UNIVERSITY OF YORK

Department of Computer Science

Microcomputer Project 1991 — Notes

1. The digital time code

The transmitter known as MSF, located at Rugby, transmits a 60 kHz carrier 24 hours per day, except for a weekly maintenance period each Tuesday between 1000 and 1400. Time information is conveyed digitally, by means of short interruptions to the carrier. The carrier is interrupted for 100 ms or more each second; the start of the interruption marks the exact second "tick". The minute "tick" is 500 ms long, and contains a *fast code* which identifies the minute which has just begun. The format of the fast code is shown in figure 1. Note, however, that the receiver is not designed to receive the fast code, and does not receive the data reliably; consequently, you should not try to use the information in the fast code. Instead, you should use the *slow code*. The slow code indicates what the time will be at the start of the *next* minute. This information is conveyed in binary-coded decimal, at a data rate of one bit per second (so it really is slow!), using a 100 ms interruption to signal a "0" and a 200 ms interruption to signal a "1", as shown in figure 2.

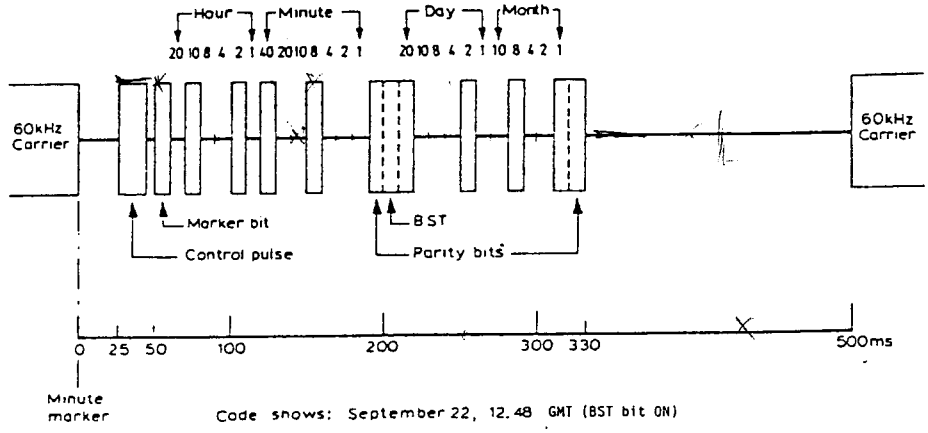

Figure 1. The fast time code inserted within the minute pulse

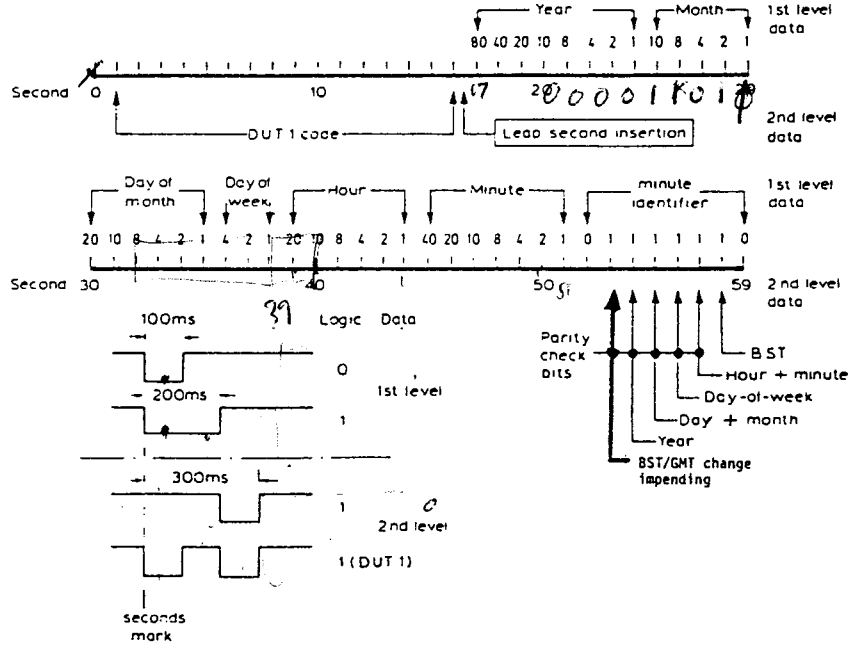

Figure 2. The slow time code

The slow code contains some information, e.g. the DUT1 code and the year, month, day of month and day of week, which you will not need for your speaking clock. The clock needs only to be able to speak the time of day; it need not give the date as well, although you may of course include extra information if you wish.

