# Contents

15 Input/Output – Peripherals

15.1 Polling ......................................................... 2
15.2 Interrupts .................................................... 3
15.3 Interrupt Vector ............................................. 3
15.4 Fast Peripherals ............................................ 4
15.5 Exercises .................................................. 5

16 System Software .............................................. 6

16.1 The Kernel .................................................. 6
16.1.1 Resource Managers .................................... 7
16.2 Exercises .................................................. 7

17 Assemblers and Compilers .................................. 8

17.1 Assembler .................................................. 8
17.2 Compiler ................................................... 9

18 Libraries and Interpreters .................................. 10

18.1 Libraries & Linker ......................................... 10
18.2 Interpreters ................................................ 11
15 Input/Output – Peripherals

The picture of a computer that we have been using so far consists of just a CPU and a main memory linked by a bus, but a computer is no use unless we can communicate with it, via various different kinds of peripherals. Logically, the bus connects everything together:

![Bus Connection Diagram]

Peripherals are very varied:
- mouse, keyboard, screen, speakers
- printer, scanner
- removable disks (floppy, CD, DVD etc.), hard disks, memory sticks
- modem, network
- clock

with many different kinds of connections:
Loosely coupled:
- via external bus (SCSI, USB, Firewire)
- via network (Ethernet, ATM, Airport)
- via port (serial RS232, parallel, PS2)
Tightly coupled:
- via internal bus (graphics)

We need some standard mechanism that can cope with any peripheral. We need to be able to:
- read/write data e.g. characters
- write control information: settings (e.g. speed), commands
- read status information e.g. ready, error

The usual solution is to design the interface of each peripheral so it appears to have its own set of peripheral registers e.g. data register(s), control register(s), status register(s).

These “registers” are usually accessed in one of two different ways:
- using special instructions
- using special addresses, used by normal instructions (e.g. load and store). This is known as memory mapped.

Memory mapping is more flexible, and more common. (using 32-bit addresses allows for 4Gbytes of memory – typical PCs have e.g. 1GB so lots of room for peripherals!)

15.1 Polling

Suppose we have a simple device (e.g. keyboard) – how might we read the next character typed by the user?

```
loop
  ADR R1, Status_Reg ; R1 points to status register
  LDRB R0, [R1] ; read status
  TST R0, #0x80 ; test ready bit (bit 7)
  BEQ loop ; if not ready, try again
  ADR R1, Data_Reg ; R1 points to data register
  LDRB R0, [R1] ; ready, so read data
```

This is known as a polling loop or busy waiting, as the CPU spends all its time asking the peripheral if it is ready yet. The TST instruction is similar to a CMP instruction, in that it does not alter its operands, but just sets flags in the CPU for use in a subsequent conditional instruction. However, it performs a logical and, rather than an arithmetic subtraction, and so can be used to test individual bits e.g. in a status register.
15.2 Interrupts

The problem with a simple polling loop like that above, is that the computer can’t do anything else while it is waiting for the character. Instead, we want some way (of appearing) to do several things at once:

- look after several peripherals
- run a program (or maybe even several programs in turn)

We need to be able to set a program running but **interrupt** it when something significant happens:

- any input peripheral has some input ready
- any output peripheral is ready for the next piece of output
- the program does something wrong
- a timer goes off, to say that the program has been running for long enough
- etc.

Suppose we want to run a program that continuously displays an animation, but reacts to the user pressing various keys on the keyboard. We have to enable the keyboard to interrupt the running program: as soon as the user types a character on the keyboard, this must pause the program, get the character, and then resume the program.

Detailed sequence of events:

- a program is running as normal
- the peripheral **interrupts** the CPU
  - stop the program at the end of the current instruction
  - save (and maybe reset) the value of important registers
  - run a special sequence of instructions that deals with the peripheral – the **interrupt handler**
  - finally, restore the value of the saved registers
- the program can now continue from its next instruction

When an interrupt happens, essentially the CPU makes it look as if a BL instruction has just called the interrupt handler routine.

The interrupt handler must save and restore any registers it uses, otherwise we may get random errors in the running program. CPSR (Current Program Status Register) contains e.g. the result of CMP instructions, which also needs to be saved and restored. To make interrupt handling faster, the ARM contains extra LR, SP and CPSR (actually SPSR – Saved PSR) registers for each kind of interrupt, so if the handler is simple enough, they don’t need to be explicitly saved and restored.

