Microelectronics Education Programme

MEP '3-CHIP PLUS' MICROPROCESSOR CONTROL SYSTEM

The '3-Chip' control system is designed for those who want to use a microprocessor system as a controller in CDT or science classes, or for demonstrating the principles of 'processing' with external gadgetry winking and whirring as a pupil motivator.

The system is designed to be as cheap as possible so that a class set of the controller board, which is the heart of the system, is expected to be affordable by schools prepared to devote an adequate budget to technology education. The controller board carries a microprocessor (6502), some 'scratchpad' RAM, a program memory and an input-output/timer chip (6522). These are the three fundamental ingredients of a microprocessor controller. 'Scratchpad' RAM is provided, which, together with a small number of other components, constitutes the "plus" in the title.

In order to make 'control' easy for a wide ability range, a variety of plug-on 'gadgets' are available for which all the volts and amps of interfacing are taken care of. One small board carries 8 switches and another 8 LEDs. A separate board carrying 4 changeover relays is available for switching external power to motors, heaters, lamps or whatever. Other separate boards are a tone generator (for playing tunes), a 4 digit display (for writing messages), a railway speed control board (forward/reverse + 3 speeds for a 12V loco), and a converter to enable the relay board to be used to control railway point motors. Additional input units include an analogue-digital converter, and a temperature transducer (which allow voltages and temperatures to be input to the system) and a sensor adaptor which allows four phototransistors (perhaps on the model railway track) to be connected as inputs.

The system was originally developed with MEP financial support by Peter Nicholls at Belper High School. Introducing machine code programming to young people of a wide range of ability has been well tried here (and other places) over a number of years (work has been going on since 1979). The key is supportive course materials which lead students through system usage into programming with a limited instruction set introduced at a carefully controlled rate. If this is done, all but the weakest (i.e. those unlikely to be graded in a CSE exam) can gain an 'awareness' of microprocessor and control by operating the system, plugging units together, keying in code or using pre-programmed EPROMs. 'Average' students can usually cope with being asked to make a small change to a few bytes of code in order to modify the behaviour of the system in some small way; and as one would expect, only 'able' students can cope with open-ended design situations where the specification of the control system has to be drawn up and then the program written to do the job. In other words, high level language is not the only route to an experience of computer control for school students. The cost advantage of machine-code controllers is high, as is their portability and low power consumption. Perhaps this is the only cost effective route to hands-on control work for whole classes at a time.

Some facility must exist within the school for 'putting' the program into a ROM of some sort. Part of the 3-CHIP system is a non-volatile-RAM loader which enables program code to be keyed into the program chip. This loader will also blow EPROMs. (If schools have other EPROM blowing facilities there is nothing to stop these being used).

A final facility of the system is its ability to "mimic" the BBC computer. The address map of the controller is the same as that of the BBC micro, so that assembler on the Beeb can be used to develop code, the computer can test-run it, and when finished it can be transferred to EPROM or non-volatile RAM. This facility may be attractive to whizz-kids who find that hand-assembly of machine-code slows up their creativity, and to teachers who have dabbled with control in BASIC (perhaps on a Beeb), but who as yet lack the confidence to dive into the machine-code pool. To mix the metaphors horribly, the assembler route offers a stepping stone into machine-code using equipment one largely has already.

The controller, RAM loader, and the range of peripheral boards described above should be available from Messrs. UNILAB LTD. at the beginning of 1985. UNILAB also manufacture the MEP 'MFA' equipment: all the peripherals for 3-CHIP are designed to be both electrically and mechanically compatible with MFA boards and the MFA computer interface. Further input and output boards have been prototyped and are likely to appear later, as is a 3-CHIP board designed around a Z-80 microprocessor - this is a dynamic range that will grow to both meet and stimulate need. It is intended that a distance-learning pack for teachers be published by the end of 1985, to coincide with general hardware availability. Workcards for use as an extension module to 'MFA' will be published in due course. Work is also underway on the writing of courseware for use in CDT courses including those based on modular technology exams and the expected new ABE microprocessor option to Control Technology.

Further details after January 1985 available from:

Unilab Limited, Clarendon Road, Blackburn, BB1 9TA
Telephone: (0254) 57643/4

C O N T E N T S

E R R A T A N O T E

Page	
2	Last para. line 10: Please read **1984** NOT 1985
	Last para. last line Please read **AEB** NOT ABE
19	For "memory module" please read "**music module**"
33	Under Assembler mnemonics please read **LDA HHLL**

3-Chip Plus Microcontroller System - Teaching Philosophy

This INSET package on microprocessor control is very similar to a 4th/5th year teaching module written for a CSE (all-ability) electronics course. The pace of the worksheets is somewhat faster than all but grade 1 students could cope with, and certainly the series of sheets would take longer than the 15 hours of CSE course time originally allocated to microprocessor work, but with these qualifications these concepts could be learnt in the classroom without modification.

The medium is the message This phrase of McLuhan's may be misappropriated again to indicate a profound truth about educational technology. This is that our students are likely to 'learn' a great deal more from the circumstances of their learning situation than from the actual words and detailed subject content that we as teachers seek to impart. 'Learning' - particularly in the ultra-rapidly changing sphere of electronics - must be about attitudes, skills, concepts and understanding. The hardware for this package has been developed with this overall educational aim in mind, rather than to satisfy the narrow objective of teaching students to write control software in machine-code.

The 'microprocessor revolution' will have various facets, one major area of micro use is control. Literally billions of microprocessors will be (are already?) in use - a **minority** in what would conventionally be described as a computer. Control applications in which the microprocessor and its accompanying program ('software') are dedicated to a particular task outnumber 'computing' applications. These control applications range from those in domestic equipment (washing machines, food mixers, central heating boilers, sewing machines ....) through control and monitoring systems in cars to industrial and manufacturing uses (including robots and CNC machine tools). The '3-chip board' which is the heart of this teaching package is in front of the student all the time and is used to perform a wide variety of tasks in conjunction with the range of add-on modules. In this way the simplicity, versatility, power and cheapness of the dedicated microcontroller which lies behind the coming 'second industrial revolution' is demonstrated without ever being explicitly stated.

Conceptual development

The worksheets introduce new concepts at a controlled rate and descriptions are full enough for the material to be used as a distance-learning package. This may mean that students who have some familiarity with microelectronics and its associated jargon will prefer to skim certain sections. No apology is made for the fact that some sheets are 'cookery'. The purpose is to give familiarity in handling the equipment and a glimpse of what the system can do before beginning to unravel the machine code. Curious students are advised to proceed through the sheets trusting that all programming ideas that are introduced apparently unexplained will be unravelled in the following sheets.

An overview of the written material is:

T1-2	Introduction to the Controller and A-ROM Loader
T3	Machine code and hexadecimal numbers
T4	Contact with the real world - ports
T5-T9	Control of Music Module, display, railway, buggy with ne machine code instructions and ideas introduced progressively
T10	Inputting data to the Controller
T11	Feedback of position from the railway
T12	Analogue inputs (voltage and temperature)
T13	Using RAM
T14	Processing of data, and control of temperature as an example of an analogue quantity.

Inevitably there are concepts that are not dealt with. The full range of facilities on the A-ROM Loader (which can also burn programs semi-permanently into EPROMs) are described in literature supplied with the hardware. If you have the burner board, try transferring a program to an EPROM or EEPROM.

The technique of drawing a flowchart is covered in the next collection of ideas ('Project Possibilites'). The concept of interrupt handling is left out of this material, as is use of the 6522 timers and other facilities, and software techniques like using look-up tables. The increasingly common mechanical device the stepper motor is squeezed into the next text. As with any subject, the tidy desire for 'completeness' needs to be kept in check. This collection of thoughts aims to provide the newcomer t control and 3-Chip Plus with a range of methods for implementing basic control functions. The overview of the material reveals tha the fundamental concepts of digital input and output, with conversion to and from an analogue voltage are covered as are sequenced control and control with feedback. The essential pictur is completed by a demonstration that microprocessors can process data, change its format and make decisions.

The problem-solving thread in these sheets is sometimes less perceptible than I would wish. The optimum structure of

* show a new idea in operation
* describe and explain it
* give further example(s)
* **set an applications problem**
* provide answers on follow-up sheet

has had to be modified somewhat in places simply to get through, with a finite amount of text, in a distance-learning environment. There is quite a lot of content to be imparted before a teacher ca feel on top of microprocessor control. With school students I would write terser learning sheets with frequent simple problems t solve, and use a good deal of verbal explanation both in groups an one-to-one as the need arose.

Peter Nicholls
December 1984

T1 Using the '3-Chip Plus' controller board

Fetch a 3-Chip Plus microcontroller. This board is the heart of everything that you will be doing. It is similar to a great many of the simpler microprocessor control systems that are already in use in industry, and in many bits of domestic equipment already on sale. The ingredients of any simple microsystem are:

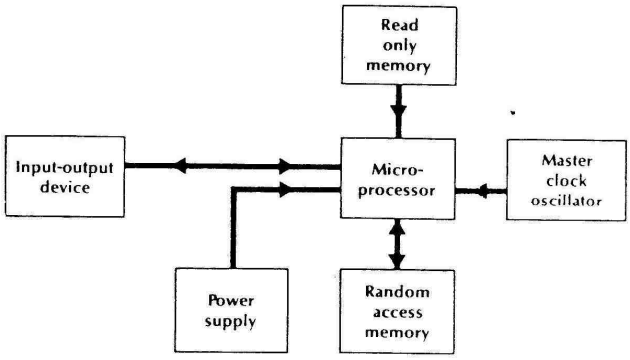


Fig. 1

On our boards the Input-Output chip is the large 6522 at the end. The microprocessor is the 6502 in the middle. The program which directs the microprocessor is stored in a Read Only Memory which clips into the black socket. The function of these components will become clearer as you go on. Meanwhile let's get a program running.

* * * * * * * *

* Fetch a 5V power supply
an 8-LED Unit
a Music Module
an 8-Switch Unit
a demonstration program (read only) memory chip labelled 'DEMO'

Any microprocessor system must have a program stored in a memory chip. The program is as essential as the processor. Put another way, the 3-Chip Plus microcontroller you have in front of you is dead until you plug into it a memory chip containing a program.

* Plug the program memory chip into the black socket:

lift the lever up
insert the chip - white mark towards the white mark on the socket
press the lever down

* Connect the 8-LED Unit to the right hand connector ("Port A") and the 8-Switch Unit to the left hand connector ("Port B")

* The memory chip has eight different programs stored in it. They are all very short and control whatever is connected to Port A of the 3-Chip Plus board. The program is selected by the switch settings on the 8-Switch Unit.

* Put all switches except the right hand one down on the 8-Switch Unit (so switch 0 only should be up). Connect the power supply to the board - the LEDs should flash on and off in blocks of four. If nothing seems to happen, press the RESET button on the board; if this does not produce results, disconnect the power and read this sheet from the beginning, checking your connections carefully (especially that the memory chip is inserted correctly and clamped).

* Switch Switch 0 back down and put up Switch 1. Press the RESET button and watch the LEDs - you should now have a 'chaser' display.

* Only Switch 2 up (put Switch 1 down again) will 'chase' the LEDs in the opposite direction. Only Switch 3 up will give you traffic lights - always remember to press RESET after selecting a new program.

* Now plug the Music Module into the connector on the right of the 8-LED Unit. Switches 4, 5 and 6 will give three different tunes (as usual press RESET after selecting the program). You can also watch the LEDs showing which control lines are on and which off, and so how different notes are selected.

* Finally, put Switch 7 up and all the rest down. Press RESET. The Switches on the 8-Switch Unit can now be used to select which note is being played.

* * * * * * * *

This sheet is to guide you to the point of familiarity with the Controller Board. It should be clear that the Board's behaviour can be changed easily by changing the program. It should be clear that the program is stored in an integrated circuit ('chip') which is plugged into the black socket. It should also be clear that the Board can control various external gadgets - you used an 8-LED Unit and Music Module for this sheet's work.

On sheet T2 the equipment for 'building' a program will be met - DEMO memory chips won't exist for everything you want to do!

Next sheet - T2

T2 Loading the Program into a Memory

The work on the last sheet was meant to show you a simple microcontroller system in operation. You put a memory chip into the socket on the microcontroller board and there were 8 programs available to you for flashing LEDs and playing tunes. The proper name for the memory chip you used is Erasible Programmable Read Only Memory and these are very widely used in microsystems for storing the program which controls the microprocessor

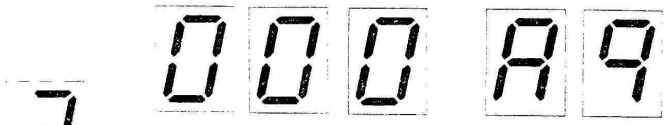
A Read Only Memory is one which you can only 'read' from (you can't write things into it). Programmable means that you can put your program into it if you have the right equipment. And it can be erased by shining ultra-violet light through the window at the top.

EPROM's are very useful for building microsystems. If you are trying to design a program you can 'burn' your first try at the program into a 'clean' EPROM using special equipment, then connect the EPROM to the microsystem to test it. The trouble is that if your first program doesn't work, you have to erase the whole EPROM (which takes about half an hour) and then reprogram it with your improved program. This is a slow business - especially since you may only want to change one instruction in the program.

For developing programs there are better ways. Fetch an A-ROM loader and an A-ROM - the 'better way' that we shall use. A-ROM stands for Alterable Read Only Memory - it's a special device into which you can load your program, and alter it if you need to. (By the way, 'A-ROM' is a bit of jargon invented for these learning sheets; it's not common currency.)

* Clamp the A-ROM into the loader (the right way round) and switch on.

* The display should show something like:



This display will show a 'prompt' character which tells you what the loader is doing.

the address of the memory 'cell' at present being displayed

the instruction code or data stored in that address. Your A-ROM could have anything there - it won't necessarily be A9

(A note in brackets here. You may be worried by 'numbers' like 'A'! Don't worry. A to F are codes with a special meaning which will be explained later. For the moment accept that there are sixteen 'characters' which can be used: 0 - 9 and A - F.)

* When the **prompt** is showing **'a'** the keypad can be used to enter an address. Press **245** and see what is stored at that address. What is in 6CE? The A-ROM has 2,048 stores in it. They have numbered addresses from 000 to 7FF.

To put something into a store, first press the key **d** on the bottom row - the prompt in the display will change to **d**. The keypad will now change the right hand pair of display digits, and these numbers will be stored into the A-ROM

* Press the a key, then 123
  Press the d key, then 45
  This will store the number 45 in the store whose address is 123.

* Press the **+1** key. The display will move on to address 124.
  Press keys **67** and the number 67 will be entered into the store at that address.
  Press +1 again and enter **89** (into store 125)

* Check that the codes in 123 and 124 have been held - you can do this by pressing the -1 button which will take you back first to 124 and then to 123.

* Switch off the power, and then switch on again. Press keys a, 1, 2, 3, and check on the display that the code 45 is still stored there. Press +1 and check the contents of 124 and 125.

  You should find that disconnecting the power did <u>not</u> mess up the memory.

