Software/Firmware developments—1

SC/MP gets Tiny-BASIC

If you've tended to scorn microcomputer systems because of the need to program in machine or assembly language, think again. National Semiconductor is releasing shortly a Tiny-BASIC interpreter for its SC/MP systems. It will be sold resident in 4k-bytes of PROM, on a PC card. Plug in the card, and you can have programs running in Tiny-BASIC within the hour!

by JAMIESON ROWE

Although microcomputers haven't been around for very long as yet, already many of the people using them have begun to show interest in the possibility of programming in one of the problem-orientated or "higher" languages like FORTRAN or BASIC. No doubt these thoughts tend to be generated most often when one is slogging through a program written in hexadecimal code!

Of course in order to be able to run programs written in a problem-orientated language, one needs to have them translated into language which the computer can understand. At present this must be done in one of two ways.

One is to use a compiler, which is itself a computer program. When fed into the machine, the compiler will take your "source code" program as written, and produce from it an "object code" equivalent in machine language. It is this version which you then feed into the machine to get your program running.

The other way is to use an interpreter, which again is also a computer program. But unlike a compiler, an interpreter doesn't produce a separate object code version of your program. Instead it must be put in the machine alongside your program, which it interprets into machine language and executes all at the same time.

Which approach is used depends mainly on whether you have a compiler or an interpreter program available, although an interpreter tends to be a little more convenient because it allows faster program development.

The problem with either approach is that both compilers and interpreters tend to be rather long. They are rather tricky and tedious to write, and when written they need quite a deal of computer memory. The latter is especially true with interpreters, which are loaded into memory along with the source program.

Because of the requirement for a fairly large memory, until recently languages like FORTRAN and BASIC have mainly been available only on larger computers.

In the last couple of years, however, a number of people working with various microcomputer systems have come up with interpreters capable of fitting into small systems, and which provide a limited sub-set of the full original BASIC language. Naturally enough the language provided by these interpreters has quickly become known as "Tiny-BASIC".

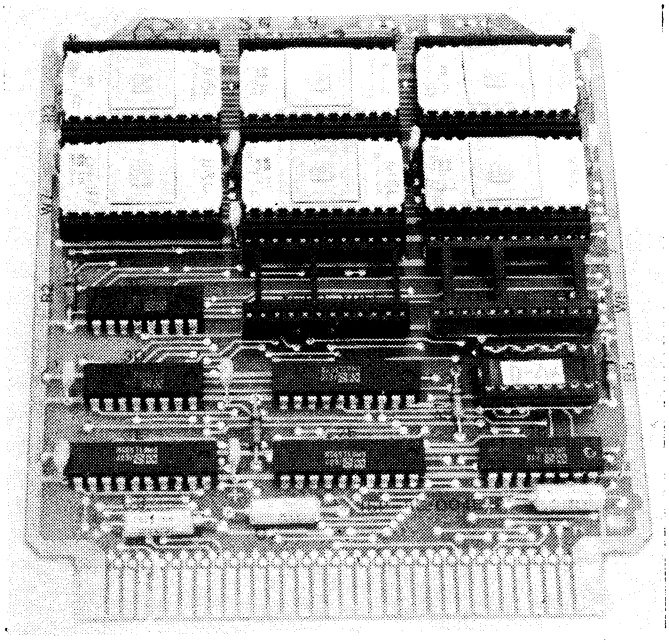
Now National Semiconductor has announced that a Tiny-BASIC interpreter will very shortly become available for its SC/MP microprocessor systems. National's Australian subsidiary, NS Electronics, expects to have it available by late January. It will be sold as a product package which comprises the interpreter itself resident in 4k of PROM on a plug-in PC card, a 2k static RAM card for the user's source program, and a user's manual. The complete package should sell for around $305 plus tax, where applicable. Those who already have SC/MP systems with at least 2k of RAM may also be able to buy just the interpreter and the manual, if they wish, by quoting the serial number of their existing RAM board.

National has called its version of Tiny-BASIC "NIBL", which stands for National Industrial BASIC Language. This is meant to emphasise that they see its applications not only in educational and hobby computing, but also and perhaps more importantly in industrial control situations.

Like most other Tiny-BASIC interpreters, NIBL is not fast. A program written in NIBL executes somewhere between 1000 and 2000 times more slowly than an equivalent program written in machine or assembly language. But this can still be fast enough for most human interfaces, and for slow real-time control applications.

And the point is that NIBL lets you write programs and get them running very much more easily and quickly. It also provides "instant" error messages during program execution, to help you in debugging. And it lets you execute single instructions in real-time conversational mode, if you wish.

Naturally enough, NIBL doesn't provide all of the facilities of full BASIC—after all, it has to fit into only 4k bytes of memory. But it does provide a very



This is the pre-release version of NIBL, which was in only 3k bytes of PROM. The final version is in 4k bytes, with the PCB having eight of the PROMs instead of six.

```
10 PRINT "HI, I WORK OUT FACTORIALS."
20 PRINT "WHAT NUMBER WOULD YOU LIKE";
30 INPUT N
40 LET F=1
50 IF N<=1 THEN GOTO 90
60 LET F=F*N
70 LET N=N-1
80 GO TO 50
90 PRINT "ITS FACTORIAL IS",F
91 PRINT "DO YOU WANT TO WORK OUT ANOTHER (1=YES,0=NO)";
92 INPUT A
93 IF A=1 THEN GO TO 20
100 END
```

```
>RUN
HI, I WORK OUT FACTORIALS.
WHAT NUMBER WOULD YOU LIKE? 6
ITS FACTORIAL IS 720
DO YOU WANT TO WORK OUT ANOTHER (1=YES,0=NO)? 1
WHAT NUMBER WOULD YOU LIKE? 7
ITS FACTORIAL IS 5040
DO YOU WANT TO WORK OUT ANOTHER (1=YES,0=NO)? 0
```

Here are the two simple Tiny-BASIC programs which the author wrote to try out the pre-release version of NIBL. One works out factorials, the other is a number game.

```
10 PRINT "HI! I WILL THINK OF A NUMBER BETWEEN 0 AND 255."
20 PRINT "WHEN I HAVE,TRY TO GUESS ITS VALUE. (I WILL HELP)"
30 LET B=1
40 LET A=@B
50 PRINT "OK, I HAVE A NUMBER"
60 PRINT "WHAT IS YOUR GUESS";
70 INPUT C
80 IF C=A THEN GOTO 140
90 IF C>A THEN GOTO 120
100 PRINT "TOO SMALL. NEXT GUESS";
110 GO TO 70
120 PRINT "TOO BIG. NEXT GUESS";
130 GO TO 70
140 PRINT "YOU GUESSED IT!!!   LET'S PLAY AGAIN."
150 LET B=B+1
160 GO TO 40
170 END
```

```
>RUN
HI! I WILL THINK OF A NUMBER BETWEEN 0 AND 255.
WHEN I HAVE,TRY TO GUESS ITS VALUE. (I WILL HELP)
OK, I HAVE A NUMBER
WHAT IS YOUR GUESS? 200
TOO BIG. NEXT GUESS? 180
TOO SMALL. NEXT GUESS? 196
YOU GUESSED IT!!!   LET'S PLAY AGAIN.
OK, I HAVE A NUMBER
WHAT IS YOUR GUESS? 128
TOO BIG. NEXT GUESS? 64
TOO BIG. NEXT GUESS? 32
TOO BIG. NEXT GUESS? 0
TOO SMALL. NEXT GUESS? 16
YOU GUESSED IT!!!   LET'S PLAY AGAIN.
OK, I HAVE A NUMBER
WHAT IS YOUR GUESS? 128
TOO BIG. NEXT GUESS? 0
TOO SMALL. NEXT GUESS? 64
TOO BIG. NEXT GUESS? 32
TOO SMALL. NEXT GUESS? 45
TOO SMALL. NEXT GUESS? 56
TOO BIG. NEXT GUESS? 60
TOO BIG. NEXT GUESS? 50
TOO SMALL. NEXT GUESS? 54
YOU GUESSED IT!!!   LET'S PLAY AGAIN.
OK, I HAVE A NUMBER
WHAT IS YOUR GUESS?
! 7 AT 70
>
```

useful subset. Here's a summary of what you get:

Valid statement forms include INPUT (for numbers only), LET, GO TO, GO SUB and RETURN, IF THEN, and PRINT. There is also a CALL statement, to allow calling machine-language subroutines. The latter facility is very valuable, of course, because it will allow NIBL to be expanded.

The final version of NIBL may also have the ability to interpret DO . . . UNTIL statements, if PROM space permits.

Program control statements provided are LIST, RUN, END, and CLEAR. The first of these may be used to list either the whole program, or alternatively a single line. An inbuilt editor allows lines to be replaced, and also additional lines added as required. All that is necessary for correct execution is that lines are numbered consecutively between 1 and 32,767.

If statement lines begin with a number, NIBL stores them away as a program for deferred execution. If a statement is not numbered, NIBL executes it immediately upon entry (following the user terminating the line with a carriage return).

NIBL is only capable of performing integer arithmetic, on numbers within the range from −32,768 to +32,767. It provides the four basic arithmetic functions, represented by the symbols +,−,* and /, together with the logic operators AND, OR and NOT, and a 16-bit random number generator function called by the label RND. Constants may be expressed in either decimal or hexadecimal.

Up to 26 variables may be used in a NIBL program, using single alphabetic characters as labels (A-Z inclusive). Parenthesis is permitted, and subroutines may be nested to 16 levels.

All of the normal relational operators are provided, including equals, greater than, less than, greater than or equals, less than or equals, and not equal to.

NIBL also provides an operator of indirection, symbolised by the "at" or "@" sign. When placed immediately preceding a variable this causes the variable to be interpreted as a decimal address in SC/MP memory space. If A is a variable with value 256, the statement LET B=@A gives variable B the value equal to the data byte in decimal memory location 256.

What is NIBL like in practice? Well, at the time of writing the final version was not yet available in Australia, but thanks to NS Electronics I was able to try an earlier pre-release version. This was in only 3k of PROMs, and didn't have some of the features which will be in the full 4k version—like the RND function or the logic functions, or the CALL statement.

However it was certainly very interesting to try the smaller version out, with one of the SC/MP LCDS systems. Even though its facilities were rather limited, it was still very nice to be programming at a higher level of abstraction and be able to make corrections on-line.

In fact after having got a couple of simple programs up and running, it was with surprise that I noticed the time: less than an hour after the NIBL card had been plugged in and the system turned on!

As the two programs might be of interest to readers, I have reproduced them on these pages. In each case the program itself is listed first, followed by a sample of the execution.

As you can see, the first program is a very simple one which calculates the factorial of a number fed in from the keyboard. It prompts the user for a number, prints out the answer and then asks if the user wishes to work out another. A negative reply causes it to halt.

Incidentally, this little program soon comes up against the inability of NIBL to cope with numbers greater than 32,767. In fact the largest number it can find the factorial for is 7; 8 causes overflow.

The longer program is a simple number guessing game. As the pre-release version of NIBL didn't have the RND function, I had to use the indirect operator to generate pseudo-random numbers by fetching instruction bytes from NIBL's own PROMs! As you can see, this worked fairly well.

The program can be quite good fun, giving you a taste of the appeal in computer games.

Of course with the final version of NIBL, it will be possible to run games like this which will be rather more satisfying, using the RND function to generate less predictable numbers. In fact quite a few games have been written in Tiny-BASIC, and should be capable of being run with NIBL.

In short, NIBL seems to be very good news for SC/MP users.

You'll be able to order NIBL from NS distributors throughout Australia. For further information, contact NS Electronics at either Cnr. Stud Road and Mountain Hwy, Bayswater, Victoria 3153, or 2-4 William Street, Brookvale, NSW 2100.

Software/Firmware development—2

A Text Editor for SC/MP

Here is a symbolic text editor program which the author has written for National Semiconductor's SC/MP low cost development system. It provides all the basic text editing functions to let you prepare programs for assemblers, compilers, etc. You will be able to buy it resident in 1k bytes of PROM and ready to go, or, alternatively, it can be fed into your system via tape or cassette, and run in RAM.

by JAMIESON ROWE

If you've ever tried punching up a paper tape of a program in assembly or problem-orientated language using a teleprinter on "local", you'll know just how frustrating it can be. Even if you are extremely careful it seems to be impossible not to make a few errors, and Murphy's Law always seems to ensure that you rarely discover these until you have typed in at least three more lines!

Of course if you realise that you made an error immediately after having done it, you can use the back-space facility and delete the wrong character(s) with the rubout key. But if you don't discover an error until later on, you are forced to either perform cut-and-paste surgery on the tape, or try stop-and-go editing of the tape while punching a new "clean" version.

Preparing long problems in this way can be very tedious and time consuming. Small wonder, then, that the people working on minicomputers and larger machines have for years been using the computer itself to make the job easier and faster. This is done by using a software utility program known as a symbolic text editor.

Using such a program, you type your own program text into a section of the computer's memory which is set aside as the "buffer". Then with the text in the buffer, the editor lets you change lines, delete lines, insert extra lines at any desired point, inspect lines or groups of lines, and finally either type out a listing or punch out a clean tape (or both).

In short, a text editor program can be a very useful item of software, and it is well worth having one even on small microcomputer systems. The only trouble is that not too many microcomputers have been provided with editor programs as yet, particularly the systems based on the more recent microprocessor chips.

To help alleviate this situation, I have written a text editor program for the National Semiconductor SC/MP lowcost development system, as described in our October issue. I chose this system because at present the SC/MP chip and its systems appear to be growing fastest in popularity, particularly in the hobby area.

The editor program itself is written in SC/MP machine language, and occupies 864 bytes of memory. It uses the 256-byte RAM in the LCDS system base as a stack and line address buffer, but can use RAM memory at any other location in SC/MP memory space for its text buffer. The larger the available RAM, the more text the editor can handle at one time.

I have arranged for the editor to be available from NS Electronics distributors, written into a pair of MM5204 512-byte PROMs. With the PROMs plugged into the correct locations on a SC/MP ROM card which is programmed for the appropriate address range (hex. 3000-3FFF), you will then have the editor permanently resident in your system, and available at any time merely by calling it at its starting address (hex. 3C00).

Alternatively, you can run the editor in RAM memory, loading it in each time you need it by means of a punched paper tape or cassette. I am reproducing a full hexadecimal listing of the program on these pages, to allow you to do this if you wish. The four-digit numbers at the start of each line are addresses; as you can see the program runs from 3C00 to 3F5F. inclusive.

Note that if you do elect to run the editor in RAM, you will need at least one 2k-byte RAM card. This will give you 1k of RAM left for the text buffer—enough for small text slabs, but barely good enough for serious work. With the editor in PROMs, even a single RAM card gives you a full 2k for the text buffer, which is very much more practical.

When the editor is called, it announces itself and then asks you to give the available text buffer range in memory. This must be supplied as two hexadecimal numbers, separated by a non-hex character such as a space or hyphen. Leading zeroes are not required, but the second number must end with a carriage return.

The editor then enters its command mode, signalling this by ringing the bell. The user may then type in a command letter, followed by a carriage return. If text is to be fed in via the keyboard, the command letter "A" is appropriate, while "R" tells the editor to read in a previously punched tape via the reader.

```
SC/MP SYMBOLIC EDITOR PROGRAM.
WRITTEN BY J.ROWE, ELECTRONICS AUSTRALIA
FOR SC/MP L.C.D.S. SYSTEMS

BASIC COMMANDS AND THEIR FUNCTIONS:
(THE CLOSING BRACKET ")" SYMBOLISES A CARRIAGE RETURN)

COMMAND                 FUNCTION

A)                      APPEND LINES TO BUFFER
R)                      READ TAPE INTO BUFFER
L) . ML) , M,NL)        LIST ALL LINES, OR LINE M, OR LINES M-N
MC) , M,NC)             CHANGE LINE M, OR LINES M-N
MI)                     INSERT LINE OR LINES BEFORE LINE M
MD) , M,ND)             DELETE LINE M, OR LINES M-N
K)                      KILL TEXT IN BUFFER
P) , MP) , M,NP)        PUNCH ALL LINES, LINE M, OR LINES M-N
B) , MB) , M,NB)        AS FOR PUNCH, BUT PUNCHES A BELL CHAR
                        AT END OF TEXT
/                       EDITOR PRINTS NUMBER OF LINES CURRENTLY
                        HELD IN TEXT BUFFER, IN DECIMAL
BELL                    WHEN IN TEXT MODE, RETURNS EDITOR TO
                        COMMAND MODE
(M AND N REPRESENT DECIMAL LINE NUMBERS)
```

Here is the basic command set of the editor, showing the various command letters, the possible arguments for each, and their functions. In text mode, a percent sign acts as a backspace.

```
3C00 04 C4 77 36 C4 FF 32 C4 7B 37 C4 16 33 3F 3F 07
3C10 C4 4F 33 3F C6 01 CA F1 C6 01 CA EF 3F C6 01 CA
3C20 EF C6 01 CA ED C4 00 CA F9 C2 F0 CA F8 C2 EF CA
3C30 F7 C4 7A 37 C4 E1 33 C4 07 3F C4 0D 3F C4 0A 3F
3C40 C4 00 CA F4 CA F3 CA F2 C4 7A 37 C4 90 CA F1 33
3C50 3F D4 7F 01 C4 3F 35 C4 2B 31 C5 03 98 3B 60 9C
3C60 F9 C1 FE CA F6 C1 FF 31 C2 F6 35 C2 F9 03 FA F3
3C70 94 02 90 12 C2 F9 03 FA F2 94 02 90 09 C2 F2 98
3C80 10 03 FA F3 94 0B C4 7A 37 C4 E1 33 C4 3F 3F 90
3C90 A9 3F D4 7F E4 0D 9C EE 3D 40 D4 70 E4 30 98 19
3CA0 40 E4 2F 9C 09 C4 3E 35 C4 B2 31 3D 90 8C 40 E4
3CB0 2C 9C D3 C4 01 CA F4 90 8F 40 D4 0F CA F5 C2 F4
3CC0 9C 04 C2 F3 90 02 C2 F2 02 01 40 70 01 70 70 01
3CD0 40 70 F2 F5 01 C2 F4 9C 05 40 CA F3 90 D9 40 CA
3CE0 F2 90 D4 35 CA F6 31 CA F5 C2 F7 CA E9 C2 F8 CA
3CF0 EA C2 F1 E4 84 98 09 C4 7A 37 C4 E1 33 C4 0A 3F
3D00 C4 7A 37 C2 F1 33 C2 EE 02 F4 01 E2 F8 98 53 C2
3D10 F8 35 C2 F7 31 3F D4 7F 01 40 E4 0A 98 1D 40 E4
3D20 07 98 3F 40 E4 25 9C 04 C5 FF 90 0F 40 98 0C E4
3D30 7F 98 08 40 E4 0D 98 01 40 CD 01 C4 77 35 CA F8
3D40 C2 F3 02 F2 F3 31 CA F7 40 E4 0D 9C B6 C2 E9 C9
3D50 00 C2 EA C9 01 C2 F9 E4 74 98 07 C2 F6 35 C2 F5
3D60 31 3D C4 3C 35 C4 30 31 C4 90 CA F1 3D 08 08 08
3D70 C4 80 90 02 C4 01 CA F1 C2 F3 9C 0A C4 01 CA F3
3D80 C2 F9 CA F2 90 08 C2 F2 9C 04 C2 F3 CA F2 C4 E1
3D90 33 C4 0A 3F C2 F1 98 15 C4 7A 37 C4 88 33 3F C4
3DA0 E1 33 C4 C0 CA F6 C4 00 3F AA F6 9C F9 C4 77 35
3DB0 C2 F3 02 F2 F3 31 C1 00 CA F6 C1 01 35 C2 F6 31
3DC0 C5 01 98 03 3F 90 F9 C4 0D 3F C4 0A 3F C2 F3 E2
3DD0 F2 98 04 AA F3 90 D6 C2 F1 98 1A 94 03 C4 07 3F
3DE0 C4 C0 CA F6 C4 00 3F AA F6 9C F9 C4 7A 37 C4 88
3DF0 33 3F C4 E1 33 C4 3C 35 C4 39 31 3D 08 08 08 08
3E00 C2 F2 9C 06 C2 F3 98 35 CA F2 BA F3 C2 F2 E2 F9
3E10 98 20 AA F3 AA F2 C4 77 35 C2 F2 02 F2 F2 31 C4
3E20 77 37 C2 F3 02 F2 F3 33 C1 00 CB 00 C1 01 CB 01
3E30 90 DA C2 F3 CA F9 C4 3C 35 C4 30 31 3D C4 3C 35
3E40 C4 83 31 3D C2 F3 98 F5 C2 F2 9C F1 C4 77 35 C2
3E50 F3 02 F2 F3 31 C1 00 CA EB C1 01 CA EC C4 3C 35
3E60 C4 E2 31 3D C2 F9 CA F2 AA F9 C4 77 35 C2 F2 02
3E70 F2 F2 31 C1 00 C9 02 C1 01 C9 03 C2 F2 E2 F3 98
3E80 04 BA F2 90 E5 C2 EB C9 02 C2 EC C9 03 AA F3 90
3E90 BB C2 F2 9C 06 C2 F3 98 A4 CA F2 C4 3C 35 C4 E2
3EA0 31 3D C2 F3 E2 F2 98 04 AA F3 90 EF C4 3C 35 C4
3EB0 30 31 3D C4 00 CA F6 CA F5 CA F4 C2 F9 02 F4 9C
3EC0 94 05 C2 F9 01 90 03 01 AA F6 40 02 F4 F6 94 02
3ED0 90 05 01 AA F5 90 F3 40 02 F4 FF 94 02 90 05 01
3EE0 AA F4 90 F3 C4 7A 37 C4 E1 33 C4 3D 3F C2 F6 98
3EF0 03 C4 31 3F C2 F5 98 03 DC 30 3F C2 F4 DC 30 3F
3F00 C4 3C 35 C4 39 31 3D 0D 0A 45 44 49 54 4F 52 20
3F10 52 45 41 44 59 2E 0D 0A 47 49 56 45 20 42 55 46
3F20 46 45 52 20 52 41 4E 47 45 3A 00 41 3F 4A 52 3F
3F30 46 4C 3D 75 43 3E 90 49 3E 43 44 3D FF 4B 3C 24
3F40 50 3D 73 42 3D 6F 00 C4 84 90 02 C4 90 CA F1 C2
3F50 F9 CA F3 AA F3 C4 3C 35 C4 E2 31 3D AA F9 90 EF
```

