This manual belongs to: .....

e-mail: .....

If found please return to owner

or leave in the microcontroller lab.

Contents

Organisation of the Laboratory 5

Introduction 7

The Laboratory Equipment 9

Programming style & practice 13

Session 1 – Simple Output 19

Session 2 – Simple Output 23

Session 3 – A (very) Primitive Operating System 29

Session 4 – Scheduling 39

Session 5 – Counters and Timers 41

Session 6 – Interrupts 47

Session 7 – System Design and Squeaky Noises 51

Session 8 – An Interrupt Controller 57

Session ? – Stepper Motors 61

Session 10 – UARTS (??) 65

Session 11 – Digital to Analogue Conversion 71

Session 12 – Analogue to Digital Conversion 75

Session 13– A Digital Oscilloscope 79

Glossary of Terms 91

More requirements …

Understanding data(sheets)

Organisation of the Laboratory

Organisation? What organisation?

Some bullet points

- Certain exercises require paperwork to be handed in. You must do this or you will not be credited with any marks!
-
- Many exercises will depend on previous work. You should therefore complete exercises in the order given.
-
- The syllabus assumes that you will spend approximately the same time working on this course outside laboratory time as you do in scheduled lab. hours. To quote the Department Undergraduate Handbook:

  > “Please note that the expectation is that students will be required to undertake approximately forty hours per week of study i.e. an average of one hour’s private study will be required for every scheduled hour of lectures, laboratories etc. and some students may require much more time than this. BEING A STUDENT IS A FULL-TIME OCCUPATION!”

-
- Some exercises include suggestions for further work. These carry no extra marks but may provide some more good practice. If you have time you may attempt some these; if not then they are a distraction so don’t bother.

Introduction

Purpose

The purpose of this course is to:

- gain more experience and confidence in assembly language programming
- learn something of the structure of the low-level embedded software
- probe the hardware/software interface
- interface computers to their environment

Prerequisites

The course will require some knowledge of assembly language programming and of basic logic design. It is assumed that participants will already have taken CS1031 and CS1211. Some of the tools used in these courses will be revisited in CS1242 – you should therefore bring your own manuals and notes from these to refer to.

CS1242 is complementary to CS1222 (particularly as regards the laboratory work). However whereas CS1222 emphasises logic design (and should be seen as a precursor to second year VLSI and processor design courses) CS1242 is primarily a software course, although a great deal of 'hardware literacy' is involved. CS1222 is therefore an advised, but not compulsory, co-requisite. A reminder of the titles of these courses is included below.

Course	Title	Lecturer(s)
CS1031	Fundamentals of Computer Architecture	Edwards
CS1211	Processor Design	Furber, Garside
CS1222	Computer Technology	Furber, Garside
CS1231	Introduction to Electronic Circuits	Cunningham

Teaching Style

This manual describes the entire course, together with some extensions and background material. There are no formal lectures, although there will be interludes of taught material within the labs. However this is a practical subject and the only way to learn it is by having a go (and making mistakes).

Because of this style all the assessment is also done on the practical work. This means that you must not only complete the course but **you must hand in any required evidence** so that we know you have done it.

The basic exercises in this course are intended to introduce a basic set of concepts. In general it is necessary to complete the basic exercise before moving on, as some of the ideas – and frequently some of the practical output – form a prerequisite for later exercixes. In addition many exercises include suggestions for extensions; such extensions are not necessary to the course but are there to provide a starting point if you want to learn more.

External References

There are a number of external references scattered through this manual. These may be to teaching resources, research projects, product data etc. and are usually to information obtainable via the World Wide Web (WWW). In general URLs are *not* given – WWW pages sometimes disappear and new and better ones are created – instead you should track down your own sources with the search engines of your choice.

Can we mark up (as a hierarchy) some sections which could be skipped/included at the reader's discretion?

The Lab. Equipment

Circuit board

The microcontroller laboratory is intended to reflect the development environment used for embedded controllers in the early 21st century. The basic system comprises a software programmable processor – in this case an ARM – with a number of peripheral I/O systems. The I/O is provided using an FPGA which can be configured by the processor; this means that both the software and the hardware is programmable and can be adapted to a particular environment. The ability to use FPGAs for customisable hardware had considerable influence on the design of systems in the 1990s. As FPGAs have increased in capacity it is now feasible to use them for … BLAH BLAH

coprocessors (e.g. signal processing)

An extreme example of this can be found in the “raw” project at the Massachusetts Institute of Technology (MIT) Laboratory for Computer Science (http://www.lcs.mit.edu).

Software Development Environment

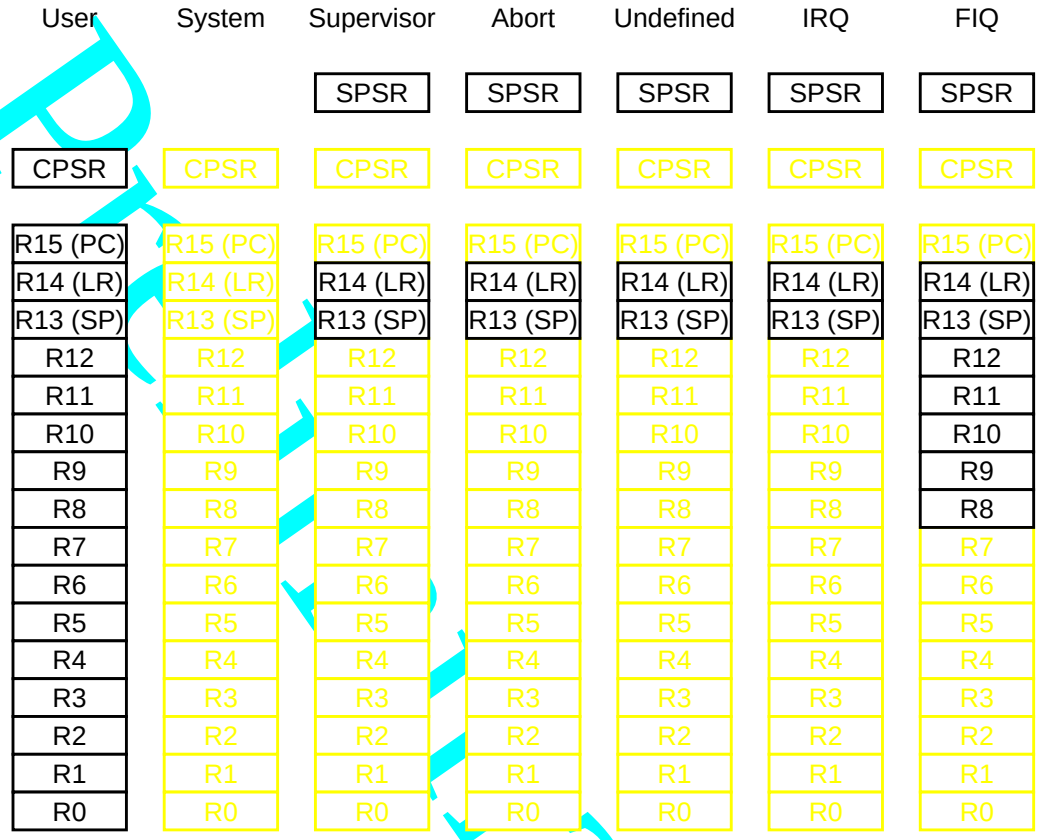
PC etc.

The software development environment used is ?????? – the same system used in CS1031 ????. The only real difference is that programmes will be downloaded to, and executed on, a real ARM processor rather than a software simulation. This gives access to the real peripheral devices …

Programmer’s crib sheet …

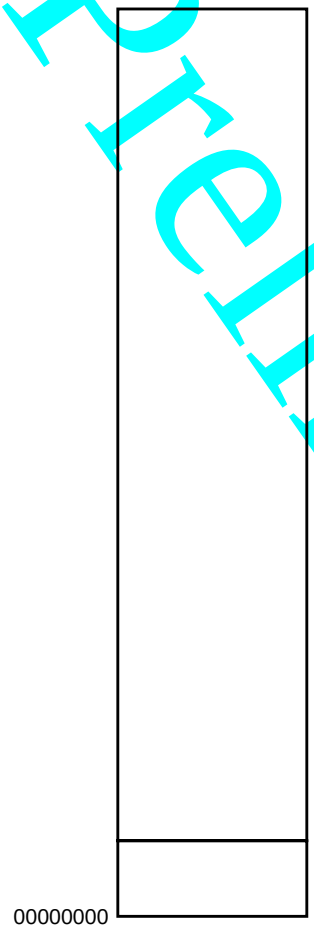
ARM programmers' model

The ARM register map and mode encodings are shown below for quick reference.



Mode	Code	Privileged
User	1 0000	No
System	1 1111	Yes
Supervisor (SVC)	1 0011	Yes
Abort	1 0111	Yes
Undefined	1 1011	Yes
Interrupt (IRQ)	1 0010	Yes
Fast Interrupt (FIQ)	1 0001	Yes

Lab. board memory map



Remember that the ARM is a 32-bit processor with byte addressing. This means that the addresses of adjacent words are different by 4. Words must be aligned with the memory (i.e. the lowest two bits of the address must be zero).

Peripheral ports

…

Predefined peripheral set(s)

Hardware Development Environment

As in CS1211, & CS1222, Powerview is used for hardware development. You should create a Powerview project for this laboratory using the usual setup script (i.e. “mk_pv 124”) and invoke it as appropriate for this laboratory (i.e. “pview 124”). The target FPGAs (Field Programmable Gate Arrays) used in this laboratory are different from those used in the other first year laboratories, so the CS124 setup links to a different library. Most of the components will have familiar names however.

Library crib.?

The hardware designs required in this laboratory are simpler than those in CS1222. However they will need to be integrated with software too, and for rapid, successful debugging use of **simulation is highly recommended**.

Compilation …

Downloading …

Programming style & practice

Structure

Some inexperienced “programmers” think that the choice of language affects the quality of the programming. The bandy terms like ‘structure’ and sneer that assembly language is, somehow, intrinsically ‘unstructured’. One of the main arguments is that there are no “IF … THEN … ELSE”s and “REPEAT … UNTIL”s, only the equivalent of “IF … GOTO”. These people will typically Dijkstra[1] as a reference. They are WRONG!

To be fair to Dijkstra this is misapplication of his thesis; he cites the ability to jump randomly from place to place in a (high-level) programme as “an invitation to make a mess”. *If* this is the case it is an invitation which should be politely declined. Here we get to the point of this section.

Programming is not a function of a language; it is much more than that. The majority of the art lies in analysing a problem and devising an algorithm which will solve that problem, which is implementable, and which is reasonably efficient. This is independent of the language chosen (or imposed).

The danger in assembly language programming is that it is much less restricted than operating in the confined syntax of a language; remember the language is compiled into machine instructions too! It can be much easier to write bad code in assembler than in many languages. It can also be easier to write good code. The point is that it is up to *you* to ensure that your code has an underlying structure.

There is not the time or space here for a course on structured programming. However here are a few tips:

- Use **procedures** (functions, subroutines) to isolate different identifiable operations. This allows a ‘top down’ ‘divide and conquer’ approach which is useful even if the procedure is only called once. In general procedures should communicate through passed **parameters**, which can be travel both ways. (Many high-level languages define functions which can have unlimited input parameters but only return a single quantity; this can encourage bad practice.)
  A procedure should have a well-defined entry and exit in the code. If you are being tempted to jump into the middle of a procedure take a cold bath and then modify your structure. The use of conditional returns can also be frowned upon, although they can prove a boon to efficiency.

- Keep careful control of the **scope** of variables. This is easy within a procedure – local variables are usually nestling in registers – but it requires self discipline not to just read or alter a variable in memory because you can. Resist! Data structures should be maintained by their own library of access procedures (a.k.a. “methods” these days). This means that, when it is necessary to edit the code,

1. Edsger W. Dijkstra, “Go To Statement Considered Harmful”, Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.
http://www.acm.org/classics/oct95/ – go look it up, it’s quite short.

changes can be handled locally.

- Loops
- Beyond the obvious issue of the length of a data item assembly language does not have explicit **types**. It is therefore your job to ensure that your variables obey your own typing rules. It is very unusual to require floating point (float, real, …) numbers in assembly language[1] but integers, bytes, characters, pointers and more complex structures are common. An integer and a pointer may both be *represented* in a 32-bit word, but they are different things; for a start it makes sense to add integers, whereas adding pointers is nonsensical!
  You may choose to adopt a convention to help identify the type of items; for example prefixing pointer names with "p_" makes them fairly obvious.

Style

Although style is a personal thing, there are some practices which are worth observing to make your source code more readable, robust and maintainable. These are not all specific to assembly language either.

Firstly **layout**. You should be familiar with the general style of assembler source code, i.e.:-

```
Label       MNEMONIC  Addresses       ; Comment
```

It is a good idea to adhere to this format – the source code looks neat and labels stand out and therefore are easy to spot. It is also sensible to break up the code with blank lines to highlight groups of related statements (very approximately equivalent to a line of source code in a high-level language).

Within this framework it is possible to enhance readability in other ways. One obvious method is to choose **sensible label names** which reflect their function. As it is often difficult to think of many different names it is usual to name the entry point to a module (procedure etc.) sensibly and then use derivatives of this by suffixing numbers for labels inside the module. This has the added bonus of indicating the extent of the module.

The **comment** field should be used to explain the meaning of the statement. It is usually possible to attach sensible comments to every line. An example of a nonsensical comment would be:

```
            ADD       R2, R2, #1      ; Increment R2
```

However:

```
            ADD       R2, R2, #1      ; Increment loop index
```

conveys significantly more information.

The #1 in the preceding example is one of the few cases where it is sensible to have immediate

1. This is a good illustration of how rarely they are needed in *any* code.

numbers appear in the address field. It is usually better to equate a number to a label and use that instead. This both conveys added information (the value 'FF' could be used for lots of things, but 'byte_mask' is clear) and allows definitions to be changed in a single place (in a separate header file or at the top of the code) rather than trawling through looking for numerous references and, probably, missing some.

Finally, as the whole programme (being structured) is broken into procedures these can also be emphasised in the source code. Some more commenting can be added to document the procedure's function, its input and output parameters, any non-local data used, any registers which are corrupted, etc. This is useful when trying to reuse or modify pieces of code, and it is the best place to keep this information in that it cannot easily become detached from the code itself.

Example

```
; This routine prints a zero-terminated string
;  pointed to by R0

Print_stringSTMFD    SP!, {R0,R1}    ; Push working regs
Print_str1  LDRB     R1, [R0], #1    ; Get byte, auto-inc.
            CMP      R1, cTTR        ; Test for terminator
            SWINE    Print_char      ; ... if not terminator
            BNE      Print_str1      ; and loop
            LDMFD    SP!, {R0,R1}    ; Pop working regs.
            MOV      PC, LR          ; Return
```

Note: this will not work in supervisor mode.

Why not?

How could you fix this?

Relocatability

It is usually an advantage if code can be executed wherever it is located in memory, i.e. it is **relocatable** or **position independent**. Depending on the processor used this is not always feasible; for example a processor may only have jump instructions which transfer control to a fixed address. However most modern processors are fairly unrestricted and, with care, code can be fully relocatable. (Note that this does *not* imply that code can be moved *during* execution.)

For example the ARM branch instructions are all **relative branches**[1], i.e. the target address is

calculated as an offset from the current position. Thus whatever address the code is placed at the branch will still be to the correct target. In fact a branch instruction could be written as:

```
            ADD       PC, PC, #offset
```

although the offset range is larger than that allowed in ‘normal’ immediate operands. ‘Normal’ branches on an ARM are therefore relocatable.

Other forms of branches are usually position independent too; for example a subroutine return will go back to the calling routine wherever that was. The exceptions are when an address is calculated or loaded from memory where extra care is needed if position independence is to be maintained.

Example: the branch instruction only provides a 24-bit (word aligned) offset, allowing branches backwards or forwards in the address space of about 32Mbytes (8Minstructions)). This is enough for almost all purposes, but if a more distant branch is required one way of providing this would be:

```
            LDR       PC, long_branch ; Load target absolute
long_branch DCD       target

            …

target      …
```

This creates a word ‘variable’ (really a constant) which contains the address of the distant routine. The value can be read in a position independent way – “`long_branch`” will be translated by the assembler into “`[R15, #-4]`” which refers to the *next* location because R15 is the current position + 8. However the value “`target`” is an absolute number which is set by the assembler and thus will not alter if the code is moved.