* Make a note of what the control keys a, d, +1 and -1 do. Note that these worksheets will use **a** to mean the 'address' command key and **A** to mean the number digit key.

o o o 0 0 0 0 0 0 0 o o o

So much for how to operate the kit. Now use it to load a program:

Press **a**, then **000**, then **d**, **A9**
(this will put the code A9 in the memory cell whose address is 000).
Press **+1** then **FF** (which will be put into location 001)
Continue to press +1 to get the next store, and enter the program, instruction by instruction, as follows:

002 8D
003 63
004 FE
005 A9
006 F0
007 8D
008 61
009 FE

```
00A  A0
00B  0A
00C  20
00D  80
00E  27
00F  A9
010  0F
011  8D
012  61
013  FE
014  A0
015  0A
016  20
017  80
018  27
019  4C
01A  05
01B  20
```

Now press a and 780, then d to allow you to enter the code:

```
780  A9
781  00
782  8D
783  6B
784  FE
785  8D
786  6E
787  FE
788  A9
789  43
78A  8D
78B  64
78C  FE
78D  A9
78E  C3
78F  8D
790  65
791  FE
792  AD
793  6D
794  FE
795  F0
796  FB
797  88
798  D0
799  F3
79A  60
```

Finally load 7FC 00 (press a, 7FC, d, 00
7FD 20 press +1, 20)

Now you know what machine code is! Soon all will be revealed; the purpose of this sheet is merely to demonstrate the equipment and to give you a feel for memory cells, their addresses, and that you can store codes in them.

* Remove the A-ROM from the loader and clamp it into the 3-Chip Plus board. Plug the 8-LED unit into Port A and switch on. The LEDs should start flashing in blocks of four - try pressing RESET if they don't.

  If this works you have successfully completed the exercise of loading a program and running it. If it won't work you have probably miskeyed somewhere. In which case, put the A-ROM back into the loader, press **a 000**, then step the memory using the +1 key (as far as location 01B) checking the contents of each cell as you go. Repeat for 780 to 79A (press **a 780** to get to the right block of memory, then step through with the +1 key), and lastly check 7FC and 7FD. There should be no difficulty in getting the program to run once it is correctly entered into the A-ROM.

Next Sheet - T3

T3 Now for the Machine Code

Some users of this learning material may have heard the jargon 'machine code' in computer conversation. It is often thought to be the most esoteric and difficult aspect of computer programming - strictly for the buffs. Yet for control applications it is usually very straightforward, as this sheet will attempt to show.

Sheet T2 had a string of machine code instructions which you had to enter into the program memory (the A-ROM). Step 1 towards getting into it is to rewrite it in groups of codes - like this:

```
000    A9 FF
002    8D 63 FE
```

The A9 (in cell 000) and the FF (in cell 001) belong together - the microprocessor makes sense of them as a pair. So it is more sensible to write them as a pair. Similarly the 8D (in cell 002), the 63 (in cell 003) and the FE (in cell 004) only make sense together - so are written as a group on one line. Notice that the address of the 'FF' is not shown - one just assumes that it is in the cell after 000, i.e. 001. But remember that the next cell in the program listing is address 002 (not 001) because 001 has already been used. The address on the next line in the list would be ..... 005 (003 and 4 are already used).

The whole of the first chunk of program written out in this way looks like this - check it against the listing on sheet T2; see that you can make sense of it:

```
000   A9 FF
002   8D 63 FE
005   A9 F0
007   8D 61 FE
00A   A0 0A
00C   20 80 27
00F   A9 0F
011   8D 61 FE
014   A0 0A
016   20 80 27
019   4C 05 20
```

This way of writing reveals a good deal more order in the code than might have been apparent at first sight. Let's try the effect of one or two changes.

* Change the contents of cell 006 to C0, so now you have

```
005   A9 C0
007   8D 61  FE
      etc.
```

(put the A-ROM in the loader, press **a 006 d C0**)

Page 10 missing

the 16 x 16's column

two hundred & fifty six's	sixteens	units	
2	2	9	means the same as 512 + 32 + 9 = 553
1	6	2	means the same as 256 + 96 + 2 = 354
3	3	6	is equal to what? (write down your answer)
0	7	4	is equal to what? (write down your answer)
4	1	15	is equal to what? (write down your answer)

that was a surprise maybe - that we can write fifteen in the column. But how else are you going to count from zero upwards? you can't put a one in the sixteens column until you get up to sixteen.

it will cause a lot of confusion if we write '15' in the column, so we invent a few new 'numerals':

Decimal	Hex.
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Note down this list.

To make clear when we are working in hex. it is usual to put some symbol with the number. Conventions vary - you may meet $A9, &A9 or A9(H) all of which tell you that A9 is a hex number. The dollar sign is probably the most common international symbol, but the BBC microcomputer uses the ampersand sign (&) to denote a hex number, so this convention has become common too.

To avoid confusion (and save ink!) these worksheets will use no symbol - just about every number on them is in hex anyway. Remember that **from now on everything is in hex unless stated otherwise.**

To firm this up in your mind, write down the decimal (ordinary, or tens counting) values of the following hex numbers:

(4) 25 (5) 3A (6) 4D (7) A5 (8) DD

Next sheet - T3(F)

T3(F) Follow-up to 'Now for the Machine Code'

1. Hexadecimal 336 = eight hundred & twenty two
2. Hexadecimal 074 = one hundred & sixteen
3. Hexadecimal 41(15) = one hundred and fifty five
4. 25 = thirty seven
5. 3A = fifty eight
6. 4D = seventy seven
7. A5 = one hundred & sixty five
8. DD = two hundred & twenty one

Next sheet - T4

To Switching Devices On and Off

An understanding of what's in a microprocessor will help you to understand how a program is constructed. You won't need to know everything - in fact one or two key facts will go a long way.

The microprocessor that we use (which is typical of most of them) has 40 pins. 16 of them connect to the program memory (the EPROM) and are used to select the address of the memory cell being looked at. These 16 pins are called the address bus.

8 pins receive back from the program memory the code or data that is stored in the cell that has been selected by the address bus. These 8 pins are called the data bus. Diagram 1 shows the idea.

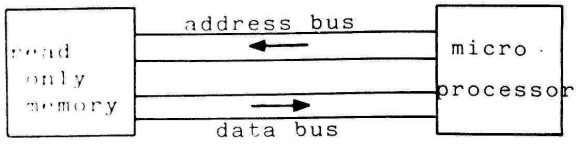


Diagram 1

That leaves 16 pins unexplained! Some of them are used as control lines, one of which, for instance, is used to tell the program memory when to put its data onto the data bus. A couple of pins go to the power supply and two to the crystal clock that keeps time accurately.

When the RESET key is pressed on the 3-Chip Plus board, the microprocessor starts at address 000 and fetches the contents of that cell. The 'processor can fetch codes from following addresses in a similar way.

Let's look again at the previous program (on sheet T3):

Address	Instruction Code	Other Information		Meaning
000	A9	FF		A9 = load into the micro-processor 'heart' a number FF = the number to be loaded
002	8D	63	FE	8D = store the contents of the microprocessor 'heart' to a memory cell FE63 = the address of the memory cell
005	A9	F0		A9 = load into the micro-processor 'heart' a number F0 = the number to be loaded
007	8D	61	FE	8D = store the contents of the microprocessor 'heart' to a memory cell. FE61 = the address of the memory cell

and so on . . .

Once you see the structure it should seem straightforward and logical. One final point before going on: notice that on the 3-Chip Plus board, "Port A" (the connector into which you plugged 8-LED Unit) is said to have **address FE61.** The ports are memory cells into which you can store data. Voltages appear on the pi of the port 'chip' (the 6522 in our case) - these voltages are controlled by the data that has been stored in the memory cell the port address.

The machine code in 005 - 009 stores the hex code F0 in port A' memory cell. The code F0 will switch on the four left-hand LED Try to crack the code relationship - what code turns on what LE

* Using the shell of the program given on sheet T2, use the A-ROM loader to put 01 into address 006
  and 00 into address 010
  Run the program in the 3-Chip Plus board - which LED flash

* Change 006 to 02; which LED flashes now?

* Try 04 and then 08 in cell 006,

* Since 04 + 02 + 01 = 07, what would you expect if 07 is in cell 006? Try it!

* Any guesses for what 10, 20, 40 and 80 will do to the outp lines if put in cell 006 of the program?

Control of something bigger

Switching LEDs on and off is only the beginning. The control voltages on the eight lines of a port can be used to control almost anything.

- Plug the RELAY Unit into the connector on the 8-LED Unit (the 8-LED Unit doesn't have to be there, it just helps you to keep watch on the control lines). If you have a motor and a suitable power supply, connect them like this:

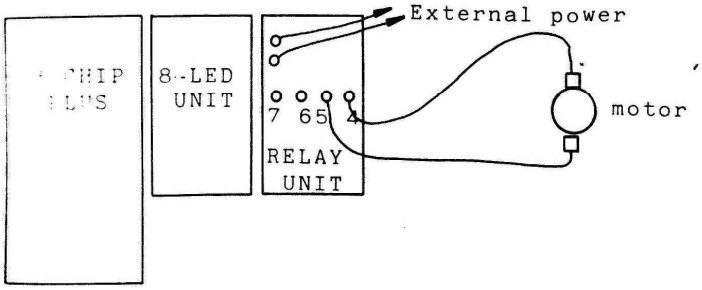


Diagram 2.

If Port A line 4 is "high" or at "logic 1" and
line 5 is "low" or at "logic 0"
the relays will be like this:

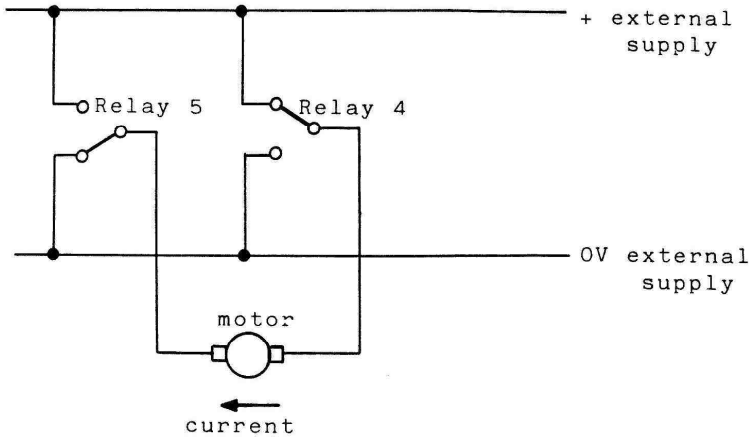


Diagram 3

You can set up this situation by putting number 10 into the program at 006 and at 010 (putting it into both these addresses stops the flashing). Note which way the motor rotates.

* If Port A line 4 is "low" or at "logic 0" and
line 5 is "high" or at "logic 1"
the relays will be like this:

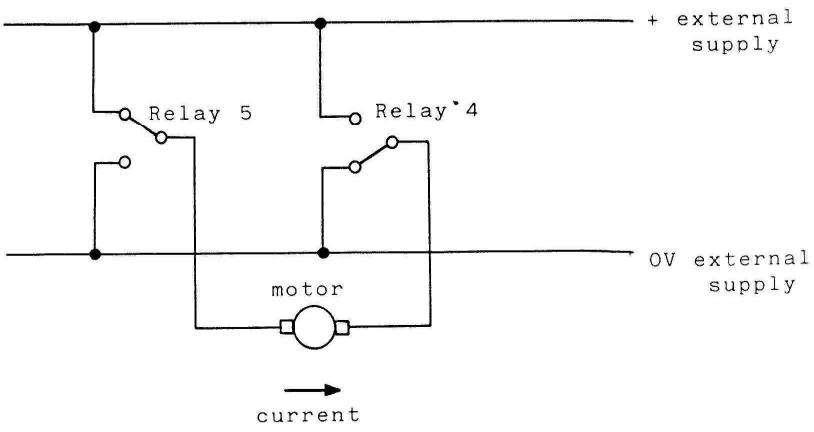


Diagram 4

This can be set up putting 20 into the program at 006 and at 010. Try it; check that the motor rotates in the opposite direction. Why does sending 20 to Port A switch the relays into the position shown?

* Finally, there are <u>two</u> possible ways of stopping the motor. Can you work out:

(a) what position relays 4 and 5 need to be in for the motor to stop?

(b) what number to put into locations 006 and 010 to get this to happen?

The answers to these questions are on the follow-up sheet.

Next Sheet - T4(F)

Page 17 missing

The relay work extended this idea. Relays 5 and 4 are connected to lines A5 and A4 (port A line 5, port A line 4) respectively. For the connections in Fig 1 the bit pattern is 0001 0000 which needs 10 in hex to be sent to port A. Fig 2 requires 0010 0000 which is 20 in hex. One code rotates the motor clockwise, the other counter- clockwise.

To stop the motor, either 0000 0000 (hex 00) or 0011 0000 (hex 30) can be used.

If you connected two motors, you would have needed to use two more bits, so, for example, 1001 0000 (hex 90) would rotate one motor clockwise and the other counter-clockwise.

Next worksheet - T5 but if a Music Module is not available you could go on to T6 or T7.

T9 Playing Tunes (Notes and Delays).

The 'MFA' Memory Module can be plugged onto the 3-Chip Plus controller and long and complicated tunes played. The Memory Module is controlled by the port, each note has a hex code, thus:

hex 00 - silence
01 - middle C
02 - C#
03 - D
04 - D#
05 - E
06 - F
07 - F#
08 - G
09 - G#
0A - A
0B - A# (B flat)
0C - B
0D - top C
0E - top C#
0F - top D

To get the feel of it, put two different codes (you choose) into addresses 006 and 010. Transfer the A-ROM into the Controller and run the program. You should get a two-tone siren (the notes of your choice). If you have problems, check the contents of program memory cells 780 - 79A, and 7FC & D.

To play tunes you need to learn how to delay (kill time!) for a selected amount of time. Try this first (load the program into the A-ROM, then run it):

```
000   A9 FF        The "port set-up routine"
002   8D 63 FE     (explained on Sheet T9)
005   A9 note      Send note code to port A
007   8D 61 FE
00A   A0 delay     delay
00C   20 80 27
00F   A9 note      send note code to port A
011   8D 61 FE
014   A0 delay     delay
016   20 80 27
019   A9 note      note to port A
01B   8D 61 FE
01E   A0 delay     delay
020   20 80 27
023   A9 note      note to port A
025   8D 61 FE
028   A0 delay     delay
02A   20 80 27
02D   4C 05 20     end of program
```

Select your own note codes for the cells labelled 'note', and the delays. Each 'blink' of delay in this case is 50ms (a twentieth of a second), so to sound a note for one second you need twenty 'blinks'. Remember, though, that all the numbers in machine code are hex. To get twenty blinks, use hex 14 (which is twenty in decimal counting).

Now let's try to play a proper tune. The MFA workcards use 'Morning has Broken' as an example - the first few notes are this:

Note	Note code	Time (ms)	'Blinks' needed	Blinks in hex
C	01	400 ms	8	08
E	05	400 ms	8	08
G	08	400 ms	8	08
C	0D	1200 ms	24	18
D		1200 ms		
B		600 ms		
A		200 ms		
G		400 ms		
A		600 ms		
B		200 ms		
A		400 ms		
G		1200 ms		

The first four lines of the table are completed - you can work out the remaining codes and delays yourself.

By repeatedly using the program block:

```
A9 note
8D 61 FE
A0 delay
20 80 27
```

you can write a program to play this much of the tune. Don't forget to start your program with the 'set-up routine' in addresses 000 - 004, and to finish it with 4C 00 20 as in the example program at the start of this sheet. Load the program into your A-ROM. Check that the code you previously put in 780 - 79A is still correct, and that 7FC and 7FD are 00 and 20 respectively.