Use this complete hexadecimal listing of the program if you wish to prepare a paper tape or cassette to run the editor in RAM, or if you are able to burn your own PROMs.

Once the text is in the buffer, you can edit it using the commands shown in the table. Note that the L, C, I, D, P and B commands may all have arguments, to specify individual lines or a group of lines. In fact the I command must have one argument, to indicate where the insertion is to take place.

The argument number or numbers must precede the command letter. Thus to list lines 12 to 15, for example, you simply type 12, 15L followed by a carriage return.

To change a line, say line 34, you simply type 34C, a carriage return, and then type in the new line text. Similarly to insert a new line or lines before an existing line, say line 17, type 17I followed by a carriage return and then type in the extra lines. To delete lines, say lines 20, 21 and 22, type 20, 22D and then a carriage return.

A single argument implies that the command should affect only the one line. Two arguments imply that the command should affect all lines between the two corresponding lines, inclusively. Thus 15, 20C implies that six new lines are to be fed in, replacing the existing lines 15, 16, 17, 18, 19 and 20.

(Continued on page 133)

An ideal microcomputer for the beginner:

"Mini Scamp"

Forget about expensive terminals: here's a REALLY low cost and simple microcomputer. It uses front-panel bit switches and LEDs for input and output, in normal binary code, making it completely self contained. Based on the National SC/MP microprocessor, it comes with a minimum of 256 words of RAM—but this is easily expanded up to 1024 words. We think it's the ideal way of getting into the exciting world of microcomputers at low cost.

by DR. JOHN KENNEWELL Physics Dept., Newcastle University

The design of this microcomputer started around October of last year with the formation of the Newcastle Microcomputer Club. It became obvious at the inaugural meeting that there were many people who would like to play with their own microcomputer, developing programming skills, yet who were unable to afford even the lowest cost kits available on the market. The problem becomes even more acute when considering the cost of a terminal to interface with these kits.

Various solutions to the terminal problem have now been presented in this magazine. Jim Rowe has described an ASCII-Baudot translator for use with surplus Baudot teleprinter machines (EA, October 1976) and also a video data terminal (EA, January and February 1977). However, either of these alternatives means an outlay of at least about $200, not including the microcomputer itself, which brings the total cost to around $300 using a small system such as the SC/MP evaluation kit.

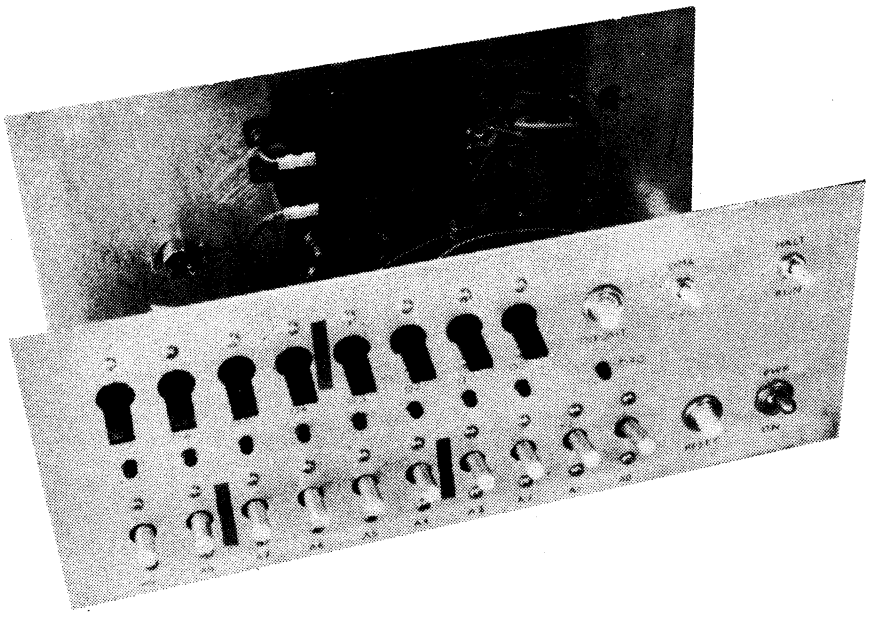
Recently Applied Technology have released a SC/MP I/O kit, which interfaces with the SC/MP evaluation kit and permits program and data entry via panel switches. While this unit undoubtedly fills a gap in available I/O hardware and seems to be enjoying great popularity, the total system cost is over $150 (including power supply). I also feel that it is not suitable for a beginner to microcomputers because it employs an operating system (in ROM) that was designed for communication with a teletype using hexadecimal characters in ASCII code. Entry of information via the panel switches thus involves a prior translation of characters into this code, and one tends to lose contact with the basic organisation of the processor (CPU). From the point of acquiring an understanding of microprocessor operation (without the complications of an intervening operating system) I feel this is undesirable.

In searching for a suitable design, and to overcome the problems mentioned above, it became apparent to me that the idea of using an available evaluation kit together with an I/O interface was not the way to go. As an operating system was not desirable, there was no need for read only memory (ROM). Building a system from scratch meant that costs could be kept down as only those features necessary were included. At the same time, I personally wished to build a much larger system than that shown here, and ease of system expansion was well to the fore in my design considerations. The TOTAL cost of the computer should fall somewhere between $50-$100 depending upon your method of construction, selection of components, and upon how many existing components you have that may be pressed into service. This is most likely to be so in the case of the power supply.

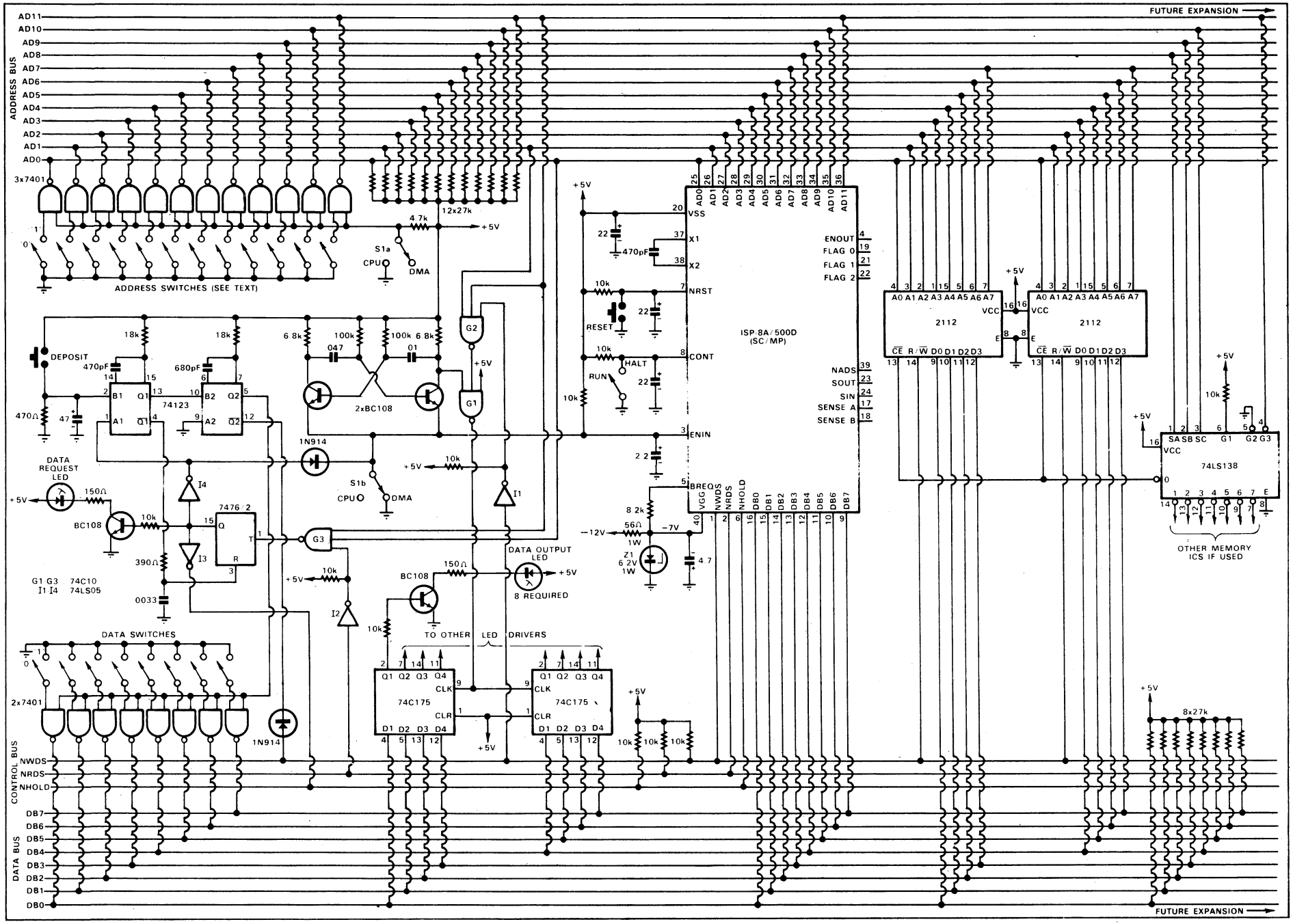
Excluding the power supply, the computer may be conveniently divided into three basic units. These are the central processing unit (CPU), the memory, and the front panel input/output circuitry. These three units communicate with one another via three system 'bus' lines: an address bus, a data bus, and a control bus.

The address bus, comprising twelve actual lines, is used by both the CPU and the front panel I/O circuit to specify which location in memory information is to be sent or received. It is also used by the CPU during program execution, to select a particular input or output device for communication with the outside world.

The data bus, of 8 lines, enables the passage of information between any two of the three units in either direction. The unit that does not participate in a given



At left is the author's prototype of his Mini Scamp, with the full circuit shown on the page opposite.


FUTURE EXPANSION
ADDRESS BUS
AD11
AD10
AD9
AD8
AD7
AD6
AD5
AD4
AD3
AD2
AD1
AD0
3x7401
ADDRESS SWITCHES (SEE TEXT)
12x27k
4.7k
+5V
S1a
CPU
DMA
DEPOSIT
470pF
680pF
18k
74123
470Ω
47
6.8k
100k
047
01
2xBC108
G2
G1
1N914
S1b
DATA REQUEST LED
150Ω
BC108
10k
7476/2
I4
I3
390Ω
0033
G1 G3 74C10
I1 I4 74LS05
DATA SWITCHES
2x7401
1N914
I2
DATA OUTPUT LED
8 REQUIRED
TO OTHER LED DRIVERS
74C175
CLK
CLR
D1 D2 D3 D4
Q1 Q2 Q3 Q4
CONTROL BUS
NWDS
NRDS
NHOLD
DATA BUS
DB7
DB6
DB5
DB4
DB3
DB2
DB1
DB0
ISP-8A/500D
(SC/MP)
VSS
X1
X2
NRST
CONT
ENIN
BREQ
VGG
NWDS
NRDS
NHOLD
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
ENOUT
FLAG 0
FLAG 1
FLAG 2
NADS
SOUT
SIN
SENSE A
SENSE B
22
470pF
RESET
HALT
RUN
8.2k
56Ω
1W
-12V
-7V
Z1
6.2V
1W
4.7
2112
A0 A1 A2 A3 A4 A5 A6 A7
VCC
CE R/W D0 D1 D2 D3
74LS138
SA SB SC G1 G2 G3
OTHER MEMORY ICS IF USED
8x27k

data transfer is disabled (i.e. put in a high impedance state) so that it does not affect the transfer.

The control bus, of only 3 lines, is used to specify whether information is to be read from the memory, or is to be written into the memory, and to place the CPU in a 'hold' condition while it waits for information to be given it via the front panel or other slow peripheral device.

Twelve address lines enable a total of 4096 words of memory and/or peripheral devices to be addressed independently by the CPU. The concept of the bus system described here makes possible the easy expansion of the computer up to this limit, if so desired, by the addition of more memory and more I/O devices. The SC/MP CPU is actually capable of directly addressing up to 65k of memory and/or peripherals. Although this may be readily accomplished with a latch and some buffer IC's it will not be discussed here further.

Of the three sections making up the system the CPU is the heart, or rather the brain, of the system. It comprises the SC/MP integrated circuit microprocessor chip, which requires two voltages for correct operation, +5V and −7V. The −7V (actually −6.2V) is provided from a nominal −12V line by means of a series dropping resistor (56 ohms, 1W) and a zener diode regulator. The other resistors in the circuit are pull-up resistors, to ensure that the appropriate pins on the SC/MP have the correct potential for normal operation.

Some of these potentials can be modified by switches on the front panel. For instance, the DMA/CPU switch can disable the CPU by placing zero potential on the ENIN terminal of SC/MP. This is necessary when data is being entered into memory via a direct memory access (DMA) from other front panel switches, as described later. The RUN/HALT switch controls the potential of the CONTinue terminal, and enables suspension of program execution at any time. The RESET pushbutton must be pressed before initial execution of each program. This ensures that all internal registers of the SC/MP are set to zero, and that the first instruction fetched from the memory will be from location one.

The capacitors on each of these switched lines are crude debounce devices, but have been found to be quite adequate. A quick or snap action when using the switches will always help in this respect.

The 470pF capacitor connected between the X1 and X2 pins determines the speed at which the processor will run. Unlike many other microprocessors, the SC/MP has all the required timing generation circuitry built in, with the exception of this one external component. The value of capacitance shown here will run the SC/MP at near its maximum speed with a 'microcycle' time of 2us. Typical program instructions in the SC/MP take from 5 to 22 microcycles to execute.

The memory section of the circuit uses two low-cost 2112 static MOS memory chips which together provide 256 words of memory, each word of 8 bits in length. These words occupy address locations starting at 0 and extended to 255 (decimal) inclusive (0-FF hexadecimal). The eight address pins on the devices are fed from the eight least significant address lines (AD0-AD7 inclusive).

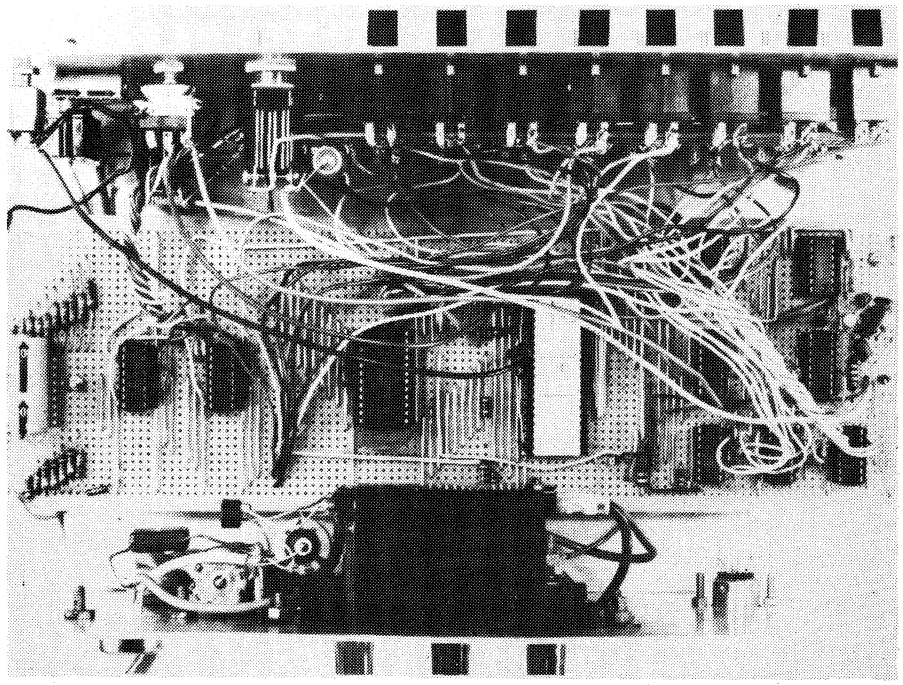
To ensure that the devices only occupy address locations 0-255, the remaining lines of the address bus are fed to a 74LS138 one-of-eight decoder. The "0" output of the decoder is then fed to the chip select (CS) inputs of the memory chips, so that the latter are only enabled or "selected" when the four address lines AD8 through AD11 are in the zero state.

Note that the 74LS138 is basically a 3-bit decoder, and has only three nominal code inputs. The most significant address line AD11 is therefore fed to one of the decoder's own chip select inputs, to achieve the desired result.

Note also that the remaining outputs available on the 74LS138 (1-7 inclusive) may be used to provide selection signals for additional memory devices. The memory of the system can thus be expanded very simply, merely by adding further pairs of 2112 devices.