Here is a very simple interrupt handler, based on the previous polling loop example:

```
STMFD SP!,{R0, R1}
ADR R1, Status_Reg
LDRB R0, [R1]
TST R0, #0x80
BEQ error
ADR R1, Data_Reg
LDRB R0, [R1]
STR R0, somewhere
ADR R1, Acknowledge
LDR R0, #1
STR R0, [R1]
LDMD SP!,{R0, R1}
SUBS PC, LR, #4 ; restores PC and CPSR
```

Interrupts normally make better use of CPU time than polling loops, and make it much easier to deal with extra peripherals etc., but it is much harder to write and debug the necessary code, and much more difficult to predict the resulting non-deterministic behaviour.

As you may have noticed in your Java programming, the concept of interrupts has been generalised to **exceptions**, to include any significant event that may interrupt the normal running of a program, whether originating in hardware or in software.

15.3 Interrupt Vector

We can use interrupt-handlers to look after all sorts of peripherals – particularly those that transmit more than a few characters a second – and several peripherals at once. All we need to do is to set the program running, and allow any
peripheral to interrupt to say that it is ready. Then, we pause the program as before, but now we need to decide which peripheral interrupted the CPU, so we can activate the handler for that particular peripheral.

We could go around each peripheral in turn, checking its status, but it would be better if each different peripheral activated a slightly different piece of code that was specifically designed to deal with it. One way of achieving this is to use an **interrupt vector**.

The interrupt vector of the ARM uses the following addresses, each holding a branch instruction to the start of the particular interrupt handler. (The last one, for fast interrupt, usually is the start of the handler, rather than a branch to it, to make the code as fast as possible.)

<table>
<thead>
<tr>
<th>Address</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>reset</td>
</tr>
<tr>
<td>0x4</td>
<td>undefined instruction</td>
</tr>
<tr>
<td>0x8</td>
<td>SWI</td>
</tr>
<tr>
<td>0xC</td>
<td>instruction fetch memory fault</td>
</tr>
<tr>
<td>(0x14: reserved)</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>normal interrupt</td>
</tr>
<tr>
<td>0x1C</td>
<td>fast interrupt</td>
</tr>
</tbody>
</table>

(Note that this means that real ARM programs do not normally start at address 0, as we assumed early on, as this area of memory is occupied by the interrupt vector – this is true for most computers.)

As you can see, ARM only allows for two different peripheral handlers, for fast and for slow peripherals. The assumption is that the ARM CPU will rely on an extra Peripheral Controller chip, which will support many different peripherals and simplify the task of deciding which has caused a particular interrupt. For example, it may provide a status register with a bit allocated to each different peripheral, so the bits can be tested in turn to decide where the interrupt came from.

### 15.4 Fast Peripherals

An interrupt handler might consist of perhaps a few hundred instructions, so to deal with each interrupt takes perhaps a millionth of a typical CPU’s time. A fast peripheral, such as a hard disk, reads and writes tens or hundreds of millions of bytes per second. If the CPU was interrupted for each byte, it would spend most of its time dealing with the disk.

However, whereas it is likely that the CPU has to do something with each character from a keyboard or modem (e.g. if the user is typing into an editor), it is unlikely to need to react to the disk until a complete block (of a few thousand characters) or even a complete file has been transmitted e.g. if a program is being loaded so it can be run, or the editor is saving a file.

Because of this, fast peripherals such as hard disks normally use **Direct Memory Access (DMA)**.

DMA means that while a disk is reading or writing data, the disk’s hardware controller repeatedly takes over for a moment to transfer each byte directly to or from the main memory. This leaves the CPU able to do something else, such as run a program or deal with a slow peripheral. Only when an entire block of data (e.g. 1k bytes) has been transferred will the disk cause a normal interrupt, to let the CPU know and to ask it what to do next.

The result is that the overall computer organisation, and the individual peripheral controllers, have to be more complicated (but fast peripheral controllers are already pretty complicated). However, DMA vastly reduces the amount of work the CPU must do, as it does not have to move each individual byte of data. (Note that the amount of work that the peripheral and the main memory must do is relatively unchanged.)

For slow peripherals, we just need a link between the CPU and the peripheral, to transmit commands from the CPU, status information from the peripheral, and data in either direction.

For fast peripherals, although we still need a link to the CPU for commands and status, we also need a link to the main memory for data. As assumed at the start of the lecture, we need a single bus linking CPU, main memory, and peripherals.
15.5 Exercises

1: How can I optimise the code for the polling loop?