Transfer the A-ROM to the 3 Chip-Plus board and listen expectantly! (press RESET if necessary). The correct code is given on sheet T5(F) if all else fails.

o o o o 0 0 0 0 o o o o

The machine code is very repetitive. It's worth building up a 'dictionary' of instructions to refer to - start now on a clean sheet with these headings:

Code	Assembler mnemonic	Name	Description
A9 NN	LDA#NN	Load accumulator immediate	Loads the microprocessor 'accumulator' with the next number in the program - NN is meant to indicate the next number.
8D LL HH	STA HHLL	Store	Stores the contents of the microprocessor accumulator to the address HHLL. Notice that to send to this address the code is 8D LL HH - the halves of the address are entered in the opposite order.
A0 NN	LDY#NN	Load Y register immediate	Loads the microprocessor 'Y Register' with the next number in the program - NN is meant to indicate the next number.
20 LL HH	JSR HHLL	Jump to sub-routine which starts at address HHLL	When this instruction is encountered, the micro-processor jumps to the address HHLL & begins to execute the machine code which is stored in that program memory cell and those following it.
4C LL HH	JMP HHLL	Jump to new location	When this instruction is encountered, the micro-processor jumps to the address HHLL & begins to execute the machine code which is stored in that program memory cell and those following it.
60	RTS	Return from sub-routine	The microprocessor goes to the point in the main program where it left off to perform the sub-routine (i.e. the point where the JSR instruction was met).

That table introduced some new terms, and concepts, which need more explaining. The microprocessor has inside it a vital 'heart' called its **accumulator**. Actually it's more like a stomach because it's where most of the digesting of numbers and data goes on. It's a store where one hex number can be held and operated on.

The other name for this sort of special store in a microprocessor is **register** - the accumulator is one register in the microprocessor. There are two other special registers in the 6502 microprocessor called the **X register** and the **Y register**. They have a number of functions, but in our program the Y register was used as a temporary store for the delay length.

'**Immediate**' is a jargon word which means 'a specified number'. Just 'load' would mean that a register was being loaded with the contents of a memory cell; 'load immediate' means load the register with the number following.

A **sub-routine** is a self-contained piece of program which the microprocessor jumps to and returns from as necessary. In our case the delay program is needed a number of times. Imagine how tedious (and wasteful of memory space) it would be to have to put in the delay program in full again and again. Instead, a 'JSR' instruction is used followed by the address at which the sub-routine starts (in our case 2780) - note that the addresses get entered 'backwards', the 80 before the 27 in this case.

You might wonder how the program gets back to where it left off after the sub-routine finishes. The final instruction is every sub-routine must be 60 - return from sub-routine. This jumps the microprocessor back to the right point in the program.

Finally, the **jump** instruction tells the microprocessor to go to a new address and start executing the code from there. so 4C 05 20 will jump the program to (2)005 which repeats the sending-notes-to-the-port program.

Now that you know what some of the instructions are, have another look at the program given near the start of this sheet and try to understand what's going on.

Next Sheet - T5(F)

T5(F) Follow-up to Notes & Delays

The complete table for 'Morning has Broken' looks like this:

Note	Note Code	Time (ms)	'Blinks' needed	'Blinks' in hex
C	01	400 ms	8	08
E	05	400 ms	8	08
G	08	400 ms	8	08
C	0D	1200 ms	24	18
D	0F	1200 ms	24	18
B	0C	600 ms	12	0C
A	0A	200 ms	4	04
G	08	400 ms	8	08
A	0A	600 ms	12	0C
B	0C	200 ms	4	04
A	0A	400 ms	8	08
G	08	1200 ms	24	18

The complete program is:

```
000   A9 FF
002   8D 63 FE
005   A9 01
007   8D 61 FE
00A   A0 08
00C   20 80 27
00F   A9 05
011   8D 61 FE
014   A0 08
016   20 80 27
019   A9 08
01B   8D 61 FE
01E   A0 08
020   20 80 27
023   A9 0D
025   8D 61 FE
028   A0 18
02A   20 80 27
02D   A9 0F
02F   8D 61 FE
032   A0 18
034   20 80 27
037   A9 0C
039   8D 61 FE
03C   A0 0C
03E   20 80 27
041   A9 0A
043   8D 61 FE
046   A0 04
048   20 80 27
04B   A9 08
04D   8D 61 FE
```

```
050   A0 08
052   20 80 27
055   A9 0A
057   8D 61 FE
05A   A0 0C
05C   20 80 27
05F   A9 0C
061   8D 61 FE
064   A0 04
066   20 80 27
069   A9 0A
06B   8D 61 FE
06E   A0 08
070   20 80 27
073   A9 08
075   8D 61 FE
078   A0 18
07A   20 80 27
07D   4C 05 20
```

with 780 - 79A and 7FC - 7FD as before

o o o o 0 0 0 0 o o o o

Your're right - that was tedious! There are better ways than this of doing repetitive tasks, but we must learn to walk before trying to run. The **sub-routine** introduced on sheet T5 is one vital way. Putting all the codes and delays into a table is another way. You might like to try this more elegant program for the whole 'Morning has Broken' tune. Put it into the A-ROM from 100 upwards:

```
100   A9 FF
102   8D 63 FE
105   A2 00
107   BD 80 21
10A   C9 00
10C   F0 F7
10E   29 0F
110   8D 61 FE
113   BD 80 21
116   29 F0
118   A8
119   20 80 27
11C   E8
11D   4C 07 21
```

Store the tune from 180 upwards:

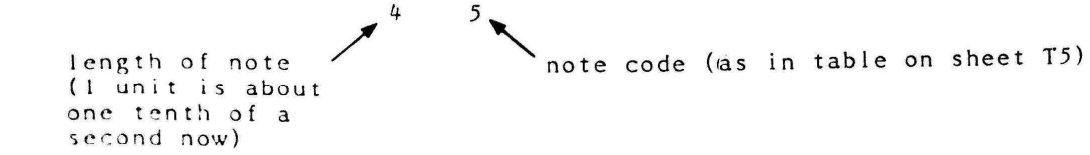
180	41	196	48
181	45	197	45
182	48	198	48
183	CD	199	CD
184	CF	19A	CA
185	6C	19B	48
186	2A	19C	45
187	48	19D	41
188	6A	19E	10
189	2C	19F	A1
18A	4A	1A0	C3
18B	C8	1A1	45
18C	41	1A2	43
18D	43	1A3	45
18E	45	1A4	C8
18F	C8	1A5	CA
190	CA	1A6	43
191	48	1A7	65
192	45	1A8	23
193	41	1A9	C1
194	C3	1AA	00
195	C3		

Then make two other small changes:

(a) reduce the delay by changing the contents of 78E to 18 (it was C3)

(b) tell the 3 Chip microprocessor to run the program put in 100 upwards by changing the contents of 7FD to 21 (it was 20).

Put the A-ROM into the 3-Chip Plus board and switch on - you should get a full version of 'Morning has Broken'! This time, the actual program is from 100 - 11F (only 32 locations), although it does depend on the delay routine in 780 - 79A. This is much more economical. Also the tune is easily changed or modified. You can put your own tune from 180 upwards - it doesn't matter how long or short it is so long as you put 00 at the end to tell the microprocessor where the end is. The coding is:



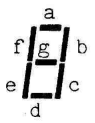
This new program uses a number of 'tricks' which will be explained bit by bit as you progress through these sheets; it is deliberately not being explained at this stage.

Next Sheet - T6 or T7, whichever is free and you haven't done. If you have done both of these, move on to sheet T8.

T6 7-Segment Display

The ports on the 3-Chip Plus board may be used to drive a 7-segment display. In our case we shall use seven lines of port A to control the segments, and four lines of port B to control the digits. To turn on a segment you store '1' to the line which controls that segment, and at the same time '1' to the digit that you want to turn on.

PORT A (address FE61) - g e f b c d a



PORT B (address FE60) - - - - z y x w

This won't yet be very clear so let's try one or two things:

First, the port set-up routine

```
000   A9 FF
002   8D 63 FE
005   8D 62 FE
```

Now, to turn on segment g of digit X we need to send

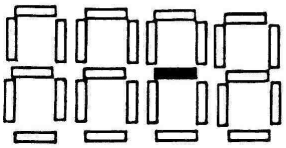
```
                  008   A9 40
    40 to port A with   00A   8D 61 FE

                  00D   A9 02
and 02 to port B with   00F   8D 60 FE

                  012   4C 12 20
```

Load this into the A-ROM. Connect the display to Port A and Port B on the 3-Chip Plus board. Transfer the A-ROM to the 3-Chip Plus board press RESET and the display should show:

- that is, segment g of digit X is on.



* Change the contents of 009 to 01 and of 00E to 08. Reconnect the A-ROM to the 3-Chip board and press RESET. This should turn on segment a of digit Z.

The key to understanding what is going on is to understand that every hexadecimal code is a shorthand way of writing eight binary digits (ones and zeros). Sheet T3 explains this - you might like to re-read it.

Sending 40 to port A is the same as sending binary 0100 0000. If you look at the organisation of port A you will see that this switches on segment g. Similarly, sending 02 to port B is the same as binary 0000 0010 which will turn on digit X. (Look at the information at the top of this sheet).

In the second example, sending 01 to port A is 0000 0001 which turns on segment a, while sending 08 to port B is 0000 1000 which turns on digit Z.

Note down the segment labels and the allocation of the bits of port A. Copy the labelling of the digits (W, X, Y & Z) and how port B is allocated.

1 What should you do to turn on segment c of digit Y? Write the answer, then test your answer by altering the program in the A-ROM and running it.

2 To display 7 you need to turn on segments a b & c. How could this be done?

Write a program for digit W, then check it by loading the A-ROM and reconnecting it to the 3-Chip Plus board.

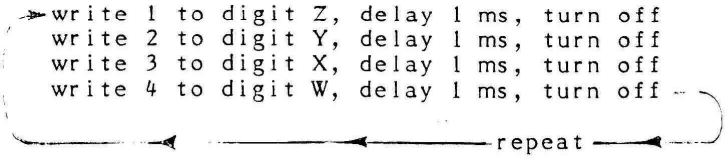
It's helpful to know codes that have to be written to port A to show each of the numbers 0 through to 9. They are easy to work out if you use a table. Copy this table into your notes.

Decimal number	Display	gef	bcda	hex. code
0	0	0011	1111	3F
1	1	0000	1100	0C
2	2	0110	1011	6B
3	3	0100	1111	4F
4	4	0101	1100	5C
5	5	0101	0111	57
6	6	0111	0111	77
7	7	0000	1101	0D
8	8	0111	1111	7F
9	9	0101	1101	5D

So to display 2 on digit Z you write 6B to port A and 08 to port B. Try it, then try: 3) display 4 on digit W. 4) 6 on digit X. 5) Work out the code to display on digit Y, then try it. 6) What are the codes for the other hexadecimal numerals?

(Add these characters to the table you already have).

Multiplexing: You might be wondering what the point of a four digit display is, if only one number is shown at a time. It is possible to display all four digits at once - or so the human eye thinks. In fact the digits are flashed on and off in rapid succession so quickly that the eye sees a continuous display. To show 1 2 3 4 the sequence is:



The software (program) to do this is given below.

Notice that the addresses are given as 200 etc. This is so that you can put the program in the A-ROM at 200 and upwards.

```
200   A9 FF          As program at the start of this sheet
202   8D 63 FE
205   8D 62 FE
208   A0 00
20A   A9 0C          Segment code for 1
20C   8D 61 FE       Store to port A
20F   A9 08          Code for digit Z
211   8D 60 FE       Store to port B
214   20 80 27       Delay 1 ms
217   A9 6B          Sends 6B (code for 2)
219   8D 61 FE                          to port A
21C   A9 04          and 04 (digit Y)
21E   8D 60 FE                      to port B
221   20 80 27       Delay 1 ms
224   A9 4F          Sends 4F (code for 3)
226   8D 61 FE                          to port A
229   A9 02          and 02 (digit X)
22B   8D 60 FE                      to port B
22E   20 80 27       Delay 1 ms
231   A9 5C          Sends 5C (code for 4)
233   8D 61 FE                           to port A
236   A9 01          and 01 (digit W)
238   8D 60 FE                      to port B
23B   20 80 27       Delay 1 ms
23E   4C 0A 22       Jump: next instruction executed: (2)20A
```

You will need to make three other changes:

```
   Change 789 to 04
          78E to 00
      and 7FD to 22
```

The first two changes will shorten the delay time down to 1 ms. The final change tells the microprocessor your program starts at 200.

Put the A-ROM back in the 3-Chip Plus board, power up and press RESET if necessary - the display should show 1 2 3 4.

o o o o 0 0 0 0 o o o o

If you have **done** sheet T5 go on now to T6(F) and then T7 (or T8 if you've done T7 too).

If you **haven't** done T5 go straight to the explanations on pages 3 and 4 (of T5) before moving to T6(F). You should then move to T5 (the practical part) and/or T7.

T6(F) Follow-up to 7-Segment Display

1 To turn on segment c of digit Y you must write 0000 0100 to port A and 0000 0100 to port B. That is 04 to port A and 04 to port B. So write 04 in address 009 of the previous program and 04 in 00E

2 To display 7 on digit W write 0000 1101 to port a and 0000 0001 to port B. That is 0D to port A and 01 to port B. 009 should be 0D and 00E should be 01 (in the previous program).

3 To display 4 on digit W, 009 should be 5C and 00E should be 01.

4 To display 6 on digit X, 009 should be 77 and 00E should be 02.

5 E requires segments a, d, e, f, g to be on, so the control code is 0111 0011 which is 73 in hex. To display this on digit Y, 009 should be 73 and 00E should be 04.

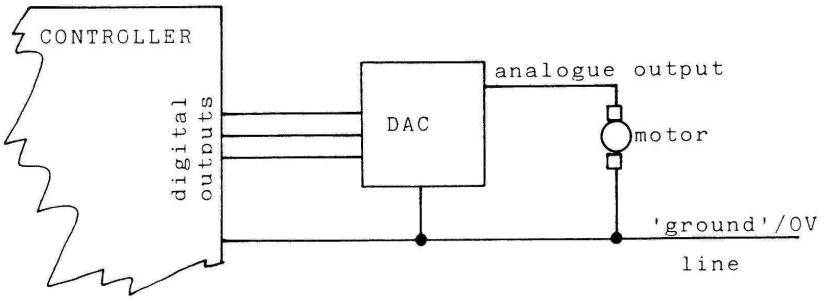
6

Display letter	g e f	b c d a	hex-code
A	0 1 1 1	1 1 0 1	7D
b	0 1 1 1	0 1 1 0	76
C	0 0 1 1	0 0 1 1	33
d	0 1 1 0	1 1 1 0	6E
E	0 1 1 1	0 0 1 1	73
F	0 1 1 1	0 0 0 1	71

Next worksheet - T5 or T7, or if you've done them both, T8

T7 Control of Speed, Brightness - and other Analogue Quantities

All the sheets so far have dealt with 'digital' outputs. That means that the control line can be 'on' or 'off', but nothing in between. Often one wants to set a voltage level, or current level to control, for example, the speed of a motor. A **digital** output will allow the motor to be switched fully on or fully off, an **analogue** output can be set to a number of levels in between. A **digital-analogue converter** (DAC) is a circuit that allows our 3-Chip Plus board to control analogue devices.