The front panel I/O section actually has the greatest circuit complexity of the three parts of the system—neglecting, of course, the tremendous internal complexity of the CPU and memory LSI chips. It has two fundamentally different modes of operation. When the DMA/CPU switch is in the DMA position, then the address switches have control of the address bus. The contents of the memory address indicated by these switches will be displayed by the LED's (L0 to L7) on the front panel.

If it is desired to change the contents of any particular memory location, the address of that location is set up on the address switches, and the data to be inserted is set up on the data switches (DS0 to DS7). If the DEPOSIT pushbutton is then depressed and released, the LED's will confirm that the data has indeed been stored in memory at that location. In this way a program may be loaded into memory. This is described in more detail a little later, using a sample program.



The inside of the author's prototype, which was built up using Veroboard. To help readers we are producing a PCB pattern — see box at lower right.

As it is more convenient to represent both data and addresses in hexadecimal rather than binary notation, it will be found convenient to group or delineate these switches into fours. PVC marking tape was used on the front panel of the prototype computer as can be seen in the accompanying photograph.

In the second mode of operation, the DMA/CPU switch is set to the CPU position. In this mode, the address switches are disabled, and have no control over the address bus. The RUN/HALT may then be set to RUN and the CPU will begin to execute whatever program instructions are in memory at this time. Also in this mode, the data switches function as an input device at the hexadecimal address 0801 (hex). Thus, under program control, data can be read into the CPU from the data switches. The

instruction to do this has the form

LD SWITCHES

where SWITCHES has a hex value of 0801. The LD instruction loads whatever data is found at the address of the operand (in this case address 0801) into the accumulator register of the CPU.

To indicate to the external world that it requires data (i.e., that the above instruction has been executed), the CPU, via signals on address lines ADO and AD11, and on the read data strobe control line (NRDS) is used to turn on a data request LED (DRQ). This is done via G3 which detects a coincidence of the above three signals and toggles the flip-flop which turns on the DRQ LED and also pulls the NHOLD control line to zero. This will cause the CPU to remain in a 'wait' condition until the line returns to a 'one' state. This will occur when the deposit button is pushed, triggering the monos, which after a small delay from the 390-ohm and 0.0033uF RC network, reset the flip-flop.

In this second mode the LED's act as an output device with an address 0802 (hex), which is selected by G2 and activated by an instruction of the form

ST LEDS

where LEDS represents the hex address 0802.

The astable multivibrator comprised of the two BC108's is disabled in the CPU mode, but in the DMA mode provides a continuous string of latching pulses to the 74C175's so that the LED's will always display the contents of the address as indicated by the address switches.

The two diodes in the circuit are used as cheap 'OR' gates for simplification. They could be replaced by another IC if desired, but they have proved quite adequate, and help to keep the cost and total package count down. For similar reasons the 7401 has been used as a "poor-man's tri-state buffer", instead of the more expensive buffer IC's manufactured especially for tri-state applications. There is no reason, however, why these latter chips, such as the 81LS97, should not be used if available.

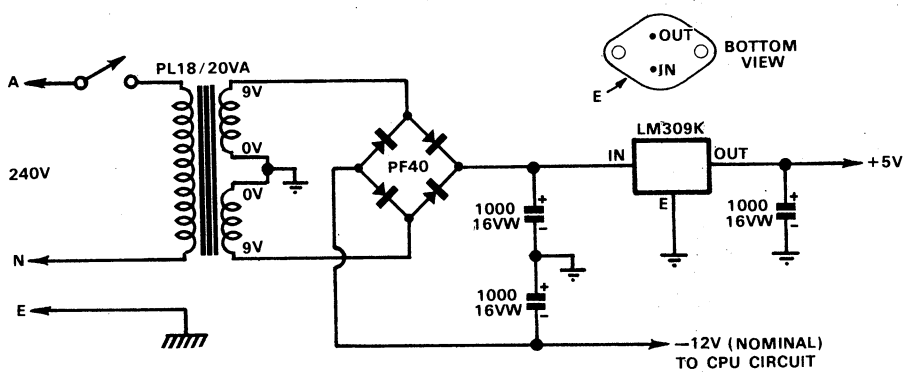


Fig. 2: A simple power supply circuit for Mini Scamp. Any other supply capable of delivering 5V at 500mA and —12V at 150mA could be used instead.

If the system is to be run with only the minimum amount of memory, (i.e., 256 words) then only 8 address switches are required ($2^8 = 256$). The remaining switch lines (AS8 to AS11) may simply be left floating. This is equivalent to placing a zero on these lines. Thus, depending upon the amount of memory you have available, anywhere from 8 to 12 address switches will be required on the front panel.

It should be noted however, that no more than 2K of memory can be accommodated on this system without modification. This is because addresses above hexadecimal 0800 (or decimal 2048) are used to reference the data switches and LED's.

Those of you who have closely followed the circuit will have realised that in fact the addresses 0801 and 0802 (hex) are not unique in their ability to reference the switches and LED's respectively. All twelve address lines instead of only two, as used at present, would be necessary to uniquely specify a single device. Thus all addresses which have a one in bit positions 0 and 11 will refer to the data switches. These all lie at locations greater than 0800 (e.g. 0803, 0805, 08A9, etc.) and so will not conflict with memory addresses less than this value.

Various techniques and methods may be used in the construction of the microcomputer. Veroboard was used to build the prototype as shown in the photograph, and probably provides the lowest cost way to go. Layout and component placement is not critical. A single length of Veroboard could be used, or the three main sections could each be constructed on smaller separate boards. This latter approach allows one to more easily interchange units if desired. (e.g. to try an 8080 CPU, or to substitute a larger memory unit).

Printed circuit boards for each section make for simpler construction and greater flexibility, and several members of our microcomputer club will probably employ this approach. However, it should be borne in mind that the cost of PCB's and their respective sockets will

Mini Scamp: a PCB is coming

Dr. Kennewell's "Mini Scamp" microcomputer design seems to us to be just what many of our readers have been waiting for: a really simple way of becoming familiar with microprocessors and their operation. Because of its low cost, its ease of expansion, and the fact that it needs no expensive terminal, we believe it could become an extremely popular project and a worthy successor to our own EDUC-8 design. To help ensure this well-deserved popularity, we are producing a low-cost PCB pattern for the project. All going well we hope to publish details next month.

```
                *BINARY COUNT AND DISPLAY
0000 08           NOP
0001 C408         LDI   8
0003 35           XPAH  1
0004 C400         LDI   0
0006 31           XPAL  1
0007 C902 LOOP    ST    2(1)
0009 8FFF         DLY   255
000B A803         ILD   COUNT
000D 90F8         JMP   LOOP
000F   00 COUNT   .BYTE 0
0010
```

```
                *MOVING LIGHTS WITH INPUT
0000 08           NOP
0001 C408         LDI   8
0003 35           XPAH  1
0004 C400         LDI   0
0006 31           XPAL  1
0007 C101 LOAD    LD    1(1)
0009 C810         ST    BITS
000B C00E LOOP    LD    BITS
000D C902         ST    2(1)
000F 1E           RR
0010 C809         ST    BITS
0012 8FFF         DLY   255
0014 B806         DLD   COUNT
0016 9CF3         JNZ   LOOP
0018 90ED         JMP   LOAD
001A   00 BITS    .BYTE 0
001B   00 COUNT   .BYTE 0
```

Here are two sample programs to help you get going with Mini Scamp. In both cases the hexadecimal numbers in the first column are memory addresses, and those in the next column are the actual code. Each pair of hex digits is an 8-bit byte.

greatly increase the cost of the overall unit, and may not be justified, particularly if further expansion is not desired.

The LED's and their transistor drivers were soldered onto a long narrow strip of Veroboard and then the LED's were glued through holes in the front panel.

The power supply requirements are quite small, and any supply giving +5V at say 0.5A and about -12V at 150mA should prove adequate. Fig. 2 gives a typical circuit for those wishing to build the power supply using new components.

It will be found that the cost of the switches can be a considerable fraction of the total computer cost. Those used for the prototype were lever switches (DPDT) from Tandy Electronics. Similar switches at a much lower cost from Electronic Disposals in Little Lonsdale Street in Melbourne have also been tried. Although satisfactory to date, only time will allow us to determine how many repeated switchings may be made before the contact resistance becomes too large for correct operation. In this regard, it is most desirable to wire both poles of the above type of switches in parallel.

Although this microcomputer was conceived mainly as an educational instrument through which an understanding of the engineering and programming concepts involved could be learnt, there is no reason why it could not be put to work in the role of a simple controller. Detection of off/on states of various devices, and the activation of relays, etc., is most easily accomplished using the sense inputs and flag outputs available on the SC/MP chip. Anyone wishing to make good use of the computer should obtain a copy of the SC/MP Technical Description from NS Electronics Pty. Ltd., in Bayswater, Victoria, or from their distributors. This manual describes all of the program instructions available on the SC/MP, and details what each of these actually does.

On the programming side, you might like to try your hand at writing a multiplication routine, a BCD to binary conversion program, the converse, i.e. binary to BCD, or even a simple program to demonstrate the function of the logical operations, AND, OR and EXCLUSIVE OR.

Although it uses a different instruction set than does the SC/MP, the advice on programming contained in the EDUC-8 handbook provides valuable information for those with little prior knowledge in this field. I have also found the hexadecimal conversion table printed in the E.A. Yearbook (1976/77) to be of great assistance when manually assembling small programs.

In order to get you started along the road in programming your microcomputer, I will describe two short demonstration programs.

The first program simply counts in binary, displaying each number on the LED's with a fixed delay between numbers. The delay is necessary to slow the computer down sufficiently for you to observe what is happening. The program listing is shown in Fig. 3.

Ignoring the first two columns for the moment, what we have is a list of instructions to the computer in 'Assembly' language. The first instruction (NOP) does nothing, and is ignored by the CPU, as the first instruction actually executed is at address one. The next four instructions load the hex address 0800 into pointer register 1. The following instruction (ST 2(1)) outputs whatever number is presently in the accumulator to the LED's. Note that the operand 2(1) means the address stored in pointer register 1 plus 2 (i.e., 0802) which is, of course, the address of the LED's.

The next instruction (DLY 255) creates the delay, while the ILD COUNT instruction adds one to the location called COUNT (which has address 000F) and then loads this number into the accumulator, ready for display on the LED's when the CPU jumps back (via the JMP LOOP instruction) to the ST 2(1) instruction.

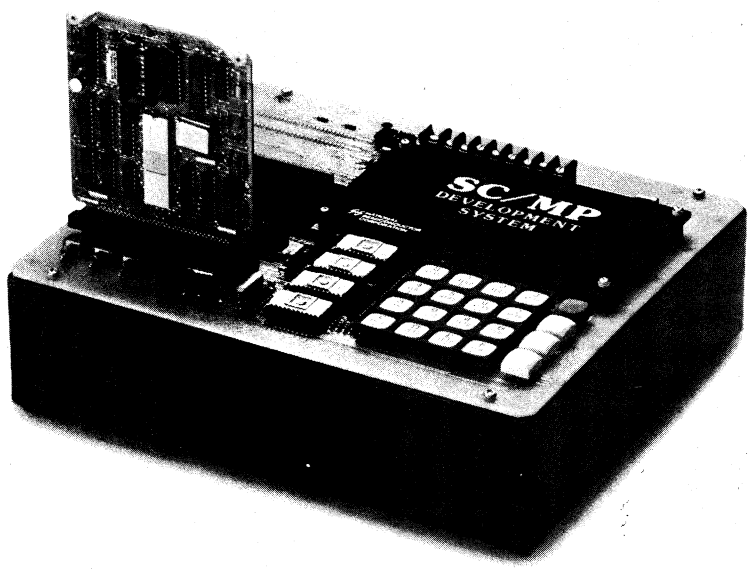
The information in the second column is the translation of the assembly language instructions detailed above into machine language form. These hexadecimal numbers may be loaded into the memory at their respective addresses shown alongside in column 1.

First, set the RUN/HALT switch to HALT, and the DMA/CPU switch to DMA. Then set all the address switches to zero, and set the hexadecimal number 08 on the data switches. Now press DEPOSIT, and the LED's should also display 08. Continue by setting the address switches to 01 and the data switches to C4. Depress DEPOSIT again. As the LDI 8 instruction is a double-byte instruction, the number 08 must now be set into address 02 followed by 35 into 03, and so on, using the above procedure. When all locations up to and including 0F have been loaded, the program is ready to be executed or run.

Set the DMA/CPU switch to CPU, depress and release the RESET button, and then set the RUN/HALT switch to RUN, and the LED's should then be counting. If not, return the appropriate switches to HALT and DMA, in that order, and check the contents of each address by successively incrementing the address switches from 00 to 0F hex.

The second program shown in Fig. 4 demonstrates both the data input facility of the computer and also the rotate instruction (RR). When run, the program will request (via DRQ) any number from the data switches. For example, when DRQ comes on, set hex 80 on the switches, then press DEPOSIT. The single light on the left will then be successively moved to the right and finally 'rotated' back to its initial position. After 256 rotations, the program will then request a new bit pattern to rotate.

SCRIMP WITH SC/MP.



Our low cost microprocessor development system.

You probably know all about how you can get microprocessor chips for a song these days. (Our SC/MP, less than ten bucks apiece in volume.)

Now you can develop their applications for a song, too. $460.

Introducing National's SC/MP Low Cost Development System (LCDS).

Not a kit or an evaluation tool, but a fully assembled, tested system with all the features necessary for development and testing of SC/MP hardware and software designs for a very broad range of applications.

Software debug is easy because there's a built-in keyboard and display.

Expansion is easy too, with a wide range of standard application cards, including our ready-made RAM and ROM/PROM cards.

Whether you prototype with cards, or start from scratch with components, the SC/MP LCDS makes it easy. And when you're through, the application cards you've used in development go right into your system!

Sold? Give your distributor a call—and ask for an ISP-8P/301. Or call (03) 729-6333. Not sold quite yet? Here's a nice coupon.

National Semiconductor Corporation,
P.O. Box 89, Bayswater, Vic. 3153.
Gentlemen:
I'm filling in this coupon so you can fill me in on your SC/MP program development system.

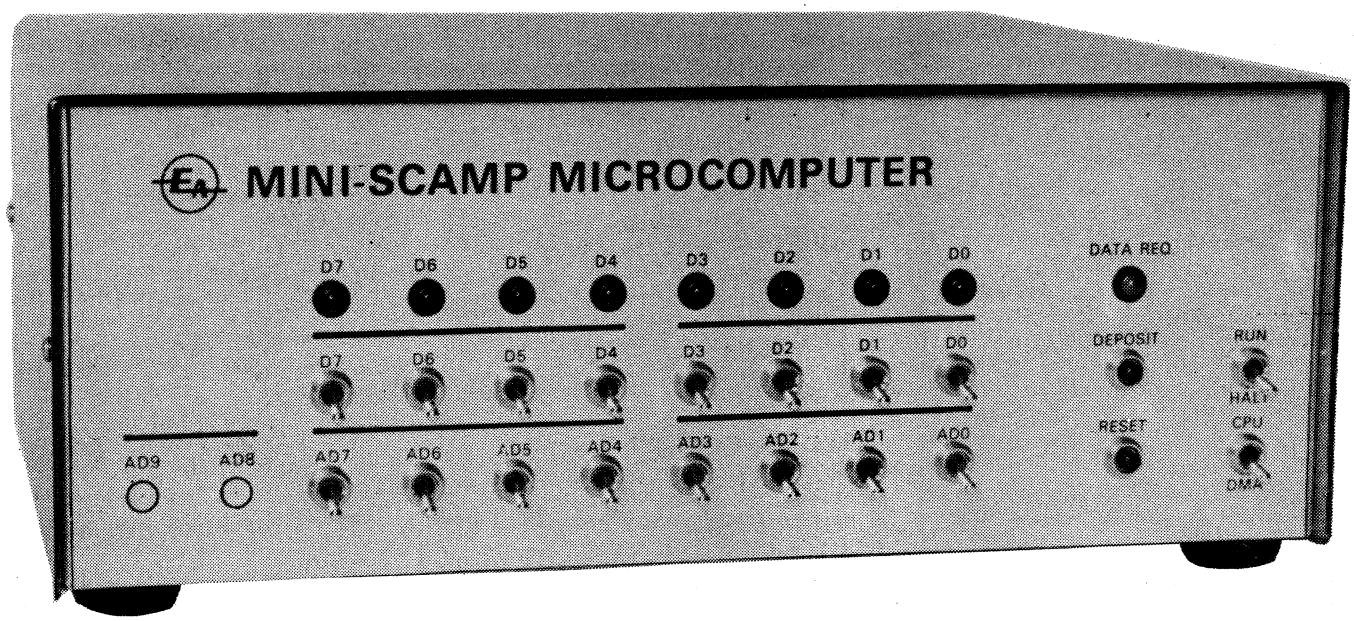
Name_____ Title_____

Company _____

Address _____

City_____ State_____ P.C._____

National Semiconductor



Look what happened to the Mini Scamp!

With some help from component suppliers, we have been able to turn Dr. Kennewell's Mini Scamp microcomputer design into a complete full-scale construction project. With almost all of the circuitry on a single PC board, it is now not just the lowest-cost complete microcomputer system available, but the easiest to build as well!

by JAMIESON ROWE

Just about everyone interested in microcomputers seems to agree that Dr. John Kennewell's Mini Scamp design has great potential. By starting from scratch with a SC/MP chip, and then designing a simple RAM-orientated system around it, he has produced an ideal microcomputer for the hobbyist and student.

It is fully self-contained, needing no expensive terminal. Programs are fed in via front-panel switches and LEDs, which can also be used to communicate with the machine when it is running—in simple binary code, the actual language used by the machine itself. What better way to learn how computers work!

At the same time, it can be built for around half the cost of any other microprocessor based system, and hundreds of dollars less than broadly comparable earlier designs like our own EDUC-8.

In other words, it is a design which should appeal to a wide variety of people, especially those still looking for a way of becoming familiar with microcomputers easily and at low cost.

While we were preparing Dr. Kennewell's article for last month's issue, the conviction grew that the project deserved to become a very popular one. But we realised that one thing was lacking: a low-cost PC board, to make it really easy to build even for those with little previous experience.

We immediately resolved to design a PCB for the project, to help ensure that it wins the popularity it deserves. And we managed to fit a small "stop press" box in the April article, to let readers know that a PCB was on the way.

Because of the box no doubt quite a few readers have been waiting for the current issue, for the promised PCB design. As you can see, we have in fact gone much further than this, and have turned Mini Scamp into a full-scale project. So that your wait should not have been in vain.