The signal which is distributed to the bench positions represents "carrier" by +6 V and "interruption" (i.e. no carrier) by –6 V. The time code is accurate to ±2 parts in 10^{12} (1 s in about 16000 years) at the transmitter, but the signal takes about 1 ms to travel from Rugby to York. You should be able to achieve an overall accuracy considerably better than that of the BBC "time pips", which are specified to only ±100 ms.

MSF is only one of many time signal stations. The Swiss station HBG is, as one might guess, the most accurate, sharing first place with Germany's DCF77. Their signals are accurate to ±5 parts in 10^{13} (1 s in more than 63000 years). The time code broadcast by the French station TDF Allouis has a special "14th July" bit.

2. Speech synthesis

2.1. Phonemes and allophones

Any spoken word can be divided into a sequence of distinct sounds called *phonemes*. The boundary between two adjacent phonemes in a word is always recognizable and distinct. For example, the word *cuckoo* contains three phonemes: KK, UH, KK, UW. The word *key* contains two phonemes, which are called KK and IH. If you listen carefully, however, you will notice that the initial sounds of *cuckoo* and of *key*, while both "k" sounds, are in fact quite different. That is because the UH is a "back vowel" and IH is a "front vowel". The terms *front* and *back* refer to the position of the tongue while the vowel is being sounded. The tongue position influences the sound of the preceding consonant "k", to the extent that an attempt to pronounce *key* with the type of "k" that starts the word *cuckoo* results in, at best, an odd and un-natural pronunciation. (If you try it, you might decide that this pronunciation is physically impossible with a human voice-producing apparatus.)

Linguists say that the phoneme KK possesses three *allophones*, i.e. three different pronunciations depending on the context. The allophone KK1 is used before front vowels (e.g. in *key*), while KK3 is used before back vowels (e.g. in *cuckoo*). KK2 is used at the end of words. The sequence of allophones which makes up *cuckoo* is KK3 UH KK3 UW2. There are 64 different allophones in English speech, and any English word can be pronounced by stringing together an appropriate sequence of allophones.

Figure 3 lists all 64 allophones, and guidelines for using them.

2.2. The sampled allophone data

There is at least one IC, the SP0256–AL2, which can synthesize any allophone used in English (actually American) speech. When this IC was introduced, in the early '70s, it was widely used. Nowadays, however, it is considerably cheaper to store the allophones as digital samples in an eprom, and to synthesize speech by simply reading the samples from the eprom and sending them to a DAC (digital-to-analogue converter). This method is also much more flexible. It is possible, for example, to have a "sex" pin on the eprom which is a high-order address pin. By setting the sex pin high or low, one could select a male or female voice. Similarly, one could store several regional accents in a large eprom.

The use of stored sampled allophone data is now the most sensible engineering choice in this application. Besides this, I was anxious when designing this project to give you an opportunity to learn about sampled audio techniques, which are becoming more and more important as memory becomes cheaper and processors become faster; and the implementation of a sampled audio system is an essential part of the project. For all these reasons, the use of the supplied allophone data is *required* in this project. You may not use the SP0256 chip. The chip is in fact difficult to obtain—it is probably not being made any more—and it is consequently becoming more and more expensive, probably for the reasons I have outlined above.

There are other methods of speech synthesis, e.g. linear predictive coding, and synthesis from sampled words or phrases. These all require too much processor power, or too much memory, or too much implementation effort to be suitable for this project.