2: Consider the “interrupt” handler for the SWI instruction.
   a) How does it get the number telling it which action to perform?
      (SWI 0 meaning output a character, SWI 1 meaning input a character, etc.)
   b) How does it obey the corresponding piece of code?

3: Think about how we might implement a tiny operating system on the ARM interpreter that you use in the labs (Komodo),
   to provide the same functionality but in a slightly more “realistic” way:
   a) which interrupts in the vector would we need to implement?
   b) what might the error interrupts do (i.e. 0x4, 0xC, 0x10)?
   c) how might we implement reset (0x0) to use it to run student programs?

4: An interrupt handler might have to deal with several peripherals (e.g. all the different slow devices), and this code has
to cope with devices being plugged in or removed. Write a loop that can check a list of peripherals to see which caused the
interrupt.

5: Suppose that I use a computer that obeys 500 million instructions per second, and that dealing with an interrupt
(including running the correct interrupt handler and moving data to or from somewhere sensible) takes the equivalent of 100
instructions. Roughly what fraction of the computer’s time is taken up by interrupts, at the rate of 1 per character or byte,
from:
   a) a keyboard, if I type at about 1 character per second.
   b) a simple printer, printing a page (60 lines * 80 characters per line) a second.
   c) a hard disc, transferring 30M bytes per second.
   d) a screen, displaying 1152*864 pixels, redrawing 50 times a second, with 24 bits of colour information per pixel.
16 System Software

So far, we have looked at how to write ARM code, and in particular, how pieces of Java can be encoded. However, this is only part of the story of how a Java program is run. For the next few lectures, we are going to look at the system software that is needed to complete the story.

There are two main driving forces behind the creation of this software:

**Code reuse** if anything is done repeatedly (e.g. getting input from a keyboard) then get it right once and for all.

**Safety** protect users from themselves and from others e.g. prevent:
- programs accidentally affecting each other when running together
- “read-only” files being overwritten
- malicious attacks

The software on our computers has several major components:

**Kernel** has direct control over all the hardware. It provides simplified access and extra facilities, such as safety features. It is sometimes known as a Virtual Machine (VM).

**Libraries** general-purpose facilities, like those provided by the kernel, but not directly safety-related (e.g. input-output of numbers, using the character input-output provided by the kernel). These may be specific to particular languages, like Java, or applications, like graphics, or intended for more general use.

**User Interface (UI)** allows users to access the facilities provided by the kernel and libraries, including the ability to run applications and development tools. It can be:
- **Textual** a Command Line Interface/Interpreter (CLI) or Shell
- **Graphical (GUI)** a desktop, using Windows, Icons, Menus and Pointers (WIMP).

**Applications** e.g. office (word/text-processing, database, spreadsheet etc.), internet (browser, ftp etc.), graphics (image manipulation, drawing etc.)

**Development tools** used to create new programs: compiler, editor, debugger, programming environment (PE), development environment (DE) etc.

One of the aims of modern computer systems is to simplify use, and so hide some or all of these distinctions from the user:
- Development tools and applications are often sold with a system, and their user interfaces are usually based on those for the kernel.
- Some applications such as database systems have their own development environments, that allow the user to create complex queries and reports.
- Some kernels only have one user interface, which is assumed to be part of it – e.g. MS Windows. However, Linux makes the distinction clear, as it usually provides several shells (e.g. sh, ash, bsh, bash, csh, ksh, tcsh, sash, tclsh, mc) and desktops/window-managers (e.g. gnome, kde, twm, fvwm, kwm, afterstep, enlightenment, windowmaker, blackbox).

16.1 The Kernel

The kernel protects the user from the hardware, and the hardware from the user:

It provides a layer of abstraction that simplifies the use of the hardware – instead of users worrying about status registers and interrupts, they can just ask the kernel e.g. to give them the next character from the keyboard.

It makes different devices look similar so that a user doesn’t have to significantly change a program to use e.g. a different input or output device – getting the next character from the keyboard appears to be essentially the same as getting the next character from a file.