The following sequence gets around this, at a price:

```
            LDR       R0, long_branch ; Load target offset
branch_instrADD       PC, PC, R0      ;
long_branch DCD       (target - branch_instr - 8)

            …

target      …
```

Here the relative offset between the ‘branch instruction’ (`ADD`) and the target has been stored and added to the PC explicitly; the “`- 8`” is needed to allow for the ‘PC+8’ effect. This is relocatable although slightly slower and requiring an extra register.

As well as code data should be position independent. Most data are stored in one of four (??) places:

1. Although not universally observed the usual convention is that “branch” refers to a flow control change made relative to the current position while “jump” refers to one to an absolute address.

- On the stack (dynamic)
- On the heap (dynamic)
- Within the code (static, often read only such as constants, strings, …)
- Another static space (e.g. global variables)

Dynamically allocated variables are not a worry here; by definition their addresses are defined at run time, usually in some space allocated by the operating system.

Data may be embedded in the code. These can be accessed via **PC relative** addressing as in the previous example, so they need not present a problem. (Note however that there is a 4Kbyte limit on the offset from the PC to the variable of interest which can encourage the slightly dubious practice of interleaving data items between procedures.) It may be feasible to use this technique for variables as well as constants but this practice is deprecated as it leads to problems in placing the programme in ROM. Finely interleaving code and data can also lead to inefficient cache usage if these items are cached separately, as they are on many high-performance processors.

The real problem is finding a place in RAM for the static variables which can be accessed globally rather than – for example – relative to the stack pointer. There is no easy answer to this. The commonest solution is probably to dedicate a register to point to a fixed area of memory (allocated by the OS when the programme starts) and use fixed offsets from that. Sadly this ‘loses’ a register.

Session 1
Simple Output

Objectives

- Hardware/software familiarisation exercise

How to download software …

How to execute/step/debug software …

How to download (predefined) hardware …

Input and output ports

To be at all useful a computer system must have some input and output (I/O) capability. The simplest I/O is provided by a **parallel port** which essentially maps a memory location into some real hardware. In the case of an output port this means that the state of the bits stored in this 'memory' are used to control some external hardware, such as an LED (Light Emitting Diode. In the case of an input port the 'memory' is some outside world device - for example a single bit's state could come from a push button switch. The devices which inhabit this space are known as **peripherals**.

In some processors there is a special I/O address space (outside the normal memory map) with special instructions which can be used for I/O peripherals. However on most RISC processors, including the ARM, there is only a single address space and so some memory locations are sacrificed for I/O; this is not a significant problem when there is 4Gbytes of addressable space.

It is frequently the case that peripheral devices do not use the full 32-bit bus; most common devices have only an 8-bit interface. This is reflected in this laboratory where all the I/O devices will be accessed eight bits (or fewer) at a time. When communicating with I/O ports you should therefore remember to use *byte* load and store instructions (**LDRB**/**STRB**).

Bit manipulation

Although some I/O devices are byte wide (or, occasionally, larger) many inputs and outputs are smaller – often single bits. For example the position of a switch or button can be represented with a single bit. It is usual to cluster several (functionally connected) I/O bits together into partial or full bytes to simplify the hardware requirements; an example is used in this exercise. Because the smallest quantity which can be addressed is (usually) a byte the issue of **bit addressing** must be handled in software. Bits are normally addressed such that bit 0 is the least

significant bit.

The ARM can alter a single byte in memory with a STRB instruction. However if it needs to change a single bit it cannot modify it without, potentially, changing the other seven bits in the byte. To avoid this the other bits must be written to their original value.

To do this the programmer must first find the original value of the byte. Usually[1] this can be done by first loading the byte and altering the bit(s) concerned in a register. Individual bits can be set by ORing the value with the appropriate bit mask; bits may be cleared by ANDing with the complement of this mask or, on the ARM, using the BIC (BIt Clear) instruction. It is a simple and worthwhile exercise to learn the basic bit mask as shown in table 1.

Bit Number	OR (BIC) mask	AND mask
7	80	7F
6	40	BF
5	20	DF
4	10	EF
3	08	F7
2	04	FB
1	02	FD
0	01	FE

Table 1: Hexadecimal bit masks

(Advanced) A trick worth knowing is that any bit manipulation can be performed using two successive bit mask ANDed and XORed with the byte in question. (Develop? Table?)

In this first exercise a single port is used for output; only six output bits are significant:-:

Bit	LED (really not confirmed)
7	Unused
6	Red (N/S)
5	Amber (N/S)
4	Green (N/S)

Table 2: 'Traffic light' bit assignments (port address = ?????)

1. Some I/O devices use the same address for reading one value and writing an unrelated one

Bit	LED (really not confirmed)
3	Unused
2	Red (E/W)
1	Amber (E/W)
0	Green (E/W)

Table 2: 'Traffic light' bit assignments (port address = ?????)

LED on/off

(single step?)

7-segment decoder ?

Traffic lights (?) 3 (6?) LEDs on ports (couple of buttons too?)

Talk to a monostable to flash light?

Modern computers execute millions of instructions per second. It only takes a few instructions to change the state of a few LEDs. In order to make the exercise viewable by a human the steps must have 'reasonable' intervals between them. This could be done by using a **delay loop** – a contruct that repeats wasting a little time many times over – but this is usually **bad** practice for several reasons[1].

To provide a more stable timing reference a hardware reference is included as another peripheral port (addresses???). This accepts a value into one port (addr??) and then asserts a signal (addr???) for a time proportional to the input number. (NOT 1s or easy factor thereof so it can't be used later).

Needs a peripheral designing!

Exercises the output, wait-'til-ready idea

Promote timer exercise to earlier?

1. These should become apparent later.

Practical

Write a programme that cycles a set of traffic lights attached to an output port at address ??????. Each light is controlled by a single active-??? bit where ?? = light off, ?? = light on.

In order to provide a timing reference …

Download a predefined hardware configuration and write something to drive it.

Further possibility
– add input buttons

Note: should encourage a parameterised “wait” subroutine.

Session 2
Simple Output

Requires – I/O port

Objectives

- Familiarisation exercise
- Parallel output/handshake(ish) sequencing (timing- software delay?)

The Liquid Crystal Display (LCD)

Character-based liquid crystal displays (LCDs) contain some ‘intelligence’ and use a (reasonably) standard interface. The interface is a parallel port which comprises data and control signals. The data bus can be 4-bits or 8-bits wide; the 4-bit mode is used when there is a very limited number of I/O signals available and need not concern us here.

The interface signals are:

Name	I/O	Function
DB[7:0]	I/O	Data bus
RS	O	Register select (address line) 0 = control, 1 = data
R/$\overline{W}$	O	Read not Write 0 = write, 1 = read
E	O	Enable High to validate other signals

Table 3: HD44780 LCD Controller Interface Signals

In order to communicate with the display the control lines must be set to the appropriate values and then **strobed** using the enable line. If the enable signal is inactive the other signal states do not matter. During a transfer the signals should be stable and the usual constraints of set-up and hold times must be met.

When writing to an output port the bits will change approximately at the same time. In order to provide the correct timing the transfer will therefore require at least three separate output commands:

1 Set up data and command

2 Set enable line active

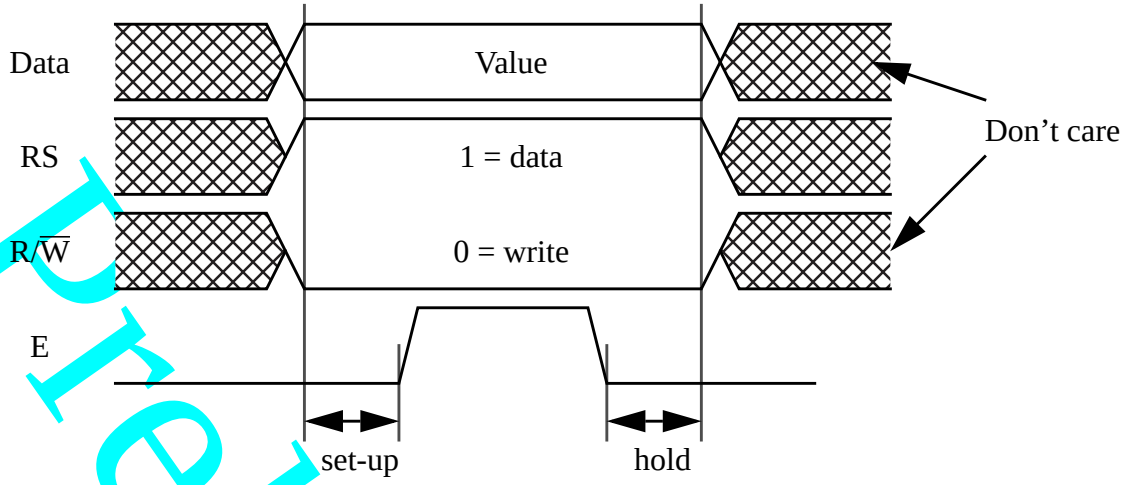


Figure 0.1: LCD data write cycle

3 Set enable line inactive

Note that it is not necessary to actively 'remove' the data; it can remain until overwritten by the next command. However it may not be possible to complete step 1 with a single operation if, for example, only one eight bit port can be accessed at a given time (as there are ten bits in the command).

It is also necessary to ensure that there is enough time *between* these steps for the LCD controller to detect and respond to the command; for example, if step 3 follows step 2 too closely the enable pulse may not be detected at all. It may be that the time between instructions is great enough to ensure this, if not the driving software must provide the appropriate delay.

WHAT IS THE CASE HERE??? Looks okay on A2e board.

The LCD controller is a HD44780, a standard part. It is smart enough to obey a small set of instructions. The simplest of these are:

- Print a character (and move cursor to next space)
- Clear the screen

A few other commands enable the cursor to be moved, the display to be scrolled and user defined characters to be downloaded. Data on the display controller may be found on the WWW if required.

Each command takes time to process. The LCD controller will generally be somewhat slower than the processor controlling it. It potentially quite easy to try to send the next character before the previous one has been printed. As this will cause confusion it is important to wait until one command is finished before sending the next. There are two ways of achieving this:

- Wait for at least the minimum command time
- Wait until the display controller is not busy

The first option is the less efficient because:

- absolute times are relatively hard to measure
- the CPU could be doing something else for some (or all) of this time

The second option is better because:

- the CPU only waits as long as necessary
- it is self-adjusting to varying times for different commands or different LCD modules

However the second option does require some feedback from the LCD. To obtain this the LCD control register must be read and bit 7 will then indicate the controller's status (0=idle, 1=busy).

To read the controller the appropriate command must again be sent, in this case {RS=0, R/$\overline{W}$=1}. Before enabling the display the data bus – which will normally be an output – must be set to be an input (HOW??). This then '**floats**' the bus (i.e. it becomes tristate) so that, when it is enabled, the LCD controller may drive it. If this is not done two different values may be driven from each end of the bus which will lead to an undefined value, excessive power dissipation and, possibly, damage to the components.

The sequence for writing a character now becomes:

1 Set command for read status {RS=0, R/$\overline{W}$=1}
2 Set data bus direction to input
3 Enable bus
4 Read LCD status byte
5 Disable bus
6 If bit 7 of status byte was high repeat from step #3
7 Set command for output data {RS=1, R/$\overline{W}$=0}
8 Set data byte for output
9 Set data bus direction to output
10 Enable bus
11 Disable bus

The action of this on the interface signals is illustrated in figure 0.2. It all sounds like a lot of

fiddling about, but each step is only one or two instructions. Furthermore once the routine is written it can be used to output all the characters you'll ever need.

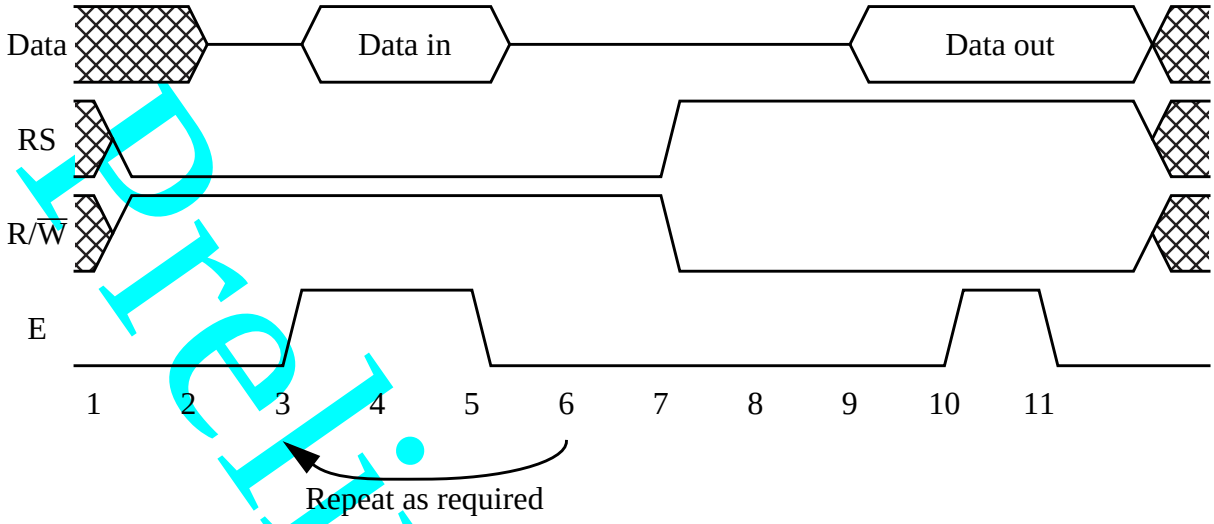


Figure 0.2: LCD character output sequence

Writing a control byte is similar – the only difference is the register addressed in step #7 (for the control register RS=0). The command for "clear screen" is 01. There are plenty of WWW sites which will help you find out about other commands if you want to do more; searching for "HD44780" is a good place to start.

The LCD Interface

The LCD uses eleven interface signals: eight of these form the data port[1], the other three being control as shown in table 4. These are connected to two 8-bit ports ID??? which in the default configuration are mapped as follows:

Port addresses ????

Port A/B control/data

The port control bytes specify the direction of the bits: 0 = Output, 1 = Input (these should be easy to remember!) Port A communicates the data bytes and needs to change direction. Port B handles the control and should always be an output. Only three bits are used:

Port	Bit(s)	Direction	Signal(s)	Active state
A	7-0	in/out	data[7:0]	-
B	0	out	RS	0 = control register 1 = data register

Table 4: LCD interface ports

1. It is possible to programme the interface to use only four bit data. This saves signals (and the remaining 7 fit neatly into a single 8-bit port) but means that two strobes need to be made for every transfer.

Port	Bit(s)	Direction	Signal(s)	Active state
B	1	out	R/$\overline{W}$	0 = write 1 = read
B	2	out	E	0 = interface inactive 1 = interface active

Table 4: LCD interface ports

Be careful not to confuse the control and data registers for the interface with the control and data registers on the LCD panel, accessible *through* the interface; figure 0.3 shows the hardware configuration.

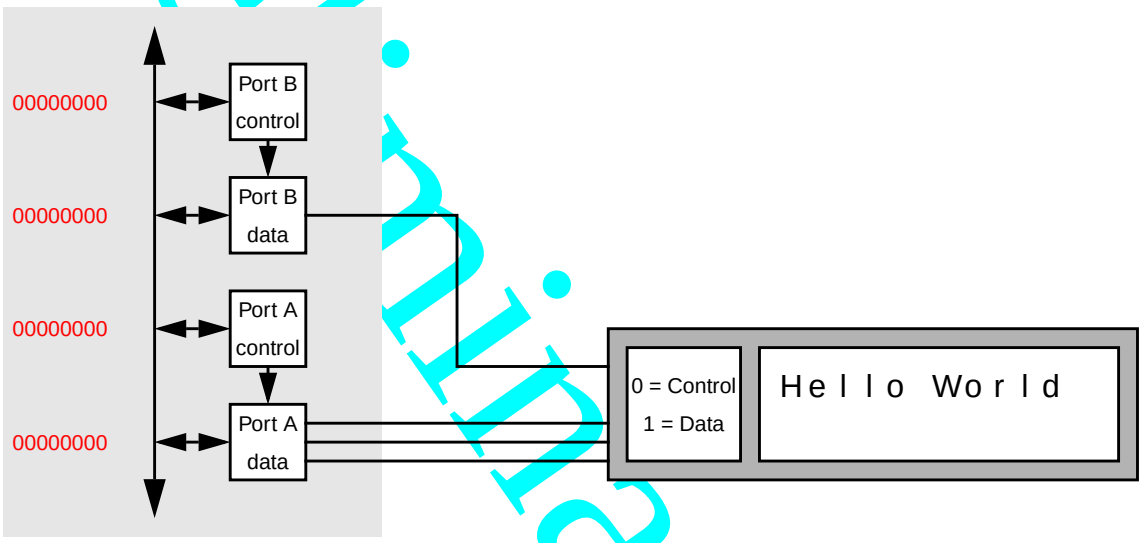


Figure 0.3: LCD controller interface

When writing characters the LCD controller accepts standard ASCII printing characters (see page 101) with a few variants such as “¥” instead of “\”. A Japanese characters set and a few Greek/mathematical symbols are also included using codes from A0-FF; you may see some of these if something goes wrong!