The diagram shows the idea - connecting the units up will help understanding.

* Plug the 8-LED Unit onto port A of the 3-Chip Plus board, then plug the Power DAC Unit onto the 8-LED board. Plug an 8-Switch Unit onto Port B (on the left hand side of the controller).

* Key the following program into the A-ROM (starting at 400):

```
400   A9 FF          The "port set-up routine"
402   8D 63 FE        (explained on sheet T9)
405   AD 60 FE       Load the accumulator from Port B
408   8D 61 FE       Store the accumulator to Port A
40B   4C 05 24       Go back to (2)405 and repeat
```

  You must also check that 7FC contains 00
  and 7FD contains 24

* Put the A-ROM back in the 3-Chip Plus controller and switch on. The 8-LED Unit should follow whatever you do to the 8-Switch Unit - wherever a switch is at logic 1 the LED should be lit. Try various switch positions.

* Connect a 15V d.c. power supply to the Power DAC (take care to get the + and - the right way round). The power pack must give smoothed d.c. - many science lab packs do, but some don't. Connect a voltmeter of 12V full scale deflection (or more) to the DAC output and/or connect a 12V lamp or d.c. motor. Motor or lamp should not draw more than 0.5A current. Set up hex 00 (= binary 0000 0000) on the input switches. Note the output voltage (or the behaviour of the lamp or motor).

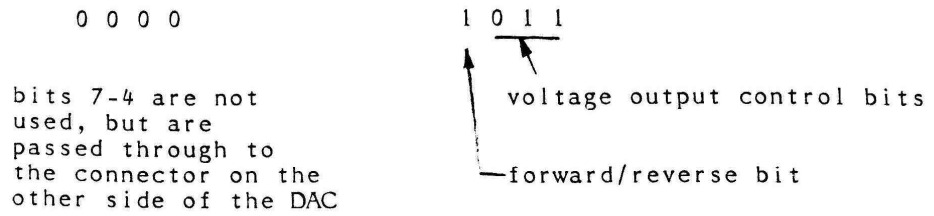
  Set up hex 01 (binary 0000 0001) on the input switches; note the output voltage (or behaviour of motor or lamp). Do this for all hex values between 00 and 07; the table below (copied into your notes) might be useful for summarising results.

Hex input	Binary input	DAC output voltage	(Ideal output)
00	0000 0000		(0V)
01	0000 0001		(1.75V)
02	0000 0010		(3.50V)
03	0000 0011		(5.25V)
04	0000 0100		(7.00V)
05	0000 0101		(8.75V)
06	0000 0110		(10.50V)
07	0000 0111		(12.25V)

  The designed output voltage is also shown in the table so that you can compare your measured voltage if you had a meter. There are bound to be small differences - don't worry about them. If a meter was not to hand, be assured that the output was close to the designed value.

  The output of the DAC is "1.75V per bit" - so for each unit on the input (which of course is copied directly to the output of port A), the analogue output rises by 1.75V.

* Change the binary input to 0000 1011 (hex 0B). The analogue output will be close to 5.25V but the polarity (direction) will be **reversed.** That is because **bit 3 controls the polarity** of the output.



To sum up, the power DAC has 7 voltage output levels (and 'off'), and can reverse the supply connections to a load. It can provide at least 0.5A current (and is overload protected). It should be driven by a 15V supply; it will work with less but the maximum output will be limited. A 6.5V supply will allow levels 0, 1, 2 and 3 to be output.

o o o o 0 0 0 0 o o o o

Sequencing the output

The controller board has done nothing of significance in that last system - the input switches could have been connected straight through to the Power - DAC. The system becomes more versatile when a sequence of events is wanted: say a motor is required to be speeded up from zero to maximum over 6s, held at maximum for 5s, then switched off until a restart button is pressed. The type of program on sheet T5 can be used to output hex numbers which set the output of the DAC and to delay for a given time before outputting the next code.

Can you write a program that will do this? Remember that the delay in the sub-routine at 780-79A is 50ms (one-twentieth of a second) multiplied by the number in the Y register.

How long must the output be at levels 01, 02, 03 etc, up to 07? What is the number needed in the Y register to achieve this? Try to write the program and load it. Don't forget to check that the delay routine in 780-79A is there (and that 789 has been put back to 43, and 78E to C3). You can load your program into the A-ROM at 500 upwards. Put 00 into 7FC and 25 into 7FD. The most interesting way to test is to connect a 12V d.c. motor to the output, or a model railway, but a 12V lamp will also do, or even a meter.

The answer is on T7(F) if you can't get it working.

T7(F) Follow up to Control of Analogue Quantities

The first part of sheet T7 should have been very straightforward. The voltages measured for the various levels were probably within 10% of the 'ideal' values - e.g. level 3 was 5.25V +/- 10%, i.e. between 4.73V and 5.77V. A more expensive DAC would get very much closer than this; this one is perfectly adequate for control of power devices in schools.

You may have noticed one new concept creep in - that of **loading** the accumulator with the contents of a memory cell whose address is given. The instruction code was 'AD' - you can enter it into your table:

Code	Assembler mnemonic	Name	Description
AD LL HH	STA HHLL	Load	Loads the accumulator with the contents of the address HHLL. As with 'store', the address is put after the code low-before-high.

Our example showed something else as well - that a Port can take in data, as well as pushing it out. The number read from address FE60 into the accumulator is the contents of that address - in other words the binary number set on the 8 switches connected to the port. More will be said later about when a port acts as an input and when as an output.

o o o o 0 0 0 0 o o o o

The program to increase the output voltage gradually to full is given below. The structure of the program is:

```
Set up port A
Code 01 to port A
Delay 1s
Code 02 to port A
Delay 1s
Code 03 to port A
Delay 1s
  .
  .
  .
  .
  .
Code 07 to port A
Delay 5s
Code 00 to port A
```

Remembering that 1s is twenty 'blinks' of delay (which is 14 in hexadecimal) we have:

```
500   A9 FF      LDA#FF      Set up as output
502   8D 63 FE   STA FE63                  port A
505   A9 01      LDA#01      Load code 01
507   8D 61 FE   STA FE61    and store it to port A
50A   A0 14      LDY#14      Twenty blinks into Y register
50C   20 80 27   JSR 2780        and jump to sub-routine
50F   A9 02      LDA#02      Load code 01
511   8D 61 FE   STA FE61       store to port A
514   A0 14      LDY#14      Delay into Y register
516   20 80 27   JSR 2780        and jump to sub routine
519   A9 03      LDA#03
51B   8D 61 FE   STA FE61
51E   A0 14      LDY#14
520   20 80 27   JSR 2780
523   A9 04      LDA#04
525   8D 61 FE   STA FE61
528   A0 14      LDY#14
52A   20 80 27   JSR 2780
52D   A9 05      LDA#05
52F   8D 61 FE   STA FE61
532   A0 14      LDY#14
534   20 80 27   JSR 2780
537   A9 06      LDA#06
539   8D 61 FE   STA FE61
53C   A0 14      LDY#14
53E   20 80 27   JSR 2780
541   A9 07      LDA#07
543   8D 61 FE   STA FE61
546   A0 64      LDY#64      One hundred 'blinks'
548   20 80 27   JSR 2780    Jump to delay sub-routine
54B   A9 00      LDA#00      Switch off
54D   8D 61 FE   STA FE61
550   4C 50 25   JMP 2550    Jump to this instruction
                             - an eternal loop.
```

Don't forget that the delay sub-routine at 780 must be right, 7FC should contain 00 and 7FD should contain 25.

Again this program is very boring - the next sheet will show a much more elegant way of handling this problem. But before moving on, learn about a couple more facilities of the 6502 microprocessor.

First, why was the sub-routine address 2780 when you put the code into the A-ROM at 780 upwards? The reason is that the microprocessor needs an address with **four** hex digits, and the zero insertion force (zif) socket on the 3-Chip Plus board is wired in at 2000 to 27FF. So address 780 in the A-ROM is seen by the microprocessor as 2780. Similarly, the jump instruction at the end of the program above sends the microprocessor back to 2550, which is the same bit of program, creating an everlasting loop.

Secondly, what about these mystery locations 7FC and 7FD? They contain what is called the **reset vector**, and again it's address low byte before address high byte. So if 7FC contains 00 and 7FD has 25 in it, when the RESET button is pressed, the microprocessor will begin by executing the machine code in address 2500 (which is, of course, address 500 in the A-ROM). Note also that the word **byte** slipped in there. Byte is such a common jargon-word that you should know that it is the same as 8 binary digits (or bits). It is therefore the same as two hex digits. Thus FE 61 is a **two byte** address (the address of port A).

Finally, notice that the program listing has had the 'assembler mnemonics' put in. Once you get used to the page full of hieroglyphics you will find them very helpful - indeed you will write the programs first with these, putting the actual machine code instructions second. It is much easier to remember that **LDY#14** means **Load the Y register with the number 14** than it is to remember that A0 14 means this. From now on all listings on these sheets will have the mnemonics as well as the machine code. Often comments will then not be necessary.

Make notes on anything you are likely to forget. It's important to know that the zif socket is at 2000 - 27FF and that the reset vector is in 27FC (low byte) and 27FD (high byte).

Next worksheet - If you haven't done T5 yet, look now at the explanations of pages 3 and 4 of it. Then do the practical part of T5 if possible. If the T5 equipment is busy or you've done it, do T6. If you've done both T5 and T6, go to T8.

T8 More Bite with less Bytes

The motor speeding up program on the last sheet was very tedious. How can more be done with less program? First meet some new instructions - copy them into your table:

Code	Assembler mnemonic	Name	Description
C9 NN	CMP#NN	Compare (Immediate)	Compares the contents of the accumulator with the next byte of the program. 'Sets' a 'flag' in the microprocessor (the zero flag) if the comparison shows that the numbers are the same.
D0 XX	BNE XX	Branch if not equal	If the zero flag is not set, the processor will branch to another address and begin to execute the code there. The next byte (XX) after D0 specifies how long the 'branch' is.
F0 XX	BEQ XX	Branch if equal	If the zero flag is set, the processor will branch to another address and begin to execute the code there. the next byte (XX) after F0 specifies how long the 'branch' is.
EE LL HH	INC HHLL	Increment memory	Increase by 1 (adds 1 to) the contents of the memory cell HHLL.

Can you now see how this program works?

```
(2)500        A9 FF       LDA#FF      Set up as output
(2)502        8D 63 FE    STA FE63              Port A
(2)505        A9 01       LDA#01      Load 01 and
(2)507        8D 61 FE    STA FE61      send to port A
(2)50A LOOP   A0 14       LDY#14      14 blinks (( = twenty) to Y reg
(2)50C        20 80 27    JSR DELAY   Jump to DELAY sub routine at
                                      2780
(2)50F        AD 61 FE    LDA FE61    Load the accumulator with the
                                            contents of port A
(2)512        C9 07       CMP#07      Compare with 07
(2)514        F0 06       BEQ DONE    If it's equal, branch over the
                                      next
                                                          06 bytes
(2)516        EE 61 FE    INC FE61    Otherwise increase port A by 1
(2)519        4C 0A 25    JMP LOOP    and jump back to (2)50A
(2)51C DONE   A0 50       LDY#50      Prepare for 4s delay
(2)51E        20 80 27    JSR DELAY   Jump to DELAY sub routine at
                                            2780
(2)521        A9 00       LDA#00      Load 00 and
(2)523        8D 61 FE    STA FE61          send to port A
(2)526 END    4C 26 25    JMP END     Twiddle thumbs and keep out of
                                                  mischief.
```

As usual the delay sub-routine, and the reset vector must be correct (in (2)780 - (2)79A and (2)7FC and D).

There is one final tip that can be given to program writers which makes life a lot easier - use **labels**. This last program was labelled to make it easier to follow - the labels were underlined. You can use any words you like - the ones above indicate what point the program is at. Names like BILL, BEN, BOB, BERT are an alternative.

To talk through that program:

2500 - 2504	set up port A as an output
2505 - 2509	output 01 from port A (connected to the Power DAC)
250A - 250E	cause a 1s delay
250F - 2511	load the accumulator with the current value of port A
2512 - 3	compare it with 07. The 'zero flag' (mentioned at the beginning of this sheet) is a single binary digit store in the microprocessor whose value must be 0 or 1. In this case, if the accumulator contents were 07, the zero flag will be 1.
2514 - 5	If (and only if) the zero flag **equals** 1, the microprocessor **branches** on 6 bytes. That means that when the port A value becomes 07, the processor will go on to 'DONE'.
2516 - 2518	If port A was not 07, this instruction will be the next, and port A will be 'incremented' (increased by 1).
2519 - 251B	Jumps back to 250A (LOOP) to go round the loop again.
251C - 2520	Causes a further 4s delay when 07 has been reached - 1s delay has already been caused
2521 - 2525	Send 00 to the port to set the output of the DAC to OV.
2526 - 2528	This instruction jumps back to itself (END) forever. This is a good way of ending a program to stop any random events happening.

Copy one more instruction code into your table, and then try a programming assignment:

Code	Assembler mnemonic	Name	Description
CE LL HH	DEC HHLL	Decrement memory	Decreases by 1 (sub tracts 1 from) the contents of the memory cell HHLL.

Can you now extend the program so that after 5 seconds at full power the output of the DAC is taken down to 0V in smooth steps? You'll need to change the code from (@)521 onwards.

As a somewhat more difficult assignment you might like to try writing one of these programs using the BNE instruction - which will skip if the comparison shows the comparison **not** to be 'true'.