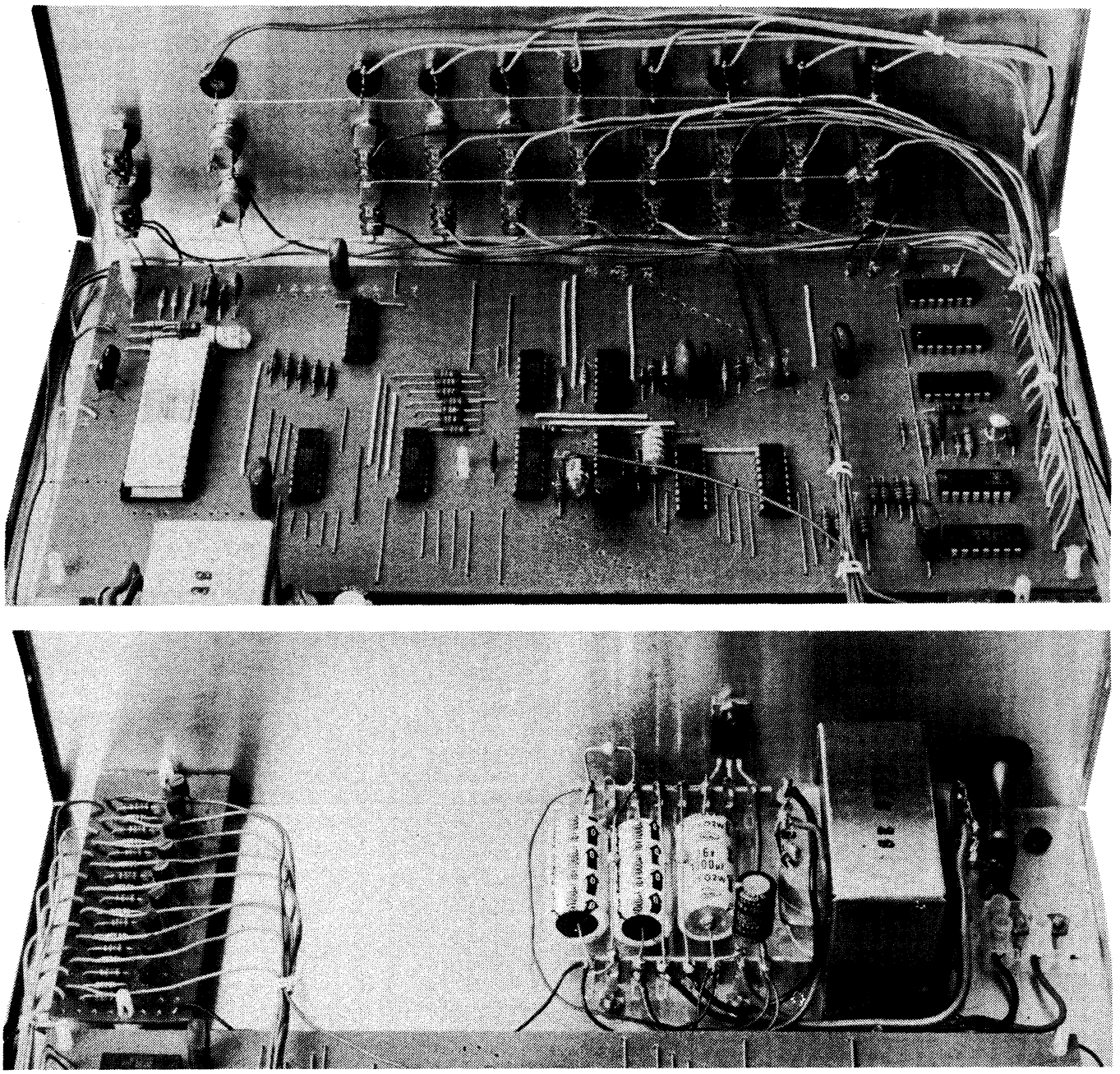
How did this happen? Well, we couldn't publish the PCB design without trying it out first, to make sure it worked. This meant getting together a set of ICs, switches and other components, and at the same time warning suppliers that the project was coming—to ensure that readers would be able to buy the parts. While we were doing this we sought reactions from suppliers also, to see if they became as keen about the project as we were.

They certainly did. In fact, some of the suppliers were so enthusiastic that they offered to make some of the components available to readers at special prices.

For example NS Electronics have offered to make available through their distributors a special deal on the SC/MP chip, all the other ICs, the transistors and LEDs. These will be available for $36.21 including tax, or $31.49 to schools and colleges who can claim a tax exemption, until the end of June.

Similarly C&K Electronics are prepared to supply the complete set of 18 toggle switches and 2 pushbuttons direct to readers for a package-deal price of $14.65 plus 50c postage, or $12.74 plus 50c postage for those who can claim a tax exemption. Their address is PO Box 101, Merrylands, NSW 2160.

When we told the story to well known kits-n-bits entrepreneur Dick Smith, his immediate question was why we weren't



As these pictures show, Mini Scamp now really looks the part, comparing well with machines costing much more. You should be able to build it for around $105 including tax, and thanks to the new PC board it should take you only a few evenings' work!

planning to describe such an excellent design as a full-scale construction project. This would then allow his firm and, others to produce a complete kit...

Needless to say, we decided there and then to do just that. And thanks to quite a bit of help from Dick Smith, NS Electronics, C&K, RCS Radio, Radio Despatch Service and Bespoke Metalwork, we have been able to produce the full project in double-quick time.

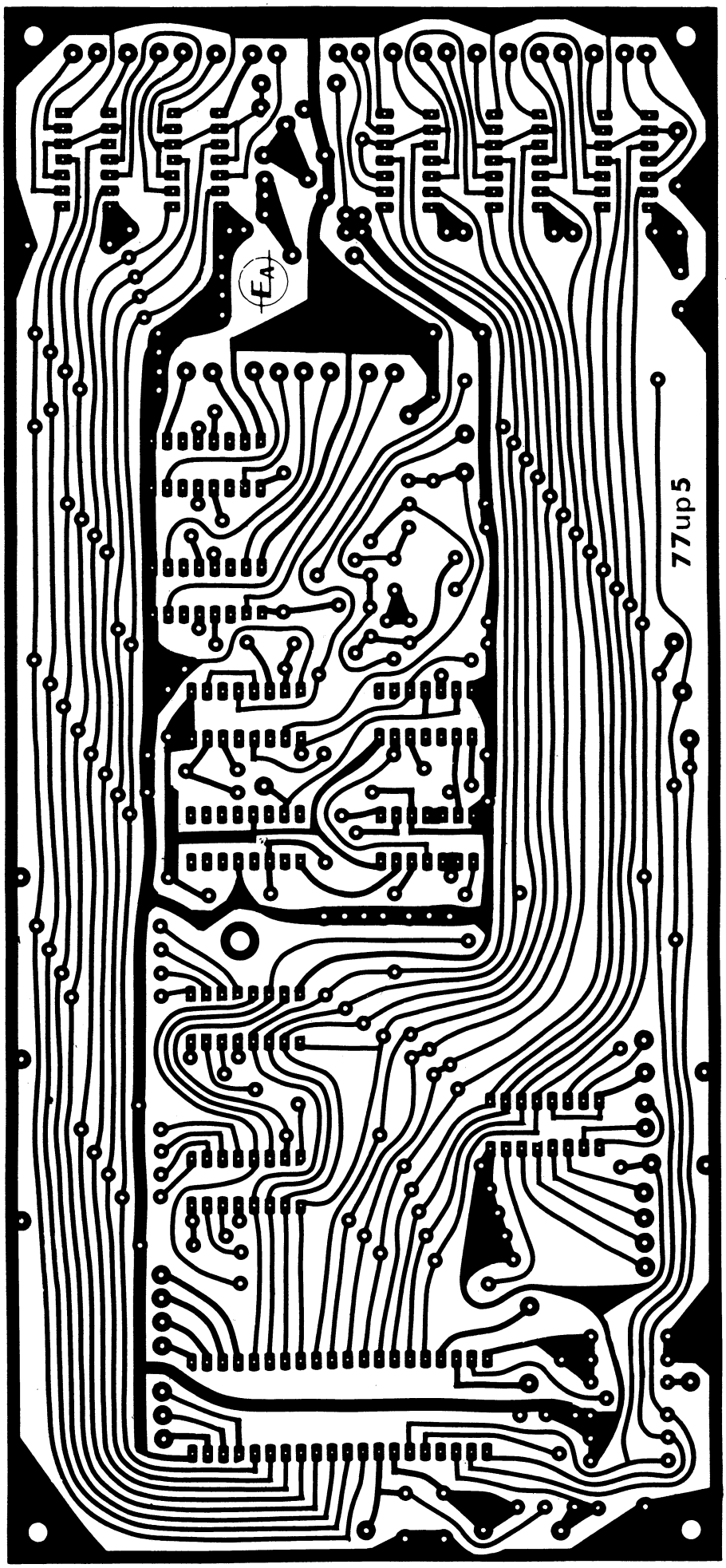
As you can see, it really looks the part, comparing very favourably with designs costing three and four times the price. Yet with most of the circuit on a single PCB measuring 254 x 117mm, you should be able to wire it up very easily in a few evenings.

The basic design has 256 bytes of RAM memory, plenty to let you cut your teeth on basic programming. However we have provided the PCB with data bus, address bus and control signal access, so that additional "outboard" memory may be added very easily. In fact all you will need to expand the memory to 1k bytes is six more 2112 memory chips, a small PCB or piece of perf-board to mount them on, and some hookup wire!

We have even allowed for a couple of additional address switches to be mounted on the front panel, if you want to expand the memory to 1k in this way.

We have also brought out all of the flag and sense pins on the SC/MP chip itself, so that it should be possible to interface Mini Scamp to a terminal later on if you wish. If there is sufficient reader interest we may be able to tell you how to add the "Kitbug" ROM into the system, with its terminal interfacing and debug routines. This should again be a relatively simple matter.

Wiring up the PCB should be quite straightforward as we have prepared an overlay diagram showing the position and orientation of all parts. There is a reasonable number of links, as the PCB is single-sided to keep the cost low, but not so many as one might have expected.



PARTS LIST FOR OUR MINI SCAMP

1 Case, 285 x 235 x 104mm
1 Printed circuit board, 254 x 117mm, coded 77up5
18 SPDT miniature toggle switches, C & K type 7101 or similar
2 Miniature pushbuttons, single pole normally open type, C & K type 8532 or similar
1 stepdown transformer, with multi-tapped 15V secondary at 1A. Type 2155 or similar

SEMICONDUCTORS

1 ISP-8A/500D microprocessor (SC/MP)
2 2112 memory chips (256 x 4)
2 74C175 quad latches
1 74C10 triple 3-input gate
1 74LS138 decoder
1 74LS05 hex inverter
5 7401 hex inverters
1 7476 dual flipflop
1 74123 dual monostable
12 BC108, BC317 or similar transistors
9 5mm diameter LEDs with panel adapters (8 red, 1 green or yellow)
1 LM340T-5, 7805 or similar 5V/1A 3-terminal regulator
4 1N914, 1N4148 or similar diodes
4 50V/1A rectifier diodes
1 6.8V/1W zener diode

RESISTORS

1W rating: 1 x 56 ohm
¼W rating: 9 x 150 ohm, 1 x 390 ohm, 1 x 470 ohm, 1 x 2.7k, 2 x 4.7k, 2 x 6.8k, 1 x 8.2k, 19 x 10k, 2 x 18k, 20 x 27k, 2 x 100k

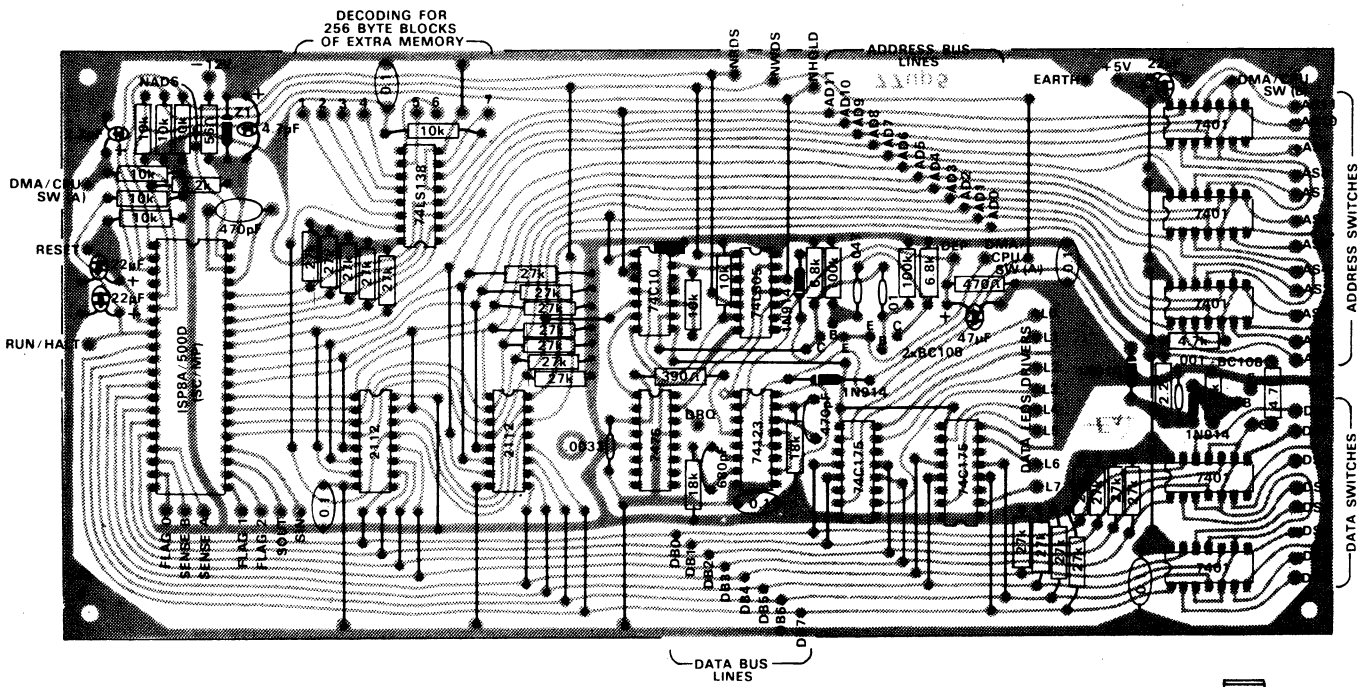
CAPACITORS

LV greencap polycarbonate: 1 x 1000pF, 1 x 3300pF, 1 x .01uF, 1 x .047uF, 5 x 0.1uF
2 470pF polystyrene or NPO ceramic
1 680pF polystyrene or ceramic
1 2.2uF 35VW tantalum
1 4.7uF 35VW tantalum
3 22uF 6VW tantalum
1 47uF 6VW tantalum
1 100uF 16VW electrolytic
3 1000uF 16VW electrolytic

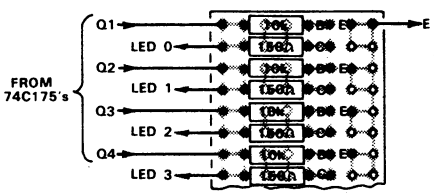
MISCELLANEOUS

Three-wire mains cord and 3-pin plug; grommet and cord clamp; 40-pin DIP socket for SC/MP (PC type); 7 x nylon PCB supports (Richco); 1 x 85 x 40mm piece of utility PCB for LED drivers; 2 x 8-lug miniature tagstrips for power supply wiring; 4 rubber feet for case; connecting wire, solder, nuts, bolts, etc.

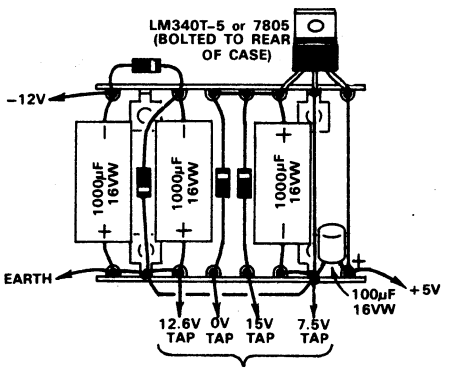
At left is the PCB pattern reproduced actual size. Etched and drilled boards should be available from the usual suppliers by the time you read this.



Wiring up the PCB should be fairly straightforward using the above diagram as a guide. Note that the address and data bus lines are brought out only for future expansion; this applies also to the flag and sense lines. Details of the LED driver and power supply wiring are shown below and at right.



HOW TO WIRE THE LED DRIVER PANEL (9 DRIVERS IN ALL)



POWER SUPPLY WIRING

Incidentally the PCB hole spacing for the SC/MP clock capacitor is 12.5mm, so that readers who wish to substitute a 1MHz quartz crystal for the existing 470pF capacitor can do so easily. This would perhaps be advisable when and if you wish to interface the Mini Scamp to a terminal, to ensure a stable and predictable data rate.

Except for the SC/MP chip itself, all of the ICs are mounted directly on the PCB without sockets. A high-quality 40-pin socket was used for the SC/MP because of its higher unit cost.

We have not included the LED driver transistors and their associated resistors on the main PCB. This would have committed builders to using the binary LED scheme, and we think that some may prefer to use a pair of 7-segment displays with hexadecimal drivers instead. By leaving the drivers off the PCB, you can take your pick.

The original 9-LED scheme has been retained for our version of Mini Scamp, as you can see. This is because we think beginners find a simple binary display easier to follow. It will also be somewhat cheaper than a hex display!

We wired the 9 driver stages on a small piece of utility PCB, of the type having an array of 4 linked-pad groups. By cutting some of the conductors to form pairs, the drivers were very easily wired, as shown in the small diagram. The piece of utility PCB measures only 85 x 40mm.

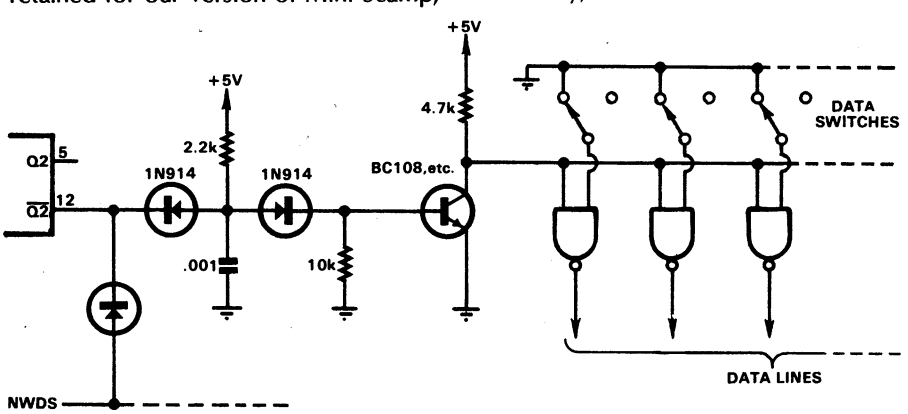
To reduce costs we elected to use one of the imported DSE 2155 transformers. As this provides only 7.5V per side on the secondary, we were unable to use the twin full-wave rectifier circuit suggested by Dr. Kennewell last month. The positive rectifier is unchanged, but we have substituted a half-wave doubler for the negative supply. This runs from the "12.6" tap (i.e., 5V with respect to the earthed centre-tap), to produce the desired –12V.

The only other change to the power supply is that we found it necessary to add a 100uF/16VW electro across the output of the 5V regulator. The wiring of the final supply is shown in a small diagram, so you can copy it if you wish.

One further point: you will find that the PCB incorporates a change to the memory deposit circuit. In particular, we have added an RC delay circuit and a buffer transistor to the drive line for the data switch gates, as shown in the diagram at left. Drive is now taken from the Q2-bar output of the 74123 (pin 12), instead of from the Q2 output.

We found this modification necessary to ensure fully reliable depositing with the 2112 memory devices, which have a critical requirement in respect to data hold time.

Well, there you have it. We think you'll agree that Mini Scamp has become quite an exciting project—and an excellent way to learn about microcomputers.



DEPOSIT TIMING MODIFICATION

Announcing the Dick Smith–Electronics Australia

Mini Scamp Contest!

How inventive are you? If you can come up with a really ingenious application for Mini Scamp, you could win a complete microcomputer system valued at over $2000...

Dick Smith Electronics and Electronics Australia magazine believe that there is a far more exciting future for micro-processors than merely providing ever more compact "number crunchers". We believe that they are going to revolutionise the electronics industry, and make it possible to perform all sorts of down-to-earth tasks which until now have seemed almost impossible. And we also believe that hobbyists can play an important and pioneering role in developing these applications, using low-cost microcomputers like our recent Mini Scamp design.