The Allophones and Guidelines for Using Them

Silence

0	PA1	(10 ms)	— before BB, DD, GG, and JH
1	PA2	(30 ms)	— before BB, DD, GG, and JH
2	PA3	(50 ms)	— before PP, TT, KK, and CH, and between words
3	PA4	(100 ms)	— between clauses and sentences
4	PA5	(200 ms)	— between clauses and sentences

Short Vowels

5	*IH	/ ɪ /	— *sitting*, stranded
6	*EH	/ ɛ /	— *extent*, gentlemen
7	*AE	/ a /	— extr*a*ct, *a*cting
8	*UH	/ ʊ /	— c*oo*kie, f*u*ll
9	*AO	/ ɔ /	— t*a*lking, s*o*ng
A	*AX	/ ə /	— l*a*pel, instr*u*ct
B	*AA	/ ɑ /	— p*o*ttery, c*o*tton

Long Vowels

C	IY	/ i /	— tre*a*t, pe*o*ple, penn*y*
D	EY	/ e /	— gre*a*t, st*a*tement, tr*ay*
E	AY	/ ɑɪ /	— k*i*te, sk*y*, m*i*ghty
F	OY	/ ɔɪ /	— n*oi*se, t*oy*, v*oi*ce
10	UW1	/ u /	— after clusters with YY: comp*u*ter
11	UW2	/ u /	— in monosyllabic words: t*wo*, f*oo*d
12	OW	/ o /	— z*o*ne, cl*o*se, sn*ow*
13	AW	/ aʊ /	— s*ou*nd, m*ou*se, d*ow*n

R-Colored Vowels

14	ER1	/ əɹ /	— lett*er*, furnit*ure*, interr*u*pt
15	ER2	/ əɹ /	— monosyllables: b*ir*d, f*er*n, b*ur*n
16	OR	/ ɔɹ /	— f*or*tune, ad*or*n, st*or*e
17	AR	/ ɑɹ /	— f*ar*m, al*ar*m, g*ar*ment
18	YR	/ iɹ /	— h*ear*, *ear*ring, *ir*responsible
19	XR	/ ɛɹ /	— h*air*, decl*are*, st*are*

Voiceless Stops

1A	PP	/ p /	— *p*leasure, am*p*le, tri*p*
1B	TT1	/ t /	— final clusters before SS: tes*ts*, i*ts*
1C	TT2	/ t /	— all other positions: *t*est, s*t*ree*t*
1D	KK1	/ k /	— before front vowels: YR, IY, IH, EY, EH, XR, AY, AE, ER, AX; initial clusters: *c*ute, *cl*own, *scr*eam
1E	KK2	/ k /	— final position: spea*k*; final clusters: ta*sk*
1F	KK3	/ k /	— before back vowels: UW, UH, OW, OY, OR, AR, AO; initial clusters: *cr*ane, *qu*ick, *cl*own, *scr*eam

Affricates

20	CH	/ tʃ /	— *ch*urch, fea*t*ure
21	JH	/ dʒ /	— *j*udge, in*j*ure

Nasal

22	MM	/ m /	— *m*ilk, alar*m*, a*m*ple
23	NN1	/ n /	— before front and central vowels: YR, IY, IH, EY, EH, XR, AE, ER, AX, AW, AY, UW; final clusters: ear*n*
24	NN2	/ n /	— before back vowels: UH, OW, OY, OR, AR, AA
25	NG	/ ŋ /	— stri*ng*, a*n*ger

Resonants

26	WW	/ w /	— *w*e, *w*arrant, ling*u*ist
27	RR1	/ r /	— initial position: *r*ead, *wr*ite, x-*r*ay
28	RR2	/ r /	— initial clusters: b*r*own, c*r*ane, g*r*ease
29	LL	/ l /	— *l*ike, he*ll*o, stee*l*
2A	EL	/ əl /	— lit*tle*, ang*le*, gent*lemen*
2B	YY1	/ j /	— clusters: c*u*te, b*eau*ty, comp*u*ter
2C	YY2	/ j /	— initial position: *y*es, *y*arn, *y*o-*y*o