Practical

Write “Hello world!” (or some similarly trite message) onto the LCD display.

Suggestions

The *procedure* for outputting characters is common to all characters. Outputting control and data is the same process except for the state of one bit. Think structure.

Control characters ??? See terminal emulator also.

…

Further possibilities

- Try reprogramming the interface to use 4-bit data and repeat the exercise.

Provide a different interface port design (same wiring, different bit mapping) for this??

Session 3
A (very) Primitive Operating System

Objectives

- Introduce system calls
- Supervisor/user mode operation?
- Structure & reuse of calls

System Initialisation

When a system is switched on a large amount of its state is **undefined**. ROM will hold its contents (of course) but modern RAM relies on a power supply to retain its contents so it will be in an unknown state (unless battery-backed). More relevantly the processor's registers (including the Programme Counter (PC) will be undefined which means the processor's behaviour cannot be predicted.

This would clearly be unacceptable! Therefore at power-on (and possibly at other times) a subset of the system state is **reset** to predefined values. It is not normal to attempt to define all the state (for example the RAM contents) as this would be too expensive, but registers such as the PC *are* defined, so that the processor will execute a known piece of code. If more initialisation is required (it usually is) then the software can perform this function.

ARM initial state

Following reset the ARM's register state is undefined except for:

- PC (a.k.a. R15) which is set to 00000000
- The control bits of the Current Program Status Register (CPSR) which is set to:
  - Interrupts {FIQ, IRQ} disabled
  - Normal ARM instruction set (not "Thumb")
  - Supervisor operating mode

If these terms are unfamiliar, don't worry just yet.

Mode

The ARM has a number of *modes* of operation. These define, in part, the register map and the operating privilege level. ARM has a simple privilege model in that all modes are privileged except the User mode. Privilege grants the ability to do certain things (such as alter the operating mode) which cannot be done from user mode. In a system with memory management only privileged tasks have access to certain areas of memory, such as the operating system work-

space and the input/output (I/O) devices. User programmes are run in user mode which means that they cannot directly interfere with the hardware. The other restrictions ensure that they cannot change the interrupt settings or change mode to escape from these restrictions.

In the lab. system there is no memory management. However let us *pretend* that there is. This means that when user code wants, for example, to print something out it cannot do it directly. In order to do this it must enter a privileged mode (typically *Supervisor* mode). But how?

The answer is a **system call**. These are also known, according to taste as traps, restarts, software interrupts, etc.; ARM uses the term software interrupt (SWI for short) so we will adopt this name within this course.

The behaviour of a SWI depends on the processor being used, but is always something akin to a procedure call; the added value is that the procedure is entered with full supervisor privilege. To prevent abuse restrictions on the procedure address mean that the instruction is not general and must jump to a (or one of a small set of) predefined adddress(es).

In an ARM a SWI instruction has the following behaviour:

- The current status (CPSR) is saved (to the supervisor SPSR – see later)
- The mode is switched to supervisor mode
- Normal (IRQ) interrupts are disabled
- ARM mode is entered (if not already in use)
- The address of the following instruction (i.e. the return address) is saved into the link register (R14)
- The PC is altered to jump to address 00000008

Some of the terminology here may still be unfamiliar!

The upshot is that all the necessary status has been saved so that the calling programme can be returned to and the processor is running a well-defined piece of (operating system) code in supervisor mode.

Exception Vectors

Exception	Mode	Vector	Link made
Reset	Supervisor	00000000	None
Undefined instruction	Undefined	00000004	$R14_{und}$ = undef. instr. + 4
SWI	Supervisor	00000008	$R14_{svc}$ = SWI instr + 4
Prefetch abort	Abort	0000000C	$R14_{abt}$ = aborted instr + 4
Data abort	Abort	00000010	$R14_{abt}$ = aborted instr + 8
–	–	00000014	–
IRQ	IRQ	00000018	$R14_{irq}$ = interrupted instr + 4
FIQ	FIQ	0000001C	$R14_{fiq}$ = interrupted instr + 4

Table 5: ARM exception 'vector' table

These 'vectors' are jumped to when the relevant exception occurs. Because each has only a single word it is usual to place a branch instruction here to a space where the real code can be placed. (Note that this is not *necessary* for FIQ, the exception requiring the fastest response!)

System Initialisation

When the processor is reset its mode and PC are initialised but *everything else* is undefined. It is therefore usual to run some initialisation code before control is passed to a user programme. This can do numerous things but here we are only concerned with a small set of operations

- Initialising exception vectors
  - …
- initialising stack pointers
  - At some time it is likely that the software will require a stack. An area of memory must be allocated for this purpose and the stack pointer must be set to one end of this area.
    In practice it is normal to have a separate stack for user and supervisor code. Furthermore ARM has provision for a separate stack (i.e. a separate stack pointer, R13) in (almost) all of its operating modes. Any stack pointer that may be required in future should be set up now.
- initialising any peripherals required
- entering user mode
  - before a "user" programme is executed user mode should be entered. This should be the last thing on the agenda because user mode code cannot switch back into a privileged mode.

Exception vectors

Clearly the reset vector must be initialised to point to the start of the code. However it is usual to initialise all the exception vectors ‘just in case’.

What behaviour should be observed if an unexpected exception occurs? Ideally there should be an attempt at error recovery, or at least reporting to the operating system. At this point however this would be over sophisticated. Some possible behaviours are:

- Restart the code (i.e. same as Reset)
- Halt
- Return, ignoring the exception

These can be mixed for the different vectors of course. If you want to try and return from an exception (such as a spurious interrupt) discuss your solution first; there are some subtleties which are not immediately apparent.

Stack pointers

Because it has so many different addressing modes it is possible for an ARM to build stacks in a number of different ways. It is important to know which model is in use because – unlike many processors – there is no explicit “PUSH” or “POP” operation; instead these are done using load and store operations.

It is recommended (though not compulsory) that the stack model known as “full descending” is used. In this model the Stack Pointer (SP) contains the address of the last item pushed, and subsequent pushes store data in lower-numbered addresses.

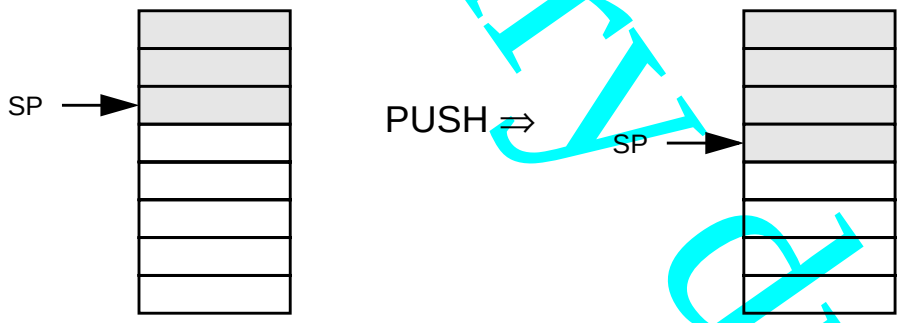


Figure 0.4: Stack push

In this mode the operation “PUSH R0” becomes:

```
STR     R0, [SP, #-4]!
```

i.e. SP is decremented before the data transfer. Note the “!” to preserve the modified SP value.

Conversely the “POP R0” operation would be:

```
LDR       R0, [SP], #4
```

i.e. the load occurs from the current SP address which is post-incremented. Note the SP value is *implicitly* written back here too (despite the absence of the "!").

In typical ARM code it is quite common to wish to push/pop several registers at the same time. This can be done using the LDM and STM (LoaD Multiple & STore Multiple) instructions.

```
STMFD     SP!, <registers>
…
LDMFD     SP!, <registers>
```

(The added advantage of this is that the syntax is easier to remember!)

The implication with a "full descending" stack is that the stack pointer should be initialised to the address *immediately above* the area reserved for the stack; the first stack push will then pre-decrement the address into the allocated region.

In a multi-tasking system it is usual to provide one stack for every process. during **context switching** the stack pointer will be moved from one process' stack to another. Whilst we do not (yet) want to do this, the 'obvious' context switching – between user programmes – is not the only context change which the system will undergo. For example interrupts (coming in a later exercise) will normally have their own context; ARM even provides separate stack pointers for the two classes of interrupt.

More germane here is the fact that the ARM has separate contexts for user programmes and the operating system. Thus there are two stacks which *need* initialising here.

Leave room … memory map …

As each mode has its own, banked, stack pointer – all of which appear as R13 – it is necessary to change mode to set up the SP appropriate to each context. This is done by altering the mode bits in the CPSR. Note that altering these bits is a **privileged** operation and cannot be done from user mode. As there is therefore 'no return' from user mode system mode is provided which gives access to the user register map whilst retaining operating system privileges.

Whilst any privileged process can write to the mode bits it is not usual to simply overwrite the entire CPSR. As a general rule only the bits which are of interest should be modified; other bits (such as the interrupt disable bits) should be *preserved*. As the programme will be unaware of their values (in general, if not in this case) they must first be read. The following code fragment illustrates a switch to IRQ mode:

```
MRS       R0, CPSR        ; Read current status
BIC       R0, #&1F        ; Clear mode field
ORR       R0, #&12        ; Append IRQ mode
MSR       CPSR_c, R0      ; Update CPSR
```

'MRS' & 'MSR' are special MOV instructions which can access the status registers; in the case of the MSR here the action is limited to the control byte ("_c"), although this is really

superfluous here. The action of masking (‘BIt Clear’) and ORing leaves the other fields in the word unaltered.

Peripheral initialisation

In this example there is only one peripheral which needs configuring before the user code can be started, namely the LCD. This should be cleared and the interface signals left in a defined state.

In general there may be a number of different devices which should be configured at this time. For example if interrupts are to be used (which will generally be the case) all the devices which could generate an interrupt must be initialised before interrupts on the processor can be enabled. More on this later (???).

Delay before start-up?

Starting the user programme

Jumping in vs. returning to a known context.

SWI despatcher

The ARM has only a single SWI call which must be used to perform all its system functions. It is therefore necessary for the called routine to determine which service is required and despatch to the appropriate handler.

Two methods have been used to differentiate ARM SWI calls:

Numbered SWI

The SWI instruction has a 24 bit field which is ignored by the processor. This can be used to indicate the type of SWI. To determine the contents of this field it is necessary to read the op. code from the service routine.

```
STR       R0, [SP, #-4]!  ; Save scratch register
LDR       R0, [LR, #-4]   ; Read SWI instruction
BIC       R0, R0, #&FF000000  ; Mask off opcode
```

R0 can the be used to select the required function.

Parameter passing

A predetermined register (such as R0) can be preloaded with the number of the service required. This sacrifices a user register but is easier to process. This method is now preferred for compatibility with the Thumb instruction set (not discussed here).

In addition to a SWI number most OS calls will require one or more parameters. For example, a call to print a character will require the character to be printed. This would normally be passed in another register. The number and type of parameters passed is a function of the particular call made.

SWI service routines

Entry

The following assumes that the SWI service number is already in R0. Note that no register contents are changed here. This code is only suitable for a small number of SWI calls due to the limits on the range of immediate numbers.

```
SWI_entry   CMP     R0, #Max_SWI    ; Check upper limit
            BHI     SWI_unknown     ;
            CMP     R0, #0          ;
            BEQ     SWI_0           ;
            CMP     R0, #1          ;
            BEQ     SWI_1           ;
            …
```

This method is serviceable for a small number of different SWI calls. However as the number of possibilities grows it is more sensible to despatch through a **jump table**.

```
SWI_entry   CMP     R0, #Max_SWI    ; Check upper limit
            BHI     SWI_unknown     ;
XYZ         ADD     R0, PC, R0 LSL #2 ; Calc. table address
            LDR     PC, [R0, #0]    ; #0? - see below

Jump_table  DCD     SWI_0           ;
            DCD     SWI_1           ;
            …
```

This code fragment exploits some features of the ARM instruction set. The ADD instruction is one of those rare cases where the in-line shifter can be used; in this case it multiplies the SWI number by four to convert it to a word address. This instruction adds this, the **offset** into the table, to the R15 value. Because R15 is the address of the current instruction *plus eight*, the offset is added to the address of the start of the jump table. This allows the PC to be loaded directly in the following instruction with an additional offset of zero. If the jump table were located elsewhere this offset could be used to correct the value. It would be good practice to calculate this value at assembly time rather than inserting the “0” explicitly. This would be done as “`XYZ+8-Jump_table`” and would keep the value correct if editing (deliberately or inadvertently) changed the code.

Note: this despatcher is *not* relocatable because …

Devise a way to correct this.

Processing

"Thou shalt not corrupt the user's state."

Like any procedure a SWI service routine will have a function, some input parameters and some output results. These should be clearly defined (this is what the comment field is for, okay?). Whilst it is possible for a service routine to corrupt other state it is generally a bad idea – it stores up trouble for later when the user has forgotten this. The best thing to do is to preserve *all* the register values except those that explicitly return a value.

This is what the stack is for.

Note in particular that a **SWI which calls a procedure** must first preserve its link register and a **SWI which calls another SWI** must first preserve its link register *and its SPSR*. These values must also be restored, of course.

Exit

The SWI instruction has changed the CPSR, generously saving a copy in SPSR_svc. When exiting from the SWI this must be restored. This can be done with the instruction:

```
MOVS      PC, LR
```

The inclusion of the "S" ('set flags bit') in an operation with the destination PC is a special case which is used to copy the SPSR back to the CPSR.

All the different kinds of SWI will have their own service routines. Using the despatcher above it is possible simply to return at the end of a service routine; however it is worth considering exiting through a short section of return code which is common to all the service routines. This is less efficient (i.e. more instructions needed in jumping into this code) but may provide lower maintenance in the future if, for example, the SWI entry routine is changed.

Practical

Modify your "Hello World" programme to run as an 'application' within a primitive operating system (OS). The 'OS' should initialise itself and anything the application might need (specifically the user stack). The user code should communicate with the peripherals using operating system calls (i.e. SWIs).

It will soon become apparent that more than one different OS call is required. As

there is only one SWI ‘vector’, different calls must be distinguished in some way.
…

…

Calls which may be useful now might include:

- Print character
- Print string
- Terminate programme

Clearly a ‘print string’ call makes this user programme somewhat trivial (you might like to print more than one message to bulk out the code) however it may be useful in future. If you do provide it note that it is sensible (structured!) for it to call the ‘print character’ routine to save code duplication.
Later we can, by modifying only the print character routine, redirect the message to a completely different place.

Session 4
Scheduling

Objectives

- ??????????????????

(Should be moved a bit later ???)

When changing context the entire user processor state must be switched. This means that all the user visible registers in the old context must be preserved and a new set must be loaded. The ARM has fifteen general purpose registers (R0-R14), the PC (R15) and the CPSR to preserve.