To encourage enthusiasts to dream up and develop interesting new microprocessor applications, we are launching this exciting new contest. The idea is simple—to the individual enthusiast, student or hobby club who can come up with the most intriguing and imaginative application for the Mini Scamp microcomputer, Dick Smith Electronics will award an outstanding prize: a complete "big brother" microcomputer system valued at more than $2000, shown opposite.

We're not looking for way-out academic applications, but down-to-earth practical ways of using microprocessors in the home, office or school. Like controlling a model train layout, or running a home movie show or providing a washing machine with custom programmable washing cycles...

The system you develop must use Mini Scamp, and its software must fit in less than 1280 bytes of RAM and/or ROM/PROM. It should preferably have been tried out in practice, to make sure there are no hidden bugs. Your entry should include a detailed description of system operation, a complete program listing with comments, and details of any custom interfacing you have used.

Judges for the contest winner will be Mini Scamp's designer Dr John Kennewell, entrepreneur Dick Smith and EA editor Jim Rowe. Their decision will be regarded as final.

Entries must be accompanied by the official entry form below (except in states where this requirement is illegal). Entries should be postmarked no later than September 30th, 1977, so you have three months to work on your entry. The winner will be announced in EA as soon as possible after that date.

CONDITIONS OF ENTRY: Entries should represent the entrant's original work. Employees of Sungravure Pty Ltd, Dick Smith Electronics Pty Ltd or any associated companies are not eligible to enter. Entries postmarked or delivered by hand later than September 30, 1977, will not be eligible.

ENTRY FORM

THE DICK SMITH–ELECTRONICS AUSTRALIA MINI SCAMP $2000 MICROCOMPUTER CONTEST

Complete this form and attach it to your entry, posting them not later than 30th September, 1977, to Microcomputer Contest, c/o Electronics Australia, Box 163, Beaconsfield, NSW 2014. A letter may be used instead of the form in States where this requirement is illegal.

NAME: .....

ADDRESS: .....

..... POSTCODE: .....

Here's what you can win:

* A NATIONAL SEMICONDUCTOR SC/MP DEVELOPMENT SYSTEM, WITH RESIDENT MONITOR & TINY-BASIC INTERPRETER!

This is an ideal system for anyone wanting to do more serious work with National Semiconductor's SC/MP. The resident monitor-debug program is a very powerful one, occupying 2k bytes of ROM. With it you can develop and debug programs very quickly and conveniently. The system has more than 2k bytes of RAM for user programs, too.

In addition, the system includes a 4k byte ROM card with a resident interpreter program for National's "NIBL", a very powerful extended version of TINY BASIC high level language.

With the ability to handle FOR . . . NEXT and DO . . . UNTIL commands, NIBL lets you develop complex programs so easily and so fast that you won't believe it. Ideal for computer games, too!

TOTAL VALUE, $1242.00

* TOGETHER WITH TWO STATRONICS MODULAR POWER SUPPLIES!

* AN E&M ELECTRONICS VIDEO DATA TERMINAL, COMPLETE WITH 12-INCH SOLID STATE TV RECEIVER!

Designed to work with almost any standard TV receiver or video monitor, the E & M Electronics EME-10 data terminal offers high performance and operational flexibility at an attractive price. The version supplied will display 16 lines of 64 characters, has roll-up and roll-down scrolling and a cursor to facilitate tabulation and similar functions. Data rate and format are fully programmable, although as supplied the terminal is set to suit the system at left. A very compact and reliable terminal, which provides full professional performance.

As shown the terminal will be supplied complete with a mating 12-inch solid state monochrome TV receiver.

TOTAL VALUE, $773.50

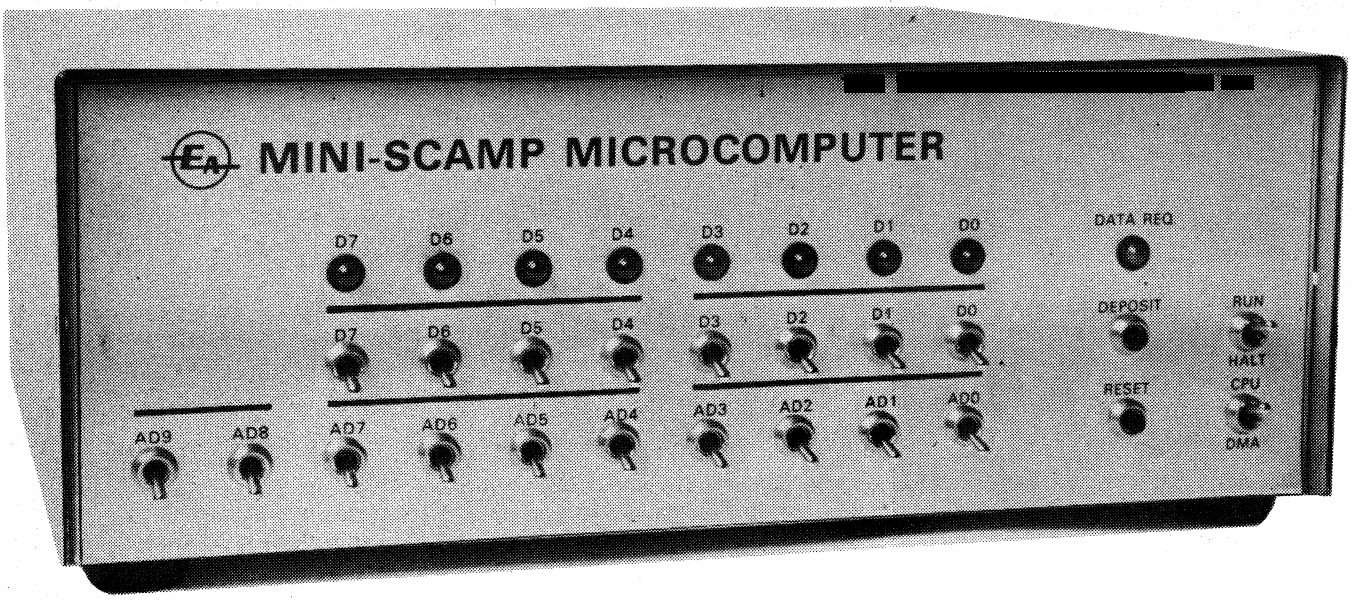
Ideal for powering microcomputer systems, the Statronics STA-53 supplies offer compact high performance with inbuilt current limiting and over-voltage protection. The 5V/3A and 12V/2A units supplied are complete with mains cords & plugs.

TOTAL VALUE, $77.62

THE COMPLETE SYSTEM, READY TO RUN, IS VALUED AT $2093.12!

Isn't that worth an effort?

SO PUT ON YOUR THINKING CAPS, AND GET CRACKING . . .



More on Mini Scamp

Judging by the tremendous interest it has generated already, our Mini Scamp looks like becoming the most popular microcomputer project ever described. This article gives details of some simple modifications and improvements to the basic design, to improve performance. It also discusses simple techniques for providing parallel interfacing.

by JAMIESON ROWE

One good thing about a popular project like Dr. Kennewell's Mini Scamp is that with so many people building it up quickly, any errors or bugs present tend to show up sooner than otherwise!

For example, the May issue had only been published a couple of days before a reader rang to point out that there was an error in the power supply wiring diagram.

If you didn't see our note last month about that error, it showed the wire from the 12.6V transformer tap as being connected to earth. In fact it should only connect to the "+" end of the 1000uF electrolytic capacitor, of course.

It took a little longer for any other bugs to show up, but after a couple of weeks we received reports of some builders having trouble with the deposit timing circuitry. Some units wouldn't load in programs properly in DMA mode, while others wouldn't load from the switches reliably under program control.

When we looked into this, we found a subtle timing problem associated with the deposit timing delay circuit shown in the May issue. With some 1N914 diodes and BC108 transistors, charge storage times were sufficient to slow down the leading edge of the pulse used to gate the data switches onto the data lines, and improper loading occurred. Usually, some or all of the bits loaded as "1's", regardless of the switch positions.

Happily, we found a simple way of fixing this trouble. All that is required is to reduce the value of the base pulldown resistor shown as 10k, to 680 ohms. This should produce reliable loading with all devices.

In the meantime, Mini Scamp designer Dr John Kennewell advised us that he had found it desirable to add a 180-ohm resistor in series with the deposit switch. This prevents the switch from taking the B1 input of the 74123 monostable right up to the 5V rail, and thus prevents the monostable from being spuriously triggered by supply line transients.

Incidentally you may have noticed that while Dr Kennewell's original circuit as shown in the April article shows a DPDT switch used for S1, performing the DMA/CPU switching, we called for only an SPDT switch in the parts list. This is because the SPDT switch is quite capable of doing the job, if its rotor is taken to earth. In the DMA position it earths the two wires from the PCB pads marked "DMA/CPU SW A", while in the CPU position it earths the single wire from the PCB pad marked "DMA/CPU SW B". Most builders seem to have deduced this for themselves, but I mention it in case you haven't got that far as yet.

If you make these few minor corrections to your Mini Scamp, you should find that it works quite well. However something which may become apparent once you begin running programs is that with the Mini Scamp as it stands, the LOAD SWITCHES instruction tends to result in an automatic STORE LEDS operation, whether you want this to happen or not.

This occurs because of the 1N914 diode tying the NWDS control line to the Q2-bar output of the 74123 monostable. The diode is used for DMA program loading, serving no useful purpose in CPU mode. However as it is still in circuit, it is able to cause spurious writing into the 74C175 LED latches, when the deposit switch is pressed as part of a LOAD SWITCHES instruction.

Again, there is a fairly simple way of preventing this from happening. All that is required is to replace the existing SPDT switch used for DMA/CPU selection, with a DPDT type. The second section of the new switch is then used to break the connection between the 1N914 diode and the NWDS line, in the CPU position.

You don't need to cut any PCB tracks to make this change, because the diode is normally connected to the NWDS line via a wire link. All you need do is cut the link, and connect the two cut ends to the appropriate lugs of the new DMA/CPU switch.

There is another modification to the basic Mini Scamp design which some builders may care to make. Although it requires a little more work, it is still fairly straightforward and involves only a handful of additional low-cost parts.

The purpose of this further modification is to ensure that only a single loading takes place from the switches when the deposit switch is pressed in CPU mode, in response to a LOAD SWITCHES instruction having lit the DRQ LED. With the existing Mini Scamp circuit multiple

loading can occur if the CPU executes another LOAD SWITCHES instruction while the deposit switch is still pressed, as the 7476 DRQ latch can re-trigger the 74123 via inverter I4. This is a side effect, because I4 is in circuit mainly to ensure that the 74123 can only be triggered in CPU mode when the DRQ latch has been set for a LOAD SWITCHES instruction. If the 74123 were to be triggered at other times, a program could be changed and caused to "crash".

The only way of preventing multiple loading with the circuit as it stands is to insert delay instructions into your program, so that one has time to release the deposit button before the CPU can get to the next LOAD SWITCHES instruction. This is not always convenient.

To obviate the multiple loading, I4 can be used to gate the 74123 by means of its clear input, instead of the A1 trigger input. This is done as shown in Fig. 1.

Note that the A1 input of the 74123 is now earthed directly, with the output of I4 now taken to the active-low clear input on pin 3. As I4 is an open-collector element, a 10k pullup resistor must be added as shown.

Diode D1 from the DMA/CPU switch is now taken to the input of I4, so that the 74123 can be enabled in DMA mode as before. The input of I4 is disconnected from the Q input of the 7476 DRQ latch, and connected to the Q-bar output via diode D2 to achieve the correct logic action. A further 10k resistor is used at the input of I4, to form a single OR gate with the two diodes.

Although this fixes the multiple loading problem, one can still get occasional faulty loads from the switches. This seems to be due to spurious triggering of the 74123 from the deposit switch, as a result of the low slope produced by the simple R-C bounce integrator.

Presumably the input of the 74123 can become unstable during the transition from logic low to logic high levels.

To fix this we suggest that you replace the deposit switch with a SPDT type, and use the originally redundant second half of the 7476 device as a debounce latch, as shown in Fig.1. This gives completely reliable operation.

Note that Fig. 1 also shows the optional switching between diode D3 and the NWDS line, and the 680 ohm base resistor in the deposit timing delay circuit.

To make these suggested modifications, you need to cut a small number of PCB tracks and add a few links under the PC board. We suggest you do this in the following order, to make sure that you don't lose track of anything.

1. change the 10k resistor at the base of the BC108 used to delay the 74123 Q-bar output signal fed to the data switch control gates to 680 ohms.
2. cut the track between pin 1 of the 74123 and pin 10 of the 74LS05.
3. cut the track between pin 10 of the 74LS05 and the anode of D1.
4. cut the track linking pin 11 of the 74LS05 to the track joining pin 13 of the 74LS05 and pin 15 of the 7476.
5. remove the 470 ohm resistor and 47uF tantalum electrolytic capacitor connected to PCB point "DEP".
6. unsolder the wire connecting the deposit switch to the PCB point "DEP", at the deposit switch.
7. replace the deposit switch with a momentary contact SPDT push-button (C&K type 8121 or similar).
8. connect the NO and NC contacts of the deposit switch to +5V rail with two 10k resistors (at the deposit switch).
9. connect the common contact of the deposit switch to GND.
10. drill two holes near pins 7 and 8 of the 7476, and connect these pins to the NO and NC contacts of the deposit switch respectively.
11. drill a hole near pin 11 of the 7476, and connect the loose wire attached to PCB point "DEP" to pin 11 of the 7476.
12. connect pin 3 of the 74123 to pin 10 of the 74LS05.
13. connect pin 1 of the 74123 to pin 8 of the 74123 (i.e., earth).
14. connect the anode of D1 to pin 11 of the 74LS05.
15. connect a 10k resistor from the anode of D1 to the +5V rail (pin 14 of the 74LS05).
16. connect the anode of additional diode D2 to pin 11 of the 74LS05, and the cathode to pin 14 of the 7476.
17. connect a 10k resistor from pin 10 of the 74LS05 to the +5V rail (pin 14 of the 74LS05).
18. join pins 2, 4, 6, 9, 12 and 16 of the 7476 to pin 11 of the 74123, and then connect them to the +5V rail via a 1k resistor.

With these modifications, your Mini Scamp should leave nothing to be desired in terms of its internal operation. But perhaps a few words are now in order for the benefit of those readers who are planning to use their Mini Scamp to control external devices.

For simple applications, you may be able to use direct interfacing to the flag and sense lines of the SC/MP chip itself, rather like that used for the serial interface shown last month. The SC/MP chip has three flag outputs and two sense inputs, and also two other pins designated "serial in" and "serial out". This allows for a total of seven input-output lines.

Fig. 2 shows how one of the flag lines or the SOUT line could be used to control a small relay. It also shows how a SPDT switch can be debounced and fed to one of the sense inputs, or the SIN input.

For more complex applications, you may wish to provide Mini Scamp with full 8-bit parallel interface ports. This can be done fairly simply, providing you can write your program so that it isn't worried

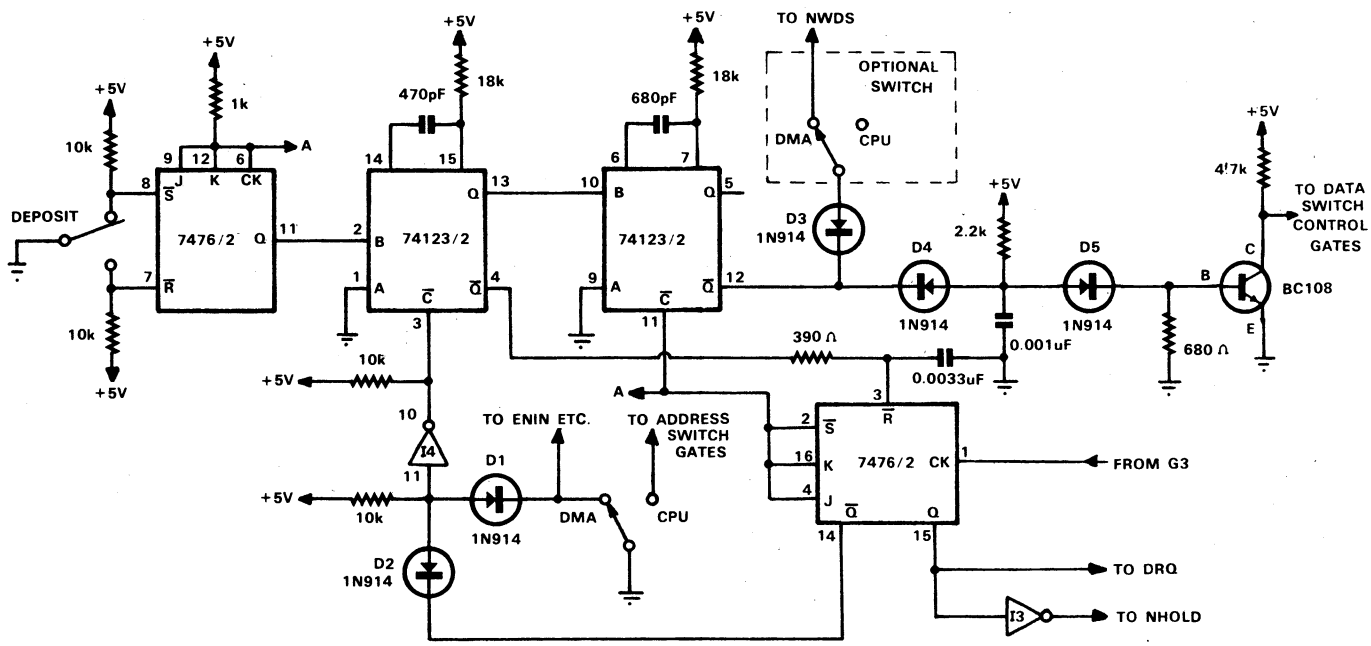


FIG. 1: MODIFIED MINISCAMP DEPOSIT SWITCH LOADING LOGIC

File No. 8/M/—

MINI SCAMP

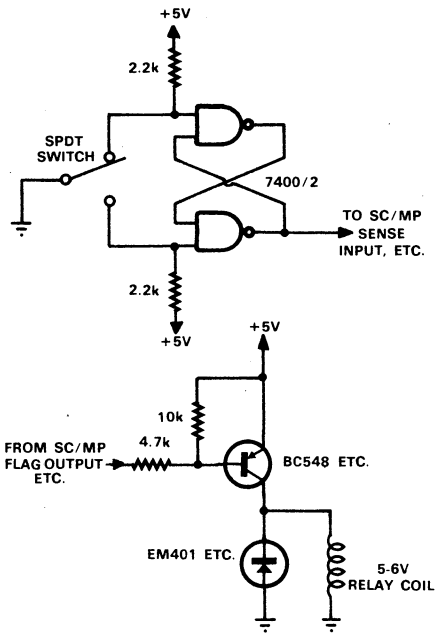


FIG. 2: SIMPLE RELAY & SWITCH INTERFACING

The three diagrams above and at right should give you some guidance on providing your Mini Scamp with interfacing to the outside world.