Voiced Fricatives

2D	VV	/ v /	— e*v*est, pro*v*e, e*v*en
2E	DH1	/ ð /	— word-initial position: *th*is, *th*en, *th*ey
2F	DH2	/ ð /	— word-final and between vowels: ba*th*e, ba*th*ing
30	ZZ	/ z /	— *z*oo, phase
31	ZH	/ ʒ /	— bei*g*e, plea*s*ure

Voiceless Fricatives

32	*FF	/ f /	—
33	*TH	/ θ /	— may be doubled for initial position and used singly in final position
34	*SS	/ s /	—
35	SH	/ ʃ /	— *sh*irt, lea*sh*, na*t*ion
36	HH1	/ h /	— before front vowels: YR, IY, IH, EY, EH, XR, AE
37	HH2	/ h /	— before back vowels: UW, UH, OW, OY, AO, OR, AR
38	WH	/ wh /	— *wh*ite, *wh*im, *tw*enty

Voiced Stops

39	BB1	/ b /	— final position: ri*b*; between vowels: fi*bb*er; in clusters: *bl*eed, *br*own
3A	BB2	/ b /	— initial position before a vowel: *b*east
3B	DD1	/ d /	— final position: play*ed*, en*d*
3C	DD2	/ d /	— initial position: *d*own; clusters: *dr*ain
3D	GG1	/ g /	— before high front vowels: YR, IY, IH, EY, EH, XR
3E	GG2	/ g /	— before high back vowels: UW, UH, OW, OY, AX; before clusters: *g*reen, *gl*ue
3F	GG3	/ g /	— before low vowels: AE, AW, AY, AR, AA, AO, OR, ER; before medial clusters: anger; and final position: peg

* These allophones can be doubled.

Figure 3. The allophones and guidelines for using them

2.3. Data compression

The allophone data were recorded by pronouncing each allophone and recording the instantaneous amplitude of the sound waveform at intervals of 100 μs. By "playing back" these samples at the same rate, *viz.* 10000 samples/s, the allophone can be re-created exactly. The samples were recorded as 7-bit unsigned binary values.

The complete set of allophone data contains 78641 samples. If these were stored one sample to a byte, a 128 kbyte eprom would be required. These are expensive; and there is also the problem that a Z80 cannot directly address such a large amount of memory. By storing 7-bit samples rather than 8-bit bytes, only 68811 bytes are required; but a 128 kbyte eprom is still required. (There is no size of eprom between 64 kbytes and 128 kbytes.) Because of the nature of the speech signals, however, most samples are close in value to the previous one. It turns out that no two successive samples differ by more than ±30. By storing, not the samples themselves, but the 6-bit *differences* between successive samples, only 58981 bytes are required, permitting the use of a 64 kbyte eprom. Let us call these differences between successive sample values *deltas*.

The file is shrunk by a further factor of 1.93 by employing *Huffman coding*. A Huffman code is a 1:1 mapping from one set of values (the *input set*) onto another set of values (the *output set*). In this application, the members of the input set are the 6-bit deltas, and the members of the output set are variable-length values which we shall call *bitcodes*. The mapping between deltas and bitcodes is fixed by me for this project, and is documented in the file `/usr/course/mcp/hufftable` (see Appendix).

The deltas are not equally common in the file; for example, zero occurs very often (i.e. two successive samples are often the same), but –29 occurs only once in the whole file. The principle behind the Huffman code is that short bitcodes are allocated to the common deltas, and longer bitcodes to the less common deltas. For example, the very common delta 0 is represented by the two-bit code 11, whereas the very uncommon delta –29 is represented by the 14-bit code 10010010100001. It is by these means that the compression is achieved.

The Huffman-compressed allophone data are in `/usr/course/mcp/allodata.s`. This is an assembly-language file, which can be assembled by `as80`. The file starts with an "index" containing 64 16-bit pointers. Each pointer points to a sequence of `.byte` directives which contain the Huffman-encoded data for one of the 64 allophones.