It provides protection – if two programs want to use some hardware resource, it makes them “play nice” e.g. if they both want to use a printer, the kernel makes sure that the two sets of output are printed one after the other, rather than all mixed together.
To do all this, the kernel:
- has complete control of all the computer hardware e.g. it includes all the interrupt handlers,
- cannot be bypassed, so prevents direct access to hardware by other programs

Therefore, hardware usually supports different modes of operation:
- a safe mode for users and applications, that prevents some kinds of potentially dangerous actions
  e.g. a program can only access its own part of memory, and so can’t mess up another program, and especially can’t mess up
  the kernel or the peripheral registers
- one (or more) privileged mode(s) for the kernel where any action can be performed.
  e.g. it can access any part of memory, including peripheral registers

For example, the ARM provides the following modes:
- User: unprivileged
- System: privileged code other than interrupt-handlers

As well, the interrupt handlers have their own privileged modes:
- SVC (supervisor): SWI
- IRQ: normal interrupts
- FIQ: fast interrupts
- Abort: memory faults
- Undef: undefined instructions

When a program running in User mode wants to ask the kernel for (supervised) access to some facility (e.g. get a character
from input) it does so by making a system call. To the programmer, this looks like a call to normal method. However, the
compiler translates this (on the ARM) using an SWI instead of a BL. The SWI interrupt handler then checks the request
number, rejects any undefined numbers, and calls the method to handle the specific request.

### 16.1 Resource Managers

The kernel essentially manages the resources provided by the hardware: the CPU, the memory, and the peripherals – disks,
communications, and user interface. The main resources and managers are:

**Process Manager** which controls the CPU, and is able to create the illusion that several things can be going on at once.

**Memory Manager** which controls the main memory, so that each of the several things going on at once has as much
memory as it needs but can only use the memory it is supposed to. In particular, this helps protect user programs
from each other, and the kernel from user programs.

**Peripheral Managers** usually there is a separate manager (often known as a device driver) for each kind of peripheral,
or even for each different peripheral, allowing information to be moved to and fro as appropriate.

There may be extra layers of software to support more complex activities, such as using the communications to talk to
the internet, or using the display for graphics, or running a queue to allow many users to access a printer, or organising
the filestore so that you can use only those files that you are supposed to.

In this case, we usually distinguish between the low level device driver, that moves data in and out of the peripheral,
and the higher level manager, that decide what data is to be moved where.

Various managers cooperate to provide services to the user, so the disk manager and the memory manager together allow the
memory to appear to be bigger than it really is, and the filestore manager relies on the disk manager and communications
manager to actually store and retrieve the information in your files.

### 16.2 Exercises

1: Which ARM interrupt handlers share the same privilege mode?
17 Assemblers and Compilers

Assemblers and compilers translate programs, written in assembly language or in a high-level language respectively, into a simpler form, for later execution by real hardware or by a software interpreter.

17.1 Assembler

Simplifies the task of writing binary programs

\[ \text{e.g. ARM program} \xrightarrow{\text{assembler}} \text{binary program} \]

The steps required to do this translation are:

1) Build words from characters, discard spaces and comments
   e.g. `loop LDR R1, fred ;a comment` \(\rightarrow\) `loop LDR R1, fred`

2) Check statements are legal
   e.g. LDR is a mnemonic, that takes two operands, etc.

3) Check user-defined names (labels)
   - declared exactly once
   - keep list of names + addresses

4) Translate to binary machine code
   - usually one-to-one
   - but: pseudo-instructions like ADR, LDRL, DEFW etc.

Formally:
1) Lexical (word) analysis
2) Syntactic (sentence structure) analysis
3) Semantic (meaning) analysis
4) Code generation

(1) + (2) usually defined by a grammar (BNF) e.g.

\[
\text{instruction: label \ mnemonic3 register ',', register ',', operand} \\
| \text{label \ mnemonic2 register ',', address} \\
| (\text{etc.}) \\
; \\
\text{mnemonic3} : \text{ADD} | \text{SUB} | (\text{etc.}) ; \\
\text{mnemonic2} : \text{LDR} | \text{STR} ; \\
\text{register} : \text{R0} | \text{R1} | (\text{etc.}) ; \\
\text{operand} : \text{register} | '\#' \text{number} | (\text{etc.}) ; \\
\text{address} : \text{number} | \text{name} | (\text{etc.}) ; \\
\text{label} : \text{name} | (\text{nothing}) ;
\]
17.2 Compiler

Makes it seem as if the computer can understand high-level language programs.

\[
\text{e.g. Java program} \rightarrow \text{compiler} \rightarrow \text{e.g. binary program, error messages?}
\]

e.g.

```java
public class Work6 {
    public static void main (String argv[]) {
        int a = 1;
        while (a * 2 > 0)
            a = a * 2;
        System.out.println("a = " + a);
    }
}
```

The steps required to do this translation are listed below. The first 4 steps are the same as for assembly, but each is much more complicated:

1) Lexical Analysis: Build words from characters, discard spaces and comments.
   There are more kinds of words: names, values, operators, punctuation, etc.
   e.g. `public class Work6 {` `public static void main (String argv[]) {`

2) Syntactic Analysis: Check sentences are legal
   There are many different kinds of declarations and statements, which can contain more complex components, such as expressions, or we can even have e.g. if-else statements that contain other statements.