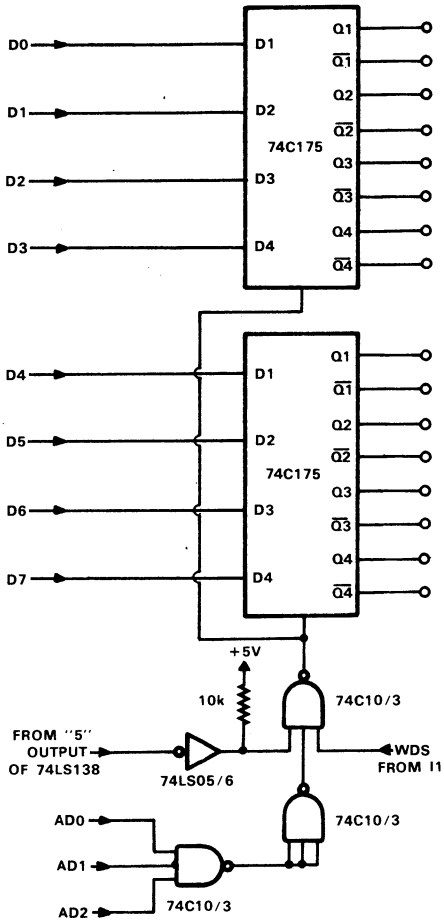


FIG. 3: SIMPLE LATCHED 8-BIT OUTPUT PORT (ADDRESS 507 HEX)

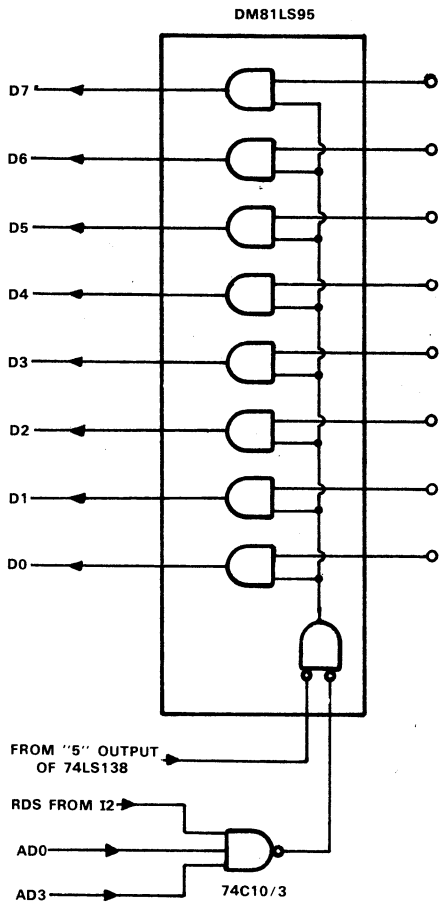


FIG. 4: SIMPLE 8-BIT INPUT PORT (ADDRESS 511 HEX)

by the ambiguous addressing which results from incomplete decoding.

A simple latched 8-bit output port is shown in Fig. 3. Two 74C175 devices are used for the latch itself, providing both active-low and active-high outputs for the eight bits. These outputs could be used to drive external circuits via buffers like those used for driving the LEDs in Mini Scamp itself.

As you can see, the address decoding is quite simple, combining the "5" output from the 74LS138 decoder with the AD0, AD1 and AD2 address line signals to give the port an effective address of 507 hex. When a STORE instruction specifying this address is executed, the WDS pulse from I1 is allowed to reach the clock inputs of the latches, writing the data into them.

Because the gating does not sense address lines 3, 4, 5, 6 or 7, the port effectively occupies other addresses besides 507. For example it could also be addressed as 50F, 517, 51F, 527 and so on. This shouldn't cause any strife as long as you remember it when writing your programs.

The same sort of simplified addressing is used in the 8-bit input port shown in Fig. 4. Here only a single 74C10 gate element is used, in conjunction with an internal two-input enabling gate provided in the DM81LS95 octal buffer. RDS pulses from inverter 12 are only effective in enabling the buffer when address lines AD0 and AD3 are high and the "5" output of the 74LS138 decoder is low, corresponding to address 511 hex.

Hence you can read the data from the port into the SC/MP accumulator by executing a LOAD instruction which references address 511. But because of the simple decoding, you can also regard the port as having address 513, 515, 517, 51B, 531 and so on.

Both of the circuits shown in Fig. 3 and 4 impose very low loading on the lines SC/MP address, data and control lines so you should be able to use them with Mini Scamp even if you have expanded the memory. However if you want to add a number of different ports along these lines, you may have to add extra buffering to ensure reliable operation.

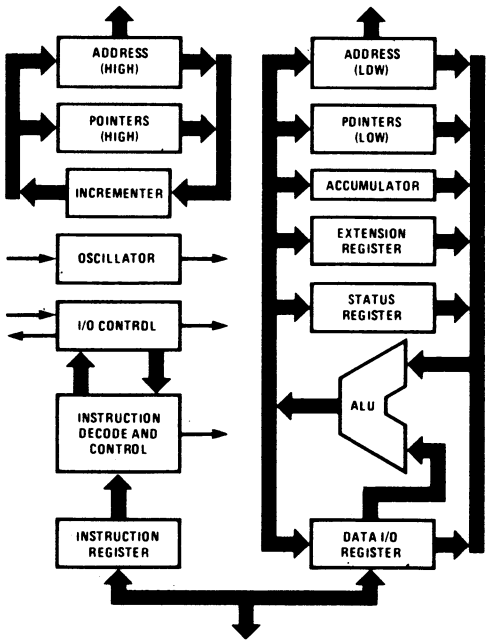
NO LOADING IN CPU MODE?

A small number of readers have found that while their deposit switch functions correctly in DMA mode, it will not operate in CPU mode. This appears to be due to excessive voltage drop across the 390 ohm resistor linking the Q1-bar output of the 74123 (pin 4) to the reset input of the 7476 DRQ latch (pin 3). This prevents the DRQ latch from being reset when the deposit switch has been operated.

The cure is to reduce the resistor to 220 ohms and increase the value of the associated capacitor from 0.0033uF to 0.0068uF.

Our fast low pitch.

SC/MP II microprocessor



Introducing a new "fast and low" version of our SC/MP: SC/MP II.

Fast! *Twice* as fast.

And low as you can get. In fact, the lowest-power n-channel MPU on the market. 225mW typical.

Another plus is that it now requires only a single + 5V power supply.

What doesn't change are the things that made SC/MP so almost-irresistable in the first place. Multiprocessing capability built-in, easy expandability, low parts count, built-in serial I/O register, control flags and sense inputs, built-in delay timer instruction, the availability of good design tools, and a training program that teaches you how to use them.

It's available off your distributor's shelf. And of course, it's cheap. (Otherwise, why would we call it our Simple Cheap Micro-Processor?)

And that's our pitch.

N.S. Electronics
A division of N.S. Distributors Pty. Ltd.
P.O. Box 89,
Bayswater, Vic., 3153

Please send details of your
- ☐ SC/MP II (ISP-8A/600D)
- ☐ SC/MP II Training Program

Name_____
Company_____
Address_____
City_____
State_____ P.C._____

National Semiconductor manufactures a full line of microprocessors, including the 8-bit INS 8080A, the single-chip 16-bit PACE, the bit-slice 16-bit IMP, the four-bit INS 4004, and low cost COPS microcontrollers.

National Semiconductor Microprocessors

Assemble your own Mini Scamp programs!

Once you have built the Mini Scamp, the next job is to write some programs and run them. In most cases this will have to be done the long way, using hand coding and entry. For those who haven't done this before, the following article will show you how.

by PETER LAZARUS*

Simple microcomputers like Mini Scamp can only run programs which are in the form of so-called "machine language". This is really nothing more than a string of 8-bit binary numbers, which are stored in consecutive memory locations. Most of the 8-bit numbers are code numbers representing particular instructions in the microprocessor's repertoire.

It is not all that easy for we humans to visualise a program in its machine language form, however, so programs are actually written in what is called a "symbolic language". The very simplest type of symbolic language is where two-digit hexadecimal numbers are used to represent the 8-bit machine language code numbers; as you might expect this is only slightly more convenient than true machine language.

A more convenient type of symbolic language is one where short easily remembered mnemonic words are used to represent each type of machine instruction. Words like "STR" to represent a store instruction, "LD" to represent a load instruction, and "JMP" to represent a jump instruction. A program written in this sort of symbolic language is very much easier to visualise and follow from a human point of view.

Of course after a program has been written and checked in a mnemonic language of this type, it must still be translated into the equivalent machine language code understood by the computer. This translation into machine language is known as "assembly".

With larger computers this assembly can be done by the computer itself, under the control of a specially-written "Assembler" program. Or it can be done by another computer altogether, under the control of a suitable "Cross Assembler" program.

For those with access to a larger computer, assembling programs can thus be quite easy. But for those of us who haven't got access to a larger machine, the task has to be done the long way. This article aims to show you how to do this for yourself, for Mini Scamp and other small microcomputers based on SC/MP.

First of all, you will need to write your program in mnemonic language. If you haven't done this before, I recommend that you get the SC/MP Programming and Assembler Manual published by National Semiconductor (Publication Number 4200094B). It costs around $10, but is virtually essential for serious programming—particularly if you haven't any previous experience.

The lower cost SC/MP Technical Description (Publication Number 4200079A) has some information on instruction formats, and an instruction summary, but this isn't really enough unless you have a fair amount of previous programming experience.

Both publications are available from National distributors, and also from suppliers like Dick Smith Electronics and Radio Despatch Service.

After writing a program, and before attempting to assemble it into machine language, you should take a sheet of paper and execute the program yourself, pretending to be the microprocessor. Follow each instruction in the program literally, writing the result at each step. In this way you should find any errors, and be able to correct them. It is important to remove as many mistakes as you can at this stage, as correcting them later can involve much more effort.

Now we are ready to assemble the program into machine language. To do this the operation code or "op-code" for each instruction must be found, together with the number of 8-bit words or "bytes" involved for each instruction.

To serve as an example, I will use the simple program given in the April Mini Scamp article. This is reproduced in Fig.1.

A worksheet should now be drawn on a piece of paper. The program should be copied onto it in the space on the right, labelled source code. The worksheet format is shown in Fig.2. For convenience a list of the particular SC/MP instructions used in the sample program, their opcodes and formats are shown in Fig.3.

The first task is to write the opcodes of the instructions in the Code column. At the same time the length column of the worksheet can be filled.

The first instruction is NOP. From Fig.3, we determine that the opcode for this is X'08 (meaning 08 hexadecimal) and the instruction length is 1. So put "08" in the code column, and "1" in the length column.

The second instruction is a load immediate with an opcode of X'C4 and a length of two bytes. So put "C4" in the code column, and "2" in the length column.

So far the process has been simple. The next, Exchange Pointer High (XPAH) instruction has a basic opcode of 34 and

```
        *BINARY COUNT AND DISPLAY.

              NOP
              LDI       8
              XPAH      1
              LDI       0
              XPAL      1
LOOP          ST        2(1)
              DLY       255
              ILD       COUNT
              JMP       LOOP
COUNT         .BYTE     0
```

Fig. 1: Dr. Kennewell's counting program used here as a sample for assembly.

*62 Collinson St, Keilor Park, Victoria 3033

a length of one byte. However this one is different to the previous ones. The opcode must be modified to contain the pointer register number. (When 'disp (pointer)' is shown in the Format column, the pointer register number is the one written in brackets.) For all other instructions having 'pointer' in the format shown in Fig.3, the opcode is similarly changed. In the case of Jump if Not Zero (JNZ) for example, the basic opcode is X'9C, but if pointer two is to be used, it becomes X'9E.

The instruction XPAH specifies pointer 1, so its opcode becomes X'35.

We follow the same procedure for all the other instructions in the sample program.

Address calculation is the next step. Now that we know the length of each instruction, we can calculate the starting locations or addresses of our instructions.

Using the SC/MP MPU, our program should begin at address zero, as the SC/MP will automatically start here after reset is pressed.

Place zero in the address column next to the first instruction. Add the first instruction length to this to get the address of the next instruction, and so on. Note that the address must be written in hexadecimal, not decimal. The hexadecimal address can be directly keyed on the Mini Scamp address switches, while decimal values would all have to be converted.

The data can be handled in the same way as the instructions were. Next to each .BYTE we place the value shown on the right. In the example, .BYTE results in a code of 00. For decimal eleven, we would code X'0B (.BYTE 11) i.e., decimal 11 converted to hexadecimal 0B. Alternatively, .BYTE expressions can be written directly in hexadecimal, such as X'7D. And in this case, the code is the same, X'7D. After writing the code for the .BYTE locations, the address can be calculated. Each .BYTE occupies one byte—i.e., a single 8-bit memory location.

The worksheet will now look like Fig. 4. This completes the first stage or "pass" of the assembly.

The second stage is to resolve labels and displacements. Labels are the names given, for convenience, to statements or locations in the mnemonic language version of the program. The example uses two such labels, 'LOOP' and 'COUNT'. Each label appearing in the operand area has to be changed to a hexadecimal code specifying its location in memory.

Displacements when specified in the Format column of Fig.3, usually represent the change of a label to a code defining its location. Displacements can also be explicitly defined, as in the DLY instruction in the example. Here, the 255 is the displacement part of the instruction, used to specify the period of the delay.

Firstly, let's handle explicit displacements. The two LDI instructions have a displacement of 8 and 0. These each occupy one byte, and are written in the code column as 08 and 00. Instructions XPAL and XPAH do not have displacements. The '1' is the pointer register, and that has been handled previously. The 'ST' has a displacement of 02, and finally the 225 displacement in the DLY instruction has to be changed to its hexadecimal equivalent of FF.

To calculate label displacements we must distinguish between two kinds. Firstly there are the labels of data areas referenced by load or store instructions, etc., and secondly there are labels used in Jumps.

Labels used in jump instructions must be changed to a displacement corresponding to the address BEFORE the one actually required. This is because the SC/MP increments its Program Counter just before fetching the next instruction.

The rule for determining the displacement for Jump instructions is:

disp = (address of label) − 1
+ two's complement of
(address of instruction + 1)

For all other label displacements the rule is:

disp = (address of label)
+ two's complement of
(address of instruction + 1)

Although these are expressed in binary notation, it is more convenient to think in hexadecimal. To calculate the two's complement of a hexadecimal number, subtract each digit from X'F, write the result, and then add one. For example, to find the complement of X'25: 15−2=13 (X'D), 15−5=10 (X'A), giving X'DA, then add one giving X'DB.

Now let's calculate the displacement of label 'LOOP'. This is a Jump instruction, so the first rule applies.

Address	Code	Length	Source Code

Fig. 2: (above): Suggested format of a worksheet for hand assembly of programs.

Fig. 3 (right): Op codes and formats for the instructions in the sample program, for reference.

Instruction and Format	Length	Opcode	Description.
DLY displacement	2	8F	Delay
ILD disp(pointer)	2	A8	Memory Increment and Load
JMP disp(pointer)	2	90	Jump
LDI displacement	2	C4	Load Immediate
NOP	1	08	No Operation
ST disp(pointer)	2	C8	Store
XPAH pointer	1	34	Exchange Pointer Low
XPAL pointer	1	30	Exchange Pointer High

Address of label: 0007
Address of instruction + 1: 000E
Two's Complement FFF2
Add: FFF9
Subtract 1: − 1
Result (take last two digits): FFF8
So the displacement is X'F8.
For the label 'COUNT' use rule two.
Address of label: 000F
Address of instruction + 1: 000C
Two's Complement FFF4
Add: 0003
Result (take last two digits): 03

Note that in both types of displacement calculation, if the first two digits are anything other than X'00 or X'FF, then the displacement is greater than 127, and with SC/MP a different addressing scheme must be used. Also a displace-

ment of X'80' (−128) is not to be used, as SC/MP will use the extension register for the displacement.

In these examples the pointer register is zero (none was specified in brackets) indicating the Program Counter. If a pointer register is to be used, then the address of the instruction plus one is to be replaced by the address loaded into the pointer register. This applies to both the above rules.

Now our program is fully assembled. It will look like Fig. 5. To enter into the memory, we consider only the address and code parts of the worksheet. Entering programs into the Mini Scamp was described in Dr. Kennewell's first article, in the April issue.

Lastly, a word about program alterations. If you want to change an instruction, it can be done provided the new instruction length is equal to or shorter than the original. If the length is equal, then change the hexadecimal code to reflect the new instruction. If the length is shorter, the new instruction occupies the first byte and a NOP (X'08) can be used to occupy the second byte.

To add extra instructions in the middle of a program can involve a lot of work—you have to re-assemble the whole program again! The easy way is to add them at the end of the program, and provide a Jump at the point you want them executed. Say for example we wanted to add three extra instructions after label 'LOOP' in our example. The three extra instructions would be added at the end (address X'0010). The DLY instruction can be replaced by a JMP instruction to transfer control to address X'0010.

Address	Code	Length	Source Code		
0000	08	1		NOP	
0001	C4	2		LDI	8
0003	35	1		XPAH	1
0004	C4	2		LDI	0
0006	31	1	LOOP	XPAL	1
0007	C9	2		ST	2(1)
0009	8F	2		DLY	255
000B	A8	2		ILD	COUNT
000D	90	2		JMP	LOOP
000F	00	1		BYTE	0

Fig. 4: How the worksheet for the sample program should look after op codes, lengths and addresses have been added to the source code.

We have to add the DLY back again before the three new instructions, and add another JMP at the end to return to X'000B. The program would look like:

```
LOOP    ST              2(1)
        JMP             EXTRA
RETN    IDL             COUNT
        .
        .
COUNT   .BYTE           0
EXTRA   DLY             255
        Extra instructions (3)
        JMP             RETN
```

This technique can be particularly handy for temporary repairs to a program, to get it going. Whether you leave the "patches" in permanently, or rewrite the program later to make it more elegant, is up to you.

Now it's your turn to write and assemble some programs. Try simple programs of no more than 20-30 statements at first, as you could quickly get discouraged attempting larger ones initially. You might attempt assembly of the other example given in the April issue, to check your understanding.

Address	Code	Length	Source Code		
0000	08	1		NOP	
0001	C4 08	2		LDI	8
0003	35	1		XPAH	1
0004	C4 00	2		LDI	0
0006	31	1		XPAL	1
0007	C9 02	2	LOOP	ST	2(1)
0009	8F FF	2		DLY	255
000B	A8 03	2		ILD	COUNT
000D	90 F8	2		JMP	LOOP
000F	00	1	COUNT	.BYTE	0

Fig. 5: The worksheet for the sample program when fully assembled, with all displacements added in the code column.

Resident Assembler for the SC/MP LCDS

National Semiconductors has announced the release of a line-by-line resident assembler for the SC/MP Low Cost Development System (LCDS). Known as SUPAK, the assembler comes in eight 512-byte PROMs or ROMs, which plug into a standard ROM/PROM card.

In the 4k-byte firmware package are actually three programs: a line-by-line assembler, a paper tape line editor and a PROM tape punch program.

The line assembler accepts a program written in limited SC/MP assembly language from a keyboard or paper reader, and assembles it directly into RAM. The editor allows insertion, deletion or replacement of lines in program source code, while the PROM tape punch will punch out a selected part of RAM for PROM programmers such as the DATA I/O, in appropriate format.

Priced at $300, SUPAK will be available shortly from NS distributors.

Using Mini Scamp to generate random numbers

Here is another article by the original designer of the Mini Scamp project, this time to help you become proficient at programming. It explains how a computer may be used to generate random and pseudo-random numbers, and gives a Mini Scamp program which demonstrates pseudorandom number generation.

by DR JOHN KENNEWELL
Physics Department, Newcastle University

A program that generates a sequence of random numbers finds many applications in the world of computing. Such programs were used in one of the first electronic computers ever constructed, at the Los Alamos laboratories during the second world war. Here, a technique known as 'Monte Carlo simulation' employed random numbers to calculate the critical masses of uranium or plutonium needed for a nuclear explosive device.