The file `allodata.s` represents the encoded data as a sequence of bytes. To see how the Huffman code works, you have to imagine the file as a sequence of bits. Successive bits in a byte are stored starting from the bottom (least significant) end of the byte. For example, the first seven bytes (in octal) of the allophone OY are

```
232     240     213     110     110     324     251
```

as you can verify by inspecting the file `allodata.s`. The sequence of 56 bits, reading the bits in each byte from right to left, is

```
01011001 00000101 11010001 00010010 00010010 00101011 10010101
```

These bits are split into Huffman codes as follows:

```
0101100 10000 0101110 10001000 10010000 10010001 0101110 010101
```

These codes correspond to the following deltas (see `/usr/course/mcp/hufftable`):

```
-8      3       8       9       10      11      8       5
```

and the successive sample values (7-bit integers in the range –64 . . +63, to be output to the DAC) are therefore

```
-8      -5      3       12      22      33      41      46
```

A Huffman code has the property that no bitcode is an initial sub-word of any other bitcode. For example, since 11 is the bitcode for the delta 0, you can be sure that no other bitcode starts with 11. The decoding

routine simply reads successive bits from the bit stream (e.g. from an eprom) until it has accumulated a complete bitcode, after which it outputs the corresponding delta. There is no possibility that the bitcode it has read is the initial section of a longer bitcode.

To aid processing, the coded samples (i.e. the sequence of bitcodes) for each allophone begin on a byte boundary in the file, and the sequence is terminated by the bitcode 010000111 (which acts as a "stopper"). Note that some deltas do not occur at all in the file, but they still have (long) bitcodes allocated to them. Note also that every possible sequence of bits can be unambiguously decoded into a sequence of deltas, i.e. there are no missing bitcodes.

Further information on Huffman codes can be found in most books on data structures.

There are at least three approaches to decoding the Huffman code. One is to load the allophone data into a 32 kbyte eprom, and to decode the Huffman code by software in real time, i.e. at a rate of 10000 samples/s. This requires some "tricky" coding, but it can be done; my demonstration clock works this way. To facilitate this approach, the particular code used has been cunningly chosen so that no bitcode which actually occurs in the data is longer than 14 bits. A Huffman code chosen to give the best possible compression would have several bitcodes longer than this, but it is difficult to decode these long bitcodes in the 100 µs available. A second approach is to perform the decoding off-line by a high-level pre-processing program, and to store the expanded data in a large eprom. This is probably the simplest method, but it does require a large (64 kbyte) eprom, and is probably not a good solution from an engineering point of view. The third approach is to use hardware support to speed up the decoding of the bitcodes.

The format of the file `allodata.s` might not be suitable for your own use. You are free to "massage" the data in (your own copy of) this file in any way you wish, either by editing the file with a standard editor, or by running a specially-written program. For example, you might decide that you would like to store a byte count as well as a pointer in the index, or that you you would like to reverse the order of bits in a byte. This is quite acceptable.

3. Design, development and testing

3.1. Hardware design

The MCP course is largely about the integration of hardware and software. I hope you find this activity interesting and rewarding. The integration of hardware and software components of a piece of equipment is an important and valued skill. Although in the computer industry you will find people who are hardware specialists and other people who are software specialists, you can be sure that in any well-managed project there will be an interchange of ideas between the "software people" and the "hardware people", and people who can talk both languages will be highly regarded. Besides this, there are all sorts of compromises which have to be made in the design of a piece of equipment. You can always make the hardware simpler at the expense of more complicated software, and *vice versa*. Judging the balance between hardware and software is an important skill which you will practise by doing this course.

You may use any reasonably available components in your design. A selection of data sheets and data books is available in X/D005. The prices of the components are also given. You should consider the economic aspects of the project as well as the technical aspects; you should not use a complicated and expensive chip if a simpler and cheaper one would do as well. This consideration is a part of "good engineering practice", for which marks will be awarded. You are not restricted to those devices for which I have provided data sheets. If you would like to use a different component in your circuit, ask me, and if I think your request is reasonable, I shall obtain a data sheet for you. There is plenty of scope for inventiveness. Note that your circuit design *must be approved* by me before you ask Gurmit to obtain the parts.

Many of the components which you will use are CMOS devices. These have delicate constitutions, and can easily succumb to even moderate electrostatic fields which sturdier chips would hardly notice. Be kind to them; earth yourself by touching the earth terminal of your power supply before touching the pins, and don't walk across the room with a CMOS IC in your hand. Be particularly careful with the eproms when you remove them for programming.