When switching context the last two of these will already have been corrupted before the operating system gains control; they will, however, be preserved somewhere. Imagine for a moment that a SWI call has been used to switch context; the user PC will be preserved in R14_svc and the user CPSR will have been copied into SPSR_svc. The process context is then pushed onto the user stack leaving R0 as a pointer to the process.

```
STMFD     SP!, {R0,R13}^  ; Save scratch & user SP
LDR       R0, [SP, #4]    ; Recover user SP
STR       LR, [R0, #-4]!  ; Save return address
STMFD     R0!, {R1-R14}^  ; Save most user regs
LDR       R2, [SP]        ; Recover user R0
MRS       R1, SPSR        ; Recover user CPSR
STMFD     R0!, {R1,R2}    ; …and save them
ADD       SP, SP, #8      ; Lose working space
```

A similar sequence can be used if the scheduler has been entered via an interrupt (e.g. a **timeslice**) although in this case it should be preceded by a decrement to LR so that the saved return address is the same in each case.

Draw out the memory map of the suspended process' stack.

In order to return to this process the following sequence suffices:

```
LDR       R1, [R0], #4    ; Recover PSR
MSR       SPSR, R1        ;
LDMFD     R0, {R0-R15}^   ; Restore and return
```

If R0 has been pointed to a different process the processor will return to a different task.

In order to **start a new process** its stack must be created and registers must be defined this stack. Usually it is assumed that a process will initialise its own registers however, so only the SP, PC and PSR need be defined.

Here R0 contains the initial user SP value, R1 the start address of the programme:

```
New_proc    STR       R1, [R0, #-4]   ; Save PC
            STR       R0, [R0, #-12]  ; Save SP
            SUB       R0, R0, #4*(16 + 1)
            MOV       R1, #&10        ; ARM code, user mode
                                      ; interrupts enabled
            STR       R1, [R0]        ;
```

This leaves the necessary values in the correct memory locations.

Why go to all this trouble when you could just change mode and jump to the process? The answer is that just because a process is created which *can* run doesn't mean that that process *must* run (now). Instead the process will be added to a list of tasks and will be executed at the operating system's discretion; for example it may be a very low priority process and therefore it is preferable to run something else first.

Points to note:

- The scheduler **despatches** a process by '**returning**' to it.
- The user process can **suspend** (or **terminate**) itself by **calling** the operating system (via a SWI)
- In a **preemptive** system the hardware can also **timeslice** a process; effectively the hardware forces the user programme into an OS call
- Something has to maintain a queue of processes …

Session 5
Counters and Timers

Requires – timer, software readable. Latch count while reading??

100Hz is good because we can keep everything in 8 bits.

Objectives

- Introduce the peripheral interface
- Introduce the timer as a peripheral device
  - Some more peripheral programming
- Frequency division by software (*or* hardware ?)
- Cursor control on LCD
  - Introduce control characters to print routine
    THESE MUST BE COMPATIBLE WITH TERMINAL EMULATOR IF WE'RE TO REDIRECT LATER!!

When operating under real-time constraints it is important to be able to measure the passage of time. Time can be measured (and delays imposed) by totalling the time it takes a section of code to execute. This is a very poor way of producing a delay because:

- Execution time may vary, for example due to cache hits or misses
  - this may be very unpredictable
- Interrupts (see later) could insert extra 'invisible' delays into the code
- If in a multi-tasking environment the code could be suspended for an arbitrary time
- The code may be ported to a machine with a different clock frequency

Thus if a real time reference is needed the programme must have access to a separate, fixed-frequency clock which will not be affected by other system activity. This is normally provided by a hardware peripheral known as a 'timer'.

Definitions

- **Counter**: a hardware circuit which increments (or decrements) according to an external stimulus.

- **Timer**: a counter circuit which counts the pulses of a regular, clock input
- ??? **Prescaler**: a divider (modulo N counter) used to reduce the number of counts a timer needs to make

A single peripheral often offers the functionality of both a counter or a timer (figure 0.5) In practice timers are far more common than simple event counters, so the following description applies primarily to them. A timer can be seen as a subsystem which counts a known (but – likely – programmable) number of clock pulses to indicate a defined interval of time. Because clocks in computer systems are often significantly faster that required a prescaler – typically dividing by a power of two[1] – is often an option to slow the counter down to a 'sensible' rate.

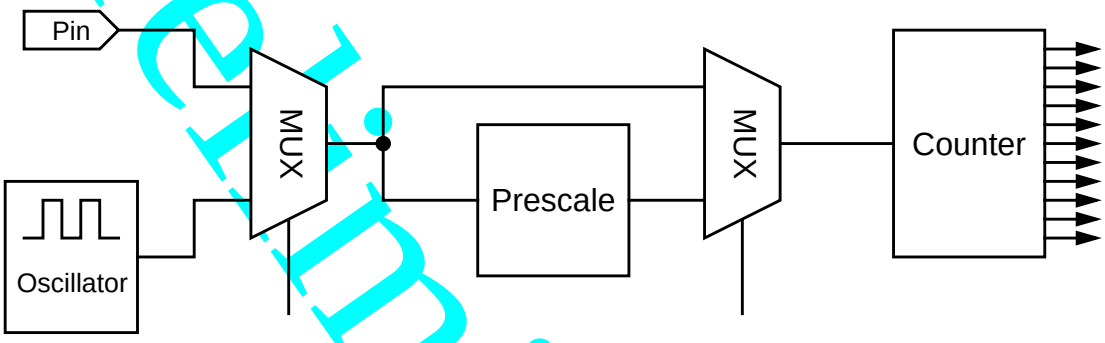


Figure 0.5: Typical counter/timer circuit

There are a number of different ways to implement a timer. The simplest method possibly to use software to count regular events. This will be much more accurate than attempting to count instructions, although it is still prone to errors if a timer event is missed. Faster clocks give more precision, but they increase the demands on the processor. The clock may also be used to produce interrupts (q.v.).

Because the hardware required is relatively, simple timers are often supported in hardware. There are several possible implementations, some examples of which are:

Free-running: a value is incremented or decremented on each clock pulse. This integer will have a fixed length (typically 8- or 16-bit) and will cycle modulo its word length. Usually the value will be readable by software and thus an interval can be calculated from the difference between the current and previous readings (beware the wrap-around). Note that there will always be some uncertainty due to the clock resolution, but this is *not* cumulative because the timer runs freely. Sometimes the counter will be writable by the processor, although this may introduce cumulative errors.

Often free-running timers have one or more comparison registers; these can be set up to a value that the counter will reach in the future and will produce an output (usually an interrupt) at that time. Example??

1. A good example is the common 32kHz 'watch' crystal – its true frequency is 32.786kHz which, if divided by two 15 times gives a 1Hz output.

One-shot: a slightly more sophisticated timer which counts a preprogrammed number of clock pulses and then signals the fact and stops. Typically the counter will count down and stop at zero. Usually it is able to provide an interrupt. This is useful when some activity is required a known time after an input; for example timing the ignition in a car engine at a known, but probably variable, time in the engine's cycle.

This is a digital **monostable**. You will already have used one of these in the "traffic lights" exercise.

Figure 0.6 shows both a free-running and a one-shot timer in 'continuous' use. In each case the timer requests attention and waits for the processor to detect this and service it. Note how the free running timer is able to maintain a regular rhythm whereas the one-shot loses time whilst waiting for the processor.

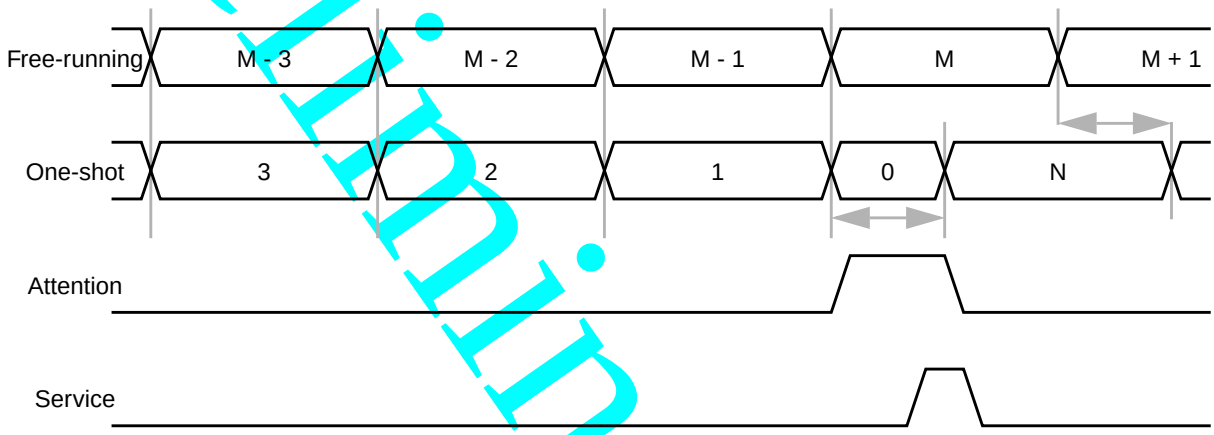


Figure 0.6: Free-running and one-shot timers compared

Reloadable: similar in function to a 'one-shot' but they do not stop at zero; instead they load themselves with a value retained in a separate register. This means that they can be programmed as modulo-N counters and then can be left to run freely. (The count can, of course, be changed.) They have an advantage over 'one-shot' timers in applications where time should be measured continuously in that they continue without processor intervention.

As well as providing processor inputs such as regular interrupts reloadable counters are used for programmable clocks within the system. For example one such timer can be used to divide a system clock down to the **baud** (q.v.) clock used on a serial line. This subsequently runs without further (direct) interaction with the processor.

Practical

Display a decimal counter running at one increment per second using the LCD as an output. It is recommended that you store the count in BCD to make printing easier (just use a standard hexadecimal print routine).

The timer provided produces output pulses(?) at 100 Hz (???). This frequency need to be divided down …

Free running timer here - use one-shot in first exercise?

Reading the timer should be a system call …

Synchronisation issue …

Counter options:

- count in binary (hex). This makes the counter easy (simply increment a variable every second) but the printing hard

Extra:

Count in binary and merely print in decimal. For this you will need a decimal output routine, which is a *non-trivial exercise*. Only attempt this if you are very keen!

The difficulty with printing in decimal is that the number to be output must be divided down first. (This is also true in hex, but dividing by 16 is much easier). The ARM does not have a general-purpose "DIV" instruction so you would need to write a function to do this.

Assuming division is available, here are two different ways of printing a 16-bit[1] unsigned number in decimal.

Decimal print – method #1

Start with a table representing the possible digit positions; for a 16-bit number (max. 65535) this would be:

```
DCD     10000           ; Note these are decimal
DCD     1000            ; numbers
DCD     100
DCD     10
DCD     1
```

Divide the number to be printed by the first entry in the table, keeping both quotient and remainder. Print the quotient, which must be a single digit.

Repeat the process using the remainder as input with subsequent lines in the table.

Decimal print – method #2

Divide the number by 10; stack the remainder and repeat until the number reaches zero. This

1. It is easy to extend to any size.

pushes the digits to be printed onto the stack in reverse order. Now pop and print the stacked digits so they appear in the correct order. This method has the advantage that it will work on an arbitrary sized number; it is also easier to suppress leading zeros if that is required. If you try this, make sure that the number 0 prints correctly though.

Session 6
Interrupts

Objectives

- Introduce interrupts
- Read buttons to change display
- Invention of user interface

What are interrupts?

Interrupts are a mechanism by which hardware can ‘call’ software.

The usual model of computer operation is that the user’s algorithm is implemented in software. Typically this is broken down into procedures which are called to perform ever simpler functions. The lowest level of calls – such as executing instructions – can be thought of as being implemented in the hardware. (????) …

An interrupt is initiated by a dedicated hardware signal to the processor. The processor monitors this signal and, if it is active, is capable of suspending ‘normal’ execution and running an **interrupt service routine**. Effectively it is as if a procedure call has been inserted between two instructions.

ARM has two separate, independent interrupt inputs called “Interrupt Request” (IRQ) and “Fast Interrupt Request” (FIQ). These exhibit similar behaviour although they have their own operating modes; in addition FIQ is a higher priority signal, so it will be serviced in preference to IRQ or, indeed, may preempt an IRQ already being serviced.

Why use interrupts?

Most processors will only run a single **thread** (stream of instructions) at a given time. It is often desirable to try to do two or more tasks at the same time (or at least *appear* to do so). Often this facility can be added with a very small hardware cost.

To illustrate by example think of an office worker writing a letter, a single task. She would also like to answer incoming telephone calls. One way to do this would be to pick up the telephone after typing each line to see if anyone is waiting; she could check to see if anyone is outside the door at the same time …

This periodic checking is known as **polling** and is clearly inefficient. Firstly it involves a lot of unnecessary effort (most of the time there will be no one there). Secondly a caller could be ignored if our author gets stuck on a particular sentence. Thirdly the ‘write letter’ process has to ‘know’ about all the other possible task that could happen in advance.

By adding a bell to the telephone it is possible to enable it to interrupt other tasks. Our heroine can concentrate on her composition without caring about callers, but can still know immediately when one wants attention.

Examples of interrupt processes

Waiting. As has already been seen it is quite common for a processor to have to wait for hardware to become ready. This might be for a fixed time or whilst waiting for external hardware to become ready (e.g. printer off line). In any case polling the hardware is a waste of time the processor could spend doing someting else.

Clocks. A clock interrupt can provide a regular timing reference whilst other processes are running. more on this below …

Input. An interrupt can allow the machine to register an 'unexpected' input, such as the user pressing a key or moving the mouse.

ARM interrupt behaviour

If the IRQ signal is active – and enabled – the interrupt service routine will be called after the current instruction has completed. The ARM inserts interrupts cleanly between instructions, just as if a procedure call had been inserted into the instruction stream.

The following actions are then performed:

- $R14_{irq}$:= address of the next instruction + 4
- $SPSR_{irq}$:= CPSR
- CPSR mode is set to IRQ mode
- IRQ is disabled in the CPSR (note a bit is **set** to **disable** the interrupt)
- The Thumb bit is cleared (if set)
- PC := 00000018

(The FIQ response is similar to the IRQ response (for "IRQ" read "FIQ") except that *both* IRQ and FIQ are disabled.)

Normally address 00000018 will contain a branch to the actual service code.

Note: the only user-accessible registers changed are PC and CPSR, copies of both of which are preserved in special interrupt registers (the PC is modified but the original value is recoverable).

State preservation

"Thou *really* shalt not corrupt the user's state."

The interrupt entry sequence preserves the minimum of the processor's state necessary. Although the interrupt mode has some of its own, private registers (more in the case or FIQ) it

is essential that the service routine does not change any of the state visible to the user process (for example the contents of R0).

Often an interrupt service routine will require some working registers; if this is the case the values in these registers **must** first be saved (the IRQ routine can have its own stack for this purpose) and restored before the user programme is resumed.

Interrupt service exit

When the interrupt service is complete the machine state must be restored. Any registers which have been corrupted must be reloaded by the code itself; finally the interrupt entry sequence must be reversed.

Note that the address saved in $R14_{irq}$ is *not* the address of the next instruction. In order to return to the correct place the saved PC must be decremented by 4. In addition the mode must be restored, interrupts reenabled etc. This is all achieved with the instruction:

```
SUBS   PC, LR, #4
```

The subtract copies the corrected return address back to the PC. The inclusion of the “S” bit, normally used to set the flags, has a special meaning when the instruction destination is the PC and it causes the current mode’s SPSR to be copied back to the CPSR. Because this was saved on interrupt entry it restores the remainder of the processor state. (Remember that the interrupt could have occurred in any mode, not just user mode.)

Practical

Convert your counter to an interrupt-driven system. Reformat the output to make a digital clock. Allow the user to set the time using some input buttons (devise your own way to do this).

When running the clock should be independent of any user programme which can be run separately.

Session 7
System Design and Squeaky Noises

Objectives

- Introduce system-level buses and address decoding
- Introduce the piezzo-electric buzzer

All the exercises up to this point have used pre-prepared hardware. This exercise requires additional circuitry to be designed and integrated with an existing system.

Pointers to file. Extract & display schematic diagram?? (later??)

Some System-on-Chip stuff - putting down and glueing together macrocells

How about the need for another parallel port to drive (e.g.) a 7-segment display???

Computer systems

The “three box” model of a computer (fig. 0.7) divides a system into a Central Processor Unit (**CPU**), **Memory** and Input/Output (**I/O**), all connected by a **bus**. In this case the bus comprises address, data and control signals which may be bundled spearately, thus inside this general bus resides an address bus and a data bus – the other signals are sometimes referred to as a control bus, but form a less coherent set and we will treat them individually.