Many games programs employ random number generators. These enable a computer to simulate the tossing of a coin, the throwing of dice, or the choosing of a card. To a certain extent random number generators can be used to make a computer appear more intelligent, or perhaps we should say more human. This is achieved by having not one, but a list of possible responses to any given situation, and then using a random number to decide which of these responses will be actually given at any particular time.

In larger computers most random number programs use the mathematical expression

$$R_{n+1} = (P \times R_n) \text{ modulo } Q$$

to generate a sequence of random numbers. R_n is the last random number calculated, and this is used to produce the next random number R_{n+1}. To start with, R_n can be put equal to any number, and this number is termed the 'seed', from which all later numbers will 'grow'. P is a suitable prime number and Q is usually 2^N where N is the number of bits per word in the computer (e.g., N = 8 for Mini Scamp). The expression 'modulo' means that the product $P \times R_n$ is divided by Q and only the REMAINDER is retained. For example 9modulo5 = 4 or 7modulo3 = 1. This type of modulo division is done very simply if $Q = 2^N$ by simply ignoring the fact that overflow has occurred in the multiplication of $P \times R_n$.

The above procedure, while quite simple, and not impossible to program for Mini Scamp, does require a multiplication routine, which could use up a considerable number of memory locations. An alternative random number generator can be obtained by simulating a shift register with feedback (see Fig. 1). Starting with any number except zero in the shift register, the next number in the sequence is obtained by shifting all bits in the register one place to the right. Bit 7 will be lost, and bit 1 is formed by EXCLUSIVE-ORing bits 6 and 7 (termed the feedback bits) of the initial number. Following numbers in the sequence are generated by simply repeating the above procedure.

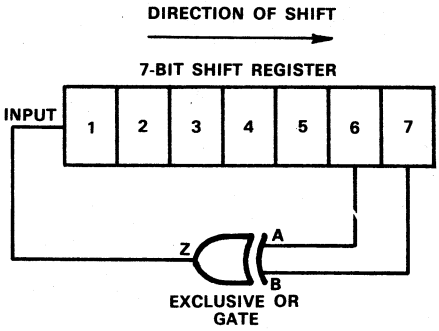


Fig. 1: A 7-bit pseudorandom sequence generator as implemented in hardware.

In point of fact the above method does not produce truly 'random' numbers. It would be more correct to say that it generates a pseudorandom sequence of numbers, because given any starting number, the sequence will always be fixed. In the case of a 7-bit register, the sequence will consist of 127 different numbers before it then starts to repeat.

It turns out however, that such a pseudorandom sequence is more useful in programming than a truly random sequence, particularly when first testing a program. Imagine testing a program that uses totally RANDOM numbers. Every time you ran it, you would obtain different results. Under these conditions it would be very difficult to tell whether the differences were due simply to the random numbers, or to a fault in your program. With a pseudorandom sequence you know that, as long as you start with the same 'seed' number each time, the results will be repeatable.

As long as the sequence is made large enough, a limited number of values from the sequence will always appear random. The randomness of such a sequence can also be increased by considering only a smaller number of bits than are used in the shift register (e.g. the lower 8 bits of a 15-bit register).

A program to implement the above-mentioned procedure is given here for the Mini Scamp. The accumulator is used as the shift register. Excluding the NOP instruction, the next four instructions are concerned with loading pointer register one with the base address for the LED's (X'0800). In this way, each random number generated can be displayed on the front panel. The next instruction loads the accumulator with the seed number which has been planted in location 'X0028, and as long as this is not zero the

INPUT A	INPUT B	OUTPUT Z
0	0	0
0	1	1
1	0	1
1	1	0

Here is the truth table definition of the exclusive-OR function, for reference.

Shift Register Length (bits)	Feedback Bits	Sequence Length
3	2,3	7
4	3,4	15
5	3,5	31
6	5,6	63
7	6,7	127
8	2,3,4,8	255
9	5,9	511
10	7,10	1023
11	9,11	2047
12	2,10,11,12	4095
13	1,11,12,13	8191
14	2,12,13,14	16383
15	14,15	32767
16	11,13,14,16	65535

Fig. 2 (left): how to make pseudorandom generators using shift registers of different lengths.

At right is the listing for the author's program to duplicate the function of the circuit of Fig. 1.

```
                        *RANDOM NUMBER GENERATOR
0000 08                 NOP
0001 C400               LDI   0
0003 31                 XPAL  1
0004 C408               LDI   8
0006 35                 XPAH  1
0007 C020   RANDOM      LD    RND
0009 9C02               JNZ   START
000B C401               LDI   1
000D D403   START       ANI   X'03
000F 9808               JZ    ZERO
0011 FC02               CAI   2
0013 9404               JP    ZERO
0015 C480               LDI   X'80
0017 9002               JMP   FIN
0019 C400   ZERO        LDI   0
001B D80C   FIN         OR    RND
001D 1C                 SR
001E C809               ST    RND
0020 C902               ST    2(1)
0022 8FFF               DLY   255
0024 8FFF               DLY   255
0026 90DF               JMP   RANDOM
0028   43   RND          BYTE  67
```

bit to be fed back to bit 1 can then be computed. If however, the seed is zero, a 'lock-out' condition will occur, as the EXCLUSIVE-OR of 0 and 0 is always 0. To avoid this possibility a value of 1 is thus placed in the accumulator.

The ANI X'03 instruction is now used to mask off all bits except 6 and 7, the lowest significant bits, as these are the feedback bits. Note that the most significant bit of the accumulator is ignored as far as the random numbers are concerned, leaving the 7 lower bits as required.

After the mask has been applied, the accumulator may contain any number from 0 to 3 inclusive. If it is zero, then the new bit one will be zero (JZ ZERO). If 2 is subtracted from the accumulator (CAI 2) and the result positive (initial number would have been 3 in this case) then the bit fed back will again be zero (JP ZERO). If neither of these conditions is true, then either bit 6 or bit 7, but not both, would have been one, and the bit to be fed back should be one. This bit is loaded temporarily into the most significant bit of the accumulator (either by LDI X'80 or LDI 0, whichever is appropriate) OR'ed with the original random number (OR RND) and then shifted right (SR) one place.

The newly created number then replaces the old random number in location X'28 and is also displayed on the LED's (ST 2(1)). The delay instructions slow the sequence down sufficiently for you to observe what is happening.

If you wish the program to stop after each new number generated, then replace the second delay instruction at location X'24 by the instruction LD 1(1) which has the hex code C101. This will cause the machine to hold with the DRQ light turned on until you press the deposit button. What you deposit into the accumulator is unimportant, as it is ignored by this program. Using this method you can either write out the complete sequence, or simply determine mentally what the next number in the sequence should be, and then press deposit to test your prediction.

You will probably note that if you keep your finger on the deposit button, the machine as originally described will continue to sequence as before (actually faster, since one delay has been omitted). Thus, if you only want one number at a time, you must depress and release the deposit button quickly (in less than one-quarter of a second). The reason for this behaviour is that, with deposit activated, the mono in the circuit will be continuously enabled, and if the SC/MP CPU requests a new data value (as it will each time around the program loop), it will have one provided it immediately without having to go into the hold state. This won't happen however, if you have modified your Mini Scamp as shown in the July issue.

It is possible to write a similar program to simulate shift registers of any other length desired. Fig. 2 shows the appropriate feedback bits to use, and the length of the sequence generated for other length registers. After becoming familiar with the program presented here, you might like to try your hand at a program for a longer sequence.

A register length of 15 is quite easy to implement using two 8-bit words. The word containing the lower 8 significant bits is subject to exactly the same test as the for the 7-bit register. However, the bit to be fed back is placed in the MSB of the word containing the higher significant bits. Then after clearing the carry/link (CCL), a rotate right with link (RRL) is performed on this word, causing the LSB of the word to be shifted into the carry/link register. If now a shift right with link (SRL) is performed on the other word, this bit will be shifted into this lower word. The procedure is illustrated in Fig. 3. In this way it is possible to simulate a register of any length whatever.

In actual use, a program such as this would be written as a subroutine in part of a larger program. Each time the main body of the program called the subroutine, it would calculate a new random number for use by that program. In many cases, only a yes/no type decision may be required, and thus only one particular bit of the random number need be considered.

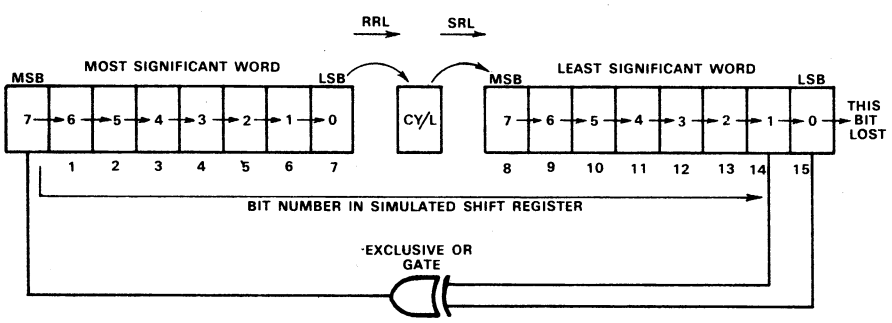


Fig. 3: Suggested way of simulating a 15-bit sequence generator.

More programs for MINI SCAMP

With many hundreds of Mini Scamps now in operation, quite a bit of software is being generated. Here are some useful utility routines which have been sent to us by interested readers.

The first routine comes from a reader in Cheltenham, Victoria, Mr C. B. Curnow. It is designed to solve one of the main problems in hand assembly of programs: calculation of instruction displacements. Needless to say this saves time and tempers, as well as obviating possible errors.

Mr Curnow introduces the routine as follows:

"Calculating short positive displacements by hand is a relatively simple job. However, when the displacement is negative or greater than the base of 16, one has to be careful to avoid mistakes. I found the article by Peter Lazarus in the August edition of EA of particular help in this regard. Nevertheless calculating displacements in anything but the shortest programs can become tiresome. I hadn't been doing this long before I decided to use the computer itself to help me by doing all the hard work.

"With the program enclosed it is simply a matter of feeding in the address of the instruction, the address of the label, and whether the displacement is for a jump instruction or not. These are fed in via the data switches in succession, in response to the DRQ light. The LEDs will then display the appropriate displacement, which can be recorded.

"The heart of the program is the set of instructions LDI 0, SCL and CAD, which calculate the two's complement of the required number. I have found this little program saves a lot of time when writing programs, and thought it might be of interest to other readers."

The remaining two routines come from a reader in Slacks Creek, Queensland, Mr Mike Nicholls. The routines are designed to be used together, in order to print or display data stored in the Mini Scamp memory at high speed: 9600 baud. Mr Nicholls introduces the routines as follows:

"I thought readers might be interested in a 9600 baud print routine for Mini Scamp and other SC/MP systems. It may be particularly useful to those people who, like myself, have a SC/MP system connected to their video terminal using the EME-1 display module. The program is quite original, being the result of a couple of late nights spent juggling with microcycles and the SC/MP instruction set.

"The main problem encountered in trying to output information at this speed is lack of time to process the data. At 9600 baud, the smallest element is 104us wide. With a 1MHz clock on the SC/MP this means only 52 microcycles in which to serialise data out of the system, and also keep a

```
                ;CALCULATE DISPLACEMENT

0000 08           NOP
0001 C408         LDI     8
0003 35           XPAH    1
0004 C400         LDI     0
0006 31           XPAL    1
0007 C101 NEW:    LD      1(1)
0009 C824         ST      LABEL
0008 8FFF         DLY     255
000D C101         LD      1(1)
000F C81F         ST      INSTR
0011 A81D         ILD     INSTR
0013 03           SCL
0014 C400         LDI     0
0016 F818         CAD     INSTR
0018 F015         ADD     LABEL
001A C815         ST      ADDR
001C 8FFF         DLY     255
001E C101         LD      1(1)
0020 9C04         JNZ     R1
0022 C00D RO:     LD      ADDR
0024 9002         JMP     DISP
0026 B809 R1:     DLD     ADDR
0028 C902 DISP:   ST      2(1)
002A 8FFF         DLY     255
002C 9009         JMP     NEW
002E 00   LABEL:  .BYTE   0
002E 00   INSTR:  .BYTE   0
0030 00   ADDR:   .BYTE   0
0031

          ;ENTER DATA WITH REQUEST
           1.ADDRESS OF LABEL
           2.ADDRESS OF INSTRUCTION
           3."1" FOR JUMP INSTRUCTIONS
             "0" FOR ALL OTHERS
```

The routine sent in by C. B. Curnow. It saves time when writing programs by using the computer to calculate the instruction displacements.

```
                ;FETCH CHARACTER

08                NOP
C413              LDI   19
C823              ST    COUNT
C4FF  LOOP:       LDI   255
8FFF              DLY   255
B81D              DLD   COUNT
9CF8              JNZ   LOOP
C4XX              LDI   XX
35                XPAH  P1
C4YY              LDI   YY
31                XPAL  P1
C4AA  NEXT:       LDI   AA
37                XPAH  P3
C4BB              LDI   BB
33                XPAL  P3
C501              LD    @ 1 (1)
9804              JZ    OUT
3F                XPPC  P3
90F3              JMP   NEXT
C400  OUT:        LDI   0
37                XPAH  P3
C400              LDI   0
33                XPAL  P3
3F                XPPC  P3
0000  COUNT:      BYTE  0

          ;XX IS HIGH ORDER BYTE OF DATA FIELD STARTING ADDRESS
          ;YY IS LOW ORDER BYTE OF SAME
          ;AA IS HIGH ORDER BYTE OF "PRINT 9600 BAUD" START ADDRESS
          ;BB IS LOW ORDER BYTE OF SAME
```

```
                ;9600 BAUD PRINT

08                NOP
01                XAE
C401              LDI   1
07                CAS
C480              LDI   -128
78                CAE
01                XAE
08                NOP
C4F7              LDI   -9
01        LOOP:   XAE
07                CAS
04                DINT
1C                SR
01                XAE
F401              ADI   1
9CF7              JNZ   LOOP
3F                XPPC  P3
```

The two routines submitted by Mike Nicholls. The "9600 baud print" routine is at top right, with the longer "fetch character" routine to the left.

check on the number of bits to go before the word finishes. This tally must be kept, but a memory reference instruction such as a DLD or an ILD, which one would normally use, takes 22 microcycles and hence makes the timing excessive.

"A way out, and the method used here, is to store the count in the extension register. The data is stored in the accumulator, so that either can be operated upon by accessing via XAE instructions.

"Further savings were gained by setting the start bit first, then setting up the data and count in this time. One point to note is that the status register is affected by this routine due to the CAS instruction in the loop. However, it may be saved if desired by using store and load routines either side of the print program. The 9600 baud serial data exits from the FO pin of the SC/MP.

"The program may reside anywhere in memory and may be called from anywhere since it is self-contained. It is intended to be used with 7-bit ASCII code, which is the normal format, and the program prints the data which is in the accumulator when it is called.

"I have also enclosed a second program which may be used in conjunction with the high speed printing routine if it suits the application. The program will run through a data field in memory, printing each character via the 9600 baud print routine until it reaches the end of the data field.

"The data field is assumed to start in the memory at address XXYY. It assumes the 9600 baud print routine is located in memory also, with starting address AABB. The program continues printing until it encounters a zero data byte, which is assumed to terminate the data field. Control is then returned to Kitbug, at location 0000.

"A delay of five seconds is placed at the start of the program, to allow switching the terminal from 110 to 9600 baud before the program begins outputting characters. When control is returned to Kitbug at the end of printing, the terminal must again be switched back to 110 baud.

"A better way would be to have logic to allow software control of terminal baud rate, via special control characters.

"The changes made to the EME-1 module in the video terminal are minimal. All that needs to be done is to change the wiring to the baud rate switch S3 so that it switches between 110 and 9600 baud, in place of the original 110-300 baud function. This simply involves changing the switch connection from pin C of LK13 to pin H."

Well, there you have them. Three useful little items of software for Mini Scamp, by courtesy of C. B. Curnow and Mike Nicholls. Our thanks to them, and we hope other Mini Scamp users find the routines of interest.

Reaction time program for Mini Scamp

Here is another simple program for Mini Scamp, from its designer Dr. John Kennewell. Designed to show the use of interrupt programming, it measures human reaction time in hundredths of a second. The additional hardware required is minimal: a pushbutton switch and a resistor!

by DR JOHN KENNEWELL
Physics Department, Newcastle University

The program to be described here can be used to measure your reaction time and display it in one-hundredths of a second on the LED's of Mini Scamp. Only minimal external hardware is required, and the program makes use of the interrupt facilities provided on the SC/MP CPU chip.

The concept of interrupt programming is a very important one, and is employed in many situations where slow peripherals (e.g., a keyboard) are communicating with the computer. In this instance, it would be a waste of time if the CPU was almost permanently in the hold mode waiting for a character to be input from the keyboard. On the other hand, if the CPU went away and did something instead of waiting on the keyboard, it might miss the character, which could be input at any time. The resolution of this problem lies in the use of interrupt programming. In this mode of operation, the CPU does what it will most of the time, but as soon as a signal appears at its interrupt input, it suspends what it is doing and jumps to a specified program routine to "service" the device that created the interrupt (in this case, to accept the next character from the keyboard).

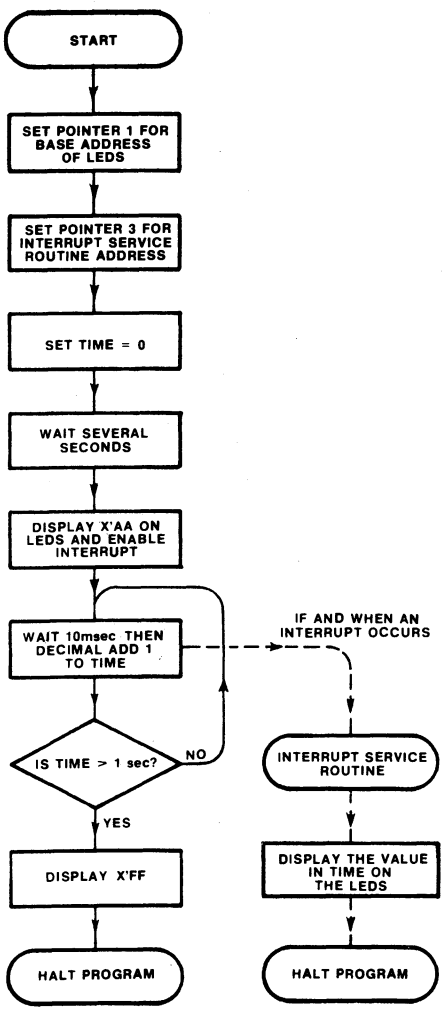


FIG. 1

In the SC/MP, the sense A input also serves as the interrupt input. However, the interrupt mechanism is only enabled after an IEN instruction has been executed in the program. It may also be disabled, with a DINT instruction.

If, at any time after an IEN instruction has been encountered, the sense A input goes high, the CPU will complete execution of the instruction it is presently working on, and then will jump to whatever address is contained in pointer register 3. It is thus essential that register 3 has been loaded with the starting address of the interrupt service routine, prior to giving the IEN command. To avoid the problems which would be caused if a second interrupt occurred while the first was still being dealt with, the interrupt facility is automatically disabled after the interrupt, and must be re-enabled, if desired, by the programmer through another IEN command.

To enable a better understanding of the operation of the program to measure reaction time, a flow chart is shown in Fig. 1. Notice that the main body of the program and the interrupt service routine are two separate programs, with no logical connection (represented by a continuous line) between them. The dotted line represents the jump that occurs between the two if and when an interrupt occurs.