To obtain a good quality sound, it is essential that the values are written to the DAC at precisely regular intervals. In particular, you should realize that you will not obtain adequate results if the DAC is connected directly to an output port, because of the unavoidable jitter (timing variations) caused by the execution of your program. Some hardware buffering will be required. Here, as elsewhere, you should think very carefully about how much you should try to achieve in hardware, and how much in software. You can vary the pitch of the speech by altering the rate at which samples are output; sample rates in the range 7 . . 11 kHz seem to work well.

Although most of your design work will be digital, some analogue design will be necessary. Don't be afraid of this! If you have completed the first-year ECS course, you will have no trouble with the analogue part of the project; and if you have not done ECS, you will learn all you need to know by reading these notes and the MCP Data Book (especially the *Basic Op-amp Circuits*), and (if necessary) *The Art of Electronics* by Horowitz and Hill (C.U.P., 1980) or any other elementary electronics text-book. The amount of analogue electronics required varies from year to year; this year you will be relieved to learn that the analogue component of the design is limited to the design of the loudspeaker amplifier (which you can lift straight from an appropriate data sheet), with perhaps an op-amp buffer or two (which you can lift from *Basic Op-amp Circuits*).

The voltage levels of the time code signal are ±6 V. Although these are standard RS232 levels, a standard RS232 line receiver cannot be used, because in operation there will be up to 45 clocks connected to a single transmitter, and a standard line receiver would draw too much current. For correct operation, the input impedance of your input stage must be at least 50 kΩ. This is very important; if you disregard this rule you might spoil the signal for other people. Don't connect RS232-level signals to chips which expect +5 V inputs!

In principle, hardware and software design should proceed in parallel. In practice, the design of the hardware tends to lead (precede) the design of the software, although the two components interact very closely. I have asked for "surgery sessions" to be timetabled for the early part of the term, and I expect that the first few of these will be devoted to the sorting out of hardware problems. It is important that you come to these early surgery sessions. When you have completed the preliminary hardware design and you have a provisional circuit diagram, *bring it along to each surgery session.*

The surgeries will continue for as long as they appear to be required. If the attendance diminishes, I shall cancel one or both of the weekly sessions.

3.2. Software design

Remember that the principles of structured programming apply as much to assembly language programming as to high-level language programming. It is hard to write readable assembly-language programs, but it can be done. Marks will be awarded for good programming style.

Most students' MCP programs turn out to be much too long (if they work), or much too short (if they don't work). The program in my demonstration cuckoo clock is 651 lines long, excluding comment lines. My program handles the cuckoo and the digital display, which are not required in your clock; furthermore, I have tended to use software methods rather than hardware methods in my design. So, I would expect your program to be rather shorter than 651 lines.

More than 20% of the lines in my program are either comment lines (i.e. comments by themselves, not attached to an instruction), or blank lines inserted to improve legibility. In addition, almost every instruction has a comment attached to it on the same line. Comments are important in assembly-language programs.

Radio is not an error-free medium. Transmission errors do occur, and must be handled gracefully. Include plenty of error checking in your program. Does your clock keep reasonably accurate time, even when the signal disappears (as it does each Tuesday, for a few hours)? Does it recover gracefully when the signal is restored?

I have not specified exactly the format of the message your clock should speak when you press the button. To be useful, the message must say, in effect, "At the <tone/stroke/cuckoo/etc.> it will be <time>", and then give an identifying tone. Messages which say what the time *was*, e.g. "<beep> ... that was <time>", are much less useful, and are discouraged.

Do not rush to use the logic analyser to attempt to solve your programming problems. If your LCD module and its associated routines from the CTS course are working, you can use them to display debugging information. Alternatively, you could use a UART or even a single bit on a parallel output port to send debugging information to the terminal. If you use either of these techniques, you can insert debugging messages in your code, in much the same way as you would debug Ada code. There is seldom any excuse for scratching your head and asking yourself "I wonder what's going on"—you have the tools to find out!

3.3. Testing

Don't expect to write the final program "straight off"! Most of the programs you write will test some aspect of the hardware. The first programs should test the part of the circuit that you built for CTS. Run some of your CTS programs, and make sure that they still work. The nature of the subsequent test programs will depend on the particular hardware design you have chosen, but there are a few general principles which it is worth pointing out. Don't test too many parts of the hardware at once. As far as possible, in your test programs, aim to test a single aspect only of the design. Don't assume that the hardware works until you have tested it. Remember that hardware can be tested by means of the oscilloscope and the logic probe as well as by writing programs which exercise it. In the later stages of testing, it will sometimes not be apparent whether it is your hardware or your software which is faulty. Remember, too, that ICs are sometimes faulty. This is quite rare. It is nearly always your fault.

Besides the oscilloscopes and the logic probes, logic analysers will be available. These are powerful diagnostic tools which are useful if you have an unusually tough hardware or software problem. It is quite easy to persuade yourself that your problem is sufficiently tough to warrant the use of the logic analyser. It usually isn't. The logic analyser generates large quantities of data, many of which are useless for the task in hand, and it is easy to waste time in separating the wheat from the chaff. Carry out the simple tests first. Are the levels on each pin of each IC correct? Test them with the logic probe. Is the clock signal present and correct? Check it with the oscilloscope. Is there noise on the power rail? Check with the oscilloscope, and add extra decoupling capacitors if necessary. A logic analyser is no substitute for clear thinking and methodical testing.

4. Writing up

Write legibly, concisely and grammatically. Make sure that the pages of your report cannot become detached from one another. A treasury tag is the best way of fastening loose pages. String and paper-clips are the worst. A "black-and-red" notebook is preferred to loose pages. These considerations apart, I do not mind in what format you submit your report, as long as you adhere to the Department's requirements, which are set out on p. 12 of the Handbook (1990 edition).

Many reports have in past years been far too long. Please don't include trivial or irrelevant material. I do not want to know what colour wire you used to wire up the circuit. I do not want to be told how to log in to a Unix system. I shall have several *feet* of reports to mark in March, so I shall welcome conciseness. A clear statement of what is required is given in the formal examination paper (attached); if your report covers all of these areas clearly and well, you will achieve full marks. As stated in the examination paper, marks will be deducted for over-long reports.

On a more positive note, I shall give particular weight to *reasoned arguments* in favour of a particular implementation method (hardware or software). This course differs from most courses which you have taken hitherto in that you yourself must plan the path you take. You must make your own decisions concerning the design of the hardware and software. If you ask my advice on a particular matter, whether to choose this or that implementation method, I am quite likely to invite you to choose for yourself. It's all part of the course. I hope you will enjoy this freedom, and that you will find the course as a whole interesting and enjoyable.

5. Questions, comments, etc.

Questions about the academic content of this project should be addressed to me. Questions about the equipment should be addressed to Gurmit Singh.

Acknowledgements

The demonstration cuckoo clock was built in the Department's workshops by Richard Jennison. The allophone samples were recorded by Anthony Moulds (who does not really sound like a drunken Dalek).

A.J. Fisher
15th November 1990

Appendix — /usr/course/mcp/hufftable

delta	freq	Huffman code
-30	0	1001001010000000
-29	1	100100101000001
-28	0	1001001010000001
-27	1	10001010010010
-26	2	1001001010001
-25	3	1001001010110
-24	1	10001010010011
-23	8	100010100101
-22	11	01000010110
-21	11	01000010111
-20	11	01000011000
-19	21	10001010011
-18	32	1001001000
-17	51	010000000
-16	34	1001001011
-15	54	010000001
-14	78	100100110
14	61	010000010
15	55	010000011
16	55	010000100
17	36	1000101000
18	26	1001001001
19	23	0100001010
20	7	100100101010
21	10	01000011001
22	8	01000011010
23	5	100100101001
24	5	010000110110
25	2	0100001101110
26	3	1001001010111
27	2	0100001101111
28	2	1000101001000
29	0	1001001010000010
30	0	1001001010000011
31	64	010000111