The answers to these problems are on sheet T8(F)

T8(F) Follow-up More Bite with less Bytes

To extend the program to drop the output steadily to zero, this code can be used:

```
(2)521 SLOW CE 61 FE  DEC FE61   Decrease port A by 1
(2)524      F0 08     BEQ END    If equal to zero branch to end
(2)526      A0 14     LDY#14     Load 14 (twenty) blinks
(2)528      20 80 27  JSR DELAY    and jump to delay sub-routine
(2)52B      4C 21 25  JMP SLOW   Jump and do again
(2)52E END  4C 2E 25  JMP END    End when port A = 00.
```

The assignment to write the code using BNE might have produced this result (for the first piece of program):

```
(2)500      A9 FF     LDA#FF
(2)502      8D 63 FE  STA FE63
(2)505      A9 01     LDA#01
(2)507      8D 61 FE  STA FE61
(2)50A LOOP A0 14     LDY#14
(2)50C      20 80 27  JSR DELAY
(2)50F      AD 61 FE  LDA FE61
(2)512      C9 07     CMP#07
(2)514      D0 0D     BNE AGIN   Branch on to 2523 if not =
(2)516      A0 50     LDY#50     Otherwise delay 4s
(2)518      20 80 27  JSR DELAY
(2)51B      A9 00     LDA#00     Output to zero
(2)51D      8D 61 FE  STA FE61
(2)520 END  4C 20 25  JMP END    and loop eternally
(2)523 AGIN EE 61 FE  INC FE61   Increase output by 1
(2)526      4C 0A 25  JMP LOOP    and go back to loop
```

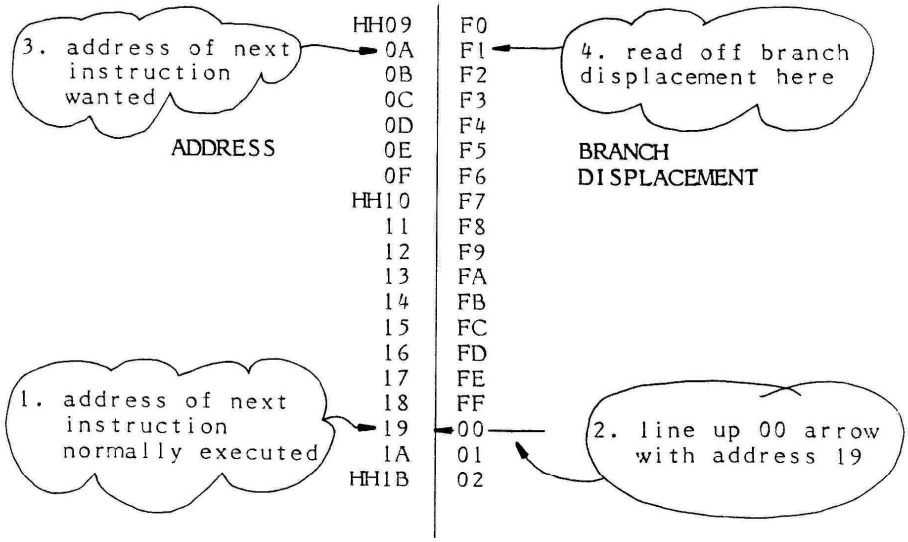
That might have caused difficulty either because yo wanted to branch **back** in the program and didn't know how to, or because you weren't sure how to work out the branch forwards. The branch forwards is not too difficult - in this program you want to miss thirteen bytes of code, which is 0D in hex. But is it possible to branch backwards? "Yes" is the answer. Look at this program:

```
(2)505       A9 01      LDA#01     Send 01
(2)507       8D 61 FE   STA FE61      to port A
(2)50A LOOP  A0 14      LDY#14     Delay
(2)50C       20 80 27   JSR DELAY          1s
(2)50F       EE 61 FE   INC FE61   Increase port A by 1
(2)512       AD 61 FE   LDA FE61
(2)515       C9 07      CMP 07     Is it 07 (at the top)?
(2)517       D0 F1      BNE LOOP   If not then do again
(2)519       A0 64      LDY#64     If yes,
(2)51B       20 80 27   JSR DELAY           delay 5s
(2)51E       A9 00      LDA#00     Output
(2)520       8D 61 FE   STA FE61           to zero
(2)523 END   4C 23 25   JMP END    and loop eternally
```

Here the program keeps branching back until the output to the port reaches 07 when the output is held steady for 5 seconds then reduced to zero.

But why does "D0 F1" take you back to LOOP? Look at the 'branch calculator' printed on a separate sheet. (You can usefully stick a photocopy of it onto a piece of card, then cut it carefully down the middle.) F1 is the **branch displacement** (so was 0D at address 2515 in the previous program). All displacements from 01 to 7F are **forwards**; all from FF to 80 are **backwards**.

Line up the arrow on card R with address HH19 on card L. 2519 is the next instruction that would be executed if it weren't for the branch instruction. The instruction that you want next is 250A: opposite this is the branch displacement **F1**.



Using the cards makes it very much easier to calculate displacements, especially negative ones (for branches backwards in the programs). Check that you can use the cards reliably by calculating the branch displacements for the program on sheet T8, and the other two programs on T8(F).

Note that "HH" stands for the high byte of the address - it could be anything (in our case it was 25). Also note that these cards can be used only for branches of about forty eight (hex 30) bytes in each direction. You could easily make cards (or a pair of concentric discs of slightly different radii) to handle the full 80 (hex) bytes in either direction.

Next Sheet - T9

Branch Calculating Card

L

ADDRESS

HH00	D1
HH01	D2
HH02	D3
HH03	D4
HH04	D5
HH05	D6
HH06	D7
HH07	D8
HH08	D9
HH09	DA
HH0A	DB
HH0B	DC
HH0C	DD
HH0D	DE
HH0E	DF
HH0F	E0
HH10	E1
HH11	E2
HH12	E3
HH13	E4
HH14	E5
HH15	E6
HH16	E7
HH17	E8
HH18	E9
HH19	EA
HH1A	EB
HH1B	EC
HH1C	ED
HH1D	EE
HH1E	EF
HH1F	F0
HH20	F1
HH21	F2
HH22	F3
HH23	F4
HH24	F5
HH25	F6
HH26	F7
HH27	F8
HH28	F9
HH29	FA
HH2A	FB
HH2B	FC
HH2C	FD
HH2D	FE
HH2E	FF
HH2F	00
HH30	01
HH31	02
HH32	03
HH33	04
HH34	05
HH35	06
HH36	07
HH37	08
HH38	09
HH39	0A
HH3A	0B
HH3B	0C
HH3C	0D
HH3D	0E
HH3E	0F
HH3F	10
HH40	11
HH41	12
HH42	13
HH43	14
HH44	15
HH45	16
HH46	17
HH47	18
HH48	19
HH49	1A
HH4A	1B
HH4B	1C
HH4C	1D
HH4D	1E
HH4E	1F
HH4F	20
HH50	21
HH51	22
HH52	23
HH53	24
HH54	25
HH55	26
HH56	27
HH57	28
HH58	29
HH59	2A
HH5A	2B
HH5B	2C
HH5C	2D
HH5D	2E

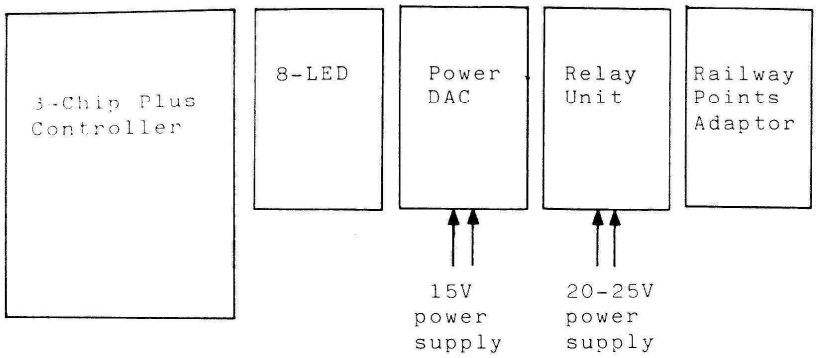
Branch Calculating Card

R

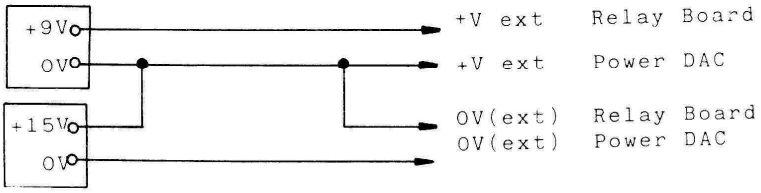
BRANCH DISPLACEMENT

T9 Sequences: Controlling a Model Railway (for example)

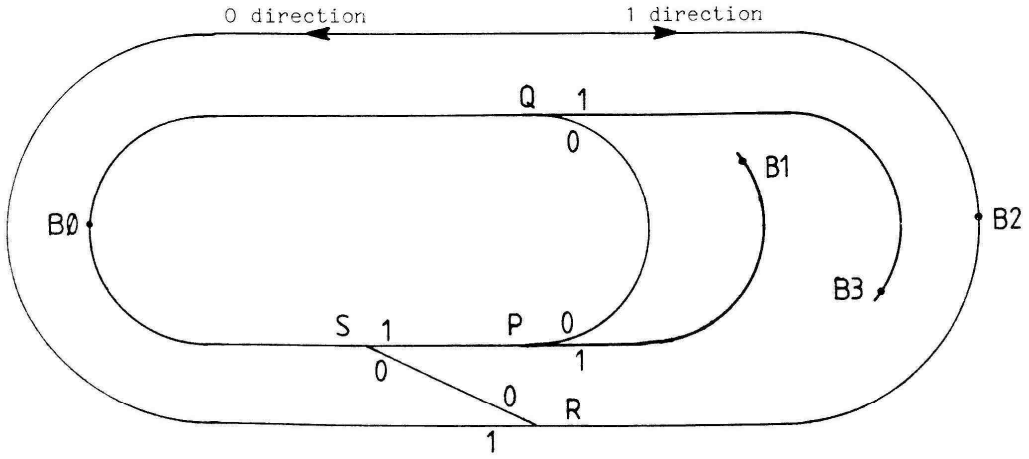
By "writing" the right code to the right port it is easy to control a model railway. Similar techniques can be used to control any other device needing switching or speed control. This sheet will explain the programming trick which was used to shorten the program for the tone generator. If you want to work with a model railway complete with points, you will need a railway points adaptor. The arrangement is:



The two power supplies shown may be two lab supplies connected together:



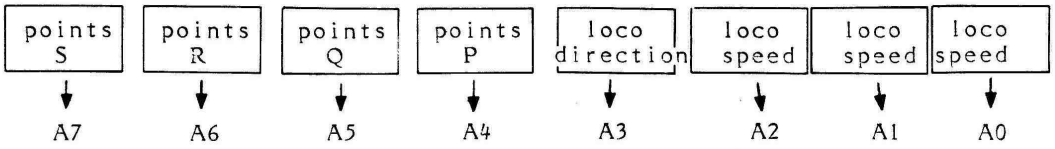
The layout of the railway discussed as an example is:



The four points are labelled P Q R and S and the directions of the points are labelled '0' and '1'.

Four position sensors are labelled B0 - B3 and may be connected to a ribbon cable or a connector. These will be used on a subsequent sheet.

PORT A will be used to control the track and PORT B to read in where the train is. The eight lines of port A are:



Suppose we send 2F to port A: in binary this is -

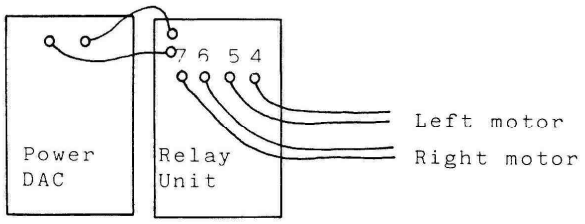
0	0	1	0	1	1	1	1
Point S will be set downwards in the diagram above	Point R will be set upwards in the diagram above	Point Q set so that the loco goes into the siding	Point P will be set upwards in the diagram above	Loco direction is clockwise	at	full	speed

If the loco were at sensor B2 and this code were sent to port A, the loco would travel to B3 at full speed. If you grasp this, it should be easy to work out what hex value will cause any required movement.

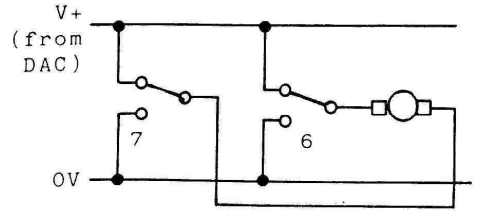
o o o o 0 0 0 0 o o o o

If you haven't a railway, you could drive a 'MFA' vehicle via the Movement Module from the port. Bits 0 and 2 control on/off for the two motors, and bits 1 and 3 the direction of each.

Alternatively, wire up the Relay Unit and the Power DAC to control and reverse two motors, like this:



Bits 2, 1 and 0 give you control over the motor speed. Bit 3 will reverse both motors.



With the relays in the position shown what will be the motor's behaviour? This correspons to logic 1 on bit 7 and logic 1 on bit 6.

With logic 0 on both bits 7 and 6 what will the motor do?

If with logic 0 on bit 7 and logic 1 on bit 6 the motor drives the vehicle forward, what will logic 1 on bit 7 and logic 0 on bit 6 cause the motor to do?

What codes will need to be output then to give full ahead, full reverse, right turn, and left turn? What codes for half speed?

If your motors are only 4.5V types, what is the **maximum** speed code you should send to the Power DAC?

o o o o 0 0 0 0 o o o o

The technique for outputting codes in a sequence involves setting up a table of values and then using them in turn. We have a delay sub-routine at (2)780 which delays the number of 'blinks' (50ms periods) that has been put into the Y register before jumping to the sub-routine. So all that is needed is a sequence:

```
Send next value in table to port A (= control code)
Send next value in table to Y Register (= delay)
Jump to DELAY sub-routine
On return from sub-routine do again
```

You may have noticed immediately that alternate values in our table are going to be control codes (interleaved with delay values). To send the loco from B2 on the track to B3, then to B1 and stop for 12s would need a sequence of codes:

```
(2)080   2F      B2 - B3 at full speed
(2)081   46      delay 3.5s which is 70 blinks (hex 46)
(2)082   B4      B3 - B1 at half speed
(2)083   80      delay 5s
(2)084   B0      stop
(2)085   FF      maximum delay time (12.2s)
```

Notice that these are put from 080 to 085 in the A-ROM. If you are controlling something other than this railway layout, work out your own codes and delays to put in the table, after deciding on a suitable sequence.

Meet four new instructions:

Code	Assembler mnemonic	Name	Description
BD LL HH	LDA HHLL,X	Load accumulator absolute, indexed X	The accumulator is loaded with the contents of the memory cell whose address is (HHLL + X), where X is the contents of the X Register.
BC LL HH	LDY HHLL,X	Load Y Register absolute, indexed X	The Y Register is loaded with the contents of the memory cell whose address is (HHLL + X), where X is the contents of the X Register.
A2 NN	LDX#NN	Load X Register immediate	Loads the microprocessor 'X Register' with the next number in the program - NN is meant to indicate the next number.
E8	INX	Increment X Register	Increments (increases by 1) the contents of the X Register.

The 'indexed' loading instructions makes possible what we want to do. BD 80 20 will load the accumulator from 2080 if the content of the X Register is 00. If the X Register contains 01, the accumulator will be loaded from 2081, and so on.

The program is:

```
(2)000        A9 FF      LDA#FF        Port A
(2)002        8D 63 FE   STA FE63          Set for output
(2)005        A2 00      LDX#00        00 into X Register
(2)007 LOOP   BD 80 20   LDA 2080,X    Load accumulator with next code
(2)00A        8D 61 FE   STA FE61       Send code to Port A
(2)00D        E8         INX           Increment the X Register
(2)00E        BC 80 20   LDY 2080,X    Load Y Register with delay
(2)011        20 80 27   JSR DELAY      Jump to DELAY sub-routine
(2)014        E8         INX           Increment the X Register
(2)015        4C 07 20   JMP LOOP      Jump to entry point of main
                                          loop
```

It's as simple as that! One obvious flaw is that the program will start outputting garbage as soon as all the values in the table have been used (in the last case, as soon as the 12s delay is up with the loco stopped). One way of overcoming this is to put 00 in the table at the end of your values, and using a branch instruction to skip to a jump-to-itself instruction at the end of the program. Can you work out the program? (and then test it!).

Finally, what should your look-up table contain to take the loco from B3, clockwise round the inner loop once at half speed, out to B2, then clockwise round the outer loop flat out? (If you are not using the railway, work out another control sequence for your gadget). The delays in this kind of control have to be about right for the thing to work correctly. We shall meet the idea of **feedback** on sheet T10 which makes it very much easier to control a mechanical system reliably.

Next Sheet - T9(F)

T9(F) Follow-up to Sequences

Driving motors from the relay unit with bit 7 and bit 6 = 1, both motor terminals will be connected to V+ and the motor will therefore be off.

Logic 0 on bits 7 and 6 will also turn the motor off.
Logic 0 on bit 7 and 1 on bit 6 will turn the motor on, perhaps clockwise (depending on how it is connected).
Logic 1 on bit 7 and 0 on bit 6 will rotate the motor in the other direction.