3) Semantic Analysis: Check user-defined names
   – declared (exactly once)
   – variable, method, class, etc. (are the different kinds used correctly?)
   – type: int, float, String, etc. (do the types match? what does + mean?)
   Keep list of names, addresses, kinds, types etc.

4) Code Generation: Translate to assembly code or binary machine code or Java byte code or . . .
   – one to many

   ```assembly
   MOV R0, #1
   STR R0, a
   while MOV R0, #2
     LDR R1, a
     MUL R1, R1, R0
     CMP R1, #0
     BLE end
   MOV R0, #2
   MUL R1, R1, R0
   STR R1, a
   B while
   end
   ```

5) Code optimisation

   ```assembly
   . . .
   MOV R4, #1
   while ADDS R0, R4, R4
     MOVGT R4, R0
     BGT while
   end
   ```
18 Libraries and Interpreters

18.1 Libraries & Linker

Libraries: increase the set of operations available to programmer, allow access to facilities provided by system etc. – how does this work?

We could copy the original text defining the library, add it to our program, and assemble/compile the whole lot together.
– tedious for us
– slow to assemble/compile
– do we need to rewrite it for each language?
– what if whoever wrote the library doesn’t want people to see their code?

Assemble/Compile these operations once and for all, so users don’t need to keep recompiling them.

Library =
– code

When we compile a program, the compiler does lots of checking, to make sure everything is being used correctly – how do we avoid recompiling the library code, but still get all the checking?

We need to somehow give the compiler a copy of the declaration of everything important in the library, but not the definition
– e.g. the list of parameters and the return type for a method, but not its body.

Library =
– code
+ associated information for checking

Linker: extra step, does whatever is necessary to attach the already-compiled library code to our compiled program.

Library code in one or more chunks
– how big is a chunk: is it one method, or is it a whole set of related things, such as a class?
– how does the linker locate a chunk, so it can copy it?

For each chunk, attach list of names + addresses of useful things in the chunk.

Linker
– searches list(s) of names + addresses in libraries to locate operations needed
– e.g. copies requested operations from the library into the user program

Library =
– chunks of code
– exported names & addresses in each chunk
+ associated information for checking each chunk

What if one library needs to use the facilities provided by another library?
– each library also has a list of its imports
– the linker has to find them as well!

Library =
– chunks of code
– exported names & addresses in each chunk
– imported names for each chunk
+ associated information for checking each chunk

When a library is being compiled, if it can be added to any program, the compiler doesn’t know where exactly it is going to be within each completed program i.e. where in memory, so what addresses should it use?

Position invariant code: compile the code so there aren’t any fixed addresses e.g. use offsets from PC

Relocatable code: compile using fixed addresses, but add information to the library so the linker can change them to whatever they end up as.

Library =
– chunks of position invariant/relocatable code
– exported names & addresses in each chunk
– imported names for each chunk
+ associated information for checking each chunk
Real computers run many programs “at the same time”, and some will be using the same library routines (read/write numbers etc.), so we will fill up memory with multiple copies of the same code – how can we avoid wasting memory?

Use **dynamic, shared** libraries, rather than **static** libraries

- Linker doesn’t actually add dynamic library code to each program
- instead, it arranges that, when the program is started, the **dynamic loader** looks for the library
- if the library is already being used somewhere, we can use that copy again
- if not, load library into memory, and start using it

```
$ ldd /usr/bin/X11/nedit
    libc.so.6 => /lib/i686/libc.so.6 (0x402ce000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Slower to start up, but saves memory.
Can upgrade dynamic libraries without having to recompile the programs that use them.
Can even load dynamic libraries while the program is running e.g. plug-ins.

### 18.2 Interpreters

An interpreter is a program that obeys “compiled” code – the code can be for a real computer or for an imaginary computer (a **virtual machine**).

**Disadvantages:**
- usually much slower than a real computer
- often hard to exactly simulate a real computer

**Advantages:**
- simplicity
- resources
- only execute code once
- some languages are intended to be interpreted
- portability
- security
- prototype hardware
- instrumentation
- software development
- debugging
- etc.