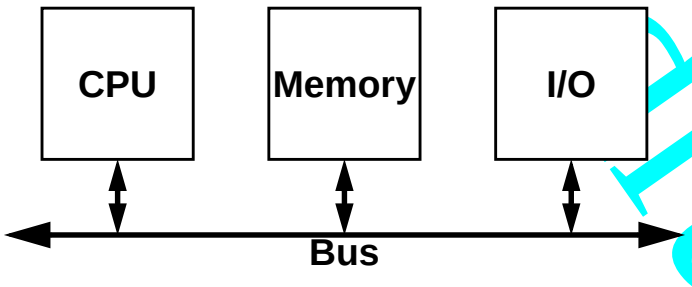


Figure 0.7: The “three box” computer model

In addition to these three large “boxes” there is a small amount of logic used in their interconnection, often known as the **glue logic**. This performs mundane functions such as distributing the system clock and decoding addresses.

To produce a complete System-on-Chip (SoC) all these boxes must be integrated into a single design. However within this laboratory that is not our concern; the CPU and memory “boxes” are already established and only the I/O subsystem needs development. The I/O is all mapped into the Xilinx??? FPGA and is therefore ‘soft’.

Internally an I/O subsystem comprises a number of devices, together with some ‘glue’ to hold them together[1]. A typical I/O subsystem is shown in figure 0.8. Here a particular area of the address space has been decoded externally (within the system ‘glue’) and is used to enable one of the various devices. Which device is selected depends on the particular address which is decoded within the internal ‘glue’. Note that it is not necessary to fill the entire address space.

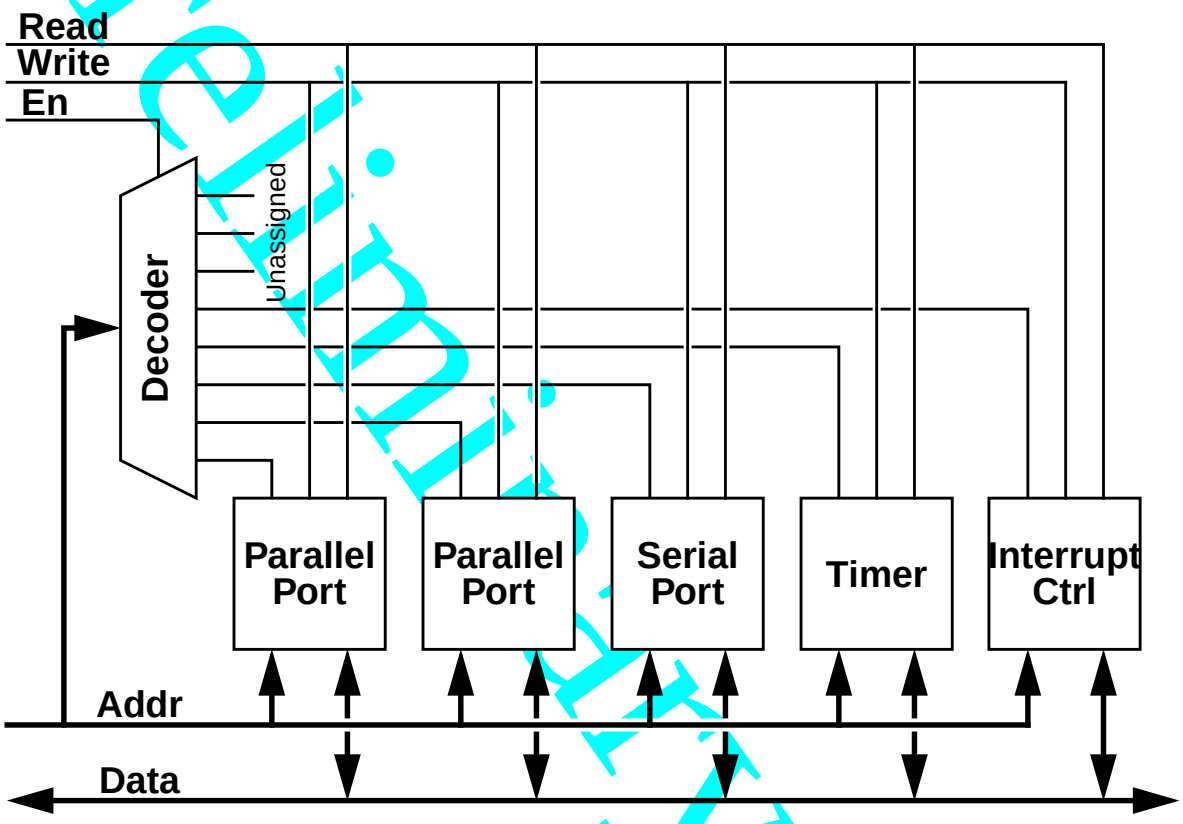


Figure 0.8: Typical Computer I/O Subsystem

Address decoding

The address decoder (together with the overall I/O Enable) determines the location of each peripheral within the address space. When choosing how to decode the peripherals it is normal to first consider the number of address lines used within each device. For example a parallel port may contain (say) two addressable registers, thus requiring one address signal for itself. However if (for example) the serial interface occupies four addresses then it would be usual to assign four addresses to the parallel ports too, and ignore those not required.

It is quite usual for peripherals to occupy only eight bits of the data bus. With a wider bus (such as on the ARM) the peripheral registers will therefore be spread four addresses apart (the inter-

1. The memory subsystem is very similar internally.

vening three bytes being ignored). Thus address signals A[1:0] cannot be used to select registers inside a peripheral. It would be normal (although not essential) to choose the next few bits to feed to the individual devices Thus, in the preceding example, the parallel ports would receive A[2] whilst the serial port would be connected to A[3:2].

When this assignment is complete the separate peripherals can be decoded. For future expandability let us say that A[4] will be reserved in case we add a peripheral with more (up to 8) registers later. The decoder shown in figure 0.8 requires three address inputs (there are five peripherals) so let us assign them to A[7:5]. This yields an address map (partially) shown in figure 0.9.

Address	Byte 3	2	1	0
XXXXXX00				Parallel A #0
XXXXXX04				Parallel A #1
XXXXXX08				Parallel A #0
XXXXXX0C				Parallel A #1
XXXXXX10				Parallel A #0
XXXXXX14				Parallel A #1
XXXXXX18				Parallel A #0
XXXXXX1C				Parallel A #1
XXXXXX20				Parallel B #0
XXXXXX24				Parallel B #1
XXXXXX28				Parallel B #0
XXXXXX2C				Parallel B #1
XXXXXX30		Unused		Parallel B #0
XXXXXX34				Parallel B #1
XXXXXX38				Parallel B #0
XXXXXX3C				Parallel B #1
XXXXXX40				Serial #0
XXXXXX44				Serial #1
XXXXXX48				Serial #2
XXXXXX4C				Serial #3
XXXXXX50				Serial #0
XXXXXX54				Serial #1
XXXXXX58				Serial #2
XXXXXX5C				Serial #3
XXXXXX60				Timer #0
XXXXXX64				Timer #1
…				…

Figure 0.9: Typical I/O Address Map

Two points are noteworthy here:

- Many addresses are unused
- Some (most?) addresses are incompletely decoded (e.g. “Serial #0”) and so the

same register appears at more than one address

Both of these characteristics are typical of I/O address maps.

It is a *really good plan* to equate (EQU) these addresses to a *meaningful* label in a header file and use only the label within the programme. This confers the following advantages:

- References to the ports become more obviously understandable
- Only a single block of references needs updating if the system is reconfigured

Piezzo-electric buzzers

Piezzo electric crystals are materials which change shape when an electric field is applied[1]. As switching a voltage on an output is a relatively easy thing to do they are widely used by, for example, mobile telephone manufacturers to make irritating squeaky noises. This can be done most simply by grounding one terminal and switching an output bit from 0 to 1 and back to 0 again at the desired frequency. By changing the switching frequency different tones can be played.

A louder noise can be produced by increasing the applied field. With purely digital electronics this can be difficult, but a doubling in amplitude can be achieved by driving both terminals of the device in *antiphase*[2]. A *really* sophisticated driver could use voltage steps to output something smoother than a square wave, although this requires considerably more effort.

Note that piezzo-electric buzzers are not very responsive at low frequencies so sound “best” at a few kilohertz.

Powerview revisited

Because the laboratory boards have all their peripherals in an FPGA it is possible for the user to define and customise these. Previous exercises have used precompiled peripherals; in this exercise you add your own.

Create a Powerview project for this laboratory using the usual setup script (i.e. “mk_pv 124”)

1. They also produce an electric field if they change shape, hence their use in firelighters.
2. i.e. at the same time, but in opposite directions.

and invoke it as appropriate for this laboratory (i.e. “pview 124”).

MORE ABOUT THE SETUP --- TEMPLATE FILES???

Interfacing to the ARM microprocessor

The peripheral interface

Description of bus interface . blah, blah (HERE??)

How to: powerview

- compile
- download

Practical

Modify the existing I/O controller to include an output to drive the piezzo buzzer. Make this play a tone, or, preferably, a tune.

The simplest way to do this is to add an addressable output port (i.e. a latch) and drive the relevant outputs in software, using a system timer.

A slightly harder method (i.e. involving more hardware) would be to build a loadable, free-running, counter which could be set at different frequencies by the CPU and could then do the tone generation on its own.

Acoustics for non-musicians

An “octave” is a factor of 2 in frequency, and is divided into eight notes (but is really *twelve* semitone divisions). In modern tuning – known as “equal temperament” – each of these divisions is equal on a logarithmic scale. This means each semitone has a frequency about 6% different from an adjacent one.

$$factor = 2^{1/12} \approx 1.059$$

Starting a scale at 1 kHz the semitones have the frequencies as shown in the table below. The bold text denotes the notes of the major scale.

Note	Frequency (kHz)	Period (µs)
Do′	**2000**	**500**
Ti	**1888**	**530**
	1782	561
La	**1682**	**595**
	1587	630
So	**1498**	**667**
	1414	707
Fa	**1335**	**749**
Mi	**1260**	**794**
	1189	841
Re	**1122**	**891**
	1059	944
Do	**1000**	**1000**

The period of the waveform is also given as this is the time taken to complete a whole cycle. It is the time which the circuits are used to determine (whatever solution is used).

Session 8
An Interrupt Controller

Objectives

- Nice, straightforward peripheral
- Enable later, complex systems
- Introduce hardware/software mixing??

Interrupt priorities

Interrupt despatch

If a processor had a dedicated interrupt signal for every possible source of interrupts, interrupt servicing would be easy. This is generally not the case; usually there are a number of different devices which are forced to share a single interrupt.

The simplest method of sharing is simple to OR the possible interrupt signals together. That way if *any* one (or more) of the devices wishes to interrupt, the processor will be able to detect this. The processor still has to be able to determine which device(s) is(are) activating the signal however.

One approach to this is to check the status of each possible device by interrogating its status register; this assumes that every device has a register containing a bit which reflects its interrupt status (or similar), but this is usual in a peripheral device. It would be normal to service these in a defined order, so that the most urgent (‘highest priority’) is examined first.

The question of what to do when a device has been serviced I leave open. Should the code return directly?, search on for other active interrupts?, start again at the beginning?, … There are arguments for all of these philosophies, and probably more besides.

The problem with checking devices in software is that it is a slow, serial process. It is much faster to be able to identify an interrupt source directly. This can be done with a dedicated interrupt controller which can allow the processor to check the status of the various interrupt signals directly (e.g. as different bits within a single register). The controller can also perform some other simple functions, such as allowing individual interrupts to be enabled/disabled. It could also perform more sophisticated functions, such as acting as a priority encoder so that the highest priority interrupt present is already encoded for the processor. A *really* clever controller might even allow the interrupt priorities to be programmable; alternatively this may be better done in software. The borderline between which functions belong in hardware and

which in software can sometimes be fuzzy!

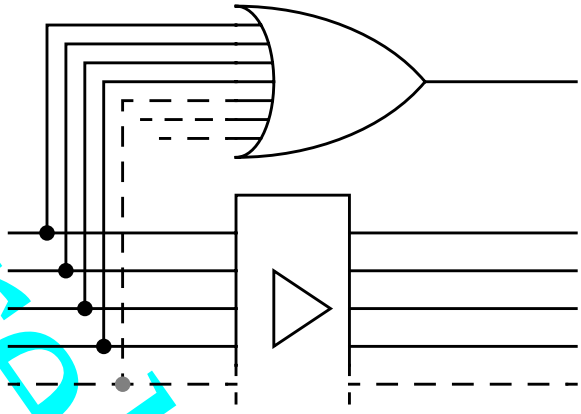


Figure 0.10: Very basic Interrupt Controller

Having determined the source of the interrupt the despatcher is analogous to the despatcher used for SWI calls.

Mention interrupt vectoring et al.

Synchronisation

Metastability …

Experimental use of call-out box.

By their nature incoming interrupts are **asynchronous**, i.e. they can occur at *any* time. This has the potential to cause problems because the processor will sample the signals at fixed times (related to its clock cycle) when it expects the interrupt signals to be stable. If a signal changes just as it's read it is open to misinterpretation.

Metastability …

Synchronisation …

Practical

Produce an interrupt controller to your own specification. It should be capable of mapping a number of interrupt inputs (say, eight) to a single processor interrupt signal, yet the processor should be able to use it to determine the interrupt source (perhaps by reading a status byte?).

An additional, useful feature is the ability to enable interrupts on an individual basis. If you include this facility think what the default ("power-up") state of the enables should be.

Session ?? (earlier than this? later?)
Stepper Motors

(Need(?) to ensure that the drivers are disabled except when in use.)

Stepper Motors

There are several ways of operating motors under computer control, but many involve analogue outputs or feedback. A **stepper motor** avoids this by presenting a digital interface. There are several forms of stepper motor, as a quick WWW search should reveal! However as a generalisation they can be regarded as rotary motors which turn with a predetermined number of fixed steps.

Stepper motors are more complex (and therefore expensive) than simple D.C. motors but offer significant advantages in applications such as robotics. For example it is possible[1] to tell how far a stepper motor has moved simply by counting the steps sent to it. As the step size can be known the exact postion of the motor can be determined at all times. It is also possible to control the direction of the motor by altering the command order (see below) and its speed by varying the time between steps. For these reasons stepper motors are popular in applications such as **robotics**.

Driving a stepper motor

Each position has an associated input which, if activated, causes the motor to move to this position by energising an electromagnet. An analogy will serve here.

Consider a clock face numbered in seconds (0-59) with a second hand driven in 6° steps (i.e. one step per second). This *could* be done with sixty individual inputs, but that would be a nightmare of wiring; instead let us use four inputs. The first input would be used to turn the motor to positions 0, 4, 8, 12, …, the second to positions 1, 5, 9, 13 … etc. Thus when an output is activated the motor will turn to *one* of the corresponding positions and remain there. In practice it will move to the nearest corresponding position, e.g. from 0 to 1 or from 8 to 9.

Thus by successively activating outputs it is possible to cause the motor to take repeated steps in the same direction. Thus an output bit pattern (assuming active high) would look like:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
1 0 0 0
...
```

By altering the pattern slightly the motor will turn in the other direction. i.e.

1. In theory, assuming steps are not lost due to overloading.