Full ahead will need 0101 0111 which is 57 (hex)
Full reverse will need 1010 0111 which is A7 (hex)
(or you could use 0101 1111 which is 5F (hex))

From the drawing you can see that bits 4 and 5 go to the left motor and bits 6 and 7 go to the right motor

So to turn right 0001 0111 which is 17 (hex) this will turn off the right motor and drive the left.

A pivotting right turn will need 1001 0111 which is 97 (hex)
turn left 0100 0111 which is 47 (hex)
pivot left 0110 0111 which is 67 (hex)

to do any of these at half (approx) speed , use 4 for the speed digit instead of 7, i.e. 14 will turn right slowly.

the Power DAC gives 1.75V per bit, so really you should not use more than 2 for the speed digit, 3 gives 5.25V output which in practice will probably be safe.

o o o o 0 0 0 0 o o o o

To test each output code fetched from the table of values, use the BEQ instruction. This will cause a branch only if the value fetched is 00 (the zero flag will be set only if the value fetched is 00).

The program will be:

```
(2,000       A9 FF      LDA#FF
(2)002       8D 63 FE   STA FE63
(2)005       A2 00      LDX#00
(2)007 LOOP  BD 80 20   LDA 2080,X
(2)00A       F0 10      BEQ END
(2)00C       8D 61 FE   STA FE61
(2)00F       E8         INX
(2)010       BC 80 20   LDY 2080,X
(2)013       F0 07      BEQ END
(2)015       20 80 27   JSR DELAY
(2)018       E8         INX
(2)019       4C 07 20   JMP LOOP
(2)01C END   4C 1C 20   JMP END
```

Don't forget to put 00 at the end of your table (in (2)086). The branch calculating cards can be used to calculate the displacements.

To go from sensor B3 on the track out of the siding needs 10100100 which is A4. Actually this could be E4 since points R don't matter. The delay is a guess - try 3s which is sixty 'blinks' or 3C in hex.

Clockwise round the inner loop needs 10001100 which is 8C. Try a delay of 6s which is one hundred and twenty blinks or 78 in hex.

To move out onto the outer loop use 00000100 which is 04 in hex, for a period of 6s again - 78 in hex.

Finally to go round the outside use 01001111 which is 4F in hex for a delay of FF, then end your table with 00 - the loco will continue looping.

So the table of codes should be:

```
(2)080   A4
(2)081   3C
(2)082   8C
(2)083   FF
(2)084   04
(2)085   C0
(2)086   4F
(2)087   FF
(2)088   00
```

The delay times can be adjusted until they are about right for the respective phases of the journey.

One final tip when working with point motors. The system used here detects **changes** in output and kicks the point appropriately. When the system is first switched on, all the point outputs are at logic 1. If the first commanded value for any point is 1 as well, it won't receive any kick. That's fine so long as it's already in the right physical position at the start of the sequence - and it may not be. It is therefore useful to 'initialise' the system before you go into a sequence of codes. Like this:

```
(2)080   08   All points to 0
(2)081   14   Delay for 1s
(2)082   A4   Start of sequence proper
(2)083   3C
(2)084   8C
(2)085   FF
      etc
```

Why use 08 and not 00; can you work it out? It's to do with the BEQ instruction.

Next Sheet - T10

T10 Inputs, and Checking Them

Most of the sheets so far have dealt with outputs and how to control them. Port B has occasionally been used for inputs; now we need to learn the hows and whys of the ports.

The chip at the end of the 3-Chip Plus board (labelled 6522) is known as a "v.i.a." - versatile interface adaptor. It provides two ports (routes to the outside world) plus a number of additional features like timers which we won't meet formally though one is used in the 50ms delay sub-routine. These ports themselves are versatile and can be both input, both output, or allocated between input and output to suit the application. How is this allocation carried out?

Associated with each port is what is known as a **data direction register** (d.d.r. for short).

D.d.r. A (for port A) is at address FE63 on 3-Chip Plus.
D.d.r. B (for port B) is at address FE62 on 3-Chip Plus.

The word 'register' means 'store' as we know, and the hex number stored in it tells the v.i.a. whether the port is going to be used as an input or output. At the start of many of our programs has been:

```
A9 FF      LDA#FF
8D 63 FE   STA FE63
```

The sole purpose of this code is to store FF in d.d.r. A so that port A will be an **output** port.

To set up port B as an input the necessary code would be:

```
A9 00      LDA#00
8D 62 FE   STA FE62
```

but if you look back to previous sheets you **won't** find this present anywhere, even when an input was made (e.g. on sheet T7). This is because after RESET on the board is pressed, both ports are set up as inputs. There is something to be said for the habit of systematically setting up all ports, even those used as inputs, but it isn't strictly necessary for an all-input port.

Those last 3 words imply that a port need not be all input or all output. In fact individual lines may be set up as you wish.

* To set up a line as an input store a '0' in the corresponding place in the d.d.r.

* To set up a line as an output store a '1' in the corresponding place in the d.d.r.

So to set up port A as all output, store 1111 1111 which is hex FF at address FE63 (the address of d.d.r. A)

To set up port B as all input store 0000 0000, hex 00, at address FE62 (the address of d.d.r. B).

To set up port A as four input (lines A7 - A4) and four output (lines A3 - A0), store 0000 1111, hex 0F, at address FE63.

o o o o 0 0 0 0 o o o o

Load this code in an A-ROM:

```
(2)000        A9 FF       LDA#FF      Store FF
(2)002        8D 63 FE    STA FE63      at d.d.r. A
(2)005        A9 00       LDA#00      Store 00
(2)007        8D 62 FE    STA FE62      at d.d.r. B
(2)00A LOOP   AD 60 FE    LDA FE60    Load port B
(2)00D        8D 61 FE    STA FE61    Store to port A
(2)010        4C 0A 20    JMP LOOP    and keep doing it
```

Don't forget to put 00 and 20 into the Reset vector location (2)7FC and (2)7FD.

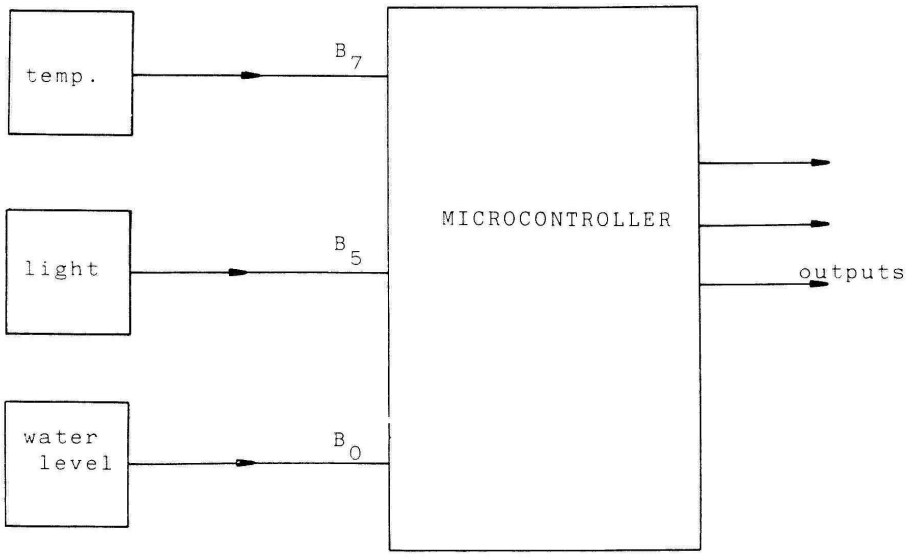
Put the A-ROM in a controller, plug an 8-LED Unit into port A and an 8-Switch Unit into port B. The LEDs should be directly controlled by the switches.

Change the program so that the content of port A is loaded into the accumulator, and then stored to port B. Swap over the Switch Unit & LED Unit (they will plug onto the other port correctly, although it looks strange). Check the behaviour.

Assuming that you did **not** change the numbers stored in the data direction registers, you will find that the system seems dead - no LEDs lighting anywhere. Now change the d.d.r. contents so that port A is set up as an input and port B an output. The system should behave properly now.

o o o o 0 0 0 0 o o o o

The need often arises to check one line to see whether the input device connected to it is now giving logic 1 (say).



Three inputs connect to the microcontroller and the program will send control voltages to different outputs according to the states of these inputs and time elapsed. An input may be checked by a process called masking. Try out this next program:

```
(2)000        A9 FF       LDA#FF        Send FF
(2)002        8D 63 FE    STA FE63        to d.d.r. A
(2)005 NONE   A9 80       LDA#80        Load 80 (1000 0000)
(2)007        2C 60 FE    BIT FE60      Bit test Port B
(2)00A        D0 11       BNE HOT       Branch if result non-zero
(2)00C        A9 20       LDA#20        Load 20 (0010 0000)
(2)00E        2C 60 FE    BIT FE60      Bit test Port B
(2)011        D0 0F       BNE LGHT      Branch if result non-zero
(2)013        A9 01       LDA#01        Load 01 (0000 0001)
(2)015        2C 60 FE    BIT FE60      Bit test Port B
(2)018        D0 0D       BNE WET       Branch if result non-zero
(2)01A        4C 05 20    JMP NONE      All zero, so check again
(2)01D        A9 03       LDA#03        Load accumulator and
(2)01F        4C 29 20    JMP NOTE      jump to note routine
(2)022 LGHT   A9 05       LDA#05        Load accumulator and
(2)024        4C 29 20    JMP NOTE      jump to note routine
(2)027 WET    A9 09       LDA#09        Load accumulator and pass on to
                                              note routine
(2)029 NOTE   8D 61 FE    STA FE61
(2)02C        A0 0A       LDY#0A
(2)02E        20 80 27    JSR DELAY           Sound pair of
(2)031        AD 61 FE    LDA FE61            siren notes
(2)034        49 FF       EOR#FF              once
(2)036        8D 61 FE    STA FE61
(2)039        A0 0A       LDY#0A
(2)03B        20 80 27    JSR DELAY
(2)03E        4C 05 20    JMP NONE      Go back to check inputs again.
```

The Reset vector needs to be 2000 again, and you need an 8-Switch Unit on Port B and a Sound Board on Port A. Switch all switches to 0, the A-ROM in the controller and press RESET. Nothing should happen. Switch input 0 to logic 1 and a siren will sound, switch it back to 0 and the siren will stop (but see below **). Switch input 5 to logic 1 and a different pair of notes will sound. Switch input 5 off again, then switch input 7 on - a different pair of notes again.

In trying to follow the program don't worry too much about the 'NOTE' routine at (2)029 - (2)03E. The odd instruction is EOR#FF: it simply inverts the state of all the bits. so when 'NOTE' is jumped to, the accumulator contains a value which is sent to port A and the routine alternates this note code with another which is the 'complement' of it. But you can use the routine without knowing this!

You do need to know about the BIT instruction though:

Code	Assembler mnemonic	Name	Description
2C LL HH	BIT HHLL	Bit test	The contents of the accumulator are used to test the contents of the cell whose address is specified. The accumulator contents specify the bit(s) to be examined. If any of those specified contain 1, the zero flag will be clear indicating a non-zero result. BNE will cause a branch. If none of those specified contain 1, the zero flag will be 'set' indicating a zero result. No branch will occur at BNE.

If that seems complicated, look at the program. The magic combination of op-codes occurs first at (2)005:

```
LDA#testbitcode
BIT portaddress
BNE whatshouldhappen
```

will set up the test. Where are the other similar tests carried out?

o o o o 0 0 0 0 o o o o

** The behaviour is unsatisfactory in that a note is left sounding constantly once the input goes low again. Can you modify the program to stop this happening?

Try switching **two** inputs to logic 1 simultaneously - what happens? Try different pairs of inputs. Can you explain this behaviour from the program? Is it satisfactory? How **ought** the system to behave?

Next Sheet - T10(F)

T10(F) Follow-up to Inputs, and Checking Them

The program to load from port A and store to port B is simply:

```
(2)000        A9 FF      LDA#FF        put FF
(2)002        8D 62 FE   STA FE62      in d.d.r. B
(2)005        A9 00      LDA#00        put 00
(2)007        8D 63 FE   STA FE63      in d.d.r. A
(2)00A LOOP   AD 61 FE   LDA FE61      Load port A
(2)00D        8D 60 FE   STA FE60      Store to port B
(2)010        4C 0A 20   JMP LOOP      and do again
```

This is the fully corrected program - with data direction registers changed too.

o o o o 0 0 0 0 o o o o

Bit tests occur

at (2)005 - (2)00B (bit 7 is tested, the microprocessor branches to (2)01D if bit 7 is 1)
at (2)00C - (2)012 (if bit 5 is 1, branch to (2)022)
and at (2)013 - (2)019 (if bit 0 is 1, branch to (2)027).

o o o o 0 0 0 0 o o o o

A note is left sounding because the NOTE routine doesn't kill the sound generator before going back to the bit tests. This is easily remedied with

```
(2)03E        A9 00      LDA#00        send 00
(2)040        8D 61 FE   STA FE61        to Port A
(2)043        4C 05 20   JMP NONE
```

Can you explain why the tone generator does not appear to go silent so long as a 'live' switch is at 1? (the code 00 is certainly sent to the port **every** time the system goes through the NOTE routine). Can you explain why the tone does not stop **immediately** the offending switch is put back to the 0 state?

If two 'live' inputs are switched at the same time, only the 'higher' one causes an effect. The program checks bit 7 (heat) before bit 5 (light) before bit 0 (wet), and a branch immediately happens to the note routine if a fault condition is detected. The system is **prioritised.** This may be a good thing - or not. Perhaps it would be better to have a siren (with two fixed tones) plus a set of indicator lamps to show which one or more fault conditions have occurred. The Music Module only uses bits 3-0 so if you connect the 8-LED Unit to 3-Chip Plus, then the Music Module, you can control LEDs 7-4 completely independently of the sound. You will need to know what the ORA instruction does before writing the program for this modified system - see sheet T11.

Next sheet - T11

T11 Controlling the Railway II - Feedback

Learning sheet T9 showed the main principles behind controlling the points and locomotive; this sheet will show you how to 'read' the sensors so that you can tell when the loco has reached a particular position. The problems of controlling a device or system by time delays should be apparent by now. Unless you know exactly how the external system (in this case the railway) will behave, it is very difficult to control precisely. Furthermore, any variation in system behaviour (like a slow-running loco) cannot be coped with. This is why **feeding back** data to the controller about the external system's behaviour is so important. The only alternative is to have an external system whose behaviour can be guaranteed - stepper motors (which some users of these sheets will have encountered) are an example here.

All the detailed instructions on this sheet are framed around a model railway as described on sheet T9, but any other controllable device which has position sensors attached can be used so long as you can interpret the guidance material into the situation.

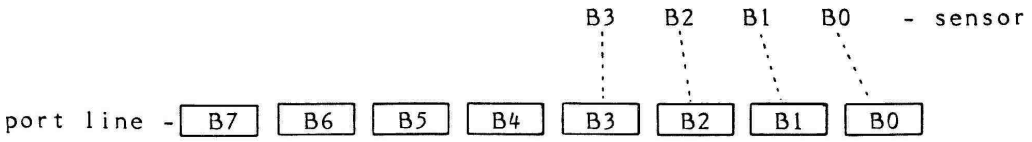
o o o o 0 0 0 0 o o o o

Key into the A-ROM the program from sheet T10 which reads the input from port B and immediately stores it to port A.

