigned int *)(0x10120004) = 0x090B61DF;
    *(volatile unsigned int *)(0x10120008) = 0x067F1800;
   }
    /***************** for 800X600 SVGA ***************
    *(volatile unsigned int *)(0x1000001c) = 0x2CAC;
    *(volatile unsigned int *)(0x10120000) = 0x1313A4C4;
    *(volatile unsigned int *)(0x10120004) = 0x0505F6F7;
    *(volatile unsigned int *)(0x10120008) = 0x071F1800;
   }
    ************************************************/
   *(volatile unsigned int *)(0x10120010) = 0x200000; // fbuf
   *(volatile unsigned int *)(0x10120018) = 0x82B;
}
```

(3). uart.c file: This is the same UART driver in Example C2.5, except that it uses the basic uputc() function to implement a uprintf() function for formatted printing.
(4). The t.c file: This file contains the main() function, which calls the show_bmp() function to display images. In the linker script, two image files, image1 and image2, are included as binary data sections in the executable image. The start position of an image file can be accessed by the symbols _binary_imageI_start generated by the linker.

```
/************** t.c file of program C2.6 ******************/
#include "defines.h"   // device base addresses, etc.
#include "vid.c"       // LCD driver
#include "uart.c"      // UART driver
extern char _binary_image1_start, _binary_image2_start;
#define WIDTH 640
int show_bmp(char *p, int start_row, int start_col)
{
   int h, w, pixel, rsize, i, j;
   unsigned char r, g, b;
   char *pp;
   int *q = (int *)(p+14);    // skip over 14-byte file header
   w = *(q+1);                // image width in pixels
   h = *(q+2);                // image height in pixels
   p += 54;                   // p-> pixels in image
   //BMP images are upside down, each row is a multiple of 4 bytes
   rsize = 4*((3*w + 3)/4);   // multiple of 4
   p += (h-1)*rsize;          // last row of pixels
   for (i=start_row; i<start_row + h; i++){
     pp = p;
     for (j=start_col; j<start_col + w; j++){
```

```
       b = *pp; g = *(pp+1); r = *(pp+2); // BRG values
       pixel = (b<<16) | (g<<8) | r;      // pixel value
       fb[i*WIDTH + j] = pixel; // write to frame buffer
       pp += 3;               // advance pp to next pixel
     }
     p -= rsize;              // to preceding row
   }
   uprintf(\nBMP image height=%d width=%d\n", h, w);
}
int main()
{
   char c,* p;
   uart_init();          // initialize UARTs
   up = upp[0];          // use UART0
   fbuf_init();          // default to VGA mode
   while(1){
     p = &_binary_image1_start;
     show_bmp(p, 0, 80); // display image1
     uprintf("enter a key from this UART : ");
     ugetc(up);
     p = &_binary_image2_start;
     show_bmp(p,120, 0); // display image2
   }
   while(1);             // loop here
}
```

(5). The mk script file: The mk script generates object code for the image files, which are included as binary data sections in the executable image.

 # mk and run script file of C2.6: The only thing new is to convert images to object files.

```
arm-none-eabi-objcopy -I binary -O elf32-littlearm -B arm image1 image1.o
arm-none-eabi-objcopy -I binary -O elf32-littlearm -B arm image2 image2.o

arm-none-eabi-as -mcpu=arm926ej-s ts.s -o ts.o
arm-none-eabi-gcc -c -mcpu=arm926ej-s t.c -o t.o
arm-none-eabi-ld -T t.ld ts.o t.o -o t.elf
arm-none-eabi-objcopy -O binary t.elf t.bin
echo ready to go?
read dummy

qemu-system-arm -M versatilepb -m 128 M -kernel t.bin -serial mon:stdio
```

2.8.4.4 Demonstration of Display Images on LCD

Figure 2.7 shows the sample outputs of running the C2.6 program in VGA mode.

The top part of Fig. 2.7 shows the UART port I/O. The bottom part of the figure shows the LCD display. When the program starts, it first displays image1 to (row = 0, col = 80) on the LCD, and it also prints the image size to UART0. Entering an input key from UART0 will let it display image2 to (row = 120, col = 0), etc. Displaying image files can be very interesting, which is the basis of computer animation. Variations to the image displaying program are listed as exercises in the Problems section for interested readers.

2.8.4.5 Display Text

In order to display text, we need a font file, which specifies the fonts or bit patterns of the ASCII chars. The font file, font.bin, is a raw bitmap of 128 ASCII chars, in which each char is represented by a 8x16 bitmap, i.e. each char is represented by 16 bytes, each byte specifies the pixels of a scan line of the char. To display a char, use the char's ASCII code value (multiplied

by 16) as an offset to access its bytes in the bitmap. Then scan the bits in each byte. For each 0 bit, write BGR = 0x000000 (black) to the corresponding pixel. For each 1 bit, write BRG = 0x111111 (white) to the pixel. Instead of black and white, each char can also be displayed in color by writing different RGB values to the pixels.

Like image files, raw font files can be included as binary data sections in the executable image. An alternative way is to convert bitmaps to char maps first. As an example, the following program, bitmap2charmap.c, converts a font bitmap to a char map.

```
/**** bitmap2charmap.c: run as a.out font.bin  >  font ****/
  #include <stdio.h>
  main(int argc, char *argv[ ])
  {
    int i, n; u8 buf[16];
    FILE *fp = fopen(argv[1], "r");     // fopen file for READ
    while((n = fread(buf, 1, 16, fp)){ // read 16 bytes
      for (i=0; i<n; i++)              // write each byte as 2 hex
         printf("0x%2x ", buf[i]);
    }
    printf("\n");
  }
```

Fig. 2.7 Demonstration of display images on LCD

Unlike raw bitmap files, which must be converted to object files first, char map files are larger in size but they can be included directly in the C code.

2.8.4.6 Color LCD Display Driver Program

The example program C2.7 demonstrates an LCD driver for displaying text. The program consists of the following components.

(1). ts.s file: The ts.s file is the same as in the example program C2.6.
(2). vid.c file: The vid.c file implements a driver for the ARM PL110 LCD display [ARM PL110, 2016]. On the Versatilepb board the base address of the Color LCD is at 0x10120000. Other registers are (u32) offsets from the base address.

```
/********  vid.c file : LCD driver ********/
00  timing0
04  timing1
08  timing2
0C  timing3
10  upperPanelframeBaseAddressRegister // use the upper panel
14  lowerPanelFrameBaseAddressRegister // NOT use lower panel
18  controlRegister // NOTE: QEMU emulated PL110 CR is at 0x18
*****************************************************/
#define RED    0
#define BLUE   1
#define GREEN  2
#define WHITE  3
extern char _binary_font_start;
int color;
u8 cursor;
int volatile *fb;
int row, col, scroll_row;
unsigned char *font;
int WIDTH = 640;  // scan line width, default to 640

int fbuf_init()
{
    int i;
    fb = (int *)0x200000;        // frame buffer at 2MB-4MB
    font = &_binary_font_start;  // font bitmap
    /********* for 640x480 VGA mode ******************/
    *(volatile unsigned int *)(0x1000001c) = 0x2C77;
    *(volatile unsigned int *)(0x10120000) = 0x3F1F3F9C;
    *(volatile unsigned int *)(0x10120004) = 0x090B61DF;
    *(volatile unsigned int *)(0x10120008) = 0x067F1800;
    *(volatile unsigned int *)(0x10120010) = 0x200000; // at 2MB
    *(volatile unsigned int *)(0x10120018) = 0x82B;
    /********** for 800X600 SVGA mode *****************
    *(volatile unsigned int *)(0x1000001c) = 0x2CAC; // 800x600
    *(volatile unsigned int *)(0x10120000) = 0x1313A4C4;
    *(volatile unsigned int *)(0x10120004) = 0x0505F6F7;
    *(volatile unsigned int *)(0x10120008) = 0x071F1800;
    *(volatile unsigned int *)(0x10120010) = 0x200000;
    *(volatile unsigned int *)(0x10120018) = 0x82B;
    **********/
    cursor = 127; // cursor = row 127 in font bitmap
}
```