The flowchart is fairly self explanatory. As the LEDs are required for display purposes, their address must be first set up in pointer register 1. The address of the interrupt service routine is also loaded into pointer register 3 at this stage.

The variable TIME is used for counting the reaction time, and is set to zero each time the program is run. A delay of several seconds is then incorporated into the program. After this time the LEDs are set to display the binary word "10101010". This particular number was chosen because it is dissimilar to any other display produced by the program. Thus, whatever the LED's were displaying before this time, they will change at this moment. This provides a visual cue to the subject whose reaction time is being determined. The moment he or she sees the LEDs change, the pushbutton of Fig. 2 should be depressed, causing a high logic level to appear at the sense A input, and thus creating an interrupt.

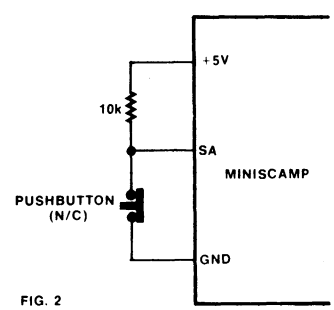


FIG. 2

However, immediately the LEDs change the program starts to execute a

```
          *REACTION TIME
0000 08             NOP
0001 C400           LDI   0
0003 37             XPAH  3
0004 C42F           LDI   X'2F          ;DISPLAY ADDRESS
0006 33             XPAL  3
0007 C400           LDI   0
0009 31             XPAL  1
000A C408           LDI   8
000C 35             XPAH  1
000D C400           LDI   0
000F C829           ST    TIME
0011 C40F           LDI   15
0013 C824           ST    DELAY
0015 8FFF   DLP     DLY   255
0017 B820           DLD   DELAY
0019 9CFA           JNZ   DLP           ;DELAY 4 SECS
001B 05             IEN                 ;ENABLE INTERRUPT
001C C4AA           LDI   X'AA
001E C902           ST    2(1)          ;FLASH 10101010
0020 C400   TLOOP   LDI   0
0022 8F08           DLY   8             ;DELAY 10 MS
0024 C014           LD    TIME
0026 EC01           DAI   1
0028 C810           ST    TIME
002A 9CF4           JNZ   TLOOP
002C C4FF           LDI   X'FF          ;REACTION TIME
002E C902           ST    2(1)          ;> 1 SEC
0030 90FE   HALT    JMP   HALT          ;STOP
0032 C006           LD    TIME          ;DISPLAY SUBROUTINE
0034 C902           ST    2(1)          ;DISP REACT TIME
0036 90FE   HALTA   JMP   HALTA         ;STOP
0038   00   DELAY   .BYTE 0
0039   00   TIME    .BYTE 0
003A                                          END
```

Fig. 3: the reaction time program listing for Mini Scamp.

loop that adds one to the variable TIME for each 10 millisecond interval that elapses until the interrupt actually occurs. Upon receiving the interrupt, the CPU jumps to the service routine whereupon it then displays the current value of TIME on the LEDs. Because decimal addition was employed to increment this variable, the readout on the LEDs contains two BCD numbers. These numbers represent the reaction time in one-hundredths of a second.

One condition we have not yet discussed is what happens if the interrupt does not occur. Well, as you may have realised from the flowchart, each time around the 10 ms counting loop there is a test made to determine if TIME contains a value indicating that more than 1000 ms (1 second) has elapsed. If it does, the program is terminated and all the LEDs are turned on to indicate this condition. Those of you who find this happening frequently when running the program would be well advised to have a physical checkup or drink less!

The actual program listing is given in Fig. 3. I have incorporated a delay of about four seconds between the time of setting the RUN/HALT switch to RUN and that when the LED's change. This may be lengthened (by increasing the number in address X'12) if you find that you are able to anticipate this change too readily. In fact, those with a little ingenuity might like to incorporate the random number generator routine of August 1977 at this point, to provide a random starting delay of between say 3 and 15 seconds.

The delay of 10 milliseconds is fairly critical if you wish to achieve accurate results with this program. Unfortunately, this value depends upon your exact clock frequency, and this will in turn depend on the exact value of the 470 pF capacitor (or crystal) used in the SC/MP clock circuit. Fortunately, even without access to an accurate frequency meter, it is possible to measure this frequency. This can be done by running an earlier program "Binary Count and Display", and determining (using a watch) how many seconds it takes to run right through a complete cycle; that is, to count from 0 through 255 and back to 0 again. The clock frequency (in megahertz) of your computer is then equal to 67.272 divided by this time. More importantly, the time of one microcycle (in microseconds) is equal to the above time (in seconds) divided by 33.636.

The number of microcycles required (R) in the 10 ms delay instruction is now given by 10,000/(microcycle time (*us*)) —72. As the actual delay in microcycles is 13 + 2 x (accumulator) + 514 x (displacement), the displacement (address X'23 in the program) is given by D = (R — 13)/514, forgetting the remainder. The accumulator value (address X'21) may be found by (R — 13 — 514 x D)/2.

The above procedure may seem rather complicated, but is worth trying to follow if you wish to make accurate measurements of reaction time.

The final point of note about the program is the method employed to "stop" the program. Although the SC/MP CPU does have a HALT instruction, it requires additional circuitry, external to the chip to implement it. Otherwise it will be regarded as an NOP instruction. The way I have chosen to implement a pseudo HALT it to simply put the machine into a tight loop (HALT JMP HALT) wherein the JMP instruction simply jumps back to itself. For most purposes the machine thus appears to be effectively halted.

After loading the program into your computer, it may be run in the usual way (i.e., CPU, RESET, RUN). The normally closed pushbutton (Fig. 2) should then be pressed as soon as the LEDs change. Your reaction time will then be displayed on the LEDs. For example, if the LEDs display 00101001 this would indicate a reaction time of 0.29 seconds. Note that the first four LEDs (from the left) are a BCD representation of the first number after the decimal point. The rightmost four LEDs give the second decimal place. A healthy young adult should have a consistent reaction time with this apparatus of between about 0.2 and 0.3 seconds. To run the program again simply go through the sequence HALT, RESET, and RUN.

Using Mini Scamp to find intermittents

Having built our Mini Scamp microcomputer and become familiar with its operation, this contributor decided to use it to help find intermittent faults in electronic equipment. In this article he describes the simple hardware interfacing used, and also gives the program he developed.

by JOHN BARRY*

I have developed a simple way of using the Mini Scamp microcomputer to help track down intermittent faults in electronic circuits. Mini Scamp is used to monitor the voltages at two points in the circuit, and give an indication of any change. It also records the values before and after the change.

I initially developed this idea while trying to repair an early model Playmaster Twin 25 amplifier, which I had built for a friend and which would fail every few days at odd times. By leaving the Mini Scamp unit connected to two points in the amplifier overnight I was able to trace the problem to one of the driver transistors (BC639) which was running fairly hot. I have subsequently used the unit to trace several other intermittent faults (I work as an Electronics Technician) and have found it saves a lot of time as it can be left unattended until the fault shows up.

As you can see from the block diagram, the external interfacing circuitry turns Mini Scamp into a simple analog-to-digital converter system with input multiplexing. The analog-to-digital conversion is performed by comparing the selected input voltage with the output of an integrator fed with a fixed input voltage, and initially reset before conversion begins. While the integrator output ramp is lower than the unknown input, Mini Scamp is used to develop a count. Then when the ramp equals the input voltage, the LM307 comparator signals this to Mini Scamp by taking the SC/MP sense-A input high. The count is then stopped, with a value which is proportional to the input voltage.

The count rate is selected so that a 2-volt input range corresponds to 200 counts. However the input signal is offset by +1 volt, so that signals of either input polarity can be handled. Thus counts from 0 to 100 correspond to inputs from —1 to 0V, while those from 100 to 200 correspond to inputs from 0 to +1V.

After a count is performed, 100 counts are subtracted from the result and if the resultant number is positive then this is taken to be a positive voltage and this number is treated as the reading (thus positive inputs result in bit 8 being zero). If after subtracting 100 counts the result is negative (bit 8 is 1) then it is a negative result (in 2's complement form). This result is converted by subtracting 1 count and then complementing the count giving a result which corresponds to the displacement below zero. The sign bit (bit 8) is set to indicate a negative result.

The three flag outputs from the SC/MP processor chip are used to zero the integrator before a count and to select the input to be read. I used reed relays to perform this switching as they were on hand, but it would possibly be easier to use CMOS switches. Also the number of input channels could be increased by decoding the flag inputs, but I found two inputs quite useful for fault finding.

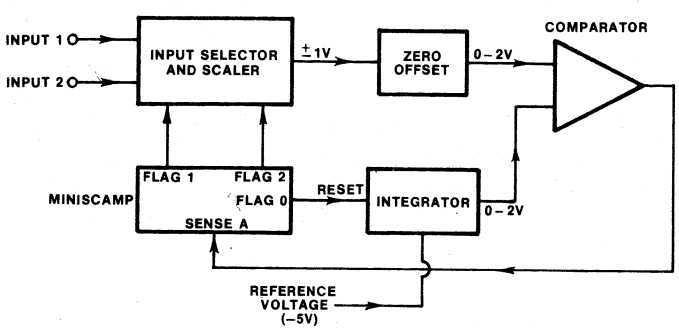
Ordinary variable resistors were used to scale the inputs to suitable values and these were calibrated using a multimeter when the unit was set up. I used +30 volt scales, as these are fairly common maximum values with solid-state equipment.

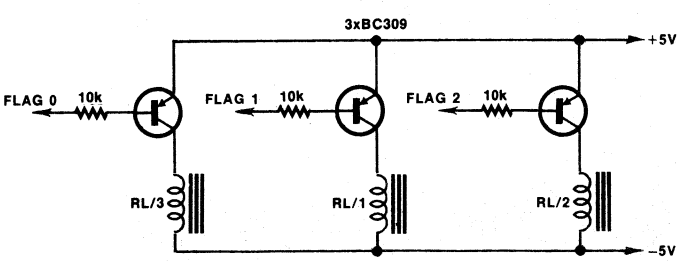
When the program is run the unit reads channel 1 and displays it on the LEDs and then halts. If this result is unsuitable the reset button allows a repeat reading. If the reading is suitable the deposit button is pressed and the unit repeats the process with channel 2. These two readings are stored for later comparison in the stack.

When the deposit button is pressed after the channel 2 reading the unit then reads both channels again and "exclusive ORs" the results with the original readings.

I discarded the three least significant digits (bits) from these results, to allow slight variations due to supply voltage changes etc. The amount of variation detected may be altered by varying the bits discarded.

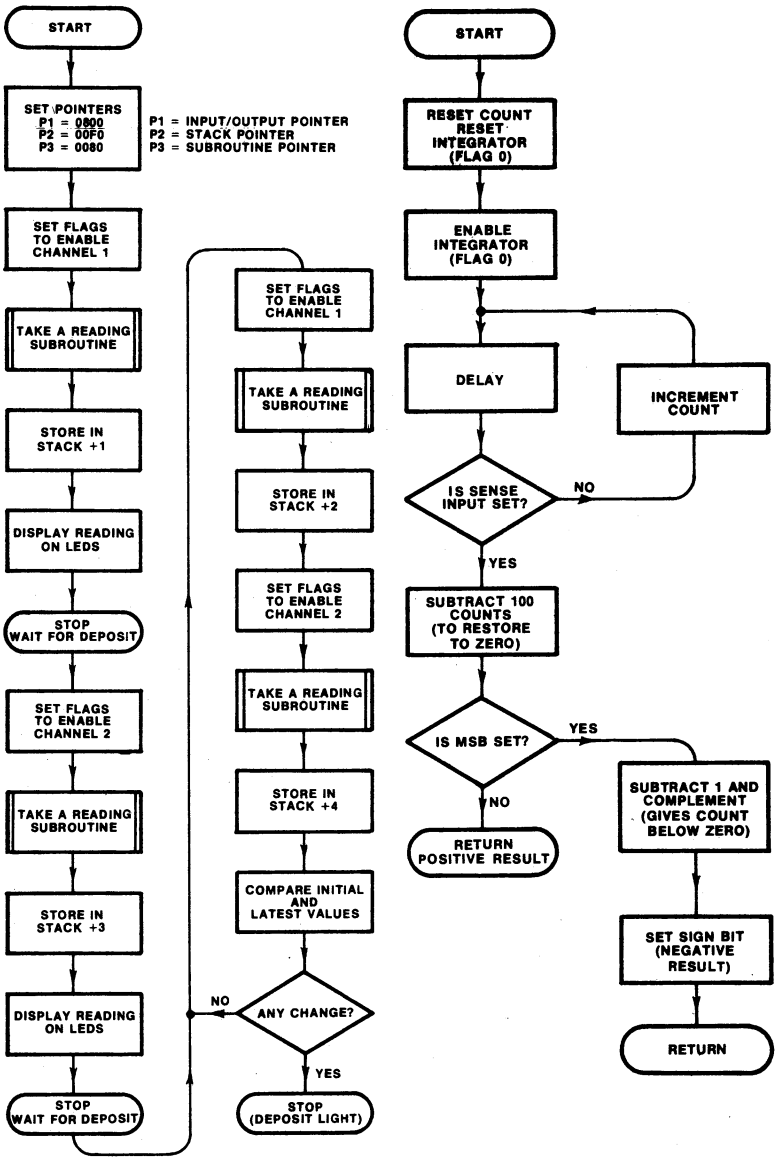
If the readings are the same, the process is repeated continously. When a variation is detected, the program stops and the deposit light comes on as an indicator. The readings may then be examined by directly examining the





Shown above is the block diagram of the author's intermittent detection system using Mini Scamp. At right are the control relay driver cricuits.

*82 Liverpool Street, Liverpool NSW.




```
0000 08       NOP
0001 C4 08    LDI
0003 35       XPAH P1
0004 C4 00    LDI
0006 31       XPAL P1
0007 C4 00    LDI
0009 36       XPAH P2
000A C4 F0    LDI
000C 32       XPAL P2
000D C4 00    LDI
000F 37       XPAH P3
0010 C4 80    LDI
0012 33       XPAL P3
0013 06       CSA
0014 D4 FD    ANI
0016 DC 04    ORI
0018 07       CAS
0019 3F       XPPC P3
001A CA 01    ST P2+1
001C C9 02    ST P1+2
001E C1 01    LD P1+1
0020 06       CSA
0021 D4 F8    ANI
0023 DC 02    ORI
0025 07       CAS
0026 C4 80    LDI
0028 33       XPAL P3
0029 3F       XPPC P3
002A CA 03    ST P2+3
002C C9 02    ST P1+2
002E C1 01    LD P1+1
0030 06    $1 CSA
0031 D4 FD    ANI
0033 DC 04    ORI
0035 07       CAS
0036 C4 80    LDI
0038 33       XPAL P3
0039 3F       XPPC P3
003A CA 02    ST P2+2
003C 06       CSA
003D D4 F8    ANI
003F DC 02    ORI
0041 07       CAS
0042 C4 80    LDI
0044 33       XPAL P3
0045 3F       XPPC P3
0046 CA 04    ST P2+4
0048 C2 02    LD P2+2
004A E2 01    XOR 02+1
004C D4 F8    ANI
004E 9C 08    JNZ
0050 C2 04    LD P2+4
0052 E2 03    XOR P2+3
0054 D4 F8    ANI
0056 98 D8    JZ
0058 C1 01 $4 LD P1+1

(SUBROUTINE TO TAKE A READING)

0080 08       NOP
0081 C4 00    LDI
0083 CA 00    ST P2
0085 06       CSA
0086 D4 FE    ANI
0088 07       CAS
0089 8F FF    DLY
008B 06       CSA
008C DC 01    ORI
008E 07       CAS
008F C4 0C $1 LDI
00C1 8F 00    DLY
00C3 06       CSA
00C4 D4 10    ANI
00C6 9C 04    JNZ
00C8 AA 00    ILD P2
00CA 90 F3    JMP
00CC 06    $2 CSA
00CD DC 80    ORI
00CF 07       CAS
00D0 C2 00    LD P2
00D2 FC 64    CAI
00D4 CA 00    ST P2
00D6 D4 80    ANI
00D8 9C 03    JNZ
00DA C2 00    LD P2
00DC 3F       XPPC P3
00DD C2 00 $3 LD P2
00DF F4 FF    ADD
00E1 CA 00    ST P2
00E3 06       CSA
00E4 D4 7F    ANI
00E6 07       CAS
00E7 C4 00    LDI
00E9 FA 00    CAD
00EB DC 80    OR
00ED 3F       XPPC P3
```

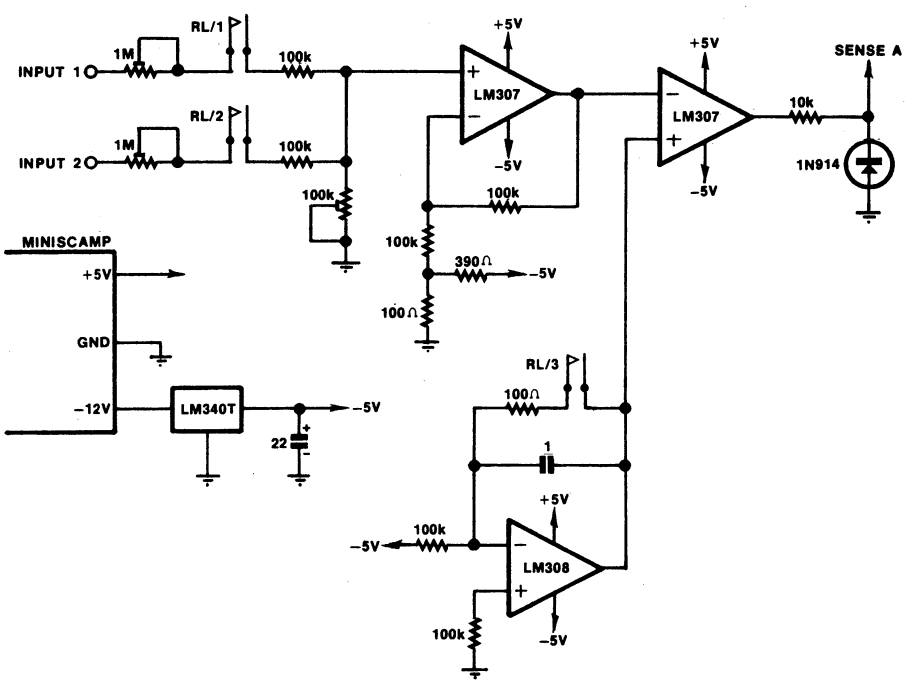
The flow charts for the author's program are shown at upper left, with the "take a reading" subroutine shown separated from the rest of the program for clarity. The full listing is shown above, with the subroutine again separated for clarity.

contents of the appropriate memory locations:
00F1 for the initial channel 1 reading
00F2 for the final channel 1 reading
00F3 for the initial channel 2 reading
00F4 for the final channel 2 reading

The output readings are of course in binary and must be converted to decimal and suitably scaled. I am at present working on a decimal readout using seven segment displays as I have also found the unit can be used simply as a digital voltmeter by inserting a loop in the program after the initial reading.

The interfacing circuitry was built on a piece of Vero-board and plugs into the back of the Mini Scamp via a 25 way cannon connector. A —5 volt rail was obtained from the —12V available in Mini Scamp using a LM320T fixed regulator.

Editor's Note: It may be desirable to add a bell, buzzer or other audible alarm to Mini Scamp for this application, so that it can attract one's attention more effectively. This could be done via a relay and driver connected to the collector of the DRQ control transistor.



The circuit for the simple analog-to-digital converter system added to Mini Scamp, and controlled by it via the relays RL1, 2 and 3.