Connect the 8-LED Unit to port A on the 3-Chip Plus board and the Sensor Input Unit (connected to the railway) to port B. Transfer the A-ROM and press the RESET key to run the program. The LEDs are now be showing you what the track sensors are sending out - they will probably be all off showing that all four sensor outputs are logic 0. (For the sensors to work reliably, the railway board must be in a well-lit room).

Shade a track sensor and one of the LEDs should come on, telling you that there is now an output from that sensor.

The way the sensors are connected to the lines of the port is:



In other words, lines B4 - B7 aren't used, and the sensors shown on the track layout plan (sheet T9) are labelled according to the port line they are connected to. Refer to the track plan, shade sensor B0 and check that LED 0 comes on. Do the same for sensors B1, B2 and B3. You may like to note down the organisation of the sensors.

The BIT instruction met on the last sheet can be used to check a sensor by a simple procedure. Load this program into an A-ROM:

```
(2)000        A9 FF       LDA#FF      Set up for output,
(2)002        8D 63 FE    STA FE63       Port A
(2)005        A0 08       LDY#08      All points to 0
(2)007        8C 61 FE    STY FE61       to Port A (via Y Reg)
(2)00A        A0 14       LDY#14      Delay time to Y Reg
(2)00C        20 80 27    JSR DELAY   and jump sub-routine to do it
(2)00F        A0 2D       LDY#2D      Points/speed to Y Reg.
(2)011        A9 08       LDA#08      Sensor code to accumulator
(2)013        20 80 20    JSR TEST    Jump to output and bit test rtn
(2)016        A0 00       LDY#00       Second points/speed to Y Reg
(2)018        8C 61 FE    STY FE61    Store to Port A
(2)01B END    4C 1B 20    JMP END     End of program.
```

==========

```
(2)080 TEST 8C 61 FE    STY FE61    Y Register to Port A
(2)083 NYET 2C 60 FE    BIT FE60    Test Port B (ref contents of
                                                          acc)
(2)086        F0 F8       BEQ NYET    If sensor zero do again
(2)088        60          RTS         Return to main program
          (The delay sub-routine at (2)780 must be intact)
```

These new instructions are used (copy them into your growing table):

Code	Assembler mnemonic	Name	Description
8C LL HH	STY HHLL	Store Y Register (absolute)	Stores the contents of the Y Register to absolute address HHLL.
60	RTS	Return from sub-routine	Placed at the end of every sub-routine to tell the microprocessor to go back to the main program.

This program
* sets up port A for output
* puts all points to 0 for 1s (see sheet T9 for the reason)
* loads the required output code into Y and the sensor test code to A (the accumulator)
* uses a sub-routine to output the points/speed code, then waits for as long as it takes for the sensor to be covered.

Transfer the A-ROM to the controller, put the loco at B2 press RESET. The loco should move from B2 to B3 and stop when it gets there - when the sensor is covered.

o o o o 0 0 0 0 o o o o

It is possible to write that last simple program using less bytes of machine code, but the use of a sub-routine will make a more sophisticated control program much simpler. The program also illustrates the idea of the use of 'procedures' and of giving structure to the program. There is a lot to be said for structuring even simple control programs so that it is clear what is happening. Put the general control operations at the 'top' of the program, and the detailed code to cause these operations to happen in separate sub-routines. This is not the place to write in details about this; books are available on the subject of structured programming and an understanding of the simple concepts of structuring BASIC can be transferred into the machine code control arena.

Finally, on this topic, two 'parameters' are 'passed' to the "output and bit test procedure". the output code is put into the Y Register and the bit test code into the accumulator before the JSR instruction occurs. The sub-routine then operates on the Y Register and accumulator contents.

You now have all the principles for writing a longer control routine. Can you write a program that will take the loco from B3 to B1 at half speed, then to B0 at speed 6, and onto the outer loop at full speed. Finally make the loco go clockwise round the track at speed 3 (for ever!). Presumably you will do this by repeated loading of the Y Register and accumulator followed by jumps to the sub-routine at (2)080.

Can you incorporate the principles of sheet T9 and use the X Register as an index register? Put your control codes at (2)0C0 onwards and load them in pairs into the Y Register and accumulator before jumping to the output and bit-test sub-routine. Think how you are going to leave the loco going eternally round the outside loop.

Bit Setting and Cleaning

Sometimes it is useful to be able to turn on or off one control line (i.e. one bit) of an output Port, without necessarily knowing or bothering to find out what state the other 7 bits are in. For example, if you wanted to change point R from position "1" to position "0" without affecting any other point or the speed output how would you do it (if you couldn't be sure of the states of all the other lines)? The answer is to use the AND instruction.

The AND instruction operates like this:

75 AND B9 is 31 - which may seem a curious result (1)

```
        But 75 is 0111 0101 in binary
            B9 is 1011 1001 in binary
                  :::: ::::
Columns where     :::: ::::
there is 1 at     :::: ::::
the top AND       :::: ::::
the bottom        :::: ::::
are               0011 0001 which is 31 in hexadecimal.
```

See if you can work out:

(1) 32 AND F1 (2) 8C AND 78 (3) F3 AND BF

Check your answers with the follow-up sheet before you go on.

Example 3 leads into the idea of masking. If you have

```
Bit:    7654  3210
        1111  0011   (F3)
        1011  1111   (BF)
          ↓
        1011  0011   All bits in the result except for bit 6
                     will be the same as in the original number
                     because to get a 1 in a result column you
                     must have 1 in the same column of both the
                     numbers you start with.
```

To put it another way, if the number you AND with (the BF in this case) has a single 0 in it when it's converted to binary, it's just like looking at the other 8 bit binary number through a mask with 7 holes cut in it. Each bit of the result will contain a 1 if there was a 1 in the first number at that place, and the result will be 0 if there was a 0 in the first number at that place - but bit 6 will always be zero: you will turn it off.

Don't worry if you can't fully understand yet, you can try out the idea on the equipment and anyway you can use the technique without fully understanding it.

Try this program:

```
(2)000       A9 FF      LDA#FF        Set up for output
(2)002       8D 63 FE   STA FE63                    Port A
(2)005       A0 08      LDY#08        All points to 0
(2)007       8C 61 FE   STY FE61      to Port A (via Y Reg)
(2)00A       A0 14      LDY#14        Delay time to Y Reg
(2)00C       20 80 27   JSR DELAY     and jump sub-routine to do it
(2)00F       A0 A5      LDY#A5        Points/speed code to Y Reg
(2)011       A9 01      LDA#01        Sensor code to accumulator
(2)013       20 80 20   JSR TEST      Jump to output and bit test
                                                      routine
(2)016       AD 61 FE   LDA FE61      Load Port A to accumulator
(2)019       29 DF      AND#DF        AND with DF (bit 5 = 0)
(2)01B       8D 61 FE   STA FE61      Store result back to Port A
(2)01E END   4C 1E 20   JMP END       END
```

Put the loco at B3 and press RESET. It should move out of the siding and when it reaches B0, point Q will move to the '0' position so that the loco can circulate the inner loop.

Finally, what do you do if you want to change one bit from 0 to 1? This time you perform a logical OR. For example 75 is 0111 0101
08 is 0000 1000

Columns shere there is a 1
at the top OR at the
bottom are 0111 1101

So bit 3 has been turned on by the operation OR#08.

Can you think up a simple route for your loco where it will be necessary to change a bit from 0 to 1 when it goes over a sensor?

These new instructions are:

Code	Assembler mnemonic	Name	Description
29 NN	AND#NN	AND accumulator (immediate)	Performs a logical AND between the contents of the accumulator and the next number in the program. The result is left in the accumulator.
09 NN	ORA#NN	OR accumulator (immediate)	Performs a logical OR between the contents of the accumulator and the next number in the program. The result is left in the accumulator.

Next sheet - T11(F)

T11(F) Follow-up to Feedback

The more complicated control program will look something like this:

```
(2)000    -  (2)00E     as before
(2)00F       A0 B4      LDY#B4          Points/speed code to Y Reg.
(2)011       A9 02      LDA#02          Sensor code to accumulator
(2)013       20 80 20   JSR TEST        Jump to output and bit test
                                                routine
(2)016       A0 BE      LDY#BE          next points/speed
(2)018       A9 01      LDA#01          next sensor
(2)01A       20 80 20   JSR TEST
(2)01D       A0 37      LDY#37          next points/speed
(2)01F       A9 04      LDA#04          next sensor
(2)021       20 80 20   JSR TEST
(2)024       A0 7B      LDY#7B          next points/speed
(2)026       A9 00      LDA#00          see explanation below **
(2)028       20 80 20   JSR TEST
(2)02B END   4C 2B 20   JMP END         END

(2)080    -  (2)088     as before
```

If you want to do this using the X Register as an index register, use this program:

```
(2)000    -  (2)00E     as before
(2)00F       A2 00      LDX#00          Index register = 00
(2)011 LOOP  BC C0 20   LDY 20C0,X      Load Y from (table start + X)
(2)014       E8         INX             Increment index register
(2)015       BD C0 20   LDA 20C0,X      Load A from (table start + X)
(2)018       20 80 20   JSR TEST        Jump to output and bit test
                                                routine
(2)01B       E8         INX             Increment index register
(2)01C       4C 11 20   JMP LOOP        Do it all again
```

Table of codes:

```
(2)0C0       B4       First points/speed code
(2)0C1       02       First sensor code
(2)0C2       BE       Second points/speed code
(2)0C3       01              etc
(2)0C4       37
(2)0C5       04
(2)0C6       7B
(2)0C7       00       see explanation below **

(2)080    -  (2)088   as before
```

** The 00 is put in as the final sensor check so that the microprocessor will be permanently 'trapped' in the sub-routine between (2)083 and (2)087. Thus the loco will go round the outer circuit of the track for ever.

o o o o 0 0 0 0 o o o o

```
1   32 AND F1 is        0011 0010
                  AND   1111 0001
       which gives      0011 0000      which is 30 in hex

2   8C AND 78 is        1000 1100
                  AND   0111 1000
       which gives      0000 1000      which is 08 in hex

3   F3 AND BF is        1111 0011
                  AND   1011 1111
       which gives      1011 0011      which is B3 in hex

          o  o  o  o  0  0  0  0  o  o  o  o
```

Perhaps the easiest example of altering one bit of the output code is to change the direction bit (bit 3) when the loco goes over sensor B2. The program might be:

```
(2)000   -  (2)00E     as before
(2)00F      A0 F6      LDY#F6        Points/speed code to Y Reg
(2)011      A9 04      LDA#04        Sensor code to accumulator
(2)013      20 80 20   JSR TEST      Jump to output and bit test
                                                routine
(2)016      AD 61 FE   LDA FE61      Load accumulator from Port A
(2)019      09 08      ORA#08        OR with 08
(2)01B      8D 61 FE   STA FE61      Store result back to Port A
(2)01E END  4C 1E 20   JMP END       End
```

Next Sheet - T12

T12 Using the A-D Converter

The microprocessor is often used to control a process by making small adjustments to an output after it has taken some measurements of, perhaps, temperature, or flow rate, or light intensity, or..... there are many possibilities.

In order to do this an **analogue to digital** converter is used. This unit senses the voltage connected to its input socket and gives an output on its 8 output lines which is the binary code for the input signal. The voltage is what is known as an **analogue** signal, the binary signal to the microprocessor is a **digital** signal. This is where the name 'A-D converter' comes from. ADC is a common abbreviation: the complement of DAC met on sheet T7.

The A-D boxes we use give a binary 1 output for 0.01V input (that is 10 mV). So if you connect 0.01V to the input socket, the output will be 0000 0001.

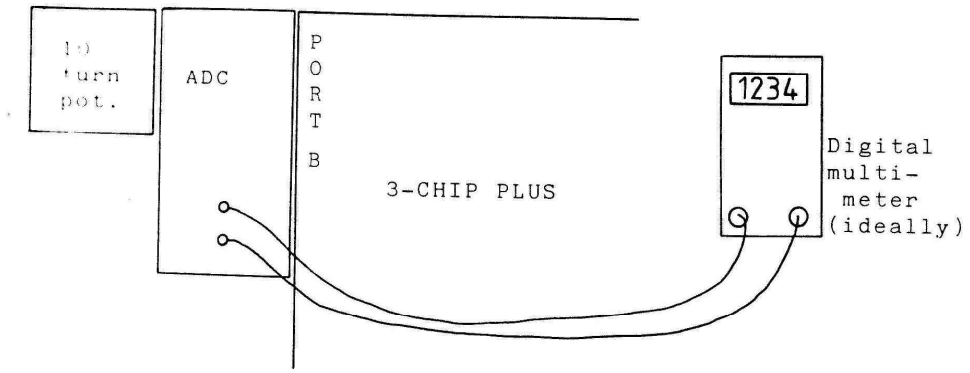
0.07V would give 000 0111 on the output lines (the binary code equals 7 in 'ordinary' counting).

1.27V would give 0111 1111 on the output lines (the binary code equals 127 in 'ordinary' counting).

2.00V would give 1100 1000 on the output lines (the binary code equals 200 in 'ordinary' counting; 200 x 0.01V = 2.00V)

o o o o 0 0 0 0 o o o o

Connect up the following circuit.



If it has one, the multimeter should be on the DC 20V range.

Rotating the potentiometer knob should change the multimeter reading. Set the voltage to 1.00V.

Key the first program on sheet T7 into the A-ROM - the program that reads the input to port B and transfers it as an output to port A. Connect an 8-LED Unit to port A.

Transfer the A-ROM to the 3-Chip Plus board and press RESET. The display should show in binary what the output from the ADC is. If the analogue voltage is 1.00V, which is one hundred lots of 0.01V, the binary output should be one hundred as well - that is 0110 0100. This tells you which LED's should be alight. Small errors are likely - but the display should not be more than two out (that is it will show between 0110 0110 and 0110 0010).

(a) Change the meter reading to 1.42V (with the potentiometer). What does the 8-LED Unit show now? Has the analogue - digital converter done its job correctly?

(b) Try the following voltages. You might like to copy the table and use it to make notes.

Volts	Binary Code (1 binary digit = 0.01V)	Actual reading of 8-LED unit.
0.36		
2.46		
1.87		

Fetch a Temperature Transducer and connect it to the ADC. The voltage output is 0.01V for every degree Celsius (above 0 C). For a typical room temperature of 20 C this is 0.20V.

20 = 0001 0100 so a number something like this should show on the display. If the room temperature is hotter or colder than this the display will show something different.

Warm the sensor with your fingers - what is your body temperature?

(c) If the display showed 0010 1110 what is the temperature of the sensor?

(d) If the display showed 0101 0101 what is the temperature of the sensor?

Sheet T14 shows how to turn the binary data into a more useful format, but first check your results with T12(F)

T12(F) Follow up to A-D Converters

(a) 1.42V = 142 lots of 0.01V

142 = 1000 1110 which is what the 8-LED Unit should display.

(b) 0.36V = 0010 0100 which is what the 8-LED Unit should display.

2.46V = 1111 0110 which is what the 8-LED Unit should display.

1.87V = 1011 1011 which is what the 8-LED Unit should display.

(c) 0010 1110 = 46 in 'ordinary' (decimal) numbers. The temperature must be 46°C therefore.

(d) 0101 0101 = 85 in decimal. The corresponding temperature is 85°C.