```
int clrpix(int x, int y)   // clear pixel at (x,y)
{
  int pix = y*640 + x;
  fb[pix] = 0x00000000;
}
int setpix(int x, int y)   // set pixel at (x,y)
{
  int pix = y*640 + x;
  if (color==RED)
     fb[pix] = 0x000000FF;
  if (color==BLUE)
     fb[pix] = 0x00FF0000;
  if (color==GREEN)
     fb[pix] = 0x0000FF00;
}
int dchar(unsigned char c, int x, int y) // display char at (x,y)
{
  int r, bit;
  unsigned char *caddress, byte;
  caddress = font + c*16;
  for (r=0; r<16; r++){
    byte = *(caddress + r);
    for (bit=0; bit<8; bit++){
      if (byte & (1<<bit))
        setpix(x+bit, y+r);
    }
  }
}
int undchar(unsigned char c, int x, int y) // erase char at (x,y)
{
  int row, bit;
  unsigned char *caddress, byte;
  caddress = font + c*16;
  for (row=0; row<16; row++){
    byte = *(caddress + row);
    for (bit=0; bit<8; bit++){
      if (byte & (1<<bit))
        clrpix(x+bit, y+row);
    }
  }
}
int scroll() // scrow UP one line (the hard way)
{
  int i;
  for (i=64*640; i<640*480; i++){
    fb[i] = fb[i + 640*16];
  }
}
int kpchar(char c, int ro, int co) // print char at (row, col)
{
   int x, y;
   x = co*8;
   y = ro*16;
   dchar(c, x, y);
}
```

```
int unkpchar(char c, int ro, int co) // erase char at (row, col)
{
   int x, y;
   x = co*8;
   y = ro*16;
   undchar(c, x, y);
}
int erasechar()  // erase char at (row,col)
{
  int r, bit, x, y;
  unsigned char *caddress, byte;
  x = col*8;
  y = row*16;
  for (r=0; r<16; r++){
     for (bit=0; bit<8; bit++){
        clrpix(x+bit, y+r);
    }
  }
}
int clrcursor() // clear cursor at (row, col)
{
  unkpchar(127, row, col);
}
int putcursor(unsigned char c) // set cursor at (row, col)
{
  kpchar(c, row, col);
}

int kputc(char c)  // print char at cursor position
{
  clrcursor();
  if (c=='\r'){    // return key
    col=0;
    putcursor(cursor);
    return;
  }
  if (c=='\n'){    // new line key
    row++;
    if (row>=25){
      row = 24;
      scroll();
    }
    putcursor(cursor);
    return;
  }
  if (c=='\b'){    // backspace key
    if (col>0){
      col--;
      erasechar();
      putcursor(cursor);
    }
    return;
  }
  // c is ordinary char case
  kpchar(c, row, col);
```

```
  col++;
  if (col>=80){
    col = 0;
    row++;
    if (row >= 25){
      row = 24;
      scroll();
    }
  }
  putcursor(cursor);
}

// The following implements kprintf() for formatted printing
int kprints(char *s)
{
  while(*s){
    kputc(*s);
    s++;
  }
}
int krpx(int x)
{
  char c;
  if (x){
     c = tab[x % 16];
     krpx(x / 16);
  }
  kputc(c);
}
int kprintx(int x)
{
  kputc('0'); kputc('x');
  if (x==0)
    kputc('0');
  else
    krpx(x);
  kputc(' ');
}
int krpu(int x)
{
  char c;
  if (x){
     c = tab[x % 10];
     krpu(x / 10);
  }
  kputc(c);
}
int kprintu(int x)
{
  if (x==0)
    kputc('0');
  else
    krpu(x);
  kputc(' ');
}
```

```
int kprinti(int x)
{
  if (x<0){
    kputc('-');
    x = -x;
  }
  kprintu(x);
}
int kprintf(char *fmt,...)
{
  int *ip;
  char *cp;
  cp = fmt;
  ip = (int *)&fmt + 1;

  while(*cp){
    if (*cp != '%'){
      kputc(*cp);
      if (*cp=='\n')
      kputc('\r');
      cp++;
      continue;
    }
    cp++;
    switch(*cp){
    case 'c': kputc((char)*ip);      break;
    case 's': kprints((char *)*ip);  break;
    case 'd': kprinti(*ip);          break;
    case 'u': kprintu(*ip);          break;
    case 'x': kprintx(*ip);          break;
    }
    cp++; ip++;
  }
}
```

2.8.4.7 Explanations of the LCD Driver Code

The LCD screen may be regarded as a rectangular box consisting of 480x640 pixels. Each pixel has a (x,y) coordinate on the screen. Correspondingly, the frame buffer, u32 fbuf[ ], is a memory area containing 480*640 u32 integers, in which the low 24 bits of each integer represents the BGR values of the pixel. The linear address or index of a pixel in fbuf[ ] at the coordinate (x,y) = (column, row) is given by the Mailman's algorithm (Chap. 2, Wang 2015).

$$\text{pixel\_index} = x + y*640;$$

The basic display functions of the LCD driver are

(1). setpix(x,y): set the pixel at (x, y) to BGR values (by a global color variable).
(2). clrpix(x,y): clear the pixel at (x, y) by setting the BGR to background color (black).
(3). dchar(char, x, y): display char at coordinate (x, y). Each char is represented by a 8x16 bitmap. For a given char value (0 to 127), dchar() fetches the 16 bytes of the char from the bitmap. For each bit in a byte, it calls clrpix(x+bitNum, y +byteNum) to clear the pixel first. This erases the old char, if any, at (x, y). Otherwise, it will display the composite bit patterns of the chars, making it unreadable. Then it calls setpix(x+bitNum, y+byteNum) to set the pixel if the bit is 1.
(4). erasechar(): erase the char at (x, y). For a memory mapped display device, once a char in written to the frame buffer, it will remain there (and hence be rendered on the screen) until it is erased. For ordinary chars, dchar() automatically erases the

original char. For special chars like the cursor, it is necessary to erase it first before moving it to a different position. This is done by the erasechar() operation.
(5). kputc(char c): display a char at the current (row, col) and move the cursor, which may cause scroll-up the screen.
(6). scroll(): scroll screen up or down by one line.
(7). The cursor: When displaying text, the cursor allows the user to see where the next char will be displayed. The ARM LCD controller does not have a cursor generator. In the LCD driver, the cursor is simulated by a special char (ASCII code 127) in which all the pixels are 1's, which defines a solid rectangular box as the cursor. The cursor may be made to blink if it is turned on/off periodically, e.g. every 0.5 s, which requires a timer. We shall show the blinking cursor later in Chap. 3 when we implement timers with timer interrupts. The putcursor() function draws the cursor at the current (row, col) position on the screen, and the erasecursor() function erases the cursor from its current position.
(8). **The printf() Function**

For any output device that supports the basic printing char operation, we can implement a printf() function for formatted printing. The following shows how to develop such a generic printf() function, which can be used for both UART and the LCD display. First, we implement a printu() function, which prints unsigned integers.

```
char *ctable = "0123456789ABCDEF";
int BASE = 10; // for decimal numbers
int rpu(u32 x)
{
    char c;    // local variable
    if (x){
       c = ctable[x % BASE];
       rpu(x / BASE);
       putc(c);
    }
}
int printu(u32 x)
{
   (x==0)? putc('0') : rpu(x);
   putc(' ');
}
```

The function rpu(x) generates the digits of x % 10 in ASCII recursively and prints them on the return path. For example, if x=123, the digits are generated in the order of '3', '2', '1', which are printed as '1', '2', '3' as they should. With printu(), writing a printd() function to print signed integers becomes trivial. By setting BASE to 16, we can print in hex. Assume that we have prints(), printd(), printu() and printx() already implemented. Then we can write a

```
int printf(char *fmt, ...)            // NOTE the 3 dots in function heading
```

function for formatted printing, where fmt is a format string containing conversion symbols %c, %s, %u, %d, %x.

```
int printf(char *fmt, ...) // most C compilers require the 3 dots
{
    char *cp = fmt;              // cp points to the fmt string
    int *ip = (int *)&fmt +1;    // ip points to first item in stack
    while (*cp){                 // scan the format string
       if (*cp != '%'){          // spit out ordinary chars
          putc(*cp);
          if (*cp=='\n')         // for each '\n'
             putc('\r');         // print a  '\r'
          cp++;
          continue;
       }
       cp++;        // cp points at a conversion symbol
```

```
       switch(*cp){ // print item by %FORMAT symbol
          case 'c' :   putc((char  )*ip);  break;
          case 's' : prints((char *)*ip);  break;
          case 'u' : printu((u32   )*ip);  break;
          case 'd' : printd((int   )*ip);  break;
          case 'x' : printx((u32   )*ip);  break;
       }
       cp++; ip++;               // advance pointers
    }
}
```

(5). The t.c file: The t.c file contains the main() function and the show_bmp() function. It first initializes both the UART and LCD drivers. The UART driver is used for I/O from the serial port. For demonstration purpose, it displays outputs to both the serial port and the LCD. When the program starts, it displays a small logo image at the top of the screen. The scroll upper limit is set to a line below the logo image, so that the logo will remain on the screen when the screen is scrolled upward.

```
/************** t.c file of C2.7 *************/
#include "defines.h"
#include "uart.c"
#include "vid.c"
extern char _binary_panda1_start;
int show_bmp(char *p, int startRow, int startCol){// SAME as before}
int main()
{
   char line[64];
   fbuf_init();
   char *p = &_binary_panda1_start;
   show_bmp(p, 0, 0); // display a logo
   uart_init();
   UART *up = upp[0];
   while(1){
     color = GREEN;
     kprintf("enter a line from UART port : ");
     uprintf("enter line from UART : ");
     ugets(up, line);
     uprintf(" line=%s\n", line);
     color = RED;
     kprintf("line=%s\n", line);
   }
}
```

(6). t.ld file: The linker script includes the object files of a font and an image as binary data sections, similar to that of the example program C2.6.
(7). mk and run script file: This is similar to that of C2.6.

2.8.4.8 Demonstration of LCD Driver Program

Figure 2.8 shows the sample outputs of running the example program C2.7. It uses the LCD driver to display both images and text on the LCD screen. In addition, it also uses the UART driver to do I/O from the serial port.

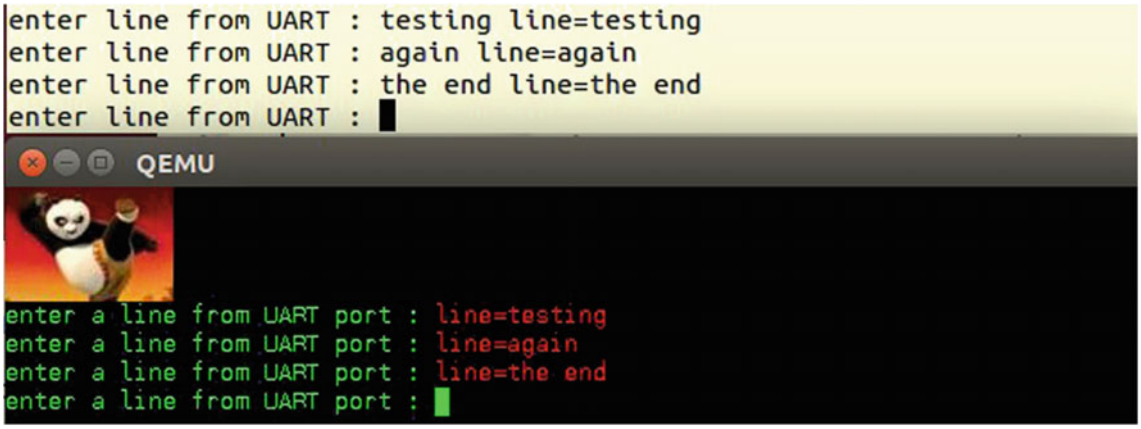


Fig. 2.8 Demonstration of display text on LCD

2.9 Summary

This chapter covers the ARM architecture, ARM instructions, programming in ARM assembly and development of programs for execution on ARM virtual machines. These include ARM processor modes, banked registers in different modes, instructions and basic programming in ARM assembly. Since most software for embedded systems are developed by cross-compiling, it introduces the ARM toolchain, which allows us to develop programs for execution on emulated ARM virtual machines. We choose the Ubuntu (14.04/15.0) Linux as the program development platform because it supports the most complete ARM toolchains. Among the ARM virtual machines, we choose the emulated ARM Versatilepb board under QEMU because it supports many commonly used peripheral devices found in real ARM based systems. Then it shows how to use the ARM toolchain to develop programs for execution on the ARM Versatilepb virtual machine by a series of programming examples. It explains the function call convention in C and shows how to interface assembly code with C programs. Then it develops a simple UART driver for I/O on serial ports, and a LCD driver for displaying both graphic images and text. It also shows the development of a generic printf() function for formatted printing to output devices that support the basic print char operation.

List of Sample Programs

C2.1: ARM assembly programming
C2.2: Sum of integer array in assembly
C2.3: Call assembly function from C
C2.4: Call C function from assembly
C2.5: UART driver
C2.6: LCD driver for displaying images
C2.7: LCD driver for displaying text

Problems

1. The example program C2.2 contains 3 highlighted instructions

```
sub  r2, r2, #1     // r2--
cmp  r2, #0         // if (r2 != 0)
bne  loop           //    goto loop;
```

   (1). If you delete the cmp instruction, the program would not work. Explain why?
   (2). However, replacing the 3 lines with

```
subs  r2, r2, #1
bne loop
```

   would work. Explain why?
2. In the assembly code of the program C2.4, a, b, c are adjacent. Modify the assembly code by letting R2 point at the first word a. Then

(1). Access a, b, c by using R2 as a base register with offsets.
(b). Access a, b, c by using R2 as a base register with post-indexed addressing.

3. In the example program C2.4, instead of defining a, b, c in the assembly code, define them as initialized globals in the t.c file:

$$\text{int } a = 1, \quad b = 2, \quad c = 0;$$

Declare them as global symbols in the ts.s file. Compile and run the program again.

4. GPIO programming: Assume that BASE is the base address of a GPIO, and the GPIO registers are at the offset addresses IODIR, IOSET, IOCLR, IOPIN. The following shows how to define them as constants in ARM assembly code and C.

```
.set BASE, 0x101E4000        // #define BASE 0x101E4000
.set IODIR, 0x000            // #define IODIR 0x000
.set IOSET, 0x004            // #define IOSET 0x004
.set IOCLR, 0x008            // #define IOCLR 0x008
.set IOPIN, 0x00C            // #define IOPIN 0x00C
```

Write a GPIO control program in both assembly and C to perform the following tasks.
(1). Program the GPIO pins as specified in Sect. 2.8.1.
(2). Determine the state of the GPIO pins.
(3). Modify the control program to make the LED blink while the input switch is closed.

5. The example program C2.6 assumes that every image size is h<=640 and width<=480 pixels. Modify the program to handle BMP images of larger sizes by

(1). Cropping: display at most 480x640 pixels.
(2). Shrinking: reduce the image size by a factor, e.g. 2 but keep the same 4:3 aspect ratio.

6. Modify the example program C2.6 to display a sequence of slightly different images to do animation.
7. Many image files, e.g. JPG images, are compressed, which must be uncompressed first. Modify the example program C2.6 to display (compressed) images of different format, e.g. JPG image files.
8. In the LCD display driver program C2.7, define tab_size = 8. Each tab key (\t) expands to 8 spaces. Modify the LCD driver to support tab keys. Test the modified LCD driver by including \t in printf() calls.
9. In the LCD driver of program C2.7, scroll-up one line is implemented by simply copying the entire frame buffer. Devise a more efficient way to implement the scroll operation. HINT: The display memory may be regarded as a circular buffer. To scroll up one line, simply increment the frame buffer pointer by line size.
10. Modify the example program C2.7 to display text with different fonts.
11. Modify the example program C2.8 to implement long jump by using the code of Sect. 2.7.3.3. Use UART0 to get user inputs but display outputs to the LCD. Verify that long jump works.
12. In the LCD driver, the generic printf() function is defined as

```
int printf(char *fmt, ...);            // note the 3 dots
```

In the implementation of printf(), it assumes that all the parameters are adjacent on stack, so that they can be accessed linearly. This seems to be inconsistent with the calling convention of ARM C, which passes the first 4 parameter in r0–r3 and extra parameters, if any, on stack. Compile the printf() function code to generate an assembly code file. Examine the assembly code to verify that the parameters are indeed adjacent on stack.

References

ARM Architectures: http://www.arm.products/processors/instruction-set-architectures, ARM Information Center, 2016
ARM Cortex-A8: "ARM Cortex-A8 Technical Reference Manual", ARM Information Center, 2010
ARM Cortex A9 MPcore: "Cortex A9 MPcore Technical Reference Manual", ARM Information Center, 2016
ARM926EJ-ST: "ARM926EJ-S Technical Reference Manual", ARM Information Center, 2008
ARM926EJ-ST: "Versatile Application Baseboard for ARM926EJ-S User guide", ARM Information Center, 2010
ARM PL011: "PrimeCell UART (PL011) Technical Reference Manual", ARM Information Center, 2016
ARM PrimeCell Color LCD Controller PL110: "ARM Versatile Application Baseboard for ARM926EF-S", ARM Information Center, 2016
ARM Programming: "ARM Assembly Language Programming", http://www.peter-cockerell.net/aalp/html/frames.html
ARM toolchain: http://gnutoolchains.com/arm-eabi, 2016
QEMU Emulators: "QEMU Emulator User Documentation", http://wiki.qemu.org/download/qemu-doc.htm, 2010

http://www.springer.com/978-3-319-51516-8

Embedded and Real-Time Operating Systems

Wang, K.C.

2017, XIX, 481 p. 93 illus., Hardcover
ISBN: 978-3-319-51516-8