```
1 0 0 0
0 0 0 1
0 0 1 0
0 1 0 0
1 0 0 0
...
```

These patterns can be mapped quite simply onto a parallel output port; stepper motors are therefore easy to control in software. (Hint: shift instructions may be useful.)

Laboratory equipment

Equipment paramenters (Speed limits, addresses et al.)

Need to get some numbers for the experimental equipment,

Need a ‘catchy’ practical here.

Mechanical considerations

A motor is a physical mechanism with properties such as inertia. It is therefore not possible to cause it to turn infinitely fast, or even as fast as a computer can output patterns. There is a minimum step time, so a delay should be provided for between steps. This delay depends on the characteristics of the motor and the load it is moving. It can also depend on its history.

Rotating systems have a number of properties which are similar to the (probably) more familiar linear mechanical systems. Thus they have *angular momentum* (sort of mass x speed[1]) which is subject to *torque* (turning force). This can be useful to know because it is possible to accelerate the motor over a period to high speeds.

The implication for stepper motors is that a naive, conservative driver must always step the motor with the maximum delay, as if the motor is stationary between steps; a more sophisticated algorithm can reduce the delay between steps to accelerate the motor once it is moving (again there is a practical speed limit, but much higher). This means that the motor can move its load to the desired position more quickly. Of course if this is done it is also necessary to have a controlled decelleration to arrive stopped at the correct position – the brakes are only as good as the accelerator!

1. This is certainly not physically rigorous!

Practical

Get from A to B

Option: accelerate motor.

Extra:

As described above the angular resolution of the stepper motor is one step. If finer positioning is necessary it is possible to move the motor in increments of half a step, purely under computer control (i.e. no mechanical aids).

How might this be done?

Session 10(?)
UARTs

Requires hardware compiler, terminal emulator on PC

Objectives

- Real world interface
- Illustrate structure in redirecting character output
- first experience in interconnection
- plenty of scope for debugging

UARTS

UART stands for “Universal Asynchronous Receiver/Transmitter”; this is a a serial interface device. “Universal” simply implies some configurability (and is mostly marketing) “Asynchronous” in this context means (perversely) at a predefined frequency and “Receiver/Transmitter” implies that the communication may be bidirectional.

Before examining the function of a UART it is necessary to define the format of the serial message. The most common serial form breaks up the transmission into bytes and transmits each byte individually. Each byte is ‘wrapped’ with some synchronisation and possibly some error detection bits (see fig. 0.11).

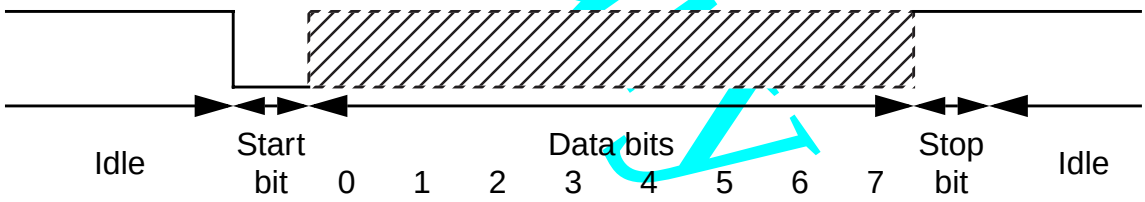


Figure 0.11: Serial data transmission

When not in use the serial line is in some known, idle state[1]. When a byte is to be sent the transmitter first *changes* the state of the line for one bit time – this is known as a **start bit** and signals that the idle period is over. Following this the data is transmitted (usually least significant bit first) and the line returns to the idle state until the next byte is to be sent. If, as is often the case, the next byte is ready immediately it is important that its start bit can be detected. If the previous data had a ‘1’ in bit 7 there would be no transition at the start of the packet, so the transmitter always insets an extra **stop bit** at the end of a packet to ensure that the line makes the ‘0’⇒’1’ transition at the start of each byte. This is essential because the receiver uses this to synchronise with the incoming data.

1. Unfortunately, by convention, this is usually the ‘high’ state (data=1) as shown in figure 0.11.

Note that – *in this example* – the minimum time to transmit an 8-bit byte is therefore ten bit times. This illustrates the difference between two commonly confused numbers: the **bit rate** and the **baud rate**.

Put simply, the baud rate is the rate at which the signal on the wire (or whatever other medium) changes, The bit rate (sometimes referred to as bits-per-second or **bps**) is the rate at which *information* is transmitted; start and stop bits do not carry information for the user. In the example above the bit rate is 80% of the baud rate – assuming that the line can be kept busy continuously.

In other examples this may be different. For example some serial lines may include an extra **parity bit** (for error checking) or may require more than one stop bit. This does not affect the baud rate but will reduce the bit rate.

The other 'performance' figure sometimes quoted is **cps** or characters-per-second. In our example this would be $\frac{1}{8}$ of the bit rate. Note that characters are not always 8-bits long; ASCII[1] was designed as a 7-bit code (for seven hole paper tape) and some systems use even fewer bits (e.g. RTTY[2] uses 5 bit characters).

The most basic function of a UART is to convert parallel data to a serial form and vice versa. This is complicated by the fact that the transmitter and receiver are **asynchronous**. Although the two units both know what a bit time is, they both judge this by their own clocks and *two different clocks can never quite tell the same time*. In practice considerable leeway is usually allowed.

Firstly let's look at the job of the transmitter. This serialises bytes as they are supplied, each bit being shifted at a predefined rate (the bit rate) – as closely as the transmitter can judge. In addition to this the transmitter must append the start and stop bits. Serialisation can be done in software but is normally uses dedicated hardware in the form of a simple Finite State Machine. This FSM stays in its idle state until a byte is written to it; it then **shifts** this out before returning to the idle state again (fig. 0.12). To prevent the next byte arriving before the current byte is completely transmitted it is usual to include some sort of **busy** signal to prevent this.

If, as may be expected, the status of the transmitter is read by a microprocessor, the busy signal must be accessible to the processor. It is therefore normal to provide the transmitter with two separate registers: a transmitter data register (often called "TxD") for the byte itself and a status/control register. So far we have only described a single bit (read-only, at that) which could occupy the transmitter status register. The processor, wishing to send a byte, can then poll the status register until the transmitter is *not* busy before writing to the data register. This process should be familiar from the LCD interface.

By renumbering states (idle = 0) this can be made with a small change to just bit 0 of a decade counter. Steal from CS121. Library issue?

1. American Standard Code for Information Interchange
2. Radio TeleTYpe(writer)

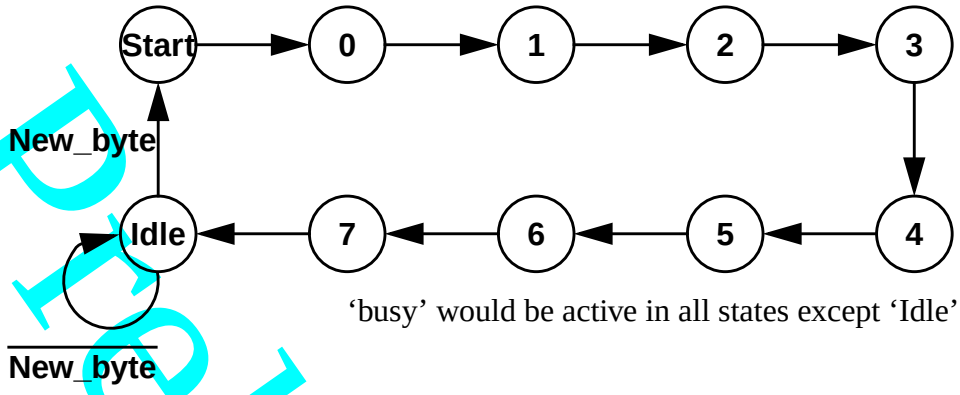


Figure 0.12: UART transmitter state diagram

A more advanced transmitter can give more freedom to the processor by a process of **double-buffering**. In the example described above the transmitter will go **idle** (not busy) around the time that the stop bit is being sent; it cannot transmit the next start bit until it has the next byte to send and so it will drop back into the idle state unless the processor can refill the data register within one bit time. This may be tricky, especially if the transmitter is serviced under interrupts (see later). It is not a disaster if the transmission idles a bit – nothing is lost – but it does waste valuable time[1].

Double buffering helps guard against this by providing an extra data register to hold the *next* byte for transmission (fig. 0.13). When the shift register finishes one byte it can then attempt to refill itself from this register. If the next byte is already present it can copy this into itself and carry on shifting without pause. Of course when the **transmitter holding register** is empty shifting has to stop, but the processor has an entire byte time to refill this. (Note: the maximum *rate* at which bytes are sent does not change, this merely buys some timing freedom.)

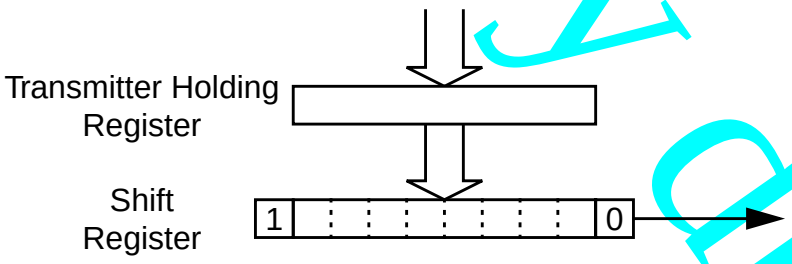


Figure 0.13: UART transmitter double buffering

Note that, in this scheme, the processor is primarily interested in the full/empty state of the transmitter holding register, not the shift register itself. Another state machine controls the transfer between the registers so that, when the shift register is empty and the holding register is full, the holding register contents are moved down.

1. The story of a receiver which cannot keep up is a different matter!

The **receiver** is basically an implementation the reverse process; data is shifted into a register one bit at a time and read out in parallel. Usually double-buffering is used here too, so the assembled byte is copied into a holding register so that the CPU has more (flexibility in) time to read it. In this case it must be assumed that data can come in at the maximum speed (i.e. the bytes are 'nose-to-tail') and so the holding register must be cleared in real time. Unlike the transmitter, where tardiness simply makes the connection less efficient, too much **latency** in the receiver will cause bytes to be lost. This is known as an **overrun error**; most UARTs will detect this condition and have a status bit to indicate that it has occurred. Some UARTs extend the Receiver Holding Register into a small First-In, First-out (**FIFO**) buffer to increase the flexibility of the interface. Note that this FIFO can be in three states: empty, not empty and full. WHAT ARE THE IMPLICATIONS?

Data recovery in the receiver is complicated by the need to **synchronise** with the incoming bit stream. The receiver knows the (approximate) bit period, but it does not know the phase of the incoming signal. If it sampled the line only at the bit frequency it might start sampling somewhere near the edge of a bit, and the mismatch between the frequencies could cause it to miss a bit (or sample the same one twice). Figure 0.14 shows a possible example of this where the receiver clock is about 4% faster than the transmitter clock.

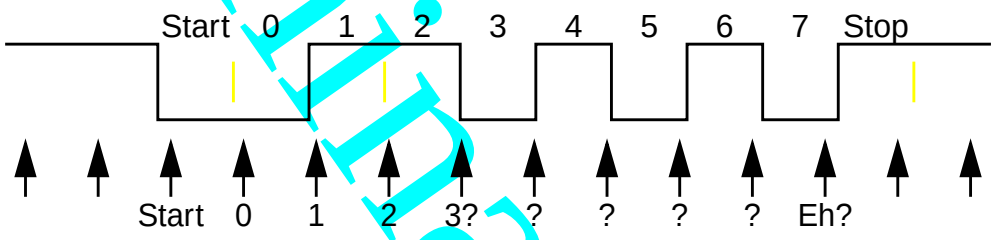


Figure 0.14: Unsuccessful attempt at serial data recovery

Clearly the sampled bits have lost synchronisation during the byte. (*In this example* this could be detected because the stop bit is not correct – an example of a **framing error**.)

Ideally samples should be taken near the centre of the incoming bits. To find this position the receiver begins by sampling at a higher frequency (traditionally 16x) and looks for the transition at the beginning of the start bit. Having found this it can count to eight (if sampling at 16x the bit period) and then take samples every sixteen clocks. Because the first sample is guaranteed to be *near* the middle of the start bit, and the bit rates between transmitter and receiver are fairly closely matched, the bits should all be sampled correctly. Resynchronisation takes place on a byte-by-byte basis, so the drift can never be too far out. Figure shows a receiver sampling at only 4x the bit frequency; the clock mismatch is 4% (as in figure 0.15) but this time the data

is recovered successfully. With a faster sampling clock a larger mismatch can be tolerated.

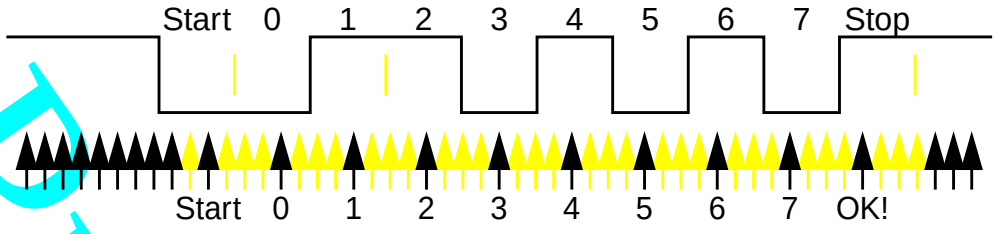


Figure 0.15: Successful attempt at serial data recovery

Practical

Using the partial UART provided (??? location/name???) (or otherwise) as a template implement a UART transmitter. This may include or omit double buffering via a Transmitter Holding Register, depending on how sophisticated you wish to be.

Use your UART to print "Hello world" (or, possibly, clock) onto your workstation via a serial cable. You can do this by replacing the character print routine from an earlier exercise.

Hints

- The heart of the unit is a **shift register**[1]. In the case of the transmitter the shift register will be Parallel In, Serial Out, or "PISO".
- Don't forget the start bit.
- Reference to figure 0.12 shows that the state machine progresses through ten states. The only choice is that it can be *disabled* from counting in one of these states.
- State numbering is arbitrary; for example the state machine does not have to code state "5" in figure 0.12 as "0101".
- Making this system work requires both correct hardware and software. You will find it *significantly easier* if you *simulate* and debug the hardware before incorporating it into the design. Sympathy will be withheld from anyone who does not do this and consequently gets stuck!

Clock requires some control codes defining in earlier exercise!

1. An appropriate text book will be more informative than the WWW here.

Really advanced students could add an interrupt facility(?)

Really advanced students could do a receiver too?

Write serial driver ?????

Interrupt routines, software buffering

Allows the introduction of ring buffers as FIFOs

Unfortunately the receiver end of things can overrun which starts to imply need for flow control and, probably, too big a can or worms.

Recommended further reading

Find one or two UART product datasheets – these are generally available on the WWW – and peruse them for their capabilities. Note that most are programmable in various ways, including the baud rate and the number of bits per character. What else if often available?

If you need a starting point the “Chip Directory” is a good place. Some possible part numbers are 2681, 8530, 16550, …

Session 11(?)
Digital to Analogue Conversion

Objectives

- DAC introduction
- Real time constraint using timer (interrupt?)

Whilst some ‘real-world’ conditions(??) are binary (light on or off, door open or closed, …) or discretely quantised (????…) many are continuous functions (speed, altitude, ???). However digital systems do not (cannot!) represent them in this way; instead they choose a number of values as approximations. An analogue quantity is then represented by the nearest digital value.

For example, the speed of a car could be represented as number in (say) kph; this will never be the *exact* speed of the vehicle, but can always be within 0.5 kph of the actual speed. In most applications this would be adequate (does the driver *want* more precision?) and quite convenient in that an 8-bit number would serve for most production vehicles.

Clearly the equipment must have some means of converting between analogue and digital values. Because digital to analogue conversion is somewhat simpler we will deal with that first.

Applications

CD player, …

A Simple Digital to Analogue Converter (DAC)

Digital control is well suited to controlling binary switches, which can be either ‘on’ or ‘off’. Imagine a set of these switches attached to (say) valves controlling the flow of water through a set of pipes. If the valves are all the same then opening two of them will allow twice the flow of water than opening just one; opening three would triple the flow, etc. Clearly the output could be switched from nothing to the maximum – in steps – with purely digital control. With enough (small) increments the output looks as if it can be continuously varied.

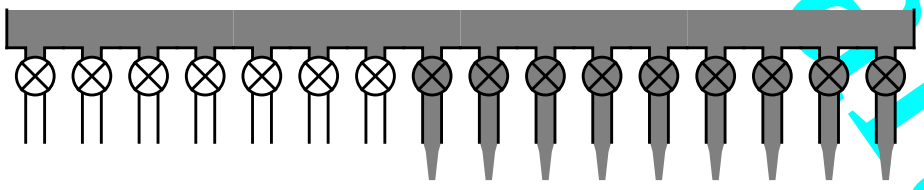


Figure 0.16:

Using 2^N identical switches forms an effective but expensive Digital to Analogue Converter (DAC). It is usually preferable to reduce the hardware requirement. The most common solution uses only N switches but each switch controls a different sized ‘pipe’. Thus the most significant bit controls the fattest pipe, the next bit a pipe of half the area and so on.

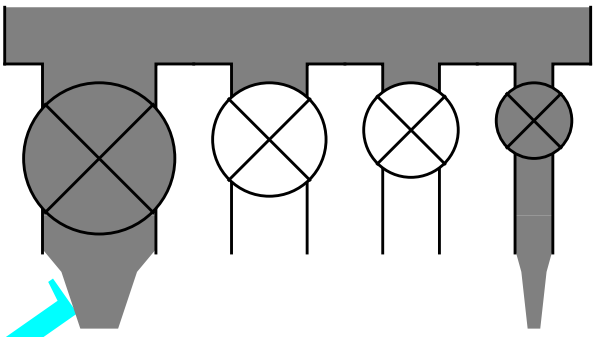


Figure 0.17:

Clearly in either of these schemes (but more especially the second) it is necessary to ensure that the ‘pipe’ conduct the expected flow quite precisely. An especially important issue is to ensure that the largest pipe conducts more than the sum of all the smaller ones (fig. 0.17), i.e. the cumulative errors do not add up to a significant proportion of the step size. This gets harder to ensure as the number of bits (‘pipes’) increases and thus the relative sizes vary more.

Whilst water flow is one possible ‘analogue’ output from a digital system it is probably an unusual one! Far more useful is the ability to control quantities such as electric current – which allows a loud-speaker to be driven in a stereo system. However the technique used is analogous, with transistors substituting for valves and …???

Analogue output

Waveform out to {’scope?, LED?, speaker?}

’Scope is easiest, but implies a need for capital equipment.

Go to two channels (or even 2.5/3 for Z) and draw up a picture on the ’scope

Session ???

Dimmer switch

Control a lamp (LED?) using phase control.

Objectives

- Another 'DAC' technique
- User interfacing revisited??

(PWM under pure software control or phase control with thyristor ???)

Phase control harder to set up and probably harder to do! :-(

Could this be used with a threshold phototransistor as a range sensor? (AP?)

Session 12(?)
Analogue to Digital Conversion

Requires DAC, comparator, plug-in box with potentiometer to provide input

Objectives

- Learn more about analogue interfacing
- "Putting it together" some more

Analogue input

Produce ADC from DAC and comparator (hardware or software?)

Throw up waveform onto PC

((Sound to light type of app. - probably too hard))

Applications

Audio sampling, frame-grabbing, …

Analogue to Digital Converters (ADCs)

There are a number of different ways to convert an analogue signal into a digital approximation; one thing all the methods have in common is that they do not convert 'directly', instead they rely on placing the analogue input in its correct position within a series of 'guesses'. [THIS IS NAFF TEXT]

This is because one of the few simple operations which can be performed with two analogue signals is a comparison; it is relatively easy to see if one voltage is higher or lower than another, rather harder to say by how much. Furthermore a 'higher or lower' output is a binary digital value. Note that, in the analogue world, the signals *cannot be equal* – there must always be a difference if you look hard enough!

The most commonly encountered ADCs work by **successive approximation**. An analogue input is introduced with a value in a known voltage range. As a simple example imagine the range is 0-16V and the input is, on this occasion (about) 6.32V. We shall perform an 8-bit conversion, so 00 represents 0V and FF represents 16.00V

The conversion begins by 'guessing' the answer. In order to maximise the information gained

the guess should cut the available range in half; therefore try the value 80. This is passed through a DAC (which will convert it to 8.00V) and compared with the input; the guess is too high.

The result of the comparison is noted in bit 7 of the guess (a zero means we were too high) and the next bit is tried; the next guess is therefore 40 ⇒ 4.00V. This guess is too low, so the next bit of the conversion (bit 6) is retained as a one, and bit 5 is set. This process is repeated until the required precision is obtained.

Table 6:

Guess (hex)	Guess (binary)	Trial voltage	Comparison
80	1000 0000	8.00	High ⇒ 0
40	**0**100 0000	4.00	Low ⇒ 1
60	0**1**10 0000	6.00	Low ⇒ 1
70	011**1** 0000	7.00	High ⇒ 0
68	011**0** 1000	6.50	High ⇒ 0
64	0110 **0**100	6.25	Low ⇒ 1
66	0110 0**1**10	6.375	High ⇒ 0
65	0110 01**0**1	6.3125	Low ⇒ 1

Observations:

- the result of each comparison is included in the accumulated result
- the process could continue to achieve higher precision
- only one DAC and one comparator are needed
- the time taken for the conversion is proportional to the number of bits required
    - there can be a significant time lag between starting the conversion and obtaining the result

The result of the conversion is therefore 65, the closest representation which could be obtained. This will be within half the least significant bit (about 0.03V) of the actual input (i.e. the input is between 6.28V and 6.34V).

Flash converters … (???)

Practical

Build an analogue to digital convertor using the DAC and comparator provided. You can use whatever hardware/software combination suits you.

Attach a potentiometer to your analogue input; read and display the input continuously on the LCD panel (?).

Session 13
A Digital Oscilloscope

Requires front end on PC THIS REQUIRES REAL WORK

IDEALLY have a 'black-box' wave generator for them to analyse

Objectives

- Introduce **sampling**
- Talk to software package – **protocols**
- Revise the concept and capabilities of an oscilloscope

An oscilloscope is a common tool for displaying and measuring waveforms. At its simplest it is an analogue device which traces out an input voltage level on a screen; the *persistence* of the screen (and the eye) helps the user to see the shape of the trace. However (usually) an analogue 'scope is only useful for viewing *repetitive* waveforms where the trace is periodically refreshed.

Many more modern 'scopes are digital; these **sample** the incoming signal and store the result in RAM from where it can be displayed, printed etc. This has the advantage that a 'one-off' event can be captured and preserved. The disadvantage of digital oscilloscopes is that it is expensive (and in some cases impossible) to sample accurately at the required speeds.

Sampling

To represent an analogue value which changes over time it is necessary to produce a two-dimensional trace. This could be done with a set of coordinates (the value together with a time stamp), however it is more usual to omit explicit time information, instead assuming that **samples** are taken at predetermined, regular intervals. This generally saves memory because only a

single set of values need storing.

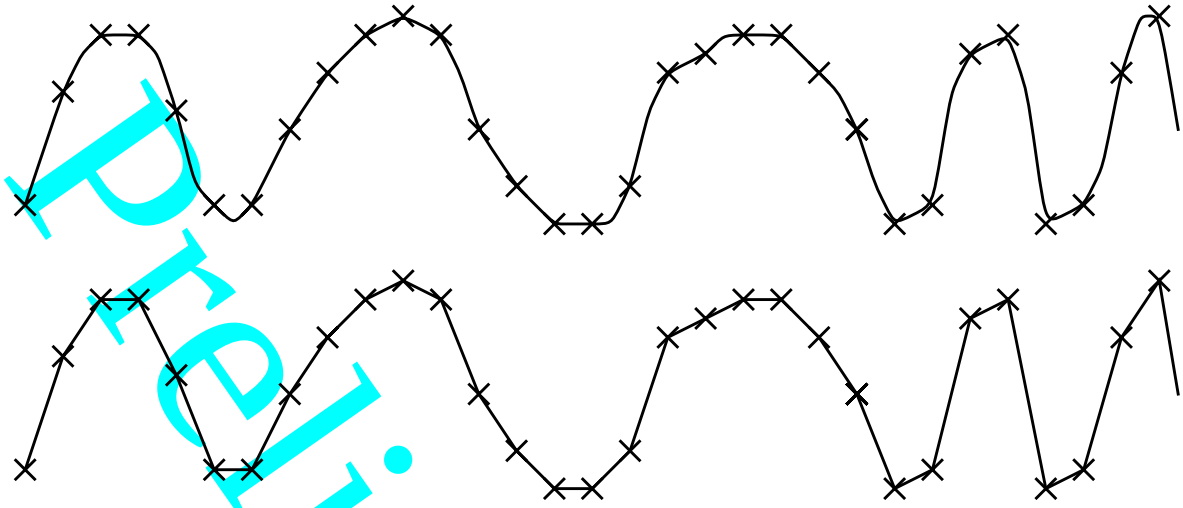


Figure 0.18: Sampling

The sampling interval may be changed to view faster or slower waves. …

Clearly there is an upper limit on sampling frequency; this governs the fastest waves which can be viewed. This is known as the Nyquist criterion, which states that the sample rate must be at least twice the highest frequency described[1]. If the sampling frequency falls below this rate **aliasing** will occur. It is therefore important to choose a sample frequency which will represent the frequency band of interest.

Figure 0.19: Aliasing figure

For example each channel (it is stereo) of a CD recording is sampled at 44.1 ksamples/s. This means that it can reproduce sounds with frequency components up to about 20kHz. As this is the upper range of human hearing anything above this is unimportant. (A dog could tell the difference between a CD an a live performance, but probably doesn't care.)

Triggering

As well as displaying a trace an oscilloscope must know which part of the trace to display. This can be especially important if the display is continually updated from a repetitive waveform

1. I'm not sure this is exactly how Nyquist expressed it.

(such as a sine wave); if the trace starts at a random point (phase) of the wave the trace will jitter about the display and be unreadable.

The usual cure is to set up a **trigger** condition, which starts the trace. At its simplest this will be the trace *crossing* a particular value (in a particular direction) – for example crossing from below to above 'zero'. The oscilloscope can hold-off starting its sweep until this condition is met and thus each sweep will start at the same point (within the resolution of the 'scope).

Practical

Build a digital oscilloscope using your existing ADC. Output your samples to a PC display via a serial line. The protocols required are described ??? …

Session ???

Digitised speech

Scope it

Sample & playback

Speaking backwards et alia

Session ???

’Phone across lab. (by IR?)

Clearly we need modulated IR; is this available from a module (hate to have to filter ourselves). If so what is the interface? Can we just send byte streams?

Protocols: How is the stream controlled? How are the packets wrapped?
Need to talk to SKB et alia to make sensible choices here.

What other games could be played? Speaking clock using user provided samples …

More Exercises

Give a choice of some bigger projects?

Disordered ideas …

Robot ‘arm’

Stepper motor controller. Race pointer from here to there.

Look at accelerating ‘arm’

Requires stepper drivers and some sort of hardware

Objectives

- Another interfacing technique
- Potential for more ‘clever’ algorithms to go faster (some ideas about torque, predictive control …)

Control/feedback

Dial up a temperature (or other), get metal block(?) to that value and retain.

Requires some hardware (calorimeter, heater, thermometer); also H&S check

Objectives

- Real control problem
- Employ simple feedback
- Illustrate overshoot, damping etc. lead-in to predictive control

---

Piezzo buzzer ...

Mouse input

Keyboard scanning

LED matrix

Weighing machine

Etch-a-sketch (stepper motors?)

Turtle (tethered or else battery problems)

UART receiver (Classic serial line driver, Xon/Xoff …)

Disc controller (data recovery is hard, but filing system could be done)

IR(?) communications - protocols? Networking

Ultrasound (or IR??) distance sensor

Acoustic owl? Follow lightsource with output?

Video

MP3

Ramping ADC

Emulation via a trap (?)

Abort handling ????

Board requirements

- ARM
- ROM (monitor)
- SRAM
- UART/drivers for download
- Xilinx
  - clock at defined freq.
  - Buttons & LEDs
  - DAC
  - comparator
  - (Analogue mux?/SAH?)
  - LCD panel
  - line drivers for serial line
  - IR tranceivers (filters?)
  - Stepper drivers
  - …
  - …

Glossary of Terms

Accumulator: A register which is used repeatedly in an arithmetic function to 'accumulate' the result.

Active: A signal which is indicating a 'true' condition (see also *asserted*, etc.).

Active edge: The *transition* of a signal (such as a *clock*) which can cause a circuit to change its *state*. See also *inactive edge*.

Active high: A signal whereby a 'high' logic level indicates that the condition is *active*.

Active low: A signal whereby a 'low' logic level indicates that the condition is *active*.

ALU: Arithmetic/Logic Unit.

AND: see *Boolean logic*.

Asserted: A signal which is indicating a 'true' condition (see also *active*, etc.).

Asynchronous: Having no fixed relationship in time. Asynchronous signals may change in any order or even simultaneously.

Attribute: An attribute is some miscellaneous value associated with some other item. They are as diverse as a propagation delay for a *gate*, a pin number on an integrated circuit or a date on a diagram.
Attributes are yellow in Powerview.

Back-annotation: The extraction of physical data from a *layout* in order to increase the accuracy of a *simulation*.

BCD: see *Binary Coded Decimal*.

Bidirectional: A *net* (or, more usually, a *bus*) which has more than one output connected to it; data thus 'flow' both ways along at least some of the wires. e.g. a processor data bus. See also *unidirectional*.

Binary: base 2, and counting systems etc. derived therefrom.

Binary arithmetic: Addition, subtraction etc. of *binary arithmetic values*.

Binary arithmetic values: Numbers with each digital signal being a single base two digit with a value of "1" or "0".

Binary Coded Decimal (BCD): A representation of numbers where four bits are used to encode the values 0..9 only. This facilitates easy translation to decimal, but makes storage less efficient (values are 'wasted') and arithmetic functions (etc.) more complex.

Binary Counter: a *counter* which is modulo 2^N.

Bit: contraction of Binary digIT. A variable which can have one of two values. The state which may be indicated by a single digital signal.

Boolean algebra: The textual manipulation of *Boolean Logic*.

Boolean logic: The combination of the *Boolean logic values* using simple functions such as AND, OR and NOT (fig. 1.1).

In1	In2	Out
0	0	0
0	1	0
1	0	0
1	1	1

AND

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	1

OR

In1	In2	Out
0	0	0
0	1	1
1	0	1
1	1	0

XOR

In	Out
0	1
1	0

NOT

Figure 1.1: Basic Boolean logic functions

Boolean logic values: Values with the two states equating to “TRUE” and “FALSE”; these are commonly abbreviated to “T” and “F” respectively.

Booth’s algorithm: A computer multiplication algorithm.

Break-out: A point where a single *net* enters or leaves a *bus*.

‘Bubble’: Small circle on component input or output indicating an inverted (or *active low*) signal.

Buffers: ‘Gates’ which perform no logic function but which increase the *drive* (and hence the *fanout*) of a signal. May also be used to provide *tristate* signals.

Buried register: A *register* in a *chip* whose outputs are not directly accessible from the ‘outside’.

Bus: a collection of *nets* usually with some similar function; for example a computer data bus may have thirty two wires (32 bits) which are used to encode a *binary* number. A bus is basically a means of keeping drawings tidy.
Buses are red in Powerview.

CAD: Computer Aided Design.

‘Chip’: Short for “silicon chip’; an *integrated circuit*.

Circuit diagram: See *schematic diagram*.

CLB: Configurable Logic Block; the basic component of a *Xilinx FPGA*.

Clock: A regular signal which is used as a time reference for other signal changes in *synchronous* circuits.

CMOS: Complementary Metal Oxide Semiconductor

Combinatorial logic: An implementation of *Boolean logic* where all the outputs depend solely on the current inputs. (see also *sequential logic*)

Component: that which is represented by a *symbol*. This is probably a whole hierarchy in itself.

Connector: a point on a *schematic* which is used to connect signals to other schematics.

Core: 1) The part of an integrated circuit which contains the functional *gates* (see also pad ring).
2) The main memory of a computer (historical).

Counter: a simple finite state machine which follows a repetitive pattern, usually cycling (upwards or downwards) through a set of N states (hence a "modulo N" counter).

Cycle: A complete operational period of a *clock*.

D latch: sometimes used for *transparent latch*.

D-type flip-flop: An edge-triggered storage device which has a single data input. The output adopts this state in response to an *active edge* of a *clock* signal.

Data selector: sometimes used for *multiplexer*.

Decade counter: A modulo 10 *counter*.

De Morgan's theorem: NOT(X AND Y) ≡ NOT(X) OR NOT(Y)
NOT(X OR Y) ≡ NOT(X) AND NOT(Y)

Dependency notation: A notation occasionally used for *schematic diagrams*. Its main characteristic is that signal names are given for the *active state* and the symbols indicate the particular active state of the signal.

Digital Signal Processor (DSP): A specialised computer or processing system used for processing signals (e.g. telephone systems, CD players).

Disable: To stop a function from occurring. Examples include: stopping data entering a latch; preventing a flip-flop from changing; and stopping data from reaching a tristate bus.

Drive: 1) The act of outputting a signal onto a net.
2) The capacity of an output to '*fanout*' to input loads.

DSP: See *Digital Signal Processor.*

Duty Cycle: the proportion of time a signal spends in its active state. Usually used to refer to very regular signals such as *clocks*. (See also *mark-space ratio*).

Edge: The change from one logic level to another (see also *transition*).

EEPROM: Electrically Erasable Programmable Read Only Memory (sometimes EEROM or "E^2ROM"; like an *EPROM* but erasure is performed electrically and therefore there is no need for a window in the package.

Enable: To allow a function to occur. Examples include: allowing data into a latch; permitting a flip-flop to change; and allowing data onto a tristate bus.

EPLD: Electrically Programmable Logic Device

EPROM: Erasable Programmable Read Only Memory; erasure is by exposure to ultra-violet (UV) light and hence the package will contain a quartz window to expose the chip.

Equivalent gate: alternative for *exclusive-NOR gate*. (See *Boolean logic*.)

Exclusive-OR (XOR): see *Boolean logic*.

Exclusive-NOR (XNOR): see *Boolean logic*.

Falling edge: A *transition* from a "H" to a "L". See also *rising edge*.

Fanout: The number of inputs driven (or which can be driven) by an output.

Feedback: Any system where (at least part of) the input is derived from the current *state* of the system.

Finite State Machine (FSM): A type of *sequential* circuit which employs *feedback* in order to follow a predefined set of *state* transitions, governed by its current state and (usually) some external inputs (fig. 1.2).

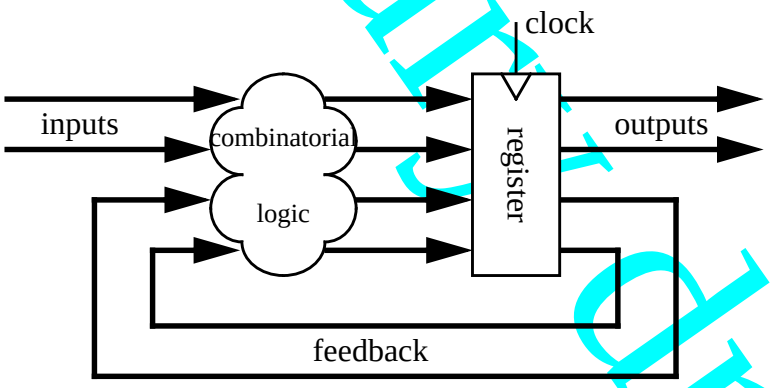


Figure 1.2: A typical Finite State Machine (FSM) structure

Flip-Flop: Any of a number of *bit* storage devices. Most commonly, but certainly not exclusively, a "D-type". See *D-type flip-flop*, *JK flip-flop*, *toggle flip-flop*, *RS flip-flop* and *transparent latch*.

FPGA: Field Programmable Gate Array. A large, user programmable circuit.

FSM: See *Finite State Machine*.

Gates: Simple circuits which provide basic *Boolean logic* functions.

Gate level: Usually the bottom level of a design hierarchy.

Glitch: A very short and unwanted pulse on a signal, typically caused by *race hazards* in networks of *gates*. Glitches are also known as “runt pulses”.

Glue: A slang term for the odd bits of logic used to hold *RTL* blocks together.

GND: Common abbreviation for *ground*. (see also *Vcc*, *Vdd*, *Vss*).

Ground: Zero volts. This is normally the ‘reference voltage’ from which other measurements are taken. In most (but not all) logic families this is a “0” or a logic low (“L”).

Hardware Description Language (HDL): A means of specifying the behaviour of a component in a form like a programming language rather than as a schematic diagram.

Hexadecimal (Hex): Base 16. Digits are represented as 0..9 then A..F. Hexadecimal is convenient as it is compact yet very easy to convert into *binary* (4 bits per digit).

HDL: See *Hardware Description Language*.

Hold time: The time data must remain stable after the *clock edge* for correct function of a *flip-flop*. See also *set-up time*.

IC: See *Integrated Circuit*.

Inactive: A signal which is indicating a ‘false’ condition.

Inactive edge: A *transition* of a signal (such as a *clock*) which cannot cause a circuit to change its *state*. See also *active edge*.

Instance: A single appearance of a symbol; for example an *AND gate* may be *instantiated* many times in a single design.

Integrated Circuit (IC): A ‘silicon *chip*’ containing from one to several million *gates*.

JK flip-flop: An edge-triggered storage device which has separate inputs allowing it to be set, cleared, toggled or not respond on a subsequent *clock* edge.

Karnaugh map: A graphical tool used for combinatorial logic minimisation.

Label: an identifier. Every *component* and *net* in a design will have a label, although most will be assigned automatically. A label can be attached by hand to identify a specific item. Note that a points in the same schematic with the same label will be connected, even if a net is not drawn between them. Labels are used to associate the pins of symbols with the correct signals in the schematic.

Labels are white in Powerview.

Latch: A *storage element*; often used as shorthand for '*transparent latch*' although sometimes used more generally.

Layout: The physical placement of *gates* on a *chip* or of chips on a *PCB*.

LCA: Logic Cell Array; A *Xilinx FPGA* design.

LED: see *Light Emitting Diode*.

Light Emitting Diode (LED): A form of diode which emits light when it is conducting electricity. LEDs are commonly used as indicator lights as they are simple, bright, low power devices. Available in various colours (typically red, amber, yellow and green, as well as infra-red) although blue LEDs are difficult to produce and therefore expensive.

Logic diagram: Sometimes used for *schematic diagram*.

Logic equations: A textual means of representing a combinatorial logic function.

Logic level: 1) A 'high' or a 'low' voltage.
2) The actual voltage values interpreted as 'high' or 'low'.

Mark-space ratio: The ratio of the time that a regular signal (such as a *clock*) is 'active' to the time that it is inactive. See also *duty cycle*.

Momentary Action: (usually referring to a switch) the action (output change) only occurs when there is some input stimulus (e.g. a push button).

Multiplexer: A circuit which allow one of a selection of signals or buses access to a single resource. Often abbreviated to "MUX".

Multiplexing: The act of using multiplexers.

MUX: Common shorthand for "*multiplexer*".

NAND: see *Boolean logic*.

Net: a set of wires connecting various *symbols*. A net may be a single simple connection or it may go to many different symbols.
Nets are red in Powerview.

NOR: see *Boolean logic*.

NOT: see *Boolean logic*.

Not-equivalent gate: alternative for *exclusive-OR gate*. (See *Boolean logic*.)

OR: see *Boolean logic*.

OTP: One Time Programmable.

Pad ring: The periphery of a *chip* which connects the core to the *pins*.

Patchboard: A circuit board which allows the rapid prototyping of circuits by simply plugging together components and wires (also known as a 'breadboard').

PCB: Printed Circuit Board. An stiff, insulating (often fibreglass) board with a pattern on conducting metal tracks etched onto its surface. Used for supporting and connecting electronic components. Easy to mass-produce.

Phase: 1)The relationship of two signals.
2)One half of a clock *cycle*.

Pin: 1) A place on a *symbol* where *nets* can be connected.
Pins are cyan (light blue) in Powerview.
2) A metal contact on the outside of an integrated circuit.

PLA: Programmable Logic Array. A customisable form of a *sum-of-products* structure.

'Place and Route': The act of choosing a position on a chip (circuit board etc.) for *gates* is known as 'placement'. 'Routing' is the subsequent connection of these components. The two operations are related. This process is often automatic nowadays. See also *layout*.

PLD: See *Programmable Logic Device*.

Product of Sums: A logic structure comprising an *AND* (product) *gate* whose inputs are the outputs of a set of *OR* (sum) *gates*. Analogous to - though more rarely encountered than - a *sum of products*.

Programmable Logic Device (PLD): A *chip* which may be customised (programmed) by the end-user.

Propagation delay: The delay between the inputs to a *gate*/circuit changing and its output changing in response.

Race hazard: A situation where signals which should arrive in a predetermined order are delayed by *gates*/wiring so that they may arrive in the wrong sequence. To be avoided! See also *skew*.

Register: A circuit component comprising a number of *flip-flops*.

Register Transfer Level (RTL): A 'level' in the design hierarchy which corresponds to the plugging together of blocks of about the complexity of a *register*. Usually one level above '*gate level*'.

Ripple Counters: An asynchronous counter where one counter output changing may cause the next counter in the chain to toggle by clocking it directly.

Rising edge: A *transition* from a "L" to a "H". See also *falling edge*.

ROM: Read Only Memory.

RS flip-flop: (sometimes SR flip-flop) a simple storage device which is Reset (to "0") or Set (to "1") by *active* pulses on the relevant input signal.

RTL: See *Register Transfer Level*.

Schematic: Shorthand for *schematic diagram*.

Schematic capture: A means of entering circuit components and wiring into a *CAD* system simply by drawing them on the screen as a *schematic diagram*. Also known as *schematic entry.*

Schematic diagram: a diagram showing the *components* of a circuit and their interconnections. Also known as a '*circuit diagram*' or just as a '*schematic*'.

Schematic entry: See *schematic capture*.

Sense: The sense of a signal determines whether it is *active high* or *active low.*

Sequential logic: A system constructed from *Boolean logic* elements where the outputs depend both on the current inputs and the past history of the system. The system therefore contains some *storage element*(*s*) which holds its current *state*. (See also *combinatorial logic*.)

Set-up time: The time data be remain stable before the *clock edge* for correct function of a *flip-flop*. See also *hold time*.

Sheet: a piece of (possibly electronic) paper upon which things are drawn. Only really relevant when a schematic is so large that it spreads across more than one sheet.

Shift register: A register which - in addition to normal register function - can also move each of its bits to an adjacent bit in response to the correct stimuli.

Simulation: Testing a circuit (or other) system without recourse to building it.

Skew: The time difference between two signals; may be caused by differences in *propagation delays*. See also *race hazard*.

SR flip-flop: see *RS flip-flop*.

State: The state of a circuit is effectively the pattern of values held within its internal storage elements; thus a modulo-8 counter would have eight possible states (usually encoded as combinations the individual states of three flip-flops. This is only applicable to sequential circuits; combinatorial circuits have no inherent state.

State diagram: A graphical means of representing the states of a system and the possible transitions between them (fig. 1.3).

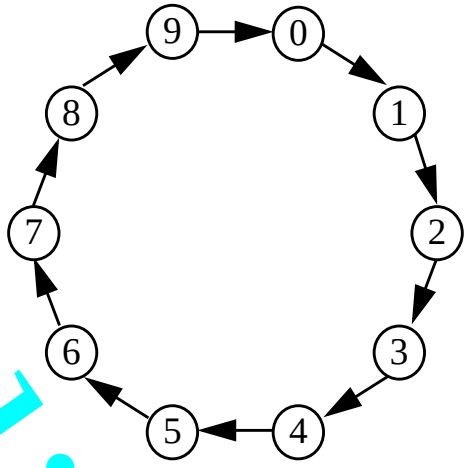


Figure 1.3: State Diagram for a Modulo-10 Up Counter

Storage element: Any circuit or component which contains *state* (e.g. a *latch*, *flip-flop* or memory).

Sum of Products: A logic structure comprising an *OR* (sum) *gate* whose inputs are the outputs of a set of *AND* (product) *gates*. A commonly encountered structure in *Programmable Logic Devices* (*PLD*s).

Symbol: a pictorial representation of a circuit component, used on a *schematic*. A symbol may represent something as simple as a *gate* or as complex as a microprocessor. It is typically bound to a schematic of the same name.

Synchronous: All signal changes are referenced to a global clock signal.

T-type flip-flop: occasionally used for *toggle flip-flop*.

Time-Domain Multiplexing (TDM): The sharing of a resource by allocating 'time-slices' to different clients.

Toggle: to move backwards and forwards between two states. Also used as an abbreviation for something which has this behaviour (e.g. a toggle switch or a *toggle flip-flop*).

Toggle Flip-Flop: A *flip-flop* which moves into the 'other' *state* every time it receives a *clock* input. Typically used as a "divide by two". (Occasionally referred to as a T-type flip-flop).

'Top-down' design: A design philosophy where the whole design is successively broken into a hierarchy of smaller elements until implementable blocks are obtained.

Transistor: An electronic switching/amplification component. Transistors are the fundamental components of almost logic *gates*.

Transition: 1) The change from one logic level to another (see also *edge*).
2) The change from one *state* to another.

Transition table: A graphical means of representing the behaviour of a sequential circuit; similar to a *truth table* but (some of) the outputs will form subsequent inputs.

Transparent latch: A level sensitive storage circuit which, when enabled passes its input(s) to its output(s) (i.e. is 'transparent').

Tristate output: A type of logic output which - in addition to the normal two states "H" and "L" ("0" & "1") - can adopt a third state, usually referred to as "Z" or "high impedance". This switches the output 'off' effectively disconnecting it from the circuit.

Truth table: A graphical means of representing a combinatorial logic function.

TTL: Transistor Transistor Logic. A bipolar logic family now becoming largely obsolete.

Unidirectional: A *net* (or, more usually, a *bus*) which has only one output connected to it; data can thus only 'flow' one way along the wires. e.g. a processor address bus. See also *bidirectional*.

Vcc: Typical label for a "1" or logic high ("H"). Derived from "collector voltage" in *TTL*; the name persists even though the derivation does not.
(See also *GND*, *Vdd*, *Vss*).

Vdd: Typical label for a "1" or logic high ("H"). Derived from "drain voltage" in *CMOS* logic. (See also *GND*, *Vcc*, *Vss*).

VHDL: *VHSIC* Hardware Description Language. A hardware description language is a means of specifying the behaviour of a component in a form like a programming language rather than as a schematic diagram.

VHSIC: Very High Speed Integrated Circuit; an US DoD research project.

Vss: Typical label for a "0" or logic low ("L"). Derived from "source voltage" in *CMOS* logic. (See also *GND*, *Vcc*, *Vdd*).

Voltage levels with the values 'high' ("H") voltage and 'low' ("L") voltage.

Xilinx: A US corporation who design and manufacture *FPGA*s.

XNOR: exclusive-NOR; see *Boolean logic*.

XOR: exclusive-OR; see *Boolean logic*.

ASCII CHARACTER SET

00	^@ NUL	Null	
01	^A SOH	Start of header	
02	^B STX	Start of text	
03	^C ETX	End of text	
04	^D EOT	End of transmission	
05	^E ENQ	Enquire	
06	^F ACK	Acknowledge Idle	
07	^G BEL	Bell block	
08	^H BS	Back space	
09	^I HT	Horizontal Tabulate	
0A	^J LF	Line feed	
0B	^K VT	Vertical Tabulate	
0C	^L FF	Form feed	
0D	^M CR	Carriage return	
0E	^N SO	Shift outtor	
0F	^O SI	Shift in	
10	^P DLE	Data link escape	
11	^Q DC1	Device control 1	
12	^R DC2	Device control 2	
13	^S DC3	Device control 3	
14	^T DC4	Device control 4	
15	^U NAK	Negative Acknowledge	
16	^V SYN	Synchronous Idle	
17	^W ETB	End of transmitted block	
18	^X CAN	Cancel	
19	^Y EM	End of medium	
1A	^Z SUB	Substitute	
1B	^[ESC	Escape	
1C	^ FS	File separator	
1D	^] GS	Group separator	
1E	^^ RS	Record separator	
1F	^_ US	Unit separator	

20	SP	Space	40	@	60	‘	
21	!		41	A	61	a	
22	"		42	B	62	b	
23	#		43	C	63	c	
24	$		44	D	64	d	
25	%		45	E	65	e	
26	&		46	F	66	f	
27	’		47	G	67	g	
28	(		48	H	68	h	
29	)		49	I	69	i	
2A	*		4A	J	6A	j	
2B	+		4B	K	6B	k	
2C	,		4C	L	6C	l	
2D	-		4D	M	6D	m	
2E	.		4E	N	6E	n	
2F	/		4F	O	6F	o	
30	0		50	P	70	p	
31	1		51	Q	71	q	
32	2		52	R	72	r	
33	3		53	S	73	s	
34	4		54	T	74	t	
35	5		55	U	75	u	
36	6		56	V	76	v	
37	7		57	W	77	w	
38	8		58	X	78	x	
39	9		59	Y	79	y	
3A	:		5A	Z	7A	z	
3B	;		5B	[	7B	{	
3C	<		5C		7C	|	
3D	=		5D	]	7D	}	
3E	>		5E	^	7E	~	
3F	?		5F	_	7F	DEL,RUBOUT	