Next Sheet - T15

T13 Using RAM

What is RAM first?! The initials stand for Random Access Memory which might not make matters any clearer! It is in fact memory into which data can be written and from which data can be read. The program that controls the micro-processor is stored in Read Only Memory (ROM for short) - you can't write data into a ROM, as the name suggests. One way of remembering what RAM can be used for is to call it Readily Alterable Memory.

A great many jobs that a microprocessor can do will need a certain amount of 'working memory' into which bits of information can be written for use later. Perhaps the easiest way of thinking about it is to consider how you would work out

$$\frac{(36.27 \times 26.93) + (14.97 \times 6.12)}{(143.27 + 678.95)}$$

using a simple 4 function calculator without a memory. You'd need a piece of paper on which to 'store' the result of each bracket in the sum before doing the final working out. You might like to try to see if you can do it without using paper - but remember, don't use the calculator memory! The answer is 1.2993693. The piece of paper you use in a calculation like this is called a 'scratchpad' by our friends on the other side of the Atlantic. It's not surprising, therefore, to find the RAM on a microprocessor control board called 'scratchpad memory', because the job it has to do is very like the piece of paper used with a calculator.

o o o o 0 0 0 0 o o o o

You have already learnt that the two Ports had addresses, FE61 and FE60 respectively. To get an output from a port you had to store the hex code you wanted to the right address.

The procedure is just the same for using the scratchpad RAM but of course the addresses are different. 3-Chip Plus has 768 bytes of RAM on the board. The RAM cells are in two different areas of memory:

from 0000 to 00FF and
from 0200 to 03FF

The first block is called 'zero page RAM' (because the high byte of its address is 00) and we will only use this area because it's simpler and you will hardly ever need more than 256 bytes in a control situation.

The program example is for a simple 'voting machine' to turn on a LED when four or more input switches are on. The program can be changed so that five (or six, or three or...) switches are needed to turn on the LED. You will use an 8 Switch Unit, but obviously if it was a voting machine you'd have 8 single switches on wires.

The main part of the program (which you can put into an A-ROM) is:

```
(2)000        A9 FF      LDA#FF     Set up port A
(2)002        8D 63 FE   STA FE63         as output
(2)005 STRT   A9 08      LDA#08     Put 08
(2)007        85 40      STA Z40           into loop  counter
(2)009        A9 00      LDA#00     Put 00
(2)00B        85 41      STA Z41           into result counter
(2)00D        AD 60 FE   LDA FE60   Load accummulator from Port B
(2)010        85 42      STA Z42    and store it to "temp".
(2)012 LOOP   A5 42      LDA Z42    Load accumulator from "temp"
(2)014        29 01      AND#01    'mask' right hand bit
(2)016        F0 02      BEQ NIL    branch if result is 0
(2)018        E6 41      INC Z41    Increase result counter by 1
(2)01A NIL    46 42      LSR Z42    Shift right "temp"
(2)01C        C6 40      DEC Z40    Decrement loop counter
(2)01E        D0 F2      BNE LOOP   Do again unless loop ctr. = 0
(2)020        A5 41      LDA Z41    Load acc. from result ctr.
(2)022        38         SEC        Set carry
(2)023        E9 04      SBC#04     Subtract 04
(2)025        10 08      BPL LED    Branch if result plus (or zero)
(2)027        A9 00      LDA#00     Otherwise send 00
(2)029        8D 61 FE   STA FE61         to port A
(2)02C        4C 05 20   JMP STRT     and go back to beginning
(2)02F LED    A9 FF      LDA#FF     Send FF
(2)031        8D 61 FE   STA FE61         to port A
(2)034        4C 05 20   JMP STRT     and go back to beginning
```

Connect an 8-LED Unit to port A and an 8-Switch Unit to port B. Transfer the A-ROM to a 3-Chip Plus board. Press RESET then check that the program works. You should find that with 4 or more switches at '1', all the LEDs will come on. Otherwise all LEDs are off. It doesn't matter which 4 switches are on.

o o o o 0 0 0 0 o o o o

The following instructions are new to you; add them to your list:

Code	Assembler mnemonic	Name	Description
85 LL	STA ZLL	Store (zero page)	Stores the contents of the accumulator to address 00LL (in zero page RAM)
A5 LL	LDA ZLL	Load (zero page)	Loads the accumulator with the contents of zero page RAM at 00LL.
46 LL	LSR ZLL	Logical shift right (zero page)	The 8 bits currently stored at zero page address 00LL are shifted right one place (so 0110 1011 would become 0011 0101)
38	SEC	Set carry	The 'carry store' is 'set' (to logic 1). This has to be done before performing a subtraction.

Code	Assembler mnemonic	Name	Description
E9 NN	SBC#NN	Subtract (with carry)	NN is subtracted from the current contents of the accumulator. The carry acts as a 'borrow' if you are doing multi-byte arithmetic. Always **set the carry** before starting a subtraction.
10 XX	BPL XX	Branch if plus	The microprocessor branches on or back (depending on the displacement XX) if the result of the previous operation in the program was positive. (Zero is deemed to be a positive quantity).

NOTE Strictly speaking the mnemonic for zero page addresses does **not** include the Z. So 'Load, zero page' is just LDA LL where LL is the address in zero page (e.g. LDA 4C loads from 004C). Putting the 'Z' in helps clarity but if you find yourself using an assembler (for example on the BBC Microcomputer), the chances are it won't like the Zs. In which case delete them!

o o o o 0 0 0 0 o o o o

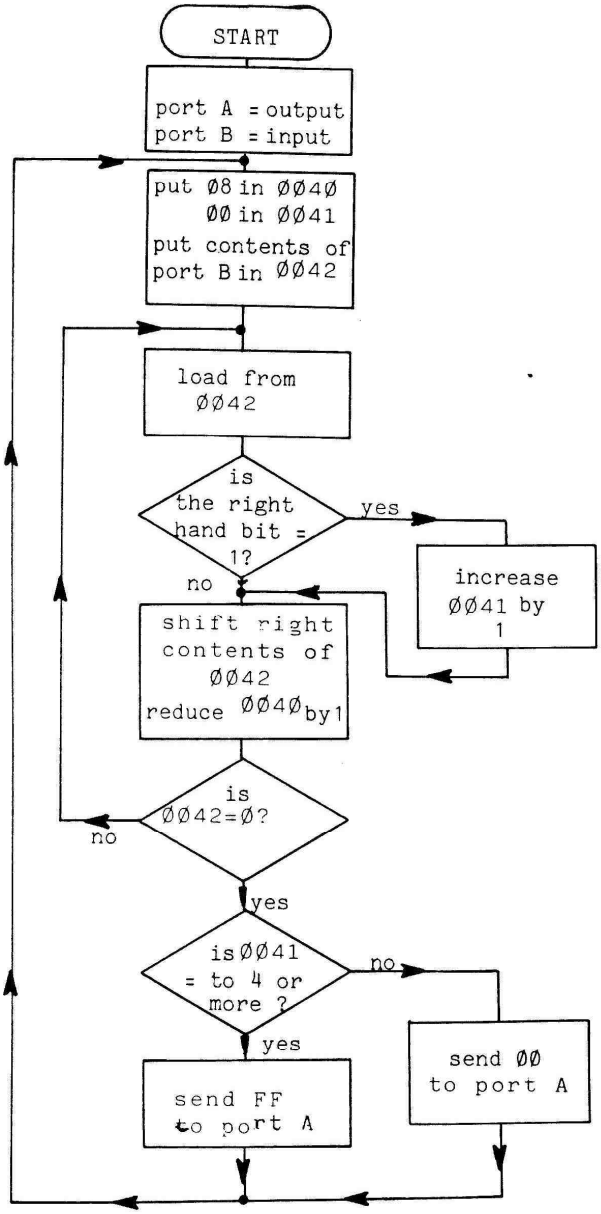
The flowchart for this program is quite complicated, but if you can't see what the program is doing by looking at the list above, the flowchart may help. The important thing to realise is that:

* RAM location 0040 is used to count how may switches have been inspected, and when this is 8 the decision is made to light the LED or not.

* RAM location 0041 is used to count how many switches are down.

* RAM location 0042 is used to store the data taken in from port B as it is shifted right a stage at a time.

o o o o 0 0 0 0 o o o o



If you think you have got the idea of how the program works you might like to try changing it to make it light the LED when 5 or more switches are on.

Can you change it so that if 4 or more switches are down LED 7 lights, and if 3 or less switches are down LED 3 lights?

Try modifying the program so that you use different RAM locations to store the data. You might use 00AC (instead of 0040) to store how many switches have been inspected, 0097 (instead of 0041) to count how many switches are down, and 0001 (instead of 0042) to store the data from port B as it is shifted right.

Next Sheet - T13(F)

T13(F) Follow up to using RAM

1 To get the program to check for 5 or more lamps you change the instruction at (2)023/4 to E9 05. this subtracts 5 from the number of switches that have been counted as down before the 'branch if plus' test is applied.

2 Leave (2)023/4 as E9 04. Change instruction (2)027/8 to A9 08. This will turn on LED 3 if too few switches are down. Remember that the LEDs number from 0 to 7 (not from 1 to 8) so LED 3 is the fourth in the line! Change (2)030 to 80 which will turn on LED 7 alone if enough switches are down.

3 This is a bit more tricky. A list of program lines that must be changed is:

```
(2)007   85 AC
(2)00B   85 97
(2)010   85 01
(2)012   A5 01
(2)018   E6 97
(2)01A   46 01
(2)01C   C6 AC
(2)020   A5 97
```

All the rest are left unchanged.

* * * * * * *

As well as meeting RAM for the first time on this sheet, you have also gone on to the next step in controlling - that is using the intelligence of the microprocessor to allow decisions to be made. To some extent the second railway control sheet showed this, but this sheet involves a slightly more sophisticated piece of decision-making. The microprocessor checked each of eight switches in turn, then made a decision as to whether or not to turn on a LED according to how many switches were closed. The microprocessor is not just a controller but an intelligent controller.

The second thing that you may have realised is how easy it was to change the number of switches that were needed to operate the LED. The alteration of a single program code achieved this. A circuit to turn on a LED when 4 switches out of 8 are down could be made out of separate integrated circuits. It would take a long time to wire up, and if you wanted to change it so that 5 switches were needed to turn on the LED, you'd have to virtually start again with your design and wiring up. The flexibility, versatility and ease of mass production of microcontrollers will lead to their widespread use in old and new control situations.

Next Sheet - T14

T14 Using the Microprocessor to Operate on Data

This learning sheet follows on from the last and shows how arithmetic can be done on data sent to the microcomputer, and how outputs can be changed according to the data. The design of a digital thermometer is considered; you have a working system when you have finished the sheet. It is relatively easy to build in a thermostat function so that the micro can keep the temperature constant.

o o o o 0 0 0 0 o o o o

If we want to give a readout of temperature (to port A) we need to:

(a) perform an analogue to digital conversation.

(b) convert the binary data to data for two decimal digits

(c) send it to port A

(d) delay for one second

(e) repeat the process

It is a bit complicated to explain exactly what the program is doing, but notes are written to tell you what each bit of the program is doing.

(1) Fetch a 3-Chip Plus board and connect a dual 7 segment LED Unit to port A, and an ADC to port B

(2) A temperature sensor should be connected to the A-D converter

Key the program into an A-ROM, transfer it to the 3-Chip Plus board and press RESET - the display will tell you what the temperature of the sensor is (in degrees Celsius). At first it will show room temperature (about 20°C). Warm it in your hand and the display will change. Body temperature is about 37°C which is just outside the range of the thermometer system. You could immerse it in a jar of warm water and compare the electronic thermometer with a mercury-in-glass thermometer, or even put it in boiling water.

o o o o 0 0 0 0 o o o o

Digital thermometer program:

```
(2)000        A9 FF       LDA#FF      Set up as output
(2)002        8D 63 FE    STA FE63              Port A
(2)005 REPT   A9 00       LDA#00      Put 00 into
(2)007        85 21       STA Z21       temp store (1)
(2)009        AD 60 FE    LDA FE60    Load Port B
(2)00C        F0 10       BEQ OUT     Quit if it's zero
(2)00E        85 22       STA Z22     Otherwise put into temp(2)
(2)010        18          CLC         Clear the carry
(2)011        F8          SED         Set decimal mode
(2)012 MORE   A5 21       LDA Z21     Load temp (1)
(2)014        69 01       ADC#01      Add 1 to it
(2)016        B0 13       BCS MUCH    Look for overload condition
(2)018        85 21       STA Z21     Not, so put back in temp (1)
(2)01A        C6 22       DEC Z22     Subtract 1 from temp (2)
(2)01C        D0 F4       BNE MORE    Repeat if not finished
(2)01E OUT    A5 21       LDA Z21     Load temp (1): the result
(2)020        8D 61 FE.   STA FE61    Send to output port
(2)023        A0 14       LDY#14      Delay period = 1s
(2)025        20 80 27    JSR DELAY   execute delay
(2)028        4C 05 20    JMP REPT    and do it all again
(2)02B MUCH   A9 FF       LDA#FF      Result more than 99 so
(2)02D        8D 61 FE    STA FE61    send FF to output port
(2)030        A0 14       LDY#14      Delay period = 1s
(2)032        20 80 27    JSR DELAY   execute delay
(2)035        4C 05 20    JMP REPT    and do it all again
```

You will have noticed that the system seemed to delay before responding to any switch changes you made. Delays are put into the program to stop the display flickering all the time. You can easily change the delay period and observe the effect.

Clearly with a two-digit display and an input that oculd reach 255 it is necessary to indicate an overload condition (input greater than decimal 99). Sending FF to the port blanks this particular display, and the program harnesses this facility - the display is switched off when a false reading would otherwise be given (for inputs of decimal 100 and above).

The purpose of this sheet is to demonstrate microprocessor control involving the collection of data, the processing of it and control with decision making. These are the basic ingredients of most more sophisticated microcontrolled systems. If you are interested in the new instructions that have appeared and how the program works, look at the follow-up sheet T14(F).

o o o o 0 0 0 0 o o o o

It is quite easy to turn the system into a thermostat as well. By the time the microprocessor has reached 01E or 02B the temperature is stored in Z21. You need to write a bit more program that will test whether the temperature is greater than or less than some desired value. You would then need to switch on or off a heater to keep the termperature constant. How could this be done when all 8 bits of both ports A and B are tied up? It is possible to use the CA2 and CB2 lines from the 6522 v.i.a. chip as additional outputs, and these could switch a relay unit connected to the heater. The principle of feedback control of an analogue quantity is important, but the connecting up and the programming are a bit too tricky for this series of sheets.

Next sheet - T14(F)

T14(F) Follow-up to Processing Data

The new instructions were:

Code	Assembler mnemonic	Name	Description
18	CLC	Clear carry	The 'carry flag' is made 0. It is used when performing additions of numbers which occupy more than one byte.
F8	SED	Set decimal mode	All additions and subtractions can be made to happen in binary-coded-decimal by inserting this instruction.
69 NN	ADC#NN	Add with carry (immediate)	Add the next number in the program to the contents of the accumulator + the carry flag. The result is left in the accumulator.
B0 XX	BCS XX	Branch if carry set	A branch happens only if the carry bit = 1
C6 LL	DEC ZLL	Decrement (zero page)	As other decrement instructions but for zero page memory locations.
	o o o o 0 0 0 0 o o o o		

The flowchart for the program which may help you to understand it if you wish is: